Building Secure and Reliable Network Applications

Kenneth P. Birman

Department of Computer Science

Cornell University

Ithaca, New York 14853

Cover image: line drawing of the golden gate bridge looking towards San Francisco?

Trademarks Cited in the Text

Preface and Acknowledgements

This book is dedicated to my family, for their support and tolerance over the two-year period that it was written. The author is grateful to so many individuals, for their technical assistance with aspects of the development, that to try and list them one by one would certainly be to omit someone whose role was vital. Instead, let me just thank my colleagues at Cornell, Isis Distributed Systems, and worldwide for their help in this undertaking. I am also greatful to Paul Jones of Isis Distributed Systems and to Francois Barrault and Yves Eychenne of Stratus France and Isis Distributed Systems, France, for providing me with resources needed to work on this book during a sabbatical that I spent in Paris, in fall of 1995 and spring of 1996. Cindy Williams and Werner Vogels provided invaluable help in overcoming some of the details of working at such a distance from home.

A number of reviewers provided feedback on early copies of this text, leading to (one hopes) considerable improvement in the presentation. Thanks are due to: Marjan Bace, David Bakken, Robert Cooper, Yves Eychenne, Dalia Malki, Raghu Hudli, David Page, David Plainfosse, Henrijk Paszt, John Warne and Werner Vogels. Raj Alur, Ian Service and Mark Wood provided help in clarifying some thorny technical questions, and are also gratefully acknowledged. Bruce Donald's emails on idiosyncracies of the Web were extremely useful and had a surprisingly large impact on treatment of that topic in this text.

Much of the work reported here was made possible by grants from the U.S. Department of Defense through its Advanced Research Projects Agency, DARPA (administered by the Office of Naval Research, Rome Laboratories, and NASA), and by infrastructure grants from the National Science Foundation. Grants from a number of corporations have also supported this work, including IBM Corporation, Isis Distributed Systems Inc., Siemens Corporate Research (Munich and New Jersey), and GTE Corporation. I wish to express my thanks to all of these agencies and corporations for their generosity.

The techniques, approaches, and opinions expressed here are my own, and may not represent positions of the organizations and corporations that have supported this research.

Introduction

Despite nearly twenty years of progress towards ubiquitous computer connectivity, distributed computing systems have only recently emerged to play a serious role in industry and society. Perhaps this explains why so few distributed systems are reliable in the sense of tolerating failures automatically, guaranteeing properties such as performance or response time, or offering security against intentional threats. In many ways the engineering discipline of reliable distributed computing is still in its infancy.

One might be tempted to reason tautologically, concluding that reliability must not be all that important in distributed systems (since otherwise, the pressure to make such systems reliable would long since have become overwhelming). Yet, it seems more likely that we have only recently begun to see the sorts of distributed computing systems in which reliability is critical. To the extent that existing mission- and even life-critical applications rely upon distributed software, the importance of reliability has perhaps been viewed as a narrow, domain-specific issue. On the other hand, as distributed software is placed into more and more critical applications, where safety or financial stability of large organizations depends upon the reliable operation of complex distributed applications, the inevitable result will be growing demand for technology developers to demonstrate the reliability of their distributed architectures and solutions. It is time to tackle distributed systems reliability in a serious way. To fail to do so today is to invite catastrophic computer-systems failures tomorrow.

At the time of this writing, the sudden emergence of the "World Wide Web" (variously called the "Web", the Information Superhighway, the Global Information Infrastructure, the Internet, or just the Net) is bringing this issue to the forefront. In many respects, the story of reliability in distributed systems is today tied to the future of the Web and the technology base that has been used to develop it. It is unlikely that any reader of this text is unfamiliar with the Web technology base, which has penetrated the computing industry in record time. A basic premise of our study is that the Web will be a driver for distributed computing, by creating a mass market around distributed computing. However, the term "Web" is often used loosely: much of the public sees the Web as a single entity that encompasses all the Internet technologies that exist today and that may be introduced in the future. Thus when we talk about the Web, we are inevitably faced with a much broader family of communications technologies.

It is clear that some form of critical mass has recently been reached: distributed computing is emerging from its specialized and very limited niche to become a mass-market commodity, something that literally everyone depends upon, like a telephone or an automobile. The Web paradigm brings together the key attributes of this new market in a single package: easily understandable graphical displays, substantial content, unlimited information to draw upon, virtual worlds in which to wander and work. But the Web is also stimulating growth in other types of distributed applications. In some intangible way, the experience of the Web has caused modern society to suddenly notice the potential of distributed computing.

Consider the implications of a societal transition whereby distributed computing has suddenly become a mass market commodity. In the past, a mass-market item was something everyone "owned". With the Web, one suddenly sees a type of commodity that everyone "does". For the most part, the computers and networks were already in place. What has changed is the way that people see them and use them. The paradigm of the Web is to connect useful things (and many useless things) to the network. Communication and connectivity suddenly seem to be mandatory: no company can possibily risk arriving late for the Information Revolution. Increasingly, it makes sense to believe that if an application *can* be put on the network, someone is thinking about doing so, and soon.

Whereas reliability and indeed distributed computing were slow to emerge prior to the introduction of the Web, reliable distributed computing will be necessary if networked solutions are to be used safely for many of the applications that are envisioned. In the past, researchers in the field wondered why the uptake of distributed computing had been so slow. Overnight, the question has become one of understanding how the types of computing systems that run on the Internet and the Web, or that will be accessed through it, can be made reliable enough for emerging critical uses.

If Web-like interfaces present medical status information and records to a doctor in a hospital, or are used to control a power plant from a remote console, or to guide the decision making of major corporations, reliability of those interfaces and applications will be absolutely critical to the users. Some may have life-or-death implications: if that physician bases a split-second decision on invalid data, the patient might die. Others may be critical to the efficient function of the organization that uses them: if a bank mismanages risk because of an inaccurate picture of how its investments are allocated, the bank could incur huge losses or even fail. In still other settings, reliability may emerge as a key determinant in the marketplace: the more reliable product, at a comparable price, may simply displace the less reliable one. Reliable distributed computing suddenly has broad relevance.

•

Throughout what follows, the term "distributed computing" is used to describe a type of computer system that differs from what could be called a "network computing" system. The distinction illuminates the basic issues with which we will be concerned.

As we use the term here, a *computer network* is a communication technology supporting the exchange of messages among computer programs executing on computational nodes. Computer networks are *data movers,* providing capabilities for sending data from one location to another, dealing with mobility and with changing topology, and automating the division of available bandwidth among contending users. Computer networks have evolved over a twenty year period, and during the mid 1990's network connectivity between computer systems became pervasive. Network bandwidth has also increased enormously, rising from hundreds of bytes per second in the early 1980's to millions per second in the mid 1990's, with gigabit rates anticipated in the late 1990's and beyond.

Network functionality evolved steadily during this period. Early use of networks was entirely for file transfer, remote login and electronic mail or news. Over time, however, the expectations of users and the tools available have changed. The network user in 1996 is likely to be familiar with interactive network browsing tools such as Netscape's browsing tool, which permits the user to wander within a huge and interconnected network of multimedia information and documents. Tools such as these permit the user to conceive of a computer workstation as a window into an immense world of information, accessible using a great variety of search tools, easy to display and print, and linked to other relevant material that may be physically stored halfway around the world and yet accessible at the click of a mouse.

Meanwhile, new types of networking hardware have emerged. The first generation of networks was built using point-to-point connections; to present the illusion of full connectivity to users, the network included a software layer for routing and connection management. Over time, these initial technologies were largely replaced by high speed long distance lines that route through various hubs, coupled to local area networks implemented using multiple access technologies such as Ethernet and FDDI: hardware in which a single "wire" has a large number of computers attached to it, supporting the abstraction of a shared message bus. At the time of this writing, a third generation of technologies is reaching the market, such as ATM hardware capable of supporting gigabit communication rates over virtual circuits, mobile connection technologies for the office that will allow computers to be moved without rewiring, and more ambitious mobile computing devices that exploit the nationwide cellular telephone grid for communications support.

As recently as the early 1990's, computer bandwidth over wide-area links was limited for most users. The average workstation had high speed access to a local network, and perhaps the local email system was connected to the Internet, but individual users (especially those working from PC's) rarely had better than 1600 baud connections available for personal use of the Internet. This picture is changing rapidly today: more and more users have relatively high speed modem connections to an Internet service provider that offers megabyte-per-second connectivity to remote servers. With the emergence of ISDN services to the home, the last link of the chain will suddenly catch up with the rest. Individual connectivity has thus jumped from 1600 baud to perhaps 28,800 baud at the time of this writing, and may jump to 1 Mbaud or more in the not distant future. Moreover, this bandwidth has finally reached the PC community, which enormously outnumbers the workstation community.

It has been suggested that technology revolutions are often spurred by discontinuous, as opposed to evolutionary, improvement in a key aspect of a technology. The bandwidth improvements we are now experiencing are so disproportionate with respect to other performance changes (memory sizes, processor speeds) as to fall squarely into the discontinuous end of the spectrum. The sudden connectivity available to PC users is similarly disproportionate to anything in prior experience. The Web is perhaps just the first of a new generation of communications-oriented technologies enabled by these sudden developments.

In particular, the key enablers for the Web were precisely the availability of adequate long-distance communications bandwidth to sustain its programming model, coupled to the evolution of computing systems supporting high performance graphical displays and sophisticated local applications dedicated to the user. It is only recently that these pieces fell into place. Indeed, the Web emerged more or less as early as it could possibly have done so, considering the state of the art in the various technologies on which it depends. Thus while the Web is clearly a breakthrough — the "killer application" of the Internet — it is also the most visible manifestation of a variety of underlying developments that are also enabling other kinds of distributed applications. It makes sense to see the Web as the tip of an iceberg: a paradigm for something much broader that is sweeping the entire computing community.

•

As the trend towards better communication performance and lower latencies continues, it is certain to fuel continued growth in distributed computing. In contrast to a computer network, a *distributed computing system* refers to computing systems and applications that cooperate to coordinate actions at multiple locations in a network. Rather than adopting a perspective in which conventional (non-distributed) application programs access data remotely over a network, a distributed system includes multiple application programs that communicate over the network, but take actions at the multiple places where the application runs. Despite the widespread availability of networking since early 1980, distributed computing has only become common in the 1990's. This lag reflects a fundamental issue: distributed computing turns out to be much harder than non-distributed or network computing applications, especially if reliability is a critical requirement.

Our treatment explores the technology of distributed computing with a particular bias: to understand why the emerging generation of critical Internet and Web technologies is likely to require very high levels of reliability, and to explore the implications of this for distributed computing technologies. A key issue is to gain some insight into the factors that make it so hard to develop distributed computing systems that can be relied upon in critical settings, and and to understand can be done to simplify the task. In other disciplines like civil engineering or electrical engineering, a substantial body of practical development rules exists that the designer of a complex system can draw upon to simplify his task. It is rarely necessary for the firm that builds a bridge to engage in theoretical analyses of stress or basic properties of the materials used, because the theory in these areas was long-ago reduced to collections of practical rules and formulae that the practitioner can treat as tools for solving practical problems.

This observation motivated the choice of the cover of the book. The Golden Gate Bridge is a marvel of civil engineering that reflects a very sophisticated understanding of the science of bridge-building. Although located in a seismically active area, the bridge is believed capable of withstanding even an extremely severe earthquake. It is routinely exposed to violent winter storms: it may sway but is never seriously threatened. And yet the bridge is also esthetically pleasing: one of the truely beautiful constructions of its era. Watching the sun set over the bridge from Berkeley, where I attended graduate school, remains among the most memorable experiences of my life. The bridge illustrates that beauty can also be resilient: a fortunate development, since otherwise, the failure of the Tacoma Narrows bridge might have ushered in a generation of bulky and overengineered bridges. The achievement of the Golden Gate bridge illustrates that even when engineers are confronted with extremely demanding standards, it is possible to achieve solutions that are elegant and lovely at the same time as they are resilient. This is only possible, however, to the degree that there exists an engineering science of robust bridge building.

We can build distributed computing systems that are reliable in this sense, too. Such systems would be secure, trustworthy, and would guarantee availability and consistency even when limited numbers of failures occur. Hopefully, these limits can be selected to provide adequate reliability without excessive cost. In this manner, just as the science of bridge-building has yielded elegant and robust bridges, reliability need not compromise elegance and performance in distributed computing.

One could argue that in distributed computing, we are today building the software bridges of the Information Superhighway. Yet in contrast to the disciplined engineering that enabled the Golden Gate Bridge, as one explores the underlying technology of the Internet and the Web one discovers a disturbing and pervasive inattention to issues of reliability. It is common to read that the Internet (developed originally by the Defense Department's Advanced Research Projects Agency, ARPA) was built to withstand a nuclear war. Today, we need to adopt a similar mindset as we extend these networks into systems that must support tens or hundreds of millions of Web users, and a growing number of hackers whose objectives vary from the annoying to the criminal. We will see that many of the fundamental technologies of the Internet and Web fundamental assumptions that, although completely reasonable in the early days of the Internet's development, have now started to limit scalability and reliability, and that the infrastructure is consequently exhibiting troubling signs of stress.

One of the major challenges, of course, is that use of the Internet has begun to expand so rapidly that the researchers most actively involved in extending its protocols and enhancing its capabilities are forced to work incrementally: only limited changes to the technology base can be contemplated, and even small upgrades can have very complex implications. Moreover, upgrading the technologies used in the Internet is somewhat like changing the engines on an airplane while it is flying. Jointly, these issues limit the ability of the Internet community to move to a more reliable, secure, and scalable architecture. They create a background against which the goals of this textbook will not easily be achieved.

In early 1995, the author was invited by ARPA to participate in an unclassified study of the survability of distributed systems. Participants included academic experts and invited experts familiar with the state of the art in such areas as telecommunications, power systems management, and banking.

This study was undertaken against a backdrop colored by the recent difficulties of the Federal Aviation Agency, which launched a project in the late 1980's and early 1990's to develop a new generation of highly reliable distributed air traffic control software. Late in 1994, after losing a huge sum of money and essentially eliminating all distributed aspects of an architecture that was originally innovative precisely for its distributed reliability features, a prototype of the proposed new system was finally delivered, but with such limited functionality that planning on yet another new generation of software had to begin immediately. Meanwhile, article after article in the national press reported on failures of air-traffic control systems, many stemming from software problems, and several exposing airplanes and passengers to extremely dangerous conditions. Such an situation can only inspire the utmost concern in regard to the practical state of the art.

Although our study did not focus on the FAA's specific experience, the areas we did study are in many ways equally critical. What we learned is that situation encountered by the FAA's highly visible project is occuring, to a greater or lesser degree, within all of these domains. The pattern is one in which pressure to innovate and introduce new forms of products leads to the increasingly ambitious use of distributed computing systems. These new systems rapidly become critical to the enterprise that developed them: too many interlocked decisions must be made to permit such steps to be reversed. Responding to the pressures of timetables and the need to demonstrate new functionality, engineers inevitably postpone considerations of availability, security, consistency, system management, fault-tolerance — what we call "reliability" in this text — until "late in the game," only to find that it is then very hard to retrofit the necessary technologies into what has become an enormously complex system. Yet, when pressed on these issues, many engineers respond that they are merely following common practice: that their systems use the "best generally accepted engineering practice" and are neither more nor less robust than the other technologies used in the same settings.

Our group was very knowledgeable about the state of the art in research on reliability. So, we often asked our experts whether the development teams in their area are aware of one result or another in the field. What we learned was that research on reliability has often stopped too early to impact the intended consumers of the technologies we developed. It is common for work on reliability to stop after a paper or two and perhaps a splashy demonstration of how a technology can work. But such a proof of concept often leaves open the question of how the reliability technology can interoperate with the software development tools and environments that have become common in industry. This represents a serious obstacle to the ultimate use of the technique, because commercial software developers necessarily work with commercial development products and seek to conform to industry standards.

This creates a quandry: one cannot expect a researcher to build a better version of a modern operating system or communications architecture: such tasks are enormous and even very large companies have difficulty successfully concluding them. So it is hardly surprising that research results are demonstrated on a small scale. Thus, if industry is not eager to exploit the best ideas in an area like reliability, there is no organization capable of accomplishing the necessary technology transition.

For example, we will look at an object-oriented technology called the Common Object Request Broker Architecture, or CORBA, which has become extremely popular. CORBA is a structural methodology: a set of rules for designing and building distributed systems so that they will be explicitly described, easily managed, and so that components can be interconnected as easily as possible. One would expect that researchers on security, fault-tolerance, consistency, and other properties would embrace such architectures, because they are highly regular and designed to be extensible: adding a reliability property to a CORBA application should be a very natural step. However, relatively few researchers have looked at the specific issues that arise in adapting their results to a CORBA setting (we'll hear about some of the ones that have). Meanwhile, the CORBA community has placed early emphasis on performance and interoperability, while reliability issues have been dealt with primarily by individual vendors (although, again, we'll hear about some products that represent exceptions to the rule). What is troubling is the sense of "disconnection" between the reliability community and its most likely users, and the implication that reliability is not accorded a very high value by the vendors of distributed systems products today.

Our study contributed towards a decision by the DoD to expand its investment in research on technologies for building practical, survivable, distributed systems. This DoD effort will focus both on developing new technologies for implementing survivable systems, and on developing new approaches to hardening systems built using conventional distributed programming methodologies, and it could make a big difference. But one can also use the perspective gained through a study such as this one to look back over the existing state of the art, asking to what degree the technologies we already have "in hand" can, in fact, be applied to the critical computing systems that are already being developed.

As it happened, I started work on this book during the period when this DoD study was underway, and the presentation that follows is strongly colored by the perspective that emerged from it. Indeed, the study has considerably impacted my own research project. I've come to the personal conclusion is that the situation could be much better if developers were simply to begin to think hard about reliability, and had greater familiarity with the techniques at their disposal today. There may not be any magic formulas that will effortlessly confer reliability upon a distributed system, but at the same time, the technologies available to us are in many cases very powerful, and are frequently much more relevant to even off the shelf solutions than is generally recognized. We need more research on the issue, but we also need to try harder to incorporate what we already know how to do into the software development tools and environments on which the majority of distributed computing applications are now based. This said, it is also clear that researchers will need to start paying more attention to the issues that arise in moving their ideas from the laboratory to the field.

Lest these comments seem to suggest that the solution is in hand, it must be understood that there are intangible obstacles to reliability that seem very subtle and yet rather pervasive. Above, it was commented that the Internet and Web is in some ways "fundamentally" unreliable, and that industry routinely treats reliability as a secondary consideration, to be addressed only in mature products and primarily in a "fire fighting" mode, for example after a popular technology is somehow compromised by hackers in a visible way. Neither of these will be easy problems to fix, and they combine to have far-reaching implications. Major standards have repeatedly defered consideration of reliability issues and security until "future releases" of the standards documents or prototype platforms. The message sent to developers is clear: should they wish to build a reliable distributed system, they will need to overcome tremendous obstacles, both internal to their companies and in the search for enabling technologies, and will find relatively little support from the vendors who sell standard computing platforms.

The picture is not uniformly grim, of course. The company I founded in 1988, Isis Distributed Systems, is one of a handful of small technology sources that do offer reliability solutions, often capable of being introduced very transparently into existing applications. (Isis now operates as a division of Stratus Computers Inc., and my own role is limited to occassional consulting). Isis is quite successful, as are many of these companies, and it would be wrong to say that there is no interest in reliability. But these isolated successes are in fact the small story. The big story is that reliability has yet to make much of a dent on the distributed computing market.

•

The approach of this book is to treat distributed computing technology in a uniform way, looking at the technologies used in developing Internet and Web applications, at emerging standards such as

CORBA, and at the technologies available to us for building reliable solutions within these settings. Many texts that set this goal would do so primarily through a treatment of the underlying theory, but our approach here is much more pragmatic. By and large, we treat the theory as a source of background information that one should be aware of, but not as the major objective. Our focus, rather, is to understand how and why practical software tools for reliable distributed programming work, and to understand how they can be brought to bear on the broad area of technology currently identified with the Internet and the Web. By building up models of how distributed systems execute and using these to prove properties of distributed communication protocols, we will show how computing systems of this sort can be formalized and reasoned about, but the treatment is consistently driven by the *practical* implications of our results.

One of the most serious concerns about building reliable distributed systems stems from more basic issues that would underly any form of software reliability. Through decades of experience, it has become clear that software reliability is a *process*, not a *property*. One can talk about design practices that reduce errors, protocols that reconfigure systems to exclude faulty components, testing and quality assurance methods that lead to increased confidence in the correctness of software, and basic design techniques that tend to limit the impact of failures and prevent them from propagating. All of these improve the reliability of a software system, and so presumably would also increase the reliability of a distributed software system. Unfortunately, however, no degree of process ever leads to more than empirical confidence in the reliability of a software system. Thus, even in the case of a non-distributed system, it is hard to say "system X guarantees reliability property Y" in a rigorous way. This same limitation extends to distributed settings, but is made even worse by the lack of a process comparable to the one used in conventional systems. Significant advances are needed in the process of developing reliable distributed computing systems, in the metrics by which we characterize reliability, the models we use to predict their behavior in "new" configurations reflecting changing loads or failures, and in the formal methods used to establish that a system satisfies its reliability goals.

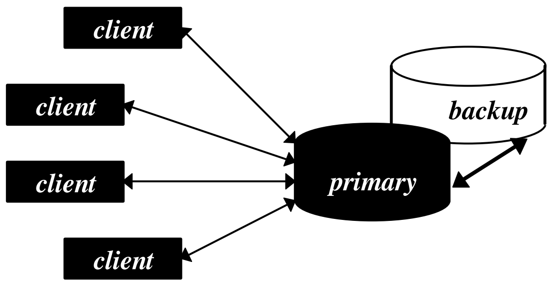


Figure I-1: An idealized client-server system with a backup server for increased availability. The clients interact with the primary server; in an air-traffic application, the server might provide information on the status of air-traffic sectors, and the clients may be air traffic controllers responsible for routing decisions. The primary keeps the backup up to date so that if a failure occurs, the clients can switch to the backup and resume operation with minimal disruption.

For certain types of applications, this creates a profound quandary. Consider the design of an air traffic control software system, which (among other services) provides air traffic controllers with information about the status of air traffic sectors (Figure I-1). Web sophisticates may want to think of this system as one that provides a web-like interface to a database of routing information maintained on a server. Thus, the controller would be presented with a depiction of the air traffic situation, with push-button style interfaces or other case-specific interfaces providing access to additional information about flights, projected tragectories, possible options for rerouting a flight, and so forth. To the air traffic controller these are the commands supported by the system; the web user might think of them as active hyperlinks. Indeed, even if air traffic control systems are not typical of what the Web is likely to support, other equally critical applications are already moving to the Web, using very much the same "programming model."

A controller who depends upon a system such as this needs an absolute assurance that if the service reports that a sector is available and a plane can be routed into it, this information is correct and that no other controller has been given the same information in regard to routing some other plane. An optimization criteria for such a service would be that it minimize the frequency with which it reports a sector as being occupied when it is actually free. A fault-tolerance goal would be that the service remain operational despite limited numbers of failures of component programs, and perhaps that it perform self-checking operations so as to take a component off-line if it somehow falls out of synchronization with regard to the states of other components. Such goals would avoid scenarios such as the one illustrated in Figure I-2, where the system state has become dangerously inconsistent as a result of a network failure that fools some clients into thinking the primary has failed, and similarly fools the primary and backup into mutually believing one-another to have crashed.

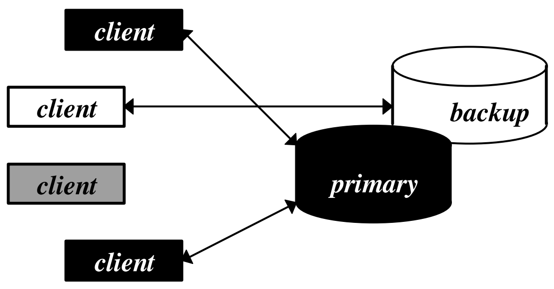


Figure I-2: A scenario that will arise in Chapter 4, when we consider the use of a standard remote procedure call methodology to build a client-server architecture for a critical setting. In the case illustrated, some of the client programs have become disconnected from the primary server, perhaps because of a transient network failure (one that corrects itself after a brief period during which message loss rates are very high). In the resulting system configuration, the primary and backup servers each consider themselves to be "in charge" of the system as a whole. There are two clients still connected to the primary (black), one to the backup (white), and one is completely disconnected (gray). Such a configuration exposes the application user to serious threats. In an air-traffic control situation, it is easy to imagine that accidents could arise if such a situation arose and was permitted to persist. The goal of this textbook is dual: to assist the reader in understanding why such situations are a genuine threat in modern computing systems, and to study the technical options for building better systems that can prevent such situations from arising. The techniques presented will sometimes have limitations, which we will also work to quantify, and to understand any reliability implications. While many modern distributed systems have overlooked reliability issues, our working hypothesis will be that this situation is changing rapidly, and that the developer of a distributed system has no choice but to confront these issues and begin to use technologies that respond to them.

Now, suppose that the techniques of this book were used to construct such a service, using the best available technological solutions, combined with rigorous formal specifications of the software components involved, and the best possible quality process. Theoretical results assure us that inconsistencies such as the one in Figure I-2 cannot arise. Years of testing might yield a very high degree of confidence in the system, yet the service remains a large, complex software artifact. Even minor changes to the system, to add a feature, correct a very simple bug, or to upgrade the operating system version or hardware, could introduce serious problems long after the system was put into production. The question then becomes: can complex software systems ever be used in critical settings? If so, are distributed systems somehow "worse", or are the issues similar?

At the core of the material treated in this book is the consideration seen in this question. There may not be a single answer: distributed systems are suitable for some critical applications and ill-suited for others. In effect, although one can build "reliable distributed software," reliability has its limits and there are problems that distributed software should probably not be used to solve. Even given an appropriate technology, it is easy to build inappropriate solutions – and, conversely, even with an inadequate technology, one can sometimes build critical services that are still useful in limited ways. The air traffic example, described above, might or might not fall into the feasible category, depending on the detailed specification of the system, the techniques used to implement the solution, and the overall process by which the result is used and maintained.

Through the material in this book, the developer will be guided to appropriate design decisions, appropriate development methodologies, and to an understanding of the reliability limits on the solutions that result from this process. No book can expect to instill the sense of responsibility that the reader may need to draw upon in order to make such decisions wisely, but one hopes that computer systems engineers, like bridge builders and designers of aircraft, are highly motivated to build the best and most reliable systems possible. Given such a motivation, an appropriate development methodology, and appropriate software tools, extremely reliable distributed software can be implemented and deployed even into critical settings. We will see precisely how this can be done in the chapters that follow.

•

Perhaps this book can serve a second purpose in accomplishing its primary one. Many highly placed industry leaders have commented to me that until reliability is forced upon them, their companies will *never* take the issues involved seriously. The investment needed is simply viewed as very large, and likely to slow the frantic rate of progress on which computing as an industry has come to depend. I believe that the tide is now turning in a way that will, in fact, force change, and that this text can contribute to what will, over time, become an overwhelming priority for the industry.

Reliability is viewed as complex and costly, much as the phrase "robust bridge" conjures up a vision of a massive, expensive, and ugly artifact. Yet, the Golden Gate Bridge is robust and is anything but massive or ugly. To overcome this instinctive reaction, it will be necessary for the industry to come to understand reliability as being compatible with performance, elegance, and market success. At the same time, it will be important for pressure favoring reliability to grow, through demand by the consumers for more reliable products. Jointly, such trends would create an incentive for reliable distributed software engineering, while removing a disincentive.

As the general level of demonstrated knowledge concerning how to make systems reliable rises, the expectation of society and government that vendors will employ such technologies is, in fact, likely to rise. It will become harder and harder for corporations to cut corners by bringing an unreliable product to market and yet advertising it as "fault-tolerant", "secure", or otherwise "reliable". Today, these terms are often used in advertising for products that are not reliable in any meaningful sense at all. One might similarly claim that a building or a bridge was constructed "above code" in a setting where the building code is completely ad-hoc. The situation changes considerably when the building code is made more explicit and demanding, and bridges and buildings that satisify the standard have actually been built successfully (and, perhaps, elegantly and without excessive added cost). In the first instance, a company can easily cut corners; in the second, the risks of doing so are greatly increased.

Moreover, at the time of this writing, vendors often seek to avoid software product liability using complex contracts that stipulate the unsuitability of their products for critical uses, the near certainty that their products will fail even if used correctly, and in which it is stressed that the customer accepts full responsibility for the eventual use of the technology. It seems likely that as such contracts are put to the test, many of them will be recognized as analogous to those used by a landlord who rents an dangerously deteriorated apartment to a tenant, using a contract that warns of the possibility that the kitchen floor could collapse without warning and that the building is a firetrap lacking adequate escape routes. A landlord could certainly draft such a contract and a tenant might well sign it. But if the landlord fails to maintain the building according to the general standards for a safe and secure dwelling, the courts would still find the landlord liable if the floor indeed collapses. One cannot easily escape the generally accepted standards for one's domain of commercial activity.

By way of analogy, we may see growing pressure on vendors to recognize their fundamental responsibilities to provide a technology base adequate to the actual uses of their technologies, like it or not. Meanwhile, today a company that takes steps to provide reliability worries that in so doing, it may have raised expectations impossibly high and hence *exposed* itself to litigation if its products fail. As reliability becomes more and more common, such a company will be protected by having used the best available engineering practices to build the most reliable product that it was capable of producing. If such a technology does fail, one at least knows that it was not the consequence of some outrageous form of negligence. Viewed in these terms, many of the products on the market today are seriously deficient. Rather than believing it safer to confront a reliability issue using the best practices available, many companies feel that they run a lower risk by ignoring the issue and drafting evasive contracts that hold themselves harmless in the event of accidents.

The challenge of reliability, in distributed computing, is perhaps the unavoidable challenge of the coming decade, just as performance was the challenge of the past one. By accepting this challenge, we also gain new opportunities, new commercial markets, and help create a future in which technology is used responsibly for the broad benefit of society. There will inevitably be real limits on the reliability of the distributed systems we can build, and consequently there will be types of distributed computing systems that should not be built because we cannot expect to make them adequately reliable. However, we are far from those limits, and are in many circumstances deploying technologies known to be fragile in ways that actively encourage their use in critical settings. Ignoring this issue, as occurs too often today, is irresponsible and dangerous, and increasingly unacceptable. Reliability challenges us as a community: it falls upon us now to respond.

A User's Guide to This Book

This book was written with several types of readers in mind, and consequently weaves together material that may be of greater interest to one type of reader with that aimed at another type of reader.

Practioners will find that the book has been constructed to be readable more or less sequentially from start to finish. The first part of the book may well be familiar material to many practitioners, but we try to approach this a perspective of understanding reliability and consistency issues that arise even when using the standard distributed systems technologies. We also look at the important roles of performance and modularity in building distributed software that can be relied upon. The second part of the book, which focuses on the Web, is of a similar character. Even if experts in this area may be surprised by some of the subtle reliability and consistency issues associated with the Web, and may find the suggested solutions useful in their work.

The third part of the book looks squarely at reliability technologies. Here, a pragmatically-oriented reader may want to skim through Chapters 13 through 16, which get into the details of some fairly complex protocols and programming models. This material is included for thoroughness, and I don't think it is exceptionally hard to understand. However, the developer of a reliable system doesn't necessarily need to know every detail of how the underlying protocols work, or how they are positioned relative to some of the theoretical arguments of the decade! The remainder of the book can be read without having worked through these chapters in any great detail. Chapters 17 and 18 look at the uses of these "tools" through an approach based on what are called wrappers, however, and chapters 19-24 look at some related issues concerning such topics as real-time systems, security, persistent data, and system management. The content is practical and the material is intended to be of a hands-on nature. Thus, the text is designed to be read more or less in order by this type of systems developer, with the exception of those parts of Chapters 13 through 16 where the going gets a bit heavy.

Where possible, the text includes general background material: there is a section on ATM networks, for example, that could be read independently of the remainder of the text, one on Corba, one on message-oriented middleware, and so forth. As much as practical, I have tried to make these sections free-standing and to index them properly, so that if one were worried about security exposures of the NFS file system, for example, it would be easy to read about that specific topic without reading the entire book as well. Hopefully, practitioners will find this text useful as a general reference for the technologies covered, and not purely for its recommendations in the area of security and reliability.

Next, some comments directed towards other researchers and instructors who may read or chose to teach from this text. I based the original outline of this treatment on a course that I have taught several times at Cornell, to a mixture of 4'th year undergraduates, professional Master's degree students, and 1'st year Ph.D. students. To facilitate the development of course materials, I have placed my slides (created using the Microsoft PowerPoint utility) on Cornell University's public file server, where they can be retrieved using FTP. (Copy the files from ftp.cs.cornell.edu/pub/ken/slides). The text also includes a set of problems that can be viewed either as thought-provoking exercizes for the professional who wishes to test his or her own understanding of the material, or as the basis for possible homework and course projects in a classroom setting.

Any course based on this text should adopt the same practical perspective as the text itself. I suspect that some of my research colleagues will consider the treatment broad but somewhat superficial;

this reflects a decision by the author to focus primarily on "systems" issues, rather than on theory or exhaustive detail on any particular topic. In making this decision, compromises had to be accepted: when teaching from this text, it may be necessary to also ask the students to read some of the more technically complete papers that are cited in subsections of interest to the instructor, and to look in greater detail at some of the systems that are are mentioned only briefly here. On the positive side, however, there are few, if any, introductory distributed systems textbooks that try to provide a genuinely broad perspective on issues in reliability. In the author's experience, many students are interested in this kind of material today, and having gained a general exposure, would then be motivated to attend a much more theoretical course focused on fundamental issues in distributed systems theory. Thus, while this textbook may not be sufficient in and of itself for launching a research effort in distributed computing, it could well serve as a foundation for such an activity.

It should also be noted that, in my own experience, the book long for a typical 12-week semester. Instructors who elect to teach from it should be selective about the material that will be covered, particularly if they intend to treat chapters 13-17 in any detail. If one has the option of teaching over two semesters, it might make sense to split the course into two parts and to include supplemental material on the Web. I suspect that such a sequence would be very popular given the current interest in network technology. At Cornell, for example, I tend to split this material into a more practical course that I teach in the fall, aiming at our professional master's degree students, followed by a more probing advanced graduate course that I or one of my colleagues teach in the spring, drawing primarily on the original research papers associated with the topics we cover. This works well for us at Cornell, and the organization and focus of the book match with such a sequence.

A final comment regarding references. To avoid encumbering the discussion with a high density of references, the book cites relevant work the first time a reference to it arises in the text, or where the discussion needs to point to a specific paper, but may not do so in subsequent references to the same work. References are also collected at the end of each chapter into a short section on related readings. It is hard to do adequate justice to such a large and dynamic area of research with any limited number of citations, but every effort has been made to be fair and complete.

Part I: Basic Distributed Computing Technologies

Although our treatment is motivated by the emergence of the Global Information Superhighway and the World Wide Web, this first part of the book focuses on the general technologies on which any distributed computing system relies. We review basic communication options, and the basic software tools that have emerged for exploiting them and for simplifying the development of distributed applications. In the interests of generality, we cover more than just the specific technologies embodied in the Web as it exists at the time of this writing, and in fact terminology and concepts specific to the Web are not introduced until Part II of the book. However, even in this first part, we do discuss some of the most basic issues that arise in building reliable distributed systems, and we begin to establish the context within which reliability can be treated in a systematic manner.

1. Fundamentals

1.1 Introduction

Reduced to the simplest terms, a *distributed computing system* is a set of computer programs, executing on one or more computers, and coordinating actions by exchanging *messages*. A *computer network* is a collection of computers interconnected by hardware that directly supports message passing. Most distributed computing systems operate over computer networks, although this is not always the case: one can build a distributed computing system in which the components execute on a single multi-tasking computer, and one can also build distributed computing systems in which information flows between the components by means other than message passing. Moreover, as we will see in Chapter 24, there are new kinds of parallel computers, called "clustered" servers, that have many attributes of distributed systems despite appearing to the user as a single machine built using rack-mounted components.

We will use the term *protocol* in reference to an algorithm governing the exchange of messages, by which a collection of processes coordinate their actions and communicate information among themselves. Much as a *program* is the set of instructions, and a *process* denotes the execution of those instructions, a protocol is a set of instructions governing the communication in a distributed program, and a distributed computing system is the result of executing some collection of such protocols to coordinate the actions of a collection of processes in a network.

This textbook is concerned with *reliability* in distributed computing systems. Reliability is a very broad term that can have many meanings, including:

- *Fault-tolerance:* The ability of a distributed computing system to recover from component failures without performing incorrect actions.
- *High availability:* In the context of a fault-tolerant distributed computing system, the ability of the system to restore correct operation, permitting it to resume providing services during periods when some components have failed. A highly available system may provided reduced service for short periods of time while reconfiguring itself.
- *Continuous availability*. A highly available system with a very "small" recovery time, capable of providing uninterrupted service to its users. The reliability properties of a continuously available system are unaffected or only minimally affected by failures.
- *Recoverability:* Also in the context of a fault-tolerant distributed computing system, the ability of failed components to restart themselves and rejoin the system, after the cause of failure has been repaired.
- *Consistency:* The ability of the system to coordinate related actions by multiple components, often in the presence of concurrency and failures. Consistency underlies the ability of a distributed system to emulate a non-distributed system.
- *Security:* The ability of the system to protect data, services and resources against misuse by unauthorized users.
- *Privacy*. The ability of the system to protect the identity and locations of its users from unauthorized disclosure.
- *Correct specification:* The assurance that the system solves the intended problem.
- *Correct implementation:* The assurance that the system correctly implements its specification.

- *Predictable performance:* The guarantee that a distributed system achieves desired levels of performance, for example data throughput from source to destination, latencies measured for critical paths, requests processed per second, etc.
- *Timeliness:* In systems subject to "real-time" constraints, the assurance that actions are taken within the specified time bounds, or are performed with a desired degree of temporal synchronization between the components.

Underlying many of these issues are questions of tolerating failures. Failure, too, can have many meanings:

- *Halting failures:* In this model, a process or computer either works correctly, or simply stops executing and crashes without taking incorrect actions, as a result of failure. As the model is normally specified, there is no way to detect that the process has halted except by timeout: it stops sending "keep alive" messages or responding to "pinging" messages and hence other processes can deduce that it has failed.
- *Fail-stop failures:* These are *accurately detectable* halting failures. In this model, processes fail by halting. However, other processes that may be interacting with the faulty process also have a completely accurate way to detect such failures. For example, a fail-stop environment might be one in which timeouts can be used to monitor the status of processes, and *no timeout occurs unless the process being monitored has actually crashed.* Obviously, such a model may be unrealistically optimistic, representing an idealized world in which the handling of failures is reduced to a pure problem of how the system should react when a failure is sensed. If we solve problems with this model, we then need to ask how to relate the solutions to the real world.
- *Send-omission failures:* These are failures to send a message that, according to the logic of the distributed computing systems, should have been sent. Send-omission failures are commonly caused by a lack of buffering space in the operating system or network interface, which can cause a message to be discarded after the application program has sent it but before it leaves the sender's machine. Perhaps surprisingly, few operating systems report such events to the application.
- *Receive-omission failures:* These are similar to send-omission failures, but occur when a message is lost near the destination process, often because of a lack of memory in which to buffer it or because evidence of data corruption has been discovered.
- *Network failures:* These occur when the network loses messages sent between certain pairs of processes.
- *Network partitioning failures:* These are a more severe form of network failure, in which the network fragments into disconnected subnetworks, within which messages can be transmitted, but between which messages are lost. When a failure of this sort is repaired, one talks about *merging* the network partitions. Network partitioning failures are a common problem in modern distributed systems, hence we will have a lot to say about them later in Part III of this text.
- *Timing failures:* These occur when a temporal property of the system is violated, for example when a clock on a computer exhibits a value that is unacceptably far from the values of other clocks, or when an action is taken too soon or too late, or when a message is delayed by longer than the maximum tolerable delay for a network connection.
- *Byzantine failures:* This is a term that captures a wide variety of "other" faulty behaviors, including data corruption, programs that fail to follow the correct protocol, and even malicious or adversarial behaviors by programs that actively seek to force a system to violate its reliability properties.

An even more basic issue underlies all of these: the meaning of computation, and the model one assumes for communication and coordination in a distributed system. Some examples of models include these:

- *Real-world networks:* These are composed of workstations, personal computers, and other sort of computing devices interconnected by hardware. Properties of the hardware and software components will often be known to the designer, such as speed, delay, and error frequencies for communication devices, latencies for critical software and scheduling paths, throughput for data generated by the system and data distribution patterns, speed of the computer hardware, accuracy of clocks, etc. This information can be of tremendous value in designing solutions to problems that might be very hard – or impossible – in a completely general sense.

  A specific issue that will emerge as being particularly important when we consider guarantees of behavior in Part III of the text concerns the availability, or lack, of accurate temporal information. Until the late 1980's. the clocks built into workstations were notoriously inaccurate, exhibiting high drift rates that had to be overcome with software protocols for clock resynchronization. There are limits on the quality of synchronization possible in software, and this created a substantial body of research and lead to a number of competing solutions. In the early 1990's, however, the advent of satellite time sources as part of the global positioning system (GPS) changed the picture: for the price of an inexpensive radio receiver, any computer could obtain accurate temporal data, with resolution in the sub-millisecond range. The degree to which GPS recievers actually replace quartz-based time sources remains to be seen, however. Thus, real-world systems are notable (or notorious) in part for having temporal information, but of potentially low quality.

- *Asynchronous computing systems:* This is a very simple theoretical model used to approximate one extreme sort of computer network. In this model, no assumptions can be made about the relative speed of the communication system, processors and processes in the network. One message from a process *p* to a process *q* may be delivered in zero time, while the next is delayed by a million years. The asynchronous model reflects an assumption about time, but not failures: given an asynchronous model, one can talk about protocols that tolerate message loss, protocols that overcome fail-stop failures in asynchronous networks, etc. The main reason for using the model is to prove properties about protocols for which one makes as few assumptions as possible. The model is very clean and simple, and it lets us focus on fundamental properties of systems without cluttering up the analysis by including a great number of practical considerations. If a problem can be solved in this model, it can be solved at least as well in a more realistic one. On the other hand, the converse may not be true: we may be able to do things in realistic systems by making use of features not available in the asynchronous model, and in this way may be able to solve problems in real systems that are "impossible" in ones that use the asynchronous model.
- *Synchronous computing systems:* Like the asynchronous systems, these represent an extreme end of the spectrum. In the synchronous systems, there is a very strong notion of time that all processes in the system share. One common formulation of the model can be thought of as having a system-wide gong that sounds periodically; when the processes in the system hear the gong, they run one "round" of a protocol, reading messages from one another, sending messages that will be delivered in the next round, and so forth. And these messages *always* are delivered to the application by the start of the next round, or not at all.

  Normally, the synchronous model also assumes bounds on communication latency between processes, clock skew and precision, and other properties of the environment. As in the case of an asynchronous model, the synchronous one takes an extreme point of view because this simplifies reasoning about certain types of protocols. Real-world systems are not synchronous – it is impossible to build a system in which actions are perfectly coordinated as this model assumes. However, if one proves the impossibility of solving some problem in the synchronous model, or proves that some problem requires at least a certain number of messages in this model, one has established a sort of lower-bound. In a real-world system, things can only get worse, because we are limited to "weaker" assumptions. This makes the synchronous model a valuable tool for understanding how hard it will be to solve certain problems.

- *Parallel shared memory systems:* An important family of system are based on multiple processors that share memory. Communication is by reading and writing shared memory locations. Clearly, the shared memory model can be emulated using message passing, and can be used to implement message communication. Nonetheless, because there are important examples of real computers that implement this model, there is considerable theoretical interest in the model *per-se*. Unfortunately, although this model is very rich and a great deal is known about it, it would be beyond the scope of this textbook to attempt to treat the model in any detail.

1.2 Components of a Reliable Distributed Computing System

Reliable distributed computing systems are assembled from basic building blocks. In the simplest terms, these are just processes and messages, and if our interest was purely theoretical, it might be reasonable to stop at that. On the other hand, if we wish to apply theoretical results in practical systems, we will need to work from a fairly detailed "real" understanding of how practical systems actually work. In some ways, this is unfortunate, because real systems often include mechanisms that are deficient in ways that seem simple to fix, or inconsistent with one another, but have such a long history (or are so deeply embedded into standards) that there may be no way to "improve" on the behavior in question. Yet, if we want to actually build reliable distributed systems, it is unrealistic to insist that we will only do so in idealized environments that support some form of theoretically motivated structure. The real world is heavily committed to standards, and the task of translating our theoretical insights into practical tools that can interplay with these standards is probably the most important challenge faced by the computer systems engineer.

It is common to think of a distributed system as operating over a layered set of network services. Each layer corresponds to a software abstraction or hardware feature, and maybe implemented in the application program itself, in a library of procedures to which the program is linked, in the operating system, or even in the hardware of the communications device. As an illustration, here is the layering of the ISO *Open Systems Interconnection (OSI)* protocol model [Tan88,Com91,CS91,CS93,CDK94]:

- *Application:* This is the application program itself, up to the points at which it performs communication operations.
- *Presentation:* This is the software associated with placing data into messages in a format that can be interpreted by the destination process(es) to which the message will be sent, and for extracting data from messages in the destination process.
- *Session:* This is the software associated with maintaining connections between pairs or sets of processes. A session may have reliability properties and may require some form of initialization or setup, depending on the specific setting with which the user is working. In the OSI model, any reliability properties are implemented by the session software, and lower layers of the hierarchy are permitted to be unreliable, e.g. by losing messages.
- *Transport:* The transport layer is responsible for breaking large messages into smaller packets that respect size limits imposed by the network communication hardware. On the incoming side, the transport layer reassembles these packets into messages, discarding packets that are identified as duplicates, or messages for which some constituent packets were lost in transmission.
- *Network:* This is the layer of software concerned with routing and low-level flow control on networks composed of multiple physical segments interconnected by what are called "bridges" and "gateways."
- *Data link:* The data link layer is normally part of the hardware that implements a communication device. This layer is responsible for sending and receiving packets, recognizing packets destined for the local machine and copying them in, discarding corrupted packets, and other "interface level" aspects of communication.

- *Physical:* The physical layer is concerned with representation of packets on the "wire", e.g. the hardware technology for transmitting individual bits and the protocol for gaining access to the wire if it is shared by multiple computers.

Application	The program using the communication connection
Presentation	Software to encode application data into messages, and to decode on reception.
Session	The logic associated with guaranteeing end-to-end properties such as reliability.
Transport	Software concerned with fragmenting big messages into small packets
Network	Routing functionality, usually limited to small or fixed-size packets
Data-link	The protocol used to represent packets on the wire

Table 1: ISO Protocol Layers

It is useful to distinguish the types of guarantees provided by the various layers as being *end-to-end* guarantees in the case of the session, presentation and application layer and *point-to-point* guarantees for layers below these. The distinction is important in complex networks where a message may need to traverse many links to reach its destination. In such settings, a point-to-point property is one that holds only on a per-hop basis: for example, the data-link protocol is concerned with a single hop taken by the message, but not with its overall route or the guarantees that the application may expect from the communication link itself. The session, presentation and application layers, in contrast, impose a more complex logical abstraction on the underlying network, with properties that hold between the end-points of a communication link that may physically extend over a complex substructure. In Part III of this textbook we will concern ourselves with increasingly elaborate end-to-end properties, until we finally extend these properties into an completely encompassing distributed communication abstraction that embraces the distributed system as a whole and provides consistent behavior and guarantees throughout. And, just as the ISO layering builds its end-to-end abstractions over point-to-point ones, so will we need to build these more sophisticated abstractions over what are ultimately point-to-point properties.

As seen in Figure 1-1, each layer is logically composed of transmission logic and the corresponding reception logic. In practice, this often corresponds closely to the implementation of the architecture: for example, most session protocols operate by imposing a multiple session abstraction over a shared (or "multiplexed")link-level connection. The packets generated by the various higher level session protocols can be conceived of as merging into a single stream of packets that are treated by the IP link level as a single "customer" for its services. Nonetheless, one should not necessarily assume that the implementation of a layered protocol architecture involves some sort of separate module for each layer. To maximize performance, the functionality of a layered architecture is often compressed into a single piece of software, and in some cases layers may be completely bypassed for types of messages where the layer would take no action – for example, if a message is very small, the OSI transport layer wouldn't need to fragment it into multiple packets, and one could imagine an implementation of the OSI stack specialized for small messages, that omits the transport layer. Indeed, the pros and cons of layered protocol architecture have become a major topic of debate in recent years [CT87, AP93, KP93, KC94, BD95].

Although the OSI layering is probably the best known, the notion of layering communication software is pervasive, and there are many other examples of layered architectures and layered software systems. Later in this textbook we will see ways in which the OSI layering is outdated, because it doesn't directly address multi-participant communication sessions and doesn't match very well with some new types of communication hardware, such as asynchronous transfer-mode (ATM) switching systems. In discussing this point we will see that more appropriate layered architectures can be constructed, although they don't match the OSI layering very closely. Thus, one can think of layering as a methodology, or layering as a very specific thing, matched to the particular layers of the OSI hierarchy. The former perspective is a popular one that is only gaining importance with the introduction of object-oriented distributed computing environments, which have a natural form of layering associated with object classes and subclasses. The later form of layering has probably become hopelessly incompatible with standard practice by the time of this writing, although many companies and governments continue to "require" that products comply with it.

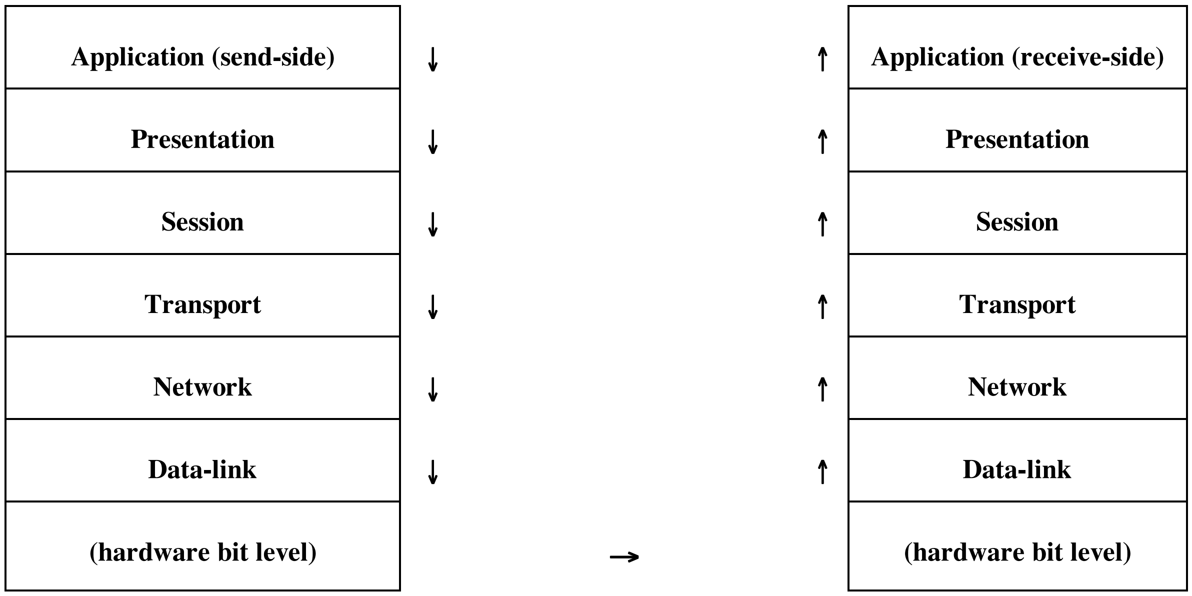


Figure 1-1: Data flow in an ISO protocol stack. Each sending layer is invoked by the layer above it and passes data off to the layer below it, and conversely on the receive side. In a logical sense, however, each layer interacts with its peer on the remote side of the connection. For example, the sender-side session layer may add a header to a message that the receive-side session layer strips off.

Stepping back somewhat, it can be argued that a layered communication architecture is primarily valuable as a *descriptive abstraction* – a model that captures the essential functionality of a real communication system but doesn't need to accurately reflect its implementation. The idea of abstracting the behavior of a distributed system in order to concisely describe it or to reason about it is a very important one. However, if the abstraction doesn't accurately correspond to the implementation, this also creates a number of problems for the system designer, who now has the obligation to develop a specification and correctness proof for the abstraction, to implement, verify and test the corresponding software, and to undertake an additional analysis that confirms that the abstraction accurately models the implementation.

It is easy to see how this process can break down; for example, it is nearly inevitable that changes to the implementation will have to be made long after a system has been deployed. If the development process is genuinely this complex, it is likely that the analysis of overall correctness will not be repeated for every such change. Thus, from the perspective of a user, abstractions can be a two-edged sword. They offer appealing and often simplified ways to deal with a complex system, but they can also be simplistic or even incorrect. And this bears strongly on the overall theme of reliability. To some degree, the very process of cleaning up a component of a system in order to describe it concisely can compromise the reliability of a more complex system in which that component is used.

Throughout the remainder of this book, we will often have recourse to models and abstractions, in much more complex situations than the OSI layering. This will assist us in reasoning about and comparing protocols, and in proving properties of complex distributed systems. At the same time, however, we need to keep in mind that this whole approach demands a sort of “meta approach”, namely a higher level of abstraction at which we can question the methodology itself, asking if the techniques by which we create reliable systems are themselves a possible source of unreliability. When this proves to be the case, we need to take the next step as well, asking what sorts of systematic remedies can be used to fight these types of reliability problems.

Can “well structured” distributed computing systems be built that can tolerate the failures of their own components? In layerings like the OSI one, this issue is not really addressed, which is one of the reasons that the OSI layering won’t work well for our purposes in this text. However, the question is among the most important ones that will need to be resolved if we want to claim that we have arrived at a workable methodology for engineering reliable distributed computing systems. A methodology, then, must address descriptive and structural issues as well as practical ones such as the protocols used to overcome a specific type of failure or to coordinate a specific type of interaction.

1.2.1 Communications Technology

The most basic communications technology in any distributed system is the hardware support for message passing. Although there are some types of networks that offer special properties, most modern networks are designed to transmit data in *packets* with some fixed but small maximum size. Each packet consists of a *header*, which is a data structure containing information about the packet, its destination and route, etc. It contains a *body*, which is the bytes that make up the content of the packet. And it may contain a *trailer*, which is a second data structure that is physically transmitted after the header and body, and would normally consist of a checksum for the packet that the hardware computes and appends to it as part of the process of transmitting the packet.

When a user’s message is transmitted over a network, the packets actually sent ‘‘on the wire’’ include headers and trailers, and may have a fixed maximum size. Large messages are sent as multiple packets. For example, Figure 1-2 illustrates a message that has been fragmented into three packets, each containing a header and some part of the data from the original message. Not all fragmentation schemes include trailers, and in the figure no trailer is shown.

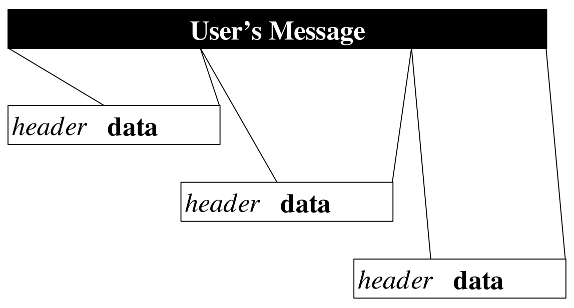


Figure 1-2: Large messages are fragmented for transmission

Modern communication hardware often permits large numbers of computers to share a single communication “fabric”. For this reason, it is necessary to specify the address to which a message should be transmitted. The hardware used for communication therefore will normally support some form of *addressing capability*, by which the destination of a message can be identified. More important to most software developers, however, are addresses supported by the transport services available on most operating systems. These *logical addresses* are a representation of location within the network, and are used to route packets to their destinations. Each time a packet makes a "hop" over a communications link, the sending computer is expected to copy the hardware address of the next machine in the path into the outgoing packet. Within this textbook, we assume that each computer has a logical address, but will have little to say about hardware addresses.

On the other hand, there are two hardware addressing features that have important implications for higher level communication software. These are the ability of the hardware to *broadcast* and *multicast* messages

A broadcast is a way of sending a message so that it will be delivered to all computers that it reaches. This may not be all the computers in a network, because of the various factors that can cause a receive omission failure to occur, but for many purposes, absolute reliability is not required. To send a hardware broadcast, an application program generally places a special logical address in an outgoing message that the operating system maps to the appropriate hardware address. The message will only reach those machines connected to the hardware communications device on which the transmission occurs, so the use of this feature requires some knowledge of network communications topology.

A multicast is a form of broadcast that communicates to a subset of the computers that are attached to a communications network. To use a multicast, one normally starts by creating a new "multicast group address" and installing it into the hardware interfaces associated with a communications device. Multicast messages are then sent much as a broadcast would be, but are only accepted, at the hardware level, at those interfaces that have been instructed to install the group address to which the message is destined. Many network routing devices and protocols watch for multicast packets and will forward them automatically, but this is rarely attempted for broadcast packets.

Chapter 2 discusses some of the most common forms of communication hardware in detail.

1.2.2 Basic transport and network services

The layer of software that runs over the communications layer is the one most distributed systems programmers deal with. This layer hides the properties of the communications hardware from the programmer. It provides the ability to send and receive messages that may be much larger than the ones supported by the underlying hardware (although there is normally still a limit, so that the amount of operating system buffering space needed for transport can be estimated and controlled). Th transport layer also implements logical addressing capabilities by which every computer in a complex network can be assigned a unique address, and can send and receive messages from every other computer.

Although many transport layers have been proposed, one set of standards has been adopted by almost all vendors. This standard defines the so-called "Internet Protocol" or IP protocol suite, and originated in a research network called the ARPANET that was developed by the U.S. government in the late 1970's [Tan88,Com91,CDK94]. A competing standard was introduced by the ISO organization in association with the OSI layering cited earlier, but has not gained the sort of ubiquitous acceptance of the IP protocol suite, and there are additional proprietary standards that are widely used by individual vendors or industry groups, but rarely seen outside their community. For example, most PC networks support a protocol called NETBIOS, but this protocol is not common in any other type of computing environment.

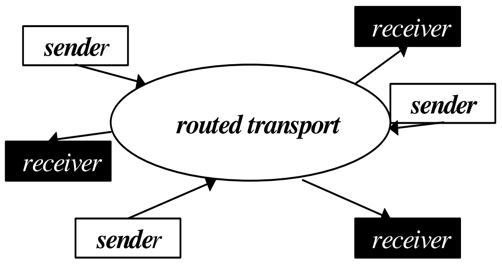


Figure 1-3: The routing functionality of a modern transport protocol conceals the network topology from the application designer.

Transport services generally offer at least the features of the underlying communication hardware. Thus, the most widely used communication services include a way to send a message to a destination, to broadcast a message, and to multicast a message. Unlike the communications hardware versions of these services, however, transport-layer interfaces tend to work with logical addresses and to automatically route messages within complex environments that may mix multiple forms of communication hardware, or include multiple communication subnetworks bridged by routing devices or computers.

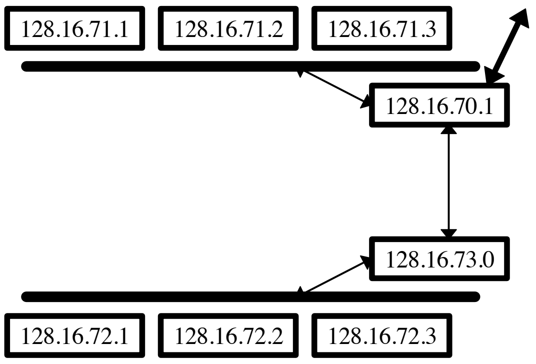


Figure 1-4: A typical network may have several interconnected subnetworks and a link to the internet

All of this is controlled using *routing tables,* like the one shown below. A routing table is a data structure local to each computer in a network – each computer has one, but the contents will generally not be identical from machine to machine. The table is indexed by the logical address of a destination computer, and entries contain the hardware device on which messages should be transmitted (the "next hop" to take). Distributed protocols for dynamically maintaining routing tables have been studied for many years, and seek to minimize the number of hops a message needs to take to reach its destination but to also spread load evenly and route around failures or congested nodes. In practice, however, static routing tables are probably more common: these are maintained by a system administrator for the network and generally offer a single route from a source to each destination. Chapter 3 discusses some of the most common transport services in more detail.

Destination	**Route Via**	**Forwarded By**	**Estimated distance**
128.16.71.*	**Outgoing link 1**	**(direct)**	**1 hop**

128.16.72.*	**Outgoing link 2**	**128.16.70.1**	**2 hops**
128.16.70.1	**Outgoing link 2**	**(direct)**	**1 hop**
****.*.*.****	**Outgoing link 2**	**128.16.70.1**	**(infinite)**

Figure 1-5: A sample routing table, such as might be the one used by computer 128.16.73.0 in Figure 1-4.

1.2.3 Reliable transport software and communication support

A limitation of the basic message passing services discussed in Section 1.2.2 is that they operate at the level of individual messages, and provide no guarantees of reliability. Messages can be lost for many reasons, including link failures, failures of intermediate machines on a complex multi-hop route, noise that causes corruption of the data in a packet, lack of buffering space (the most common cause), and so forth. For this reason, it is common to layer a reliability protocol over the message passing layer of a distributed communication architecture. The result is called a *reliable communication channel*. This layer of software is the one that the OSI stack calls the "session layer", and corresponds to the TCP protocol of the Internet. UNIX programmers may be more familiar with the notion from their use of "pipes" and "streams" [Rit84].

The protocol implementing a reliable communication channel will typically guarantee that lost messages will be retransmitted and that out-of-order messages will be resequenced and delivered in the order sent. Flow control and mechanisms that choke back the sender when data volume becomes excessive are also common in protocols for reliable transport [Jac88]. Just as the lower layers can support one-to-one, broadcast and multicast communication, these forms of destination addressing are also potentially interesting in reliable transport layers. Moreover, some systems go further and introduce additional reliability properties at this level, such as authentication (a trusted mechanism for verifying the identity of the processes at the ends of a communication connection), or security (trusted mechanisms for concealing the data transmitted over a channel from processes other than the intended destinations). In Chapter 3 we will begin to discuss these options, as well as some very subtle issues concerned with how and when connections report failure.

1.2.4 "Middleware": Software tools, utilities, and programming languages

The most interesting issues that we will consider in this textbook are those relating to programming environments and tools that live in the middle, between the application program and the communications infrastructure for basic message passing and support for reliable channels.

Examples of important middleware services include the naming service, the file system, the time service, and the security "key" services used for authentication in distributed systems. We will be looking at all of these in more detail below, but we review them briefly here for clarity.

A naming service is a collection of user-accessible directories that map from application names (or other selection criteria)to network addresses of computers or programs. Name services can play many roles in a distributed system, and represent an area of intense research interest and rapid evolution. When we discuss naming, we'll see that the whole question of what a name "represents" is itself subject to considerable debate, and raises important questions about notions of abstraction and services in distributed computing environments. Reliability in a name service involves issues such as trust – can one trust the name service to truthfully map a name to the correct network address? How can one know that the object at the end of an address is the same one that the name service was talking about? These are fascinating issues, and we will have a lot to say about them later (see, for example, Section 7.2).

From the outset, though, the reader may want to consider that if an intruder breaks into a system and is able to manipulate the mapping of names to network addresses, it will be possible to interpose all sorts of "snooping" software components in the path of communication from an application to the services it is using over the network. Such attacks are now common on the Internet and reflect a fundamental issue, which is that most network reliability technologies tend to trust the lowest level mechanisms that map from names to addresses and that route messages to the correct host when given a destination address.

A time service is a mechanism for keeping the clocks on a set of computers closely synchronized and close to "real time". Time services work to overcome the inaccuracy of inexpensive clocks used on many types of computers, and are important in applications that either coordinate actions using real-time, or that make use of time for other purposes, such as to limit the lifetime of a cryptographic key or to timestamp files when they are updated. Much can be said about time in a distributed system, and we will spend a considerable portion of this textbook on issues that revolve around the whole notion of "before" and "after" and their relation to intuitive notions of time in the real world. Clearly, the reliability of a time service will have important implications for the reliability of applications that make use of time, so time services and associated reliability properties will prove to be important in many parts of this textbook.

Authentication services are, perhaps surprisingly, a new technology that is lacking in most distributed computing environments. These services provide trustworthy mechanisms for determining who sent a message, for making sure that the message can only be read by the intended destination, and for restricting access to private data so that only authorized access can occur. Most modern computing systems evolved from a period when access control was informal and based on a core principle of trust among users. One of the really serious implications is that distributed systems that want to superimpose a security or protection architecture on a heterogeneous environment must overcome a pervasive tendency to accept requests without questioning them, to believe the user-id information including in messages without validating it, and to route messages wherever they may wish to go.

If banks worked this way, one could walk up to a teller in a bank that one had never visited before and pass that person a piece of paper requesting a list of individuals that have accounts in the branch. Upon studying the response and learning that "W. Gates" is listed, one could then fill out an account balance request in the name of W. Gates, asking how much money is in that account. And after this, one could withdraw some of that money, up to the bank's policy limits. At no stage would one be challenged: the identification on the various slips of paper would be trusted for each operation. Such a world model may seem bizarrely trusting, but it is the model from which modern distributed computing systems emerged.

1.2.5 Distributed computing environments

An important topic around which much of this book is oriented concerns the development of general purpose tools from which specialized distributed systems can be constructed. Such tools can take many forms, ranging from the purely conceptual – for example, a methodology or theory that offers useful insight into the best way to solve a problem or that can help the developer confirm that a proposed solution will have a desired property. A tool can offer practical help at a very low level, for example by eliminating the relatively mechanical steps required to encode the arguments for a remote procedure call into a message to the server that will perform the action. A tool can embody complex higher level behavior, such as a protocol for performing some action or overcoming some class of errors. Tools can even go beyond this, taking the next step by offering mechanisms to control and manage software built using other tools.

It has become popular to talk about distributed systems that support *distributed operating environments* – well integrated collections of tools that can be used in conjunction with one another to carry out potentially complex distributed programming tasks. Examples of distributed programming environments are the Open Network Computing (ONC) environment of SUN Microsystems, The Distributed Computing (DCE) of Open Software Foundation, the various CORBA-compliant programming tools that have become popular among C++ programmers who work in distributed settings, and the Isis Toolkit and the Horus system; these last two being systems developed by the author of this text and his colleagues, which will be discussed in Chapter 18.

Distributed systems architectures undertake to step even beyond the notion of a distributed computing environment. An architecture is a general set of design principles and implementation standards by which a collection of "compliant" systems can be developed. In principle, multiple systems that implement the same architecture will interoperate, so that if vendors implement competing solutions, the resulting software can still be combined into a single system with components that might even be able to communicate and cooperate with one another. The Common Request Broker, or CORBA, is probably the best known distributed computing architecture; it is useful for building systems using an object-oriented approach in which the systems are developed as modules that cooperate. Thus, CORBA is an architecture, and the various CORBA-based products that comply with the architecture are distributed computing environments.

1.2.6 End-user applications

One might expect that the "end of the line" for a layered distributed systems architecture would be the application level, but this is not necessarily the case. A distributed application might also be some sort of operating system service built over the communications tools that we have been discussing. For example, the distributed file system is an application in the sense of the OSI layering, but the user of a computing system might think of the file system as an operating system service over which applications can be defined and executed. Within the OSI layering, then, an application is any free-standing solution to a well defined problem that presents something other than a communications abstraction to its users. The distributed file system is just one example among many. Others include message bus technologies, distributed database systems, electronic mail, network bulletin boards, and the World-Wide-Web. In the near future, computer supported collaborative work systems and multimedia digital library systems are likely to emerge as further examples in this area.

A limitation of a layering like the OSI hierarchy is that it doesn't really distinguish these sorts of applications, which provide services to higher level distributed applications, from what might be called end-user solutions, namely programs that operate over the communications layer to directly implement commands for a human being. One would like to believe that there is much more structure to a distributed air traffic control system than to a file transfer program, yet the OSI hierarchy views both as examples of ''applications.'' We lack a good classification system for the various types of distributed applications.

In fact, even complex distributed applications may merely be components of even larger-scale distributed systems – one can easily imagine a distributed system that uses a distributed computing toolkit to integrate an application that exploits distributed files with one that stores information into a distributed database. In an air-traffic control environment, availability may be so critical that one is compelled to run multiple copies of the software concurrently, with one version backing up the other. Here, the entire air traffic control system is at one level a complex distributed application in its own right, but at a different ‘‘meta’’ level, is just a component of an over-arching reliability structure visible on a scale of hundreds of computers located within multiple air traffic centers.

1.3 Critical Dependencies

One of the major challenges to building reliable distributed systems is that computer networks have evolved to have a great many “dependencies” on a variety of technologies. Some of the major ones are identified in Figure 1-6, however the set is growing steadily and this figure is not necessarily complete. Critical applications often introduce new servers and critical components not shown here, nor does the figure treat dependencies on hardware components of the distributed infrastructure, such as the communication network itself, power supply, or hardware routers. Moreover, the telecommunications infrastructure underlying a typical network application is itself a complex network with many of the same dependencies internal to itself, together with additional ones such as the databases used to resolve mobile telephone numbers or to correctly account for use of network communication lines.

Fortunately, many of these services are fairly reliable, and one can plan around potential outages of such critical services as the network information service. The key issue is to understand the technology dependencies that can impact reliability issues for a specific application and to program solutions into the network to detect and work around potential outages. In this textbook we will be studying technical options for taking such steps. The emergence of integrated environments for reliable distributed computing will, however, require a substantial effort from the vendors offering the component technologies: an approach in which reliability is left to the application inevitably overlooks the problems that can be caused when such applications are forced to depend upon technologies that are themselves unreliable for reasons beyond the control of the developer.

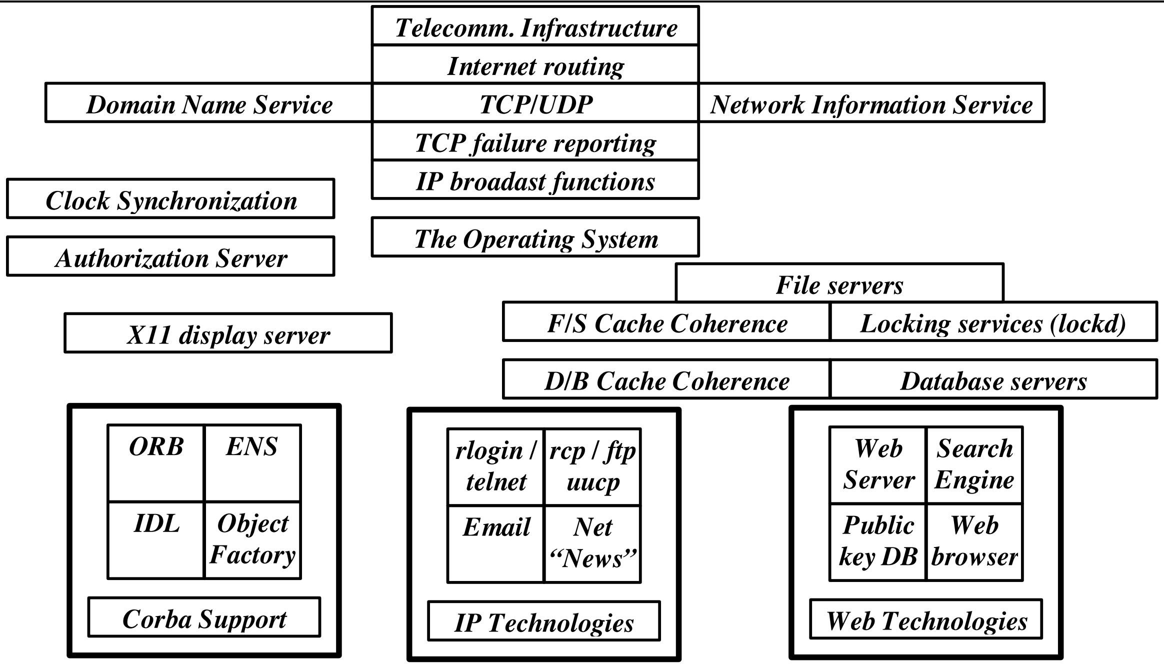


Figure 1-6: Technologies on which a distributed application may "depend" in order to provide correct, reliable behavior. The figure is organized so that dependencies are roughly from top to bottom (the lower technologies being dependent upon the upper ones), although a detailed dependency graph would be quite complex. Failures in any of these technologies can result in visible application-level errors, inconsistency, security violations, denial of service, or other problems. These technologies are also interdependent in complex and often unexpected ways. For example, some types of UNIX workstations will hang (freeze) if the NIS (network information service) server becomes unavailable, even if there are duplicate NIS servers that remain operational. Moreover, such problems can impact an application that has been running normally for an extended period and is not making any explicit new use of the server in question.

1.4 Next Steps

While distributed systems are certainly layered, Figure 1-6 makes it clear that one should question the adequacy of any simple layering model for describing reliable distributed systems. We noted, for example, that many governments have mandated the use of the ISO layering for description of distributed software. Yet there are important reliability technologies that require structures inexpressible in this layering, and it is unlikely that those governments intended to preclude the use of reliable technologies. More broadly, the sorts of complex layerings that can result when tools are used to support applications that are in turn tools for still higher level applications are not amenable to any simple description of this nature. Does this mean that users should refuse the resulting complex software structures, because they cannot be described in terms of the standard? Should they accept the perspective that software should be used but not described, because the description methodologies seem to have lagged the state of the art? Or should governments insist on new standards each time a new type of system finds it useful to step outside of the standard?

Questions such as these may seem narrow and almost pointless, yet they point to a deep problem. Certainly, if we are unable to even describe complex distributed systems in a uniform way, it will be very difficult to develop a methodology within which one can reason about them and prove that they respect desired properties. On the other hand, if a standard proves unwieldy and constraining, it will eventually become difficult for systems to adhere to it.

Perhaps for these reasons, there has been little recent work on layering in the precise sense of the ISO hierarchy: most researchers view this as an unpromising direction. Instead, the notions of structure and hierarchy seen in ISO have reemerged in much more general and flexible ways: the object class hierarchies supported by technologies in the CORBA framework, the layered protocol stacks supported in operating systems like UNIX or the *x*-Kernel, or in systems such as Horus. We'll be reading about these uses of hierarchy later in the textbook, and the ISO hierarchy remains popular as a simple but widely understood framework within which to discuss protocols.

1.5 Additional Reading

General discussion of network architectures and the ISO hierarchy: [Tan88, Com91, CS91, CS93, ANSA91a, ANSA91b, ANSA89, CD90, CDK94, XTP95]. Pros and Cons of layered architectures: [CT87, RST88, RST89, Ous90, AP93, KP93, KC94, BD95]. Reliable stream communication: [Rit84, Jac88, Tan88, Com91, CS91, CS93, CDK94]. Failure Models and Classification: [Lam78b, Lam84, Ske82b, FLP85, ST87, CD90, Mar90, Cri91a, CT91, CHT92, GR93, SM94].

2. Communication Technologies

Historically, it has rarely been necessary to understand details of the hardware components from which a computing system was constructed if one merely wishes to develop software for it. The pressure to standardize operating systems, and the presentation of devices within them, created a situation in which it sufficed to understand the way that the operating system abstracted a device in order to use it correctly.

For example, there are a great many designs for computer disk storage units and the associated device controllers. Each design offers its own advantages and disadvantages when compared with the others, and any systems architect charged with selecting a data storage device would be wise to learn about the state of the art before making a decision. Yet, from a software perspective, device attributes are largely hidden. The developer normally considers a disk to be a device on which files can be stored, having various layout parameters that can be tuned to optimize I/O performance, and characterized by a set of speed and reliability properties. Developers of special classes of applications, such a multi-media image servers, may prefer to work with a less abstracted software interface to the hardware, exploiting otherwise hidden features at the cost of much greater software complexity. But for the normal user, one disk is much like any other.

To a considerable extent, the same is true for computer networking hardware. There are a number of major classes of communications devices, differing in speed, average access latency, maximum capacity (packets per second, bytes of data per second), support for special addressing modes, etc. However, since most operating systems implement the lowest layers of the OSI hierarchy as part of the device driver or communications abstraction of a system, applications can treat these devices interchangeably. Indeed, it can be quite difficult to determine just what the communications topology of a system actually is, because many operating systems lack services that would permit the user to query for this information.

In the remainder of this chapter, we review communication hardware in very superficial terms, giving just enough detail so that the reader should be familiar with technology names and properties, but without getting into the level of technical issues that would be important in designing the network topology for a demanding enterprise.

Throughout this chapter, the reader will notice that we use the term *packet* to refer to the type of messages that can be exchanged between communications devices. The distinction between a packet and a *message*, throughout this book, is that a message is a logical object generated by the application for transmission to one or more destinations. A message may be quite large, and can easily exceed the limits imposed by the operating system or the communications hardware. For transmission, messages are therefore fragmented into one or more packets, if necessary. A packet, then, is a hardware level message that often respects hardware-imposed size and format constraints, and may contain just a fragment of an application-level message.

2.1 Types of Communication Devices

Communications devices can be coarsely partitioned into functional classes:

- *Point to point:* This is a class of devices implementing packet or data passing between two computers. A good example is a pair of modems that communicate over a telephone wire. The Internet is composed of point to point communications devices that form a wide-area architecture, to which individual local-area networks are connected through "Internet Gateway" devices.

- *Multiple access:* This class of devices permit many computers to share a single communications medium. For example, using the popular *ethernet* architecture, a single coaxial cable can be used to wire a floor of a building or some other moderately large area. Computers can be connected to this cable by "tapping" into it, which involves inserting a special type of needle through the outer cover of the coaxial conductor and into the signal conducting core. The device interfaces implement a protocol in hardware to avoid collisions, which occur if several machines attempt to send packets at the same time.
- *Mesh, tree, and crossbar architectures:* This class of devices consists of point to point links that connect the individual computer to some form of switching mechanism. Messages are typically very small, and are routed at hardware link speeds through the switches and to their destinations. Connections of this sort are most often used in parallel computers, but are also being adapted for very high speed communication in clusters of more conventional computing nodes.
- *ATM switches:* Asynchronous Transfer Mode, or ATM, is an emerging standard for packet-switching between computers, over communications links and switches of varied speeds and properties. ATM is based on a star architecture, in which optical fibers connect individual computers to switches; these behaving much like the communication buses seen in parallel computers. ATM is designed for very high speed communications, including optical fiber that can operate at speeds of 2.5Gbits per second or more. Even the first generation systems are extremely fast, giving performance of 155Mbits/second for individual connections ("OC3" in the ATM terminology).
- *Bridges:* A bridge or *router* (we'll use the term bridge to avoid confusion with the notion of routing) is a special-purpose communications computer that links multiple networking devices, by forwarding the packets received on either device onto the other. Bridges introduce some latency, which is called a "hop delay", but normally operate as fast as the devices they interconnect. Bridges tend to lose packets if a destination network is heavily loaded and data cannot be forwarded as fast as it is received from the originating network. Bridges can be programmed to forward messages selectively; this is often exploited to avoid a problem whereby the load on a network can grow without limit as the size of the network is increased – the load on a bridged network is the sum of the load local to a segment, and the load forwarded over the bridge, and can be much less than the sum of the loads on all segments.

2.2 Properties

Communications devices vary enormously in their properties, although this variability is often concealed by the layers of systems software through which applications operate. In simple terms, communications devices can be "rated" by a series of metrics:

- *The maximum data throughput of the device.* Speed is normally measured in terms of the number of bytes of data per second that can be transmitted. Vendors often quote figures in terms of bits per second, referring to the performance seen "on the wire" as information is transmitted. In either case, one should be aware that speed figures often do not include overhead such as start and stop bits, headers and trailers, and mandatory dead-space between packets. These factors can greatly reduce the effective performance of a device, making it difficult to obtain even as much as half of the maximum theoretical performance from an application program. Indeed, it is not uncommon for the most easily used communication primitives to offer performance that is an order of magnitude or more poorer than that of the hardware! This often forces the application designer to chose between performance and software complexity.
- *The number of packets per second that can be sent.* Many devices have a start-up overhead associated with sending packets, and for some devices the sending interface must wait for access to the communications medium. These factors combine to limit the number of packets per second that a device can send, and when packets can be of variable size, can also imply that to achieve maximum data throughput, the application must send large packets.

- *The end-to-end latency of the device*. This is a measure of how much time elapses from when a packet starts to be transmitted and when it is first presented to the receiving machine, and is generally an quoted as an average figure, that will actually vary depending on the degree to which the network is loaded at the time a packet is transmitted.
- *The reliability of the device*. All commonly used communications hardware includes automatic mechanisms for detection and rejection of corrupted data. These methods operate using checksums (CRC computations) and are not infallible, but in practice it is normal to treat communications hardware as failing only by packet loss. The reliability of a communications technology is a measure of the percentage of packets that can be lost in the interface or on the wire, as an average. Average reliability is often very high – not uncommonly, hardware approaches perfect reliability. However, it should be kept in mind that an average loss rate may not apply in an exceptional situation, such as an interface that is experiencing intermittent failures, a poorly connected ethernet tap, or a pattern of use that stresses some sort of uncommon loss problem associated with a technology. From the perspective of the builder of a reliable distributed system, these factors imply that a communications device should be considered somewhat bimodal: having one reliability level in the normal case, but perhaps having a second, much poorer reliability level, in exceptional cases that the system may need to tolerate or reconfigure around.
- *Security*. This is a measure of the extent to which the device protects the contents of packets from eavesdroppers. Some devices achieve very high levels of security in hardware; others are completely open to eavesdropping and require that measures be taken in software if data security is desired.
- *Privacy*. This is a measure of the extent to which the device conceals the source and destination of packets from eavesdroppers. Few devices offer privacy properties, and many security features are applied only to the data portion of a packet, and hence offer little help if privacy is desired. However, there are technologies for which privacy is a meaningful concept. For example, on an ethernet, interfaces can be programmed to respond to a small set of addresses within a very large space of potential addresses. This feature potentially allows the destination of a packet to be concealed from listeners. On the other hand, the ethernet standard never permits the address of the sender to be reprogrammed, and consequently will always reveal the address of the communications interface from which a packet was sent.

2.3 Ethernet

At the time of this writing, ethernet is the most widely used communications technology for local-area networks (networks within a limited physical region, such as a single floor of a building). Bridged ethernets are the most common technology for networks within small enterprises, such as a large company at a single site.

As summarized earlier, the basic technology underlying an ethernet is a shared coaxial cable, on which signals are transmitted using a modulation technology similar to that of a radio. Packets have a fixed maximum size of 1400 bytes, but the size can be varied as long as this limit is not exceeded. In practice, software that runs over Ethernet will often be limited to approximately 1024 bytes of "payload" in each packet; the remaining 376 bytes are then available for headers, trailers, and data representation information. The ethernet itself, however, treats the entire object as data. The specific encoding used to represent packets will not be important to us here, but the basic idea is that each interface is structured into a sending side, and a listening side, and the latter is continuously active.

To receive a message, the listening side waits until it senses a packet header. The incoming packet is then copied to a memory buffer internal to the ethernet interface. To be accepted, a packet must have a valid checksum, and must specify an destination address that the interface has been preprogrammed to accept. Specifically, each interface has some number of programmable address registers, consisting of a "pattern mask" and a corresponding "value mask", each 32-bits in length. The pattern mask specifies bits within the destination address that must exactly match the corresponding bits of the value mask. A pattern mask that selects for all bits of the address will require an exact match between the packet and the value mask. A pattern mask that selects no bits will match every incoming packet – an interface with such an address loaded is said to be in *promiscuous mode*.

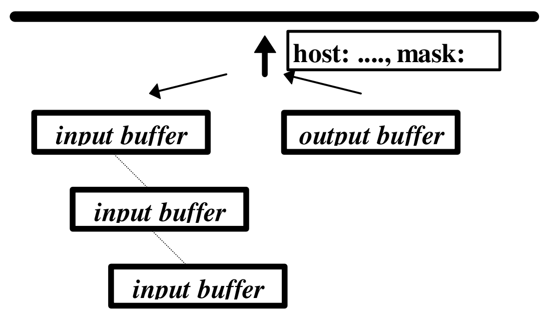


Figure 2-1: Ethernet interface with one queued output buffer and three available input buffers. A table of up to 64 (host,mask) pairs controls input selectivity.

A received packet is copied into memory in the host computer, or discarded if no memory for an incoming packet is available. The host is then interrupted. Most ethernet interfaces permit the host to enqueue at least two memory regions for incoming messages, and some permit the host to chain a list of memory regions. Most also permit multiple (address,mask) pairs to be loaded into the interface, often as many as 64.

To send a packet, the ethernet interface waits for a pause between packets – a time when its listening side is idle. It then transmits the packet, but also listens to its own transmission. The idea is that if two ethernets both attempt to send at the same time, a collision will occur and the messages will overwrite one another, causing a noise burst. The receive logic will either fail immediately, or the checksum test will fail, since anything the interfaces read back in will be damaged by the collision, and the sending logic will recognize that a problem has occurred. The hardware implements an *exponential back-off* algorithm, in which the sending side delays for a randomly selected period of time within an interval that starts at a small value but doubles with each successive collision up to a maximum upper value. Although the probability of a collision on a first attempt to send can be high when the ethernet becomes loaded, exponential back-off has been shown to give very good average access behavior with excellent fairness properties[1]. Because collisions are often detectable within a few bits after starting to send, ethernets lose little data to collisions even under heavy load, and the back-off algorithm can be shown to provide very uniform delays for access to the medium over very large numbers of senders and very large excess loads.

As a general rule, although a single interface can send multiple packets, a small amount of dead space will separate each packet in the stream, because a small amount of work by the operating system is normally needed before each successive packet can be transmitted, and because the ethernet hardware logic requires a small amount of time to compare the checksum on the echo of the outgoing packet, and to trigger an interrupt to the device driver, before starting to send a new packet. In contrast, when more than one interface is used to send data, sequences of "back to back" packets can be generated, potentially forcing the interface to accept several packets in a row with almost no delay between them. Obviously, this can result in packet loss if the chain of memory for incoming messages is exhausted. However, precisely because an ethernet is shared, the probability that any one interface will be the destination for any large number of back-to-back packets is low. File system servers and bridges, which are more likely to receive

[1] Developers of real-time computing systems, for settings such as process-control, have developed deterministic back-off algorithms that have extremely predictable behavior under heavy load. In these approaches, the loaded behavior of an ethernet is completely characterized -- provided, of course, that all interfaces use the same algorithm.

back-to-back packets, compensate for this by using long chains of buffers for incoming messages, and implementing very lightweight logic for dealing with received messages as quickly as possible.

An interesting feature of the ethernet is that it supports both broadcast and multicast in hardware. The two features are implemented in the same way. Before any communication is undertaken, the ethernet interface is preloaded with a special address – one that is the same on all machines within some set. Now, if a packet that contains this address is transmitted, all machines in that set will receive a copy, because all of their interfaces will detect a match.

To support broadcast, a special address is agreed upon and installed on all interfaces in the entire network, typically at the time the machine is first booted. Broadcast packets will now be received by every machine, offering a way to distribute the same data to a great many machines at very low cost. However, one should keep in mind that each interface individually computes the checksum, and hence that some interfaces may discard a packet as corrupted, while others accept it. Moreover, some machines may lack input buffers and hence may be incapable of accepting packets that are correctly received by the interface. Thus, ethernet broadcast can potentially send a single packet to all machines on a network, but in practice the technology is not a reliable one.

Multicast uses precisely the same approach, except that a subset of machines pick a group address that is unique within the network and install this into their interfaces. Messages destined to a multicast address will be accepted (up to checksum failures) by just these machines, and will be ignored by others. The maximum number of multicast addresses that an interface can support varies from vendor to vendor. As in the case of broadcast, hardware multicast is reasonably reliable but not absolutely so.

Earlier, we commented that even a very reliable communications device may exhibit modal behavior whereby reliability can be much poorer for certain communication patterns. One example of this problem is termed the *broadcast- or multicast-storm*, and arises when broadcast or multicast is used by multiple senders concurrently. In this situation, it becomes much more likely that a typical network interface will actually need to accept a series of back-to-back packets – with sufficiently many senders, chains of arbitrary length can be triggered. The problem is that in this situation the probability that two back to back messages are destined to the same machine, or set of machines, becomes much higher than for a more typical point-to-point communication load. As a result, the interface may run out of buffers for incoming packets and begin to drop them.

In a broadcast storm situation, packet loss rises to very high levels, because network interfaces become chronically short of memory for incoming packets. The signature of a broadcast storm is that heavy use of broadcast or multicast by multiple senders causes a dramatic increase in the packet loss rate throughout the network. Notice that the problem will affect any machine that has been programmed to accept incoming broadcasts, not just the machines that make meaningful use of the packets after they arise. Fortunately, the problem is uncommon if just a single sender is initiating the broadcasts, because a single sender will not generate back-to-back packets.

2.4 FDDI

FDDI is a multi-access communication technology based upon a ring architecture, in which interfaces are interconnected by shielded "twisted pair" wiring. An interface plays a dual role:

- As a *repeater*, an FDDI interface receives messages from the interface to its left, accepts those messages that match an incoming address pattern (similar to ethernet), and then (accepted or not), forwards the message to the interface on the right. Forwarding occurs bit by bit or in small blocks of bits, so the delay associated with forwarding packets can be very low.

- As a *transmitter*, an FDDI interface waits until it has permission to initiate a packet, which occurs when there is no other packet being forwarded, and then sends its packet to the interface on its right. When the packet has completed its trip around the ring, the transmitter receives it from the interface to the left and deletes it. Status information is available in the packet trailer, and can be used to immediately retransmit a packet that some intended destination was unable to accept because of a shortfall of memory or because it detected a checksum error.

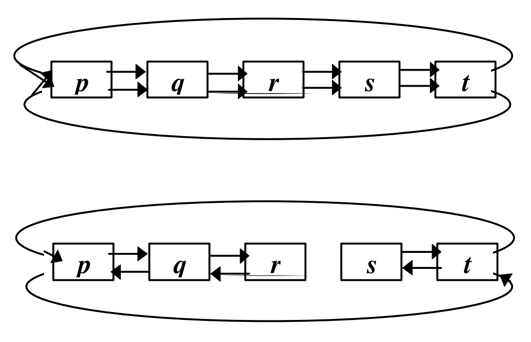


Figure 2-2: An FDDI ring is able to heal itself if a link breaks; as seen in the lower example.

Finally, FDDI has a built-in fault-tolerance feature: if a link fails, FDDI will automatically reconfigure itself to route around it, as illustrated in

Figure 2-2. In contrast, a severed ethernet will either become inoperative, or will partition into two or more segments that are disconnected from one-another..

FDDI throughput (150Mbits/second) is about 15 times greater than standard 10-Mbit ethernet, although high speed 100-Mbit ethernet interfaces have recently been introduced that can approach FDDI performance. Latency for an FDDI ring, in particular, is poorer than that of an ethernet, because of the protocol used to wait for permission to send, and because of delays associated with forwarding the packet around the ring. In complex environments, ethernets or FDDI rings may be broken into segments, which are connected by some form of bridge or routing device; these environments have latency and throughput properties that are more difficult to quantify, because the topology of the interconnection path between a pair of computers can significantly impact the latency and throughput properties of the link.

2.5 B-ISDN and the Intelligent Network

B-ISDN is a standard introduced by telecommunications switching companies for advanced telephone services. ISDN stands for "integrated services digital network", and is based on the idea of supporting a communications system that mixes digital voice with other types of digital data, including video. The "B" stands for broadband and is a reference to the underlying data links, which operate at the extremely high speeds needed to handle these kinds of data.

Today, the telephone system is relatively inflexible: one can telephone from more or less any telephone to any other, but the telephone numbers correspond to static locations. To the degree that mobile telephones are used, one calls the mobile device using a scheme in which requests are routed through the home location; thus, were a traveller to attempt to use a cellular telephone in Japan to contact another cellular traveller in Japan, the phone call might require participation of their home telephone companies in upstate New York.

In the future, however, a great variety of innovative new telephone-based communication services will become available. These will include telecommunications services that mix various forms of media: voice, image, computer data, and abstract data types capable of performing computations or retrieving desired information. Mobility of users and services will greatly increase, as will the sophistication of the routing mechanisms used to track down the entity to which a call should be connected. Thus, our Japanese travellers will be connected through some direct service, and a call for a home-delivered pizza will be routed to a delivery truck in the neighborhood. Moreover, whereas contemporary telecommunications systems are very "different" from computer communications architectures such as the World Wide Web, these will be increasingly integrated in the future and will eventually merge into a single infrastructure with the properties of both.

In many communities, ISDN will bring high bandwidth connections into the home, at reasonable cost. This development will revolutionize use of the Web, which is currently bandwidth limited "from the curb to the home" and hence only useable in limited ways from home computing platforms.

High speed communication is already available in the workplace, however, and this trend is already creating innovative new businesses. It is widely expected that a boom in commerce associated with the communications infrastructure will follow as it reaches the average household. For example, one can imagine small companies that offer customized services to a world-wide clientel over the network, permitting entreprenurial activities that will tap into an immense new market. The computer user of the future may well use the machine to shop in Paris, decorate his or her apartment with original African art comissioned directly from the artist, and visit potential travel destinations through the network before booking a hotel room. France's experience with Minitel is often cited as a sign that such a trend can succeed: the Web, with its richer user environment, is similar to Minitel but could reach a much larger community.

Of course, the promise of this new world comes with challenges. As we come to rely more and more heavily on innovative communications technologies for day to day activities and these new forms of information-based work and commerce grow in importance, the reliability requirements placed on the underlying technology infrastructure will also grow. Thus, there is a great potential for economic growth and increased personal freedom associated with the new communications technologies, but also a hidden implication that the software implementing such systems be secure, private, fault-tolerant, consistent in its behavior, and trustworthy.

These properties do not hold for many of the prototype systems that have so excited the public, and a significant change in the mindset of the developers of such applications will be needed before reliability of this sort becomes routine. For example, fraudulent use of telephone systems has grown with the introduction of new forms of flexibility in the system, and attacks of all forms on secure or critical information-based applications have risen steadily in recent years. These range from malicious or careless insiders whose actions disrupt critical systems, to aggressive attacks by hackers, terrorists, and other agents whose goal is to cause damage or to acquire secrets.

Figure 2-3: Challenges of an emerging information superhighway include providing guarantees of privacy, availability, consistency, security, and trustworthiness in a wide range of innovative applications. Prototypes of the new services that may someday become critical often lack the sorts of strong guarantees that will eventually be required if these types of systems are to play the role that is envisioned by the public and governments.

Layered over B-ISDN, which is an infrastructure standard, is the emerging *intelligent network*, an elaborate software architecture for supporting next-generation telecommunications services. Such services have been slower to emerge than had originally been predicted, and many companies have become cautious in accessing intelligent network prospects for the near-term future. However, it may simply be the case that the intelligent network has been harder to deploy than was originally predicted: there are many signs that the long-predicted revolution in telecommunications is really happening today.

The B-ISDN architecture is elaborate, and the intelligent network is even more so. Our focus on reliability of general distributed computing systems prevents us from discussing either in any detail here. However, one interesting feature of B-ISDN merits mention. Although destination addresses in ISDN are basically telephone numbers, ISDN interprets these in a very flexible manner that permits the architecture to do far more than just creating connections between telephones and other paired devices. Instead, the ISDN architecture revolves around the notion of an intelligent switching system: each packet follows a route from the source through a series of switches to its destination. When this route is first established, each switch maps the destination telephone number to an appropriate outgoing connection (to another switch), to a local point of data delivery (if this switch serves the destination), or to a *service*. The decision is made by looking up the telephone number in a database, using a software procedure that can be reprogrammed to support very elaborate behaviors.

For example, suppose that a telephone company wanted to offer a special communications service to computer vendors in some metropolitan area. This service would offer a single telephone number to each vendor and would arrange to automatically route a call to the mobile telephone of the computer repair person physically closest to the caller.

To implement this service using B-ISDN, the telephone company would make use of a database it already maintains, giving the locations of mobile telephone units. As a call is routed into a switch, the company would sense that the destination is one of the special telephone numbers assigned to this new service, and would invoke a database search algorithm programmed to lookup the physical address of the caller, and then match this with the locations of service vehicles for the called organization to pick the closest one, routing the call to the computer vendor's switchboard if the lookup fails. Although timing constraints for this process are demanding (actions are generally required within a small fraction of a second), modern computers are becoming fast enough to work within these sorts of deadlines. And, one can imagine a great number of other B-ISDN services, including message centers, automatic playback of pre-recorded information, telephone numbers that automatically patch into various types of public or private information bases, and so forth. The potential is huge.

B-ISDN is also illustrative of how advances in telecommunications switching technology are creating new demands for reliable distributed software services. It is common to require that telephone systems maintain extremely high levels of reliability – a typical requirement is that not more than one call in 100,000 be dropped, and downtime for an entire switch may be limited to seconds per year or less – switches are increasingly used to support critical services such as 911 emergency numbers and communication between air traffic controllers and police vehicles. caller.

Reliability of this sort has many implications for developers of advanced switching systems. The switches themselves must be paired, and protocols for doing so have been standardized as part of an architecture called Signalling System 7 (SS7), which is gradually entering into world-wide use. The co-processors on which intelligent services reside are often constructed using fault-tolerant computing hardware. The software that implements the switching logic must be self-managing, fault-tolerant, and capable of supporting on-line upgrades to new versions of applications and of the operating system itself. And, because many services require some form of distributed database, such as the database of location information that arose in the telephone dispatch example above, sets of coprocessors will often need to be organized into distributed systems that manage dynamically changing replicated data and take actions in a consistent but decentralized manner. For example, routing a call may require independent routing decisions by the service programs associated with several switches, and these decisions need to be based upon consistent data or the call will eventually be dropped, or will be handled incorrectly.

B-ISDN, then, and the intelligent network that it is intended to support, represent good examples of settings where the technology of reliable distributed computing is required, and will have a major impact on society as a whole. Given solutions to reliable distributed computing problems, a vast array of useful telecommunication services will become available starting in the near future and continuing over the decades to come. One can imagine a telecommunications infrastructure that is nearly ubiquitous and elegantly integrated into the environment, providing information and services to users without the constraints of telephones that are physically wired to the wall and computer terminals or televisions that weigh many pounds and are physically attached to a company's network. But the dark side of this vision is that without adequate attention to reliability and security, this exciting new world will also be erratic and failure-prone.

2.6 ATM

Asynchronous Transfer Mode, or ATM, is an emerging technology for routing small digital packets in telecommunications networks. When used at high speeds, ATM networking is the "broadband" layer underlying B-ISDN; thus, an article describing a B-ISDN service is quite likely to be talking about an application running on an ATM network that is designed using the B-ISDN architecture.

ATM technology is considered especially exciting both because of its extremely high bandwidth and low latencies, and because this connection to B-ISDN represents a form of direct covergence between the telecommunications infrastructure and the computer communications infrastructure. With ATM, for the first time, computers are able to communicate directly over the data transport protocols used by the telephone companies. Over time, ATM networks will be more and more integrated with the telephone system, offering the possibility of new kinds of telecommunications applications that can draw immediately upon the world-wide telephone network. Moreover, ATM opens the door for technology migration from those who develop software for computer networks and distributed systems into the telecommunications infrastructure and environment.

The packet switches and computer interfaces needed in support of ATM standards are being deployed rapidly in industry and research settings, with performance expected to scale from rates comparable to those of a fast ethernet for first-generation switches to gigabit rates in the late 1990's and beyond. ATM is defined as a routing protocol for very small packets, containing 48 bytes of payload data with a 5-byte header. These packets traverse routes that must be pre-negotiated between the sender, destination, and the switching network. The small size of the ATM packets leads some readers to assume that ATM is not really "about" networking in the same sense as an ethernet, with its 1400-byte packets. In fact, however, the application programmer normally would not need to know that messages are being fragmented into such a small size, tending instead to think of ATM in terms of its speed and low latency. Indeed, at the highest speeds, ATM cells can be thought of almost as if they were fat bits, or single words of data being transferred over a backplane.

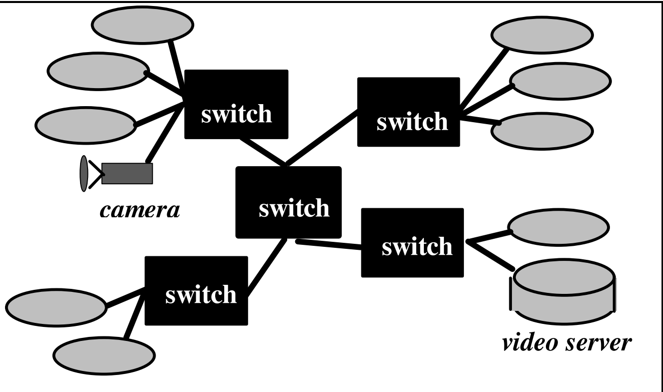


Figure 2-4: Client systems (gray ovals) connected to an ATM switching network. The client machines could be PC's or workstations, but can also be devices, such as ATM frame grabbers, file servers, or video servers. Indeed, the very high speed of some types of data feeds may rule out any significant processor intervention on the path from the device to the consuming application or display unit. Over time, software for ATM environments may be more and more split into a "managerial and control" component that sets up circuits and operates the application and a "data flow" component that moves the actual data without direct program intevension. In contrast to a standard computer network, an ATM network can be integrated directly into the networks used by the telephone companies themselves, offering a unique route towards eventual convergence of distributed computing and telecommunications.

ATM typically operates over point-to-point fiber-optic cables, which route through switches. Thus, a typical ATM installation might resemble the one shown in Figure 2-4. Notice that in this figure, some devices are connected directly to the ATM network itself and not handled by any intermediary processors. The rationale for such an architecture is that ATM devices may eventually run at such high data rates[2] (today, an "OC3" ATM network operates at 155Mbits/second (Mbps), and future "OC24" networks will run at a staggering 1.2Gbps) that any type of software intervention on the path between the data source and the data sink would be out of the question. In such environments, application programs will more and more be relegated to a supervisory and control role, setting up the links and turning the devices on and off, but not accessing the data flowing through the network in a direct way. Not shown are adaptors that might be used to interface an ATM directly to an ethernet or some other local area technology, but these are also available on the market today and will play a big role in many furture ATM installations. These devices allow an ATM network to be attached to an ethernet, token ring, or FDDI network, with seamless communication through the various technologies. They should be common by late in the 1990's.

The ATM header consists of a VCI (2 bytes, giving the virtual circuit id), a VPI (1 byte giving the virtual path id), a flow-control data field for use in software, a packet type bit (normally used to distinguish the first cell of a multi-cell transmission from the subordinate ones, for reasons that will become clear momentarily), a cell "loss priority" field, and a 1-byte error-checking field that typically contains a checksum for the header data. Of these, the VCI and the packet type (PTI) bit are the most heavily used, and the ones we discuss further below. The VPI is intended for use when a number of virtual circuits connect the same source and destination; it permits the switch to multiplex such connections in a manner that consumes less resources than if the VCI's were used directly for this purpose. However, most current ATM networks set this field to 0, and hence we will not discuss it further here.

There are three stages to creating and using an ATM connection. First, the process initiating the connection must construct a "route" from its local switch to the destination. Such a route consists of a path of link addresses. For example, suppose that each ATM switch is able to accept up to 8 incoming links and 8 outgoing links. The outgoing links can be numbered 0-7, and a path from any data source to

[2] ATM data rates are typically quoted on the basis of the maximum that can be achieved through any single link. However, the links multiplex through switches and when multiple users are simultaneously active, the maximum individual performance may be less than the maximum performance for a single dedicated user. ATM bandwidth allocation policies are an active topic of research.

any data sink can then be expressed as a series of 3-bit numbers, indexing each successive hop that the path will take. Thus, a path written as 4.3.0.7.1.4 might describe a route through a series of 6 ATM switches. Having constructed this path, a virtual circuit identifier is created and the ATM network is asked to "open" a circuit with that identifier and path. The ATM switches, one by one, add the identifier to a table of open identifiers and record the corresponding out-link to use for subsequent traffic. If a bidirectional link is desired, the same path can be set up to operate in both directions. The method generalizes to also include multicast and broadcast paths. The VCI, then, is the virtual circuit identifier used during the open operation.

Having described this, however, it should be stressed that many early ATM applications depend upon what are called "permanent virtual channels", namely virtual channels that are preconfigured by a systems administrator at the time the ATM is installed, and changed rarely (if ever) thereafter. Although it is widely predictated that dynamically created channels will eventually dominate the use of ATM, it may turn out that the complexity of opening channels and of ensuring that they are closed correctly when an endpoint terminates its computation or fails will emerge as some form of obstacle that presents this step from occuring.

In the second stage, the application program can send data over the link. Each outgoing message is fragmented, by the ATM interface controller, into a series of ATM packets or "cells". These cells are prefixed with the circuit identifier that is being used (which is checked for security purposes), and the cells then flow through the switching system to their destination. Most ATM devices will discard cells in a random manner if a switch becomes overloaded, but there is a great deal of research underway on ATM scheduling and a variety of so-called *quality of service* options will become available over time. These might include guarantees of minimum bandwidth, priority for some circuits over others, or limits on the rate at which cells will be dropped. Fields such as the packet type field and the cell loss priority field are intended for use in this process.

It should be noted, however, that just as many early ATM installations use permanent virtual circuits instead of supporting dynamically created circuits, many also treat the ATM as an ethernet emulator, and employ a fixed bandwidth allocation corresponding roughly to what an ethernet might offer. It is possible to adopt this approach because ATM switches can be placed into an emulation mode in which they support broadcast, and early ATM software systems have taken advantage of this to layer the TCP/IP protocols over ATM much as they are built over an ethernet. However, fixed bandwidth allocation is inefficient, and treating an ATM as if it were an ethernet somewhat misses the point! Looking to the future, most reseachers expect this emulation style of network to gradually give way to direct use of the ATM itself, which can support packet-switched multicast and other types of communication services. Over time, "value-added switching" is also likely to emerge as an important area of competition between vendors; for example, one can easily imagine incorporating encryption and filtering directly into ATM switches and in this way offering what are called *virtual private network* services to users (Chapters 17 and 19).

The third stage of ATM connection management is concerned with closing a circuit and freeing dynamically associated resources (mainly, table entries in the switches). This occurs when the circuit is no longer needed. ATM systems that emulate IP networks or that use permanent virtual circuits are able to skip this final stage, leaving a single set of connections continuously open, and perhaps dedicating some part of the aggregate bandwidth of the switch to each such connection. As we evolve to more direct use of ATM, one of the reliability issues that may arise will be that of detecting failures so that any ATM circuits opened by a process that later crashed will be safely and automatically closed on its behalf. Protection of the switching network against applications that erroneously (or maliciously) attempt to monopolize resources by opening a great many virtual circuits will also need to be addressed in future systems.

ATM poses some challenging software issues. Communication at gigabit rates will require substantial architectural evolution and may not be feasible over standard OSI-style protocol stacks, because of the many layers of software and protocols that messages typically traverse in these architectures. As noted above, ATM seems likely to require that video servers and disk data servers be connected directly to the "wire", because the overhead and latency associated with fetching data into a processor's memory before transmitting it can seem very large at the extremes of performance for which ATM is intended. These factors make it likely that although ATM will be usable in support of networks of high performance workstations, the technology will really take off in settings that exploit novel computing devices and new types of software architectures. These issues are already stimulating rexamination of some of the most basic operating system structures, and when we look at high speed communication in Chapter 8, many of the technologies considered turn out to have arisen as responses to this challenge.

Even layering the basic Internet protocols over ATM has turned out to be non-trivial. Although it is easy to fragment an IP packet into ATM cells, and the emulation mode mentioned above makes it straightforward to emulate IP networking over ATM networks, traditional IP software will drop an entire IP packet if any part of the data within it is corrupted. An ATM network that drops even a single cell per IP packet would thus seem to have 0% reliability, even though close to 99% of the data might be getting through reliably. This consideration has motivated ATM vendors to extend their hardware and software to understand IP and to arrange to drop *all* of an IP packet if even a single cell of that packet must be dropped, an example of a simple quality-of-service property. The result is that as the ATM network becomes loaded and starts to shed load, it does so by beginning to drop entire IP packets, hopefully with the result that other IP packets will get through unscathed. This leads us to the use of the packet type identifier bit: the idea is that in a burst of packets, the first packet can be identified by setting this bit to 0, and subsequent "subordinate" packets identified by setting it to 1. If the ATM must drop a cell, it can then drop all subsequent cells with the same VCI until one is encountered with the PTI bit set to 0, on the theory that all of these cells will be discarded in any case upon reception, because of the prior lost cell.

Looking to the future, it should not be long before IP drivers or special ATM firmware is developed that can buffer outgoing IP packets briefly in the controller of the sender and selectively solicit retransmission of just the missing cells if the receiving controller notices that data is missing. One can also imagine protocols whereby the sending ATM controller might compute and periodically transmit a parity cell containing the exclusive-or of all the prior cells for an IP packet; such a parity cell could then be used to reconstruct a single missing cell on the receiving side. Quality of service options for video data transmission using MPEG or JPEG may soon be introduced. Although these suggestions may sound complex and costly, keep in mind that the end-to-end latencies of a typical ATM network are so small (tens of microseconds) that it is entirely feasible to solicit the retransmission of a cell or two this even as the data for the remainder of the packet flows through the network. With effort, such steps should eventually lead to very reliable IP networking at ATM speeds. But the non-trivial aspects of this problem also point to the general difficulty of what, at first glance, might have seemed to be a completely obvious step to take. This is a pattern that we will often encounter throughout the remainder of the book!

2.7 Cluster and Parallel Architectures

Parallel supercomputer architectures, and their inexpensive and smaller-scale cousins, the cluster computer systems, have a natural correspondence to distributed systems. Increasingly, all three classes of systems are structured as collections of processors connected by high speed communications buses and with message passing as the basic abstraction. In the case of cluster computing systems, these communications buses are often based upon standard technologies such as fast ethernet or packet switching similar to that used in ATM. However, there are significant differences too, both in terms of scale and properties. These considerations make it necessary to treat cluster and parallel computing as a special case of distributed computing for which a number of optimizations are possible, and where special considerations are also needed in terms of the expected nature of application programs and their goals vis-a-vis the platform.

In particular, cluster and parallel computing systems often have built-in management networks that make it possible to detect failures extremely rapidly, and may have special purpose communication architectures with extremely regular and predictable performance and reliability properties. The ability to exploit these features in a software system creates the possibility that developers will be able to base their work on the general-purpose mechanisms used in general distributed computing systems, but to optimize them in ways that might greatly enhance their reliability or performance. For example, we will see that the inability to accurately sense failures is one of the hardest problems to overcome in distributed systems: certain types of network failures can create conditions indistinguishable from processor failure, and yet may heal themselves after a brief period of disruption, leaving the processor healthy and able to communicate again as if it had never been gone. Such problems do not arise in a cluster or parallel architecture, where accurate failure detection can be "wired" to available hardware features of the communications interconnect.

In this textbook, we will not consider cluster or parallel systems until Chapter 24, at which time we will ask how the special properties of such systems impacts the algorithmic and protocol issues that we consider in the previous chapters. Although there are some important software systems for parallel computing (PVM is the best known [GDBJ94]; MPI may eventually displace it [MPI96]), these are not particularly focused on reliability issues, and hence will be viewed as being beyond the scope of the current treatment.

2.8 Next steps

Few areas of technology development are as active as that involving basic communication technologies. The coming decade should see the introduction of powerful wireless communication technologies for the office, permitting workers to move computers and computing devices around a small space without the rewiring that contemporary devices often require. Bandwidth delivered to the end-user can be expected to continue to rise, although this will also require substantial changes in the software and hardware architecture of computing devices, which currently limits the achievable bandwidth for traditional network architectures. The emergence of exotic computing devices targetted to single applications should begin to displace general computing systems from some of these very demanding settings.

Looking to the broader internet, as speeds are rising, so too is congestion and contention for network resources. It is likely that virtual private networks, supported through a mixture of software and hardware, will soon become available to organizations able to pay for dedicated bandwidth and guaranteed latency. Such networks will need to combine strong security properties with new functionality, such as conferencing and multicast support. Over time, it can be expected that these data oriented networks will merge into the telecommunications "intelligent network" architecture, which provides support for voice, video and other forms of media, and mobility. All of these features will present the distributed application developer with new options, as well as new reliability challenges.

Reliability of the telecommunications architecture is already a concern, and that concern will only grow as the public begins to insist on stronger guarantees of security and privacy. Today, the rush to deploy new services and to demonstrate new communications capabilities has somewhat overshadowed robustness issues of these sorts. One consequence, however, has been a rash of dramatic failures and attacks on distributed applications and systems. Shortly after work on this book began, a telephone "phreak" was arrested for reprogramming the telecommunications switch in his home city in ways that gave him nearly complete control over the system, from the inside. He was found to have used his control to misappropriate funds through electronic transfers, and the case is apparently not an isolated event.

Meanwhile, new services such as “caller id” have turned out to have unexpected side-effects, such as permitting companies to build databases of the telephone numbers of the individuals who contact them. Not all of these individuals would have agreed to divulge their numbers.

Such events, understandably, have drawn considerable public attention and protest. As a consequence, they contribute towards a mindset in which the reliability implications of technology decisions are being given greater attention. Such the trend continue, it could eventually lead to wider use of technologies that promote distributed computing reliability, security and privacy over the coming decades.

2.9 Additional Reading

Addtional discussion of the topics covered in this chapter can be found in [Tan88, Com91, CS91, CS93,CDK94]. An outstanding treatment of ATM is [HHS94].

3. Basic Communication Services

3.1 Communications Standards

A communications standard is a collection of specifications governing the types of messages that can be sent in a system, the formats of message headers and trailers, the encoding rules for placing data into messages, and the rules governing format and use of source and destination addresses. In addition to this, a standard will normally specify a number of protocols that a provider should implement.

Examples of communications standards that are used widely, although not universally so, are:

- *The Internet Protocols*. These protocols originated in work done by the Defense Department Advanced Research Projects Agency, or DARPA, in the 1970's, and have gradually grown into a wider scale high performance network interconnecting millions of computers. The protocols employed in the internet include IP, the basic packet protocol, and UDP, TCP and IP-multicast, each of which is a higher level protocol layered over IP. With the emergence of the Web, the Internet has grown explosively during the mid 1990's.
- *The Open Systems Interconnect Protocols*. These protocols are similar to the internet protocol suite, but employ standards and conventions that originated with the ISO organization.
- *Proprietary standards*. Examples include the Systems Network Architecture, developed by IBM in the 1970's and widely used for mainframe networks during the 1980's, DECnet, developed at Digital Equipment but discontinued in favor of open solutions in the 1990's, Netware, Novell's widely popular networking technology for PC-based client-server networks, and Banyan's Vines system, also intended for PC's used in client-server applications.

During the 1990's, the emergence of "open systems", namely systems in which computers from different vendors and running independently developed software, has been an important trend. Open systems favor standards, but also must support current practice, since vendors otherwise find it hard to move their customer base to the standard. At the time of this writing, the trend clearly favors the Internet protocol suite as the most widely supported communications standard, with the Novell protocols strongly represented by force of market share. However, there protocol suites were designed long before the advent of modern high speed communications devices, and the commercial pressure to develop and deploy new kinds of distributed applications that exploit gigabit networks could force a rethinking of these standards. Indeed, even as the Internet has become a "de facto" standard, it has turned out to have serious scaling problems that may not be easy to fix in less than a few years (see Figure 3-1).

The remainder of this chapter focuses on the Internet protocol suite because this is the one used by the Web. Details of how the suite is implemented can be found in [Com91,CS91,CS93].

3.2 Addressing

The *addressing* tools in a distributed communication system provide unique identification for the source and destination of a message, together with ways of mapping from symbolic names for resources and services to the corresponding network address, and for obtaining the best route to use for sending messages.

Addressing is normally standardized as part of the general communication specifications for formatting data in messages, defining message headers, and communicating in a distributed environment.

Within the Internet, several address formats are available, organized into "classes" aimed at different styles of application. Each class of address is represented as a 32-bit number. Class A internet addresses have a 7-bit network identifier and a 24-bit host-identifier, and are reserved for very large networks. Class B addresses have 14 bits for the network identifier and 16 bits for the host-id, and class C has 21 bits of network identifier and 8 bits for the host-id. These last two classes are the most commonly used. Eventually, the space of internet addresses is likely to be exhausted, at which time a transition to an extended IP address is planned; the extended format increases the size of addresses to 64 bits but does so in a manner that provides backwards compatibility with existing 32-bit addresses. However, there are many hard problems raised by such a transition and industry is clearly hesitant to embark on what will be a hugely disruptive process.

Internet addresses have a standard ASCII representation, in which the bytes of the address are printed as signed decimal numbers in a standardized order. For example, this book was edited on host gunnlod.cs.cornell.edu, which has internet address 128.84.218.58. This is a class B internet address, with network address 42 and host-id 218.58. Network address 42 is assigned to Cornell University, as one of several class B addresses used by the University. The 218.xxx addresses designate a segment of Cornell's internal network, namely the ethernet to which my computer is attached. The number 58 was assigned within the Computer Science Department to identify my host on this ethernet segment.

A class D internet address is intended for special uses: IP multicasting. These addresses are allocated for use by applications that exploit IP multicast. Participants in the application join the multicast group, and the internet routing protocols automatically reconfigure themselves to route messages to all group members.

The string "gunnlod.cs.cornell.edu" is a symbolic name for IP address. The name consists of a machine name (gunnlod, an obscure hero of Norse mythology) and a suffix (cs.cornell.edu) designating the Computer Science Department at Cornell University, which is an educational institution in the United States. The suffix is registered with a distributed service called the domain name service, or DNS, which supports a simple protocol for mapping from string names to IP network addresses.

Here's the mechanism used by the DNS when it is asked to map my host name to the appropriate IP address for my machine. DNS has a top-level entry for "edu" but doesn't have an Internet address for this entry. However, DNS resolves cornell.edu to a gateway address for the Cornell domain, namely host 132.236.56.6. Finally, DNS has an even more precise address stored for cs.cornell.edu, namely 128.84.227.15 – a mail server and gateway machine in the Computer Science Department. All messages to machines in the Computer Science Department pass through this machine, which intercepts and discards messages to all but a select set of application programs.

DNS is itself structured as a hierarchical database of slowly changing information. It is hierarchical in the sense that DNS servers form a tree, with each level providing addresses of objects in the level below it, but also *caching* remote entries that are frequently used by local processes. Each DNS entry tells how to map some form of ascii hostname to the corresponding IP machine address or, in the case of commonly used services, how to find the service representative for a given host name.

Thus, DNS has an entry for the IP address of gunnlod.cs.cornell.edu (somewhere), and can track it down using its resolution protocol. If the name is used rapidly, the information may become cached local to the typical users and will resolve quickly; otherwise the protocol sends the request up the hierarchy to a level at which DNS knows how to resolve some part of the name, and then back down the hierarchy to a level that can fully resolve it. Similarly, DNS has a record telling how to find a mail transfer agent running the SMTP protocol for gunnlod.cs.cornell.edu: this may not be the same machine as gunnlod itself, but the resolution protocol is the same.

Internet Brownouts: Power Failures on the Data Superhighway?

Begining in late 1995, clear signs emerged that the Internet was beginning to overload. One reason is that the "root" servers for the DNS architecture are experiencing exponential growth in the load of DNS queries that require action by the top levels of the DNS hierarchy. A server that saw 10 queries per minute in 1993 was up to 250 queries per second in early 1995, and traffic was doubling every three months. Such problems point to fundamental aspects of the Internet that were based on assumptions of a fairly small and lightly loaded user population that repeatedly performed the same sorts of operations. In this small world, it makes sense to use a single hierarchical DNS structure with caching, because cache hits were possible for most data. In a network that suddenly has millions of users, and that will eventually support billions of users, such design considerations must be reconsidered: only a completely decentralized architecture can possibly scale to support a truely universal and world-wide service.

These problems have visible but subtle impact on the internet user: they typically cause connections to break, or alert boxes to appear on your Web browser warning you that the host possessing some resource is "unavailable." There is no obvious way to recognize that the problem is not one of local overload or congestion, but in fact is an overloaded DNS server or one that has crashed at a major Internet routing point. Unfortunately, such problems have become increasingly common: the Internet is starting to experience brownouts. Indeed, the Internet became largely unavailable because of failures of this nature for many hours during one crash in September of 1995, and this was hardly an unusual event. As the data superhighway becomes increasingly critical, such brownouts represent increasingly serious threats to reliability.

Conventional wisdom has it that the Internet does not follow the laws of physics, there is no limit to how big, fast and dense the Internet can become. Like the hardware itself, which seems outmoded almost before it reaches the market, we assume that the technology of the network is also speeding up in ways that outrace demand. But the reality of the situation is that the *software architecture* of the Internet is in some basic ways *not* scalable. Short of redesigning these protocols, the Internet won't keep up with growing demands. In some ways, it already can't.

Several problems are identified as the most serious culprits at the time of this writing. Number one in any ranking: the World Wide Web. The Web has taken over by storm, but it is inefficient in the way it fetches documents. In particular, as we will see in Chapter 10, the HTTP protocol often requires that large numbers of connections be created for typical document transfers, and these connections (even for a single HTML document) can involve contacting many separate servers. Potentially, each of these connection requests forces the root nodes of the DNS to respond to a query. With millions of users "surfing the network", DNS load is skyrocketing.

Bandwidth requirements are also growing exponentially. Unfortunately, the communication technology of the Internet is scaling more slowly than this. So overloaded connections, particularly near "hot sites", are a tremendous problem. A popular Web site may receive hundreds of requests per second, and each request must be handled *separately*. Even if the identical bits are being transmitted concurrently to hundreds of users, each user is sent its own, private copy. And this limitation means that as soon as a server becomes useful or interesting, it also becomes vastly overloaded. Yet ven though identical bits are being sent to hundreds of thousands of destinations, the protocols offer no obvious way to somehow multicast the desired data, in part because Web browsers explicitly make a separate connection for each object fetched, and only specify the object to send after the connection is in place. At the time of this writing, the best hope is that popular documents can be cached with increasing efficiency in "web proxies", but as we will see, doing so also introduces tricky issues of reliability and consistency. Meanwhile, the bandwidth issue is with us to stay.

Internet routing is another area that hasn't scaled very well. In the early days of the Internet, routing was a major area of research, and innovative protocols were used to route around areas of congestion. But these protocols were eventually found to be consuming too much bandwidth and imposing considerable overhead: early in the 1980's, 30% of Internet packets were associated with routing and load-balancing. A new generation of relatively static routing protocols was proposed at that time, and remain in use today. But the assumptions underlying these "new" reflected a network that, at the time, seemed "large" because it contained hundreds of nodes. A network of tens of millions or billions of nodes poses problems that could never have been anticipated in 1985. Now that we have such a network, even trying to understand its behavior is a major challenge. Meanwhile, when routers fail (for reasons of hardware, software, or simply because of overload), the network is tremendously disrupted.

The Internet Engineering Task Force (IETF), a governing body for the Internet and for Web protocols, is working on this problems. This organization sets the standards for the network and has the ability to legislate solutions. A variety of proposals are being considered: they include ways of optimizing the Web protocol called HTTP, and other protocol optimizations.

Some service providers are urging the introduction of mechanisms that would charge users based on the amount of data they transfer and thus discourage overuse (but one can immediately imagine the parents of an enthusiastic 12-year old forced to sell their house to pay the monthly network bill). There is considerable skepticism that such measures are practical. Bill Gates has suggested that in this new world, one can easily charge for the "size of the on-ramp" (the bandwidth of one's connection), but not for the amount of information a user transfers, and early evidence supports his perspective. In Gate's view, this is simply a challenge of the new Internet market.

There is no clear solution to the Internet bandwidth problem. However, as we will see in the textbook, there are some very powerful technologies that could begin to offer answers: coherent replication and caching being the most obvious remedy for many of the problems cited above. The financial motivations for being first to market with the solution are staggering, and history shows that this is a strong incentive indeed.

Figure 3-1: The data superhighway is experiencing serious growing pains. Growth in load has vastly exceeded the capacity of the protocols used in the Internet and World-Wide-Web. Issues of consistency, reliability, and availability in technologies such as the ones that support these applications are at the core of this textbook.

The internet address specifies a machine, but the identification of the specific application program that will process the message is also important. For this purpose, internet addresses contain a field called the port number, which is at present a 16-bit integer. A program that wishes to receive messages must bind itself to a port number on the machine to which the messages will be sent. A predefined list of port numbers is used by standard system services, and have values in the range 0-1023. Symbolic names have been assigned to many of these predefined port numbers, and a table mapping from names to port numbers is generally provided.

For example, messages sent to gunnlod.cs.cornell.edu that specify port 53 will be delivered to the DNS server running on machine gunnlod, or discarded if the server isn't running. Email is sent using a subsystem called SMTP, on port-number 25. Of course, if the appropriate service program isn't running, messages to a port will be silently discarded. Small port numbers are reserved for special services and are often "trusted", in the sense that it is assumed that only a legitimate SMTP agent will ever be connected to port 25 on a machine. This form of trust depends upon the operating system, which decides whether or not a program should be allowed to bind itself to a requested port.

Port numbers larger than 1024 are available for application programs. A program can request a specific port, or allow the operating system to pick one randomly. Given a port number, a program can register itself with the local network information service (NIS) program, giving a symbolic name for itself and the port number that it is listening on. Or, it can send its port number to some other program, for example by requesting a service and specifying the internet address and port number to which replies should be transmitted.

The randomness of port selection is, perhaps unexpectedly, an important source of security in many modern protocols. These protocols are poorly protected against intruders, who could attack the application if they were able to guess the port numbers being used. By virtue of picking port numbers randomly, the protocol assumes that the barrier against attack has been raised substantially, and hence that it need only protect against accidental delivery of packets from other sources: presumably an infrequent event, and one that is unlikely to involve packets that could be confused with the ones legitimately used by the protocol on the port. Later, however, we will see that such assumptions may not always be safe: modern network hackers may be able to steal port numbers out of IP packets; indeed, this has become a serious enough problem so that proposals for encrypting packet headers are being considered by the IETF.

Not all machines have identical byte orderings. For this reason, the internet protocol suite specifies a standard byte order that must be used to represent addresses and port numbers. On a host that does not use the same byte order as the standard requires, it is important to byte-swap these values before sending a message, or after receiving one. Many programming languages include communication libraries with standard functions for this purpose.

Finally, notice that the network services information specifies a protocol to use when communicating with a service – TCP, when communicating with the uucp service, UDP when communication with the tftp service (a file transfer program), and so forth. Some services support multiple options, such as the domain name service. As we discussed earlier, these names refer to protocols in the internet protocol suite.

3.3 Internet Protocols

This section presents the three major components of the internet protocol suite: the IP protocol, on which the others are based, and the TCP and UDP protocols, which are the ones normally employed by applications. We also discuss some recent extentions to the IP protocol layer in support of IP multicast protocols. There has been considerable discussion of security for the IP layer, but no single proposal has gained wide acceptance as of the time of this writing, and we will say very little about this ongoing work for reasons of brevity.

3.3.1 Internet Protocol: IP layer

The lowest layer of the internet protocol suite is a connectionless packet transport protocol called IP. IP is responsible for unreliable transport of variable size packets (but with a fixed maximum size, normally 1400 bytes), from the sender's machine to the destination machine. IP packets are required to conform to a fixed format consisting of a variable-length packet header, a variable-length body, and an optional trailer. The actual lengths of the header, body, and trailer are specified through length fields that are located at fixed offsets into the header. An application that makes direct use of IP is expected to format its packets according to this standard. However, direct use of IP is normally restricted because of security issues raised by the prospect of applications that might exploit such a feature to "mimic" some standard protocol, such as TCP, but do so in a non-standard way that could disrupt remote machines or create security loopholes.

Implementations of IP normally provide routing functionality, using either a static or dynamic routing architecture. The type of routing used will depend upon the complexity of the installation and its configuration of of the internet software, and is a topic beyond the scope of this textbook.

In 1995, IP was enhanced to provide a security architecture whereby packet payloads can be encrypted to prevent intruders from determining packet contents, and providing options for signatures or other authentication data in the packet trailer. Encryption of the packet header is also possible within this standard, although use of this feature is possible only if the routing layers and IP software implementation on all machines in the network agree upon the encryption method to use.

3.3.2 Transport Control Protocol: TCP

TCP is a name for the connection-oriented protocol within the internet protocol suite. TCP users start by making a TCP connection, which is done by having one program set itself up to *listen* for and *accept* incoming connections, while the other *connects* to it. A TCP connection guarantees that data will be delivered in the order sent, without loss or duplication, and will report an "end of file" if the process at either end exits or closes the channel. TCP connections are byte-stream oriented: although the sending program can send blocks of bytes, the underlying communication model views this communication as a continuous sequence of bytes. TCP is thus permitted to lose the boundary information between messages, so that what is logically a single message may be delivered in several smaller chunks, or delivered together with fragments of a previous or subsequent message (always preserving the byte ordering, however!). If very small messages are transmitted, TCP will delay them slightly to attempt to fill larger packets for efficient transmission; the user must disable this behavior if immediate transmission is desired.

Applications that involve concurrent use of a TCP connection must interlock against the possibility that multiple write operations will be done simultaneously on the same channel; if this occurs, then data from different writers can be interleaved when the channel becomes full.

3.3.3 User Datagram Protocol: UDP

UDP is a message or "datagram" oriented protocol. With this protocol, the application sends messages which are preserved in the form sent and delivered intact, or not at all, to the destination. No connection is needed, and there are no guarantees that the message will get through, or that messages will be delivered in any particular order, or even that duplicates will not arise. UDP imposes a size limit of 8k bytes on each message: an application needing to send a large message must fragment it into 8k chunks.

Internally, UDP will normally fragment a message into smaller pieces, which correspond to the maximum sizeof an IP packet, and matches closely with the maximum size packet that an ethernet can transmit in a single hardware packet. If a UDP packet exceeds the maximum IP packet size, the UDP packet is sent as a series of smaller IP packets. On reception, these are reassembled into a larger packet. If any fragment is lost, the UDP packet will eventually be discarded.

The reader may wonder why this sort of two-level fragmentation scheme is used – why not simply limit UDP to 1400 bytes, too? To understand this design, it is helpful to start with a measurement of the cost associated with a communication system call. On a typical operating system, such an operation has a minimum overhead of 20- to 50-thousand instructions, regardless of the size of the data object to be transmitted. The idea, then, is to avoid repeatedly traversing long code paths within the operating system. When an 8k-byte UDP packet is transmitted, the code to fragment it into smaller chunks executes "deep" within the operating system. This can save 10's of thousands of instructions.

One might also wonder why communication needs to be so expensive, in the first place. In fact, this is a very interesting and rather current topic, particularly in light of recent work that has reduced the cost of sending a message (on some platforms) to as little as 6 instructions. In this approach, which is called *Active Messages* [ECGS92, EBBV95], the operating system is kept completely off the message path, and if one is willing to paya slightly higher price, a similar benefit is possible even in a more standard communications architecture (see Section 8.3). Looking to the future, it is entirely plausible to believe that commercial operating systems products offering comparably low latency and high throughput will start to be available in the late 1990's. However, the average operating system will certainly not catch up with the leading edge approaches for many years. Thus, applications may have to continue to live with huge and in fact unecessary overheads for the time being.

3.3.4 Internet Packet Multicast Protocol: IP Multicast

IP multicast is a relatively recent addition to the Internet protocol suite [Der88,Der89,DC90]. With IP multicast, UDP or IP messages can be transmitted to groups of destinations, as opposed to a single point to point destination. The approach extends the multicast capabilities of the ethernet interface to work even in complex networks with routing and bridges between ethernet segments.

IP multicast is a session-oriented protocol: some work is required before communication can begin. The processes that will communicate must create an IP multicast address, which is a class-D Internet address containing a multicast identifier in the lower 28 bits. These processes must also agree upon a single port number that all will use for the communication session. As each process starts, it installs IP address into its local system, using system calls that place the IP multicast address on the ethernet interface(s) to which the machine is connected. The routing tables used by IP, discussed in more detail below, are also updated to ensure that IP multicast packets will be forwarded to each destination and network on which group members are found.

Once this setup has been done, an IP multicast is initiated by simply sending a UDP packet with the IP multicast group address and port number in it. As this packet reaches a machine which is included in the destination list, a copy is made and delivered to local applications receiving on the port. If several are bound to the same port on the same machine, a copy is made for each.

Like UDP, IP multicast is an unreliable protocol: packets can be lost, duplicated or delivered out of order, and not all members of a group will see the same pattern of loss and delivery. Thus, although one can build reliable communication protocols over IP multicast, the protocol itself is inherently unreliable.

When used through the UDP interface, a UDP multicast facility is similar to a UDP datagram facility, in that each packet can be as long as the maximum size of UDP transmissions, which is typically 8k. However, when sending an IP or UDP multicast, it is important to remember that the reliability observed may vary from destination to destination. One machine may receive a packet that others drop because of memory limitations or corruption caused by a weak signal on the communications medium, and the loss of even a single fragment of a large UDP message will cause the entire message to be dropped. Thus, one talks more commonly about IP multicast than UDP multicast, and it is uncommon for applications to send very large messages using the UDP interface. Any application that uses this transport protocol should carefully instrument loss rates, because the effective performance for small messages may actually be better than for large ones due to this limitation.

3.4 Routing

Routing is the method by which a communication system computes the path by which packets will travel from source to destination. A routed packet is said to take a series of *hops*, as it is passed from machine to machine. The algorithm used is generally as follows:

- An application program generates a packet, or a packet is read from a network interface.
- The packet destination is checked and, if it matches with any of the addresses that the machine accepts, delivered locally (one machine can have multiple addresses, a feature that is sometimes exploited in networks with dual hardware for increased fault-tolerance).
- The *hop count* of the message is incremented. If the message has a maximum hop count and would exceed it, the message is discarded. The hop count is also called the *time to live*, or TTL, in some protocols.
- For messages that do not have a local destination, or class-D multicast messages, the destination is used to search the routing table. Each entry specifies an address, or a pattern covering a range of addresses. An outgoing interface is computed for the message (a list of outgoing interfaces, if the message is a class-D multicast). For a point-to-point message, if there are multiple possible routes, the least costly route is employed. For this purpose, each route includes an estimated cost, in hops.
- The packet is transmitted on interfaces in this list, other than the one on which the packet was received.

A number of methods have been developed for maintaining routing tables. The most common approach is to use *static routing*. In this approach, the routing table is maintained by system administrators, and is never modified while the system is active.

Dynamic routing is a class of protocols by which machines can adjust their routing tables to benefit from load changes, route around congestion and broken links, reconfigure to exploit links that have recovered from failures. In the most common approaches, machines periodically distribute their routing tables to nearest neighbors, or periodically broadcast their routing tables within the network as a whole. For this latter case, a special address is used that causes the packet to be routed down every possible interface in the network; a hop-count limit prevents such a packet from bouncing endlessly.

The introduction of IP multicast has resulted in a new class of routers that are static for most purposes, but that maintain special dynamic routing policies for use when an IP multicast group spans several segments of a routed local area network. In very large settings, this *multicast routing daemon* can take advantage of the *multicast backbone* or *mbone* network to provide group communication or conferencing support to sets of participants working at physically remote locations. However, most use of IP multicast is limited to local area networks at the time of this writing, and wide-area multicast remains a somewhat speculative research topic.

3.5 End-to-end Argument

The reader may be curious about the following issue. The architecture described above permits packets to be lost at each hop in the communication subsystem. If a packet takes many hops, the probability of loss would seem likely to grow proportionately, causing the reliability of the network to drop linearly with the diameter of the network. There is an alternative approach in which error correction would be done hop by hop. Although packets could still be lost if an intermediate machine crashes, such an approach would have loss rates that are greatly reduced, at some small but fixed background cost (when we discuss the details of reliable communication protocols, we will see that the overhead need not be very high). Why, then, do most systems favor an approach that seems likely to be much less reliable?

In a classic paper, Jerry Saltzer and others took up this issue in 1984 [SRC84]. This paper compared "end to end" reliability protocols, which operate only between the source and destination of a message, with "hop by hop" reliable protocols. They argued that even if reliability of a routed network is improved by the use of hop-by-hop reliability protocols, it will still not be high enough to completely overcome packet loss. Packets can still be corrupted by noise on the lines, machines can crash, and dynamic routing changes can bounce a packet around until it is discarded. Moreover, they argue, the measured average loss rates for lightly to moderately loaded networks are extremely low. True, routing exposes a packet to repeated threats, but the overall reliability of a routed network will still be very high on the average, with worst case behavior dominated by events like routing table updates and crashes that hop-by-hop error correction would not overcome. From this the authors conclude that since hop-by-hop reliability methods increase complexity and reduce performance, and yet must still be duplicated by end-to-end reliability mechanisms, one might as well use a simpler and faster link-level communication protocol. This is the "end to end argument" and has emerged as one of the defining principles governing modern network design.

Saltzer's paper revolves around a specific example, involving a file transfer protocol. The paper makes the point that the analysis used is in many ways tied to the example and the actual reliability properties of the communication lines in question. Moreover, Saltzer's interest was specifically in reliability of the packet transport mechanism: failure rates and ordering. These points are important because many authors have come to cite the end-to-end argument in a much more expansive way, claiming that it is an absolute argument against putting any form of "property" or "guarantee" within the communication subsystem. Later, we will be discussing protocols that *need* to place properties and guarantees into subsystems, as a way of providing system-wide properties that would not otherwise be achievable. Thus, those who accept the "generalized" end-to-end argument would tend to oppose the use of these sorts of protocols on philisophical (one is tended to say "religious") grounds.

A more mature view is that the end-to-end argument is one of those situations where one should accept its point with a degree of skepticism. On the one hand, the end-to-end argument is clearly correct in situations where an analysis comparable to Saltzer's original one is possible. However, the end-to-end argument cannot be applied blindly: there are situations in which low level properties are beneficial and genuinely reduce complexity and cost in application software, and for these situations, an end-to-end approach might be inappropriate, leading to more complex applications that are error prone or, in a practical sense, impossible to construct.

For example, in a network with high link-level loss rates, or one that is at serious risk of running out of memory unless flow control is used link-to-link, an end-to-end approach may result in near-total packet loss, while a scheme that corrects packet loss and does flow control at the link level could yield acceptable performance. Thus, then, is a case in which Saltzer's analysis could be applied as he originally formulated it, but would lead to a different conclusion. When we look at the reliability protocols presented in the third part of this textbook, we will see that certain forms of consistent distributed behavior (such as is needed in a fault-tolerant coherent caching scheme) depend upon system-wide agreement that must be standardized and integrated with low-level failure reporting mechanisms. Omitting such a mechanism from the transport layer merely forces the application programmer to build it as part of the application; if the programming environment is intended to be general and extensible, this may mean that one makes the mechanism part of the environment or gives up on it entirely. Thus, when we look at distributed programming environments like the CORBA architecture, seen in Chapter 6, there is in fact a basic design choice to be made: either such a function is made part of the architecture, or by omitting it, no application can achieve this type of consistency in a general and interoperable way except with respect to other applications implemented by the same development team. These examples illustrate that, like many engineering arguments, the end-to-end approach is highly appropriate in certain situations, but not uniformly so.

3.6 O/S Architecture Issues, Buffering, Fragmentation

We have reviewed most stages of the communication architecture that interconnects a sending application to a receiving application. But what of the operating system software at the two ends?

The communications software of a typical operating system is modular, organized as a set of components that subdivide the tasks associated with implementing the protocol stack or stacks in use by application programs. One of these components is the *buffering* subsystem, which maintains a collection of kernel memory buffers that can be used to temporarily store incoming or outgoing messages. On most UNIX systems, these are called *mbufs*, and the total number available is a configuration parameter that should be set when the system is built. Other operating systems allocate buffers dynamically, competing with the disk I/O subsystem and other I/O subsystems for kernel memory. All operating systems share a key property, however: the amount of buffering space available is limited.

The TCP and UDP protocols are implemented as software modules that include interfaces up to the user, and down to the IP software layer. In a typical UNIX implementation, these protocols allocate some amount of kernel memory space for each open communication "socket", at the time the socket is created. TCP, for example, allocates an 8-kbyte buffer, and UDP allocates two 8k-byte buffers, one for transmission and one for reception (both can often be increased to 64kbytes). The message to be transmitted is copied into this buffer (in the case of TCP, this is done in chunks if necessary). Fragments are then generated by allocating successive memory chunks for use by IP, copying the data to be sent into them, prepending an IP header, and then passing them to the IP sending routine. Some operating systems avoid one or more of these copying steps, but this can increase code complexity, and copying is sufficiently fast that many operating systems simply copy the data for each message multiple times. Finally, IP identifies the network interface to use by searching the routing table and queues the fragments for transmission. As might be expected, incoming packets trace the reverse path.

An operating system can drop packets or messages for reasons unrelated to the hardware corruption or duplication. In particular, an application that tries to send data as rapidly as possible, or a machine that is presented with a high rate of incoming data packets, can exceed the amount of kernel memory that can safely be allocated to any single application. Should this happen, it is common for packets to be discarded until memory usage drops back below threshold. This can result in unexpected patterns of message loss.

For example, consider an application program that simply tests packet loss rates. One might expect that as the rate of transmission is gradually increased, from one packet per second to 10, then 100, then 1000 the overall probability that a packet loss will occur would remain fairly constant, hence packet loss will rise in direct proportion to the actual number of packets sent. Experiments that test this case, running over UDP, reveal quite a different pattern, illustrated in Figure 3-2; the left graph is for a sender and receiver on the same machine (the messages are never actually put on the wire in this case), and the right the case of a sender and receiver on identical machines connected by an ethernet.

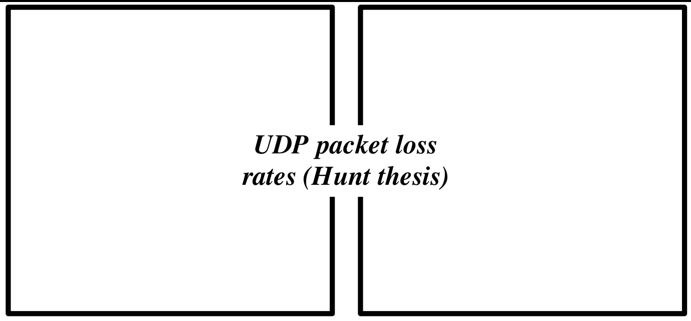


Figure 3-2: Packet loss rates for Unix over ethernet (the left graph is based on a purely local communication path, while the right one is from a distributed case using two computers connected by a 10-Mbit ethernet). This data is based on a study reported as part of a doctoral dissertation by Guerney Hunt.

As can be seen from the figure, the packet loss rate, as a percentage, is initially low and constant: zero for the local case, and small for the remote case. As the transmission rate rises, however, both rates rise. Presumably, this reflects the increased probability of memory threshold effects in the operating system. However, as the rate rises still further, behavior breaks down completely! For high rates of communication, one sees bursty behavior in which some groups of packets are delivered, and others are completely lost. Moreover, the aggregate throughput can be quite low in these overloaded cases, and the operating system often reports no errors at all the sender and destination – on the sending side, the loss occurs after UDP has accepted a packet, when it is unable to obtain memory for the IP fragments. On the receiving side, the loss occurs when UDP packets turn out to be missing fragments, or when the queue of incoming messages exceeds the limited capacity of the UDP input buffer.

The quantized scheduling algorithms used in multitasking operating systems like UNIX probably accounts for the bursty aspect of the loss behavior. UNIX tends to schedule processes for long periods, permitting the sender to send many packets during congestion periods, without allowing the receiver to run to clear its input queue in the local case, or giving the interface time to transmitted an accumulated backlog in the remote case. The effect is that once a loss starts to occur, many packets can be lost before the system recovers. Interestingly, packets can also be delivered out of order when tests of this sort are done, presumably reflecting some sort of stacking mechanisms deep within the operating system. Thus, the same measurements might yield different results on other versions of UNIX or other operating systems. However, with the exception of special purpose communication-oriented operating systems such as QNX (a real-time system for embedded applications), one would expect a “similar” result for most of the common platforms used in distributed settings today.

TCP behavior is much more reasonable for the same tests, but there are other types of tests for which TCP can behave poorly. For example, if one processes makes a great number of TCP connections to other processes, and then tries to transmit multicast messages on the resulting 1-many connections, the measured throughput drops worse than linearly, as a function of the number of connections, for most operating systems. Moreover, if groups of processes are created and TCP connections are opened between them, pairwise, performance is often found to be extremely variable – latency and throughput figures can vary wildly even for simple patterns of communications.

UDP or IP multicast gives the same behavior as UDP. However, the user ofmulticast should also keep in mind that many sources of packet loss can result in different patterns of reliability for different recievers. Thus, one destination of a multicast transmission may experience high loss rates even if many other destinations receive all messages with no losses at all. Problems such as this are potentially difficult to detect and are very hard to deal with in software.

3.7 Xpress Transfer Protocol

Although widely available, TCP, UDP and IP are also limited in the functionality they provide and their flexibility. This has motivated researchers to investigate new and more flexible protocol development architectures that can co-exist with TCP/IP but support varying qualities of transport service that can be matched closely to the special needs of demanding applications.

Prominent among such efforts is the Xpress Transfer Protocol (XTP), which is a toolkit of mechanisms that can be exploited by users to customize data transfer protocols operating in a point to point or multicast environment. All aspects of the the protocol are under control of the developer, who sets option bits during individual packet exchanges to support a highly customizable suite of possible comunication styles. References to this work include [SDW92,XTP95,DFW90].

XTP is a connection oriented protocol, but one in which the connection setup and closing protocols can be varied depending on the needs of the application. A connection is identified by a 64-bit key; 64-bit sequence numbers are used to identify bytes in transit. XTP does not define any addressing scheme of its own, but is normally combined with IP addressing. An XTP protocol is defined as an exchange of XTP messages. Using the XTP toolkit, a variety of options can be specified for each message transmitted; the effect is to support a range of possible "qualities of service" for each communication session. For example, an XTP protocol can be made to emulate UDP or TCP-style streams, to operate in an unreliable source to destination mode, with selective retransmission based on negative acknowledgements, or can even be asked to "go back" to a previous point in a transmission and to resume. Both rate-based and windowing flow control mechanisms are available for each connection, although one or both can be disabled if desired. The window size is configured by the user at the start of a connection, but can later be varied while the connection is in use, and a set of *traffic* parameters can be used to specify requirements such as the maximum size of data segments that can be transmitted in each packet, maximum or desired burst data rates, and so forth. Such parameters permit the development of general purpose transfer protocols that can be configured at runtime to match the properties of the hardware environment.

This flexibility is exploited in developing specialized transport protocols that may look like highly optimized version of the standard ones, but that can also provide very unusual properties. For example, one could develop a TCP-style of stream that will reliable provided that the packets sent arrive "on time", using a user-specific notion of time, but that drops packets if they timeout. Similarly, one can develop protocols with out-of-band or other forms of priority-based services.

At the time of this writing, XTP was gaining significant support from industry leaders whose future product lines potentially require flexibility from the network. Video servers, for example, are poorly matched to the communication properties of TCP connections, hence companies that are investing heavily in "video on demand" face the potential problem of having products that work well in the laboratory but not in the field, because the protocol architecture connecting customer applications to the server is inappropriate. Such companies are interested in developing proprietary data transport protocols that would essentially extend their server products into the network itself, permitting fine-grained control over the communication properties of the environment in which their servers operate, and overcoming limitations of the more traditional but less flexible transport protocols.

In Chapters 13 through 16 we will be studying special purpose protocols designed for settings in which reliability requires data replication or specialized performance guarantees. Although we will generally present these protocols in the context of streams, UDP, or IP-multicast, it is likely that the future will bring a considerably wider set of transport options that can be exploited in applications with these sorts of requirements.

There is, however, a downside associated with the use of customized protocols based on technologies such as XTP: they can create difficult management and monitoring problems, which will often go well beyond those seen in standard environments where tools can be developed to monitor a network and to display, in a well organized manner, the status of the network and applications. Such tools benefit from being able to intercept network traffic and to associate the message sent with the applications sending them. To the degree that technologies such as XTP lead to extremely specialized patterns of communication that work well for individual applications, they may also reduce this desirable form of regularity and hence impose obstacles to system control and management.

Broadly, one finds a tension within the networking community today. On the one side are developers convinced that their special-purpose protocols are necessary if a diversity of communications products and technologies are to be feasible over networks such as the Internet. In some sense this community generalizes to also include the community that develops special purpose reliability protocols and that may need to place "properties" within the network to support those protocols. On the other stand the system administrators and managers, whose lives are already difficult, and who are extremely resistant to technologies that might make this problem worse. Sympathizing with them are the performance experts of the operating systems communications community: this group favors an end-to-end approach because it greatly simplifies their task, and hence tends to oppose technologies such as XTP because they result in irregular behaviors that are harder to optimize in the general case. For these researchers, knowing more about the low level requirements (and keeping them as simple as possible) makes it more practical to optimize the corresponding code paths for extremely high performance and low latency.

From a reliability perspective, one must sympathize with both points of view: this textbook will look at problems for which reliability requires high performance or other guarantees, and problems for which reliability implies the need to monitor, control, or manage a complex environment. If there is a single factor that prevents a protocol suite such as XTP from "sweeping the industry", it seems likely to be this. More likely, however, is an increasingly diverse collection of low-level protocols, creating ongoing challenges for the community that must administer and monitor the networks in which those protocols are used.

3.8 Next Steps

There is a sense in which it is not surprising that problems such as the performance anomalies cited in the previous sections would be common on modern operating systems, because the communication subsystems have rarely been designed or tuned to guarantee good performance for communication patterns such as were used to produce Figure 3-2. As will be seen in the next few chapters, the most common communication patterns are very regular ones that would not trigger the sorts of pathological behaviors caused by memory resource limits and stressful communication loads.

However, given a situation in which most systems must in fact operate over protocols such as TCP and UDP, these behaviors do create a context that should concern students of distributed systems reliability. They suggest that even systems that behave well most of the time may break down catastrophically because of something as simple as a slight increase in load. Software designed on the assumption that message loss rates are low may, for reasons completely beyond the control of the developer, encounter loss rates that are extremely high. All of this can lead the researcher to question the appropriateness of modern operating systems for reliable distributed applications. Alternative operating systems architectures that offer more controlled degradation in the presence of excess load represent a potentially important direction for investigation and discussion.

3.9 Additional Reading

On the Internet protocols: [Tan88, Com91, CS91, CS93, CDK94]. Performance issues for TCP and UDP: [Com91, CS91, CS93, ALFxx, KP93, PP93, BMP94, Hun95]. IP Multicast: [FWB85, Dee88, Dee89, DC90, Hun95]. Active Messages: [ECGS92, EBBV95]. End-to-end argument: [SRC84]. Xpress Transfer Protocol: [SDW92, XTP95, DFW90].

4. RPC and the Client-Server Model

The emergence of "real" distributed computing systems is often identified with the *client-server* paradigm, and a protocol called *remote procedure call* which is normally used in support of this paradigm. The basic idea of a client-server system architecture involves a partitioning of the software in an application into a set of *services*, which provide a set of operations to their users, and *client programs*, which implement applications and issue requests to services as needed to carry out the purposes of the application. In this model, the application processes do not cooperate directly with one another, but instead share data and coordinate actions by interacting with a common set of servers, and by the order in which the application programs are executed.

There are a great number of client-server system structures in a typical distributed computing environment. Some examples of servers include the following:

- *File servers*. These are programs (or, increasingly, combinations of special purpose hardware and software) that manage disk storage units on which files systems reside. The operating system on a workstation that accesses a file server acts as the "client", thus creating a two-level hierarchy: the application processes talk to their local operating system. The operating system on the client workstation functions as a single client of the file server, with which it communicates over the network.
- *Database servers*. The client-server model operates in a similar way for database servers, except that it is rare for the operating system to function as an intermediary in the manner that it does for a file server. In a database application, there is usually a library of procedure calls with which the application accesses the database, and this library plays the role of the client in a client-server communications protocol to the database server.
- *Network name servers*. Name servers implement some form of map from a symbolic name or service description to a corresponding value, such as an IP addresses and port number for a process capable of providing a desired service.
- *Network time servers*. These are processes that control and adjust the clocks in a network, so that clocks on different machines give consistent time values (values with limited divergence from one-another. The server for a clock is the local interface by which an application obtains the time. The clock service, in contrast, is the collection of clock servers and the protocols they use to maintain clock synchronization.
- *Network security servers*. Most commonly, these consist of a type of directory in which public keys are stored, as well as a key generation service for creating new secure communication channels.
- *Network mail and bulletin board servers*. These are programs for sending, receiving and forwarding email and messages to electronic bulletin boards. A typical client of such a server would be a program that sends an electronic mail message, or that displays new messages to a human who is using a news-reader interface.
- *WWW servers*. As we learned in the introduction, the World-Wide-Web is a large-scale distributed document management system developed at CERN in the early 1990's and subsequently commercialized. The Web stores hypertext documents, images, digital movies and other information on *web servers*, using standardized formats that can be displayed through various browsing programs. These systems present point-and-click interfaces to hypertext documents, retrieving documents using web document locators from web servers, and then displaying them in a type-specific manner. A web server is thus a type of enhanced file server on which the Web access protocols are supported.

In most distributed systems, services can be instantiated multiple times. For example, a distributed system can contain multiple file servers, or multiple name servers. We normally use the term *service* to denote a set of servers. Thus, the *network file system service* consists of the network file servers for a system, and the *network information service* is a set of servers, provided on UNIX systems, that map symbolic names to ascii strings encoding "values" or addresses. An important question to ask about a distributed system concerns the binding of applications to servers.

We say that a *binding* occurs when a process that needs to talk to a distributed service becomes associated with a specific server that will perform requests on its behalf. Various binding policies exist, differing in how the server is selected. For an NFS distributed file system, binding is a function of the file pathname being accessed – in this file system protocol, the servers all handle different files, so that the pathname maps to a particular server that owns that file. A program using the UNIX network information server normally starts by looking for a server on its own machine. If none is found, the program broadcasts a request and binds to the first NIS that responds, the idea being that this NIS representative is probably the least loaded and will give the best response times. (On the negative side, this approach can reduce reliability: not only will a program now be dependent on availability of its file servers, but it may be dependent on an additional process on some other machine, namely the NIS server to which it became bound). The CICS database system is well known for its explicit load-balancing policies, which bind a client program to a server in a way that attempts to give uniform responsiveness to all clients.

Algorithms for binding, and for dynamically rebinding, represent an important topic to which we will return in Chapter 17, once we have the tools at our disposal to solve the problem in a concise way.

A distributed service may or may not employ *data replication*, whereby a service maintain more than one copy of a single data item to permit local access at multiple locations, or to increase availability during periods when some server processes may have crashed. For example, most network file services can support multiple file servers, but do not replicate any single file onto multiple servers. In this approach, each file server handles a partition of the overall file system, and the partitions are disjoint from one another. A file can be replicated, but only by giving each replica a different name, placing each replica on an appropriate file server, and implementing hand-crafted protocols for keeping the replicas coordinated. Replication, then, is an important issue in designing complex or highly available distributed servers.

Caching is a closely related issue. We say that a process has *cached* a data item if it maintains a copy of that data item locally, for quick access if the item is required again. Caching is widely used in file systems and name services, and permits these types of systems to benefit from locality of reference. A *cache hit* is said to occur when a request can be satisfied out of cache, avoiding the expenditure of resources needed to satisfy the request from the *primary store* or *primary service*. The Web uses document caching heavily, as a way to speed up access to frequently used documents.

Caching is similar to replication, except that cached copies of a data item are in some ways second-class citizens. Generally, caching mechanisms recognize the possibility that the cache contents may be stale, and include a policy for validating a cached data item before using it. Many caching schemes go further, and include explicit mechanisms by which the primary store or service can invalidate cached data items that are being updated, or refresh them explicitly. In situations where a cache is actively refreshed, caching may be identical to replication – a special term for a particular style of replication.

However, "generally" does not imply that this is always the case. The Web, for example, has a cache validation mechanism but does not actually require that web proxies validate cached documents before providing them to the client; the reasoning is presumably that even if the document were validated at the time of access, nothing prevents it from changing immediately afterwards and hence being stale by the time the client display it, in any case. Thus a periodic refreshing scheme in which cached documents are refreshed every half hour or so is in many ways equally reasonable. A caching policy is said to be *coherent* if it guarantees that cached data is indistinguish to the user from the primary copy. The web caching scheme is thus one that does not guarantee coherency of cached documents.

4.1 RPC Protocols and Concepts

The most common communication protocol for communication between the clients of a service and the service itself is *remote procedure call*. The basic idea of an RPC originated in work by Nelson in the early 1980's [BN84]. Nelson worked in a group at Xerox Parc that was developing programming languages and environments to simplify distributed computing. At that time, software for supporting file transfer, remote login, electronic mail, and electronic bulletin boards had become common. Parc researchers, however, and ambitious ideas for developing other sorts of distributed computing applications, with the consequence that many researchers found themselves working with the lowest level message passing primitives in the Parc distributed operating system, which was called Cedar.

Much like a more modern operating system, message communication in Cedar supported three communication models:

- Unreliable datagram communication, in which messages could be lost with some (hopefully low) probability;
- Broadcast communication, also through an unreliable datagram interface.
- Stream communication, in which an initial connection was required, after which data could be transferred reliably.

Programmers found these interfaces hard to work with. Any time a program *p* needed to communicate with a program *s*, it was necessary for *p* to determine the network address of *s*, encode its requests in a way that *s* would understand, send off the request, and await a reply. Programmers soon discovered that certain basic operations needed to be performed in almost any network application, and that each developer was developing his or her own solutions to these standard problems. For example, some programs used broadcasts to find a service with which they needed to communicate, others stored the network address of services in files or hard-coded them into the application, and still others supported directory programs with which services could register themselves, and supporting queries from other programs at runtime. Not only was this situation confusing, it turned out to be hard to maintain the early versions of Parc software: a small change to a service might "break" all sorts of applications that used it, so that it became hard to introduce new versions of services and applications.

Surveying this situation, Bruce Nelson started by asking what sorts of interactions programs really needed in distributed settings. He concluded that the problem was really no different from function or procedure call in a non-distributed program that uses a presupplied library. That is, most distributed computing applications would prefer to treat other programs with which they interact much as they treat presupplied libraries, with well known, documented, procedural interfaces. Talking to another program would then be as simple as invoking one of its procedures – a *remote procedure call* (RPC for short).

The idea of remote procedure call is compelling. If distributed computing can be transparently mapped to a non-distributed computing model, all the technology of non-distributed programming could be brought to bear on the problem. In some sense, we would already know how to design and reason about distributed programs, how to show them to be correct, how to test and maintain and upgrade them, and all sorts of preexisting software tools and utilities would be readily applicable to the problem.

Unfortunately, the details of supporting remote procedure call turn out to be non-trivial, and some aspects result in "visible" differences between remote and local procedure invocations. Although this wasn't evident in the 1980's when RPC really took hold, the subsequent ten or fifteen years saw considerable theoretical activity in distributed computing, out of which ultimately emerged a deep understanding of how certain limitations on distributed computing are reflected in the *semantics*, or properties, of a remote procedure call. In some ways, this theoretical work finally lead to a major breakthrough in the late 1980's and early 1990's, when researchers learned how to create distributed computing systems in which the semantics of RPC are precisely the same as for local procedure call (LPC). In Part III of this text, we will study the results and necessary technology underlying such a solution, and will see how to apply it to RPC. We will also see, however, that such approaches involve subtle tradeoffs between semantics of the RPC and performance that can be achieved, and that the faster solutions also weaken semantics in fundamental ways. Such considerations ultimately lead to the insight that RPC cannot be transparent, however much we might wish that this was not the case.

Making matters worse, during the same period of time a huge engineering push behind RPC elevated it to the status of a standard – and this occurred *before* it was understand how RPC could be made to accurately mimic LPC. The result of this is that the standards for building RPC-based computing environments (and to a large extent, the standards for object-based computing that followed RPC in the early 1990's) embody a non-transparent and unreliable RPC model, and that this design decision is often fundamental to the architecture in ways that the developers who formulated these architectures probably did not appreciate. In the next chapter, when we study stream-based communication, we will see that the same sort of premature standardization affected the standard streams technology, which as a result also suffer from serious limitations that could have been avoided had the problem simply been better understood at the time the standards were developed.

In the remainder of this chapter, we will focus on standard implementations of RPC. We will look at the basic steps by which an program RPC is coded in a program, how that program is translated at compile time, and how it becomes bound to a service when it is executed. Then, we will study the encoding of data into messages and the protocols used for service invocation and to collect replies. Finally, we will try to pin down a semantics for RPC: a set of statements that can be made about the guarantees of this protocol, and that can be compared with the guarantees of LPC.

We do not, however, give detailed examples of the major RPC programming environments: DCE and ONC. These technologies, which emerged in the mid 1980's, represented proposals to standardize distributed computing by introducing architectures within which the major components of a dtsributed computing system would have well-specified interfaces and behaviors, and within which application programs could interoperate using RPC by virtue of employing standard RPC interfaces. DCE, in particular, has become relatively standard, and is available on many platforms today [DCE94]. However, in the mid-1990's, a new generation of RPC-oriented technology emerged through the Object Management Group, which set out to standardize object-oriented computing. In a short period of time, the CORBA [OMG91] technologies defined by OMG swept past the RPC technologies, and for a text such as the present one, it now makes more sense to focus on CORBA, which we discuss in Chapter 6. CORBA has not so much changed the basic issues, as it has broadened the subject of discourse by covering more kinds of system services than did previous RPC systems. Moreover, many CORBA systems are implemented as a layer over DCE or ONC. Thus, although RPC environments are important, they are more and more hidden from typical programmers and hence there is limited value in seeing examples of how one would program applications using them directly.

Many industry analysis talk about CORBA implemented over DCE, meaning that they like the service definitions and object orientation of CORBA, and that it makes sense to assume that these are build using the service implementations standardized in DCE. In practice, however, CORBA makes as much sense on a DCE platform as on a non-DCE platform, hence it would be an exaggeration to claim that CORBA on DCE is a *de-facto* standard today, as one sometimes reads in the popular press.

The use of RPC leads to interesting problems of reliability and fault-handling. As we will see, it is not hard to make RPC work if most of the system is working well. When a system malfunctions, however, RPC can fail in ways that leave the user with no information at all about what has occurred, and with no apparent strategy for recovering from the situation.There is nothing new about the situations we will be studying – indeed, for many years, it was simply assumed that RPC was subject to intrinsic limitations, and that there being no obvious way to improve on the situation, there was no reason that RPC shouldn't reflect these limitations in its semantic model. As we advance through the book, however, and it becomes clear that there *are* realistic alternatives that might be considered, this point of view becomes increasingly open to question.

Indeed, it may now be time to develop a new set of standards for distributed computing. The existing standards are flawed, and the failure of the standards community to repair these flaws has erected an enormous barrier to the development of reliable distributed computing systems. In a technical sense, these flaws are not tremendously hard to overcome – although the solutions would require some reengineering of communication support for RPC in modern operating systems. In a practical sense, however, one wonders if it will take a "Tacoma Narrows" event to create real industry interest in taking such steps.

One could build an RPC environment that would have few, if any, user-visible incompatibilities from a more fundamentally rigorous approach. The issue then is one of education – the communities that control the standards need to understand the issue better, and need to understand the reasons that this particular issue represents such a huge barrier to progress in distributed computing. And, the community needs to recognize that the opportunity vastly outweighs the reengineering costs that would be required to seize it. With this goal in mind, let's take a close look at RPC.

4.2 Writing an RPC-based Client or Server Program

The programmer of an RPC-based application employs what is called a *stub generation* tool. Such a tool is somewhat like a macro preprocessor: it transforms the user's original program into a modified version, which can be linked to an RPC runtime library.

From the point of view of the programmer, the server or client program looks much like any other program. Normally, the program will *import* or *export* a set of interface definitions, covering the remote procedures that will be obtained from remote servers or offered to remote clients, respectively. A server program will also have a "name" and a "version", which are used to connect the client to the server. Once coded, the program is compiled in two stages: first the stub generator is used to map the original program into a standard program with added code to carry out the RPC, and then the standard program is linked to the RPC runtime library for execution.

RPC-based application or server programs are coded in a programming style very similar to a non-distributed program written in C for UNIX: there is no explicit use of message passing. However, there is an important aspect of RPC programming that differs from programming with local procedure calls: the separation of the service interface definition, or IDL[3], from the code that implements it. In an RPC application, a service is considered to have two parts. The interface definition specifies the way that the service can be located (its name), the data types used in issuing requests to it, and the procedure calls that it supports. A *version number* is included to provide for evolution of the service over time – the idea being that if a client is developed to use version 1.1 of a service, there should be a way to check for compatibility if it turns out that version 1.0 or 2.3 is running when the client actually gets executed.

The basic actions of the RPC library were described earlier. In the case of a server program, the library is responsible for registering the program with the RPC directory service program, which is normally provided as part of the RPC runtime environment. An RPC client program will automatically perform the tasks needed to connect query the directory to find this server and to connect to it, creating a client-server binding. For each of the server operations it invokes, code will be executed to *marshall* a representation of the invocation into a message – that is, information about the way that the procedure was called and values of the parameters that were passed. Code is included to send this message to the service, and to collect a reply; on the server side, the stub generator creates code to read in such a message, invoke the appropriate procedure with the arguments used by the remote caller, and to marshall the results for transmission back to the caller. Issues such as user-id handling, security and privacy, and handling of exceptions are often packaged as part of a solution. Finally, back on the caller side, the returning message will be demarshalled and the result made to look like the result of a local procedure.

Although much of this mechanism is automatic and hidden from the programmer, RPC programming differs from LPC programming in many ways. Most noticeable is that most RPC packages limit the types of arguments that can be passed to a remote server, and some also limit the size (in bytes) of the argument information. For example, suppose that a local procedure is written to search a list, and an LPC is performed to invoke this procedure, passing a pointer to the head of the list as its argument. One can ask whether this should work in an RPC environment – and if so, how it can be supported. If a pointer to the head of the list is actually delivered to a remote program, that pointer will not make sense in the remote address space where the operation will execute. So, it would be natural to propose that the pointer be dereferenced, by copying the head of the list into the message. Remotely, a pointer to the copy can be provided to the procedure. Clearly, however, this will only work if one chases *all* the pointers in question – a problem because many programs that use pointers have some representation for an uninitialized pointer, and the RPC stub generator may not know about this.

For example, in building a balanced tree, it is common to allocate nodes dynamically as items are inserted. A node that has no descendents would still have left and right pointer fields, but these would be initialized to *nil* and the procedure to search nodes would check for the *nil* case before dereferencing these pointers. Were an RPC marshalling procedure to automatically make a copy of a structure to send to the remote server, it would need to realize that for this particular structure, a pointer value of *nil* has a special meaning and should not be "chased".

---

[3] It is common to call the interface to a program its "IDL", although IDL actually is a short-hand for Interface Definition Language, which is the language used to write down the description of such an interface. Historically, this seems to represent a small degree of resistance to the overuse of acronyms by the distributed systems standardization community. Unfortunately, the resistance seems to have been short-lived: CORBA introduces at least a dozen new 3-letter acronyms, "ATM" has swept the networking community, and 4- and 5-letter acronyms (as the available 3-letter combinations are used up) seems certain to follow!

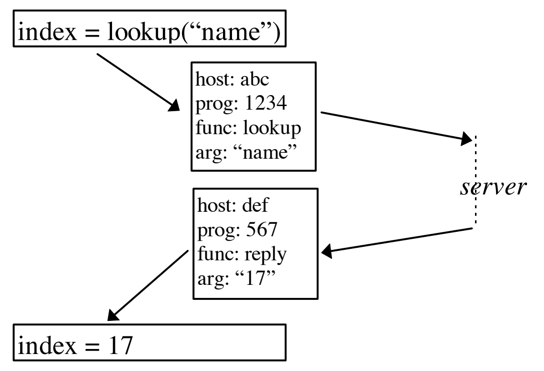


Figure 4-1: Remote procedure call involves creating a message that can be sent to the remote server, which unpacks it, performs the operation, and sends back a message encoding the result.

The RPC programmer sees issues such as these as a set of restrictions. Depending on the RPC package used, different approaches may be used to attack them. In many packages, pointers are simply not legal as arguments to remote procedures. In others, the user can control a copying mechanism to some degree, and in still fancier systems, the user must provide general purpose structure traversal procedures that will be used by the RPC package to marshall arguments. Further complications can arise if a remote procedure may modify some of its arguments. Again, the degree to which this is supported at all, and the degree to which the programmer must get involved, vary from package to package.

Perhaps ironically, RPC programmers tend to complain about this aspect of RPC no matter how it is handled. If a system is highly restrictive, the programmer finds that remote procedure invocation is annoying because one is constantly forced to work around the limitations of the invocation package. For example, if an RPC package imposes a size limit on the arguments to a procedure, an application that works perfectly well in most situations may suddenly fail because some dynamically defined object has grown too large to be accepted as an RPC parameter. Suddenly, what was a single RPC becomes a multi-RPC protocol for passing the large object in chunks, and a perfectly satisfied programmer has developed distinct second thoughts about the transparency of RPC. At the other extreme are programming languages and RPC packages in which RPC is extremely transparent. These, however, often incur high overheads to copy information in and out, and the programmer is likely to be very aware of these because of their cost implications. For example, a loop that repeatedly invokes a procedure with one parameter changing and others (including a pointer to some large object) may be quite inexpensive to invoke in the local case. But if the large object will be copied to a remote program on every invocation, the same loop may cost a fortune when coded as part of a distributed client-server application, forcing the program to be redesigned to somehow pass the object to the remote server prior to the computational loop. These sorts of issues, then, make programming with RPC quite different from programming with LPC.

RPC also introduces error cases that are not seen in LPC, and the programmer needs to deal with these. An LPC would never fail with a "binding error", or a "version mismatch" or a "timeout." In the case of RPC, all of these are possibilities – a binding error would arise if the server is not running when the client is started. A version mismatch might occur if a client was compiled against version 1 of a server, but the server has now been upgraded to version 2. A timeout could result from a server crash, or a network problem, or even a problem on the client's computer. Many RPC applications would view these sorts of problems as unrecoverable errors, but fault-tolerant systems will often have alternative sources for critical services and will need to fail-over from a primary server to a backup. The code to do this is potentially complex, and in most RPC environments, must be implemented by the application developer on a case-by-case basis.

4.3 The RPC Binding Problem

The *binding* problem arises when an RPC client program needs to determine the network address of a server capable of providing some service it requires. Binding can be approached from many perspectives, but the issue is simplified if issues associated with the *name service* used are treated separately, as we do here.

Disregarding its interactions with the name service, a binding service is primarily a protocol by which the RPC system verifies compatibility between the client and server and establishes any connections needed for communication.

The compatibility problem is important in systems that will operate over long periods of time, during which maintenance and the development of new versions of system components will inevitably occur. Suppose that a client program *c* was developed and tested using server *s*, but that we now wish to install a new version of *s*, *c*, or both. Upgrades such as these create a substantial risk that some old copy of *c* will find itself talking to a new copy of *s*, or vice versa. For example, in a network of workstations it may be necessary to reload *c* onto the workstations one by one, and if some machines are down when the reload occurs, an old copy of *c* could remain on its disk. Unless *c* is upgraded as soon as the machine is rebooted – and this may or may not occur, depending on how the system is administered – one would find an old *c* talking to an upgraded *s*. It is easy to identify other situations in which problems such as this could arise.

It would be desirable to be able to assume that all possible "versions" of *s* and *c* could somehow communicate with all other versions, but this is not often the case. Indeed, it is not necessarily even desirable. Accordingly, most RPC environments support a concept of *version number* which is associated with the server IDL. When a client program is compiled, the server IDL version is noted in software. This permits the inclusion of the client's version of the server interface directly in the call to the server. When the match is not exact, the server could reject the request as being incompatible, perform some operation to map the old-format request to a new-format request, or even preserve multiple copies of its functionality, running the version matched to the caller.

Connection establishment is a relatively mechanical stage of binding. Depending on the type of client-server communication protocol that will be used, messages may be transmitted using unreliable datagrams or over reliable communication streams such as X.25 or TCP. Unreliable datagram connections normally do not require any initial setup, but stream connections typically involve some form of open or initialization operation. Having identified the server to which a request will be issued, the binding mechanism would normally perform this open operation.

The binding mechanism is sometimes used to solve two additional problems. The first of these is called the "factory" problem, and involves starting a server when a service has no currently operational server. In this approach, the first phase of binding looks up the address of the server and learns that the server is not currently operational (or, in the connection phase, a connection error is detected and from this the binder deduces that the server has failed). The binder then issues a request to a *factory* in which the system designer has stored instructions for starting a server up when needed. After a suitable pause, the binder cycles back through its first phase, which presumably succeeds.

The second additional problem arises in the converse situation, when the binder discovers multiple servers that could potentially handle this client. The best policy to use in such situations depends very much on the application. For some systems, a binder should always pick a server on the same machine as the client, if possible, and should otherwise pick randomly. Other systems require some form of load-balancing, while still others may implement an *affinity* policy under which a certain server might be especially well suited to handling a particular client for reasons such as the data it has cached in memory, or the type of requests the client is expected to issue once binding has been completed.

Binding is a relatively expensive operation. For example, in the DCE RPC environment, binding can be more than 10 times as costly as RPC. However, since binding only occurs once for each client-server pair, this high cost is not viewed as a major problem in typical distributed computing systems.

4.4 Marshalling and Data Types

The purpose of a data marshalling mechanism is to represent the caller's arguments in a way that can be efficiently interpreted by a server program. In the most general cases, this mechanism deals with the possibility that the computer on which the client is running uses a different data representation than the computer on which the server is running.

Mashalling has been treated at varying levels of generality, and in fact there exists a standard, ASN.1, for *self-describing data objects* in which a specific representation is recommended. In addition to ASN.1, several major vendors have adopted data representations of their own, such as SUN Microsystem's External Data Representation (XDR) format, which is used in the widely popular Network File System (NFS) protocol.

The basic issues that arise in a data marshalling mechanism, then, are these. First, integer representations vary for the most common CPU chips. On some chips the most significant byte of an integer is also the low byte of the first word in memory, while on others the most significant byte is stored in the high byte of the last word of the integer. These are called little-endian and big-endian representations. At one point in the 1980's, computers with other representations – other byte permutations – were on the market, but at the time of this writing the author is not aware of any other surviving formats.

A second representation issue concerns data alignment. Some computers require that data be aligned on 32-bit or even 64-bit boundaries, while others may have weaker alignment rules, for example by supporting data alignment on 16-bit boundaries. Unfortunately, such issues are extremely common. Compilers know about these rules, hence the programmer is typically unaware of them. However, when a message arrives from a remote machine that may be using some other alignment rule, the issues becomes an important one. An attempt to fetch data directly from a message without attention to this issue could result in some form of machine fault, or could result in retrieval of garbage. Thus, the data representation used in messages must encode sufficient information to permit the destination computer to find the start of object in the message, or the sender and destination must agree in advance on a packed representation that will be used for messages "on the wire" even if the sender and destination themselves share the same rules and differ from the standard. Needless to say, this is a topic capable of generating endless and fascinating debate among computer vendors whose machines use different alignment or data representations.

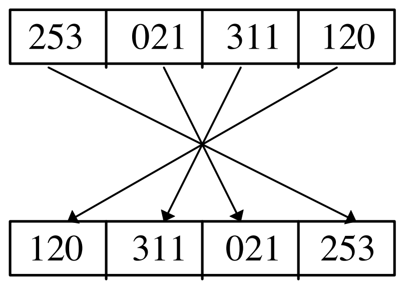


Figure 4-2: The same number (here, a 32-bit integer) may be represented very differently on different computer architectures. One role of the marshalling an demarshalling process is to modify data representations (here, by permuting the bytes) so that values can be interpreted correctly upon reception.

A third issue arises from the existence of multiple floating point representations. Although there is an IEEE standard floating point representation, which has become widely accepted, some computer vendors use non-standard representations for which conversion would be required, and even within computers using the standard, byte ordering issues can still arise.

A forth issue concerns pointers. When transmitting a complex structure in which there are pointers, the marshalling mechanism needs to either signal that the user has requested something illegal, or to somehow represent these pointers in a way that will permit the receiving computer to "fix" them upon reception of the request. This is especially tricky in languages like LISP, which require pointers and hence cannot easily legislate against them in RPC situations. On the other hand, passing pointers raises additional problems: should the pointed-to object be included in the message, transferred only upon use (a "lazy" scheme), or handled in some other way?

Finally, a marshalling mechanism may need to deal with incompatibilities in the basic data types available on computers. For example, a pair of computers supporting 64-bit integers in hardware may need to exchange messages containing 64-bit integer data. The marshalling scheme should therefore be able to represent such integers. On the other hand, when this type of message is sent to a computer that uses 32-bit integers the need arises to truncate the 64-bit quantities so that they will fit in the space available, with an exception being generated if data would be lost by such a truncation. Yet, if the message is merely being passed through some sort of intermediary, one would prefer that data not be truncated, since precision would be lost. In the reverse direction, sign extension or padding may need to be performed to convert a 32-bit quantity into an equivalent 64-bit quantity, but only if the data sent is a signed integer. Thus, a completely general RPC package needs to put a considerable amount of information into each packet, and may need to do quite a bit of work to represent data in a universal manner. On the other hand, such an approach may be much more costly than one that supports only a very limited set of possible representations, or that compiles the data marshalling and demarshalling operations directly into inline code.

The approach taken to marshalling varies from RPC package to package. SUN's XDR system is extremely general, but requires the user to code marshalling procedures for data types other than the standard base types of the system. With XDR, one can represent any desired data structure, even dealing with pointers and complex padding rules. At the other end of the spectrum are marshalling procedures that transmit data in the binary format used by the sender, are limited to only simple data types, and perhaps do little more than compatibility checking on the receive side. Finally, schemes like ISDN.1 are often used with RPC stub generators, which automatically marshall and demarshall data, but impose some restrictions on the types of objects that can be transmitted.

As a general rule of thumb, users will want to be aware that the more general solutions to these problems are also more costly. If the goal is extremely speed, it may make sense to design the application itself to produce data in a form that is inexpensive to marshall and demarshall. The cost implications of failing to do so can be surprising, and in many cases, it is not even difficult to redesign an interface so that RPC to it will be cheap.

4.5 Associated Services

No RPC system lives in isolation. As we will see below, RPC is often integrated with a security mechanism, and because security keys (and some parts of the RPC protocol itself) use timestamps, with a clock synchronization mechanism. For this reason, one often talks about distributed computing "environments" that include tools for implementing client-server applications including an RPC mechanism, security services and time services. Elaborate environments may go well beyond this, including system instrumentation and management interfaces and tools, fault-tolerance tools, and so-called Forth Generation Language (4GL) tools for building applications using graphical user interfaces (GUI's). Such approaches can empower even unskilled users to develop sophisticated distributed solutions. In this section we briefly review the most important of these services.

4.5.1 Naming services

A naming service maintains one or more *mappings* from some form of name (normally symbolic) to some form of value (normally, a network address). Naming services can operate in a very narrow, focused way – for example, the Domain Naming Service of the TCP/IP protocol suite maps short service names, in ascii, to IP addresses and port numbers, requiring exact matches. At the other extreme, one can talk about extremely general naming services that are used for many sorts of data, allow complex pattern matching on the name, and may return other types of data in addition to, or instead of, an address. One can even go beyond this, to talk about secure naming services that can be trusted to only give out validated addresses for services, very dynamic naming services that deal with applications like mobile computing systems in which hosts have addresses that change constantly, and so forth.

In standard computer systems at the time of this writing, three naming services are widely supported and used. Mentioned above, the Domain Name Service (DNS) is the least functional but most widely used. It responds to requests on a standard network port address, and for the "domain" in which it is running can map short (8 character) strings to internet port numbers. DNS is normally used for static services, which are always running when the system is operational and do not change port numbers at all. For example, the email protocol uses DNS to find the remote mail daemon capable of accepting incoming email to a user on a remote system.

The Network Information Service (NIS), previously called Yellow Pages (YP), is considerably more elaborate. NIS maintains a collection of maps, each of which has a symbolic name (e.g. "hosts", "services", etc.) and maps ascii keywords to an ascii value string. NIS is used on UNIX systems to map host names to internet addresses, service names to port numbers, etc. Although NIS does not support pattern matching, there are ways for an application to fetch the entire NIS database, one line at a time, and it is common to include multiple entries in an NIS database for a single host that is known by a set of aliases. NIS is a distributed service that supports replication: the same data is normally available from any of a set of servers, and a protocol is used to update the full set of servers if an entry changes. However, NIS is not designed to support rapid updates: the assumption is that NIS data consists of mappings like the map from host name to internet address, which change very rarely. A 12-hour delay before NIS information is updated is not unreasonable given this model, hence the update problem is solved by periodically refreshing the state of each NIS server by having it re-read the contents of a set of files in which the mapping data is actually stored. As an example, NIS is often used to store password information on UNIX systems.

X.500 is an international standard that many expect will eventually replace NIS. This service, which is designed for use by applications running the ISO standard remote procedure call interface and ISDN.1 data encoding, operates much like an NIS server. No provision has been made in the standard for replication or high performance update, but the interface does support some limited degree of pattern matching. As might be expected from a standard of this sort, X.500 addresses a wide variety of issues, including security and recommended interfaces. However, reliability issues associated with availability and consistency of the X.500 service (i.e. when data is replicated) have not yet been tackled by the standards organization.

Looking to the future, there is considerable interest in using X.500 to implement general purpose White-Pages (WP) servers, which would be explicitly developed to support sophisticated pattern matching on very elaborate databases with detailed information about abstract entities. Rapid update rates, fault-tolerance features, and security are all being considered in these proposals. At the time of this writing, it appears that the Web will require such services and hence that the work on universal resource naming for use in the Web will be a major driving force for evolution in this overall area.

4.5.2 Time services

With the launch of the so-called Global Positioning System satellites, micro-second accuracy become possible in workstations equipped with inexpensive radio receivers. Unfortunately, however, accurate clocks remain a major problem in the most widely used computer workstations and network technologies. We will have a great to say about this in Chapter 20, but some background may still be useful here.

At the time of this writing, the usual clock for a PC or workstation consists of a quartz-based chip much like the one in a common wristwatch, accurate to within a few seconds per year. The initial value of such a clock is either set by the vendor or by the user, when the computer is booted. As a result, in any network of workstations, clock can give widely divergent readings and can drift with respect to one-another at significant rates. For these reasons, there has been considerable study of algorithms for clock synchronization, whereby the clocks on invidual machines can be adjusted to give behavior approximating that of a shared global clock. In Chapter 20, we will discuss some of the algorithms that have been proposed for this purpose, their ability to tolerate failures, and the analyses used to arrive at theoretical limits on clock accuracy.

However, much of this work has a limited lifetime. GPS receivers can give extremely accurate time, and GPS signals are transmitted frequently enough so that even inexpensive hardware can potentially maintain time accurate to microseconds. By broadcasting GPS time values, this information can be propagated within a network of computers, and although some accuracy is necessarily lost when doing so, the resulting clocks are still accurate and comparable to within tens of microseconds. This development can be expected to have a major impact on the way that distributed software is designed – from a world of asynchronous communication and clocks that can be inaccurate by many times the average message latency in the network, GPS based time could catapult us into a domain in which clock resolutions considerably exceed the average latency between sending a message and when it is received. Such developments make it very reasonable to talk about synchronous (time-based) styles of software design and the use of time in algorithms of all sorts.

Even coarsely synchronized clocks can be of value in distributed software. For example, when comparing versions of files, microsecond accuracy is not needed to decide if one version is more current than another: accuracy of seconds or even tens of seconds may be adequate. Security systems often have a notion of expiration associated with keys, but for these to be at risk of "attacks" an intruder would need a way to set a clock back by days, not fractions of a second. And, although we will see that RPC protocols use time to detect and ignore very old, stale, messages, as in the case of a security mechanism a clock would need to be extremely inaccurate for such a system to malfunction.

4.5.3 Security services

In the context of an RPC environment, security is usually concerned with the *authentication* problem. Briefly stated, this is the problem of providing applications with accurate information about the user-id on behalf of which a request is being performed. Obviously, one would hope that the user-id is related in some way to the user, although this is frequently the weak link in a security architecture. Given an accurate source of user identifications, however, the basic idea is to avoid intrusions that can compromise user-id security through break-ins on individual computers and even replacements of system components on some machines with versions that have been compromised and hence could malfunction. As in the case of clock services, we will looking more closely at security later in the textbook (Chapter 19) and hence limit ourselves to a brief review here.

To accomplish authentication, a typical security mechanism (for example, the Kerberos security architecture for DCE [SNS88, Sch94]) will request some form of password or one-time key from the user at login time, and periodically thereafter, as keys expire on the basis of elapsed time. This information is used to compute a form of secure user-identification that can be employed during connection establishment. When a client binds to a server, the security mechanism authenticates both ends, and also (at the option of the programmer) arranges for data to be encrypted on the wire, so that intruders who witness messages being exchanged between the client and server have no way to decode the data contained within them. (Unfortunately, however, this step is so costly that many applications disable encryption and simply rely upon the security available from the initial connection setup). Notice that for such a system to work correctly, there must be a way to “trust” the authentication server itself: the user needs a way to confirm that it is actually talking to the authentication server, and to legitimate representatives of the services it wishes to use. Give the anonymity of network communication, these are potentially hard problems.

In Chapter 19, we will look closely at distributed security issues (for example, we will discuss Kerberos in much more detail), and also at the relationship between security and other aspects of reliability and availability – problems that are often viewed as mutually exclusive since one replicates information to make it more available, but would tend to restrict and protect it to make it more secure. We will also look at emerging techniques for protecting privacy, namely the “true” user-id’s of programs active in a network. Although the state of the art does not yet support construction of high performance, secure, private applications, this should be technically feasible within the not-distant future. Of course, technical feasibility does not imply that the technology will become widely practical and hence useful in building reliable applications, but at least the steps needed to solve the problems are increasingly understood.

4.5.4 Threads packages

Yet a fourth component of a typical RPC system is the lightweight threads package, which enables a single program to handle multiple tasks at the same time. Although threads are a general concept and indeed have rather little to do with communication per-se, they are often viewed as necessary in distributed computing systems because of the potential for deadlock if threads are *not* present.

To understand this point, it is helpful to contrast three ways of designing a communication system. A single-threaded message-based approach would correspond to a conventional style of programming extended directly to message passing. The programmer would use system calls like *sendto* and *recvfrom* as desired to send and receive messages. If there are several things happening at the same time in a program structured this way, however, the associated bookkeeping can be a headache (see Figure 4-3).

Threads offer a simple way to eliminate this problem: each thread executes concurrently with the others, and each incoming request spawns a new thread to handle it. While an RPC is pending the thread that issues it blocks (waits) in the procedure call that invoked the RPC. To the degree that there is any bookkeeping to worry about, the associated state is represented directly in the local variables of this procedure and in the call itself: when the reply is received, the procedure returns (the thread resumes execution), and there is no need to track down information about why the call was being done: this is "obvious" to the calling procedure. Of course, the developer does need to implement adequate synchronization to avoid concurrency-related bugs, but in general this is not a hard thing to do. The approach overcomes many forms of problems that are otherwise hard to address.

For example, consider a situation in which an RPC server is also the client of some other server, which is in turn the client of still additional servers. It is entirely possible that a cycle could form, in which RPC *a* by process *x* on process *y* leads to an RPC *b* by *y* on *z*, and so forth, until finally some process in the chain makes a request back to the original process, *x*. If these calls were LPC calls, such a sequence would simply be a form of recursion. For a single-threaded RPC system, however, *x* will be busy performing RPC *a* and hence would be unresponsive, creating a deadlock. Alternatively, *x* would need to somehow save the information associated with sending RPC *a* while it is handling this new incoming request. This is the bookkeeping problem aluded to above.

Yet a third option is known as "event dispatch" and is typical of windowing systems, in which each action by the user (mouse motion or clicks, keyboard entries) results in delivery of an "event" record to a central dispatching loop. The application program typically registers a set of procedure callbacks to perform when events of interest are received: if the left mouse button is pressed, invoke *left_button()*. Arguments to these callbacks tell the program exactly what occured: the cursor was at position 132,541 when the mouse button was pressed, this is inside such and such a window, etc. One can use the same approach to handle event dispatch in message-based systems: incoming messages are treated as "events" and result in callbacks to handler procedures.

The approaches can also be combined: event dispatch systems can, for example, fork a new thread for each incoming message. In the most general approach, the callback is registered with some indication of how it should be performed: by forking a thread, by direct procedure call, or perhaps even by some other method, such as enqueuing the event on an event queue. This last approach is used in the Horus system, which we will discuss in Chapter 18.

At the time of this writing, although this is not universally the case, many RPC systems are built directly over a lightweight threads package. Each incoming RPC is handled by a new thread, eliminating the risk of deadlock, but forcing the programmer to learn about lightweight threads, preemption, mutual exclusion mechanisms, and other issues associated with concurrency. In this text, we will present some protocols that in which processes are assumed to be multi-threaded, so that the initiator of a protocol can also be a participant in it. However, we will not explicitly discuss thread packages or make use of any special features of particular packages.

Threads: A Personal Prospective

Speaking from personal experience, I have mixed feelings on the issue of threads. Early in my career I worked with protocols implemented directly over a UDP datagram model. This turned out to be very difficult: such a system needs to keep track of protocol "state" in some form of table, matching replies with requests, and is consequently hard to program. For example, suppose that a distributed file server is designed to be single-threaded. Such a file server may handle many applications at the same time, so it will need to send off one request, perhaps to read a file, but remain available for other requests, perhaps by some other application that wants to write a file. The information needed to keep track of the first request (the read that is pending) will have to be recorded in some sort of pending activities table, and later matched with the incoming reply from the remote file system. Having implemented such an architecture once, I would not want to do it again.

This motivated me to move to RPC-style protocols, using threads. We will be talking about the Isis Toolkit, which is a system that I implemented (with help from others!) in the mid 1980's, and in which lightweight threads were employed extensively. Many Isis users commented to me that they had never used threads before working with Isis, and were surprised at how much the approach simplified things. This is certainly the case: in a threaded system, the procedure handling the "read" would simply block waiting for the reply, while other procedures can be executed to handle other requests. The necessary bookkeeping is implicit: the blocked procedure has a local state consisting of its calling stack, local variables, and so forth. Thus there is no need to constantly update a table of pending activities.

Of course, threads are also a potential source of insidious programming bugs. In Isis, the benefits of threads certainly outweighed the problems associated with them, but it also is clear that this model requires a degree of programming sophistication that goes somewhat beyond standard single-threaded programming. It took me at least a year to get in the habit of thinking through the potential reentrancy and ordering issues associated with concurrency and to become comfortable with the various styles of locking needed to overcome these problems. Many users report the same experience. Isis, however, is perhaps an unusually challenging case because the order in which events happened is very important in this system, for reasons that we will study in Part III of the textbook.

In more recent work, I have teamed up with Robbert Van Renesse, who is the primary author of the Horus system (we discuss this in considerable detail in Chapter 18). Horus, like Isis, was initially designed to use threads and is extremely sensitive to event ordering. But when testing very demanding application, Robbert found that threads were a serious a source of overhead and "code bloat": overhead because a stack for a thread consumes 16k bytes or more of space, which is a lot of space in a system that can handle tens of thousands of messages per second, and excess code because of the necessary synchronization. Yet, as in the case of Isis, Horus sometimes needs threads: they often make it easy to do things that would be very hard to express in a non-threaded way.

Robbert eventually extended Horus to use an event-dispatch model much more like the one in Windows NT, which offers threads as an option over a basic event dispatch mechanism. This step, which substantially simplified many parts of Horus, left me convinced that supporting threads over an event dispatch architecture is the right way to go. For cases in which a thread is needed, it is absolutely vital that they be available. However, threads bring a considerable amount of baggage, which may be unecessary in many settings. An event dispatch style of system gives the developer freedom to make this choice and has a "default" behavior that is lightweight and fast. On the other hand, I am still convinced that event dispatch systems that lack the option of forking a thread when one is desired are often unwieldy and very difficult to use; this approach should be avoided.

Figure 4-3: Rather than chosing between threads and event-dispatch, an approach that supports threads as an option over event dispatch offers more flexibility to the developer.

The use of threads in this manner remains debatable. UNIX programs have heavily favored this approach, and the UNIX community generally understands the issues that must be addressed and minimizes their difficulty. Indeed, with experience, threaded programming is not all that hard. One merely needs to get in the habit of enforcing necessary synchronization using appropriate interlocks. However, the PC community tends to work with an event-based model that lacks threads, in which the application is visualized as a dispatcher for incoming events and all callbacks are by procedure invocation. Thus, the PC community has its own style of programming, and it is largely non-threaded. Windows NT further complicates this picture: it supports threads, and yet uses an event-oriented style of displatching throughout the operating system; if a user wants to create a thread to handle an event, this is easily done but not "forced" upon the programmer.

4.6 The RPC Protocol

The discussion up to this point has focused on client/server computing and RPC from the perspective of the user. A remote procedure call *protocol* is concerned with the actual mechanism by which the client process issues a request to a server, and by which the reply is transmitted back from the server to the client. We now look at this protocol in more detail.

Abstractly, the remote procedure call problem, which an RPC protocol undertakes to solve, consists of emulating LPC using message passing. LPC has a number of "properties" – a single procedure invocation results in exactly one execution of the procedure body, the result returned is reliably delivered to the invoker, and exceptions are raised if (and only if) an error occurs.

Given a completely reliable communication environment, which never loses, duplicates, or reorders messages, and given client and server processes that never fail, RPC would be trivial to solve. The sender would merely package the invocation into one or more messages, and transmit these to the server. The server would unpack the data into local variables, perform the desired operation, and send back the result (or an indication of any exception that occurred) in a reply message. The challenge, then, is created by failures.

Were it not for the possibility of process and machine crashes, an RPC protocol capable of overcoming limited levels of message loss, disorder and even duplication would be easy to develop (Figure 4-4). For each process to which it issues requests, a client process maintains a message sequence number. Each message transmitted carries a unique sequence number, and (in most RPC protocols) a time stamp from a global clock – one that returns roughly the same value throughout the network, up to clock synchronization limits. This information can be used by the server to detect very old or duplicate copies of messages, which are discarded, and to identify received messages using what are called *acknowledgment* protocol-messages.

The basic idea, then, is that the client process transmits its request and, until acknowledgments have been received, continues to retransmit the same messages periodically. The server collects messages and, when the full request has been received, performs the appropriate procedure invocation. When it transmits its reply, the same sort of reliable communication protocol is used. Often, the acknowledgement is delayed briefly in the hope that the reply will be sent soon, and can be used in place of a separate acknowledgement.

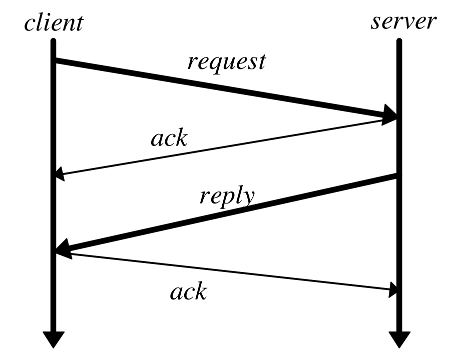


Figure 4-4: Simple RPC interaction, showing packets that contain data (dark) and acknowledgements (light)

A number of important optimizations have been proposed by developers of RPC-oriented distributed computing environments. For example, if one request will require the transmission of multiple messages, because the request is large, it is common to inhibit the sending of acknowledgments during the transmission of the burst of messages. In this case, a *negative acknowledgement* is sent if the receiver detects a missing packet; a single ack confirms reception of the entire burst when all packets have been successfully received (Figure 4-5). Similarly, it is common to delay the transmission of acknowledgment packets in the hope that the reply message itself can be transmitted instead of an acknowledgment: obviously, the receipt of a reply implies that the corresponding request was delivered and executed.

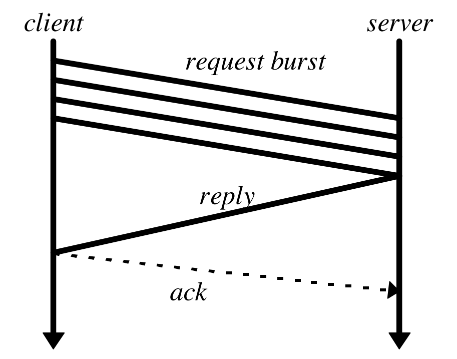


Figure 4-5: RPC using a burst protocol; here the reply is sent soon enough so that an ack to the burst is not needed.

Process and machine failures, unfortunately, render this very simple approach inadequate. The essential problem is that because communication is over unreliable networking technologies, when a process is unable to communicate with some other process, there is no way to determine whether the problem is a network failure, a machine failure, or both (if a process fails but the machine remains operational the operating system will often provide some status information, permitting this one case to be accurately sensed).

When an RPC protocol fails by timing out, but the client or server (or both) remains operational, it is impossible to know what has occurred. Perhaps the request was never received, perhaps it was received and executed but the reply was lost, and perhaps the client or server crashed while the protocol was executing. This creates a substantial challenge for the application programmer who wishes to build an application that will operate reliably despite failures of some of the services upon which it depends.

A related problem concerns the issue of what are called *exactly once semantics*. When a programmer employs LPC, the invoked procedure will be executed exactly once for each invocation. In the case of RPC, however, it is not evident that this problem can be solved. Consider a process *c* that issues an RPC to a service offered by process *s*. Depending upon the assumptions we make, it may be very difficult even to guarantee that *s* performs this request *at most* once. (Obviously, the possibility of a failure precludes a solution in which *s* would perform the operation exactly once).

To understand the origin of the problem, consider the possible behaviors of an arbitrary communication network. Messages can be lost in transmission, and as we have seen this can prevent process *c* from accurately detecting failures of process *s*. But, the network might also misbehave by delivering a message after an unreasonably long delay. For example, suppose that a network router device fails by jamming up in such a manner that until the device is serviced, the software within it will simply wait for the hardware to be fixed. Obviously, there is no reason to simply assume that routers won't behave this way, and in fact it is known that some routers definitely could behave this way. Moreover, one can imagine a type of attack upon a network in which an intruder records messages for future replay.

One could thus imagine a situation in which process *s* performs a request from *c*, but then is presented with the same request after a very long delay (Figure 4-6). How can process *s* recognize this as a duplicate of the earlier request?

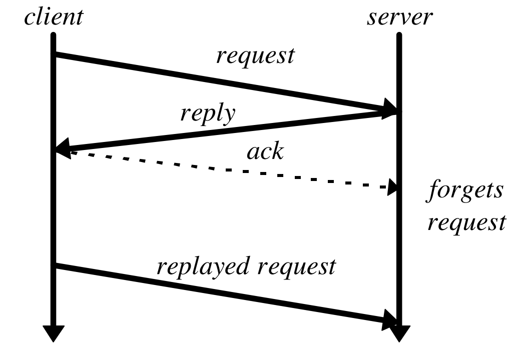


Figure 4-6: If an old request is replayed, perhaps because of a transient failure in the network, a server may have difficulty protecting itself against the risk of re-executing the operation.

Depending upon the specific protocol used, an RPC package can use a variety of barriers to protect itself against replays of long-delayed messages. For example, the package might check timestamps in the incoming messages, rejecting any that are very old. Such an approach, however, presumes that clocks are synchronized to a reasonable degree and that there is no danger that a message will be replayed with a modified timestamp – an action that might be well within the capabilities of a sophisticated intruder. The server could use a connect-based binding to its clients, but this merely pushes the same problem into the software used to implement network connections – and as we shall see shortly, the same issues arise and remain just as intractable at that level of a system. The server might maintain a list of currently valid users, and could insist that each message be identified by a monotonically increasing sequence number – but a replay could, at least theoretically, reexecute the original binding protocol.

Analyses such as these lead us to two possible conclusions. One view of the matter is that an RPC protocol should take reasonable precautions against replay but not be designed to protect against extreme situations such as replay attacks. In this approach, an RPC protocol might claim to guarantee *at most once semantics*, meaning that provided that the clock synchronization protocol has not been compromised or some sort of active attack been mounted upon the system, each operation will result in either a single procedure invocation or, if a communication or process failure occurs, in no invocation. An RPC protocol can similarly guarantee *at least once semantics,* meaning that if the client system remains operational indefinitely, the operation will be performed at least once but perhaps more than once. Notice that both types of semantics come with caveats: conditions (hopefully very unlikely ones) under which the property would still not be guaranteed. In practice, most RPC environments guarantee a weak form of at most once semantics: only a mixture of an extended network outage and a clock failure could cause such systems to deliver a message twice, and this is not a very likely problem.

A different approach, also reasonable, is to assume a very adversarial environment and protect the server against outright attacks that could attempt to manipulate the clock, modify messages, and otherwise interfere with the system. Security architectures for RPC applications commonly start with this sort of extreme position, although it is also common to weaken the degree of protection to obtain some performance benefits within less hostile subsets of the overall computing system. We will return to this issue and discuss it in some detail in Chapter 19.

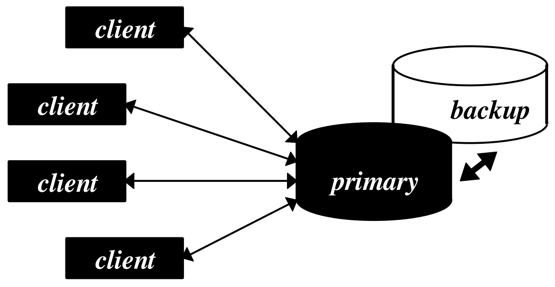


Figure 4-7: Idealized primary-backup server configuration. Clients interact with the primary and the primary keeps the backup current.

4.7 Using RPC in Reliable Distributed Systems

The uncertainty associated with RPC failure notification and the weak RPC invocation semantics seen on some system pose a challenge to the developer of a reliable distributed application.

A reliable application would typically need multiple sources of critical services, so that if one server is unresponsive or faulty the application can re-issue its requests to another server. If the server behaves as a "read only" information source, this may be an easy problem to solve. However, as soon as the server is asked to deal with dynamically changing information, even if the changes are infrequent compared to the rate of queries, a number of difficult consistency and fault-tolerance issues arise. Even questions as simple as load balancing, so that each server in a service spanning multiple machines will do a roughly equal share of the request processing load, can be very difficult to solve or reason about.

For example, suppose that an application will use a primary-backup style of fault-tolerance, and the requests performed by the server affect its state. The basic idea is that an application should connect itself to the primary, obtaining services from that process as long as it is operational. If the primary fails, the application will "fail over" to the backup. Such a configuration of processes is illustrated in Figure 4-7. Notice that the figure includes multiple client processes, since such a service might well be used by many client applications at the same time.

Consider now the design of a protocol by which the client can issue an RPC to the primary-backup pair such that if the primary performs the operation, the backup learns of the associated state change. In principle, this may seem simple: the client would RPC to the server, which would compute the response and then RPC to the backup, sending it the request it performed, the associated state change, and the reply being returned to the client. Then the primary would return the reply, as show in Figure 4-8.

This simple protocol is, however, easily seen to be flawed if the sorts of problems we discussed in the previous section might occur while it was running [BG95]. Take the issue of timeout. In this solution, two RPC's occur, one nested within the other. Either of these, or both, could fail by timeout, in which case there is no way to know with certainty what state the system was left in. If, for example, the client sees a timeout failure, there are quite a few possible explanations: the request may have been lost, the reply may have been lost, and either the primary or the primary and the backup may have crashed. Failover to the backup would only be appropriate if the primary is indeed faulty, but there is no accurate way to determine if this is the case, except by waiting for the primary to recover from the failure – not a very "available" approach.

The matter is further complicated by the presence of more than one client. One could easily imagine that different clients could observe different and completely uncorrelated outcomes for requests issued simultaneously but during a period of transient network or computer failures. Thus, one client might see a request performed successfully by the primary, another might conclude that the primary is apparently faulty and try to communicate with the backup, and yet a third may have timed out both on the primary *and* the backup! We use the term *inconsistent* in conjunction with this sort of uncoordinated and potentially incorrect behavior. An RPC system clearly is not able to guarantee the consistency of the environment, at least when the sorts of protocols discussed above are employed, and hence reliable programming with RPC is limited to very simple applications.

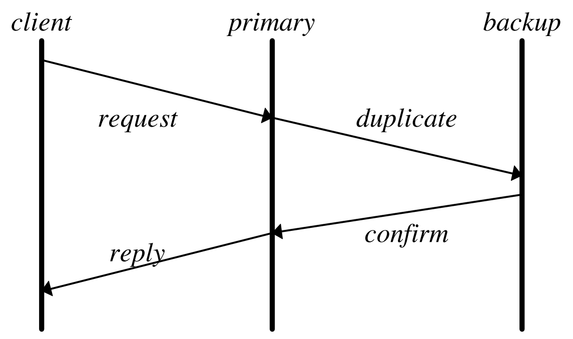


Figure 4-8: Simplistic RPC protocol implementing primary-backup replication.

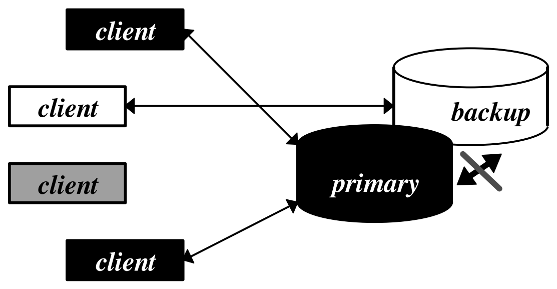


Figure 4-9: RPC timeouts can create inconsistent states, such as this one, in which two clients are connected to the primary, one to the backup, and one is disconnected from the service. Moreover, the primary and backup have become disconnected from one another — each considers the other faulty. In practice, such problems are easily provoked by transient network failures. They can result in serious application-level errors. For example, if the clients are air traffic controllers and the servers advise them on the safety of air traffic routing changes, this scenario could lead two controllers to route different planes into the same sector of the airspace! Understanding that such problems represent serious threats to reliability and exploring solutions to them are the major goals of this textbook.

The line between “easily” solved RPC applications and very difficult ones is not a very clear one. For example, one major type of file server accessible over the network is accessed by an RPC protocol with very weak semantics, which can be visible to users. Yet this protocol, called the NFS (Network File System) protocol, is widely popular and has the status of a standard, because it is easy to implement and widely available on most vendor computing systems. NFS is discussed in some detail in Section 7.3 and so we will be very brief here.

One example of a way in which NFS behavior reflects an underlying RPC issues arises when creating a file. NFS documentation specifies that the file creation operation should return the error code EEXISTS if a file already exists at the time the create operation is issued. However, there is also a case in which NFS can return error EEXISTS even though the file did not exist when the create was issued. This occurs when the create RPC times out, but the request was in fact delivered to the server and was performed successfully. NFS automatically reissues requests that fail by timing out and will retry the create operation, which now attempts to re-execute the request and fails because the file is now present. In effect, NFS is unable to ensure at most once execution of the request, and hence can give an incorrect return code. Were NFS implemented using LPC (as in the “LFS” or “local file system”), this behavior would not be possible.

NFS illustrates one approach to dealing with inconsistent behavior in an RPC system. By weakening the semantics presented to the user or application program, NFS is able to provide acceptable behavior despite RPC semantics that create considerable uncertainty when an error is reported. In effect, the erroneous behavior is simply redefined to be a “feature” of the protocol.

A second broad approach that will interest us here involves the use of agreement protocols by which the components of a distributed system maintain consensus on the status (operational or failed) of one another. A rigorous derivation of the obligations upon such consensus protocols, the limitations on this approach, and the efficient implementation of solutions will be topics of chapters later in this textbook (Section 13.3). Briefly, however, the idea is that any majority of the system can be empowered to “vote” that a minority (often, just a single component) be excluded on the basis of apparently faulty behavior.

Such a component is cut off from the majority group: if it is not really faulty, or if the failure is a transient condition that corrects itself, the component will be prevented from interacting with the majority system processes, and will eventually detect that it has been dropped. It can then execute a rejoin protocol, if desired, after which it will be allowed back into the system.

With this approach, failure becomes an abstract event – true failures can trigger this type of event, but because the system membership is a self-maintained property of the system, the inability to accurately detect failures need not be reflected through inconsistent behavior. Instead, a conservative detection scheme can be used, which will always detect true failures while making errors infrequently, in a sense we will make precise in Section 13.9.

By connecting an RPC protocol to a group membership protocol that runs such a failure consensus algorithm, a system can resolve one important aspect of the RPC error reporting problems discussed above. The RPC system will still be unable to accurately detect failures, hence it will be at risk of incorrectly reporting operational components as having failed. However, the behavior will now be consistent throughout the system: if component *a* observes the failure of component *b*, than component *c* will also observe the failure of *b*, unless *c* is also determined to be faulty. In some sense, this approach eliminates the notion of failure entirely, replacing it with an event that might be called "exclusion from membership in the system." Indeed, in the case where *b* is actually experiencing a transient problem, the resulting execution is much like being exiled from one's country, or like being shunned: *b* is prevented from communicating with other members of the system and learns this. Conversely, the notion of a majority allows the operational part of the system to initiate actions on behalf of the full membership in the system. "The system" now becomes identified with a rigorous concept: the output of the system membership protocol, which can itself be defined formally and reasoned about using formal tools.

As we move beyond RPC to consider more complex distributed programming paradigms, we will see that this sort of consistency is often required in non-trivial distributed applications. Indeed, there appears to be a dividing line between the distributed applications that give non-trivial coordinated behavior at multiple locations, and those that operate as completely decoupled interacting components, with purely local correctness criteria. The former type of system requires the type of consistency we have encountered in this simple case of RPC error reporting. The latter type of system can manage with error detection based upon timeouts – but is potentially unsuitable for supporting any form of consistent behavior.

4.8 Related Readings

A tremendous amount has been written about client-server computing, and several pages of references could easily have been included here. Good introductions into the literature, including more detailed discussions of DCE and ASN.1, can be found in [BN84, Tan88, CS93, CDK94]. On RPC performance, the "classic" reference is [SB89]. Critiques of the RPC paradigm appear in [TR88, BR94]. On the problem of inconsistent failure detection with RPC: [BG95]. Other relevant publications include [BCLF94, BCLF95, BD95, BKT90, BM90, BN84, Bro94, EBBV95, EKO95, GA91, HP94, Jac88, Jac90, MRTR90, Ras86, SB89, TL93]. A good reference to DCE is [DCE94] and to OLE-2 is [Bro94]. Kerberos is discussed in [SNS88, BM90, Sch94].

5. Streams

In Section 1.2 we introduced the idea of a reliable communications channel, or stream, that could overcome message loss and out-of-order delivery in communication sessions between a source and destination process. In this chapter we briefly discuss the mechanisms used to implement the streams abstraction [Rit84], including the basic sliding window numbering scheme, flow control and error correction mechanisms. We describe the most common performance optimizations, which are based on dynamic adjustments of window size and transmission of short packet bursts. Finally, we look more closely at the reliability properties of a sliding window protocol, including the consistency of fault-notification. Recall that the inconsistency of fault-notification represented a drawback to using RPC protocols in reliable distributed applications. Here, we'll ask two sorts of questions about stream reliability: how streams themselves behave, and how an RPC protocol operating over a stream can be expected to behave.

5.1 Sliding Window Protocols

The basic mechanism underlying most streams protocols is based on a data structure called a sliding window. More specifically, a typical bidirectional stream requires two sliding windows, one for each direction. Moreover, each window is duplicated on the sending and receiving sides. A window has limited size, and is usually organized a list of window slots, each having a fixed maximum size.

For example, the TCP protocol normally operates over the IP protocol. It uses slots limited by the IP packet size (1400 bytes), each divided between a TCP header and a payload portion. The size of a TCP window can vary dynamically, and the default values are often adjusted by the vendor to optimize performance on their platform.

A sliding window protocol treats the data communication from sender to destination as a continuous stream of bytes. (The same approach can be applied to messages, too, but this is less common and in any case, doesn't change the protocol in a significant way). The window looks into this stream and represents the portion of the stream that is actively being transmitted at a given point in time. Bytes that have already passed through the window have been received and delivered; they can be discarded by the sender. A sliding window protocol is illustrated in Figure 5-1.

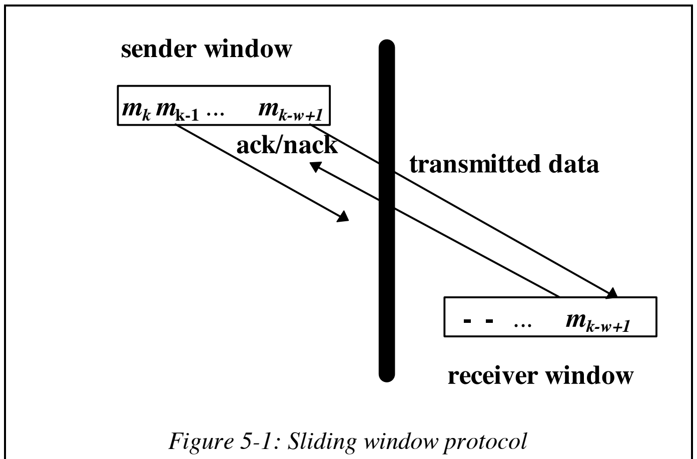


Figure 5-1: Sliding window protocol

Here we see a window of size w which is full on the sender side. Only some of the contents have been successfully received, however; in particular, m_k and m_{k-1} have yet to be received. The acknowledgement for m_{k-w+1} and those bytes still in the window are in transit; the protocol will be operating on them to overcome packet loss, duplication, and out-of-order delivery. As we will can see here, the sender's window can lag behind the receivers window, so the receiver will often have "holes" in the window representing bytes that have not yet been received — indeed, that may not yet have been sent.

In the window illustrated by Figure 5-1, the sender's window is full. If an attempt is made to transmit additional data, the sender process would be "blocked" waiting until the window drains. This data is delayed on the sender side, and may wait quite a while before being sent.

Each slot in the window has some form of sequence number and length. For TCP, byte numbers are used for the sequence number. Typically, these numbers can wrap (cycle), but only after a very long period of time. For example, the sequence number 0 could be reused after the TCP channel has transmitted a great deal of data. Because of this, a sliding window has some potential to be confused by very old duplicate packets, but the probability of this happening (except through an intruder who is intentially attacking the system) is extremely small.

For example, many readers may have had the experience of using the *telnet* program to connect to a remote computer over a network. Telnet operates over TCP, and you can sometimes see the effect of this queuing. Start a very large listening in your telnet window, and then try to interrupt it. Often, many pages will scroll past before the interrupt signal breaks through! What you are seeing here is the backlog of data that was already queued up to be transmitted at the time the interrupt signal was received. This data was written by the program that creates listings to your remote terminal, though a TCP connection. The listing program is much faster than TCP, hence it rapidly overflows the sliding window and builds a backlog of bytes. The amount of backlog, like the size of the TCP window, is variable from platform to platform, but 8k bytes is typical. Finally, when this amount of data has piled up, the next attempt to write to the TCP connection will block, delaying the listing generation program. The interrupt signal kills the program while it is in this state, but the 8k or so of queued data, plus any data that was in the window, will still be transmitted to your terminal.[4]

5.1.1 Error Correction

The most important role for a sliding window protocol is to overcome packet loss, duplication and out of order delivery. This is done by using the receive-side window to reorder incoming data.

As noted earlier, each outgoing packet is labeled with a sequence number. The sender transmits the packet, then sets a timer for retransmission if an acknowledgement is not received within some period of time. Upon reception, a packet isslotted into the appropriate place in the window, or discarded as a duplicate if it repeats some packet already in the window or has a sequence number too small for the window. The sender-side queueing mechanism ensures that a packet sequence number will not be too large, since such a packet would not be transmitted.

The receiver now sends an acknowledgement for the packet (even if it was a duplicate), and also may send a negative acknowledgement for missing data, it if is clear that some data has been dropped.

In bidirectional communication, it may be possible to send acknowledgement and negative acknowledgement information on packets being sent in the other direction. Such "piggybacking" is desirable, when practical, because of the high overhead associated with sending a packet. Sending fewer packets, even if the packets would have been very small ones, is extremely beneficial for performance of a

4. Some computers also clear any backlog of data to the terminal when you cause a keyboard interrupt. Thus, if you strike interrupt several times in a row, you may be able to avoid seeing all of this data print; each interrupt will clear several thousand bytes of data queued for output. This can create the impression that the first interrupt didn't actually kill the remote program, and that several attempts were required before the program received the signal. In fact, the first interrupt you entered is always the one that kills the program, and the slow response time is merely a reflection of the huge amounts of data that the system has buffered in the connection from the remote program to your terminal!

distributed application. Slightly longer packets, in fact, will often have essentially identical cost to shorter ones: the overhead of sending a packet of *any* size is considerably larger than the cost of sending a few more bytes. Thus, a piggybacked acknowledgement may be essentially free, in terms of the broader cost of communication within a system.

Streams normally implement some form of keep-alive messages, so that the sender and receiver can monitor one-another's health. If a long period passes with no activity on the channel, a process can timeout and close the channel, pronouncing its remote counterpart "faulty". As discussed in earlier chapters, such a timer-based failure detection mechanism can never guarantee accuracy, since communication problems will also trigger the channel shutdown protocol. However, this mechanism is part of the standard stream implementations.

5.1.2 Flow Control

A second role for the sliding window protocol is to match the speed of data generation and the speed of data consumption. With a little luck, the sender and receiver may be able to run continuously, getting the same throughput as if there were no channel between them at all. Latency, of course, may be higher than if the sender was directly connected to the receiver, but many applications are insensitive to latency. For example, a file transfer program is sensitive to latency when a request is issued to transfer a file for the first time. Once a file transfer is underway, however, throughput of the channel is the only relevant metric, and if the throughput is higher than the rate with which the receiver can consume the data, the "cost" of remote file access may seem very small, or negligible.

The way that this rate matching occurs is by allowing more than one packet to be in the sliding window at a time. For example, the window illustrated in Figure 5-1 is shown with *w* slots, all filled. In such a situation one might see *w* packets all on the wire at the same time. It is more likely, however, that some of this data will have already been received and that the acknowledgements are still in transit back to the sender. Thus, if the rate of data generated by the sender is close to that of the receiver, and if the window is large enough to contain the amount of data the sender can generate in a typical round-trip latency, the sender and the receiver may remain continously active.

Of course, there are many applications in which the sender and receiver operate at different rates. If the receiver is faster, the receive window will almost always be empty, hence the sender will never delay before sending data. If the sender is faster, some mechanism is needed to slow it down. To this end, it is not uncommon to associate what is called a *high-water/low-water* threshold with a sliding window.

The high-water threshold is the capacity of the window; when this threshold is reached, any attempt to send additional data will block. The low-water threshold is a point below which the window must drain before such a blocked sender is permitted to resume transmission. The idea of such a scheme is to avoid paying the overhead of rescheduling the sender if it would block immediately. For example, suppose that the window has capacity 8kb. By setting the low-water threshold to 2kb, the sender would sleep until there is space for 6kb of new data before resuming communication. This would avoid a type of thrashing in which the sender might be rescheduled when the window contains some very small amount of space, which the sender would immediately overflow. The number 8kb, incidentally, is a typical default for streaming protocols in operating systems for personal computers and workstations; higher performance systems often use a default of 64kb, which is also a hard limit for standard implementations of TCP.

5.1.3 Dynamic Adjustment of Window Size

Suppose that the sender and receiver operate at nearly the same rates, but that the window is too small to permit the sender and receiver rates to match. This would occur if the sender generates data fast enough to completely fill the window before a message can travel round-trip to the receiver and back. In such a situation, the sender will repeatedly block much as if the receiver was slower than it. The receiver, however, will almost always have an empty window and will be active only in brief bursts. Clearly, the system will now incur substantial idle time for both sender and receiver, and will exhibit relatively low utilization of the network. The need to avoid this scenario has motivated the development of a number of mechanisms to dynamically grow the sender window.

On the other side of the coin, there are many potential problems if the window is permitted to become too large. Such a window will consume scarce kernel memory resources, reducing the memory available for other purposes such as buffering incoming packets as they are received off the network, caching disk and virtual memory blocks, and managing windows associated with other connections. Many of these situations are manifested by increased rates of low-level packet loss, which occurs when the lowest level transport protocols are unable to find temporary buffering space for received packets.

The most widely used dynamic adjustment mechanism was proposed by Van Jacobson in a Ph.D. thesis completed at Stanford University in 1988 [Jac88]. Jacobson's algorithm is based on a exponential growth in the window size when it appears to be too small, i.e. by doubling, and a linear backoff when the window appears to be too large. The former condition is detected when the sender's outgoing window is determined to be full, and yet only a single packet at a time is acknowledged, namely the one with the smallest slot number. The later condition is detected when high rates of retransmission occur. Exponential growth means that the number of window slots is increased by doubling, up to some pre-arranged upper limit. Linear backoff means that the window size is reduced by decrementing the number of slots on the sender's side. Under conditions where these changes are not made too frequently, they have been demonstrated to maintain a window size close to the optimal.

5.1.4 Burst Transmission Concept

When the performance of a sliding window protocol is studied closely, one finds that the window size is not the only performance limiting factor. The frequency of acknowledgements and retransmissions is also very important and must be kept to a minimum: both types of packets are purely overhead. As noted earlier, acknowledgements are often piggybacked on other outgoing traffic, but this optimization can only be exploited in situations where there actually are outgoing packets. Thus, the method is useful primarily in applications that exhibit a uniform, bidirectional flow of data. A separate mechanism is needed to deal with the case of messages that flow primarily in a single direction.

When the window size is small and the packets transmitted are fairly large, it is generally difficult to avoid acknowledgements of every packet. However, transmission of data in *bursts* can be a useful tactic in sliding window protocols that have relatively large numbers of slots and send relatively small packets [BN84]. In a burst transmission scheme, the sender attempts to schedule transmissions in bursts of several packets that are sent successively, with little or no delay between them. The sender then pauses before sending more data. The receiver uses the complementary algorithm, delaying its acknowledgements for a period of time in the hope of being able to acknowledge a burst of packets with a single message. A common variation upon this scheme uses a *burst bit* to indicate that a packet will be closely followed by some other packet; such schemes are also sometimes known as using *packet trains*. The last packet in the burst or train is recognizable as such because its burst bit is clear, and flushes the acknowledgement back to the sender.

The biggest benefits of a burst transmission algorithm are seen in settings where very few instructions are needed to transmit a message, and in which the sender and receiver are closely rate-matched. In such a setting, protocol designers strive to achieve a sort of perfect synchronization in which the transmission of data in the outgoing direction is precisely matched to a minimal flow of acknowledgements back to the sender. Tactics such as these are often needed to squeeze the maximum benefit out of a very high performance interconnect, such as the message bus of a parallel supercomputer. However, they work well only if the communication activity is in nearly complete control of the sender and receiving machine, if message transmission latencies (and the variation in message latencies) are known and small, and if the error rate is extremely low.

5.2 Negative-Acknowledgement Only

As communication bandwidths have risen, the effective bandwidth lost to acknowledgements can become a performance-limiting factor.This trend has lead to the development of what are called *negative acknowledgement* protocols, in which packets are numbered in a traditional way, but are not actually acknowledged upon reception. The sender uses a rate-based flow control method to limit the volume of outgoing data to a level the receiver is believed capable of accepting. The receiver, however, uses a traditional windowing method to reorder incoming messages, delete duplicates, and uses negative acknowledgement messages to solicit retransmission of any missing packets.

The exponential increase/linear backoff flow control algorithm is often combined with a negative acknowledgement scheme to maximize performance. In this approach, which we first saw in conjunction with variable window-size adjustments, the sender increases the rate of transmission until the frequency of lost packets that must be retransmitted exceeds some threshold. The sender then backs off, reducing its rate steadily until the sender's rate and the receiver's rate are approximately matched, which is detected when the frequency of negative acknowledgements drops below some second (lower) threshold.

5.3 Reliability, Fault-tolerance, and Consistency in Streams

It is common for programming manuals and vendor product literature to characterize streaming protocols as "reliable", since they overcome packet loss. Scrutinized closely, however, it is surprisingly difficult to identify ways in which these protocols really guarantee reliability of a sort that could be depended upon in higher level applications [BG95]. Similar to the situation for a remote procedure call, a stream depends on timeouts and retransmissions to deal with communication errors, and reports a failure (breaking the stream) when the frequency of such events exceeds some threshold. For example, if the communication line connecting two machines is temporarily disrupted, the stream connections between them will begin to break. They will not, however, break in a coordinated manner. Quite the contrary, each stream will break on its own, after a delay that can vary widely depending on how active the stream was immediately before the communication problem occurred: a stream that was recently active will remain open for a longer period of time, while a stream that was inactive for a period of time before communication was disrupted will be closer to its timeout and hence will break sooner. It is common to see an interval of as much as several minutes between the fastest detection of a communication failure and the slowest.

The problem that this creates is that many application programs interpret a broken stream to imply the failure of the program at the other end. Indeed, if a program fails, the streams to it will break eventually; in some situations this occurs within milliseconds (i.e. if the operating system senses the failure and closes the stream explicitly). Thus, some applications trigger failure recovery actions in situations that can also be the result of mundane communication outages that are rapidly repaired. Such applications may be left in an inconsistent state. Moreover, the long delays between when the earliest "broken channel" occurs and when the last one occurs can create synchronization problems, whereby one part of an application starts to reconfigure to recover from the failure while other parts are still waiting for the failed component to respond. Such skewed executions stress the application and can reveal otherwise hidden bugs, and may also have puzzling effects from the perspective of a user, who may see a screen that is partially updated and partially frozen in an old state, perhaps for an extended period of time.

As an example, consider the client-server structure shown below, in which a client program maintains streams to two server programs. One server is the primary: it responds to requests in the normal mode of operation. The second is the backup, and takes over only if the primary crashes. Streams are used to link these programs with one-another. The precise replication and fault-tolerance mechanism used is not important for the point we wish to make; later in the text (Section 15.3.4) we will see how primary-backup problems such as this can be solved.

Notice that the stream connecting the client to the server has broken. How should the client interpret this scenario? One possibility is that the primary server has actually failed; if so, the connection between the primary server and the backup will eventually break too (although it may take some time before this occurs), in which case the backup will eventually take over the role of the primary and service to the client will be restored. A second possibility is that the client itself has failed, or is on a computer that has lost its communications connectivity to the outside world. In this case the client will eventually shut down, or will see its backup connection break too. Again, this may take some time to occur. Yet a third possibility is that the connection was lost as a consequence of a transient communication problem. In this case the client should reconnect to the server, which will have seen the complementary situation and eventually have concluded that the client has failed.

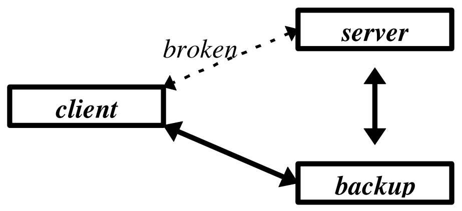


Figure 5-2: Inconsistently broken streams

The problem is that in this last situation, the reliability properties of the stream have basically been lost. The stream was designed to overcome packet loss and communication disruptions, but in the scenario illustrated by the figure, it has done neither of those things. On the contrary, the inconsistent behavior of the separate streams present has turned out to *cause* a reliability problem. Data in the channel from the client to the server may have reached it, or it may have been lost when the connection was severed. Data from the server to the client may similarly have been lost. Thus, the client and the server must implement some protocol at the application level that will handle retransmission of requests that could have been dropped when a channel broke, and that will resolicit any data that the server may have been sending, supressing duplicate data that the client has already seen. In effect, a client that will need to reconnect to a server in a situation such as this must implement a mechanism similar to the sliding window protocol used in the stream itself! It treats the connection much like a UDP connection: relatively reliable, but in the limit, not trustworthy. These sorts of problems should make the developer very cautious as to the reliability properties of streams.

In some ways, the inconsistency of the scenario illustrated by Figure 5-2 is even more troubling. As noted above, many applications treat the notification that a stream has broken as evidence that the endpoint has failed. In the illustrated setting, this creates a situation in which all three participants have differing, inconsistent, views of the system membership. Imagine now that the same sort of problem occured in a very large system, with hundreds of component programs, some of which might even have multiple connections between one-another. The lack of coordination between streams means that these can break in almost arbitrary ways (for example, one of two connections between a pair of programs can break, while the other connection remains established), and that almost any imaginable "membership view" can arise. Such a prospect makes it extremely difficult to use streams as a building block in applications intended to be extremely reliable.

The success stories for such an approach are almost entirely associated with settings in which the communications hardware is extremely reliable, so that the failure detection used to implement the streams protocols is actually reasonably accurate, or in which one can easily give up on a transfer, as in the case of the Web (which is built over stream-style protocols, but can always abort a request just by closing its stream connections). Unfortunately, not many local area networks can claim to offer the extreme levels of reliability required for this assumption to accurately approximate the hardware, and this is almost never the case in wide-area networks. And not many applications can get away with just telling the user that the "remote server is not responding or has failed." Thus, streams must be used with the utmost care in constructing reliable distributed applications.

In Chapter 13, we will discuss an approach to handling failure detection that could overcome this limitation. The idea is to introduce a protocol by which the computers in the network maintain agreement on system membership, triggering actions such as breaking a stream connection only if the agreement protocol has terminated and the full membership of the computer system agrees that the endpoint has failed, or at least will be treated as faulty. The approach is known to be practical and is well understood both from a theoretical perspective and in terms of the software needed to support it. Unfortunately, however, the standard streams implementations are based upon widely accepted specifications that *mandate* the use of timeout for failure detection. Thus, the streams implementations available from modern computer vendors will not become consistent in their failure reporting any time soon. Only a widespread call for consistency could possibly lead to revision of such major standards as the ones that specify how the TCP or ISO streams protocol should be implemented.

5.4 RPC over a Stream

It is increasingly common to run RPC protocols over stream protocols such as TCP, to simplify the implementation of the RPC interaction itself. In this approach, the RPC subsystem establishes a stream connection to the remote server and places it into an urgent transmission mode, whereby outgoing data is immediately transmitted to the destination. The reliability mechanisms built into the TCP protocol now subsume the need for the RPC protocol to implement any form of acknowledgement or retransmission policy of its own. In the simplest cases, this reduces RPC to a straightforward request-response protocol. When several threads multiplex the same TCP stream, sending RPC's over it concurrently, a small amount of additional code is needed to provide locking (so that data from different RPC requests is not written concurrently to the stream, which could interleave it in some undesired manner), and to demultiplex replies as they are returned from the server to the client.

It is important to appreciate that the reliability associated with a stream protocol will not normally improve (or even change) the reliability semantics of an RPC protocol superimposed upon it. As we saw above, a stream protocol can report a broken connection under the same conditions where an RPC protocol would fail by timing out, and the underlying stream-oriented acknowledgement and retransmission protocol will not affect these semantics in any useful way. The major advantage of running RPC over a stream is that by doing so, the amount of operating system software needed in support of communication is reduced: having implemented flow control and reliability mechanisms for the stream subsystem, RPC becomes just another application-level use of the resulting operating system abstraction. Such an approach permits the operating system designer to optimize the performance of the stream in ways that might not be possible if the operating system itself were commonly confronted with outgoing packets that originate along different computational paths.

5.5 Related Readings

The best general references are the textbooks by Coulouris *et. al.,* Tanenbaum and Comer: [CDK94, Tan88, Com91, CS93]. On the inconsistency of failure detection in streams: [BG95]. There has been a considerable amount of recent work on optimizing streams protocols (particularly TCP) for high performance network hardware. An analysis of TCP costs, somewhat along the lines of the RPC cost analysis in [SB89], can be found in [CJRS89]. Work on performance optimization of TCP includes [Jac88, Jac90, Kay94, KP93]. A summary of other relevant papers can be found in [Com91, Ten90, BD95]. Other papers included in the biliography of this text include [BMP94, Com91, CS93, CT87, DP93, EBBV95, FJML95, Jac88, Jac90, KC93, KP94, MRTR90, PHMA89, RAAB88a, RAAB88b, RST88, RST89, SDW92, Tan88, CDK94].

6. CORBA and Object-Oriented Environments

With the emergence of object-oriented programming languages, such as Modula and C++, came a recognition that object-orientation could play a role similar to that of the OSI hierarchy, but for complex distributed systems. In this view, one would describe a computing system in terms of the set of objects from which it was assembled, together with the rules by which these objects interact with one another. Object oriented system design became a major subject for research, with many of the key ideas pulled together for the first time by a British research effort, called the Advanced Network Systems Architecture group, or ANSA. In this chapter, we will briefly discuss ANSA, and then focus on a more recent standard, called CORBA, which draws on some of the ideas introduced by ANSA, and has emerged as a widely accepted standard for objected oriented distributed computing.

6.1 The ANSA Project

The ANSA project, headed by Andrew Herbert, was the first systematic attempt to develop technology for modelling complex distributed systems [ANSA89, ANSA91a, ANSA91b]. ANSA stands for the "Advanced Network Systems Architecture", and was intended as a technology base for writing down the structure of a complex application or system and then translating the resulting description into a working version of that system in a process of stepwise refinement.

Abstractly, ANSA consists of a set of "models" that deal with various aspects of distributed systems design and representation problem. The "enterprise" model is concerned with the overall functions and roles of the organizational structure within which the problem at hand is to be solved. For example, an air-traffic control system would be an application within the air-traffic control organization, an "enterprise". The "information" model represents the flow of information within the enterprise; in an air-traffic application this model might describe flight-control status records, radar inputs, radio communication to and from pilots, and so forth. The "computation" model is a framework of programming structures and program development tools that are made available to developers. The model deals with such issues as modularity of the application itself, invocation of operations, paramter passing, configuration, concurrency and synchronization, replication, and the extension of existing languages to support distributed computing. The "engineering" and "technology" models reduce these abstractions to practice, poviding the implementation of the ANSA abstractions and mapping these to the underlying runtime environment and its associated technologies.

In practical terms, most users viewed ANSA as a a set of rules for system design, whereby system components could be described as "objects" with published interfaces. An application with appropriate permissions could obtain a "handle" on the object and invoke its methods using the procedures and functions defined in this interface. The ANSA environment would automatically and transparently deal with such issues as fetching objects from storage, launching programs when a new instance of an object was requested, implementing the object invocation protocols, etc. Moreover, ANSA explicitly included features for overcoming failures of various kinds, using transactional techniques drawn from the database community and process group techniques in which sets of objects are used to implement a single highly available distributed service. We will consider both types of technology in considerable detail in Part III of the text, hence we will not do so here.

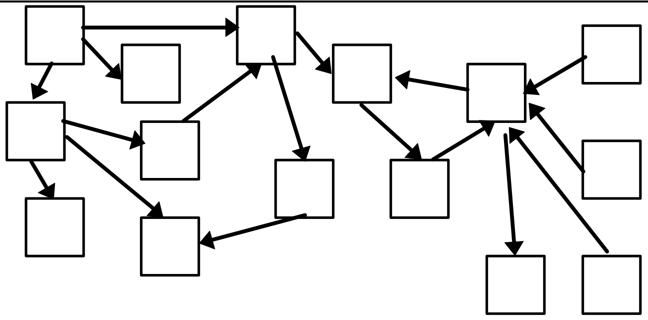


Figure 6-1: Distributed objects abstraction. Objects are linked by object references and the distributed nature of the environment is hidden from users. Access is uniform even if objects are implemented to have special properties or internal structure, such as replication for increased availability or transactional support for persistence. Objects can be implemented in different programming languages but this is invisible to users

ANSA treated the objects that implement a system as the concrete realization of the "enterprise computing model" and the "enterprise information model." These models captured the essense of of the application as a whole, treating it as a single abstraction even if the distributed system as implemented necessarily contained many components. Thus, the enterprise computing model might support the abstraction of a collision avoidance strategy for use by the air-traffic control enterprise as a whole, and the enterprise data model might define the standard data objects used in support of this service. The actual implementation of the service would be reached by a series of refinements in which increasing levels of detail are added to this basic set of definitions. Thus, one passes from the abstraction of a collision avoidance strategy to the more concrete concept of a collision avoidance subsystem located at each of a set of primary sites and linked to one-another to coordinate their actions, and from this notion to one with further refinements that define the standard services composing the collision avoidance system as used on a single air-traffic control workstation, and then still further to a description of how those services could be implemented.

In very concrete terms, the ANSA approach required the designer to write down the sort of knowlege of distributed system structure that, for many systems, is implicit but never encoded in a machine-readable form. The argument was that by writing down these system descriptions, a better system would emerge: one in which the rationale for the structure used was self-documenting, in which detailed information would be preserved about the design choices and objectives that the system carries out, and in this manner the mechanisms for future evolution could be made a part of the system itself. Such a design promotes extensibility and interoperability, and offers a path to system management and control. Moreover, ANSA designs were expressed in terms of objects, whose locations could be anywhere in the network (Figure 6-1), with actual issues of location entering only the design was further elaborated, or in specific situations where location of an object might matter (Figure 6-2). This type of object-oriented, location-transparent design has proved very popular with distributed systems designers.

6.2 Beyond ANSA to CORBA

While the ANSA technology *per se* has not gained a wide following, these ideas have had a huge impact on the view of system design and function adopted by modern developers. In particular, as the initial stages of the ANSA project ended, a new project was started by a consortium of computer vendors. Called the Common Object Request Broker Architecture, CORBA undertakes to advance standards permiting interoperation between complex object-oriented systems potentially built by diverse vendors [OMG91].

Although CORBA undoubtedly drew on the ANSA work, the architecture represents a consensus developed within an industry standards organization called the Object Management Group, or OMG, and differs from the ANSA architecture in many respects. The mission of OMG was to develop architecture standards promoting interoperability between systems developed using object-oriented technologies. In some ways, this represents a less ambitious objective than the task with which ANSA was charged, since ANSA set out both to develop an all-encompassing architectural vision for building enterprise-wide distributed computing systems, and to encorporate reliability technologies into its solutions. However, as CORBA has evolved, it has begun to tackle many of the same issues. Moreover, CORBA reaches well beyond ANSA by defining a very large collection of what are called "services", which are CORBA-based subsystems that have responsibility for specific tasks, such as storing objects or providing reliability, and that have specific interfaces. ANSA began to take on this problem late in the project and did not go as far as CORBA.

At the time of this writing, CORBA was basically a framework for building DCE-like computing environments, but with the priviso that different vendors might offer their own CORBA solutions with differing properties. In principle, adherence to the CORBA guidelines should permit such solutions to interoperate — e.g. a distributed system programmed using a CORBA product from Hewlett Packard should be useful from within an application developed using CORBA products from SUN Microsystems, IBM, or some other CORBA-compliant vendor. Interoperability of this sort, however, is planned for late in 1996, and hence there has been little experience with this specific feature.

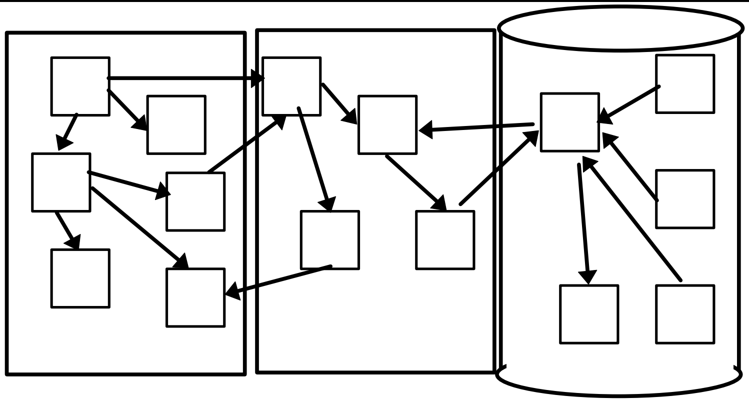


Figure 6-2: In practice, the objects in a distributed system execute on machines or reside in storage servers. The runtime environment works to conceal movement of objects from location to location, activation of servers when they are initially referenced after having been passively stored, fault-tolerance issues, garbage collection, and other issues that span multiple objects or sites.

6.3 OLE-2 and Network OLE

As this book was bring written Microsoft Corporation had just launched a major drive to extend its proprietary object-oriented computing standard, OLE-2, into a distributed object-oriented standard aimed squarely at the internet. The network OLE-2 specification, when it emerges, is likely to have at least as big an impact on the community of PC users as Corba is having on the UNIX community. However, until the standard is actually released, it is impossible to comment upon it. Experts with whom the author has spoken predict that network OLE will be generally similar to Corba, but with a set of system services more closely parallel to the ones offered by Microsoft in its major network products: NT/Server and NT/Exchange. Presumably, these would include integrated messaging, email, and conferencing tools, system-wide security through encryption technologies, and comprehensive support for communicating using multi-media objects. One can only speculate as to more advanced features, such as the group computing technologies and reliability treated in Part III of this text.

6.4 The CORBA Reference Model

The key to understanding the structure of a CORBA environment is the Reference Model [OMG91], which consists of a set of components that a CORBA platform should typically provide. These components are fully described by the CORBA architecture, but only to the level of interfaces used by application developers and functionality. Individual vendors are responsible for deciding how to implement these interfaces and how to obtain the best possible performance; moreover, individual products may offer solutions that differ in offering optional properties such as security, high availability, or special guarantees of behavior that go beyond the basics required by the model.

At a minimum, a CORBA implementation must supply an *Object Request Broker*, or ORB, which is responsible for matching a requestor with an object that will perform its request, using the object reference to locate an appropriate target object. The implementation will also contain translation programs, responsible for mapping implementations of system components (and their IDL's) to programs that can be linked with a runtime library and executed. A set of *Object Services* provide the basic functionality needed to create and use objects: these include such functions as creating, deleting, copying, or moving objects, giving them names that other objects can use to bind to them, and providing security. An interesting service about which we will have more to say below is the *Event Noticication Service* or ENS: this allows a program to register its interest in a class of events. All events in that class are then reported to the program. It thus represents a communication technology different from the usual RPC-style or stream-style of connection. A set of *Common Faciities* contains a collection of standardized applications that most CORBA implementations are expected to support, but that are ultimately optional: these include, for example, standards for system management and for electronic mail that may contain objects. And finally, of course, there are *Application Objects* developed by the CORBA user to solve a particular problem.

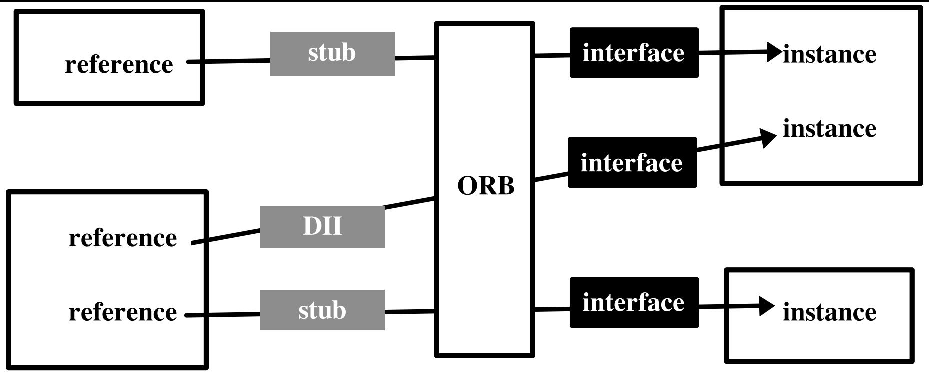


Figure 6-3: The conceptual architecture of CORBA uses an object request broker as an intermediary that directs object invocations to the appropriate object instances. There are two cases of invocations: the static one, which we focus on in the text, and the dynamic invocation interface (DII), which is more complex to use and hence not discussed here. (Source: Shapiro)

In many respects the Object Request Broker is the core of a CORBA implementation. Similar to the function of a communications network or switching system, the ORB is responsible for delivering object invocations that originate in a client program to the appropriate server program, and routing the reply back to the client. The ability to invoke an object, of course, does not imply that the object that was invoked is being used correctly, has a consistent state, or is even the most appropriate object for the application to use. These broader properties fall back upon the basic technologies of distributed computing that are the general topic of this textbook; as we will see, CORBA is a way of *talking about* solutions but not *a specific set of prebuilt solutions*. Indeed, one could say that because CORBA worries about syntax but not semantics, the technology is largely superficial: a veneer around a set of technologies. However, this particular veneer is an important and sophisticated one, and also creates a context within which a principled and standardized approach to distributed systems reliability becomes possible.

For many users, object-oriented computing means programming in C++, although SmallTalk and Ada are also object-oriented languages, and one can develop object-interfaces to other languages like Fortran and Cobol. Nonetheless, C++ is the most widely used language, and is the one we focus on in the examples presented in the remainder of this chapter. Our examples are drawn directly from the "programmer's guide" for Orbix, an extremely popular CORBA technology at the time of this writing.

```
// grid server example for Orbix
// IDL -- in file grid.idl
interface grid {
        readonly attribute short height;
        readonly attribute short width;

        void set(in short n, in short m, in long value);
        void get(in short n, in short m);
};
```

Figure 6-4: IDL interface to a server for a "grid" object coded in Orbix, a popular CORBA-compliant technology.

An example of a CORBA object interface, coded in the Orbix interface defintion language (IDL), is shown in Figure 6-4. This interface publishes the services available from a "grid" server, which is intended to manage two-dimensional tables such as are used in spread-sheets or relational databases. The server exports two read-only values, width and height, which can be used to query the size of a grid object. There are also two operations which can be

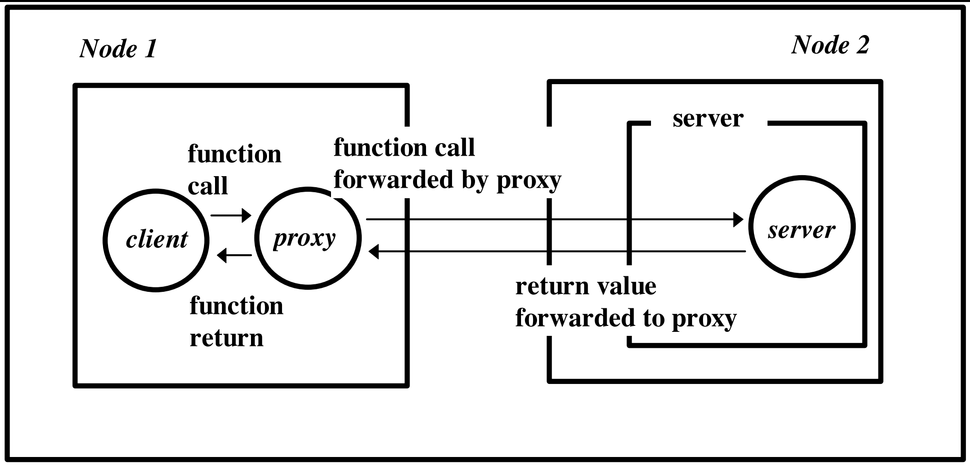

Node 1
Node 2
server
function call
function call forwarded by proxy
client
proxy
server
function return
return value forwarded to proxy

Figure 6-5: Orbix conceals the location of objects by converting remote operations into operations on local proxy objects, mediated by stubs. However, remote access is not completely transparent in standard CORBA applications if an application is designed for reliability. For example, error conditions differ for local and remote objects. Such issues can be concealed by integrating a reliability technology into the CORBA environment, but transparent reliability is not a standard part of CORBA, and solutions vary widely from vendor to vendor.

performed upon the object: "set", which sets the value of an element, and "get" which fetches the value. Set is of type "void", meaning that it does not return a result; get, on the other hand, returns a long integer.

To build a grid server, the user would need to write a C++ program that implements this interface. To do this, the IDL compiler is first used to transform the IDL file into a standard C++ header file in which Orbix defines the information it will need to implement remote invocations on behalf of the client. The IDL compiler also produces two forms of "stub" files, one of which implements the client side of the "get" and "set" operations; the other implements the "server" side. These stub files must be compiled and linked to the respective programs.

If one were to look at the contents of the header file produced for the grid IDL file, one would discover that "width" and "height" have been transformed into functions. That is, when the C++ programmer references an attribute of a grid object, a function call will actually occur into the client-side stub procedures, which can perform an RPC to the grid server to obtain the current value of the attribute.

We say RPC here, but in fact a feature of CORBA is that it provides very efficient support for invocations of local objects, which are defined in the same address space as the invoking program. The significance of this is that although the CORBA IDL shown could be used to access a remote server that handles one or more grid objects, it can also be used to communicate to a completely local instantiation of a grid object, contained entirely in the address space of the calling program. Indeed, the concept goes even further: in Orbix+Isis, a variation of Orbix, the grid server could be replicated using an object group for high availability. And in the most general case, the grid object's clients could be implemented by a server running under some other CORBA-based environment, such as IBM's DSOM product, HP's DOMF, SUN's DOE, Digital Equipment's ObjectBroker, or other object-oriented environments with which CORBA can communicate using an "adapter", such as Microsoft's OLE. CORBA implementations thus have the property that object location, the technology or programming language used to build an object, and even the ORB under which it is running can be almost completely transparent to the user.

```
// C++ code fragment for grid implementation class
#include "grid.hh" // generated from IDL

class grid_i: public gridBOAImpl {
        short     m_height;
        short     m_width;
        long      **m_a;
   public:
        grid_i(short h, short w);   // Constructor
        virtual ~grid_i();          // Destructor
        virtual short width(CORBA::Environment &);
        virtual short height(CORBA::Environment &);
        virtual void set(short n, short m, long value,
                 CORBA::Environment &);
        virtual long get(short n, short m,
                 CORBA::Environment &);
};
```

Figure 6-6: Orbix example of a grid implementation class corresponding to grid IDL.

What exactly would a grid server look like? If we are working in C++, grid would be a C++ program that includes an "implementation class" for grid objects. Such a class is illustrated in Figure 6-6, again drawing on Orbix as a source for our example. The "Environment" parameter is used for error handling with the client. The BOAImpl extention ("gridBOAImpl") designates that this is a Basic Object Adaptor Implementation for the "grid" interface. Figure 6-7 shows the code that might be used to implement this abstract data type.

Finally, our server needs an enclosing framework: the program itself that will execute this code. The code in Figure 6-8 provides this; it implements a single grid object and declares itself to be ready to accept object invocations. The grid object is not named in this example, although it could have been, and indeed the server could be designed to create and destroy grid objects dynamically at runtime.

```
#include “grid_i.h”
#include <iostream.h>

void main() {
        grid_i myGrid(100,100);
        // Orbix objects can be named but this is not
        // needed for this example
        CORBA::Orbix.impl_is_ready();
        cout <<“server terminating” << endl;
}
```

Figure 6-8: Enclosing program to declare a grid object and accept requests upon it.

The user can now declare to Orbix that the grid server is available by giving it a name and storing the binary of the server in a file, the pathname of which is also provided to Orbix. The Orbix “life cycle service” will automatically start the grid server if an attempt is made to access it when it is not running.

```
// Implementation of grid class
#include “grid_i.h”

grid_i::grid_i(short h, short w) {
        m_height = h;
        m_width = w;
        m_a = new long* [h];
        for (int i = 0; i < h; i++)
                m_a[i] = new long [w];
}
grid_i::~grid_i() {
        for (int i = 0; i < m_height; i++)
                delete[ ] m_a[i];
        delete[ ] m_a;
}
short grid_i::width(CORBA::Environment &) {
        return m_width;
}
short grid_i::height(CORBA::Environment &) {
        return m_height;
}
void grid_i::set(short n, short m, long value, CORBA::Environment &) {
        m_a[n][m] = value;
}
void grid_i::get(short n, short m, CORBA::Environment &) {
        return m_a[n][m];
}
```

Figure 6-7: Server code to implement the grid_i class in Orbix.

```
        #include "grid.hh"
        #include <iostream.h>

        void main() {
                grid *p;

                p = grid::_bind(":gridSrv");
                cout << "height is " << p->height() << endl;
                cout << "width is " << p->width() << endl;
                p->set(2, 4, 123);
                cout << "grid(2, 4) is " << p->get(2, 4) << endl;
                p->release();
        }
```

Figure 6-9: Client program for the grid object; assumes that the grid was "registered" under the server name "gridSrv". This example lacks error handling; an elaborated version with error handling appears in Figure 6-10.

CORBA supports several notions of reliability. One is concerned with recovering from failures, for example when invoking a remote server. A second reliability mechanism is provided for purposes of reliable interactions with persistent objects, and is based upon what is called a "transactional" architecture. We discuss transactions elsewhere in this text and will not digress onto that subject at this time. However, the basic purpose of a transactional architecture is to provide a way for applications to perform operations on complex persistent data structures without interfering with other concurrently active but independent operations, and in a manner that will leave the structure intact even if the application program or server fails while it is running. Unfortunately, as we will see in Chapter 21, transactions are primarily useful in applications that are structured as database systems on which programs operate using read and update requests. Such structures are important in distributed systems, but there are many distributed applications that match the model poorly, and for them, transactional reliability is not a good approach.

```
#include "grid.hh"
#include <iostream.h>

void main() {
        grid *p;

        TRY {
                p = grid::_bind(":gridSrv");
        }
        CATCHANY {
                cerr << "bind to object failed" << endl;
                cerror << "Fatal exception " << IT_X << endl;
                exit(1);
        }
        TRY {
                cout << "height is " << p->height() << endl;
        }
        CATCHANY {
                cerr << "call to height failed" << endl;
                cerror << "Fatal exception " << IT_X << endl;
                exit(1);
        }
        ... etc ...
}
```

Figure 6-10: Illustration of Orbix error handling facility. Macros are used to catch errors; if one occurs, the error can be caught and potentially worked around. Notice that each remote operation can potentially fail, hence exception handling would normally be more standardized. A handler for a high availability application would operate by rebinding to some other server capable of providing the same functionality. This can be concealed from the user, which is the approach used in systems like Orbix+Isis or Electra, a CORBA technology layered over the Horus distributed system.

Outside of its transactional mechanisms, however, CORBA offers relatively little help to the programmer. For example, Orbix can be notified that a server application can be run on one of a number of machines. When a client application attempts to use the remote application, Orbix will automatically attempt to bind to each machine in turn, selecting at random the first machine which confirms that the server application is operational. However, Orbix does not provide any form of automatic mechanisms for recovering from the failure of such a server after the binding is completed. The reason for this is that a client process that is already communicating with a server may have a complex state that reflects information specific to that server, such as cached records with record identifiers that came from the server, or other forms of data that differ in specific ways even among servers that are, broadly speaking, able to provide the same functionality. To rebind the client to a new server, one would somehow need to refresh, rebuild, or roll back this server-dependent state. And doing so is potentially very hard; at a minimum, considerable detailed knowledge of the application will be required.

The same problems can also arise in the server itself. For example, consider a financial trading service, in which the prices of various stocks are presented, and which is extremely dynamic due to rapidly changing market data. The server may need to have some form of setup that it uses to establish a client profile, and may have an internal state that reflects the events that have occured since the client first bound to it. Even if some other copy of the server is available and can provide the same services, there could be a substantial time lag when rebinding and there may be a noticable discontinuity if the new server, lacking this "state of the session", starts its financial computations from the current stream of incoming data. Such events will not be transparent to the client using the server and it is unrealistic to try and hide them.

The integration of of a wider spectrum of reliability enhancing technologies with CORBA represents an important area for research and commercial development, particularly if reliability is taken in the broad sense of security, fault-tolerance, availability, and so forth. High performance, commercially appealing products will be needed that demonstrate the effectiveness of the architectural features that result: when we discuss transactions on distributed objects, for example, we will see that merely supporting transactions through an architecture is not likely to make users happy. Even the execution of transactions on objects raises deeper issues that would need to be resolved for such a technology to be accepted as a genuinely valid reliability enhancing tool. For example, the correct handling of a transactional request by a non-transactional service is unspecified in the architecture.

More broadly, CORBA can be viewed as the ISO hierarchy for object oriented distributed computing: it provides us with a framework within which such systems can be described and offers ways to interconnect components without regard for the programming language or vendor technologies used in developing them. Exploiting this to achieve critical reliability in distributed settings, however, stands as a more basic technical challenge that CORBA does not directly address. CORBA tells us how to structure and present these technologies, but not how to build them.

In Chapters 13-18 we will discuss process group computing and associated technologies. The Orbix product is unusual in supporting a reliability technology, Orbix+Isis [O+I95], based on process groups, and that overcomes these problems. Such a technology is a form of replication service, but the particular one used to implement Orbix+Isis is extremely sophisticated in comparison to the most elementary forms of replication service, and the CORBA specifications for this area remain very tentative. Thus, Orbix+Isis represents a good response to these reliability concerns, but the response is specific to Orbix and may not correspond to a broader CORBA response to the issues that arise. We will be studying these issues in more detail later, and hence will not present the features of Orbix+Isis here.

6.5 TINA

TINA-C is the "Telecommunications Information Network Architecture Consortium", and is an organization of major telecommunications services providers that set out to look at ANSA and CORBA-like issues from a uniquely telecommunications perspective [TINA96]. At the time of this writing, TINA was in the process of specifying a CORBA-based architecture with extensions to deal with issues of realtime communication, reliability, and security, and with standard telecommunications-oriented services that go beyond the basic list of CORBA services implemented by typical CORBA-compatible products. The TINA varient of CORBA is expected by many to have a dramatic impact on the telecommunications industry. Specific products aimed at this market are already being announced: Chorus Systemes' COOL-ORB is a good example of a technology that focuses on the realtime, security and reliability needs of the telecommunications industry by providing a set of object services and architectural features that reflect embedded applications typical of large-scale telecommunications applications, in contrast to the more general computing market to which products like Orbix appear to be targetted.

6.6 IDL and ODL

IDL is the language used to define an object interface (in the TINA standard, there is an ODL language that goes beyond IDL is specifying other attributes of the object in addition to its interface). CORBA defines an IDL for the various languages that can be supported: C++, SmallTalk, Ada95, and so forth. The most standard of these is the IDL for C++, and the examples given above are expressed in C++ for that reason. However, expanded use of IDL for other programming languages is likely in the future.

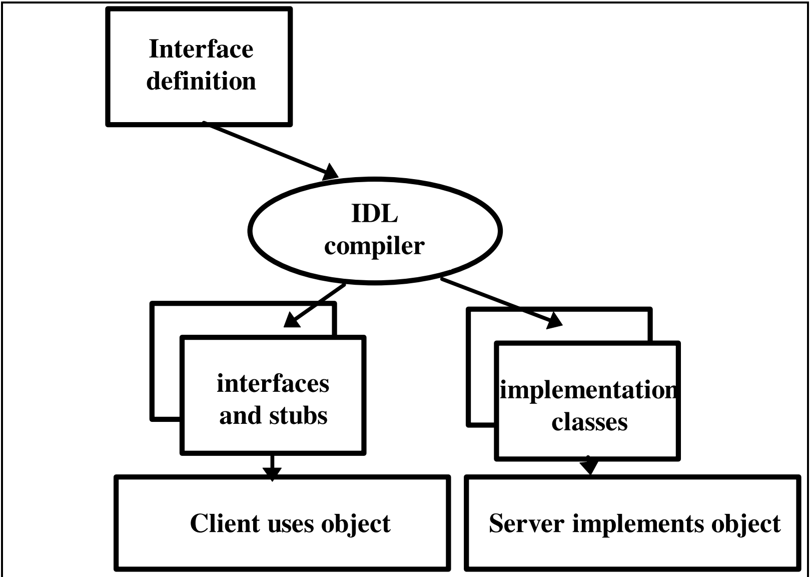


Figure 6-11: From the interface defintion, the IDL compiler creates stub and interface files which are used by clients that invoke the object and by servers that implement it.

The use of C++ programs in a CORBA environment can demand a high level of sophistication in C++ programming. In particular, the operator overload functionality of C++ can conceal complex machinery behind deceptively simple interfaces. In a standard programming language one expects that an assignment statement such as **a = b** will execute rapidly. In C++ such an operation may involve allocation and initialization of a new abstract object and a potentially costly copying operation. In CORBA such an assignment may involve costly remote operations on a server remote from the application program that executes the assignment statement. To the programmer, CORBA and C++ may appear as a mixed blessing: through the CORBA IDL, operations such as assignment and value references can be transparently extended over a distributed environment, which can seem like magic. But the magic is potentially tarnished by the discovery that a single assignment might now take seconds (or hours) to complete!

Such observations point to a deficiency in the CORBA IDL language and, perhaps, the entire technology as currently conceived. IDL provides no features for specifying <u>*behaviors*</u> of remote objects that are desirable or undesireable consequences of distribution. There is no possibility of using IDL to indicate a performance property (or cost, in the above example), or to specify a set of fault-tolerance guarantees for an object that differ from the ones normally provided in the environment. Synchronization requirements or assumptions made by an object, or guarantees ofered by the client, cannot be expressed in the language. This missing information, potentially needed for reliability purposes, can limit the ability of the programmer to fully specify a complex distributed system, while also denying the user the basic information needed to validate that a complex object is being used correctly.

One could argue that the IDL should be limited to specification of the interface to an object, and that any behavioral specifications would be managed by other types of services. Indeed, in the case of the Life Cycle service, one has a good example of how the CORBA community approaches this problem: the life-cycle aspects of an object specification are treated as a special type of data managed by this service, and are not considered to be a part of the object interface specification. Yet the author would is convinced that this information often belongs in the interface specification, in the sense that these types of properties may have direct implications for the user that accesses the object and may be information of a type that is important in establishing that the object is being used correctly. That is, the author is convinced that the specification of an object involves more than the specification of its interfaces, and indeed that the interface specification involves more than just the manner in which one invokes the object. In contrast, the CORBA community considers behavior to be orthogonal to interface specification, and hence relegates behavioral aspects of the object's specification to the special-purpose services directly concerned with that type of information. Unfortunately, it seems likely that much basic research will need to be done before this issue is addressed in a convincing manner.

6.7 ORB

An Object Request Broker, or ORB, is the component of the runtime system that binds client objects to the server objects they access, and that interprets object invocations at runtime, arranging for the invocation to occur on the object that was referenced. (CORBA is thus the OMG's specification of the ORB and of its associated services). ORB's can be thought of as "switching" systems through which invocation messages flow. A fully compliant CORBA implementation supports interoperation of ORB's with one-another over TCP connections, using what is called the GIOP protocol. In such an interoperation mode, any CORBA server can potentially be invoked from any CORBA client, even if the server and client were built and are operated on different versions of the CORBA technology base.

Associated with the ORB are a number of features designed to simplify the life of the developer. An ORB can be programmed to automatically launch a server if it is not running when a client accesses it (this is called "factory" functionality), and can be asked to automatically filter invocations through user-supplied code that automates the handling of error conditions or the verification of security properties. The ORB can also be programmed to make an intelligent choice of object if many objects are potentially capable of handling the same request; such a functionality would permit, for example, load balancing within a group of servers that replicate some database.

6.8 Naming Service

A CORBA naming service is used to bind names to objects. Much as a file system is organized as a set of directories, the CORBA naming architecture defines a set of *naming contexts*, and each name is interpreted relative to the naming context within which that name is registered. The CORBA naming architecture is potentially a very general one, but in practice, many applications are expected to treat it as an object-oriented generalization of a traditional naming hierarchy. Such applications would build hierarchical naming context graphs (directory trees), use ascii style pathnames to identify objects, and standardize the sets of attributes stored for each object in the naming service (size, access time, modification time, owner, permissions, etc.) The architecture, however, is sufficiently flexible to allow a much broader notion of names and naming.

A CORBA name should not be confused with an object reference. In the CORBA architecture, an object reference is essentially a pointer to the object. Although a reference need not include specific location information, it does include enough information for an ORB to find a path to the object, or to an ORB that will know how to reach the object. Names, in contrast, are symbolic ways of naming these references. By analogy to a UNIX file system, a CORBA object name is like a pathname (and similar to a pathname, more than one name can refer to the same object). A CORBA object reference is like a UNIX *vnode* reference: a machine address and an identifier for a file *inode* stored on that machine. From the name one can lookup the reference, but this is a potentially costly operation. Given the object reference one can invoke the object, and this (one hopes) will be quite a bit cheaper.

6.9 ENS

The CORBA Event Notification Service or ENS provides for notifications of asynchronous "events" to applications that register an interest in those events by obtaining a handle, to which events can be posted and on which events can be received. Reliability features are optionally supplied. The ENS is best understood in terms of what is called the *publish/subscribe* communications architecture[5]. In this approach, messages are produced by *publishers* which label each new message using a set of *subjects* or *attributes*. Separately, applications that wish to be informed when events occur on a given subject will *subscribe* to that subject or will poll for messages relating to the subject. The role of the ENS is to reliably bind the publishers to the subscribers, ensuring that even though the publishers do not know who the subscribers will be, and vice versa, messages are promptly and reliably delivered to them.

Two examples will make the value of such a model more clear. Suppose that one were using CORBA to implement a software architecture for a large brokerage system or a stock exchange. The ENS for such an environment could be used to broadcast stock trades as they occur. The events in this example would be "named" using the stock and bond names that they describe. Each broker would subscribe to the stocks of interest, again using these subject names, and the application program would then receive incoming quotes and display them to the screen. Notice that the publisher program can be developed without knowing anything about the nature of the applications that will use the ENS to monitor its outputs: it need not have compatible types or interfaces except with respect to the events that are exchanged between them. And the subscriber, for its part, does not need to be bound to a particular publisher: if a new data source of interest is developed it can be introduced into the system without changing the existing architecture.

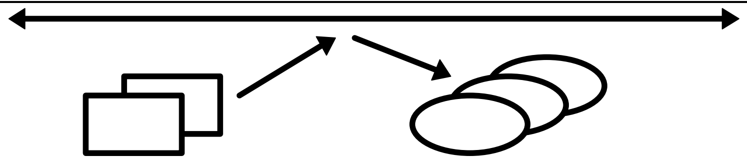


Figure 6-12: The CORBA ENS is a form of message "bus" that supports a publish/subscribe architecture. The sources of events (blue) and consumers (violet) need not be explicitly aware of one another, and the sets can change dynamically. A single object can produce or consume events of multiple types, and in fact an object can be both producer and consumer.

A second example of how the ENS can be useful would arise in system management and monitoring. Suppose that an application is being developed to automate some of the management functions that arise in a very large VLSI fabrication facility. As time goes by, the developers expect to add more and more sources of information and introduce more and more applications that use this information to increase the efficiency and productivity of the factory. An ENS architecture facilitates doing so, because it permits the developers to separate the *information architecture* of their application from its *implementation architecture*. In such an example, the information architecture is the structure of the ENS event "space" itself: the subjects under which events may be posted, and the types of events that can arise in each subject. The sources and consumers of the events can be introduced later, and will in general be unaware of one-another. Such a design preserves tremendous flexibility and facilitates an evolutionary design for the system. After basic functionality is in place, additional functions can be introduced in a gradual way and without disrupting existing software. Here, the events would be named according to the aspect of factory function to which they relate: status of devices, completion of job steps, scheduled downtime, and so forth. Each application program would subscribe to those classes of events relevant to its task, ignoring all others by not subscribing to them.

[5] It should be noted however that the ENS lacks the sort of subject "mapping" facilities that are central to many publish-subscribe message-bus architectures, and is in this sense a more primitive facility than some of the message bus technologies that will be discussed later in the text, such as the Teknekron Information Bus (TIB).

Not all CORBA implementations include the ENS. For example, the basic Orbix product described above lacks an ENS, although the Orbix+Isis extention makes use of a technology called the Isis Message Distribution Service to implement ENS functionality in an Orbix setting. This, in turn, was implemented using the Isis Toolkit, which we will discuss in more detail in Chapter 17.

6.10 Life Cycle Service

The Life Cycle Service or LCS standardizes the facilities for creating and destroying objects, and for copying them or moving them within the system. The service includes a *factory* for manufacturing new objects of a designated type. The Life Cycle Service is also responsible for scheduling backups, periodically compressing object repositories to reclaim free space, and initiating other "life cycle" activities. To some degree, the service can be used to program object-specific management and supervisory functions, which may be important to reliable control of a distributed system. However, there is at present limited experience with life cycle issues for CORBA objects, hence these possibilities remain an area for future development and research.

6.11 Persistent Object Service

The Persistent Object Service or POS is the CORBA equivalent of a file system. This service maintains collections of objects for long-term use, organizing them for efficient retrieval and working closely with its clients to give application-specific meanings to the consistency, persistency, and access control restrictions implemented within the service. This permits the development of special-purpose POS's, for example to maintain databases with large numbers of nearly identical objects organized into relational tables, as opposed to file system-style storage of very irregular objects, etc.

6.12 Transaction Service

Mentioned earlier, the transaction service is an embedding of database-style transactions into CORBA architecture. If implemented, the service provides a *concurrency control* service for synchronizing the actions of concurrently active transactions, *flat* and (optionally) *nested* transactional tools, and special-purpose persistent object services that implement the transactional *commit* and *abort* mechanisms. The Transaction Service is often used with the *relationship service* which tracks relationships among sets of objects, for example if they are grouped into a database or some other shared data structure. We will be looking at the transactional execution model in Section 7.4 and in Chapter 21.

6.13 Inter-Object Broker Protocol

The IOB, or Inter-Object Broker Protocol, is a protocol by which ORB's can be interconnected. The protocol is intended for use both between geographically dispersed ORB's from a single vendor, and to permit interoperation between ORB's developed independently by different vendors. The IOB includes definitions of a standard object reference data structure by which an ORB can recognize a foreign object reference and redirect it to the appropriate ORB, and definitions of the messages exchanged between ORB's for this purpose. The IOB is defined for use over a TCP channel; should the channel break or not be available at the time a reference is used, the corresponding invocation will return an exception.

6.14 Future CORBA Services

The evolution of CORBA continues to advance the coverage of the architecture, although not all vendor products will include all possible CORBA services. Future services now under discussion include archival storage for infrequently accessed objects, backup/restore services, versioning services, data interchange and internationalization services, logging and recovery services, replication services for promoting high availability, and security services. Real-time services are likely to be added to this list in a future round of CORBA enhancements, as will other sorts of reliability and robustness-enhancing technologies.

6.15 Properties of CORBA Solutions

While the CORBA architecture is impressive in its breadth, the user should not be confused into believing that CORBA therefore embodies solutions for the sorts of problems that were raised in the first chapters of this book, or the ones we consider in Chapter 15. To understand this point, it is important to again stress that CORBA is a somewhat "superficial" technology in specifying the way things *look* but not *how they should be implemented*. In language terminology, CORBA is concerned with syntax but not semantics. This is a position that the OMG adopted intentionally, and the key players in that organization would certainly defend it. Nonetheless, it is also a potentially troublesome aspect of CORBA, in the sense that a correctly specified CORBA application may still be underspecified (even in terms of the interface to the objects) for purposes of verifying that the objects are used correctly or for predicting the behavior of the application.

Among other frequently cited concerns about CORBA is that the technology can require extreme sophistication on the part of developers, who must at a minimum understand exactly how the various object classes operate and how memory management will be performed. Lacking such knowledge, which is not an explicit part of the IDL, it may be impossible to use a distributed object efficiently. Even experts complain that CORBA exception handling can be very tricky. Moreover, in very large systems there will often be substantial amounts of old code that must interoperate with new solutions. Telecommunications systems are sometimes said to involve millions or tens of millions of lines of such software, perhaps written in outmoded programming languages or incorporating technologies for which source code is not available. To gain the full benefits of CORBA, however, there is a potential need to use CORBA all through a large distributed environment. This may mean that large amounts of "old code" must somehow be retrofitted with CORBA interfaces and IDL's, neither a simple nor an inexpensive proposition.

The reliability properties of a particular CORBA environment depend on a great number of implementation decisions that can vary from vendor to vendor, and often will do so. Indeed, CORBA is promoted to vendors precisely because it creates a "level playing field" within which their products can interoperate but compete: the competition would revolve around this issue of relative performance, reliability, or functionality guarantees. Conversely, this implies that individual applications cannot necessarily count upon reliability properties of CORBA if they wish to maintain a high degree of portability: such applications must in effect assume the least common denominator. Unfortunately, this least level of guarantees, in the CORBA architectural specification is quite weak: invocations and binding requests can fail, perhaps in inconsistent ways, corresponding closely to the failure conditions we identified for RPC protocols that operate over standard communication architectures. Security, being optional, must be assumed not to be present. Thus, CORBA creates a framework within which reliability technologies can be standardized, but as currently positioned, the technology base is not necessarily one that will encourage a new wave of reliable computing systems.

On the positive side, CORBA vendors have shown early signs of using reliability as a differentiator for their products. Iona's Orbix product is offered with a high availability technology based on process group computing (Orbix+Isis [O+I95]) and a transactional subsystem based on a popular transactional technology (Orbix+Tuxedo [O+T95]). Other major vendors are introducing reliability tools of their own. Thus, while reliability may not be a standard property of CORBA applications, and may not promote portability between CORBA platforms, it is at least clear that CORBA was conceived with the possibility of supporting reliable computing in mind. Most of the protocols and techniques discussed in the remainder of this textbook are compatible with CORBA in the sense that they could be used to implement standard CORBA reliability services, such as its replication service or the event notification service.

6.16 Related Readings

On the ANSA project and architecture: [ANSA89, ANSA91a, ANSA91b]. Another early effort in the same area was Chronus: [GDS86, STB86]. On Corba: [OMG91] and other publications available from the Object Management Group, a standards organization; see the Web page [http://www.omg.org]. For the Corba products cited, such as Orbix: the reader should contact the relevant vendor. On TINA: [TINA96]. On DCE [DCE94], and OLE-2 [OLE94]; material discussing network OLE had not yet been made available at the time of this writing.

7. Client-Server Computing

7.1 Stateless and Stateful Client-Server Interactions

Chapters 4 to 6 focused on the communication protocols used to implement RPC and streams, and on the semantics of these technologies when a failure occurs. Independent of the way that a communication technology is implemented, however, is the question of how the programming paradigms that employ it can be exploited in developing applications, particularly if reliability is an important objective. In this chapter, we examine client-server computing technologies, assuming that the client-server interactions are by RPC, perhaps implemented directly, and perhaps issued over streams. Our emphasis is on the interaction between architectural issues and the reliability properties of the resulting solutions. This topic will prove particularly important when we begin to look closely at the Web, which is based on what is called a "stateless" client-server computing paradigm, implemented over stream connections to Web servers.

7.2 Major Uses of the Client-Server Paradigm

The majority of client-server applications fall into one of two categories, which can be broadly characterized as being the file-server or *stateless* architectures, and the database-styled transactional or *stateful* architectures. Although there are a great many client-server systems that neither manage files nor any form of database, most such systems share a very similar design with one or the other of these. Moeover, although there is an important middle ground consisting of stateful distributed architectures that are not transactional, these sorts of applications have only emerged recently and continue to represent a fairly small percentage of the client-server architectures found in real systems. Accordingly, by focusing on these two very important cases, we will establish some basic intuitions about the broader technology areas of which each is representative, and of the state of practice at the time of this writing. In Part III of the text we will discuss stateful distributed systems architectures in more general terms and in much more detail, but in doing so will also move away from the "state of practice" as of the mid 1990's into technologies that may not be widely deployed until late in the decade or beyond.

A stateless client-server architecture is one in which neither the clients nor the server need to maintain accurate information about one-another's status. This is not to say that the clients cannot "cache" information obtained from a server, and indeed the use of caches is one of the key design features that permit client-server systems to perform well. However, such cached information is understood to be potentially stale, and any time an operation is performed on the basis of data from the cache, some sort of validation scheme must be used to ensure that the outcome will be correct even if the cached data has become invalid.

More precisely, a stateless client-server architecture has the property that *servers do not need to maintain an accurate record of their current set of clients, and can change state without engaging in a protocol between the server and its clients*. Moreover, when such state changes occur, *correct behavior of the clients is not affected*. The usual example of a stateless client-server architecture is one in which a client caches records that it has copied from a name server. These records might, for example, map from ascii names of bank accounts to the IP address of the bank server maintaining that account. Should the IP address change (i.e. if an account is transferred to a new branch that uses a different server), a client that tries to access that account will issue a request to the wrong server. Since the transfer of the account is readily detected, this request will fail, causing the client to refresh its cache record by looking up the account's new location; the request can then be reissued and should now reach the correct server. This is illustrated in Figure 7-1. Notice that the use of cached data is transparent to (concealed from) the

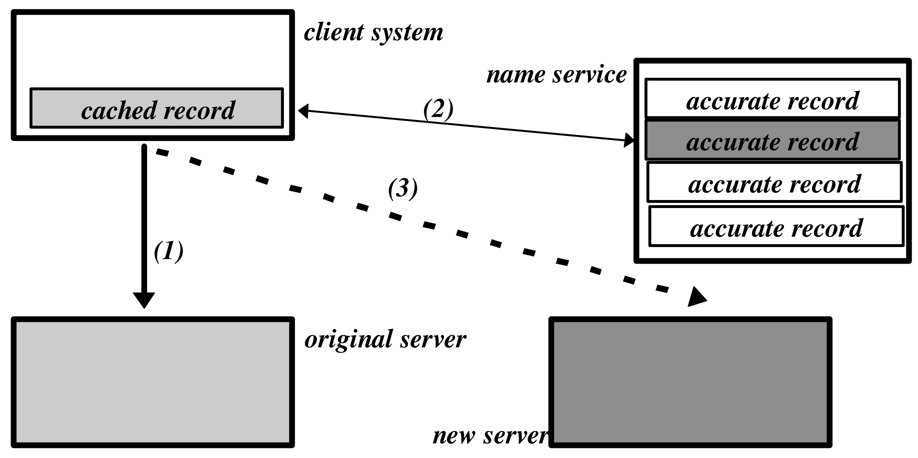


Figure 7-1: In this example, a client of a banking database has cached the address of the server handling a specific account. If the account is transferred, the client's cached record is said to have become "stale". Correct behavior of the client is not compromised, however, because it is able to detect staleness and refresh the cached information at runtime. Thus, if an attempt is made to access the account, the client will discover that it has been transferred (step 1) and will look up the new address (step 2), or be told the new address by the original server. The request can then be reissued to the correct server (step 3). The application program will benefit from improved performance when the cached data is correct, which is hopefully the normal case, but never sees incorrect or inconsistent behavior if the cached data is incorrect. The key to such an architecture lies in the ability to detect that the cached data has become stale when attempting to use it, and in the availability of a mechanism for refreshing the cache transparent to the application.

application program, which benefits through improved performance when the cached record is correct, but is unaffected if an entry becomes stale and must be refreshed at the time it is used.

One implication of a stateless design is that the server and client are independently responsible for ensuring the validity of their own states and actions. In particular, the server makes no promises to the client, except that the data it provides was valid at the time it was provided. The client, for its part, must carefully protect itself against the possibility that the data it obtained from the server has subsequently become stale.

Notice that a stateless architecture does not imply that there is no form of "state" shared between the server and its clients. On the contrary, such architectures often share state through caching, as seen in the above example. The fundamental property of the stateless paradigm is that correct function doesn't require that the server keep track of the clients currently using it, and that the server can change data (reassigning the account in this example) without interacting with the clients that may have cached old copies of the data item. To compensate for this, the client side of such a system will normally include a mechanism for detecting that data has become stale, and for refreshing it when an attempt is made to use such stale data in an operation initiated by the client.

The stateless client-server paradigm is one of the most successful and widely adopted tools for building distributed systems. File servers, perhaps the single most widely used form of distributed system, are typically based on this paradigm. The Web is based on stateless servers, for example, and this is often cited as one of the reasons for its rapid success. One could conclude that this pattern repeates the earlier success of NFS: a stateless file system protocol that was tremendously successful in the early 1980's and fueled the success of Sun Microsystems, which introduced the protocol and was one of the first companies to invest heavily in it. Moreover, many of the special purpose servers developed for individual applications employ a stateless approach. However, as we will see below, stateless architectures also carry a price: systems built this way are often have limited reliability or consistency guarantees.

It should also be mentioned that stateless designs are a principle but not an absolute rule. In particular, there are many file systems (we will review some below) that are stateless in some ways but make use of coherently shared state in other ways. In this section we will call such designs stateful, but the developers of the systems themselves might consider that they have adhered to the principle of a stateless design in making "minimum" use of coherently shared state. Such a philosophy recalls the end-to-end philosophy of distributed systems design, in which communications semantics are left to the application layer except when strong performance or complexity arguments can be advanced in favor of putting stronger guarantees into an underlying layer. We will not take a position on this issue here (or if we do, perhaps it is our position that if an architecture guarantees the consistency of distributed state, then it is stateful!). However, to the degree that there is a design philosophy associated with stateless client-server architectures, it is probably most accurate to say that it is one that "discourages" the use of consistently replicated or coherently shared state except where such state is absolutely necessary.

In contrast, a stateful architecture is one in *which information is shared between the client and server in such a manner that the client may take local actions under the assumption that this information is correct*. In the example of Figure 7-1, this would have meant that the client system would never need to retry a request. Clearly, to implement such a policy, the database and name mapping servers would need to track the set of clients possessing a cached copy of the server for a record that is about to be transferred. The system would need to somehow lock these records against use during the time of the transfer, or to invalidate them so that clients attempting to access the transferred record would first look up the new address. The resulting protocol would guarantee that if a client is permitted to access a cached record, that record will be accurate, but it would do so at the cost of complexity (in the form of the protocol needed when a record is transferred) and delay (namely, delays visible to the client when trying to access a record that is temporarily locked, and/or delays within the servers when a record being transferred is found to be cached at one or more clients).

Even from this simple example, it is clear that stateful architectures are potentially complex. If one indeed believed that the success of the NFS and Web was directly related to their statelessness, it would make sense to conclude that stateful architectures are inherently a bad idea: the theory would be that anything worth billions of dollars to its creator must be good. Unfortunately, although seductive, such a reaction also turns out to implicitly argue against reliability, coherent replication, many kinds of security, and other "properties" that are inherently stateful in their formulation. Thus, while there is certainly a lesson to be drawn here, the insights to be gained are not shallow ones.

Focusing on our example, one might wonder when, if ever, coherent caching is be desirable. Upon close study, the issue can be seen to be a tradeoff between performance and necessary mechanism. It is clear that a client system with a coherently cached data item will obtain performance benefits by being able to perform actions correctly using local data (hence, avoiding a round-trip delay over the network), and may therefore be able to guarantee some form of "real time" response to the application. For applications in which the cost of communicating with a server is very high, or where there are relatively strict real-time constraints, this could be an extremely important guarantee. For example, an air-traffic controller contemplating a proposed course change for a flight would not tolerate long delays while checking with the database servers in the various air sectors that flight will traverse. Similar issues are seen in many real-time applications, such as computer assisted conferencing systems and multimedia playback systems. In these cases one might be willing to accept additional mechanism (and hence complexity) as a way of gaining a desired property. A stateless architecture will be much simpler, but cached data may be stale and hence cannot be used in the same ways.

Another factor that favors the use of stateful architectures is the need to reduce load on a heavily used server. If a server is extremely popular, coherent caching offers the possibility of distributing the data it manages, and hence the load upon it, among a large number of clients and intermediate nodes in the network. To the degree that clients manage to cache the right set of records, they avoid access to the server and are able to perform what are abstractly thought of as distributed operations using purely local computation. A good example of a setting where this is becoming important is the handling of popular Web documents. As we will see when we discuss the Web, its architecture is basically that of a file server, and heavy access to a popular document can therefore result in enormous load on the unfortunate server on which that document is stored. By caching such documents at other servers, a large number of potential sources ("shadow copies") of the document can be created, permitting the load of servicing incoming requests to be spread over many copies. But if the document in question is updated dynamically, one now faces a coherent caching question.

If records can be coherently cached *while preserving the apparent behavior of a single server*, the performance benefits and potential for ensuring that real-time bounds will be satisfied can be immense. In distributed systems where reliability is important, it is therefore common to find that which coherent caching is beneficial or even necessary. The key issue is to somehow package the associated mechanisms to avoid paying this complexity cost repeatedly. This is one of the topics that will be discussed in Chapter 15 of this textbook.

There is, however, a second way to offer clients some of the benefits of stateful architecture, but without ensuring that remotely cached data will be maintained in a coherent state. The key to this alternative approach is to use some form of abort or "back out" mechanism to roll back actions taken by a client on a server, under conditions where the server detects that the client's state is inconsistent with its own state, and to force the client to roll back its own state and, presumably, retry its operation with refreshed or corrected data. This underlies the transactional approach to building client-server systems. As noted above, transactional database systems are the most widely used of the stateful client-server architectures.

The basic idea in a transactional system is that the client's requests are structured into clearly delimited transactions. Each transaction begins, encompasses a series of *read* and *update* operations, and then ends by *commiting* in the case where the client and server consider the outcome to be successful, or *aborting* if either client or server has detected an error. An aborted transaction is backed out both by the server, which erases any effects of the transaction, and by the client, which will typically restart its request at the point of the original "begin", or report an error to the user and leave it to the user to decide if the request should be retried. A transactional system is one that supports this model, guaranteeing that the results of committed transactions will be preserved and that aborted transactions will leave no trace.

The connection between transactions and "statefulness" is as follows. Suppose that a transaction is running, and a client has read a number of data items and issued some number of updates. Often it will have locked the data items in question for reading and writing, a topic we discuss in more detail in Chapter 21. These data items and locks can be viewed as a form of shared state between the client and the server: the client basically trusts the server to ensure that the data it has read is valid until it commits or aborts and releases the locks that it holds. Just as our cached data was copied to the client in the earlier examples, all of this information can be viewed as knowledge of the server's state that the client caches. And the relationship is mutual: the server, for its part, holds an image of the client's state in the form of updates and locks it holds on behalf of the partially completed transactions.

Now, suppose that something causes the server's state to become inconsistent with that of the client, or vice versa. Perhaps the server crashes and then recovers, and in this process some information that the client had provided to the server is lost. Or, perhaps it becomes desirable to change something in the database without waiting for the client to finish its transaction. In a stateless architecture we would not have had to worry about the state of the client. In a transactional implementation of a stateful architecture, on the other hand, the server can exploit the abort feature by arranging that the client's transaction be aborted, either immediately, or later when the client tries to commit it. This frees the server from needing to worry about the state of the client. In effect, an abort or rollback mechanism can be used as a tool by which a stateful client-server system is able to recover from a situation where the client's view of the state shared with the server has been rendered incorrect.

In the remainder of this chapter, we review examples of stateless file-server architectures from the research and commercial community, stateful file-server architectures (we will return this topic in a much more general way in Chapter 15), and stateful transactional architectures as used in database systems. As usual, our underlying emphasis is on reliability implications of these architectural alternatives.

7.3 Distributed File Systems

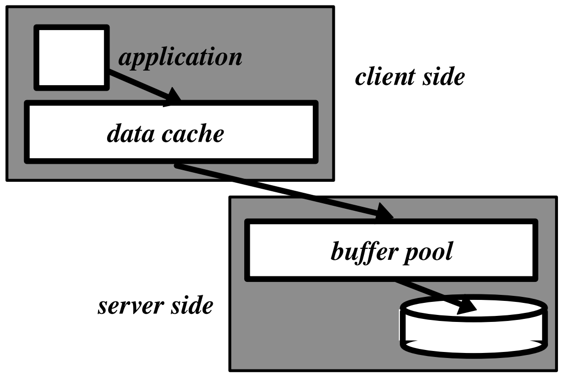


Figure 7-2: In a stateless file system architecture, the client may cache data from the server. Such a cache is similar in function to the server's buffer pool, but is not guaranteed to be accurate. In particular, if the server modifies the data that the client has cached, it has no record of the locations at which copies may have been cached, and no protocol by which cached data can be invalidated or refreshed. The client side of the architecture will often include mechanisms that partially conceal this limitation, for example by validating that cached file data is still valid at the time a file is opened. In effect, the cached data is treated as a set of "hints" that are used to improve performance but should not be trusted in an absolute sense.

We have discussed the stateless approach to file server design in general terms. In this section, we look at some specific file system architectures in more detail, to understand the precise sense in which these systems are stateless, how their statelessness may be visible to the user, and the implications of statelessness on file system reliability.

Client-server file systems normally are structured as shown in Figure 7-2. Here, we see that the client application interacts with a cache of file system blocks and file descriptor objects maintained in the client workstation. In UNIX, the file descriptor objects are called *vnodes* and are basically server-independent representations of the *inode* structures used within the server to track the file. In contrast to an inode, a vnode ("virtualized inode") has a unique identifier that will not be reused over the lifetime of the server, and omits detailed information such as block numbers. In effect, a vnode is an access key to the file, obtained by the client's file subsystem during the file open protocol, and usable by the server to rapidly validate access and locate the corresponding inode.

On the server side, the file system is conventionally structured, but is accessed by messages containing a vnode identifier and an offset into the file, rather than block by block. The vnode identifier is termed a *file handle* whereas the vnode itself is a representation of the contents of the inode, omitting information that is only useful on the server itself but including information that the client application programs might request using UNIX *stat* system calls.

The policies used for cache management differ widely within file systems. NFS uses a write-through policy when updating a client cache, meaning that when the client cache is changed, a write operation is immediately initiated to the NFS server. The client application is permitted to resume execution before the write operation has been completed, but because the cache has already been updated in this case, programs running on the client system will not normally observe any sort of inconsistency. NFS issues an *fsync* operation when a file is closed, causing the close operation to delay until all of these pending write operations have completed. The effect is to conceal the effects of asynchronous writes from other applications executed on the same client computer. Users can also invoke this operation, or the *close* or *fflush* operations, which also flush cached records to the server.

In NFS, the server maintains no information at all about current clients. Instead, the file handle is created in such a manner that mere possession of a file handle is considered proof that the client successfully performed an open operation. In systems where security is an issue, NFS uses digital signature technology to ensure that only legitimate clients can obtain a file handle, and that they can do so only through the legitimate NFS protocol. These systems often require that file system handles be periodically reopened, so that the lifetime of encryption keys can be limited.

On the client side, cached file blocks and file handles in the cache represent the main form of state present in an NFS configuration. The approach used to ensure that this information is valid represents a compromise between performance objectives and semantics. Each time a file is opened, NFS verifies that the cached file handle is still valid. The file server, for its part, treats a file handle as invalid if the file has been written (by some other client system) since the handle was issued. Thus, by issuing a single open request, the client system is able to learn whether the data blocks cached on behalf of the file are valid or not, and can discard them in the latter case.

This approach to cache validation poses a potential problem, which is that if a client workstation has cached data from an open file, changes to the file that originate at some other workstation will not invalidate these cached blocks, and no attempt to authenticate the file handle will occur. Thus, for example, if process *p* on client workstation *a* has file *F* open, and then process *q* on client workstation *b* opens *F*, writes modified data into it, and then closes it, although *F* will be updated on the file server, process *p* may continue to observe stale data for an unlimited period of time. Indeed, short of closing and reopening the file, or accessing some file block that is not cached, *p* might never see the updates!

One case where this pattern of behavior can arise is when a pipeline of processes is executed with each process on a different computer. If *p* is the first program in such a pipeline and *q* is the second program, *p* could easily send a message down the pipe to *q* telling it to look into the file, and *q* will now face the stale data problem. UNIX programmers often encounter problems such as this and work around them by modifying the programs to use *fflush* and *fsync* system calls to flush the cache at *p* and to empty *q*'s cache of cached records for the shared file.

NFS vendors provide a second type of solution to this problem through an optional locking mechanism, which is accessed using the *flock* system call. If this optional interface is used, the process attempting to write the file would be unable to open it for update until the process holding it open for reads has released its read lock. Conceptually, at least, the realization that the file needs to be unlocked and then relocked would sensitize the developer of process *p* to the need to close and then reopen the file to avoid access anomalies, which are well documented in NFS. At any rate, file sharing is not all that common in UNIX, as demonstrated in studies by Ousterhout et. al. [ODHK85], where it was found that most file sharing is between programs executed sequentially from the same workstation.

The NFS protocol is thus stateless but there are situations in which the user can glimpse the implementation of the protocol precisely because its statelessness leads to weakened semantics compared to an idealized file system accessed through a single cache. Moreover, as noted in the previous chapter, there are also situations in which the weak error reporting of RPC protocols is reflected in unexpected behavior, such as the file *create* operation of Section 4.7, which incorrectly reported that a file couldn't be created because a reissued RPC fooled the file system into thinking the file already existed.

Similar to the basic UNIX file system, NFS is designed to prefetch records when it appears likely that they will soon be needed. For example, if the application program reads the first two blocks of a file, the NFS client-side software will typically fetch the third block of that file without waiting for a read request, placing the result in the cache. With a little luck, the application will now obtain a cache hit, and be able to start processing the third block even as the NFS system fetches the fourth one. One can see that this yields similar performance to simply transferring the entire file at the time it was initially opened. Nonetheless, the protocol is relatively inefficient in the sense that each block must be independently requested, whereas a streaming-style of transfer could avoid these requests and also handle acknowledgements more efficiently. Below, we will look at some file systems that explicitly perform whole-file transfers and that are able to outperform NFS when placed under heavy load.

For developers of reliable applications, the reliability of the file server is of obvious concern. One might want to know how failures would affect the behavior of operations. With NFS, as normally implemented, a failure can cause the file server to be unavailable for long periods of time, can partition a client from a server, or can result in a crash and then reboot of a client. The precise consequences depend on the user set the file system up. For the situations where a server becomes unreachable or crashes and later reboots, the client program may experience timeouts that would be reported to the application layer as errors, or it may simply retry its requests periodically, for as long as necessary until the file server restarts. In the latter case, an operation to be reissued after a long delay, and there is some potential for operations to behave unexpectedly as in the case of *create*. Client failures, on the other hand, are completely ignored by the server.

Because the NFS client-side cache uses a write-through policy, in such a situation a few updates may be lost but the files on the server will not be left in an extremely stale state. The locking protocol used by NFS, however, will not automatically break locks during a crash, hence files locked by the client will remain locked until the application detects this condition and forces the locks to be released, using commands issued from the client system or from some other system. There is a mode in which failures automatically cause locks to be released, but this action will only occur when the client workstation is restarted, presumably to avoid confusing network partitions with failure/reboot sequences.

Thus, while the stateless design of NFS simplifies it considerably, the design also introduces serious reliability concerns. Our discussion has touched on the risk of processes seeing stale data when they access files, the potential that writes could be lost, and the possibility that a critical file server might become unavailable due to a network or computer failure. Were one building an application for which reliability is critical, any of these cases could represent a very serious failure. The enormous success of NFS should not be taken as an indication that reliable applications can in fact be built over it, but rather as a sign that failures are really not all that frequent in modern computing systems, and that most applications are not particularly critical! In a world where hardware was less reliable or the applications were more critical, protocols such as the NFS protocol might be considerably less attractive.

Our discussion is for the case of a normal NFS server. There are versions of NFS that support replication in software for higher availability (R/NFS, Deceit [Sie92], HA-NFS [BEM91], and Harp [LGGJ91, LLSG92]), as well as dual-ported NFS server units in which a backup server can take control of the file system. The former approaches employ process-group communication concepts of a sort we will discuss later, although the protocol used to communicate with client programs remains unchanged. By doing this, the possibility for load-balanced read access to the file server is created, enhancing read performance through parallelism. At the same time, these approaches allow continuous availability even when some servers are down. Each server has its own disk, permitting tolerance of media failures. And, there is a possibility of varying the level of the replication selectively, so that critical files will be replicated but non-critical files can be treated using conventional non-replicated methods. The interest in such an approach is that any overhead associated with file replication is incurred only for files where there is also a need for high availability, and hence the multi-server configuration comes closer to also giving the capacity and performance benefits of a cluster of NFS servers. Many users like this possibility of "paying only for what they use."

The dual-ported hardware approaches, in contrast, primarily reduce the time to recovery. They normally require that the servers reside in the same physical location, and are intolerant of media failure, unless a "mirrored disk" is employed. Moreover, these approaches do not offer benefits of parallelism: one pays for two servers, or for two servers and a mirror disk, as a form of insurance that the entire file system will be available when needed. These sorts of file servers are, consequently, expensive. On the other hand, their performance is typically that of a normal server – there is little or no *degradation* because of the dual configuration.

Clearly, if the performance degradation associated with replication can be kept sufficiently small, the mirrored server and/or disk technologies will look expensive. Early generations of cluster-server technology were slow enough to maintain mirroring as a viable alternative. However, the trend seems to be for this overhead to become smaller and smaller, in which case the greater flexibility and enhanced read performance, due to parallelism, would argue in favor of the NFS cluster technologies.

Yet another file system reliability technology has emerged relatively recently, and involves the use of clusters or arrays of disks to implement a file system that is more reliable than any of the component disks. Such so-called RAID file systems [PGK88] normally consist of a mixture of hardware and software: the hardware for mediating access to the disks themselves, and the software to handle the buffer pool, oversee file layout and optimize data access patterns. The actual protocol used to talk to the RAID device over a network would be the same as for any other sort of remote disk: thus, it might be the NFS protocol, or some other remote file access protocol; the use of RAID in the disk subsystem itself would normally not result in protocol changes.

RAID devices typically require physical proximity of the disks to one-another (needed by the hardware). The mechanism that implements the RAID is typically constructed in hardware, and employs a surplus disk to maintain redundent data in the form of parity for sets of disk blocks; such an approach permits a RAID system to tolerate one or more disk failures or bad blocks, depending on the way the system is configured. A RAID is thus a set of disks that mimics a single more reliable disk unit with roughly the summed capacity of its components, minus overhead for the parity disk. However, even with special hardware, management and configuration of RAID systems can require specialized software architectures [WSS95]

Similar to the case for a mirrored disk, the main benefits of a RAID architecture are high availability in the server itself, together with large capacity and good average seek-time for information retrieval. In a large-scale distributed application, the need to locate the RAID device at a single place, and its reliance on a single source of power and software infrastructure, often mean that in practice such a file server has the same distributed reliability properties as would any other form of file server. In effect, the risk of file server unavailability as a source of downtime is reduced, but other infrastructure-related sources of file system unavailability remain to be addressed. In particular, if a RAID file system implements the NFS protocol, it would be subject to all the limitations of the NFS architecture.

7.4 Stateful File Servers

The performance of NFS is limited by its write-through caching policy, which has lead developers of more advanced file systems to focus on improved caching mechanisms and, because few applications actually use the optional locking interfaces, on greater attention to cache validation protocols. In this section, we briefly survey some of the best known stateful file systems. Our treatment is brief because of the

The Andrew File System was developed at Carnegie Mellon University and subsequently used as the basis of a world-wide file system product offered by Transarc Inc [SHNS85, SS96]. The basic ideas in Andrew are easily summarized.

AFS was built with the assumption that the Kerberos authentication technology would be available. We present Kerberos in Chapter 19, and hence limit ourselves to a brief summary of the basic features of the system here. At the time that a user logs in (and later, periodically, if the user remains connected long enough for timers to expire), Kerberos prompts for a password. Using a secure protocol that employs DES to encrypt sensitive data, the password is employed to authenticate the user to the Kerberos server, which will now act as a trustworthy intermediary in establishing connections between the user and file servers that it will access. The file servers similarly authenticate themselves to the Kerberos authentication server at startup.

File system access is by whole file transfer, except in the case of very large files, which are treated as sets of smaller ones. Files can be *cached* in the AFS subsystem on a client, in which case requests are satisfied out of the cached information whenever possible (in fact, there are two caches, one of file data and one of file status information, but this distinction need not concern us here). The AFS server tracks the clients that maintain cached copies of a given file, and, if the file is opened for writing, uses *callbacks* to inform those clients that the cached copies are no longer valid. Additional communication from the client to the server occurs frequently enough so that if a client becomes disconnected from the server, it will soon begin to consider its cached files to be potentially stale. (Indeed, studies of AFS file server availability have noted that disconnection from the server is a more common source of denial of access to files in AFS than genuine server downtime).

AFS provides a strong form of security guarantee, based on *access control lists* at the level of entire directories. Because the Kerberos authentication protocol is known to be highly secure, AFS can trust the user identification information provided to it by client systems. Short of taking over a client workstation, an unauthorized user would have no means of gaining access to cached or primary copies of a file for which access is not permitted. AFS destroys cached data when a user logs out or an authorization expires and is not refreshed [Sat89, SNS88, Sch94, Bir85, LABW92, BM90].

In its current use as a wide-area file system, AFS has expanded to include some 1,000 servers and 20,000 clients in 10 countries, all united within a single file system name space [SS96]. Some 100,000 users are believed to employ the system on a regular basis. Despite this very large scale, 96% of file system accesses are found to be resolved through cache hits, and server inaccessibility (primarily due to communications timeouts) was as little as a few minutes per day. Moreover, this is found to be true even when a significant fraction of file references are to remote files. AFS users are reported to have had generally positive experiences with the system, but (perhaps not surprisingly) complain about poor performance when a file is not cached and must be copied from a remote file server. Their subjective experience presumably reflects the huge difference in performance between AFS in the case where a file is cached and that when a copy must be downloaded over the network.

Figure 7-3: The Andrew File System (AFS) is widely cited for its strong security architecture and consistency guarantees.

complexity of the implementations of the systems we review: the choice is between a superficial treatment and one that would be quite lengthy. In many respects, stateful file servers are basically implementations of a specialized form of data replication mechanism. In Chapters 13-16 and 21, we will be looking at such mechanisms in a stepwise, structured manner, hence a decision was made to omit some details here. This is unfortunate, because the file systems we now review are a rich and extremely interesting group of distributed systems that incorporate some very innovative methods for overcoming the technical problems that they confront. The reader is strongly encouraged to read some of the original research papers on these and related systems.

Work on stateful file systems architectures can be traced in part to an influential study of file access patterns in the Sprite system at Berkeley [BHKSO91]. This work sought to characterize the file system workload along a variety of axes: read/write split, block reuse frequency, file lifetimes, and so forth. The findings, although not surprising, were at the same time eye-openers for many of the researchers in the area. In this study, it was discovered that nearly nearly all file access was sequential, and that there was very little sharing of files between different programs. When file sharing was observed, the prevailing pattern was the simplest one: one program tended to write the file, in its entirety, and then some other program would read the same file. Often (indeed, in most such cases), the file would be deleted shortly after it was created. In fact, most files survived for less than 10 seconds or longer than 10,000 seconds. The importance of cache consistency was explored in this work (it turned out to be quite important, but relatively easy to enforce for the most common patterns of sharing), and the frequency of write-write sharing of files was shown to be so low that this could almost be treated as a special case. (Later, there was considerable speculation that on systems with significant database activity, this finding would have been affected). Moreover, considerable data was extracted on patterns of data transfer from server to client: rate of transfer, percentage of the typical file that was transferred, etc. Out of this work came a new generation of file systems that used closer cooperation between client and file system to exploit such patterns.

Examples of well-known file systems that take employ a stateful approach to provide increased performance (as opposed to availability) are AFS (see Figure 7-3) [Sat89, SHNS85, HKMN87] and Sprite[OCDN88, SM89], a research file system and operating system developed at U.C. Berkeley. On the availability side of the spectrum, the Coda project [KS91, MES95], a research effort at Carnegie Mellon University, takes these ideas one step further, integrating them into a file system specifically for use on mobile computers which operate in a disconnected, or partially connected, mode. Ficus, a project at UCLA, uses a similar approach to deal with file replication in very wide-area networks with non-uniform connectivity and bandwidth properties. To varying degrees, these systems can all be viewed as stateful ones in which some of the information maintained within client workstations is guaranteed to be coherent. The term stateful is used a little loosely here, particularly in comparison with the approaches we will examine in Chapter 15. Perhaps it would be preferable to say that these systems are “more stateful” than the NFS architecture, gaining performance through the additional state. Among the four, only Sprite actually provides strong cache coherence to its clients [SM89]. The other systems provide other forms of guarantees, which are used either to avoid inconsistency or to resolve inconsistencies after they occur. Finally, we will briefly discuss XFS, a file system under development at U.C. Berkeley that exploits the file system memory of client workstations as an extended buffer pool, paging files from machine to machine over the network to avoid the more costly I/O path from a client workstation over the network to a remote disk.

Both AFS and Sprite replace the NFS write-through caching mechanism and file-handle validation protocols with alternatives that reduce costs. The basic approach in AFS is to cache entire files, informing the server that a modified version of a file may exist in the client workstation. Through a combination of features such as whole-file transfers on file open and for write back to the server, and by having the file server actively inform client systems when their cached entries become invalid, considerable performance improvements are obtained with substantially stronger file access semantics than for NFS. Indeed, the workload on an AFS server can be an order of magnitude or more lower than that for an NFS server, and the performance observed by a client is comparably higher for many applications. AFS was commercialized subsequent to the intial research project at CMU, becoming the component technology for a line of enterprise file systems (“world-wide file systems”) marketed Transarc, a subsidiary of IBM. The file system is discussed further in Figure 7-3.

Sprite, which caches file system blocks (but uses a large 4k block size), takes the notion of coherent caching one step further, using a protocol in which the server actively tracks client caching, issuing callbacks to update cached file blocks if updates are received. The model is based on the caching of individual data blocks, not whole files, but the client caches are large enough to accomodate entire files. The Sprite approach leads to such high cache hit rates that the server workload is reduced to almost pure writes, an observation that triggered some extremely interesting work on file system organizations for workloads that are heavily biased towards writes. Similar to AFS, the technology greatly decreases the I/O load and CPU load on the servers that actually manage the disk.

Sprite is unusual in two ways. First, the system implements several different caching policies depending upon how the file is opened: one policy is for read-only access; a second and more expensive one is used for *sequential write access,* which occurs when a file is updated by one workstation and then accessed by a second one later (but in which the file is never written simultaneously from several system), and a third policy is used for *concurrent write access*, which occurs when a file is written concurrently from several sources. This last policy is very rarely needed because Sprite does not cache directories and is not often used in support of database applications, which are among the few known to use concurrent write sharing heavily. Secondly, unlike NFS, Sprite does not use a write-through policy. Thus, a file that is opened for writing, updated, then closed and perhaps reopened by another application on the same machine, read, and then deleted, would remain entirely in the cache of the client workstation. This particular sequence is commonly seen in compilers that run in multiple passes and generate temporary results, and in editors that operate on an intermediate copy of a file which will be deleted after the file is rewritten and closed. The effect is to greatly reduce traffic between the client and the server relative to what NFS might have, but also to leave the server quite far out of date with respect to a client system that may be writing cached files.

Sequential write sharing is handled using version numbers. When a client opens a file, the server returns the current version number, permitting the client to determine whether or not any cached records it may have are still valid. When a file is shared for concurrent writing, a more costly but simple scheme is used, whereby none of the clients are permitted to cache it. If the status of a file changes because a new open or close has occured, Sprite issues a callback to other clients that have the file open, permitting them to dynamically adapt their caching policy in an appropriate manner. Notice that because a stateless file system such as NFS has no information as to its current client set, this policy would be impractical to implement within NFS. On the other hand, Sprite faces the problem that if the callback RPC fails, it must assume that the client has genuinely crashed; the technology is thus not tolerant of communication outages that can partition a file server from its clients. Sprite also incurs costs that NFS can sometimes avoid: both *open* and *close* operations must be performed as RPC's, and there is at least one extra RPC required (to check consistency) in the case where a file is opened, read quickly, and then closed than in NFS.

The recovery of a Sprite server after a crash can be complicated, because some clients may have had files opened in a cached for writing mode. To recover, the server makes use of its knowledge of the set of clients that had cached files for writing, which is saved in a persistent storage area, and of the fact that the consistency state of a file cannot change without the explicit approval of the server. This permits the server to track down current copies of the files it manages and to bring itself back to a consistent state.

The developers of Sprite commented that "most of the complexity in the recovery mechanism comes in detecting crashes and reboots, rather than in rebuilding state. This is done by tracking the passage of RPC packets, and using periodic "keepalive" packets, to detect when a client or server has crashed or rebooted: the same mechanism also suffices to detect network partitions. There is some cost to tracking RPC packets but a reliable crash and reboot detetection mechanism is of course useful for other purposes besides recovering file server state[SM89]." This comment may at first seem confusing, because we have seen that RPC mechanisms cannot reliably detect failures. However, Sprite is not subject to the restrictions we cited earlier because it can deny access to a file while waiting to gain access to the most current version of it. Our concerns about RPC arose in relation to trying to determine the cause of an RPC failure "in real-time". A system that is able to wait for a server to recover is fortunate in not needing to solve this problem: if an apparent failure has occured, it can simply wait for the problem to be repaired if to do otherwise might violate file system consistency guarantees.

Experiments have shown the Sprite cache-consistency protocols to be highly effective in reducing traffic to the file server and preserving the illusion of a single-copy of each file. Performance of the system is extremely good, utilization of servers very low, and the anomalous behaviors that can arise with NFS are completely avoided. However, the technology is trusting of user-id's, and hence suffers from some of the same security concerns that we will present in relation to NFS in Chapter 19.

Coda is a file system for disconnected use. It can be understood as implementing a very generalized version of the whole-file caching methods first introduced in AFS: where AFS caches individual files, Coda caches groups of files and directories so as to maintain a complete cached copy of the user's entire file system or application. The idea within Coda is to track updates with sufficient precision so that the actions taken by the user while operating on a cached copy of part of the file system can be merged automatically into the master file system from which the files were copied. This merge occurs when connection between the disconnected computer and the main file system server is reestablished. Much of the sophistication of Coda is concerned with tracking the appropriate sets of files to cache in this manner, and with optimizing the merge mechanisms so that user intervention can be avoided when possible.

The challenge faced by Coda is easy appreciated when the following example is considered. Suppose that Fred and Julia are collaborating on a major report to an important customer of their company. Fred is responsible for certain sections of the report and Julia for others, but these sections are also cited in the introductory material and "boilerplate" used to generate the report as a whole. As many readers of this textbook will appreciate, there are software tools for this type of collaborative work of varying ease of use. The most primitive tools provide only for locking of some sort, so that Julia can lock Fred out of a file while she is actually editing it. More elaborate ones actually permit multiple users to concurrently edit the shared files, annotating one-another's work, and tracking precisely who changed what through multiple levels of revisions. Such tools typically view the document as a form of database and keep some type of log or history showing how it evolved through time.

If the files in which the report are contained can be copied onto portable computers that become disconnected from the network, however, these annotations will be introduced independently and concurrently on the various copies. Files may be split or merged while the systems are disconnected from each other, and even the time of access cannot be used to order these events, since the clocks on computers can drift or be set incorrectly for many reasons. Thus, when copies of a complex set of files are returned to the file system from which they were removed, the merge problem becomes a nontrivial one both at the level of the file system itself (which may have to worry about directories that have experienced both delete and add operations of potentially conflicting sorts in the various concurrent users of the directory), and also at the level of the application and its notion of file semantics.

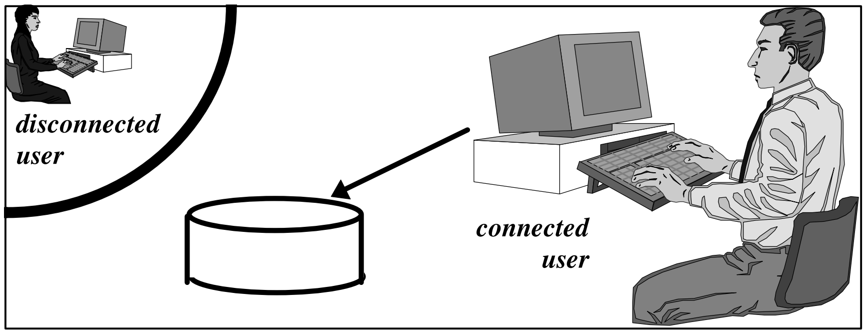


Figure 7-4: Challenges of disconnected operation.

The Ficus system, developed by Jerry Popek's group at UCLA [RHRS94], explores a similar set of issues but focuses on an enterprise computing environment similar to the world-wide file system problems to which AFS has been applied in recent years. (For brevity we will not discuss a previous system developed by the same group, Locus [WPEK92]). In Ficus, the model is one of a large-scale file system built of file servers that logically maintain replicas of a single "file system image". Communication connectivity can be lost and servers can crash, hence at any point, a server will have replicas of some parts of the file system and will be out of touch with some other replicas for the same data. This leads to an approach in which file type information is used both to limit the updates that can be performed while a portion of the file system is disconnected from other segments, and to drive a file merge process when communication is reestablished [HP95]. Where Coda is focused on disconnected operation, however, Ficus emphasizes support for patterns of communication seen in large organizations that experience bandwidth limits or partitioning problems that prevent servers from contacting each other for brief periods of time. The resulting protocols and algorithms are similar to the ones used in Coda, but place greater attention on file-by-file reconciliation methods, where Coda is oriented towards mechanisms that deal with groups of files as an ensemble.

All of these systems are known for additional contributions beyond the ones we have discussed. Coda, for example, makes use of a recoverable virtual memory mechanism that offers a way to back out changes made to a segment of virtual memory, using a logging facility that performs replay on behalf of the user. Ficus is also known for work on "stackable" file systems, in which a single file system interface is used to provide access to a variety of types of file-like abstractions. These contributions, and others not cited here, are beyond the scope of our present discussion.

Not surprisingly, systems such as Coda and Ficus incorporate special purpose programming tools and applications that are well matched to their styles of disconnected and partially connected operation [RHRS94, MES95]. These tools include, for example, email systems that maintain logs of actions taken against mailboxes, understanding how to delete mail that has been deleted while in a disconnected mode, or to merge emails that arrived separately in different copies of a mailbox that was split within a large-scale distributed environment. One can speculate that over time, a small and fairly standard set of tools might emerge from such research, with which developers would implement specialized "disconnected applications" that rely on well-tested reconciliation methods to recorrect inconsistencies that arise during periods of disconnected interaction. At the time of this writing, however, the author was not aware of any specific toolkits of this nature.

The last of the stateful file systems mentioned at the start of this section is XFS, a Berkeley project that seeks to exploit the memory of the client workstations connected to a network as a form of distributed storage region for a very high performance file server [ADNP95]. XFS could be called a "serverless network file system", although in practice the technology would more often be paired to a conventional file system which would serve as a backing store. The basic idea of XFS, then, is to distribute the contents of a file system over a set of workstations so that when a block of data is needed, it can be obtained by a direct memory-to-memory transfer over the network rather than by means of a request to a disk server which, having much less memory at its disposal, may then need to delay while fetching it from the disk itself.

XFS raises some very complex issues of system configuration management and fault-tolerance. The applications using an XFS need to know what servers belong to it, and this set changes dynamically over time. Thus, there is a membership management problem that needs to be solved in software. Workstations are reliable, but not completely reliable, hence there is a need to deal with failures; XFS does this by using a RAID-style storage scheme in which each set of n workstations is backed by an $n+1$*'st* machine that maintains a parity block. If one of the $n+1$ machines fails, the missing data can be regenerated from the other n. Moreover, XFS is dynamically reconfigurable, creating some challenging synchronization issues. On the positive side, all of this complexity brings with it a dramatic performance improvement when XFS is compared with more traditional server architectures. For this textbook, it is appealing to see XFS as an instance of the sort of server that can be built using the techniques of Chapter 15, in which we present protocols to solve problems like these, and also show how they can be packaged in the form of readily used tools. (It should be noted that XFS draws heavily on the log-structured file system work of Rosenblum and Ousterhout [RO91], a technology that is beyond the scope of this text).

The reliability properties of these stateful file systems go well beyond those of NFS. For AFS and Sprite, reliability is limited by the manner in which the servers detect the failure of clients, since a failed client clears its cache upon recovery and the server needs to update its knowledge of the state of the cache accordingly. In fact, both AFS and Sprite detect failures through timeouts, hence there can be patterns of failure that would cause a client to be incorrectly sensed as having failed, leaving its file system cache corrupted until some future attempt to validate cache contents occurs, at which point the problem would be detected and reported. In Sprite, network partition failures are considered unlikely because the physical network used at Berkeley is quite robust, and in any case, network partitions cause the client workstations to initiate a recovery protocol. Information concerning the precise handling of network partitions, or about methods for replicating AFS servers, was not available to the author at the time of this

The Lotus Notes system is a commercial database product that uses a client-server model to manage collections of documents, which can draw upon a great variety of applications (word-processing, spreadsheets, financial analysis packages, and so forth). The system is widely popular because of the extremely simple sharing model that it supports and its close integration with email and "chat" facilities, supporting what has become known as a "groupware" collaboration model. The term *computer supported collaborative work,* or CSCW, is often used in reference to activities that are supported by technologies such as Lotus Notes.

Notes is structured as a client-server architecture. The client system is a graphical user interface that permits the user to visualize information within the document database, create or annotate documents, "mine" the database for documents satisfying some sort of a query, and to exchange email or send memos which can contain documents as attachments. A security facility permits the database to be selectively protected using passwords, so that only designated users will have access to the documents contained in those parts of the database. If desired, portions of especially sensitive documents can be encrypted so that even a database administrator would be unable to access them without the appropriate passwords.

Lotus Notes also provides features for *replication* of portions of its database between the client systems and the server. Such replication permits a user to carry a self-contained copy of the desired documents (and others to which they are attached) and to update them in a disconnected mode. Later, when the database server is back in contact with the user, updates are exchanged to bring the two sets of documents back into agreement. Replication of documents is also possible among Notes servers within an enterprise, although the Notes user must take steps to limit concurrent editing when replication is employed. (This is in contrast with Coda, which permits concurrent use of files and works to automatically merge changes). At the time of this writing, Notes did not support replication of servers for increased availability, and treated each server as a separate security domain with its own users and passwords.

Within the terminology of this chapter, Lotus Notes is a form of partially stateful file server, although presented through a sophisticated object model and with powerful tools oriented towards cooperative use by members of work groups. Many of the limitations of stateless file servers are present in Notes, however, such as the need to restrict concurrent updates to documents that have been replicated. The Notes environment user environment is extremely well engineered and is largely successful in presenting such limitations and restrictions as features that the skilled Notes user learns to employ. In effect, by drawing on semantic knowledge of the application, the Lotus Notes developers were able to work around limitations associated with this style of file server. The difficulty encountered in distributed file systems is precisely that they lack this sort of semantic knowledge and are consequently forced to solve such problems in complete generality, leading to sometimes surprising or non-intuitive behavior that reveals their distributed infrastructure.

writing. XFS is based on a failure model similar to that of AFS and Sprite, in which crash-failures are anticipated and dealt with in the basic system architecture, but partitioning failures that result in the misdiagnosis of apparent crash failures is not an anticipated mode of failure.

Coda and Ficus treat partitioning as part of their normal mode of operation, dealing with partitioning failures (or client and server failures) using the model of independent concurrent operation and subsequent state merge that was presented earlier. Such approaches clearly trade higher availability for a more complex merge protocol and greater sophistications within the applications themselves.

7.5 Distributed Database Systems

Distributed database systems represent the other very large use of client-server architectures in distributed systems. Unlike the case of distributed file systems, however, database technologies use a special programming model called the *transactional approach* and support this through a set of special protocols

[Gra79, GR93]. The reliability and concurrency semantics of a database are very well understood through this model, and its efficient implementation is a major topic of research – and an important arena for commercial competition. For the purposes of this introductory chapter, we will simply survey the main issues, returning to implementation issues later, in Chapter 21. Consistent with the practical tone of this text as a whole, we will not engage in a detailed treatment of the theory of serializability or transactional systems, although this is a rich area of study about which a great deal is known [BHG87].

Transactional systems are based upon a premise that applications can be divided into client programs and server programs, such that the client programs have minimal interactions with one another. Such an architecture can be visualized as a set of wheels, with database servers forming the hubs to which client programs are connected by communication pathways – the spokes. One client program can interact with multiple database servers, but although the issues this raises are very well understood, such "multi-database" configurations are relatively uncommon in commercial practice. To good approximation, then, existing client-server database applications consist of some set of disjoint groups, each group containing a database server and its associated clients, with no interaction between client programs except through sharing a database, and with very few, if any, client programs that interact with multiple databases simultaneously. Moreover, although it is known how to replicate databases for increased availability and load-balancing [BHG87, GR93], relatively little use is made of this option in existing systems. Thus, the "hubs" of distributed database systems rarely interact with one another. (We'll see why this is the case in Part III of the textbook; ultimately, the issue turns out to be one of performance).

A central premise of the approach is that each interaction by a client with the database server can be structured as a *begin* event, followed by a series of database operations (these would normally be database queries, but we can think of them as *read* and *update* operations and ignore the details), followed by a *commit* or *abort* operation. Such an interaction is called a *transaction*, and a client program will typically issue one or more transactions, perhaps interacting with a user or the outside world between the completion of one transaction and the start of the next. A transactional system should guarantee the persistence of committed transactions, although we will see that high availability database systems sometimes weaken this guarantee to boost performance. When a transaction is aborted, on the other hand, its effects are completely rolled back, as if the transaction had never even been issued.

Transactional client-server systems are stateful: each action by the client assumes that the database remembers various things about the previous operations done by the same client, such as locking information that arises out of the database concurrency control model and updates that were previously performed by the client as part of the same transaction. The clients can be viewed as maintaining coherent caches of this same information during the period while a transaction is active (not yet committed).

The essential property of the transactional execution model, which is called the *serializability* model, is that it guarantees isolation of concurrent transactions. That is, if transactions T_1 and T_2 are executed concurrently by client processes p and q, the effects will be as if T_1 had been executed entirely before T_2, or entirely after T_2 – the database actively prevents them from interfering with one another. The reasoning underlying this approach is that it will be easier to write database application programs to assume that the database is idle at the time the program executed. Rather than force the application programmer to cope with real-world scenarios in which multiple applications simultaneously access database, the database system is only permitted to interleave operations from multiple transactions if it is certain that the interleaving will not be noticeable to users. At the same time, the model frees the database system to schedule operations in a way that keeps the server as busy as possible on behalf of a very large number of concurrent clients.

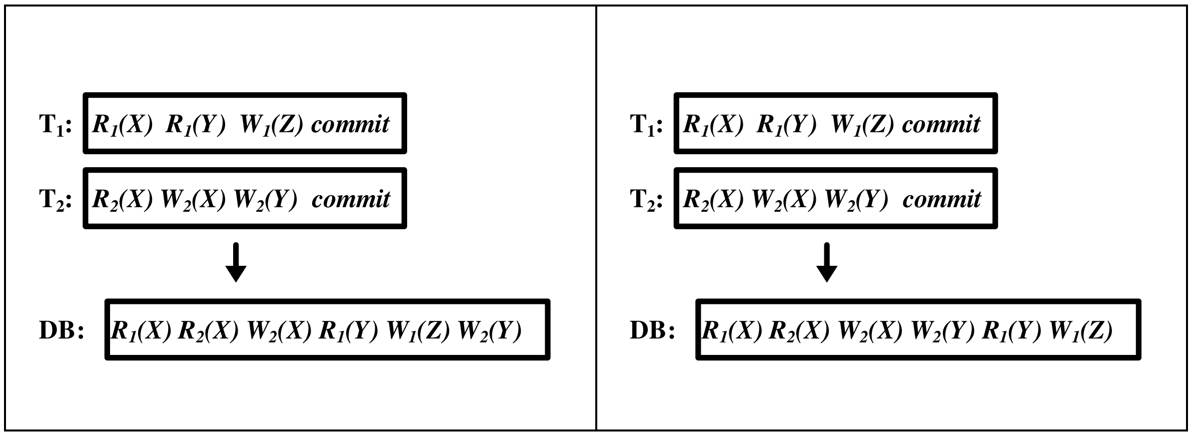

T_1: $R_1(X)$ $R_1(Y)$ $W_1(Z)$ commit
T_2: $R_2(X)$ $W_2(X)$ $W_2(Y)$ commit
DB: $R_1(X)$ $R_2(X)$ $W_2(X)$ $R_1(Y)$ $W_1(Z)$ $W_2(Y)$
T_1: $R_1(X)$ $R_1(Y)$ $W_1(Z)$ commit
T_2: $R_2(X)$ $W_2(X)$ $W_2(Y)$ commit
DB: $R_1(X)$ $R_2(X)$ $W_2(X)$ $W_2(Y)$ $R_1(Y)$ $W_1(Z)$

Figure 7-5: A non-serializable transaction interleaving (left), and one serializable in the order T_2, T_1 (right). Each transaction can be understood as a trace that records the actions of a program that operates on the database, oblivious to other transactions that may be active concurrently. In practice, of course, the operations become known as the transaction executes, although our example shows the situation at the time these two transactions reach their commit points. The database is presented with the operations initiated by each transaction, typically one by one, and schedules them by deciding when to execute each operation. This results in an additional "trace" or log showing the order in which the database actually performed the operations presented to it. A serializable execution is one that leaves the database in a state that could have been reached by executing the same transactions one by one, in some order, and with no concurrency.

Notice that simply running transactions one at a time would achieve the serializability property[6]. However, it would also yield poor performance, because each transaction may take a long time to execute. By running multiple transactions at the same time, and interleaving their operations, a database server can give greatly improved performance, and system utilization levels will rise substantially, just as a conventional uniprocessor can benefit from multi-tasking. Even so, database systems sometimes need to delay one transaction until another completes, particularly when transactions are very long. To maximize performance, it is common for client-server database systems to require (or at least strongly recommend) that transactions be designed to be as short as possible. Obviously, not all applications fit these assumptions, but they match the needs of a great many computing systems.

There are a variety of options for implementing the serializability property. The most common is to use locking, for example by requiring that a transaction obtain a read-lock on any data item that it will read, and a write-lock on any data item it will update. Read-locks are normally non-exclusive: multiple transactions are typically permitted to read the same objects concurrently. Write-locks, however, are mutually exclusive: only one transaction can hold such a lock at a time. In the most standard locking protocol, called *2-phase locking*, transactions retain all of their locks until they commit or abort, and then release them as a group. It is easy to see that this achieves serializability: if transaction T_j reads from T_i, or updates a variable after T_i does so, T_j must first acquire a lock that T_i will have held exclusively for its update operation. Transaction T_j will therefore have to wait until T_i has committed and will be serialized after T_i. Notice that the transactions can obtain read locks on the same objects concurrently, but because read operations commute, they will not affect the serialization order (the problem gets harder if a transaction may need to "upgrade" some of its read locks to write locks!)

[6] An important special case arises in settings where each transaction can be represented as a single operation, performing a desired task and then committing or aborting and returning a result. Many distributed systems are said to be "transactional" but in fact operate in this much more restrictive manner. However, even if the application perceives a transaction as being initiated with a single operation, the database system itself may execute that transaction as a series of operations. These observations motivate a number of implementation decisions and optimizations, which we discuss in Chapter 21.

Concurrency control (and hence locking) mechanisms can be classified as *optimistic* or *pessimistic*. The locking policy described above is a pessimistic one, because each lock is obtained before the locked data item is accessed. An optimistic policy is one in which transactions simply assume that they will be successful in acquiring locks and perform the necessary work in an opportunistic manner. At commit time, the transaction also verifies that its optimistic assumption was justified (that it got lucky, in effect), and aborts if it now turns out that some of its lock requests should in fact have delayed the computation. As one might expect, a high rate of aborts is a risk with optimistic concurrency control mechanisms, and they can only be used in settings where the granularity of locking is small enough so that the risk of a real locking conflict between two transactions is actually very low.

The pessimistic aspect of a pessimistic concurrency control scheme reflects the assumption that there may be frequent conflicts between concurrent transactions. This makes it necessary for a pessimistic locking scheme to operate in a more conventional manner, by delaying the transaction as each new lock request arises until that lock has been granted; if some other transaction holds a lock on the same item, the requesting transaction will now be delayed until the lock holding transaction has committed or aborted.

Deadlock is an important concern with pessimistic locking protocols. For example, suppose that T_i obtains a read lock on x and then requests a write lock on y. Simultaneously, T_j obtains a read lock on y and then requests a write lock on x. Neither transaction can be granted its lock, and in fact one transaction or the other (or both) must now be aborted. At a minimum, a transaction that has been waiting a very long time for a lock will normally abort; in more elaborate schemes, an algorithm can obtain locks in a way that avoids deadlock, or can use an algorithm that explicitly detects deadlocks when they occur and overcomes them by aborting one of the deadlocked transactions. Deadlock-free concurrency control policies can also be devised: for example, by arranging that transactions acquire locks in a fixed order, or by using a very "coarse" locking granularity so that any given transaction requires only one lock. We will return to this topic, and related issues, in Chapter 21 when we discuss techniques for actually implementing a transactional system.

Locking is not the only way to implement transactional concurrency control. Other important techniques include so-called "timestamped" concurrency control algorithms, in which each transaction is assigned a logical time of execution, and its operations are performed as if they had been issued at the time given by the timestamp. Timestamped concurrency control is relatively uncommon in the types of systems that we consider in this text, hence for reasons of brevity we omit any detailed discussion of the approach. We do note, however, that optimistic timestamped concurrency control mechanisms have been shown to give good performance in systems where there are few true concurrent accesses to the same data items, and that pessimistic locking schemes give the best performance in the converse situation, where a fairly high level of conflicting operations result from concurrent access to a small set of data items. Additionally, timestamped concurrency control is considered preferable when dealing with transactions that do a great deal of writing, while locking is considered preferable for transactions that are read-intensive. Weihl has demonstrated that the two styles of concurrency control cannot be mixed: one cannot use timestamps for one class of transactions and locks for another, on the same database. However, he does give a hybrid scheme that combines features of the two approaches and works well in systems with mixtures of read-intensive and write-intensive transactions.

It is common to summarize the properties of a client-server database system so that the mnemonic ACID can be used to recall them:

- *Atomicity:* Each transaction is executed as if it were a single indivisible unit. The term *atomic* will be used throughout this text to refer to operations that have multiple sub-operations but that are performed in an all-or-nothing manner.

- *Concurrency:* Transactions are executed so as to maximize concurrency, in this way maximizing the degrees of freedom available within the server to schedule execution efficiently (for example, by doing disk I/O in an efficient order).
- *Independence:* Transactions are designed to execute independently from one another. Each client is written to execute as if the entire remainder of the system were idle, and the database server itself prevents concurrent transactions from observing one-another's intermediate results.
- *Durability:* The results of committed transactions are persistent.

Notice that each of these properties could be beneficial in some settings but could represent a disadvantage in others. For example, there are applications in which one wants the client programs to cooperate explicitly. The ACID properties effectively constrain such programs to interact using the database as an intermediary. Indeed, the overall model makes sense for many classical database applications, but is less well suited to message based distributed systems consisting of large numbers of servers and in which the programs coordinate their actions and cooperate to tolerate failures. All of this will add up to the perspective that complex distributed systems need a mixture of tools, which should include database technology but not legislate that databases be used to the exclusion of other technologies. Later we will have much more to say about this topic.

We turn now to the question raised earlier: the sense in which transactional systems are "stateful", and the implications that this has for client-server software architectures.

A client of a transactional system maintains several forms of state during the period that the transaction executes. These include the transaction id by which operations are identified, the intermediate results of the transactional operation (for example, values that were read while the transaction was running, or values that the transaction will write if it commits), and any locks or concurrency control information that has been acquired while the transaction was active. This state is all "shared" with the database server, which for its part must keep original values of any data objects updated by non-committed transactions, keep updates sorted by transactional-id to know which values to commit if the transaction is successful, and maintain read-lock and write-lock records on behalf of the client, blocking other transactions that attempt to access the locked data items while allowing access to the client holding the locks. The server thus knows which processes are its active clients, and must monitor their health in order to abort transactions associated with clients that fail before committing (otherwise, a failure could leave the database in a locked state).

The ability to use commit and abort is extremely valuable in implementing transactional systems and applications. In addition to the role of these operations in defining the "scope" of a transaction for purposes of serializability, they also represent a tool that can be used directly by the programmer. For example, an application be designed to assume that a certain class of operations (such as selling a seat on an airline) will succeed, and to update database records as it runs under this assumption. Such an algorithm would be optimistic in much the same sense as a concurrency control scheme can be optimistic. If, for whatever reason, the operation encounters an error condition (no seats available on some flight, customer credit card refused, etc.) the operation can simply abort and the intermediate actions that were taken will be erased from the database. Moreover, the serializability model ensures that applications can be written without attention to one another: transactional serializability ensures that if a transaction would be correct when executed in isolation, it will also be correct when executed concurrently against a database server that interleaves operations for increased performance.

The transactional model is also valuable from a reliability perspective. The isolation of transactions from one-another avoids inconsistencies that might arise if one transaction were to see the partial results of some other transaction. For example, suppose that transaction T_1 increments variable x by 1 and is executed concurrently with transaction T_2, which decrements x by 1. If T_1 and T_2 read x

concurrently they might base their computations on the same initial value of x. The *write* operation that completes last would then erase the other update. Many concurrent systems are prone to bugs because of this sort of mutual exclusion problem; transactional systems avoid this issue using locking or other concurrency control mechanisms that would force T_2 to wait until T_1 has terminated, or the converse. Moreover, transactional abort offers a simple way for a server to deal with a client that fails or seems to hang: it can simply timeout and abort the transaction that the client initiated. (If the client is really alive, its attempt to commit will eventually fail: transactional systems never guarantee that a commit will be successful). Similarly, the client is insulated from the effects of server failures: it can modify data on the server without concern that an inopportune server crash could leave the database in an inconsistent state.

There is, however, a negative side to transactional distributed computing. As we will see in Chapter 21, transactional programming can be extremely restrictive. The model basically prevents programs from cooperating as peers in a distributed setting, and although extensions have been proposed to overcome this limitation, none seems to be fully satisfactory. That is, transactions really work best for applications in which there is a computational “master” process that issues requests to a set of “slave” processors on which data is stored. This is, of course, a common model, but it is not the only one. Any transactional application in which several processes know about each other and execute concurrently is hard to model in this manner.

Moreover, transactional mechanisms can be costly, particularly when a transaction is executed on data that has been replicated for high availability or distributed over multiple servers. The locking mechanisms used to ensure serializability can severely limit concurrency, and it can be very difficult to deal with transactions that run for long periods of time, since these will often leave the entire server locked and unable to accept new requests. It can also be very difficult to decide what to do if a transaction aborts unexpectedly: should the client retry it? Report to the user that it aborted? Decisions such as these are very difficult, particularly in sophisticated applications in which one is essentially forced to find a way to “roll forward”.

For all of these reasons, although transactional computing is a powerful and popular tool in developing reliable distributed software systems, it does not represent a complete model or a complete solution to all reliability issues that arise.

7.6 Applying Transactions to File Servers

Transactional access to data may seem extremely well matched to the issue of file server reliability. Typically, however, file servers either do not implement transactional functionality, or do so only for the specific case of database applications. The reasons for this illustrate the sense in which a mechanism such as transactional data access may be unacceptably constraining in non-transactional settings.

General purpose computing applications make frequent and extensive use of files. They store parameters in files, search directories for files with special names, store temporary results in files that are passed from phase to phase of a multiphase computation, implement ad-hoc structures within very large files, and even use the existence or non-existence of files and the file protection bits as persistent locking mechanisms, compensating for the lack of locking tools in operating systems such as UNIX.

As we saw earlier, file systems used in support of this model are often designed to be stateless, particularly in distributed systems. That is, each operation by a client is a complete and self-contained unit. The file system maintains no memory of actions by clients, and although the clients may cache information from the file system (such as handles pointing to open file objects), they are designed to refresh this information if it is found to be stale when referenced. Such an approach has the merit of extreme simplicity. It is certainly not the only approach: some file systems maintain coherent caches of file system blocks within client systems, and these are necessarily stateful. Nonetheless, the great majority of distributed file systems are stateless.

The introduction of transactions on files thus brings with it stateful aspects that are otherwise avoided, potentially complicating any otherwise simple system architecture. However, transactions pose more problems than mere complexity. In particular, the locking mechanisms used by transactions are ill-matched to the pattern of file access seen in general operating systems applications.

Consider the program that was used to edit this manuscript. When started, it displays a list of files that end with the extension ".doc", and waited for the author to select the file on which he wished to work. (The author sometimes chose this moment to get a hot cup of coffee, sharpen a pencil, or play a quick game of solitaire!). Eventually, the file selected and open, an extended editing session ensued, perhaps even appearing to last overnight or over a weekend if some distraction prevented the author from closing the file and exiting the program before leaving for the evening. In a standard transactional model, each of the read accesses and each of the write accesses would represent an operation associated with the transaction, and transactional serialization ordering would be achieved by delaying these operations as needed to ensure that only serializable executions are permitted, for example with locks.

This now creates the prospect of a file system containing directories that are locked against updates (because some transaction has read the contents), files that are completely untouchable (because some transaction is updating, or perhaps even deleting the contents), and of long editing sessions that routinely end in failure (because locks may be broken after long delays, forcing the client program to abort its transaction and start again from scratch)! It may not seem obvious that such files should pose a problem, but suppose that a transaction's behavior was slightly different as a result of seeing these transient conditions? That transaction would not be correctly serialized if the editing transaction was now aborted, resulting in some other state. No transaction should have been allowed to see the intermediate state.

Obviously, this analysis could be criticized as postulating a clumsy application of transactional serializability to the file system. In practice, one would presumably adapt the model to the semantics of the application. However, even for the specific case of transactional file systems, the experience systems has been less than convincing. For example, at Xerox the early versions of the "Clearinghouse" software (a form of file system used for email and other user-profile information) offered a fully transactional interface. Over time, this was greatly restricted because of the impracticality of transactional concurrency control in settings that involve large numbers of general-purpose applications.

Moreover many file-based applications lack a practical way to assign a transaction-id to the logical transaction. As an example, consider a version control software system. Such a system seems well matched to the transactional model: a user checks out a file, modifies it, and then checks it in; meanwhile, other users are prevented from doing updates and can only read old copies. Here, however, many individual programs may operate on the file over the period of the "transaction". Lacking is any practical way to associate an identifier with the series of operations. Clearly, the application programs themselves can do so, but one of the basic principles of reliability is to avoid placing excessive trust in the correctness of individual applications; in this example, the correctness of the applications would be a key element of the correctness of the transactional architecture, a very questionable design choice.

On the other hand, transactional file systems offer important benefits. Most often cited among these are the atomic update properties of a transaction, whereby a set of changes to files are made entirely, or not at all. This has resulted in proposals for file systems that are transactional in the limited sense of offering failure atomicity for updates, but without carrying this to the extreme of also providing transactional serializability. Hagmann's use of "group commit" to reimplement the Cedar file system [Hag87] and IBM's QuickSilver file system [SW91] are examples of a research efforts that are viewed as very successful in offering such a compromise. However, transactional atomicity remains uncommon in the mostly widely used commercial file system products because of the complexity associated with a stateful file system implementation. The appeal of stateless design, and the inherent reliability associated with an architecture in which the clients and servers take responsibility only for their own actions and place limited trust in information that they don't own directly, continues to rule the marketplace.

The most popular alternative to transactions is the atomic "rename" operation offered by many commercially standard file systems. For complex objects represented as a single file, or as a rooted graph of files, an application can atomically update the collection by creating a new root object containing the modifications, or pointing to modified versions of other files, and then "rename" the result to obtain the equivalent effect of an atomic commit, with all the updates being installed simultaneously. If a crash occurs, it suffices to delete the partially modified copy; the original version will not be affected. Despite having some minor limitations, designers of fairly complex file systems applications have achieved a considerable degree of reliability using operations such as rename, perhaps together with an "fsync" operation that forces recent updates to an object or file out to the persistent disk storage area.

In conclusion, it is tempting to apply stateful mechanisms and even transactional techniques to file servers. Yet similar results can be obtained, for this particular application, with less costly and cumbersome solutions. Moreover, the simplicity of a stateless approach has enormous appeal in a world where there may be very little control over the software that runs on client computers, and in which trust in the client system will often be misplaced. In light of these considerations, file systems can be expected remain predominantly stateless even in settings where reliability is paramount.

More generally, this point illustrates an insight to which we will return repeatedly in this book. Reliability is a complex goal and can require a variety of "tools". While a stateless file system may be adequately reliable for one use, some other application may find its behavior hopelessly inconsistent and impossible to work around. A stateful database architecture works wonderfully for database applications, but turns out to be difficult to adapt to general purpose operating systems applications that have "less structure", or that merely have a non-transactional structure. Only a diversity of tools, integrated in an environment that encourages the user to match the tool to the need, can possibly lead to reliability in the general sense. No single approach will suffice.

7.7 Message Oriented Middleware

An emerging area of considerable commercial importance, *Message Oriented Middleware* is concerned with extending the client-server paradigm so that clients and servers can be operated asynchronously. This means, for example, that a client may be able to send requests to a server that is not currently operational for batch processing later, and that a server may be able to schedule requests from a request queue without fear of delaying a client application that is waiting for a reply. We discuss "MOMS" later in this text, in conjunction with other distributed computing paradigms that fall out of the strict, synchronous-style, client-server architectures that were the focus of this chapter. The interested reader is refered to Section 11.4.

7.8 Related Topics

The discussion of this chapter has merely touched upon a very active area for both commercial products and academic research. Although NFS is probably the most widely used distributed file system technology, other major products are doing well in the field. For example, Transarc's AFS product (based on a research system developed originally at Carnegie Mellon University) is widely cited for its advanced security and scalability features. AFS is often promoted as a secure, "world-wide" file system technology.

Later, when we discuss NFS security, it will become clear that this is potentially a very important property and represents a serious reliability exposure in distributed computing configurations that use NFS. Locus Computing Corporation's "Locus" product has similar capabilities, but is designed for environments with intermittent connectivity. On the PC side, major file system products are available in Microsoft as part of its Windows NT server technology, from Banyan, and from Novell.

Stateful database and transactional technologies represent one of the largest existing markets for distributed computing systems. Major database products include Sybase, Informix and Oracle; all of these include client-server architectures. There are dozens of less well known but very powerful technologies. OLTP technologies, which permit transaction operations on files and other special purpose data structures, are also a major commercial market: well known products include Tuxedo and Encina; and there are (again) a great many less well known but very successful similar technologies available in this market.

On the research side of the picture, much activity centers around the technical possibilities created by ATM communication, with its extremely high bandwidths and low latencies. File systems that page data over an ATM and that treat the client buffer pools as a large distributed buffering resource shared by all clients are being developed: such systems gain enormous performance benefits from the substantial enlargement in the file system buffer pool that results, and at the same time benefit because the latency incurred when fetching data over the network is orders of magnitude lower than that of fetching data from a remote disk. Examples of this style of research include the XFS project at Berkeley[ADNP95], the Global Memory project at University of Washington [FMPK95], and the CRL project at MIT [JKW95]. Such architectures create interesting reliability and consistency issues closely related to the technologies will be will be discussing in Part III, and it is likely that for them to succeed, these issues must be solved using techniques like the ones we present in discussing process group computing systems.

The changing technology picture is indirectly changing the actual workload presented to the database or file server that resides at the end of the line. One major area of research has concerned the creation of parallel file servers using arrays of inexpensive disks on which an error correcting code is employed to achieve a high degree of reliability. Such RAID file servers have high capacity because they aggregate large numbers of small disks; good data transfer properties, again because they can benefit from parallelism, and good seek time not because the small disks are especially fast, but rather because load is shared across them and this reduces the effective length of the I/O request queue to each drive. Research on striping data across a RAID system to optimize its response time has yielded further performance improvements.

In the past, file systems and database servers saw a mixed read-write load with a bias towards read operations, and were organized accordingly. But as the percentage of active data resident in the buffer pools of clients has risen, the percentage of read requests that actually reach the server has dropped correspondingly. A modern file system server sees a work load that is heavily biased towards update traffic. Best known of the work in this area is Rosenblum's log-structured file system (LFS) [RO91], developed as part of Ousterhout's Sprite project at Berkeley. LFS implements an append-only data structure (a log) which it garbage collects and compacts using background scavenger mechanisms. Fast indexes permit rapid read access to the file system but, because most of the disk I/O is in the form of writes to the log, the system gains a tremendous performance boost. Seltzer, studying similar issues in the context of database systems, showed that similar benefits were possible. One can anticipate that the technology trends now seen in the broad marketplace will continue to shift basic elements of the low level file system architecture creating further opportunities for significant improvements in average data access latencies and in other aspects of client-server performance.

7.9 Related Readings

The author is not aware of any good general reference on NFS itself, although the standard is available from Sun Microsystems and is widely supported. NFS performance and access patterns is studied in [ODHK85] and extended to the Sprite file system in [BHKSO91]. NFS-like file systems supporting replication include [Sie92, BEM91, LGGS91, LLSG92, KLS85, DEC95]. Topics related to the CMU file system work that lead to AFS are covered in [Sat89, SNS88, Sch94, BIR85, LABW92, BM90, Spe85, SHNS85, HKMN87]. Coda is discussed in [KS91, MES95]. RAID is discussed in [PGK88]. Sprite is discussed in [OCDN88, SM89, NWO87]. Ficus is discussed in [RHRS94], Locus in [WPEK92, HP95]. XFS is discussed in [ADNP95]. Work on global memory is covered in [FMPK95, JKW95]. Database references for the transactional approach: [Gra79, BHG87, GR93]. Tandem's system is presented in [BGH87]. Nomadic transactional systems are covered in [AK93, Ami93]. Transactions on file systems are discussed in [Hag87, SW91]. Related work is treated in [LCJS87, LS83, MOS82, MES93].

8. Operating System Support for High Performance Communication

The performance of a communication system is typically measured in terms of the latency and throughput for typical messages that traverse that system, starting in a source application and ending at a destination application. Accordingly, these issues have received considerable scrutiny within the operating systems research community, which has developed a series of innovative proposals for improving performance in communications-oriented applications. Below, we review some of these proposals.

There are other aspects of communication performance that matter a great deal when building a reliable distributed application, but that have received considerably less attention. Prominent among these are the loss characteristics of the communication subsystem. In typical communications architectures, messages are generated by a source application which passes them to the operating system. As we saw early in this textbook, such messages will then travel down some form of protocol stack, eventually reaching a device driver that arranges for the data to be transmitted on the wire. Remotely, the same process is repeated.

Such a path offers many opportunities for inefficiency and potential message loss. Frequently, the layer-to-layer "hand-offs" that occur involve copying the message from one memory region or address space to another, perhaps with a header prepended or a suffix appended. Each of these copying operations will be costly (even if other costs such as scheduling are even more costly), and if a layer is overloaded or is unable to allocate the necessary storage, a message may be lost without warning. Jointly, the consumption of CPU and memory resources by the communication subsystem can become very heavy during periods of frequent message transmission and reception, triggering overload and high rates of message loss. In Chapter 3 we saw that such losses can sometimes become significant. Thus, while we will be looking at techniques for reducing the amount of copying and the number of cross address space control transfers needed to perform a communication operation, the reader should also keep in mind that by reducing copying, these techniques may also be reducing the rate of message loss that occurs in the protocol stack.

The *statistical properties* of communication channels represent an extremely important area for future study. Most distributed systems, and particularly the ones intended for critical settings, assume that communication channels offer identical and independent "quality of service" properties to each packet transmitted. For example, it is typically implicit in the design of a protocol that if two packets are transmitted independently, then the observed latency, data throughput and probability of loss, will be identical. Such assumptions match well with the properties of communications hardware during periods of light, uniform load, but the layers of software involved in implementing communication stacks and routing packets through a complex network can seriously distort these underlying properties.

Within the telecommunications community, bandwidth sharing and routing algorithms have been developed that are fair in the sense of dividing available bandwidth fairly among a set of virtual circuits of known "expected traffic" levels. But the problem of achieving fairness in a packet switched environment with varying loads from many sources is much harder and is not at all well understood. One way to think about this problem is to visualize the operating systems layers through which packets must travel, and the switching systems used to route packets to their destinations, as a form of "filter" that can distort the distribution of packets in time and superimpose errors on an initially error-free data stream. Such a perspective leads to the view that these intermediary software layers introduce noise into the distributions of inter-message latency and error rates.

Such a perspective is readily confirmed by experiment. The most widely used distributed computing environments exhibit highly correlated communication properties: if one packet is delayed, the next will probably be delayed too. If one packet is dropped in transmission, the odds are that the next will be as well. As one might expect, however, such problems are a direct consequence of the same memory constraints and layered architectures that also introduce the large latency and performance overheads that the techniques presented below are designed to combat. Thus, although the techniques discussed in this chapter were developed to provide higher performance, and were not specifically intended to improve the statistical properties of the network, they would in fact be expected to exhibit better statistical behavior than does the standard distributed systems architecture simply be eliminating layers of software that introduce delays and packet loss.

8.1 Lightweight RPC

Performance of remote procedure calls has been a major topic of research since RPC programming environments first became popular. Several approaches to increasing RPC performance have had particularly significant impact.

The study of RPC performance as a research area surged in 1989 when Schroeder and Burrows undertook to precisely measure the costs associated with RPC on the Firefly operating system [SB89]. These researchers started by surveying the costs of RPC on a variety of standard platforms. Their results have subsequently become outdated because of advances in systems and processor speeds, but the finding that RPC performance varies enormously even in relative terms probably remains true today. In their study, the range of performance was from 1.1ms to do a null RPC (equivalent to 4,400 instructions) on the "Cedar" system, highly optimized for the Dorado multi-processor, to 78ms (195,000 instructions) for a very general version of RPC running on a major vendor's top of the line platform (at that time). One interesting finding of this study was that the number of instructions in the RPC code path was often high (the "average" in the systems they looked at was approximately 6,000 for systems with many limitations and about 140,000 for the most general RPC systems). Thus faster processors would be expected to have a big impact on RPC performance, which is one of the reasons that the situation has improved somewhat since the time of this study.

Using a bus analyzer to pin down costs to the level of individual machine cycles, this effort lead to a ten-fold performance improvement in the RPC technology under investigation, which was based originally on the Berkeley UNIX RPC. Among the optimizations that had the biggest impact were the elimination of copying within the application address space by marshaling data directly into the RPC packet using an inline compilation technique, and the implementation of an RPC "fast path" that eliminated all generality in favor of a hand-coded RPC protocol using the fewest instructions possible, subject to the constraint that the normal O/S protection guarantees would be respected. (It is worthwhile to note that on PC operating systems, which often lack protection mechanisms and provide applications with direct access to the I/O devices, even higher performance can often be achieved, but at the cost of substantially reduced security and hence exposure of the system as a whole to bugs and intrusion by viruses).

Soon after this work on Firefly RPC was completed, researchers at the University of Washington became interested in other opportunities to optimize communication paths in modern operating systems. Lightweight RPC originated with the observation that as computing systems adopt RPC-based architectures, the use of RPC in *non-distributed* settings is rising as rapidly as is RPC over a network. Unlike a network, RPC in the non-distributed case can accurately sense many kinds of failures and because the same physical memory is potentially visible to both sender and destination, the use of shared memory mechanisms represents an appealing option for enhancing performance. Bershad, Anderson and others set out to optimize this common special case [BALL89].

A shared memory RPC mechanism typically requires that messages be allocated within pages, starting on page boundaries and with a limit of one message per page. In some cases, the pages used for message passing are from a special pool of memory maintained by the kernel, in others, no such restriction applies but there may be other restrictions, such as limits on passing data structures that contain pointers. When a message is sent, the kernel modifies the page table of the destination to map the page containing the message into the address space of the destination process. Depending on the operating system, the page containing the message may be mapped out of the memory of the sender, modified to point to an empty page, or marked as read-only. In this last approach (where the page is marked as read only) some systems will trap write-faults and make a private copy if either process attempts a modification. This method is called "copy on write," and was first supported in the Mach microkernel [Ras86].

If one studies the overheads associated with RPC in the local, shared memory case, the cost of manipulating the page tables of the sender and destination and of context switching between the sending and receiving processes emerges as a major factor. The University of Washington team focused on this problem in developing what they called a *Lightweight Remote Procedure Call* facility (LRPC). In essence, this approach one reduces time for local RPC both by exploiting shared memory and by avoiding excess context switches. Specifically, the messages containing the RPC arguments are placed in shared memory, while the invocation itself is done by changing the current page table and flushing the TLB so that the destination process is essentially invoked in co-routine style, with the lowest overhead possible given that virtual memory is in use on the machine. The reply from the destination process is similarly implemented as a "direct" context switch back to the sender process.

Although LRPC may appear to be as costly as normal RPC in the local case, the approach actually achieves substantial savings. First, a normal RPC is implemented by having the client program perform a message send followed by a separate message receive operation, which blocks. Thus, two system calls occur, with the message itself being copied into the kernel's data space, or (if shared memory is exploited) a message descriptor being constructed in the kernel's data space. Meanwhile, the destination process will have issued a receive request and would often be in a blocked state. The arrival of the message makes the destination process runnable, and on a uniprocessor this creates a scheduling decision, since the sender process is also runnable in the first stage of the algorithm (when it has sent its request and not yet performed the subsequent receive operation). Thus, although the user might expect the sender to issue its two system calls and then block, causing the scheduler to run and activate the destination process, other sequences are possible. If the scheduler runs right after the initial send operation, it could context switch to the RPC server leaving the client runnable. It is now possible that a context switch back to the client will occur, and then back to the server again, before the server replies. The same sequence may then occur when the reply is finally sent.

We thus see that a conventional operating system requires four system calls to implement an LRPC operation, and that although a minimum of two context switches must occur, it is easily possible for an additional two context switches to take place, and if the execution of the operating system scheduler represents a significant cost, the scheduler may run two or more times more than the minimum. All of these excess operations are potentially costly.

Accordingly, LRPC is implemented using a special system call whereby the client process combines its send and receive operations into a single request, and the server (which will normally delay waiting for a new RPC request after replying to the client) issues the reply and subsequent receive as a single request. Moreover, execution of the scheduler is completely bypassed.

As in the case of RPC, the actual performance figures for LRPC are of limited value because processor speeds and architectures have been evolving so rapidly. One can get a sense of the improvment by looking at the number of instructions required to perform an LRPC. Recall that the Schroeder and Burrows study had found that thousands of instructions were required to issue an RPC. In contrast, the LRPC team calculated that only a few hundred instructions are "necessarily" required to perform an LRPC — a small enough number to make such factors as TLB misses (caused when the hardware cache associated with the virtual memory mapping system is flushed) rise to have an important impact on performance. LRPC was, in any case, somewhat more expensive than the theoretical minimum: about 50% slower measured in terms of round-trip latency or instructions executed for a null procedure call. Nonetheless, this represents a factor of at least five when compared to the performance of typical RPC in the local case, and ten or more when the approach is compared to the performance of a fairly heavy-weight vendor supported RPC package.

So dramatic is this effect that some operating systems vendors began to support LRPC immediately after the work was first reported. Others limited themselves to fine tuning their existing implementations, or improving the hardware used to connect their processors to the network. At the time of this writing, RPC performances have improved somewhat but faster processors are no longer bringing commensurate improvements in RPC performance. Vendors tend to point out that RPC performance, by itself, is only one of many factors that enter into overall system performance, and that optimizing this one case to an excessive degree can bring diminishing returns. They also argue for generality even in the local case, and hence that LRPC is undesirable because it requires a different RPC implementation than for the remote case and thus increases the complexity of the operating system for a scenario that may not be as common in commercial computing settings as it seems to be in academic research laboratories.

To some degree, these points are undoubtably valid ones: when an RPC arrives at a server, the program that will handle it may need to be scheduled, it may experience page faults, buffering and caching issues can severely impact its performance, and so forth. On the other hand, the performance of a null RPC or LRPC is entirely a measure of operating system overhead, and hence is "wasted time" by any reasonable definition. Moreover, the insights gained in LRPC are potentially applicable to other parts of the operating system: Bershad, for example, on to demonstrate that the same idea can be generalized using a notion of *thread activations* and *continuations*, with similarly dramatic impact on other aspects of operating system performance. This work seems not to have impacted the commercial operating systems community, at least at the time of this writing.

8.2 Fbuf's and the xKernel Project

During the same period, the University of Arizona, under Larry Peterson, developed a series of innovative operating system extensions for high performance communication. Most relevant to the topic of this chapter are the x-Kernel, a stand-alone operating system for developing high speed communications protocols, and the *fbufs* architecture [DP93], which is a general purpose technique for optimizing stack-structured protocols to achieve high performance. Both pieces of work were done out of the context of any particular operating system, but are potentially applicable to most standard vendor-supported operating systems.

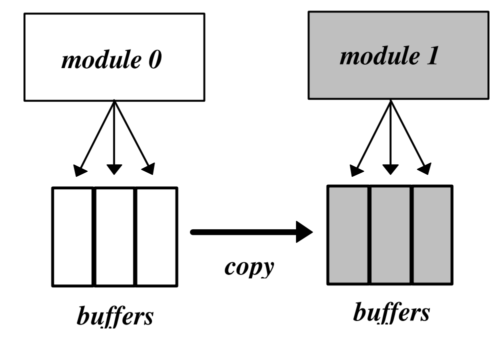


Figure 8-1: In a conventional layered architecture, as messages pass from layer to layer (here shown from left to right), messages and headers may need to be copied repeatedly. This contributed to high overhead. In this illustration, the white and gray buffers are independent regions in virtual memory.

The x-Kernel [PHMA89] is an operating system dedicated to the implementation of network protocols for experimental research on performance, flow-control, or other issues. The assumption that x-Kernel applications are purely communication-oriented greatly simplified the operating system design, which confines itself to addressing those issues encountered in the implementation of protocols, while omitting support for elaborate virtual memory mechanisms, special purpose file systems, and many of the other operating facilities that are considered mandatory in modern computing environments.

Recall from the early chapters of this text that many protocols have a layered structure, with the different layers having responsibility for different aspects of the overall communication abstraction. In x-Kernel, protocols having a layered structure are represented as a partially ordered graph of modules. The application process involves a protocol by issuing a procedure call to one of the root nodes in such a graph, and control then flows down the graph as the message is passed from layer to layer. x-Kernel includes built-in mechanisms for efficiently representing messages and managing their headers, and for dynamically restructuring the protocol graph or the route that an individual message will take, depending upon the state of the protocols involved and the nature of the message. Other x-Kernel features include a thread-based execution model, memory management tools, and timer mechanisms.

Using the x-Kernel, Peterson implemented several standard RPC and stream protocols, demonstrating that his architecture was indeed powerful enough to permit a variety of such protocols to co-exist, and confirming its value as an experimental tool. Layered protocol architectures are often thought to be inefficient but Peterson suggested a number of design practices that, in his experience, avoided overhead and permitted highly modular protocol implementations to perform as well as the original monolithic protocols on which his work was based. (Later, researchers such as Tennenhouse confirmed both that standard implementations of layered protocols, particularly in the UNIX streams architecture, have potentially high overheads, but also that appropriate design techniques can be used to greatly reduce these costs).

Peterson’s interest in layered protocols subsequently lead him to look at performance issues associated with layered or pipelined architectures, in which modules of a protocol operate in protected memory regions (Figure 8-1). To a limited degree, systems like UNIX and NT have an architecture such as this; UNIX streams, for example, are based on a modular architecture that is supported directly within the kernel. As an example, an incoming message is passed up a stack that starts with the device driver and then includes each of the streams modules that have been “pushed” onto the streams connection, terminating finally in a cross-address-space transfer of control to the application program. UNIX programmers think of such a structure as a form of “pipe” implemented directly in the kernel. Unfortunately, like a pipe, a stream can involve significant overhead.

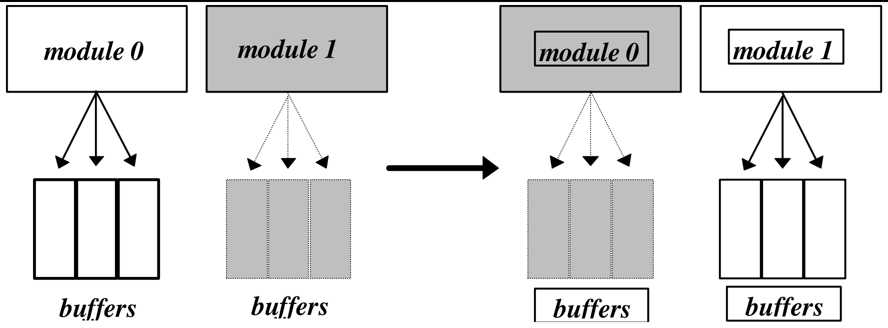


Figure 8-2: In Peterson's scheme, the buffers are in fact shared using virtual memory, exploiting protection features to avoid risk of corruption. To "pass" a buffer, access to it is enabled in the destination address space and disabled in the sender's address space. (Above, the white buffers represent real pointers and the gray ones represent invalid page-table entries pointing to the same memory regions but with access disabled). When the buffer finally reaches the last module in the pipeline it is freed and reallocated for a new message arriving from the left. Such an approach reduces the overhead of layering to the costs associated with manipulation of the page table entries associated with the modules comprising the pipeline.

Peterson's *fbufs* architecture focuses on the handling of memory in pipelined operating systems contexts such as these. An fbuf is a memory buffer for use by a protocol; it will typically contain a message or a header for a message. The architecture concerns itself with the issue of mapping such a buffer into the successive address spaces within which it will be accessed, and with the protection problems that arise if modules are to be restricted so that they can only operate on data that they "own". The basic approach is to cache memory bindings, so that a protocol stack that is used repeatedly can reuse the same memory mappings for each message in a stream of messages. Ideally, the cost of moving a packet from one address space to another can be reduced to the flipping of a protection bit in the address space mappings of the sending and receiving modules (Figure 8-2). The method completely eliminates copying, while retaining a fairly standard operating system structure and protection boundaries.

8.3 Active Messages

At the University of California, Berkeley, and Cornell University, researchers have explored techniques for fast message passing in parallel computing systems. Culler and von Eicken observed that operating system overheads are the dominant source of overhead in message-oriented parallel computing systems [ECGS92, TL93]. Their work resulted in an extremely aggressive attack on communication costs, in which the application interacts directly with an I/O device and the overhead for sending or receiving a message can be reduced to as little as a few instructions. The CPU and latency overhead of an operating system is slashed in this manner, with important impact on the performance of parallel applications. Moreover, as we will see below, similar ideas can be implemented in general purpose operating systems.

An *active message* is a type of message generated within a parallel application that takes advantage of knowledge that the program running on the destination node of a parallel computer is precisely the same as the program on the source node to obtain substantial performance enhancements. In the approach, the sender is able to anticipate much of the work that the destination node would normally have to do if the source and destination were written to run on general purpose operating systems. Moreover, because the source and destination are the same program, the compiler can effectively short-circuit much of the work and overhead associated with mechanisms for general-purpose message generation and for dealing with heterogeneous architectures. Finally, because the communications hardware in parallel computers does not lose messages, active messages are designed for a world in which message loss and processor failure do not occur.

The basic approach is as follows. The sender of a message generates the message in a format that is preagreed between the sender and destination. Because the destination is running the same program as the sender and is running on the same hardware architecture, such a message will be directly interpretable by the destination without any of the overhead for describing data types and layout that one sees in normal RPC environments. Moreover, the sender places the address of a handler for this particular class of message into the header of the message. That is, a program running on machine *A* places an address of a handler that resides within machine *B* directly into the message. On the reception machine, as the message is copied out of the network interface, its first bytes are already sufficient to transfer control to a handler compiled specifically to receive messages of this type. This reduces the overhead of communication from the tens of thousands of instructions common on general purpose machines to as few as five to ten instructions. In effect, the sender is able to issue a procedure call directly into the code of the destination process, with most of the overhead being that associated with triggering an interrupt on the destination machine and with copying data into the network on the sending side and out of the network on the receiving side. In some situations (for example, when the destination node is idle and waiting for an incoming request) even the interrupt can be eliminated by having the destination wait in a tight polling loop.

Obviously, active messages make sense only if a single application is loaded onto multiple nodes of a parallel computer, such as the CM5 or SP2, and hence has complete trust in those programs and accurate knowledge of the memory layout of the nodes with which it communicates. In practice, the types of systems that use the approach normally have identical programs running on each node. One node is selected as the "master" and controls the computation, while the other nodes, its "slaves", take actions on the orders of the master. The actual programming model visible to the user is one in which a sequential program initiates parallel actions by invoking parallel "operations" or procedures, which have been programmed to distribute work among the slaves and then to wait for them to finish computing before taking the next step. This model is naturally matched to active messages, which can now be viewed as optimizing normal message passing to take advantage of the huge amount of detailed information available to the system regarding the way that messages will be handled. In these systems, there is no need for generality, and generality proves to be expensive. Active messages are a general way of

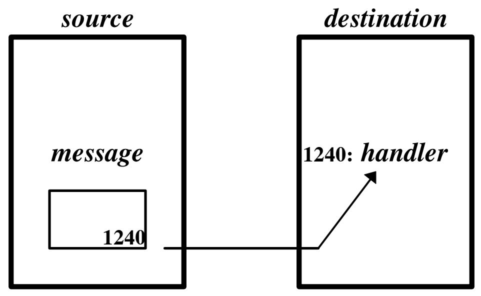


Figure 8-3: An active message includes the address of the handler to which it should be passed directly in the message header. In contrast with a traditional message passing architecture, in which such a message would be copied repeatedly through successively lower layers of the operating system, an active message is copied directly into the network adapter by the procedure that generates it in the application program, and is effectively transferred directly to the application-layer handler on the receiving side with no additional copying. Such a "zero copy" approach reduces communication latencies to a bare minimum and eliminates almost all overhead on the messages themselves. However, it also requires a high level of mutual trust and knowledge between source and destination, a condition that is more typical of parallel supercomputing applications than general distributed programs.

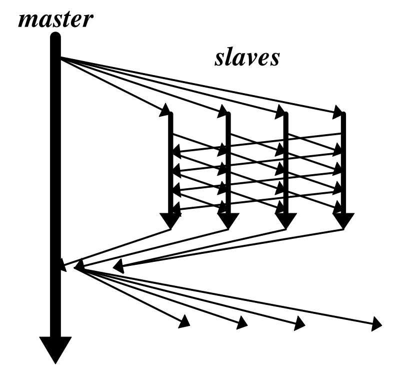


Figure 8-4: A typical parallel program employs a sequential master thread of control that initiates parallel actions on slave processors and waits for them to complete before starting the next computational step. While computing, the slave nodes may exchange messages, but this too tends to be both regular and predictable. Such applications match closely with the approach to communication used in Active Messages, which trades generality for low overhead and simplicity.

optimizing to extract the maximum performance out of the hardware by exploiting this prior knowledge.

Active messages are useful in support of many programming constructs. The approach can be exploited to build extremely inexpensive RPC interactions, but is also applicable to direct language support for data replication or parallel algorithms in which data or computation is distributed over the modes of a parallel processor. Culler and von Eicken have explored a number of such options, and reported particular success with language-based embedding of active messages within a parallel version of the C programming language they call “split C”, and in a data-parallel language called ID-90.

8.4 Beyond Active Messages: U-Net

At Cornell University, Von Eicken has continued the work start started in his study of Active Messages, looking for ways of applying the same optimizations in general purpose operating systems connected to shared communication devices. U-Net is a communications architecture designed for use within a standard operating system such as UNIX or NT, and is intended to provide the standard protection guarantees taken for granted in these sorts of operating systems [EBBV95]. These guarantees are provided, however, in a way that imposes extremely little overhead relative to the performance that can be attained in a dedicated application that has direct control over the communications device interface. U-Net gains this performance using an implementation that is split between traditional software functionality integrated into the device driver, and non-traditional functionality implemented directly within the communications controller interfaced to the communications device. Most controllers are programmable, hence the approach is more general than it may sound, although it should also be acknowledged that existing systems very rarely reprogram the firmware of device controllers to gain performance!

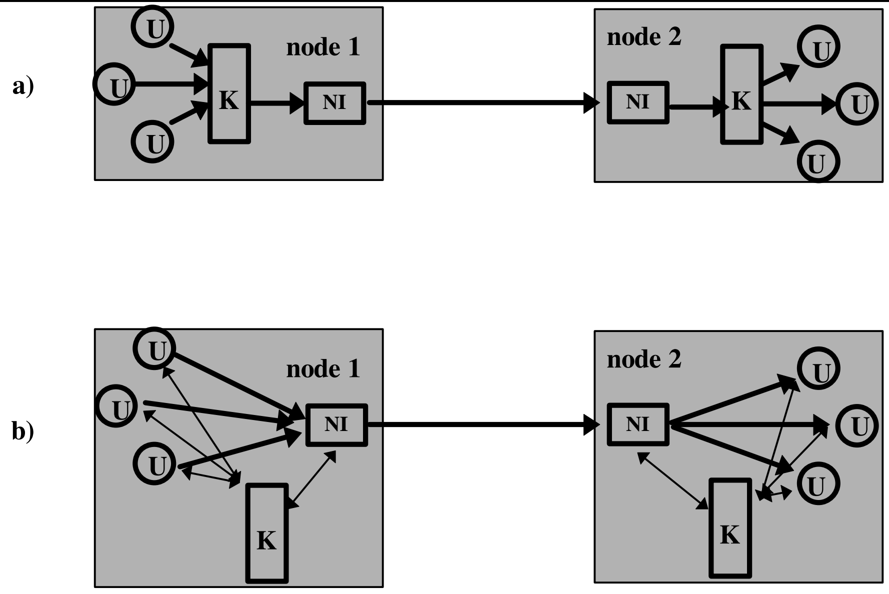


Figure 8-5: In a conventional communications architecture, all messages pass through the kernel before reaching the I/O device (a), resulting in high overheads. U-Net bypasses the kernel for I/O operations (b), while preserving a standard protection model.

The U-Net system starts with an observation we have made repeatedly in prior chapters, namely that the multiple layers of protocols and operating system software between the application and the communication wire represent a tremendous barrier to performance, impacting both latency and throughput. U-Net overcomes these costs by restructuring the core operating system layers that handle such communication so that channel setup and control functions can operate "out of band", while the application interacts directly with the device itself. Such a direct path results in minimal latency for the transfer of data from source to destination, but raises significant protection concerns: if an application can interact directly with the device, there is no obvious reason that it will not be able to subvert the interface to violate the protection on memory controlled by other applications or break into communication channels that share the device but were established for other purposes.

The U-Net architecture is based on a notion of a *communications segment,* which is a region of memory shared between the device controller and the application program. Each application is assigned a set of pages within the segment for use in sending and receiving messages, and is prevented from accessing pages not belong to it. Associated with the application are three queues: one pointing to received messages, one to outgoing messages, and one to free memory regions. Objects in the communication segment are of fixed size, simplifying the architecture at the cost of a small amount of overhead.

Each of these communication structures is bound to a U-Net channel, which is a communication session for which permissions have been validated, linking a known source to a known destination over an established ATM communication channel. The application process plays no role in specifying the hardware communication channels to which its messages will be sent: it is restricted to writing in memory buffers that have been allocated for its use and updating the send, receive and free queue appropriately. These restrictions are the basis of the U-Net protection guarantees cited earlier.

U-Net maps the communication segment of a process directly into its address space, pinning the pages into physical memory and disabling the hardware caching mechanisms so that updates to a segment will be applied directly to that segment. The set of communication segments for all the processes using U-Net is mapped to be visible to the device controller over the I/O bus of the processor used; the controller can thus initiate DMA or direct memory transfers in and out of the shared region as needed and without delaying for any sort of setup. A limitation of this approach is that the I/O bus is a scarce resource shared by all devices on a system, and the U-Net mapping excludes any other possible mapping for this region. However, some machines (for example, on the cluster-style multiprocessors discussed in Chapter 24), there are no other devices contending for this mapping unit, and dedicating it to the use of the communications subsystem makes perfect sense.

The communications segment is directly monitored by the device controller. U-Net accomplishes this by reprogramming the device controller, although it is also possible to imagine an implementation in which a kernel driver would provide this functionality. The controller watches for outgoing messages on the send queue; if one is present, it immediately sends the message. The delay between when a message is placed on the send queue and when sending starts is never larger than a few microseconds. Incoming messages are automatically placed on the receive queue unless the pool of memory is exhausted; should that occur, any incoming messages are discarded silently. To accomplish this, U-Net need only look at the first bytes of the incoming message, which give the ATM channel number on which it was transmitted. These are used to index into a table maintained within the device controller that gives the range of addresses within which the communications segment can be found, and the head of the receive and free queues are then located at a fixed offset from the base of the segment. To minimize latency, the addresses of a few free memory regions are cached in the device controller's memory

Such an approach may seem complex because of the need to reprogram the device controller. In fact, however, the concept of a programmable device controller is a very old one (IBM's channel architecture for the 370 series of computers already supported a similar "programmable channels" architecture nearly twenty years ago). Programmability such as this remains fairly common, and device drivers that download code into controllers are not unheard of today. Thus, although unconventional, the U-Net approach is not actually "unreasonable". The style of programming required is similar to that used when implementing a device driver for use in a conventional operating system..

With this architecture, U-Net achieves impressive application-to-application performance. The technology easily saturates an ATM interface operating at the OC3 performance level of 155Mbits/second, and measured end-to-end latencies through a single ATM switch are as low as 26us for a small message. These performance levels are also reflected in higher level protocols: versions of UDP and TCP have been layered over U-Net and shown capable of saturating the ATM for packet sizes as low as 1k bytes; similar performance is achieved with a standard UDP or TCP technology only for very large packets of 8k bytes or more. Overall, performance of the approach tends to be an order of magnitude or more better than with a conventional architecture for all metrics not limited by the raw bandwidth of the ATM: throughput for small packets, latency, and computational overhead of communication. Such results emphasize the importance of rethinking standard operating system structures in light of the extremely high performance that modern computing platforms can achieve.

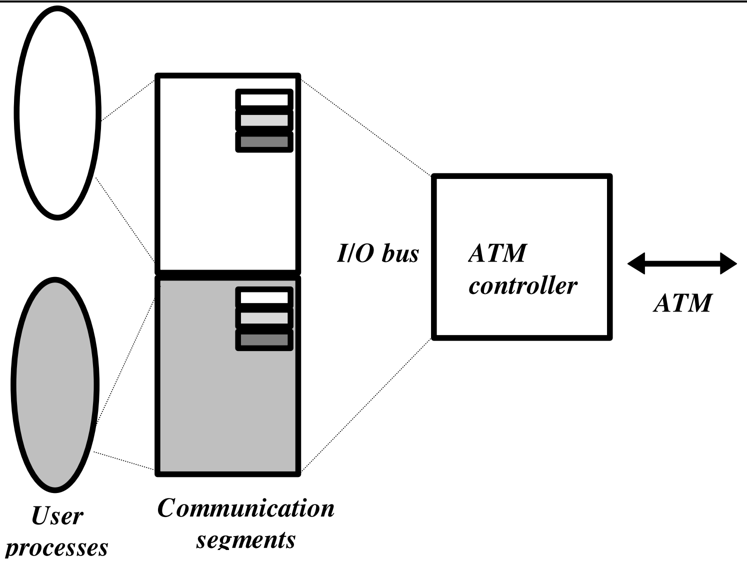


Figure 8-6: U-Net shared memory architecture permits the device controller to directly map a communications region shared with each user process. The send, receive and free message queues are at known offsets within the region. The architecture provides strong protection guarantees and yet slashes the latency and CPU overheads associated with communication. In this approach, the kernel assists in setup of the segments but is not interposed on the actual I/O path used for communication once the segments are established.

Returning to the point made at the start of this chapter, a technology like U-Net also improves the statistical properties of the communication channel. There are fewer places at which messages can be lost, hence reliability increases and, in well designed applications, may approach perfect reliability. The complexity of the hand-off mechanisms employed as messages pass from application to controller to ATM and back up to the receiver is greatly reduced, hence the measured latencies are much "tighter" than in a conventional environment, where dozens of events could contribute towards variation in latency. Overall, then, U-Net is not just a higher performance communication architecture, but is also one that is more conducive to the support of extremely reliable distributed software.

8.5 Protocol Compilation Techniques

U-Net seeks to provide very high performance by supporting a standard operating system structure in which a non-standard I/O path is provided to the application program. A different direction of research, best known through the results of the SPIN project at University of Washington [BSPS95], is concerned with building operating systems that are dynamically extensible through application programs coded in a special type-safe language and linked directly into the operating system at runtime. In effect, such a technology compiles the protocols used in the application into a form that can be executed close to the device driver. The approach results in speedups that are impressive by the standards of conventional operating systems, although less dramatic than those achieved by U-Net.

The key idea in SPIN is to exploit dynamically loadable code modules to place the communications protocol very close to the wire. The system is based on Modula-3, a powerful modern programming language similar to C++ or other modular languages, but "type safe". Among other guarantees, type safety implies that a SPIN protocol module can be trusted not to corrupt memory or to leak dynamically allocated memory resources. This is in contrast with, for example, the situation for a streams module, which must be "trusted" to respect such restrictions.

SPIN creates a runtime context within which the programmer can establish communication connections, allocate and free messages, and schedule lightweight threads. These features are sufficient to support communications protocols such as the ones that implement typical RPC or streams modules, as well as for more specialized protocols such as might be used to implement file systems or to maintain cache consistency. The approach yields latency and throughput improvements of as much as a factor of two when compared to a conventional user-space implementation of similar functionality. Most of the benefit is gained by avoiding the need to copy messages across address space boundaries and to cross protection boundaries when executing the short code segments typical of highly optimized protocols. Applications of SPIN include support for streams-style extensibility in protocols, but also less traditional operating systems features such as distributed shared memory and file system paging over an ATM between the file system buffer pools of different machines.

Perhaps more significant, a SPIN module has control over the conditions under which messages are dropped because of a lack of resources or time to process them. Such control, lacking in traditional operating systems, permits an intelligent and controlled degradation if necessary, a marked contrast with the more conventional situation in which as load gradually increases, a point is reached where the operating system essentially collapses, losing a high percentage of incoming and outgoing messages, often without indicating that any error has occurred.

Like U-Net, SPIN illustrates that substantial performance gains in distributed protocol performance can be achieved by concentrating on the supporting infrastructure. Existing operating systems remain "single-user" centric in the sense of having been conceived and implemented with dedicated applications in mind. Although such systems have evolved successfully into platforms capable of supporting distributed applications, they are far from optimal in terms of overhead imposed on protocols, data loss characteristics, and length of the I/O path followed by a typical message on its way to the wire. As work such as this enters the mainstream, significant reliability benefits will spill over to end-users, who often experience the side-effects of the high latencies and loss rates of current architectures as sources of unreliability and failure.

8.6 Related Readings

For work on kernel and microkernel architectures for high speed communication: Ameoba [MRTR90, RST88, RST89]. Chorus [AGHR89, RAAB88a, RAAB88b]. Mach [RAS86]. QNX [Hil92]. Sprite [OCDN88]. Issues associated with the performance of threads are treated in [ABLL91]. Packet filters are discussed in the context of Mach in [MRA87]. The classic paper on RPC cost analysis is [SB89], but see also [CT87]. TCP cost analysis and optimizations are presented in [CJRS89, Jac88, Jac90, KF93, Jen90]. Lightweight RPC is treated in [BALL89]. Fbufs and the xKernel in [DP83, PHMA89, AP93]. Active Messages are covered in [ECGS92, TL93] and U-Net in [EBBV95]. SPIN is treated in [BSPS95].

Part II: The World Wide Web

This second part of the textbook focuses on the technologies that make up the World Wide Web, which we take in a general sense that includes internet email and "news" as well as the Mosaic-style of network document browser that has seized the public attention. Our treatment seeks to be detailed enough to provide the reader with a good understanding concerning the key components of the technology base and the manner in which they are implemented, but without going to such an extreme level of detail as to lose track of our broader agenda, which is to understand how reliable distributed computing services and tools can be introduced into the sorts of critical applications that may soon be placed on the Web.

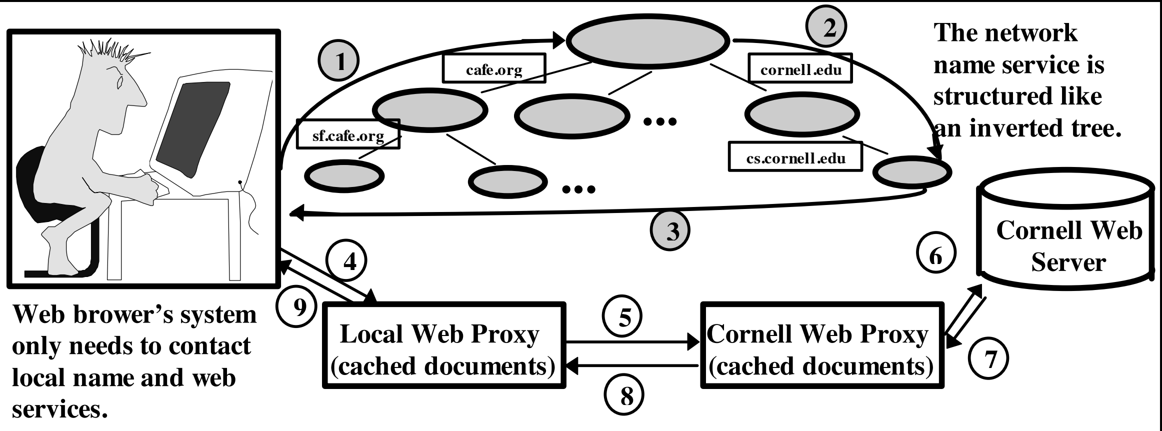
9. The World Wide Web

As recently as 1992 or 1993, it was common to read of a coming revolution in communications and computing technologies. Authors predicted a future information economy, the emergence of digital libraries and newspapers, the prospects of commerce over the network, and so forth. Yet the press was also filled with skeptical articles, suggesting that although there might well be a trend towards an information superhighway, it seemed to lack on-ramps accessible to normal computer users.

In an astonishingly short period of time, this situation has reversed itself. By assembling a relatively simple client-server application using mature, well-understood technologies, a group of researchers at CERN and at the National Center for Supercomputing Applications (NCSA) developed system for downloading and displaying documents over a network. They employed an object-oriented approach in which their display system could be programmed to display various types of objects: audio, digitized images, text, hypertext documents represented using the hypertext markup language (a standard for representing complex documents), and other data types. They agreed upon a simple resource location scheme, capable of encoding the information needed to locate an object on a server and the protocol with which it should be accessed. Their display interface integrated these concepts with easily used, powerful, graphical user interface tools. And suddenly, by pointing and clicking, a completely unsophisticated user could access a rich collection of data and documents over the internet. Moreover, authoring tools for hypertext documents already existed, making it surprisingly easy to create elaborate graphics and sophisticated hypertext materials. By writing simple programs to track network servers, checking for changed content and following hypertext links, substantial databases of web documents were assembled, against which sophisticated information retrieval tools could be applied. Overnight, the long predicted revolution in communications took place.

Two years later, there seems to be no end to the predictions for the potential scope and impact of the information revolution. One is reminded of the early days of the biotechnology revolution, during which dozens of companies were launched, fortunes were earned, and the world briefly overlooked the complexity of the biological world in its unbridled enthusiasm for a new technology. Of course, initial hopes can be unrealistic. A decade or so later, the biotechnology revolution is beginning to deliver on some of its initial promise, but the popular press and the individual in the street have long since become disillusioned.

The biotechnology experience highlights the gap that often forms between the expectations of the general populace, and the deliverable reality of a technology area. We face a comparable problem in distributed computing today. On the one hand, the public seems increasingly convinced that the information society has arrived. Popular expectations for this technology are hugely inflated, and it is being deployed on a scale and rate that is surely unprecedented in the history of technology. Yet, the fundamental science underlying web applications is in many ways very limited. The vivid graphics and ease with which hundreds of thousands of data sources can be accessed obscures more basic technical limitations, which may prevent the use of the Web for many of the uses that the popular press currently anticipates.


1
2
3
4
5
6
7
8
9
cafe.org
cornell.edu
sf.cafe.org
cs.cornell.edu
The network name service is structured like an inverted tree.
Cornell Web Server
Web brower's system only needs to contact local name and web services.
Local Web Proxy (cached documents)
Cornell Web Proxy (cached documents)

The web operates like a postal service. Computers have "names" and "addresses," and communication is by the exchange of electronic "letters" (messages) between programs. Individual systems don't need to know how to locate all the resources in the world. Instead, many services, like the name service and web document servers, are structured to pass requests via local representatives, which forward them to more remote ones, until the desired location or a document is reached.

For example, to retrieve the web document www.cs.cornell.edu/Info/Projects/HORUS, a browser must first map the name of the web server, www.cs.cornell.edu, to an address. If the address is unknown locally, the request will be forwarded up to a central name server and then down to one at Cornell (1-3). The request to get the document itself will often pass through one or more web "proxies" on its way to the web server itself (4-9). These intermediaries save copies of frequently used information in short-term memory. Thus, if many documents are fetched from Cornell, the server address will be remembered by the local name service, and if the same document is fetched more than once, one of the web proxies will respond rapidly using a saved copy. The term *caching* refers to the hoarding of reused information in this manner.

Our web surfer looks irritated, perhaps because the requested server "is overloaded or not responding." This common error message is actually misleading because it can be provoked by many conditions, some of which don't involve the server at all. For example, the name service may have failed or become overloaded, or this may be true of a web proxy , opposed to the Cornell web server itself. The Internet addresses for any of these may be incorrect, or stale (e.g. if a machine has been moved). The Internet connections themselves may have failed or become overloaded.

Although caching dramatically speeds response times in network applications, the web does not track the locations of cached copies of documents, and offers no guarantees that cached documents will be updated. Thus, a user may sometimes see a stale (outdated) copy of a document. If a document is complex, a user may even be presented with an inconsistent mixture of stale and up-to-date information.

With wider use of the web and other distributed computing technologies, critical applications will require stronger guarantees. Such applications depend upon correct, consistent, secure and rapid responses. If an application relies on rapidly changing information, stale responses may be misleading, incorrect, or even dangerous, as in the context of a medical display in a hospital, or the screen image presented to an air-traffic controller.

One way to address such concerns is to arrange for cached copies of vital information such as resource addresses, web documents, and other kinds of data to be maintained consistently and updated promptly. By reliably replicating information, computers can guarantee rapid response to requests, avoid overloading the network, and avoid "single points of failure". The same techniques also offer benefits from scaleable parallelism, where incoming requests are handled cooperatively by multiple servers in a way that balances load to give better response times.

As we will see below, the basic functionality of the Web can be understood in terms of a large collection of independently operated servers. A web browser is little more than a graphical interface capable of issuing remote procedure calls to such a server, or using simple protocols to establish a connection to a server by which a file can be downloaded. The model is stateless: each request is handled as a separate interaction, and if a request times out, a browser will simply display an error message. On the other hand, the simplicity of the underlying model is largely concealed from the user, who has the experience of a "session" and a strong sense of continuity and consistency when all goes well. For example, a user who fills in a graphical form seems to be in a dialog with the remote server, although the server, like an NFS server, would not normally save any meaningful "state" for this dialog.

The reason that this should concern us becomes clear when we consider some of the uses to which web servers are being put. Commerce over the internet is being aggressively pursued by a diverse population of companies. Such commerce will someday take many forms, including direct purchases and sales between companies, and direct sales of products and information to human users. Today, the client of a web server who purchases a product provides credit card billing information, and trusts the security mechanisms of the browser and remote servers to protect this data from intruders. But, unlike a situation in which this information is provided by telephone, the Web is a shared packet forwarding system in which a number of forms of intrusion are possible. For the human user, interacting with a server over the Web may seem comparable to interacting to a human agent over a telephone. The better analogy, however, is to shouting out one's credit card information in a crowded train station.

The introduction of encryption technologies will soon eliminate the most extreme deficiencies in this situation. Yet data security alone is just one element of a broader set of requirements. As the reader should recall from the first chapters of this text, RPC-based systems have the limitation that when a timeout occurs, it is often impossible for the user to determine if a request has been carried out, and if a server sends a critical reply just when the network malfunctions, the contents of that reply may be irretrievably lost. Moreover, there are no standard ways to guarantee that an RPC server will be available when it is needed, or even to be sure that an RPC server purporting to provide a desired service is in fact a valid representative of that service. For example, when working over the Web, how can a user convince him or herself that a remote server offering to sell jewelry at very competitive prices is not in fact fraudulent? Indeed, how can the user become convinced that the web page for the bank down the street is in fact a legitimate web page presented by a legitimate server, and not some sort of a fraudulent version that has been maliciously inserted onto the Web? At the time of this writing, the proposed web security architectures embody at most partial responses to these sorts of concerns.

Full service banking and investment support over the Web is likely to emerge in the near future. Moreover, many banks and brokerages are developing web-based investment tools for internal use, in which remote servers price equities and bonds, provide access to financial strategy information, and maintain information about overall risk and capital exposure in various markets. Such tools also potentially expose these organizations to new forms of criminal activity, insider trading and fraud. Traditionally banks have kept their money in huge safes, buried deep underground. Here, one faces the prospect of prospect that billions of dollars will be protected primarily by the communications protocols and security architecture of the Web. We should ask ourselves if these are understood well enough to be trusted for such a purpose.

Web interfaces are extremely attractive for remote control of devices. How long will it be before such an interface is used to permit a plant supervisor to control a nuclear power plant from a remote location, or permit a physician to gain access to patient records or current monitoring status from home? Indeed, a hospital could potentially place all of its medical records onto web servers, including everything from online telemetry and patient charts to x-rays, laboratory data, and even billing. But when this development occurs, how will we know that hackers cannot, also, gain access to these databases, perhaps even manipulating the care plans for patients?

A trend towards critical dependence on information infrastructure and applications is already evident within many corporations. There is an increasing momentum behind the idea of developing "corporate knowledge bases" in which the documentation, strategic reasoning, and even records of key meetings would be archived for consultation and reuse. It is easy to imagine the use of a web model for such purposes, and this author is aware of several efforts directed to developing products based on this concept.

Taking the same idea one step further, the military sees the Web as a model for future information based conflict management systems. Such systems would gather data from diverse sources, integrating it and assisting all levels of the military command hierarchy in making coordinated, intelligent decisions that reflect the rapidly changing battlefield situation and that draw on continuously updated intelligence and analysis. The outcome of battles may someday depend on the reliability and integrity of information assets.

Libraries, newspapers, journals and book publishers are increasingly looking to the Web as a new paradigm for publishing the material they assemble. In this model, a subscriber to a journal or book would read it through some form of web interface, being charged either on a per-access basis, or provided with some form of subscription.

The list goes on. What is striking to this author is the extent to which our society is rushing to make the transition, placing its most critical activities and valuable resources on the Web. A perception has been created that to be a viable company in the late 1990's, it will be necessary to make as much use of this new technology as possible. Obviously, such a trend presupposes that web servers and interfaces are reliable enough to safely support the envisioned uses.

Many of the applications cited above have extremely demanding security and privacy requirements. Several involve situations in which human lives might be at risk if the envisioned Web application malfunctions by presenting the user with stale or incorrect data; in others, the risk is that great sums of money could be lost, a business might fail, or a battle lost. Fault-tolerance and guaranteed availability are likely to matter as much as security: one wants these systems to protect data against unauthorized access, but also to guarantee rapid and correct access by authorized users.

Today, reliability of the Web is often taken as a synonym for *data security*. When this broader spectrum of potential uses is considered, however, it becomes clear that reliability, consistency, availability and trustworthiness will be at least as important as data security if critical applications are to be *safely* entrusted to the Web or the Internet. Unfortunately, however, these considerations rarely receive attention when the decision to move an application to the Web is made. In effect, the enormous enthusiasm for the *potential* information revolution has triggered a great leap of faith that it has already arrived. And, unfortunately, it already seems to be too late to slow, much less reverse, this trend. Our only option is to understand how web applications can be made sufficiently reliable to be used safely in the ways that society now seems certain to employ them.

Unfortunately, this situation seems very likely to deteriorate before any significant level of awareness that there is even an issue here will be achieved. As is traditionally the case in technology areas, reliability considerations are distinctly secondary to performance and user-oriented functionality in the development of web services. If anything, the trend seems to a form of latter-day gold rush, in which companies are stampeding to be first to introduce the critical servers and services on which web commerce will depend. Digital cash servers, signature authorities, special purpose web search engines, and services that map from universal resource names to locations providing those services are a few examples of these new dependencies; they add to a list that already included such technologies as the routing and data transport layers of the internet, the domain name service, and the internet address resolution protocol. To a great degree, these new services are promoted to potential users on the basis of functionality, not robustness. Indeed, the trend at the time of this writing seems to be to stamp "highly available" or "fault-tolerant" or more or less any system capable of rebooting itself after a crash. As we have already seen, recovering from a failure can involve much more than simply restarting the failed service.

The trends are being exacerbated by the need to provide availability for "hot web sites", which can easily be swamped by huge volumes of requests from thousands or millions of potential users. To deal with such problems, web servers are turning to a variety of ad-hoc replication and caching schemes, in which the document corresponding to a particular web request may be fetched from a location other than its ostensible "home." The prospect is thus created of a world within which critical data is entrusted to web servers which replicate it for improved availability and performance, but without necessarily providing strong guarantees that the information in question will actually be valid (or detectably stale) at the time it is accessed. Moreover, standards such as HTTP V1/0 remain extremely vague as to the conditions under which it is appropriate to cache documents, and when they should be refreshed if they may have become stale.

Broadly, the picture would seem to reflect two opposing trends. On the one hand, as critical applications are introduced into the Web, users may begin to depend on the correctness and accuracy of web servers and resources, along with other elements of the internet infrastructure such as its routing layers, data transport performance, and so forth. To operate safely, these critical applications will often require a spectrum of behavioral *guarantees*. On the other hand, the modern internet offers guarantees in none of these areas, and the introduction of new forms of web services, many of which rapidly become indispensable components of the overall infrastructure, is only exacerbating the gap. Recalling our list of potential uses in commerce, banking, medicine, the military, and others, the potential for very serious failures becomes apparent. We are moving towards a world in which the electronic equivalents of the bridges that we traverse may collapse without warning, in which road signs may be out of date or intentionally wrong, and in which the agents with which we interact over the network may sometimes be clever frauds controlled by malicious intruders.

As a researcher, one can always adopt a positive attitude towards such a situation, identifying technical gaps as "research opportunities" or "open questions for future study." Many of the techniques presented in this textbook could be applied to web browsers and servers, and doing so would permit those servers to overcome some (not all!) of the limitations identified above. Yet it seems safe to assume that by the time this actually occurs, many critical applications will already be operational using technologies that are only superficially appropriate.

Short of some major societal pressure on the developers and customers for information technologies, it is very unlikely that the critical web applications of the coming decade will achieve a level of reliability commensurate with the requirements of the applications. In particular, we seem to lack a level of societal consciousness of the need for a reliable technical base, and a legal infrastructure that assigns responsibility for reliability to the developers and deployers of the technology. Lacking both the pressure to provide reliability and any meaningful notion of accountability, there is very little to motivate developers to focus seriously on reliability issues. Meanwhile, the prospect of earning huge fortunes overnight has created a near hysteria to introduce new Web-based solutions in every imaginable setting.

As we noted early in this textbook, society has yet to demand the same level of quality assurance from the developers of software products and systems as it does from bridge builders. Unfortunately, it seems that the negative consequences of this relaxed attitude will soon become all too apparent.

9.1 Related Readings

On the Web: [BCLF94, BCLF95, BCGP92, GM95a, GM95b]. There is a large amount of online material concerning the Web, for example in the archives maintained by Netscape Corporation [http://www.netscape.com].

10. The Major Web Technologies

This chapter briefly reviews the component technologies of the World-Wide-Web [BCGP92, BCLF94] (but not on some of the associated technologies, such as email and network bulletin boards, which are considered in Chapter 11). The Web draws on the basic client-server and stream protocols that were discussed earlier, hence there is a strong sense in which the issue here is how those technologies can be *applied* to a distributed problem, not the development of a new or different technology base. In the case of the Web, there are three broad technology areas that arise. A *web browser* is a program for interfacing to a *web server*. There are various levels of browsers but the most widely used are based on graphical windowing displays, which permit the display of textual material including sophisticated formatting directives, graphical images, and implement access through hypertext links on behalf of the user. Web browser's also have a notion of a object type, and will run the display program appropriate to a given type when asked to do so. This permits a user to download and replay a video image file, audio file, or other forms of sophisticated media. (Fancier display programs typically download access information only, then launch a viewer of their own that pulls the necessary data and, for example, displays it in real-time).

Web servers and the associated notion of web "proxies" (which are intermediaries that can act as servers by responding to queries using cached documents) represent the second major category of web technologies. This is the level at which issues such as coherent replication and caching arise, and in which the Web authentication mechanisms are currently implemented.

The third major technology area underlying the Web consists of the *search engines* that locate web documents and index them in various ways, implementing query-style access on behalf of a web user. These search engines have two "sides" to them: a user-interface side, in which they accept queries from a web user and identify web resources that match the specified request, and a document finding side, which visits web servers and follows hyperlinks to locate and index new documents. At present, few users think in terms of search engines as playing a "critical" role in the overall technology area. This could change, however, to the degree that individuals become dependent upon search engines to track down critical information and to report it promptly. One can easily imagine a future in which a financial analyst would become completely reliant upon such interfaces, as might a military mission planner, an air traffic controller, or a news analyst. If we believe that the Web will have the degree of impact that now seems plausible, such developments begin to seem very likely.

Looking to the future, these technologies will soon be supplanted by others. Security and authentication services, provided by various vendors, are emerging to play a key role in establishing trustworthy links between web users and companies from which they purchase services; these security features include data encryption, digital signatures with which the identity of a user can be validated, and tools for performing third-party validation of transactions whereby an intermediary trusted by two parties mediates a transaction between them. Digital cash and digital banks will surely emerge to play an important role in any future commercial digital market. Special purpose telecommunications service providers will offer servers that can be used to purchase telecommunications connections with special properties for conferences, remote teleaccess to devices, communication lines with guarantees of latency, throughput, or error rate, and so forth. Web implementations of "auction" facilities will permit the emergence of commodities markets in which large purchases of commodities can be satisfied through a process of bidding and bid matching. Completely digital stock exchanges will follow soon after. Thus, while the early use of the web is primarily focused on a paradigm of remote access and retrieval, the future of the web will come closer and closer to creating a virtual environment that emulates many of the physical abstractions on which contemporary society resides, while also introducing new paradigms for working, social interaction, and commerce. And these new electronic worlds will depend upon a wide variety of critical services to function correctly and reliably.

10.1 Hyper-Text Markup Language (HTML)

The Hyper-Text Markup Language, or HTML, is a standard for representing textual documents and the associated formatting information needed to display them. HTML is quite sophisticated, and includes such information as text formatting attributes (font, color, size, etc.), a means for creating lists, specifying indentation, and tools for implementing other standard formats. HTML also has conditional mechanisms, means for displaying data in a concise form that can later be expanded upon request by the user, and so forth. The standard envision various levels of compliance, and the most appropriate level for use in the Web has become a significant area of debate within the community. For brevity, however, we do not treat these issues in the present textbook.

HTML offers ways of *naming* locations in documents, and for specifying what are called *hypertext links* or *meta-links*. These links are textual representations of a document, a location in a document, or a "service" that the reader of a document can access. There are two forms of HTML links: those representing embedded documents, which are automatically retrieved and displayed when the parent document is displayed, and conditional links, which are typically shown in the form of some sort of "button" that the user can select to retrieve the specified object. These buttons can be true buttons, regions within the document text (typically highlighted in color and underlined), or regions of a graphical image. This last approach is used to implement touch-sensitive maps and pictures.

10.2 Virtual Reality Markup Language (VRML)

At the time of this writing, a number of proposals have emerged for VRML languages, which undertake to represent virtual reality "worlds" -- three-dimensional or interactive data structures in which browsing has much of the look and feed of navigation in the real world. Although there will certainly be a vigorous debate in this area before standards emerge, it is easy to imagine an evolutionary path whereby interaction with the Web will become more and more like navigation in a library, a building, a city, or even the world.

It is entirely likely that by late in the 1990's, Web users who seek information about hotels in Paris, France will simply fly there through a virtual reality interface, moving around animated scenes of Paris and even checking the rooms that they are reserving, all from a workstation or PC. An interactive agent, or "avatar", may welcome the visitor and provide information about the hotel, speaking much like the talking heads already seen on some futuristic television shows. Today, obtaining the same information involves dealing with flat 2-dimensional Web servers that present HTML documents to their users, and with flat text-oriented retrieval systems; both are frequently cited as important impediments to wider use of the Web. Yet, a small number of innovative research groups and companies are already demonstrating VRML systems and language proposals.

Unfortunately, at the time of this writing the degree of agreement on VRML languages and interfaces was still inadequate to justify any extended treatment in the text. Thus although the author is personally convinced that VRML systems may represent the next decisive event in the trend towards widespread adoption of the Web, there is little more that can be said about these systems except that they represent an extremely important development that merits close attention.

10.3 Universal Resource Locators (URLs)

When a document contains a hypertext link, that link takes the form of a *universal resource locator,* or URL. A URL specifies the information needed by a web server to track down a specified document. This typically consists of the protocol used to find that document (i.e. "ftp" or "http", the hypertext transfer protocol), the name of the server on which the document resides (i.e.

"www.cs.cornell.edu"), an optional internet port number to use when contacting that server (otherwise the default port number is used), and a path name for the resource in question relative to the default for that server. The syntax is somewhat peculiar for historical reasons that we will not discuss here.

For example, Cornell's Horus research project maintains a world-wide-web page with URL **http://www.cs.cornell.edu/Info/Projects/Horus.html***,* meaning that the hypertext transfer protocol should be used over the Internet to locate the server **www.cs.cornell.edu** and to connect to it using the default port number. The document **Info/Projects/Horus.html** can be found there. The extension **.html** tells the web browser that this document contains HTML information and should be displayed using the standard **html** display software. The "://" separator is a form of syntactic convention and has no special meaning. Variant forms of the URL are also supported; for example, if the protocol and machine name are omitted, the URL is taken to represent a path. Such a path can be a network path ("//" followed by a network location), an absolute path ("/" followed by a file name in the local file system), or a relative path ("a file name which does not start with a "/", and which is interpreted relative to the directory from which the browser is running). In some cases a port number is specified after the host name; if it is omitted (as above), port number 80 is assumed.

Most web users are familiar with the network path form of URL, because this is the form that is used to retrieve a document from a remote server. Within a document, however, the "relative path" notation tends to be used heavily, so that if a document and its subdocuments are all copied from one server to another, the subdocuments can still be found.

10.4 Hyper-Text Transport Protocol (HTTP)

The hypertext transport protocol is one of the standard protocols used to retrieve documents from a web server [BCLF95]. In current use, http and ftp are by far the most commonly used file transfer protocols, and are supported by all web browsers of which this author is familiar. In the future, new transfer protocols implementing special features or exploiting special properties of the retrieved object may be introduced. HTTP was designed to provide lightness (in the sense of ease of implementation) and speed, which is clearly necessary in distributed, collaborative, hypermedia applications. However, as the scale of use of the Web has expanded, and load upon it has grown, it has become clear that HTTP does not really provide either of these properties. This has resulted in a series of "hacks" that improve performance but also raise consistency issues, notably through the growing use of Web proxies that cache documents.

Web browsers typically provide extensible interfaces: new types of documents can be introduced, and new forms of display programs and transfer protocols are therefore needed to retrieve and display them. This requirement creates a need for flexibility at multiple levels: search, front-end update options, annotation, and selective retrieval. For this purpose, HTTP supports an extensible set of methods that are typically accessed through different forms of URL and different document types (extensions like .txt, .html, etc). The term URI (Universal Resource Indentifier) has become popular to express the idea that the URL may be a "locator" but may also be a form of "name" that indicates the form of abstract service that should be consulted to retrieve the desired document. As we will see shortly, this permits an HTTP server to construct documents upon demand, with content matched to the remote user's inquiry.

The hypertext transfer protocol itself is implemented using a very simple RPC-style interface, in which all messages are represented as human-readable ascii strings, although often containing encoded or even encrypted information. Messages are represented in the same way that internet mail passes data in messages. This includes text and also a form of encoded text called the Multipurpose Internet Mail Extensions or MIME (the HTTP version is "MIME-like" in the sense that it extends a normal MIME scheme with additional forms of encoding). However, HTTP can also be used as a generic protocol for contacting other sorts of document repositories, including document caches (these are often called "proxies"), gateways that may impose some form of firewall between the user and the outside world, and other servers that handle such protocols as Gopher, FTP, NNTP, SMTP, and WAIS. When this feature is used, the HTTP client is expected to understand the form of data available from the protocol it employs and to implement the necessary mechanisms to convert the resulting data into a displayable form and to display it to the user.

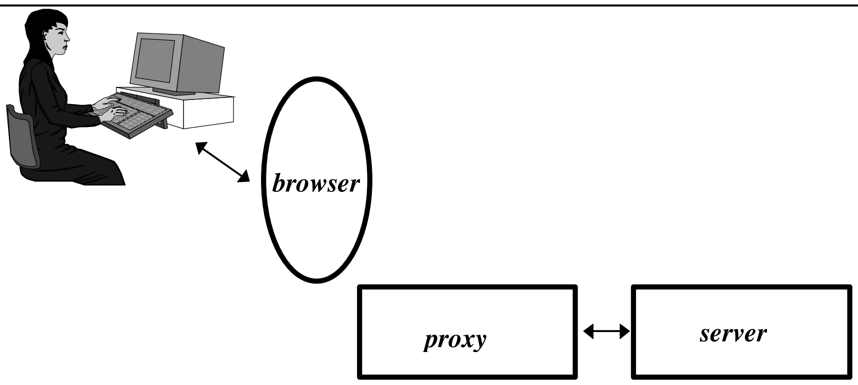


Figure 10-1: Components of a typical Web application. The user interacts with a graphical browser, which displays HTML documents and other graphical objects and issues HTTP commands to the servers on which objects needed by the user are stored. A proxy may be used to cache responses. Historically, HTTP applications have not fully specified what is or is not a "cacheable" response, hence the use of this feature varies depending upon the origin of the proxy. Individual browsers may be capable of specialized display behaviors, such as rasterized display of graphical images or execution of pseudo-code programs written in languages such as Java or Visual Basic. Although not shown above, there may be more than one level of proxy between the browser and server, and requests may "tunnel" through one or more firewalls before reaching the server. Moreover, messages passing over the Internet are relatively insecure and could be intercepted, read, and even modified on the path in either direction, if a Web security architecture is not employed.

In the normal case, when HTTP is used to communicate with a web server, the protocol employs a client-server style of request-response, operating over a TCP connection that the client makes to the server and later breaks after its request has been satisfied. Each request takes the form of a request method or "command", a URI, a protocol version identifier, and a MIME-like message containing special parameters to the request server. These may include information about the client, keys or other proofs of authorization, arguments that modify the way the request will be performed, and so forth. The server responds with a status line that gives the message's protocol version, and outcome code (success or one of a set of standard error codes), and then a MIME-like message containing the "content" associated with the reply. In normal use the client sends a single request over a single connection, and receives a single response back from the server. More complicated situations can arise if a client interacts with an HTTP server over a connection that passes through proxies which can cache replies, gateways, or other intermediaries; we return to these issues in Section 10.7.

HTTP messages can be compressed, typically using the UNIX compression tools "gzip" or "compress". Decompression is done in the browser upon receipt of a MIME-like message indicating that the body type has compressed content.

The HTTP commands consist of the following:

• *Get*. The get command is used to retrieve a document from a web server. Normally, the document URL is provided as an argument to the command, and the document itself is returned to the server in its response message. Thus, the command "GET //www.cs.cornell.edu/Info.html HTTP/1.0" could be used to request that the document "Info.html" be retrieved from "www.cs.cornell.edu", compressed and encoded into a MIME-like object, and returned to the requesting client. The origin of the resource is included but does not preclude caching: if a proxy sees this request it may be able to satisfy it out of a cache of documents that includes a copy of the Info.html previously retrieved from www.cs.cornell.edu. In such cases, the client will be completely unaware that the document came from the proxy and not the server that keeps the original copy.

There are some special cases in which a get command behaves differently. First, there are cases in which a server should calculate a new HTML document for each request. These are handled by specifying a URL that identifies a program in a special area on the web server called the cgi-bin area, and encodes arguments to the program in the pathname suffix (the reader can easily observe this behavior by looking at the pathname generated when a search request is issued to one of the major web search engines, such as Lycos or Yahoo). A web server that is asked to retrieve one of these program objects will instead run the program, using the pathname suffix as an argument, and creating a document as output in a temporary area which is then transmitted to the client. Many form-fill queries associated with web pages use this approach, as opposed to the "post" command which transmits arguments in a manner that requires slightly more sophisticated parsing and hence somewhat more effort on the part of the developer.

A second special case arises if a document has moved; in this case, the get command can send back a redirection error code to the client that includes the URL of the new location. The browser can either reissue its request or display a short message indicating *this document has moved* ***here***. A conditional form of *get* called *If-Modified-Since* can be used to retrieve a resource only if it has changed since some specified data, and is often used to refresh a cached object: if the object has not changed, minimal data is moved.

The get operation does not change the state of the server, and (in principle) the server will not need to retain any memory of the get operations that it has serviced. In practice many servers cheat on the rules in order to prefetch documents likely to be needed in future get operations, and some servers keep detailed statistics about the access patterns of clients. We will return to this issue below; it raises some fairly serious concerns both about privacy and security of web applications.

• *Head*. The head command is similar to get, but the server must not send any form of entity body in the response. The command is typically used to test a hypertext link for validity or to obtain accessibility and modification information about a document without actually retrieving the document. Thus, a browser that periodically polls a document for changes could use the head command to check the modification time of the document and only issue a get command if the document indeed has changed.

• *Post*. The post command is used to request that the destination server accept the information included in the request as a new "subordinate" of the resource designated by the path. This command is used for annotation of existing resources (the client "posts" a "note" on the resource), posting of a conventional message to an email destination, bulletin board, mailing list, or chat session, providing a block of data obtained through a form-fill, or extend a database or file through an "append" operation.

This set of commands can be extended by individual servers. For example, a growing number of servers support a subscription mechanism by which each update to a document will automatically be transmitted for as long as a connection to the server remains open. This feature is needed by services that dynamically send updates to displayed documents, for example to provide stock market quotes to a display that shows the market feed in real-time. However, unless such methods are standardized through the "Internet Task Force" they may only be supported by individual vendors. Moreover, special purpose protocols may sometimes make more sense for such purposes: the display program that displays a medical record could receive updates to the EKG part of the displayed "document", but it could also make a connection to a specified EKG data source and map the incoming data onto the part of the document that shows the EKG. The latter approach may make much more sense than one in which updates are received in HTTP format, particularly for data that is compressed in unusual ways or for which the desired quality of service of the communication channels involves unusual requirements or a special setup procedure.

Status codes play a potentially active role in HTTP. Thus, in addition to the standard codes ("created", "accepted", "document not found") there are codes that signify that a document has moved permanently or temporarily, providing the URL at which it can be found. Such a response is said to "redirect" the incoming request, but can also be used in load-balancing schemes. For example, certain heavily used web sites are implemented as clusters of computers. In these cases, an initial request will be directed to a load balancing server that redirects the request using a "temporary" URL to whichever of the servers in the cluster is presently least loaded. Because the redirection is temporary, a subsequent request will go back to the front-end server.

A curious feature of HTTP is that the client process is responsible both for opening *and for closing* a separate TCP connection for each command performed on the server. If retrieval of a document involves multiple get operations, multiple channels will be opened, one for each request. One might question this choice, since the TCP channel connection protocol represents a source of overhead that could be avoided if the browser were permitted to maintain connections for longer periods. Such an architecture is considered inappropriate, however, because of the potentially large number of clients that a server may be simultaneously handling. Thus, although it might seem that servers could maintain state associated with its attached channels, in practice is this not done. Even so, the solution can leave the server with a lot of resources tied up on behalf of channels. In particular, in settings where internet latencies are high (or when clients fail), servers may be left with a large number of open TCP connections, waiting for the final close sequence to be executed by the corresponding clients. For a heavily loaded server, these open connections represent a significant form of overhead.

10.5 Representations of Image Data

Several standards are employed to compress image data for storage in web servers. These include *GIF*, an encoding for single images, *MPEG* and *JPEG,* which encode video data consisting of multiple frames, and a growing number of proprietary protocols. Text documents are normally represented using html, but postscript is also supported by many browsers, as is the "rich text format" used by Microsoft's text processing products.

In the most common usage, GIF files are retrieved using a rasterized method in which a low quality image can be rapidly displayed and then gradually improved as additional information is retrieved. The idea is to start by fetching just part of the date (perhaps, every fourth raster of the image), and to interpolate between the rasters using a standard image interpolation scheme. Having finished this task, half of the remaining rasters will be fetched and the interpolation recomputing using this additional data; now, every other raster of the image will be based on valid data. Finally, the last rasters are fetched and the interpolation becomes unnecessary. The user is given the impression of a photographic image that gradually swims into focus. Depending on the browser used, this scheme may sweep from top of the image to bottom as a form of "wipe", or some sort of randomized scheme may be used. Most browsers permit the user to interrupt an image transfer before it finishes, so that a user who accidentally starts a very slow retrieval can work with the retrieved document even before it is fully available.

This type of retrieval is initiated using options to the "get" command, and may require compatibility between the browser and the server. A less sophisticated browser or server may not support rasterized retrieval, in which case the rasterization option to "get" will be ignored and the image displayed top to bottom in the standard manner. The most sophisticated browsers now on the market maintain a type of "device driver" which is used to customize their style of retrieval to the type of web server and code version number from which a document is retrieved.

In contrast to the approach used for GIF files, MPEG and JPEG files, and documents represented in formats other than HTML, are normally transferred to a temporary space on the user's file system, for display by an appropriate viewer. In these cases, the file object will typically be entirely transferred before the viewer can be launched, potentially resulting in a long delay before the user is able to see the video data played back or the contents of the text document.

The web is designed to be extensible. Each type of object is recognized by its file extension, and each web server is configured with *viewer* programs for each of these types. It is expected that new file types will be introduced over time, and new types of viewers developed to display the corresponding data. However, although such viewers can often be downloaded over the network, users should be extremely cautious before doing so. A web document "viewer" is simply a program that the user downloads and runs, and there is nothing to prevent that program from taking actions that have nothing at all to do with the ostensible display task. The program could be a form of virus or worm, or designed to damage the user's computer system or to retrieve data from it and send it to third parties. For this reason, the major vendors of web browsers are starting to offer libraries of certified viewers for the more important types of web data. Their browsers will automatically download these types of viewers, which are in some ways similar to dynamically loaded executables in a standard operating system. When the user attempts to configure a new and non-standard viewer, on the other hand, the browser may warn against this or even refuse to do so.

An important class of viewers are those that use their own data retrieval protocols to fetch complex image data. These viewers are typically launched using very small, compact image descriptions that can be understood as domain-specific URL's. Once started, the viewer uses standard windowing primitives to discover the location of its display window on the screen, and then begins to retrieve and display data into this location in real-time. The advantage of such an approach is that it avoids the need to download the full image object before it can be displayed. Since an image object may be extremely large, there are enormous advantages to such an approach, and it is likely that this type of specialized image display will become more and more common in the future.

10.6 Authorization and Privacy Issues

Certain types of resources require that the web browser authenticate its requests by including a special field, WWW-authorization field with the request. This field provides *credentials* containing the authentication information that will be used to decide if permission for the request should be granted. Credentials are said to be valid within a *realm*.

The basic HTTP authentication scheme is based on a model in which the user must present a user-id and password to obtain credentials for access to a realm [BCLF95]. The user-id and password are transmitted in a slightly obscured but insecure mode: they are translated to a representation called base64, encoded as an ascii string of digits, and sent over the connection to the server. This approach is only secure to the degree that the communication channel to the server is secure; if an intruder were to capture such an authorization request in transit over the network (for example by installing a "packet sniffer" at a gateway), the same information could later be presented to the same realm and server to authenticate access by the intruder. Nonetheless, the basic authentication scheme is required from all servers, including those that can operate with stronger protection. Browsers that communicate with a server for which stronger security is available will often warn the user before sending a message that performs basic authentication.

When transferring genuinely sensitive information, web applications typically make use of a trusted intermediary that provides session keys, using what is called public key encryption to authenticate channels and then a secret key encryption scheme to protect the data subsequently sent on that channel (the so-called *secure sockets layer* is described more fully in [IETF95, DB96]). At the core of this approach is a technology for publishing keys that can be used to encrypt data so that it can be read only by a process that holds the corresponding private key. The basic idea is that the public keys for services to be used by a client can be distributed to that client in some way that is hard to disrupt or tamper with, and the client can than create messages that are illegible to any process other than the desired server. A client that has created a key pair for itself can similarly publish its public key, in which case it will be able to receive messages that only it can read. Because public key cryptography is costly, the recommended approach involves using a public key handshake to generate a secret key with which the data subsequently exchanged on the channel can be encrypted; in this manner, a faster protocol such as DES or RC4 can be employed for any large objects that need to be transferred securely.

We will have more to say about security architectures for distributed systems in Chapter 19, and hence will not discuss any details here.

There are ways to attack this sort of security architecture, but they are potentially difficult to mount. If an intruder can break or steal the private keys used by the client or server, it may be possible to misrepresent itself as one or the other and initiate secured transactions at leisure. Another option is to attack the stored public key information, so as to replace a public key with a falsified one that would permit a faked version of a server to mimic the real thing. Realistically, however, these would both be a very difficult types of attack to engineer without some form of insider access to the systems on which the client and server execute, or an unexpectedly fast way of breaking the cryptographic system used to implement the session keys. In practice, it is generally believed that although the "basic" authentication scheme is extremely fragile, the stronger web security architecture should be adequate for most commercial transactions between individuals, *provided however that the computer on which the client runs can be trusted*. Whether the same schemes are adequate to secure transactions between banks, or military systems that transmit orders to the battlefield, remains an open question.

Web technologies raise a number of privacy issues that go beyond the concerns one may have about connection security. Many HTTP requests either include sensitive information such as authentication credentials, or include fields that reveal the identity of the sender, URI's of documents being used by the sender, or software version numbers associated with the browser or server. These forms of information all can be misused. Moreover, many users employ the same password for all their authenticated actions, hence a single "corrupt" server that relies on the basic authentication scheme might reveal a password that can be used to attack "secure" servers that use the basic scheme.

Web servers are often considered to be digital analogs of libraries. Within the United States, it is illegal for a library to maintain records of the documents that a client has examined in the past: only "current" locations of documents may be maintained in the records. Web servers that keep logs of accesses may thus be doing something that would be illegal if the server were indeed the legal equivalent of a library. Nonetheless, it is widely reported that such logging of requests is commonly done, often to obtain information on typical request patterns. The concern, of course, is that information about the private reading habits of individuals is concerned to be personal and protected in the United States, and logs that were gathered for a mundane purpose such as maintaining statistics on frequency of access to parts of a document base might be abused for some less acceptable purpose.

Access patterns are not the only issue here. Knowledge of a URI for a document within which a pointer to some other document was stored may be used to gain access to the higher level document, by "following the link" backwards. This higher level document may, however, be private and sensitive to the user who created it. With information about the version numbers of software on the browser or server, an intruder may be able to attack one or both using known security holes. A proxy could be subverted and modified to return incorrect information in response to "get" commands, or to modify data sent in "put" commands, or to replay requests (even encrypted ones), which will then be performed more than once to the degree that the server was genuinely stateless. These are just a few of the most obvious concerns that one could raise about HTTP authentication and privacy.

These considerations point to the sense in which we tend to casually trust web interfaces in ways that may be highly inappropriate. In a literal sense, use of the web is a highly *public* activity today: much of the information passed is basically insecure, and even the protection of passwords may be very limited. Although security is improving, the stronger security mechanisms are not yet standard. Even if one trusts the security protocol implemented by the Web, one must also trust many elements of the environment: for example, one may need to "trust" that the copy of a secure web browser that one has downloaded over the network wasn't modified in the network on the way to the user's machine, or modified on the server itself from which it was retrieved. How can the user be sure that the browser that he or she is using has not been changed in a way that will prevent it from following the normal security protocol? These sorts of questions turn out to lack good answers.

One thinks of the network as anonymous, but user-id information is present in nearly every message sent over it. Patterns of access can be tracked and intruders may be able to misrepresent a compromised server as one that is trusted using techniques that are likely to be undetectable to the user. Yet the familiarity and "comfort" associated with the high quality of graphics and easily used interfaces to web browsers and key services lulls the user into a sense of trust. Because the system "feels" private, much like a telephone call to a mail-order sales department, one feels safe in revealing credit card information or other relatively private data. With the basic authentication scheme of the Web, doing so is little different from jotting it down on the wall of a telephone booth. The secure authentication scheme is considerably better, but is not yet widely standard.

Within the Web community, the general view of these issues is that they represent fairly minor problems. The Web security architecture (the cryptographic one) is considered reasonably strong, and although the various dependencies cited above are widely recognized, it is also felt that do not correspond to gaping exposures or "show stoppers" that could prevent digital commerce on the Web from taking off. The laws that protect private information are reasonably strong in the United States, and it is assumed that these offer recourse to users who discover that information about themselves is being gathered or used inappropriately. Fraud and theft by insiders is generally believed to be a more serious problem, and the legal system again offers the best recourse to such problems. For these reasons, most members of the Web community would probably feel more concerned about overload, denial of services due to failure, and consistency than about security.

From the standpoint of the author of this textbook, though, the bottom line is not yet clear. It would be nice to believe that security is a problem of the past, but a bit more experience with the current web security architecture will be needed before one can feel confident that it has no unexpected problems that clever intruders might be able to exploit. In particular, it is troubling to realize that the current security architecture of the Web depends upon the integrity of software that will increasingly be running on unprotected PC platforms, and that may be have been downloaded from unsecured sites on the Web. While Java and other intepreted languages could reduce this threat, it seems unlikely to go away soon. In the current environment, it would be surprising *not* to see the emergence of computer viruses that specialize in capturing private keys and revealing them to external intruders without otherwise damaging the host system. This sort of consideration (and we will see a related problem when we talk about non-PC systems that depend upon standard file systems like NFS) can only engender some degree of skepticism about the near-term prospects for real security in the Web.

10.7 Web Proxy Servers

In Figure 10-1 a *server proxy* was shown between the browser and document server. Such proxies are a common feature of the world wide web, and are widely perceived as critical to the eventual scalability of the technology. A proxy is any intermediary process through which HTTP operations pass on their way to the server specified in the document URL. Proxies are permitted to cache documents or responses to certain categories of requests, and in future systems may even use cached information to dynamically construct responses on behalf of local users.

This leads to a conceptual structure in which each server can be viewed as surrounded by a ring of proxies that happen to be caching copies of documents associated with it (Figure 10-2). However, because the web is designed as a stateless architecture, this structure is not typically represented: one could deduce a possible structure from the log of requests to the server, but information is not explicitly maintained in regard to the locations of copies of documents. Thus, a web server would not typically have a means by which it could inform proxies that have cached documents when the primary copy changes. Instead, the proxies periodically refresh the documents they manage by using the “head” command to poll the server for changes, or the conditional “get” command to simply pull an updated copy if one is available.

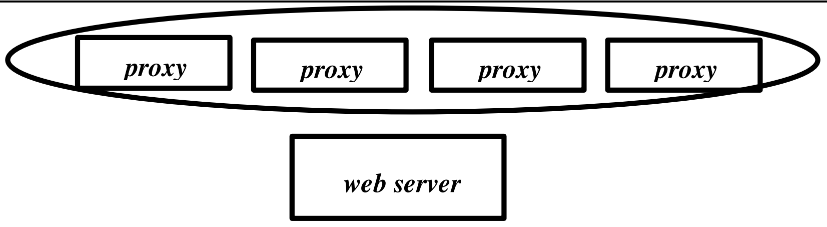


Figure 10-2: Conceptually, the proxies that cache a document form a distributed "process group", although this group would not typically be explicitly represented, a consequence of the stateless philosophy used in the overall web architecture.

In Chapters 13-16 this textbook we will be looking at techniques for explicitly managing groups of processes that need to coherently replicate data, such as web documents. These techniques could be used to implement coherent replication within a set of web proxies, provided that one is prepared to relax the stateless system architecture normally used between the proxies and the primary server. Looking to the future, it is likely that web documents will be more and more “dynamic” in many settings, making such coherency a problem of growing importance to the community selling web-based information that must be accurate to have its maximum value.

In the most common use of web proxies today, however, their impact is to increase availability at the cost of visible inconsistency when documents are updated frequently. Such proxies reduce load on the web server and are often able to respond to requests under conditions when a web server might be inaccessible, crashed, or overloaded. However, unless a web proxy validates every document before returning a cached copy of it, which is not a standard behavior, a proxy may provide stale data to its users for a potentially unbounded period of time, decreasing the perceived reliability of the architecture. Moreover, even if a proxy does refresh a cached record periodically, the Web potentially permits the use of multiple layers of proxy between the user and the server that maintains the original document. Thus, knowing that the local proxy has tried to refresh a document is not necessarily a strong guarantee of consistency. “Head” operations cannot be cached, hence if this command is used to test for freshness there is a reasonable guarantee that staleness can be detected. But all types of “get” commands can be cached, so even if a document is known to be stale, there may be no practical way to force an uncooperative proxy to pass a request through to the primary server.

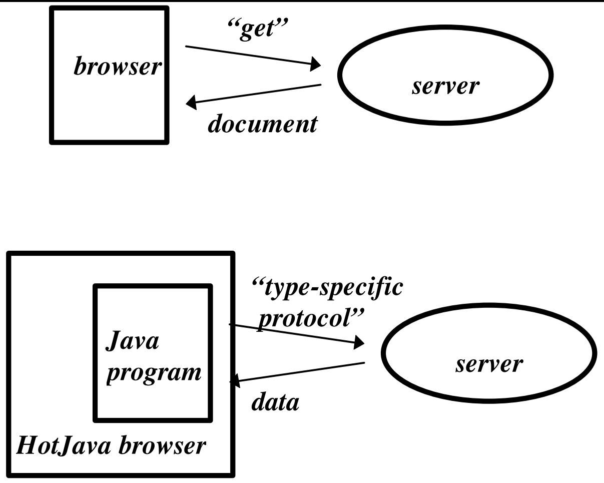


Figure 10-3: In a conventional Web interface, the user's requests result in retrieval of full documents (top). The browser understands HTML and can display it directly; for other types of documents it will copy the incoming object to a temporary location and then execute the appropriate display program. If an object type is unknown, the user may be forced to import potentially untrustworthy programs over the network. When using an agent language such as Java (bottom), the browser becomes an interpreter for programs that execute directly on the user's workstation and that can implement type-specific protocols for retrieving data over the network and displaying it (right). Not only does the security of the architecture improve (the HotJava browser is designed with many protection features), but the ability to execute a program on the user's workstation means that data transmission from server to client can be optimized in a type-specific way, interact with non-standard servers (that may use protocols other than HTTP), and dynamically retrieve data over a period of time (unlike the approach used for HTML, which involves retrieving a document just once and then breaking the connection).

10.8 Java, HotJava, and Agent Based Browsers

One way to think of an HTML document is as a form of program that the browser "executes" interpretively. Such a perspective makes it natural to take the next step and to consider sending a genuine program to the browser, which it could execute local to the user. Doing so has significant performance and flexibility benefits and has emerged as a major area of research. One way to obtain this behavior is to introduce new application-specific document types. When a user accesses such a document, his or her browser will download the associated data file and then run a type-specific display program to display its contents. If the type of the file is a new one not previously known to the browser, it will also download the necessary display program, which is called an "agent". But this is clearly a risky proposition: the agent may well display the downloaded data, but nothing prevents it from also infecting the host machine with viruses, scanning local files for sensitive data, or damaging information on the host.

Such considerations have resulted in research on new forms of *agent programming languages* [Rei94] that are safe and yet offer the performance and flexibility benefits of downloaded display code. Best known among the programming languages available for use in programming such display agents are SUN Microsystem's HotJava browser, which downloads and runs programs written in an object-oriented language called Java [GM95a, GM95b]. Other options also exist. The TCL/TK ("Tickle-Toolkit") language has become tremendously popular, and can be used to rapidly prototype very sophisticated display applications Ous94]. Many industry analysis predict that Visual Basic, an extremely popular programming language for designing interactive PC applications, will rapidly emerge as a major alternative to Java. Interestingly, all of these are *interpreted* languages. The security problems associated with importing untrustworthy code are increasingly causing companies that see the Web as their future to turn to interpretation as a source of protection against hostile intrusion into a machine on which a browser is running.

The Java language [GM95b] is designed to resemble C++, but has built-in functions for interaction with a user through a graphical interface. These are called "applets" and consist of little graphical application objects that perform such operations as drawing a button or a box, providing a pull-down menu, and so forth. The language is touted as being robust and secure, although security is used here in the sense of protection against viruses and other forms of misbehavior by imported applications; the Java environment provides nothing new for securing the path from the browser to the server, or authenticating a user to the server.

Interestingly, Java has no functions or procedures, and no notion of data structures. The entire model is based on a very simple, pure object interface approach: programmers work *only* with object classes and there methods. The argument advanced by the developers of Java is that this "functional" model is adequate and simple, and by offering only one way to express a given task, the risk of programmer errors is reduced and the standardization of the resulting applications increased. Other "missing features" of Java include multiple inheritance (a problemantic aspect of C++), operator overloading (in Java, an operator means just what it seems to mean; in C++, an operator can mean almost anything at all), automatic coercions (again, a costly C++ feature that is made explicit and hence "controlled" in Java), pointers and goto statements. In summary, Java looks somewhat similar to C or C++, but is in fact an extremely simplified subset, really containing the absolute minimum mechanisms needed to program sophisticated display applications without getting into trouble or somehow contaminating the client workstation on which the downloaded applet will execute.

Java is a multithreaded language, offering many of the same benefits as are seen in RPC servers that uses threads. At the same time, however, Java is designed with features that protect against concurrency bugs. It supports dynamic memory allocation, but uses a memory management model that has no pointers or pointer arithmetic, eliminating one of the major sources of bugs for typical C and C++ programs. A background garbage collection facility quietly cleans up unreferenced memory, making memory leaks less likely than in C or C++ where memory can be "lost" while a program executes. The language provides extensive compile-time checking, and uses a second round of run-time checking to prevent Java applications from attempting to introduce viruses onto a host platform or otherwise misbehaving in ways that could crash the machine. The later can even protect against programs written to look like legitimate Java "object codes" but that were compiled using "hostile compilers."

Although "security" of a Java application means "safe against misbehavior by imported agents", Java was designed with the secure sockets layer of the Web in mind. The language doesn't add anything new here, but does include support for the available network security options such as firewalls and the security features of HTTP. Individual developers can extend these or use them to develop applications that are safe against intruders who might try and attack a server or steal data by snooping over a network. Java can thus claim to have closed the two major security holes in the Web: that of needing to important untrusted software onto a platform, and that of the connection to the remote server.

The actual role of a Java program is to build a display for the user, perhaps using data solicited from a Java server, and to interact with the user through potentially sophisticated control logic. Such an approach can drastically reduce the amount of data that a server must send to its clients. For example, a medical data server might need to send graphs, charts, images, and several other types of objects to the user interface. Using an HTML approach, the server would construct the necessary document upon request and send it to the display agent, which would then interactively solicit subdocuments needed to form the display. The graphical data would be shipped in the form of GIF, MPEG or JPEG images, and the entire document might require the transmission of megabytes of information from server to display agent.

By writing a Java program for this purpose, the same interaction could be dramatically optimized. Unlike conventional browsers, the HotJava browser is designed without any built-in knowledge of the protocols used to retrieve data over the Internet. Thus, where a standard browser is essentially an "expert" in displaying HTML documents retrieved using HTTP, HotJava understands both HTML and HTTP though classes of display and retrieval objects that implement code needed to deal with retrieving such documents from remote servers and displaying them to the user. Like any browser, HotJava includes built-in object classes for such standard Internet protocols and objects as HTTP, HTML, SMTP (the mail transfer protocol), URL's (making sense of Web addresses), GIF, NNTP (the news transfer protocol), FTP, and Gopher. However, the user can add new document types and new retrieval protocols to this list — in fact the user can even add new kinds of document addresses, if desired. At runtime, the HotJava browser will ask the appropriate class of object to resolve the address, fetch the object, and display it.

Sometimes, the browser may encounter an object type that it doesn't know how to display. For example, a downloaded Java applet may contain references to other Java objects with which the browser is unfamiliar. In this case, the browser will automatically request that the server transfer the Java display code needed to display the unknown object class. The benefit of this approach is that the server can potentially maintain a tremendously large database of object types -- in the limit, each object on the server can be a type of its own, or the server could actually construct a new object type for each request. Abstractly it would seem that the browser would needed unlimited storage capacity to maintain the methods needed to display such a huge variety of objects, but in practice, by downloading methods as they are needed, the actual use of memory in the browser is very limited. Moreover, this model potentially permits the server to revise or upgrade the display code for an object, perhaps to fix a bug or add a new feature. The next time that the browser downloads the method, the new functionality will immediately be available.

The developers of Java talk about the language as supporting "dynamic content" because new data types and the code needed to display them can be safely imported from a server, at runtime, without concern about security violations. One could even imagine a type of server that would construct a new Java display program in response to each incoming request, compile it on the fly, and in this way provide new *computed* object classes dynamically. Such an approach offers intriguing new choices in the endless tension for generality without excess code or loss of performance.

Indeed, Java programs can potentially use non-standard protocols to communicate with the server from which they retrieve data. Although this feature is somewhat limited by the need for the HotJava browser to maintain a trusted and secure environment, it still means that Java applications can break away from the very restricted HTTP protocol, implementing flexible protocols for talking to the server and hence providing functionality that would be hard to support directly over HTTP.

Returning to our medical example, these features make Java suitable for supporting specialized display programs that might be designed to compute medical graphics directly from the raw data. The display objects could also implement special-purpose data compression or decompression algorithms matched to the particular properties of medical image data. Moreover, the language can potentially support a much richer style of user interface than would otherwise be practical: if it makes sense to do so, a display object could permit its users to request that data be rescaled, that a graph be rotated, certain features be highlighted, and so forth, all in an application-specific manner and without soliciting additional data from the server. Whatever makes sense to the application developer can be coded by picking an appropriate document representation and designing an appropriate interactive display program in the form of an interpreted Java object or objects. This is in contrast to a more standard approach in which the browser has a very limited set of capabilities and any other behavior that one might desire must be implemented on the server.

Although Java was initiated lauded for its security features, it wasn't long before security concerns about Java surfaced. Many of these may have been corrected by the time this book goes to press in late 1996, but it may be useful to briefly touch upon some examples of these concerns simply to make the point that reliability doesn't come easily in distributed systems, and that the complex technologies (such as Hot Java) that are most promising in the long term can often reduce reliability in the early period soon after they are introduced. Security threats associated with downloaded agents and other downloaded software may well become a huge problem in the late 1990's, because on the one hand we see enormous enthusiasm for rapid adoption of these technologies in critical settings, and yet on the other hand, the associated security problems have yet to be fully qualified and are far from having been convincingly resolved.

The author is aware of at least two issues that arose early in the Java "life cycle". The first of these was associated with a feature by which compiled Java code could be downloaded from Java servers to the Hot Java browser. Although Java is intended primarily for interpretive use, this compilation technique is important for performance, and in the long term, it is likely that Java will be an increasingly compiled language. Nonetheless, the very early versions of the object code down-loading mechanism apparently had a bug that clever hackers could exploit to download malicious software that might cause damage to client file systems. This problem was apparently fixed soon after it appeared, and before any major use had been made of it by the community that develops viruses. Yet one can only wonder how many early versions of the Hot Java browser are still in use, and hence still exposed to this bug.

A second problem was reported in Spring of 1996, and was nicknamed the "Black Widow applet." Java Black Widow applets are hostile programs created to infect the systems of users who surf the Web, using Java as their technology (see http://www.cs.princeton.edu/sip/pub/secure96.html). These programs are designed to interfere with their host computers, primarily by consuming RAM and CPU cycles, so as to lower the performance available to the user. Some of these applets also make use of the compiled code problems of earlier Java servers, and make use of this ability to a third party on the Internet and, without the PC owner's knowledge, transfer information out of the user's computer by subverting the HTTP protocol. Even sophisticated firewalls can be penetrated because the attack is launched from within the Java applet, which operates behind the firewall. Many users are surprised to realize that there maybe untrustworthy Web sites that could launch an attack on a browser, and indeed many may not even be aware that they are using a browser that supports the Java technology and hence is at risk. Apparently, many of these problems will son be fixed in new versions of the browsers offered by major vendors, but again, one can only wonder how many older and hence flawed browsers will remain in the field, and for how long.

One can easily imagine a future in which such problems would lead the vendors to create private networks within which only trusted web sites are available, and to limit the ability of their browser technologies to download applications from untrusted sites. Without any doubt, such a world would appeal to the very large network operators, since the user's PC would effectively be controlled by the vendor if this were to occur: in effect, the user's system would be able to download information only from the network service provider's servers and those of its affiliates. Yet one must also wonder if the promise of the Web could really be achieved if it is ultimately controlled by some small number of large companies. For the Web to emerge as a thriving economic force and a major contributor to the future information-based economy, it may be that only a free-enterprise model similar to the current Internet will work. If this is so, we can only hope that the security and reliability concerns that threaten the Internet today will be overcome to a sufficient degree to enable wider and wider use of the technology by users who have no particular restrictions imposed upon their actions by their network provider.

As an example, there is much current research into what are called "sandbox" technologies, which consist of profiles that describe the expected behavior of an agent application and that can be enforced by the browser that downloads it. To the degree that the profile itself is obtained from a trustworthy source and cannot be compromised or modified while being downloaded (perhaps a risky assumption!), one could imagine browsers that product themselves against untrusted code by restricting the actions that the downloaded code can perform. The major vendors would then begin to play the role of certification authorities, providing (or "signing") profile information, which is perhaps a more limited and hence less intrusive activity for them than to completely "control" some form of virtual private network and to restrict their browsers to operate only within its confines.

10.9 GUI Builders and Other Distributed CASE Tools

Java is currently the best known of the Web agent languages, but in time it may actually not be the most widely successful. As this book was being written, companies known for their graphical database access tools were hard at work on converting these into Web agent languages. Thus, languages like Visual Basic (the most widely used GUI language on PC systems) and Power Builder (a GUI building environment supporting a number of programming languages) are likely to become available in Java-like forms, supporting the development of graphical display agents that can be sent by a server to the user's Web browser, with the same sorts of advantages offered by Java. Database products like Oracle's Power Objects may similarly migrate into network-enabled versions over a short period of time. By offering tight integration with database systems, these developments are likely to make close coupling of database servers with the Web much more common than it was during the first few years after the Web phenomenon began.

Moreover, outright execution of downloaded programs may become more common and less risky over times. Recall that Java was introduced primarily as a response to the risks of downloading and executing special purpose display programs for novel object types. If this sort of operation was less of a risk, there would be substantial advantages to a non-interpretive execution model. In particular, Java is unlikely to perform as well as compiled code, although it has a big advantage in being portable to a wide variety of architectures. Yet on relatively standard architectures, such as PC's, this advantage may not be all that important, and the performance issue could be critical to the success of the agent language.

Earlier in this text we discussed *object code editing* of the sort investigated by Lucco and Graham. These technologies, as we saw at the time, offer a way to contain the potential actions of a piece of untrusted software, permitting a platform to import a function or program and yet to limit its actions to a set of operations that are considered safe. Object code editing systems represent a viable alternative to the Java model: one could easily imagine using them to download compiled code, encapsulate it to eliminate the risk of ill effect, and then to directly execute it on the client's workstation or PC. Object code editors are potentially language independent: the program downloaded could be written in C, C++, assembler language, or Basic. Thus they have the benefit of not requiring the user to learn and work with a new programming language and model, as is the case for Java. It seems likely to this author that object code editors will emerge to play an increasingly important role in the world of agents in the future, particularly if signficant use of VRML applications begins to create a demanding performance problem on the client side.

10.10 Tacoma and the Agent Push Model

The agent languages described above, such as Java, are characterized by supporting a "pull" model of computation. That is, the client browser pulls the agent software from the server, and executes it locally. However, there are applications in which one would prefer the converse model: one in which the browser builds an agent which is then sent to the server to execute remotely. In particular, this would seem to be the case for applications in which the user needs to browse a very large database but only wishes to see a small number of selections that satisfy some property. In a conventional "pull" model such a user ultimately depends on the flexibility of the search interface and capabilities of the web server; to the degree that the server is limited, the user will need to retrieve entire documents and study them.

Consider a situation in which the user is expected to pay for these documents. For example, the server might be an image archive, and the user may be purchasing specific images for use in a publication. The owner's of the image archive won't want to release images for casual browsing, because the data is a valuable resource that they own. Each retrieved image may carry a hefty price and require some form of contractual agreement. In this case, short of somehow offering low-quality images to the browser and selling the high quality ones (a viable model only for certain classes of application), there seems to be a serious obstacle to a Web-based solution.

The TACOMA language, developed by researchers at the University of Tromso and at Cornell University, works to overcome these problems by offering an agent push model, in which the agent goes to the data, does some work, and may even migrate from server to server before ultimately returning results to the end-user [JvRS95a, JvRS95b, JvRS96, AJ95]. TACOMA was originally designed for use in the StormCast system [Joh94], a weather and environmental monitoring application about which we will hear more about in future chapters.

The basic TACOMA problem area is easily described. StormCast collects and archives huge amounts of weather and environmental data in the far north, storing this information at a number of archive servers. The goal is to be able to use this sort of data to construct special-purpose weather forecasts, such as might be used by local airports or fishing vessels.

Not suprisingly, the extreme weather conditions of the Arctic make general weather prediction difficult. To predict the weather in a specific place, such as the fiords near Tromso, one needs to combine local information about land topography and prevailing winds with remote information. If a storm is sweeping in from the north, data may be needed from a weather server off shore to the north; if the current prediction suggests that weather to the south is more important, the predictive software may need to extract data from an archive to the south. Moreover, predictions may need to draw on satellite data and periodically computed weather modelling information generated sporadically by supercomputing centers associated with the Norweigan Meteorology organization. Thus, in the most general case, a local weather prediction could combine information extracted from dozens of archives containing gigabytes of potentially relevant information. It is immediately clear that the push model supported by Java-like languages cannot address this requirement. One simply doesn't want to move the data to the browser in this case.

Using TACOMA, the user writes programs in any of a number of supported languages, such as C, C++ , Tcl or Perl. These programs are considered to "travel" from server to server carrying "briefcases" in which data is stored. Arriving at a server, a TACOMA agent will meet with an execution agent that unpacks the program itself and the data from the briefcase, compiles the program, and executes it. A variety of security mechanisms are employed to limit the ill effects of agents and to avoid the unintended proliferation of agents within the system. TACOMA itself implements the basic encapsulation mechanisms needed to implement briefcases (which are basically small movable file systems, containing data files organized within folders), and provides support for the basic "meet" primitive by which an agent moves from place to place. Additional servers provide facilities for longer term data storage, preprogrammed operations such as data retrieval from local databases, and navigation aids for moving within a collection of multiple servers.

Thus, to implement the weather prediction application described above, the application programmer would develop a set of agent programs. When a weather prediction is requested, these agents would be dispatched to seek out the relevant data, perhaps even doing computation directly at the remote data archive, and sending back only the information actually needed by the user. The results of the search would then be combined and integrated into a single display object on the user's workstation.

Similarly, to overcome the image retrieval problems we discussed, a TACOMA agent might be developed that understands the user's search criteria. The agent could then be sent to the image archive to search for the desired images, sending back a list of images and their apparent quality relative to the search criteria.

It can be seen that the push model of agents raises a number of hard problems that don't arise in a pull setting: management of the team of agents that are executing on behalf of a given user, termination of a computation when the user has seen adequate results, garbage collection of intermediate results, limiting the resources consumed by agents, and even navigating the network. Moreover, it is unlikely that end-users of a system will want to do any sort of programming, so TACOMA needs to be seen as a sort of agent middleware, used by an application programmer but hidden from the real user. Nonetheless, it is also clear that there are important classes of applications that the Java-style of pull agent will not be able to address, and which a push style of agent could potentially solve. It seems very likely that as the Web matures, both forms of agent language will be of increasing importance.

10.11 Web Search Engines and Web Crawlers

An important class of web servers are the *search engines*, which permit the user to retrieve URL's and short information summaries about documents of potential interest. Such engines typically have two components. A *web crawler* is a program that hunts for new information on the web and revisits sites for which information has been cached, revalidating the cache and refreshing information that has changed. Such programs normally maintain lists of web servers and URL's. By retrieving the associated documents, the web crawler extracts keywords and content information for use in resolving queries, and also obtains new URI's that can be exploited to find more documents.

A web search engine is a program that performs queries in a database of document descriptions maintained by a web crawler. Such search engines accept queries in various forms (written language, often english, is the most popular query language), and then use document selection algorithms that attempt to match the words used in the query against the contents of the documents. Considerable sophistication within the search engine is required to ensure that the results returned to the user will be sensible ones, and to guarantee rapid response. A consequence is that information retrieval, already an important research topic in computer science today, has become a key to the sucess of the Web.

Future web search programs are likely to offer customizable search criteria by which documents can be located and presented to the user based on ongoing interests. An investor might have an ongoing interest in documents that predict future earnings for companies represented in a stock portfolio, a physician in documents relating to his or her specialization, and a lover of fine wines in documents that review especially fine wines or in stores offering those wines at particularly good prices. Increasingly, web users will be offered these sorts of functionality's by the companies that today offer access to the internet and its email, chat and bulletin board services.

10.12 Important Web Servers

Although the Web can support a great variety of servers, business or "enterprise" use of the Web is likely to revolve around a small subset that support a highly standardized commercial environment. Current thinking is that these would consist of the following:

- *Basic Web servers.* These would maintain the documents and services around which the enterprise builds its information model and applications.
- *Web commerce or merchant servers.* These would play the role of an online bank or checkbook. Financial transactions that occur over the network (buying and selling of products and services) would occur through the electronic analog of issuing a quote, responding with a purchase order, delivery of the product or service, billing, and payment by check or money transfer. Data encryption technologies and secure electronic transfer for credit-card purchases will be crucial to ensuring that such commerce servers can be trusted and protected against third-party attack or other forms of intrusion.
- *Web exchange servers.* These are servers that integrate functionality associated with such subsystems as electronic mail, fax, chat groups and news groups, and multiplexing communication lines. Many such servers will also incorporate firewall technologies. Group scheduling, such as has been popularized by Lotus Notes, may also become a standard feature of such servers.
- *Web-oriented database servers.* As noted earlier, the early use of the Web has revolved around databases of HTML documents, but this is not likely to be the case in the long term. Over time, database servers will be increasingly integrated into the Web model. Indeed, it seems very likely that just as database and transactional systems are predominent in other client-server applications, they will ultimately dominate in Web environments.

10.13 Future Challenges

Although the explosive popularity of the Web makes it clear that the existing functionality of the system is more than adequate to support useful applications, evolution of the broader technology base will also require further research and development. Some of the major areas for future study include the following:

- *Improved GUI builders and browsers.* Although the first generation of browsers has already revolutionized the Web, the second wave of GUI builder technologies promises to open distributed computing to a vastly larger community. This development could revolutionize computing, increasing the use of networking by orders of magnitude and tremendously amplifying the existing trend towards critical dependency upon distributed computing systems.

  An interesting issue concerns the likely reliability impact of the widespread use of GUI builders, which might be chacterized as the CASE tools of distributed computing. On the positive side, GUI technologies encourage a tremendous degree of regularity in terms of application structure: every Java application resembles every other Java application in terms of the basic computing model and the basic communication model, although this model admits considerable customization. But on the negative side, the weak intrinsic reliability of many GUI execution models and the consistency issues cited earlier are likely to become that much more visible as hundreds of thousands of application developers suddenly become "internet enabled". If these sorts of problems turn out to have common, visible consequences, the societal impact over time could be considerable.

  Thus, we see a tradeoff here that may only become more clear with increased experience. The author's intuition is that wider use of the Web will create growing pressure to "do something" about network reliability and security, and that the latter topic is receiving much more serious attention than the former. This could make reliability, in the sense of availability, consistency, managability, timely and guaranteed responsiveness, emerge as one of the major issues of the coming decade. But it is also possible that 90% of the market will turn out to be disinterested in such issues, with the exception of their need to gain sufficient security to support electronic commerce, and that the vendors will simply focus on that 90% while largely overlooking the remaining 10%. This could lead us to a world of secure banking tools imbedded into inconsistent and unreliable application software.

The danger, of course, is that if we treat reliability issues casually, we may begin to see major events in which distributed systems unreliability has horrific consequences, in the sense of causing accidents, endangering health and privacy, bringing down banks, or other similarly frightening outcomes. Should terrorism ever become a major problem on the network, one could imagine scenarios in which that 10% exposure could suddenly loom as an immense problem: as discussed in the introduction, we already face considerable risk through the increasing dependence of our telecommunications systems, power systems, banking systems and air traffic control systems on the network, and this is undoubtedly just the beginning of a long term trend.

If there is light in this particular tunnel, it is that a number of functionality benefits turn out to arise as "side effects" of the most effective reliability technologies, discussed in Part III of this text. It is possible that the desire to build better groupware and conferencing systems, better electronic stock markets and trading systems, and better tools for mundane applications like document handling in large organizations will drive developers to look more seriously at the same technologies that also turn out to promote reliability. If this occurs, the knowledge base and tool base for integrating reliability solutions into GUI environments and elaborate agent programming languages could expand substantially, making it reliability both easier and more transparent to achieve, and more widely accessible.

- *Universal Resource Names.* Universal resource locators suffer from excessive specificity: they tell the browser precisely where a document can be found. At the same time, they often lack information that may be needed to determine which version of a document is desired. Future developments will soon result in the introduction of universal resource names capable of uniquely identifying a document regardless of where it may be cached, and including additional information to permit a user to validate its authenticity or to distinguish between versions. Such a universal resource name would facilitate increased use of caching within web servers and proxies other than the originating server where the original copy of the document resides. Important issues raised by this direction of research include the management of consistency in a world of replicated documents that may be extensively cached.

- *Security and Commerce Issues.* The basic security architecture of the Web is very limited and rather trusting of the network. As noted earlier, a number of standards have been proposed in the areas of web security and digital cash. These proposals remain difficult to evaluate and compare with one another, and considerable work will be needed before widely acceptable standards are available. These steps are needed, however, if the web is to become a serious setting for commerce and banking.

- *Availability of Critical Web Data.* Security is only of the reliability issues raised by the Web. Another important concern is availability of critical resources, such as medical documents that may be needed in order to treat patients in a hospital, banking records needed in secure financial applications, and decision support documents needed for split-second planning in settings such as battlefields. Current web architectures tend to include single points of failure, such as the web server responsible for the original copy of a document, and the authentication servers used to establish secure web connections. When these critical resources are inaccessible or down, critical uses of the web may be impossible. Thus, technologies permitting critical resources to be replicated for fault-tolerance and higher availability will be of growing importance as critical applications are shifted to the Web.

- *Consistency and the Web.* Mechanisms for caching web documents and replicating critical resources raise questions about the degree to which a user can trust a document to be a legitimate and current version of the document that was requested. With existing web architectures, the only way to validate a document is to connect to its home server and use the "head" command to confirm that it has not changed since it was created. Moreover, there is no guarantee that as a user retrieves a set of linked documents, they will not be simultaneously updated on the originating server. Such a situation could result in a juxtoposition of stale and current documents, yielding a confusing or inconsistent result. More broadly, we need to understand what it means to say that a document or set of documents are seen in mutually consistent states, and how this property can be guaranteed by the Web. Where documents are replicated or cached, the same requirement extends to the replicas. In Chapter 17 we will consider solutions to these sorts of problems, but their use in the Web remains tentative, and many issues will require further research, experimentation, and standardization.

10.14 Related Readings

On the Web: [BCLF94, BCLF95, BCGP92]. For Java, [GM95a, GM95b]. For Tacoma, [JvRS95a, JvRS95b, JvRS96, AJ95]. There is a large amount of online material concerning the Web, for example in the archives maintained by Netscape Corporation [http://www.netscape.com].

11. Related Internet Technologies

The Web is just the latest of a series of internet technologies to have gained extremely wide acceptance. In this chapter we briefly review some of the other important members of this technology family, including both old technologies such as mail and file transfer and new ones such as high speed message bus architectures and security firewalls. Details on many of the technologies discussed here can be found in [COM93].

11.1 File Transfer Tools

The earliest networking technologies were those supporting file transfer in distributed settings. These typically consist of programs for sending and receiving files, commands for initiating transfers and managing file transfer "queues" during periods when transfers back up, utilities for administering the storage areas within which files are placed while a transfer is pending, and policies for assigning appropriate ownership and access rights to transferred files.

The most common file transfer mechanism in modern computer systems is that associated with the FTP protocol, which defines a set of standard message formats and request types for navigating in a file system, searching directories, and moving files. FTP includes a security mechanism based on password authentication; however, these passwords are transmitted in an insecure way over the network, exposing them to potential attack by intruders. Many modern systems employ non-reusable passwords for this reason.

Other well known file transfer protocols include the UNIX-to-UNIX copy program (UUCP), and the file transfer protocol standardized by the ISO protocol suite. Neither protocol is widely used, however, and FTP is a defacto standard within the internet.

11.2 Electronic Mail

Electronic mail was the first of the internet applications to gain wide popularity, and remains a dominant technology at the time of this writing. Mail systems have become steadily easier to use and more sophisticated over time, and email users are supported by increasingly sophisticated mail reading and composition tools.

Underlying the email system are a small collection of very simple protocols, of which the Simple Mail Transfer Protocol, or SMTP, is most widely used and most standard. The architecture of a typical mailing system is as follows. The user composes a mail message, which is encoded into ascii (perhaps using a MIME representation) and then stored in a queue of outgoing email messages. Periodically, this queue is scanned by a mail daemon program, which uses SMTP to actually transmit the message to its destinations. For each destination, the mail daemon establishes a TCP connection, delivers a series of email messages, and receives acknowledgments of successful reception after the received mail is stored on the remote incoming mail queue. To determine the location of the daemon that will receive a particular piece of mail, the DNS for the destination host is queried.

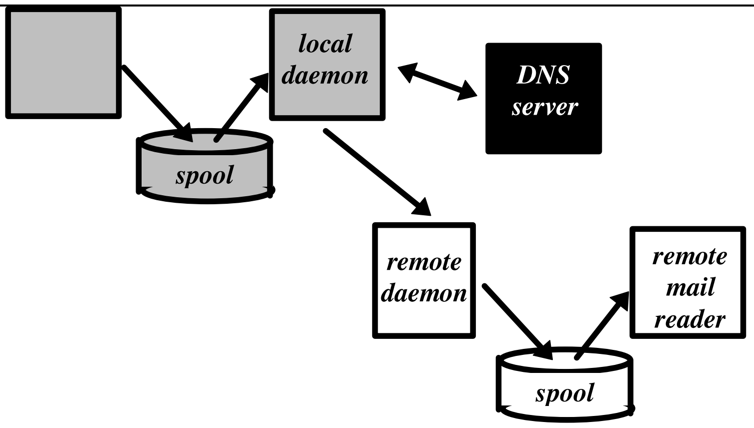


Figure 11-1: Steps in sending an email. The user composes the email and it is stored in a local spool for transmission by the local mail daemon. The daemon contacts the DNS to find the remote SMTP daemon's address for the destination machine, then transfers the mail file and, when an acknowledgment is received, deletes it locally. On the remote system, incoming mail is delivered to user mailboxes. The protocol is relatively reliable but can still lose mail if a machine crashes while transferring messages into or out of a spool, or under other unusual conditions such as when there are problems forwarding an email.

Users who depend upon network mail systems will be aware that the protocol is robust, but not absolutely reliable. Email may be lost after it has been successfully acknowledged, for example if the remote machine crashes just after receiving an email. Email may be delivered incorrectly, for example if the file system of a mail recipient is not accessible at the time the mail arrives and hence forwarding or routing instructions are not correctly applied. Further, there is the risk that email will be inappropriately deleted if a crash occurs as the user starts to retrieve it from a mailbox. Thus, although the technology of email is fairly robust, experienced users learn not to rely upon it in a critical way. To a limited degree, the "return receipt" mechanisms of modern mailers can overcome these difficulties, but heterogeneity prevents these from representing a completely satisfactory solution. Similarly, use of the more sophisticated email mechanisms, such as email with attached files, remains limited by incompatibilities between the mail reception processes that must interpret the incoming data.

11.3 Network Bulletin Boards (newsgroups)

Network bulletin boards evolved in parallel with the email system, and hence share many of the same properties. Bulletin boards differ primarily in the way that messages are viewed and the way that they are distributed.

As most readers will be aware, a bulletin board is typically presented to the user as a set of articles, which may be related to one-another in what are called "conversations" or "threads". These are represented by special fields in the message headers that identify each message uniquely and permit one message to refer to another. Messages are typically stored in some form of directory structure and the programs used to view them operate by displaying the contents of the directory and maintaining a simple database in which the messages each user has read are tracked.

The news distribution protocol, implemented by the NNTP daemon, is simple but highly effective. It is based on a notion of flooding. Each news message is posted to a news group or groups. Associated with each news group is a graph representing the connections between machines that wish to accept copies of postings to the group. To post a message, the user creates it and enqueues it in an outgoing news area, where the news daemon will eventually find it. The daemon for a given machine will periodically establish contact with some set of machines "adjacent" to it in the news distribution graph, exchanging messages that give the current set of available postings and their subjects. If a daemon connects to an adjacent daemon that has not yet received a copy of some posting, it forwards it over the link, and vice versa.

This protocol is fairly reliable, but not absolutely so. Similar to the case of email, an ill-timed crash can cause a machine to lose a copy of a recently received news posting after it has been confirmed, in which case there may be a gap in the news sequence for the corresponding group unless some other source happens to offer a copy of the same message on a different connection. Messages that are posted concurrently may be seen in different orders by different readers, and if a posting does not explicitly list *all* of the prior postings on which it is dependent, this can be visible to readers, because their display programs will not recognize that one message predates another. The display algorithms can also be fooled into displaying messages out of order by clock synchronization errors, which can erroneously indicate that one message is earlier than another that it actually follows.

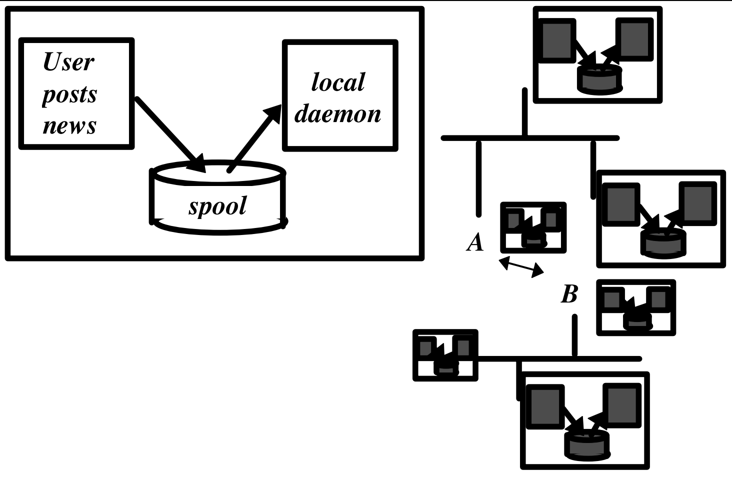


Figure 11-2: The network news protocol, NNTP, "floods" the network by gossip between machines that have news articles and machines that have yet to receive them. In this example, a message posted by a user reaches a forwarding node, A, which gossips with B by exchanging messages indicating the newsgroups for which each has recently received new postings. If B has not yet received the posting A just received, and is interested in the newsgroup, it will pull a copy from A. Failures of the network or of intermediate forwarding nodes can prevent articles from reaching their destinations quickly, in which case they will expire and may never reach some destinations. Thus, the protocol is quite reliable but now "always" reliable.

The news protocol is known to suffer from a variety of security problems. It is trivial to forge a message by simply constructing what appears to be a legitimate news message and placing it in the reception area used by the news daemons. Such messages will be forwarded even if they misrepresent the name of the sender, the originating machine, or other information. Indeed, scripts for *spamming* newsgroups have become popular: these permit an individual to post a single message to a great number of newsgroups. To a very limited degree, the news distribution protocol has improved with time to resist such attacks, but for a user with even a small degree of sophistication, the technology is open to abuse.

Thus, while news systems are generally reliable, it would be inappropriate to use them in critical settings. In any use for which the authenticity of messages is important, the context in which they were sent is significant, or the guarantee that postings will definitely reach their destinations is required, the technology is only able to provide partial solutions.

11.4 Message Oriented MiddleWare Systems (MOMS)

Most major distributed systems vendors offer products in what has become known as the "message oriented middleware" or MOMS market. Typical of these products are Digital Equiment's MessageQ product line, IBM's MQSeries products, and the so-called "asynchronous message agent technology" available in some object-oriented computing systems. For example, CORBA Event Notification Services are likely to be positioned as MOMS products.

Broadly, these products fall into two categories. One very important area is concerned with providing network access to mainframe systems. IBM's MQSeries product is focused on this problem, as are perhaps a dozen comparable products from a variety of vendors, although as noted below, MQSeries can also be useful in other settings. Technologies of this sort typically present the mainframe through a service interface abstraction that permits the distributed systems application developer to use a client-server architecture to develop their applications. These architectures are frequently asynchronous in the sense that the sending of a request to the mainframe system is decoupled from the handling of its reply, much as if one were sending mail to the mainframe server which will later send mail back containing the results of some inquiry. The message queueing system lives between the clients and the mainframe server, accepting the outgoing messages, transmitting them to the mainframe using the protocols appropriate for the mainframe operating system, arranging for the requests to be executed in a reliable manner (in some cases, even launching the associated application, if it is not already running), and then repeating the sequence in the opposite direction when the reply is sent back. Of course, these products are not confined to the mainframe connectivity problem, and many systems use them as front-ends to conventional servers running on network nodes or workstations. However, the mainframe connectivity issue seems to be driving force behind this market.

The second broad category of products uses a similar architecture but is intended more as a high-level message passing abstraction for direct use in networked applications. In these products, of which DEC's MessageQ is perhaps typical, the abstraction presented to the user is of named "mailboxes" to which messages can be sent by applications on the network, much as user's send email to one-another. Unlike email, the messages in question contain binary data, but the idea is very similar. Later, authorized applications dequeue the incoming messages for processing, sending back replies if desired, or simply consuming them silently. As one might expect, these products contain extensive support for such options as priority levels (so that urgent messages can skip ahead of less critical ones), flow control (so that message queues won't grow without limit if the consumer process or processes are slow), security, queue management, load-balancing (when several processes consume from the same queues), data persistance and fault-tolerance (for long-running applications), etc.

If the model of this second category of message queuing products is that of an email system used at a program-to-program level, the performance is perhaps closer to that of a special-purpose file system. Indeed, many of these systems work very much as a file system would work: adding a message to a queue is done by appending the message to a file representing the queue, and dequeueing a message is done by reading from the front of the file and freeing the corresponding disk space for reuse.

The growing popularity of message-oriented middleware products is typically due to their relative ease of use when compared to datagram style message communication. Applications that communicate using RPC or datagrams need to have the producer and consumer processes running at the same time, and must engage in a potentially complex binding protocol whereby the consumer or server process registers itself and the producer or client process locates the server and establishes a connection to it. Communication is, however, very rapid once this connection establishment phase has been completed. In contrast, a message-oriented middleware system does not require that the producer and consumer both be running at the same time, or even that they be knowledgeable of one-another: a producer may not be able to predict the process that will dequeue and execute its request, and a consumer process may be developed long before it is known what the various producers of messages it consumes will be. The downside of the model is that these products can be very slow in comparison to direct point-to-point communication over the network (perhaps by a factor of hundreds!), and that they can be hard to manage, because of the risk that a queue will leak messages and grow very large or that other subtle scheduling effects will cause the system to become overloaded and to thrash.

There is a good online source of additional information on middleware products, developed by the "Message Oriented Middleware Association", or MOMA:
http://www.sbexpos.com/sbexpos/associations/moma/home.html
Information on specific products should be obtained from the corresponding vendors.

11.5 Message Bus Architectures

Starting with the V operating system in 1985 [CZ85] and the Isis "news" application in 1987 [BJ87a], a number of distributed systems have offered a bulletin board style of communication directly to application programs; MIT' Zephyr system followed soon after [DEFJ88]. In this technology, which is variously called "message bus" communication, "message queues", "subject-based addressing", or "process group" addressing, processes register interest in message subjects by *subscribing* to them. The same or other processes can then send out messages by *publishing* them under one or more subjects. The message bus system is responsible for matching publisher to subscriber in a way that is efficient and transparent to both: the publisher is typically unaware of the current set of subscribers (unless it wishes to know) and the subscriber is typically not aware of the current set of publishers (again, unless it has some reason to ask for this information).

Message bus architectures are in most respects very similar to network bulletin boards. An application can subscribe to many subjects, just as a user of a bulletin board system can monitor many bulletin board topics. Both systems typically support some form of hierarchical name space for subjects, and both typically allow one message to be sent to multiple subjects. The only significant difference is that message bus protocols tend to be optimized for high speed, using broadcast hardware if possible, and typically deliver messages as soon as they reach their destination, through some form of *upcall* to the application process. In contrast, a bulletin board system usually requires a polling interface, in which the reader checks for new news and is not directly notified at the instant a message arrives.

The usual example of a setting in which a message bus system might be used is that of a financial trading application or stock exchange. In such systems, the subjects to which the messages are sent are typically the names of the financial instruments being traded: /equities/ibm, or /bonds/at&t. Depending on the nature of the message bus, reliability guarantees may be non-existent, weak, or very strong. The V system's process group technology illustrates the first approach: an application subscribed by joining a process group and published by sending to it, with the group name corresponding to the "subject". V multicasts to process groups lacked any sort of strong reliability guarantees, hence such an approach usually will deliver messages but not always. V transmitted these messages using hardware multicast features of the underlying transport technology, or point-to-point transport if hardware was not available.

The Teknekron Information Bus (TIB) [OPSS93] and Isis Message Distribution System (MDS) [Gla96] are good examples of modern technologies that support this model with stronger reliability properties. TIB is extremely popular in support of trading floors, and has had some success in factory automation environments. Isis is used in similar settings but where reliability is an especially important attribute: stock exchange systems, critical online control software in factories, and telecommunications applications.

Teknekron's TIB architecture is relatively simple, but is still sufficient to provide high performance, scalability, and some degree of fault-tolerance when publishers fail. In this system, messages are typically transmitted by the publisher using hardware broadcast, with an overlaid retransmission mechanism that ensures that messages will be delivered reliably and in the order they were published provided that the publisher doesn't fail. Point-to-point communication is used if a subject has only a small number of subscribers. Much as for a stream protocol, overload or transient communication problems can cause exceptional conditions in which messages would be lost, but such conditions are uncommon in the settings where TIB is normally used.

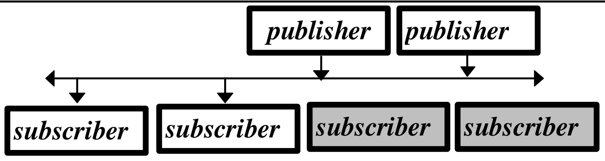


Figure 11-3: Message-bus architectures (also known as publish/subscribe systems) originated as an application of process groups in the V system, and a fault-tolerant version was included in early versions of the Isis Toolkit. These technologies subsequently became commercially popular, particularly in financial and factory-floor settings. A benefit of the architecture is that the subscriber doesn't need to know who will publish on a given subject, or vice-versa: here, messages on the "white" subject will reach the "white" but not the "gray" subscribers, a set that can change dynamically and that may grow to include applications not planned at the time the publishers were developed. A single process can publish upon or subscribe to multiple subjects. This introduces desirable flexibility in a technology that can also achieve extremely high performance by using hardware multicast or broadcast, as well as fault-tolerance or consistency guarantees if the technology base is implemented with reliability as a goal An object-oriented encapsulation of message bus technology is provided as part of CORBA, through its "Event Notification Service" or ENS. In this example the application is divided into publishers and subscribers, but in practice a single process can play both roles, and can subscribe to many "subjects". Wide-area extensions normally mimic the gossip scheme used in support of network bulletin boards.

The TIB system provides a failover capability if there are two or more equivalent publishers for a given class of subjects. In such a configuration, subscribers are initially connected to a primary publisher; the backup goes through the motions of publishing data but TIB in fact inhibits the transmission of any data. However, if the primary publisher fails, the TIB system will eventually detect this. Having done so, the system will automatically reconfigure so that the subscriber will start to receive messages from the other source. In this manner, TIB is able to guarantee that messages will normally be delivered in the order they are sent, and will normally not have gaps or out of sequence delivery. These properties can, however, be violated if the network becomes severely overloaded, a failure occurs on the publisher site, or the subscriber becomes temporarily partitioned away from the network. In these cases a gap in the sequence of delivered messages can occur during the period of time required for the "failover" to the operational server

The Isis Message Distribution System (MDS) is an example of a message bus that provides very strong reliability properties. This system is implemented using a technology based on reliable process groups (discussed in Chapters 13-18), in which agreement protocols are used to ensure that messages will be delivered to all processes that are subscribing to a subject, or to none. The approach also permits the "active replication" of publishers, so that a backup can provide precisely the same sequence of messages as the primary. By carefully coordinating the handling of failures, MDS is able to ensure that even if a failure does occur, the sequence of messages will not be disrupted: all subscribers that remain connected to the system will see the same messages, in the same order, even if a publisher fails, and this order will not omit any messages if the backup is publishing the same data as the primary [Gla96].

MDS is implemented over hardware multicast, but uses this feature only for groups with large fanout; point-to-point communication is employed for data transport when the number of subscribers to a subject is small, or when the subscribers are not on the same branch of a local area network. The resulting architecture achieves performance comparable to that of TIB in normal cases, but under overload, when TIB can potentially lose messages, Isis MDS will typically choke back the publishers and, if the load becomes extreme, may actually drop slow receivers from the system as a way to catch up. These different design choices are both considered to represent "reliable" behavior by the vendors of the two products; clearly, the real issue for any given application will be the degree of match between the reliability model and the needs of the applications that consume the data. MDS would be preferable in a system where ordering and guaranteed delivery are very important to the end-user; TIB might be favored in a setting where continued flow of information is ultimately valued more than the ordering and reliability properties of the system.

The reader may recall that the CORBA Event Notification Service (ENS) uses a message-bus architecture. TIB is in fact notable for supporting an object-oriented interface similar to the one required for implementations of this service. The Isis MDS has been integrated into the Orbix+Isis product, and hence can be used as an object-oriented CORBA ENS through Orbix [O+I95].

Both systems also provide a form of message spooling and playback facility. In TIB, this takes the form of a subscriber that spools all messages on specified subjects to disk, replaying them later upon request. MDS also includes a spooling technology that can store messages for future replay to a process. The MDS implementation of this playback technology preserves the ordering and reliability attributes of the basic publication mechanism, and is carefully synchronized with the delivery of new messages so that a subscriber can obtain a "seamless" playback of spooled messages followed by the immediate delivery of new messages, in a correct order and without gaps or duplication.

Both the TIB and MDS architectures can be extended to large-scale environments using a protocol much like the one described in the previous section for network bulletin boards.

11.6 Internet Firewalls and Gateways

Internet firewalls have recently emerged to play a nearly ubiquitous role in internet settings. We discuss firewalls in more detail in Chapter 19, and consequently limit ourselves to some very brief comments here.

A firewall is a message filtering system that resides at the perimeter of a distributed system, where messages enter and leave it from the broader internet. The specific architecture used may be that of a true packet filter, or one that permits application-level code to examine the incoming and outgoing packets (so-called "application-level proxy" technology). Although firewalls are not normally considered to be distributed programs, if a network has multiple access points, the firewall will be instantiated separately at each. Considered as a set, the collection of firewall programs will then be a distributed system, although the distributed aspect may be implicit.

Firewall programs typically operate by examining each message on the basis of source, destination, and authentication information (if enabled). Messages are permitted to pass through only if they satisfy some criteria controlled by the system administrator or, in the case of an application-level proxy technology, if the application programs that compose the firewall consider the message to be acceptable. In this manner, a network can be made porous for network bulletin board and email messages, but opaque to incoming FTP and remote login attempts, can be made to accept only packets digitally signed by an acceptable user or originating machine, etc.

A *gateway* is a program placed outside of a protected domain that offers users a limited set of options for accessing information protected within the domain. A gateway may permit selected users to establish login sessions with the protected domain, for example, but only after challenging the user to provide a one-time password or some other form of authentication. Gateways often provide mechanisms for file transfer as well, again requiring authentication before transfers are permitted.

For typical internet sites, the combination of firewalls and gateways provides the only real security. Although passwords are still employed within the firewall, such systems may be open to other forms of attack if an intruder manages to breach the firewall or to gain access permission from the gateway. Nonetheless, these represent relatively strong barriers to intrusion. Whereas systems that lack gateways and firewalls report frequent penetrations by hackers and other agents arriving over the network, firewalls and gateways considerably raise the barrier to such attacks. They do not create a completely secure environment, but they do offer an inexpensive way to repel all but the most sophisticated attackers.

11.7 Related Readings

Most internet technologies are documented through the so-called *Request for Comments* (RFC) reports, which are archived at various sites within the network, notably on a server maintained by SRI Corporation. The message bus technologies cited earlier were [CZ85, BJ87a, DEFJ88, OPSS93, Gla96].

Part III: Reliable Distributed Computing

In this third and final part of the textbook, we ask how distributed computing systems can be made reliable, motivated by our review of servers used in Web settings, but seeking to generalize beyond these specific cases to include future servers that may be introduced by developers of new classes of critical distributed computing applications. Our focus is on communications technologies, but we do review persistent storage technologies based on the transactional computing model, particularly as it has been generalized to apply to objects in distributed environments.

12. How and Why Computer Systems Fail

Throughout the remainder of this part of the text, we will be concerned with technologies for making real distributed systems reliable [BR96]. Before undertaking this task, it will be useful to briefly understand the reasons that distributed systems fail. Although there are some dramatic studies of the consequences of failures (see, for example, [Pet95]), our treatment draws primarily from work by Jim Gray [GBH87, Gra90, GR93], who studied this issue while at Tandem Computers, and on presentations by Anita Borr [BW92], who was a developer of Tandem's transactional system architecture [Bar81], and Ram Chilaragee, who has studied the same question at IBM [Chill92]. All three researchers focused on systems designed to be as robust as possible and might have drawn different conclusions had they looked at large distributed systems that incorporate technologies built with less stringent reliability standards. Unfortunately, there seems to have been relatively little formal study of failure rates and causes in systems that were *not* engineered with reliability as a primary goal, despite the fact that a great number of systems used in critical settings include components with this property.

12.1 Hardware Reliability and Trends

Hardware failures were a dominant consideration in architecting reliable systems until late in the 1980's. Hardware can fail in many ways, but as electronic packaging has improved and the density of integrated circuits increased, hardware reliability has grown enormously. This improved reliability reflects the decreased heat production and power consumption of smaller circuits, the reduction in the number of off-chip connections and wiring, and improved manufacturing techniques. A consequence is that hardware-related system downtime is fast becoming a minor component of the overall reliability concerns faced in a large, complex distributed system.

To the degree that hardware failures remain a significant reliability concern today, the observed problems are most often associated with the intrinsic limitations of connectors and mechanical devices. Thus, computer network problems (manifest through message loss or partitioning failures, where a component of the system becomes disconnected from some other component) are high on the list of hardware-related causes of failure for any modern system. Disk failures are also a leading cause of downtime in systems dependent upon large file or database servers, although RAID-style disk arrays can protect against such problems to a limited degree.

A common hardware-related source of downtime has very little to do with failures, although it can seriously impact system availability and perceived reliability. Any critical computing system will, over its lifecycle, live through a series of hardware generations. These can force upgrades, because it may become costly and impractical to maintain old generations of hardware. Thus, routine maintenance and downtime for replacement of computing and storage components with more modern version must be viewed as a planned activity that can emerge as one of the more serious sources of system unavailability if not dealt with through a software architecture that can accommodate dynamic reconfiguration of critical parts of the system while the remainder of the system remains online. This issue of planning for future upgrading, expansion, and for new versions of components extends throughout a complex system, encompassing all its hardware and software technologies.

12.2 Software Reliability and Trends

Early in this text, we observed that software reliability is best understood as a process, encompassing not just the freedom of a system from software bugs, but also such issues as the software design methodology, the testing and lifecycle quality assurance process used, the quality of self-checking mechanisms and of user-interfaces, the degree to which the system implements the intended application (i.e. the quality of match between system specification and problem specification), and the mechanisms provided for dealing with anticipated failures, maintenance, and upgrades. This represents a rich, multidimensional collection of issues and few critical systems deal with them as effectively as one might wish. Software developers, in particular, often view software reliability in simplified terms, focusing exclusively on the software specification that their code must implement, and on its correctness with regard to that specification.

Even this narrower issue of correctness remains an important challenge; indeed, many studies of system downtime in critical applications have demonstrated that even after rigorous testing, software bugs account for a substantial fraction of unplanned downtime (figures in the range of 25% to 35% are common), and that this number is extremely hard to reduce (see, for example, Ivars Peterson's recent book *Fatal Defect* [Pet95]). Jim Gray and Bruce Lindsey, who have studied reliability issues in transactional settings, once suggested that the residual software bugs in mature systems can be classified into two categories, which they called *Bohrbugs* and *Heisenbugs* [GBH87, GR93].

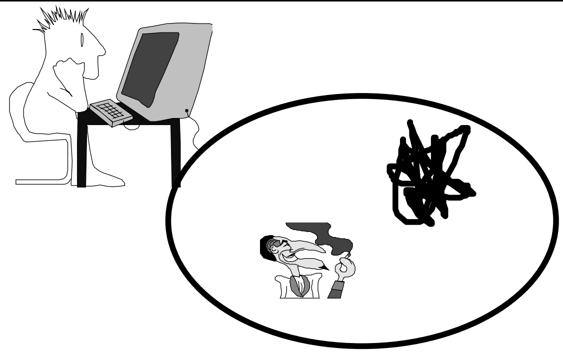


Figure 12-1: Developers are likely to discover and fix Bohrbugs, which are easily localized and reproducible sources of errors. Heisenbugs are fuzzy and hard to pin down. Often, these bugs are actually symptoms of some other bug which doesn't cause an immediate crash; the developer will tend to work around them but may find them extremely hard to fix in a convincing way. The frequency of such bugs diminishes very slowly over the life cycle of an application.

A Bohrbug is a solid, reproducible problem: if it occurs, and one takes note of the circumstances, the scenario can be reproduced and the bug will repeat itself. The name is intended to remind us of Bohr's model of the atomic nucleus: a small hard object, well localized in space. Gray and Lindsey found that as systems mature, the relative frequency of Bohrbugs drops steadily over time, although other studies (notably by Anita Borr) suggest that the population of Bohrbugs is periodically replenished when a system must be upgraded or maintained over its lifecycle.

Heisenbugs are named for the Heisenberg model of the nucleus: a complex wave function that is influenced by the act of observation. These bugs are typically side-effects of problems that occurred much earlier in an execution, such as overrunning an array or accidentally dereferencing a pointer after the object to which it points has been freed. Such errors can corrupt the application in a way that will cause it to crash, but not until the corrupted data structure is finally referenced, which may not occur until long after the bug actually was exercised. Because such a bug is typically a symptom of the underlying problem, rather than an instance of the true problem itself, Heisenbugs are exquisitely sensitive to the order of execution. Even with identical inputs a program that crashed once may run correctly back in the laboratory.

Not surprisingly, the major source of crashes in a mature software system turns out to be Heisenbugs. Anita Borr's work actually goes further, finding that most attempts to fix Heisenbugs actually make the situation worse than it was in the first place. This observation is not surprising to engineers of complex, large software systems: Heisenbugs correspond to problems that can be tremendously hard to track down, and are often fixed by patching around them at runtime. Nowhere is the gap between theory and practice in reliable computing more apparent than in the final testing and bug correction stages of a major software deployment that must occur under time pressure or a deadline.

12.3 Other Sources of Downtime

Jointly, hardware and software downtime, including downtime for upgrades, is typically said to account for some two-thirds of system downtime in critical applications. The remaining third of downtime is attributable to planned maintenance such as making backups, and environmental factors, such as power outages, air conditioning or heating failures, leaking pipes, and other similar problems. (The author is reminded of an early job in which he was asked to track down a bug that was causing a critical computing system to crash at a major New York City hospital, always early in the morning or late in the day. Users of the system were convinced that the problem was load-related and yet the developers had failed to reproduce any errors under the most extreme forms of load-based stress test. The problem finally turned out to be caused by power fluctuations associated with the underground subway system during rush hour, which accidentally coincided with periods of heavy use at the beginning and end of the working day!)

Although there may be little hope of controlling these forms of downtime, the trend is to try and treat them using software techniques that distribute critical functionality over sufficient numbers of computers, and separate them to a sufficient degree, so that redundancy can overcome unplanned outages. Having developed software capable of solving such problems, downtime for hardware maintenance, backups, or other routine purposes can often be treated in the same way as are other forms of outages. Such an approach tends to view system management, monitoring and online control as a part of the system itself: a critical system should, in effect, be capable of modeling its own configuration and triggering appropriate actions if critical functionality is compromised for any reason. In the chapters that follow, this will motivate us to look at issues associated with having a system monitor its own membership (the set of processes that compose it), and dynamically, adapting itself in a coordinated, consistent manner if changes are sensed. Although the need for brevity will prevent us from treating system management issues in the degree of detail that the problem deserves, we will develop the infrastructure on which reliable management technologies can be implemented, and will briefly survey some recent work specifically on the management problem.

12.4 Complexity

Many developers would argue that the single most serious threat to distributed systems reliability is the *complexity* of many large distributed systems. Indeed, distributed systems used in critical applications often interconnect huge numbers of components using subtle protocols, and the resulting architecture may be extremely complex. The good news, however, is that when such systems are designed for reliability, the techniques used to make them more reliable may also tend to counteract this complexity.

For example, in the chapters that follow we will be looking at replication techniques that permit critical system data and services to be duplicated as a way to increase reliability. When this is done correctly, the replicas will be consistent with one another and the system as a whole can be thought of as containing just a single instance of the replicated object, but one that happens to be more reliable or more secure than any single object normally would be. If the object is active (a program), it can be *actively replicated* by duplicating the inputs to it and consolidating the outputs it produces. These techniques lead to a proliferation of components but also impose considerable regularity upon the set of components. They thus control the complexity associated with the robustness intervention.

Going forward, we will be looking at system management tools that monitor sets of related components, treating them as groups within which a common management, monitoring or control policy can be applied. Again, by factoring out something that is true for all system components in a certain class or set of classes, these techniques reduce complexity. What were previously a set of apparently independent objects are now explicitly seen to be related objects that can be treated in similar ways, at least for purposes of management, monitoring or control.

Broadly, then, we will see that although complexity is a serious threat to reliability, complexity can potentially be controlled by capturing and exploiting regularities in distributed system structure — regularities that are common when such systems are designed to be managed, fault-tolerant, secure, or otherwise reliable. To the degree that this is done, the system structure becomes more explicit and hence complexity is reduced. In some ways, the effort of building the system will increase: this structure needs to be specified, and needs to remain accurate as the system subsequently evolves. But in other ways, the effort is decreased: by managing a set of components in a uniform way, one avoids the need to do so in an ad-hoc basis that may be similar for the members of the set but not identical merely as an artifact of having developed the component management policies independently.

These observations are a strong motivation for looking at technologies that can support grouping of components in various ways and for varied purposes. However, they also point to a secondary consideration: unless such technologies are well integrated with system development software tools, they will prove to be irritating and hard to maintain as a system is extended over time. As we will see, researchers have been more active on the former problem than on the latter one, but this situation has now begun to change, particularly with the introduction of CORBA-based reliability solutions that are well integrated with CORBA development tools.

12.5 Detecting failures

Surprisingly little work has been done on the problem of building failure detection subsystems. A consequence is that many distributed systems detect failures using timeouts, an error-prone approach that forces the application to overcome inaccurate failure detections in software.

Recent work by Werner Vogels [Vog96] suggests that many distributed systems may be able to do quite a bit better. Vogels makes the analogy between detecting a failure and discovering that one's tenant has disappeared. If a landlord were trying to contact a tenant whose rent check is late, it would be a little extreme to contact the police after trying to telephone that tenant once, at an arbitrary time during the day, and not receiving any reply. More likely, the landlord would telephone several times, inquire of neighbors, check to see if the mail is still being collected and if electricity and water is being consumed, and otherwise check for indirect evidence of the presense or absense of the tenant.

Modern distributed systems offer a great number of facilities that are analogous to these physical options. The management information base of a typical computing node (it's MIB) provides information on the active processes and their consumption of resources such as memory, computing time, and I/O operations. Often, the network itself is instrumented, and indeed it may sometimes be possible to detect a network partition in an accurate way by querying MIB's associated with network interface and routing nodes. If the operating system on which the application in question is running is accessible, one can sometimes ask it about the status of the processes it is supporting. And, in applications designed with fault-tolerance in mind, there may be the option of integrating self-checking mechanisms directly into the code, so that the application will periodically verify that it is healthy and take some action, such as resetting a counter, each time the check succeeds. Through such a collection of tactics, one can potentially detect "most" failures rapidly and accurately, and even distinguish partitioning failures from other failures such as crashes or application termination. Vogels has implemented a prototype of a failure investigator service that uses these techniques, yeilding much faster and better failure detection than is traditionally assumed possible in distributed systems. Unfortunately, though, this approach is not at all standard. Many distributed systems rely entirely on timeouts for failures; as one might expect, this results in a high rate of erroneous detections and a great deal of complexity in order to overcome their consequences.

12.6 Hostile Environments

The discussion of this chapter has enumerated a great variety of reliability threats that a typical distributed system may need to anticipate and deal with. The problems considered, however, were all of a nature that might be considered "routine", in the sense that they all fall into the category of building software and hardware to be robust against anticipated classes of accidental failures, and to be self-managed in ways that anticipate system upgrades and maintenance events.

Yet, it is sometimes surprising to realize that the Internet is a hostile environment, and growing more so. Modern computer networks are shared with a huge population of computer literate users, whose goals and sense of personal ethics may differ tremendously from those of the system developer. Whether intentionally or otherwise, these network users represent a diffuse threat, who may unexpectedly probe a distributed system for weaknesses, or even to subject it to a well planned and orchestrated assault without prior warning.

The intentional threat spectrum is as varied as the accidental threat spectrum reviewed earlier. The most widely known of the threats are computer viruses, which are software programs designed to copy themselves from machine to machine, and to do damage to the machines on which they manage to establish themselves. (A benign type of virus that does no damage is called a *worm,* but because the mere presence of an unanticipated program can impact system reliability, it is perhaps best to take the view that all undesired intrusions into a system represent a threat to reliable behavior). A virus may attack a system by violating assumptions it makes about the environment or the network, breaking through security codes and passwords, piggybacking a ride on legitimate messages, or any of a number of other routes. Attacks that exploit several routes at the same time are more and more common, for example simultaneously compromising some aspect of the telecommunications infrastructure on which an application depends while also presenting the application with an exceptional condition that it can only handle correctly when the telecommunications subsystem is also functioning.

Other types of intentional threats include unauthorized users or authorized users who exceed their normal limitations. In a banking system, one worries about a rogue trader or an employee who seeks to divert funds without detection. A disgruntled employee may seek to damage the critical systems or data of an organization that is perceived as having wronged him or her. In the most extreme case, one can imagine hostile actions directed at a nation's critical computing systems during a period of war, or terrorism. Today, this sort of *information warfare* may seem like a suitable topic for science fiction writers, yet as society shifts increasingly critical activities onto computing and communications technology, the potential targets for attack will eventually become rich enough to interest military adversaries.

Clearly, no computing system can be protected against every conceivable form of internal and external threat. Distributed computing can, however, offer considerable benefits against a well known and fully characterized threat profile. By distributing critical functionality over sets of processes that must cooperate and coordinate their actions in order to perform sensitive functions, the barrier against external threats can be raised very high. A terrorist who might easily overcome a system that effectively lacks any defenses at all would face a much harder problem overcoming firewalls, breaking through security boundaries, and interfering with critical subsystems designed to continue operating correctly even if some limited number of system components crash or are compromised. Later, we will discuss virtual private network technologies that take such approaches even further, preventing all communication within the network except that initiated by authenticated users. Clearly, if a system uses a technology such as this, it will be relatively hard to break into. However, the cost of such a solution may be higher than most installations can afford.

As the developer of a critical system, the challenge is to anticipate the threats that it must overcome, and to do so in a manner that balances costs against benefits. Often, the threat profile that a component subsystem may face will be localized to that component, hence the developer may need to go to great lengths in protecting some especially critical subsystems against reliability and security threats, while using much more limited and less costly technologies elsewhere in the same system. Our obligation in this textbook involves a corresponding issue: that of understanding not just how a reliability problem can be solved, but also how the solution can be applied in a selective and localized manner, so that a developer who faces a specific problem in a specific context can draw on a solution tailored to that problem and context, without requiring that the entire system be reengineered to overcome a narrow threat.

Today, we lack a technology with these attributes. Most fault-tolerance and security technologies demand that the developer adopt a fault-tolerant or secure computing and communications architecture from the first lines of code entered into the system. With such an approach, fault-tolerance and security become very hard to address late in the game, when substantial amounts of technology already exist. Unfortunately, however, most critical systems are built up out of preexisting technology, which will necessarily have been adapted to the new use and hence will necessarily be confronted with new types of reliability and security threats that were not anticipated in the original setting. Needed, then, is a technology base that is flexible enough to teach us how to overcome a great variety of possible threats, but that is also flexible enough to be used in a narrow and selective manner (so that the costs of reliability are localized to the component being made reliable), efficient (so that these costs are as low as possible), and suitable for being introduced *late in the game,* when a system may already include substantial amounts of preexisting technology.

The good news, however, is that current research is making major strides in the desired direction. In the chapters that follow, we will be looking at many of the fundamental challenges that arise in overcoming various classes of threats. We will discuss computing models that are dynamic, self-managed, and fault-tolerant, and will see how a technology based on *wrapping* preexisting interfaces and components with look-alike technologies that introduce desired robustness features can be used to harden complex, pre-existing systems, albeit with many limitations. Finally, we will consider some of the large-scale systems issues raised when a complex system must be managed and controlled in a distributed setting. While it would an overstatement to claim that all the issues have been solved, it is clear that considerable progress towards an integrated technology base for hardening critical systems is being made.

This author has few illusions about reliability: critical computing systems will continue to be less reliable than they should be until the customers and societal users of such systems demand reliability, and the developers begin to routinely concern themselves with understanding the threats to reliability in a given setting, and planning a strategy for responding to those threats and for testing the response. However, there is reason to believe that in those cases where this process does occur, a technology base capable of rising to the occasion can be provided.

12.7 Related Readings

On dramatic system failures and their consequences: [Gib94, Pet95]. How and why systems fail and what can be done about it: [GBH87, Gra90, GR93, BW92, Chill92, BR96]. On the failure investigator: [Vog96]. On understanding failures [Cri96].

13. Guaranteeing Behavior in Distributed Systems

13.1 Consistent Distributed Behavior

It is intuitively natural to expect that a distributed system should be able to mimic the behavior of a non-distributed one. Such a system would benefit from its distributed architecture to gain better performance, fault-tolerance, or other advantages, and yet would implement a specification that may originally have been conceived with no particular attention to issues of distribution [BR96]. In effect, the specification of this type of distributed system talks about the desired behavior of ''the system'' as if the system were a single entity.

Because system designers rarely think in terms of concurrent behaviors of a system, describing a system as if it were actually not distributed and only worrying about its distributed nature later is in fact a very natural way to approach distributed software development. A developer who adopts this approach would probably say that the "intended interpretation" of the design methodology is that the final system should behave *consistently with the specification*. Pressed to explain this more rigorously, such a developer would probably say that the behavior of a correct distributed implementation of the specification should be indistinguishable from a possible non-distributed behavior of a non-distributed program conforming with the specification. Earlier, we saw that transactional systems use such an approach when they require serializability, and our hypothetical developer might well cite that example as an illustration of how this concept might be put into practice. In this part of the book, we will explore ramifications of such an approach, namely the requirement that there be a way to "explain" distributed behaviors of the system in terms of simpler, non-distributed, system specifications.

One can ask a number of questions about distributed consistency. The purpose of this first chapter is to set down some of these questions and to start exploring at least some of the answers. Other answers, and in some cases, refined questions, will appear in subsequent chapters of the text. The broad theme of this chapter, however, starts with a recognition that the concept of reliability built into the standard technologies we have discussed until now is at best a very ad-hoc one. At worst, it may be fundamentally meaningless. To do better, we need to start by associating meaning with at least some intuitively attractive distributed reliability goal.

For example, we asked if a reliable stream was ultimately more reliable than an RPC protocol. And, in some deep sense, the answer seems to be a negative one: a stream built using timeouts for failure detection is ultimately no more reliable than a protocol like RPC, which retries each request few times before timing out. In both cases, data that has been acknowledged is known to have reached its destination: the RPC provides such acknowledgments in the form of a reply, while the stream does so implicitly by accepting data beyond the capacity of its window (if a stream has accepted more data than its window limitation, the sender can deduce that the initial portion of the data must have been acknowledged). But the status of an RPC in progress, or of data still in the window of a stream, is uncertain in very similar senses, and if an error is reported the sender will not have any way to know what happened to that data. We saw earlier that such an error may occur even if neither sender nor destination fails. Yet streams are often considered a "reliable" way of transferring data. Obviously, they are more reliable than datagrams most of the time, but it would be an overstatement to claim that a stream is reliable in some absolute sense.

To say stronger things about a distributed system or protocol, we need to understand the conditions under which consistent behavior is achievable, and the senses in which consistency is achievable. This will turn out to be a deep, but solvable, problem. The solution reveals a path to a suprisingly rich class of distributed computing systems in which all sorts of guarantees and properties can be achieved. Beyond merely offering rigorous ways of overcoming communication failures, these will include distributed security properties, self-management, load-balancing, and other forms of dynamic adaptation to the environment. All of these are properties that fall under the broad framework of consistent behaviors that are, to a degree, achievable in distributed settings.

We will also discover some hard limits on what can be done in a distributed system. Certain forms of consistency and fault-tolerance are provably impossible, and although these impossibility results leave us a great deal of room to maneuver, they still circumscribe the achievable behaviors of distributed systems with strong properties.

13.2 Warning: Rough Road Ahead!

Before launching into this material, it may be useful to offer a few words of warning to the reader. The material in this chapter and the next two is of a somewhat different nature that most of what we have covered up to the present. Previous chapters have surveyed the state of the art in distributed computing, and to do so it has rarely been necessary to deal with abstractions. In these next few chapters, we'll also be surveying the state of the art, but in an area of distributed computing that is concerned primarily with abstractions, and in which some of the protocols used are very subtle. This may make the treatment seem very abstract and "hard" by comparison with what we have done up to now, and with the style of the remainder of the book, which returns largely to higher level issues.

In the introduction to this book we commented that in a better world, distributed systems engineers could reach over to a well-stocked shelf of tools and technologies for reliable distributed computing, finding powerful computer-aided design tools that would make fault-tolerant computing or other reliability goals transparently achievable. But with the exception of a small number of products (the author is thinking of the Electra system, which will be discussed in Chapter 18, and the Orbix+Isis product line), there are few technologies that offer a plug-and-play approach to reliability.

Despite the emergent character of reliability as a software products market, however, we do know a great deal about building reliable systems, and there are some very powerful research systems that demonstrate how these concepts can be put to work in practice. We'll review quite a few of them, and most of the systems we describe are available to potential users either for research efforts or as free, public-domain software distributions. It seems very likely that the set of available products will also expand rapidly in coming years. Moreover, as we will see in Chapters 16 and 17, some of the existing groupware technologies are designed to be directly useable in their present form, even if better packaging would make it quite a bit easier to do so.

Thus, there is really little alternative but for the potential technology consumer to approach this area by trying to acquire a somewhat deeper perspective on what can, and what cannot, be accomplished, and by reviewing the research prototypes of solutions for the area. To the degree that a developer has deep pockets, many of these technologies could be reimplemented in-house. Less well-heeled developers may be able to work with the existing products, and will often be able to gain access to public-domain technologies or research prototypes. Finally, companies that offer products where reliability might represent a significant market advantage could "size" the development effort that would be required to develop and deploy new product lines with reliability as a major feature by looking at the protocols that are needed by such systems, viewing the research prototypes as proofs of concept that could be imitated at known cost. In the absense of that shelf of reliability technologies in the local software store, this seems to be the most reasonable way to approach the area.

Those readers whose interest in this text is primarily practical may find the remainder of this chapter and Chapters 14, 15 and 16 excessively abstract or "theoretical". The author has made every effort to ensure that Chapters 17-25 can be read without detailed knowledge of the material that follows, and hence such readers should be able to skim these sections without being lost in the remainder of the text.

13.3 Membership in a Distributed System

The first parts of this book have glossed over what turns out to be one of the fundamental questions that we will ask about a distributed system, namely that of determining what processes belong to the system. Before trying to solve this problem, it will be helpful to observe that there are two prevailing interpretations of what membership should mean in a distributed setting. The more widely used interpretation concerns itself with dynamically varying subsets of a static maximal set of processes.

This "static" view of membership corresponds to the behavior of transactional database systems, and other systems in which some set of servers are associated with physical resources such as the databases files being managed, and collection of multimedia images, or some sort of hardware. In such a system, the name of a process would not normally be its process identifier (pid), but rather will be derived from the name of the resource it manages. If there are three identical replicas of a database, we might say that the process managing the first replica is named *a*, the second *b*, and the third *c*, and reuse these same names even if replica *a* crashes and must be rebooted. One would model this type of system as having a fixed maximum membership, {*a,b,c*}, but as operating with a dynamically varying subset of the members. We will call this a static membership model because the subsets are defined over a static population of processes.

Similarly, one can define a "dynamic" model of system membership. In this model, processes are created, execute for a period of time, and then terminate. Each process has a unique name that will never be reused. In a dynamic membership model, the system is defined to be that set of processes that are operational at a given point in time. A dynamic system or set of processes would begin execution when some initial membership is booted, and then would evolve as processes join the system (having been created) or leave the system (having terminated or crashed). Abstractly, the space of possible process names is presumably finite, so one could argue that this dynamic approach to membership isn't really so different from the static one. In practice, however, a static system typically has such a small number of components, perhaps as few as three or four, that the problem is genuinely a very different one. After all, it would not be uncommon for a complex distributed system to spawn hundreds or thousands of processes per hour as it executes. Indeed, many systems of this sort are defined as if the space of processor identifiers were in fact infinite, under the assumption that the system is very unlikely to have any processor that remains operational long enough to actually start reusing identifiers. After all, even if a system spawns 100 processes per second, it would take one and a half years to exhaust a 32-bit process identifier "space".

One could imagine an argument in favor of a hybrid model of system membership; one in which a dynamically managed set of processes surrounds a more static core set of servers. Such a model would be more realistic than the static and dynamic models, because real computing networks do tend to have fixed sets of servers on which the client workstations and computers depend. The main disadvantage of using this mixed model as the overall system model is that it leads to very complex descriptions of system behavior. For this reason, although it is technically feasible to work with a mixed model, the results presented in this text are expressed in terms of the static and dynamic models of system membership. More specifically, we will work by generalizing results from a static model into a more dynamic one.

In designing systems, however, it is often helpful for the developer to keep in mind that even if a dynamic membership model is employed, information about static resources can still be useful. Indeed, by drawing on such information, it may be possible to solve problems that, in a purely dynamic model, could not be solved. For example, we will look at situations where groups of processes have membership that varies over time, increasing as processes join, and decreasing as they depart or fail. Sometimes, if a failure may have partitioned such a group it is hard to know which component of the group "owns" some critical resource. However, suppose that we also notice that two out of three of the computers attached to this resource are accessible in one component of the partitioned group, and hence that at most one of the three is accessible in the other component. Such information is sufficient to let the former component treat itself as the owner of the "service" associated with those computers: the other component will recognize that it must limit its actions to avoid conflict. Thus, even though we focus here on developing a dynamic membership model, by doing so we do not preclude the use of static information as part of the algorithms used by the resulting groups.

13.4 Time in Distributed Systems

In discussing the two views of system membership, we made casual reference to temporal properties of a system. Clearly, the notion of time represents a second fundamental component of any distributed computing model. In the simplest terms, a distributed system is any set of processes that communicate by message passing and carry out desired actions over time. Specifications of distributed behavior often include such terms as "when", "before", "after", and "simultaneously" and we will need to develop the tools to make this terminology rigorous.

In non-distributed settings, time has an obvious meaning — at least to non-physicists. The world is full of clocks, which are accurate and synchronized to varying degrees. Something similar is true for distributed systems: all computers have some form of clock, and clock synchronization services are a standard part of any distributed computing environment. Moreover, just as in any other setting, these clocks have limited accuracy. Two different processes, reading their local clocks at the same instant in (real) time, might observe different values, depending on the quality of the clock synchronization algorithm. Clocks may also drift over long periods of time.

The use of time in a distributed system raises several sorts of problems. One obvious problem is to devise algorithms for synchronizing clocks accurately. In Chapter 20 we will look at several such algorithms, including some very good ones. However, even given very accurate clocks, communication systems operate at such high speeds that the use of physical clocks for fine-grained temporal measurements can only make sense for processes sharing the same clock, for example by operating on the same computer. This leads to something of a quandary: in what sense is it meaningful to say that one event occurs and then another does so, or that two events are concurrent, if no means is available by which a program could label events and compare their times of occurrence?

Looking at this question in 1978, Leslie Lamport proposed a model of logical time that answers this question [Lam78b, Lam84]. Lamport considered sets of processes (they could be static or dynamic) that interact by message passing. In his approach, the execution of a process is modeled as a series of atomic events, each of which requires a single unit of logical time to perform. More precisely, his model represents a process by a tuple $(E_p, <_p)$ where E_p is set of events that occurred within process p, and $<_p$ is a partial order on those events. The advantage of this representation is that it captures any concurrency available within p. Thus, if a and b are events within p, $a <_p b$ means that a happens before b, in some sense meaningful to p. For example, b might be an operation that reads a value written by a, b could have acquired a lock that a released, or p might be executing sequential code in which operation b isn't initiated until after a has terminated.

Notice that there are many levels of granularity at which one might describe the events that occur as a process executes. At the level of the components from which the computer was fabricated, computation consists of concurrent events that implement the instructions or microinstructions executed by the user's program. At a higher level, a process might be viewed in terms of statements in a programming language, control-flow graphs, procedure calls, or units of work that make sense in some external frame of reference, such as operations on a database. Concurrency within a process may arise from interrupt handlers, parallel programming constructs in the language or runtime system, or from the use of lightweight threads. Thus, when we talk about the events that occur within a process, it is understood that the designer of a system will typically have a granularity of representation that seems natural for the distributed protocol or specification at hand, and that events are encoded to this degree of precision. Within this text, most examples will be at a very coarse level of precision, in which we treat all the local computation that occurs within a process between when it sends or receives a first message and when it sends or receives a second message as a single event, or perhaps even as being associated with the send or receive event itself.

Lamport models the sending and receiving of messages as events. Thus, an event a could be the sending of a message m, denoted *snd(m),* the reception of m, denoted *rcv(m),* or the delivery of m to application code, denoted *deliv(m).* When the process at which an event occurs is not clear from context, we will add the process identifier as a subscript: thus, $snd_p(m)$, $rcv_p(m)$ and $deliv_p(m)$. The reasons for separating receive events from delivery events is to be able to talk about protocols that receive a message and do things to it, or delay it, before letting the application program see it. Not every message sent will necessarily be received, and not every message received will necessarily be delivered to the application; the former property depends upon the reliability characteristics of the network, and the latter upon the nature of the message.

Consider a process p with an event *snd(m)* and a process q in which there is a corresponding event *rcv(m),* for the same message m. Clearly, the sending of a message precedes its receipt. Thus, we can introduce an additional partial order that orders send and receive events for the same messages. Denote this communication ordering relation by $<_m$ so that we can write $snd_p(m) <_m rcv_q(m)$.

This leads to a definition of logical time in a distributed system as the transitive closure of the $<_p$ relations for the processes p that comprise the system, and $<_m$. We will write $a \rightarrow b$ to denote the fact that a and b are ordered within this temporal relation, which is often called the potential causality relation for the system. In words, we will say that a happened before b. If neither $a \rightarrow b$ nor $b \rightarrow a$, we will say that a and b occur *concurrently*.

Potential causality is useful in many ways. First, it allows us to be precise when talking about the temporal properties of algorithms used in distributed systems. For example, up to now, when we have used phrasing such as "at a point in time" or "when" in relation to a distributed execution, it may not have been clear just what it means to talk about an instant in time that spans a set of processes composing the system. Certainly, the discussion at the start of this chapter, in which it was noted that clocks in a distributed system will not often be sufficiently synchronized to measure time, should have raised concerns about the notion of simultaneous events. An instant in time should correspond to a set of simultaneous events, one per process in the system, but the most obvious way of writing down such a set (namely, writing the state of each process as that process reaches some designated time) would not physically realizable by any protocol we could implement as a part of such a system.

Consider, however, a set of concurrent events, one per process in a system. Such a set potentially represents an instantaneous snapshot of a distributed system, and even if the events did not occur at precisely the same instant in real time, there is no way to determine this from within the system. We will use the term *consistent cut* to refer to a set of events with this property [CL85]. A consistent snapshot is the full set of events that happen before or on a consistent cut. Note that a consistent snapshot will include the state of communication channels "at the time" of the consistent cut: the messages in the channels will be those for which the snapshot contains a *snd* event but lacks a corresponding *rcv* event.

Figure 13-1 illustrates this notion: the red "cuts" are inconsistent because they include message receive events but exclude the correspond sending events. The green cuts satisfy the consistency property. If one thinks about process execution timelines as if they were made of rubber, the green cuts correspond to possible distortions of the execution in which time never flows "backwards"; the red cuts correspond to distortions that violate this property.

If a program or a person were to look at the state of a distributed system along an inconsistent cut (i.e. by contacting the processes one by one to check each individual state and then assembling a picture of the system as a whole from the data so obtained), the results could be confusing and meaningless. For example, if a system manages some form of data using a lock, it could appear that multiple processes hold the lock simultaneously. To see this, imagine that process *p* holds the lock and then sends a message to process *q* in which it passes the lock to *q*. If our cut happened to show *q* after it received this message (and hence obtained the lock) but showed *p* before it sent it (and hence when it still held the lock), *p* and *q* would appear to both hold the lock. Yet in the real execution, this state never arose. Were a developer trying to debug a distributed system, considerable time could be wasted in trying to sort out real bugs from these sorts of "virtual" bugs introduced as artifacts of the way the system state was collected!

The value of consistent cuts is that they represent states the distributed system might actually have been in at a single instant in real-time. Of course, there is no way to know which of the feasible cuts for a given execution correspond to the "actual" real-time states through which the system passed, but Lamport's observation was that in a practical sense, to even ask this question reveals a basic misunderstanding of the nature of time in distributed systems. In his eyes, the consistent cuts for a distributed system are the *more meaningful* notion of simultaneous states for that system, while external time, being inaccessible within the system, is actually *less* meaningful. Lacking a practical way to make real-time clocks that are accurate to the resolution necessary to accurately timestamp events, he would argue that real-time is in fact not a very useful property for protocols that operate at this level. Of course, we can still use real-time for other purposes that demand lesser degrees of accuracy, and will reintroduce it later, but for the time being, we accept this perspective. Babaoglu and Marzullo discuss some uses of consistent cuts in [BM93].

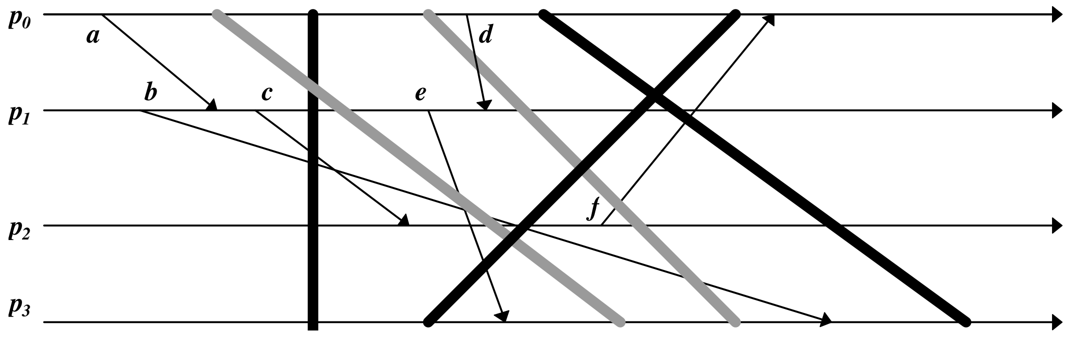


Figure 13-1: Examples of consistent (black) and inconsistent (gray) cuts. The gray cuts illustrate states in which a message receive event is included but the corresponding send event is omitted. Consistent cuts represent system states that could have arisen at a single instant in real-time. Notice, however, that a consistent cut may not actually capture simultaneous states of the processes in question (that is, a cut might be instantaneous in real-time, but there are many consistent cuts that are not at all simultaneous), and that there may be many such cuts through a given point in the history of a process.

Potential causality is a useful tool for reasoning about a distributed system, but it also has more practical significance. There are several ways to build logical clocks with which causal relationships between events can be detected, to varying degrees of accuracy.

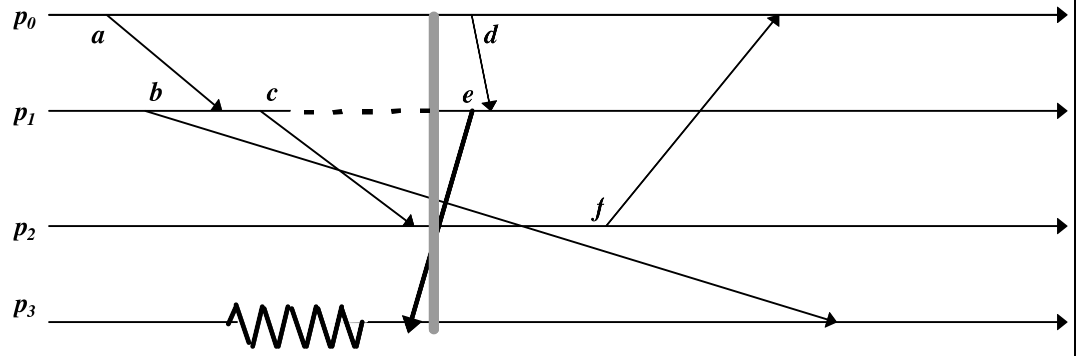


Figure 13-2: Distorted timelines that might correspond to faster or slower executions of the processes illustrated in the previous figure. Here we have redrawn the earlier execution to make an inconsistent cut appear to be physically instantaneous, by slowing down process p_1 (dotted lines) and speeding up p_3 (jagged). But notice that to get the cut "straight" we now have message e̲ travelling "backwards" in time, an impossibility! The black cuts in the earlier figure, in contrast, can all be straightened without such problems. This lends intuition to the idea that a consistent cut is a state that could have occured at an instant in time, while an inconsistent cut is a state that could not have occured in real-time.

A very simple logical clock can be constructed by associating a counter with each process and message in the system. Let LT_p be the logical time for process p (the value of p's copy of this counter), and let LT_m be the logical time associated with message m (also called the logical timestamp of m). The following rules are used to maintain these counters.

1. If $LT_p<LT_m$ process p sets $LT_p = LT_m+1$
2. If $LT_p \geq LT_m$ process p sets $LT_p = LT_p+1$
3. For other events, process p sets $LT_p = LT_p+1$

We will use the notation $LT(a)$ to denote the value of LT_p when event a occurred at process p. It can easily be shown that if $a \rightarrow b$, $LT(a)<LT(b)$: From the definition of the potential causality relation, we know that if $a \rightarrow b$, there must exist a chain of events $a \equiv e_0 \rightarrow e_1 \ldots \rightarrow e_k \equiv b$, where each pair are related either by the event ordering $<_p$ for some process p or by the event ordering $<_m$ on messages. By construction, the logical clock values associated with these events can only increase, establishing the desired result. On the other hand, $LT(a)<LT(b)$ does not imply that $a \rightarrow b$, since concurrent events may have the same timestamps.

For systems in which the set of processes is static, logical clocks can be generalized in a way that permits a more accurate representation of causality. A vector clock is a vector of counters, one per process in the set [Fid88, Mat89, SES89]. Similar to the notation for logical clocks, we will say that VT_p and VT_m represent the vector times associated with process p and message m, respectively. Given a vector time VT, the notation VT[p] denotes the entry in the vector corresponding to process p.

The rules for maintaining a vector clock are similar to the ones used for logical clocks, except that a process only increments its own counter. Specifically:

1. Prior to performing any event, process p sets $VT_p[p] = VT_p[p]+1$
2. Upon delivering a message m, process p sets $VT_p = max(VT_p, VT_m)$

In (2), the function *max* applied to two vectors is just the element by element maximum of the respective entries. We now define two comparison operations on vector times. If $VT(a)$ and $VT(b)$ are vector times, we will say that $VT(a) \leq VT(b)$ *if* $\forall I$: $VT(a)[i] \leq VT(b)[i]$. When $VT(a) \leq VT(b)$ *and* $\exists i$: $VT(a)[i] < VT(b)[i]$ we will write $VT(a) < VT(b)$.

In words, a vector time entry for a process p is just a count of the number of events that have occured at p. If process p has a vector clock with $Vt_p[q]$ set to six, this means that some chain of events has caused p to hear (directly or indirectly) from process q subsequent to the sixth event that occured at process q. Thus, the vector time for an event e tells us, for each process in the vector, how many events occured at that process causally prior to when e occured. If $VT(m) = [17,2,3]$, corresponding to processes $\{p,q,r\}$, we know that 17 events occured at process p that causally precede the sending of m, 2 at process q, and 3 at process r.

It is easy to see that vector clocks accurately encode potential causality. If $a \rightarrow b$, then we again consider a chain of events related by the process or message ordering: $a \equiv e_0 \rightarrow e_1 \ldots \rightarrow e_k \equiv b$. By construction, at each event the vector time can only increase (that is, $VT(e_i) < VT(e_{i+1})$), because each process increments its own vector time entry prior to each operation, and receive operations compute an element by element maximum. Thus, $VT(a) < VT(b)$. However, unlike what a logical clock, the converse also holds: if $VT(a) < VT(b)$, then $a \rightarrow b$. To see this, let p be the process at which event a occurred, and consider $VT(a)[p]$. In the case where b also occurs at process p, we know that $\forall I$: $VT(a)[i] \leq VT(b)[i]$, hence if a and b are not the same event, a must happen before b at p. Otherwise, suppose that b occurs at process q. According to the algorithm, process q only changes $VT_q[p]$ upon delivery of some message m for which $VT(m)[p] > VT_q[p]$ at the event of the delivery. If we denote b as e_k and $deliv(m)$ as $e_{k\text{-}1}$, the send event for m as $e_{k\text{-}2}$, and the sender of m by q', we can now trace a chain of events back to a process q'' from which q' received this vector timestamp entry. Continuing this procedure, we will eventually reach process p. We will now have constructed a chain of events $a \equiv e_0 \rightarrow e_1 \ldots \rightarrow e_k \equiv b$, establishing that $a \rightarrow b$, the desired result.

In English, this tells us that if we have a fixed set of processes and use vector timestamps to record the passage of time, we can accurately represent the potential causality relationship for message sent and received, and other events, within that set. Doing so will also allow us to determine when events are concurrent: this is the case if neither $a \rightarrow b$ nor $b \rightarrow a$.

There has been considerable research on optimizing the encoding of vector timestamps, and the representation presented above is far from the best possible in a large system [Cha91]. For a very large system, it is considered preferable to represent causal time using a set of event identifiers, $\{e_0, e_1, \ldots e_k\}$ such that the events in the set are concurrent and causally precede the event being labeled [Pet87, MM93]. Thus if $a \rightarrow b$, $b \rightarrow d$ and $c \rightarrow d$ one could say that event d took place at causal time $\{b,c\}$ (meaning "after events b and c"), event b at time $\{a\}$, and so forth. In practice the identifiers used in such a representation would be process identifiers and event counters maintained on a per-process basis, hence this *precedence order* representation is recognizable as a compression of the vector timestamp The precedence-order representation is useful in settings where processes can potentially construct the full $\rightarrow$ relation, and in which the level of true concurrency is fairly low. The vector timestamp representation is preferred in settings where the number of participating processes is fairly low and the level of concurrency may be high.

Logical and vector clocks will prove to be powerful tools in developing protocols for use in real distributed applications. For example, with either type of clock we can identify sets of events that are concurrent and hence satisfy the properties required from a consistent cut. The method favored in a specific setting will typically depend upon the importance of precisely representing the potential causal order, and on the overhead that can be tolerated. Notice however that while logical clocks can be used in systems with dynamic membership, this is not the case for a vector clock. All processes that use a vector clock must be in agreement upon the system membership used to index the vector. Thus vector clocks, as formulated here, require a static notion of system membership. (Later we will see that they can be used in systems where membership changes dynamically as long as the places where the changes occur are well defined and no communication spans those "membership change events").

The remainder of this chapter focuses on problems for which logical time, represented through some form of logical timestamp, represents the most natural temporal model. In many distributed applications, however, some notion of "real-time" is also required, and our emphasis on logical time in this section should not be taken as dismissing the importance of other temporal schemes. Methods for synchronizing clocks and for working within the intrinsic limitations of such clocks are the subject of Chapter 20, below.

13.5 Failure Models and Reliability Goals

Any discussion of reliability is necessarily phrased with respect to the reliability "threats" of concern in the setting under study. For example, we may wish to design a system so that its components will automatically restart after crash failures, which is called the *recoverability* problem. Recoverability does not imply continuous availability of the system during the periods before a faulty component has been repaired. Moreover, the specification of a recoverability problem would need to say something about how components fail: through clean crashes that never damage persistent storage associated with them, in other limited ways, in arbitrary ways that can cause unrestricted damage to the data directly managed by the faulty component, and so forth. These are the sorts of problems typically addressed using variations on the transactional computing technologies introduced in Section 7.5, and to which we will return in Chapter 21.

A higher level of reliability may entail *dynamic availability,* whereby the operational components of a system are guaranteed to continue providing correct, consistent behavior even in the presence of some limited number of component failures. For example, one might wish to design a system so that it will remain available provided that at most one failure occurs, under the assumption that failures are clean ones that involve no incorrect actions by the failing component before its failure is detected and it shuts down. Similarly, one might want to guarantee reliability of a critical subsystem up to *t* failures involving arbitrary misbehavior by components of some type. The former problem would be much easier to solve, since the data available at operational components can be trusted; the latter would require a voting scheme in which data is trusted only when there is sufficient evidence as to its validity so that even if *t* arbitrary faults were to occur, the deduced value would still be correct.

At the outset of this book, we gave names to these failures categories: the benign version would be an example of a *halting* failure, while the unrestricted version would fall into the *Byzantine* failure model. An extremely benign (and in some ways not very realistic) model is the *failstop* model, in which machines fail by halting and the failures are *reported* to all surviving members by a notification service (the challenge, needless to say, is implementing a means for accurately detecting failures and turning into a reporting mechanism that can be trusted not to make mistakes!)

In the subsections that follow, we will provide precise definitions of a small subset of the problems that one might wish to solve in a static membership environment subject to failures. This represents a rich area of study and any attempt to exhaustively treat the subject could easily fill a book. However, as noted at the outset, our primary focus in the text is to understand the most appropriate reliability model for realistic distributed systems. For a number of reasons, a dynamic membership model is more closely matched to the properties of typical distributed systems than the static one; even when a system uses a small hardware base that is itself relatively static, we will see that availability goals frequently make a dynamic membership model more appropriate for the application itself. Accordingly, we will confine ourselves here to a small number of particularly important problems, and to a very restricted class of failure models.

13.6 Reliable Computing in a Static Membership Model

The problems on which we now focus are concerned with replicating information in a static environment subject to failstop failures, and with solving the same problem in a Byzantine failure model. By replication, we mean supporting a variable that can be updated or read and that behaves like a single non-faulty variable even when failures occur at some subset of the replicas. Replication may also involve supporting a locking protocol, so that a process needing to perform a series of reads and updates can prevent other processes from interfering with its computation, and in the most general case this problem becomes the transactional one discussed in Chapter 7.5. We'll use replication as a sort of "gold standard" against which various approaches can be compared in terms of cost, complexity, and properties.

Replication turns out to be a fundamental problem for other reasons, as well. As we begin to look at tools for distributed computing in the coming chapters, we will see that even when these tools do something that can seem very far from "replication" per se, they often do so by replicating other forms of state that permit the members of a set of processes to cooperate implicitly by looking at their local copies of this replicated information.

Some examples of replicated information will help make this point clear. The most explicit form of replicated data is simply a replicated variable of some sort. In a bank, one might want to replicate the current holdings of Yen as part of a distributed risk-management strategy that seeks to avoid over-exposure to Yen fluctuations. Replication of this information means that it is made locally accessible to the traders (perhaps world-wide): their computers don't need to fetch this data from a central database in New York but have it directly accessible at all times. Obviously, such a model entails supporting updates from many sources, but it should also be clear why one might want to replicate information this way. Notice also that by replicating this data, the risk that it will be inaccessible when needed (because lines to the server are overloaded or the server itself is down) is greatly reduced.

Similarly, a hospital might want to view a patient's medication record as a replicated data item, with copies on the workstation of the patient's physician, displayed on a "virtual chart" at the nursing station, visible next to the bed on a status display, and availably on the pharmacy computer. One could, of course, build such a system to use a central server and design all of these other applications as clients of the server that poll it periodically for updates, similar to the way that a web proxy refreshes cached documents by polling their home server. But it may be preferable to view the data as replicated if, for example, each of the applications needs to represent it in a different way, and needs to guarantee that its version is up to date. In such a setting, the data really is replicated in the conceptual sense, and although one might chose to implement the replication policy using a client-server architecture, doing so is basically an implementation decision. Moreover, such a central-server architecture would create a single point of failure for the hospital, which can be highly undesirable.

An air traffic control system needs to replicate information about flight plans and current trajectories and speeds. This information resides in the database of each air traffic control center that tracks a given plane, and may also be visible on the workstation of the controller. If plans to develop "free flight" systems advance, such information will also need to be replicated within the cockpits of planes that are close to one-another. Again, one could implement such a system with a central server, but doing so in a setting as critical as air traffic control makes little sense: the load on a central server would be huge, and the single point failure concerns would be impossible to overcome. The alternative is to view the system as one in which this sort of data is replicated.

We previously saw that web proxies can maintain copies of web documents, caching them to satisfy "get" requests without contacting the document's home server. Such proxies form a group that replicate the document — although in this case, the web proxies typically would not know anything about each other, and the replication algorithm depends upon the proxies polling the main server and noticing changes. Thus, document replication in the web is not able to guarantee that data will be consistent. However, one could imagine modifying a web server so that when contacted by caching proxy servers of the same "make", it tracks the copies of its documents and explicitly refreshes them if they change. Such a step would introduce consistent replication into the web, an issue about which we will have much more to say in Sections 17.3 and 17.4.

Distributed systems also replicate more subtle forms of information. Consider, for example, a set of database servers on a parallel database platform. Each is responsible for some part of the load and backs up some other server, taking over for it in the event that it should fail (we'll see how to implement such a structure below). These servers replicate information concerning which servers are included in the system, which server is handling a given part of the database, and what the status of the servers (operational or failed) is at a given point in time. Abstractly, this is replicated data which the servers use to drive their individual actions. As above, one could imagine designating one special server as the "master" which distributes the rules on the basis of which the others operate, but that would just be one way of implementing the replication scheme.

Finally, if a server is extremely critical, one can "actively replicate" it by providing the same inputs to two or more replicas [BR96, Bir91, BR94, Coo85, BJ87a, RBM96]. If the servers are deterministic, they will now execute in lock step, taking the same actions at the same time, and thus providing tolerance of limited numbers of failures. A checkpoint/restart scheme can then be introduced to permit additional servers to be launched as necessary.

Thus, replication is an important problem in itself, but also because it underlies a great many other distributed behaviors. One could, in fact, argue that replication is the most fundamental of the distributed computing paradigms. By understanding how to solve replication as an abstract problem, we will also gain insight into how these other problems can be solved.

13.6.1 The Distributed Commit Problem

We begin by discussing a classical problem that arises as a subproblem in several of the replication methods that follow. This is the *distributed commit* problem, and involves performing an operation in an all-or-nothing manner [Gra79, GR93].

The commit problem arises when we wish to have a set of processes that all agree on whether or not to perform some action that may not be possible at some of the participants. To overcome this initial uncertainty, it is necessary to first determine whether or not all the participants will be able to perform the operation, and then to communicate the outcome of the decision to the participants in a reliable way (the assumption is that once a participant has confirmed that it can perform the operation, this remains true even if it subsequently crashes and must be restarted). We say that operation can be *committed* if the participants should all perform it Once a commit decision is reached, this requirement will hold even if some participants fail and later recover. On the other hand, if one or more participants are unable to perform the operation when initially queried, or some can't be contacted, the operation as a whole *aborts*, meaning that no participant should perform it.

Consider a system composed of a static set S containing processes $\{p_0, p_1, \ldots p_n\}$ that fail by crashing and that maintain both *volatile* data, which is lost if a crash occurs, and *persistent* data, which can be recovered after a crash in the same state that it had at the time of the crash. An example of persistent data would be a disk file; volatile data is any information in a processor's memory on some sort of a scratch area that will not be preserved if the system crashes and must be rebooted. It is frequently much cheaper to store information in volatile data hence it would be common for a program to write intermediate results of a computation to volatile storage. The commit problem will now arise if we wish to arrange for all the volatile information to be saved persistently. The all-or-nothing aspects of the problem reflect the possibility that a computer might fail and lose the volatile data it held; in this case the desired outcome would be that no changes to any of the persistent storage areas occur.

As an example, we might wish for all of the processes in S to write some message into their persistent data storage. During the initial stages of the protocol, the message would be sent to the processes which would each store it into their volatile memory. When the decision is made to try and commit this data, the processes clearly cannot just modify the persistent area, because some process might fail before doing so. Consequently, the commit protocol involves first storing the volatile information into a persistent but "temporary" region of storage. Having done so, the participants would signal their ability to commit.

If all the participants are successful, it is safe to begin transfers from the temporary area to the "real" data storage region. Consequently, when these processes are later told that the operation as a whole should commit, they would copy their temporary copies of the message into a permanent part of the persistent storage area. On the other hand, if the operation aborts, they would not perform this copy operation. As should be evident, the challenge of the protocol will be to handle with the recovery of a participant from a failed state; in this situation, it must determine whether any commit protocols were pending at the time of its failure and, if so, whether they terminated in a commit or an abort state.

A distributed commit protocol is normally initiated by a process that we will call the *coordinator;* assume that this is process p_0 . In a formal sense, the objective of the protocol is for p_0 to solicit votes for or against a commit from the processes in *S*, and then to send a *commit* message to those processes only if all of the votes are in favor commit, and otherwise to send an *abort*. To avoid a trivial solution in which p_0 always sends an *abort*, we would ideally like to require that if all processes vote for commit and no communication failures occur, the outcome should be commit. Unfortunately, however, it is easy to see that such a requirement is not really meaningful because communication failures can prevent messages from reaching the coordinator. Thus, we are forced to adopt a weaker non-triviality requirement, by saying that if all processes vote for commit and all the votes reach the coordinator, the protocol should commit.

A commit protocol can be implemented in many ways. For example, RPC could be used to query the participants and later to inform them of the outcome, or a token could be circulated among the participants which they would each modify before forwarding, indicating their vote, and so forth. The most standard implementations, however, are called two- and three-phase commit protocols, often abbreviated as 2PC and 3PC in the literature.

13.6.1.1 Two-Phase Commit

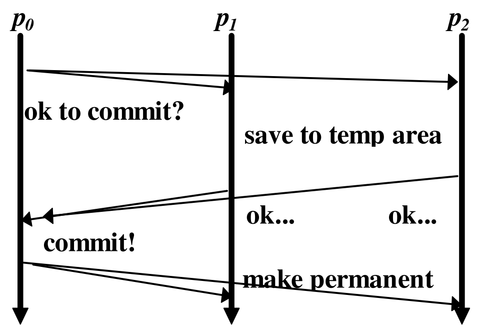
A 2PC protocol operates in rounds of multicast communication. Each phase is composed of one round of messages to the participants, and one round of replies from the recipients to the sender. The coordinator initially selects a unique identifier for this run of the protocol, for example by concatenating its own process id to the value of a logical clock. The protocol identifier will be used to distinguish the messages associated with different runs of the protocol that happen to execute concurrently, and in the remainder of this section we will assume that all the messages under discussion are labeled by this initial identifier.

The coordinator starts by sending out a first round of messages to the participants. These messages normally contain the protocol identifier, the list of participants (so that all the participants will know who the other participants are), and a message "type" indicating that this is the first round of a 2PC protocol. In a static system where all the processes in the system participate in the 2PC protocol, the list of participants can be omitted because it has a well-known value. Additional fields can be added to this message depending on the situation in which the 2PC was needed. For example, it could contain a description of the action that the coordinator wishes to take (if this is not obvious to the participants), a reference to some volatile information that the coordinator wishes to have copied to a persistent data area, and so forth. 2PC is thus a very general tool that can solve any of a number of specific problems, which share the attribute of needing an all-or-nothing outcome and the property that participants must be asked if they will be able to perform the operation before it is safe to assume that they can do so.

Each participant, upon receiving the first round message, takes such local actions as are needed to decide if it can vote in favor of commit. For example, a participant may need to set up some sort of persistent data structure, recording that the 2PC protocol is underway and saving the information that will be needed to perform the desired action if a commit occurs. In the example from above, the participant would copy its volatile data to the temporary persistent region of the disk and then "force" the records to the disk. Having done this (which may take some time), the participant sends back its vote. The coordinator collects votes, but also uses a timer to limit the duration of the first phase (the initial round of outgoing messages and the collection of replies). If a timeout occurs before the first phase replies have all been collected, the coordinator aborts the protocol. Otherwise, it makes a commit or abort decision according to the votes it collects.[7]

Now we enter the second phase of the protocol, in which the coordinator sends out commit or abort messages in a new round of communication. Upon receipt of these messages, the participants take the desired action or, if the protocol is aborted, they delete the associated information from their persistent data stores. Figure 13-3 illustrates this basic skeleton of the 2PC protocol.

[7] As described, this protocol already violates the non-triviality goal that we expressed earlier. No timer is really "safe" in an asynchronous distributed system, because an adversary could just set the minimum message latency to the timer value plus one second, and in this way cause the protocol to abort despite the fact that all processes vote commit and all messages will reach the coordinator. Concerns such as this can seem unreasonably narrowminded, but are actually important in trying to pin down the precise conditions under which commit is possible. The practical community (to which this textbook is targetted) tends to be fairly relaxed about such issues, while the theory community (whose work this author tries to follow closely) tends to take problems of this sort very seriously. It is regretable but perhaps inevitable that some degree of misunderstanding results from these different points of view. In reading this particular treatment, the more formally inclined reader is urged to interpret the text to mean what the author meant to say, not what he wrote!



```
Coordinator:
   multicast: ok to commit?
   collect replies
      all ok  => send commit
      else    => send abort
Participant:
   ok to commit =>
      save to temp area, reply ok
   commit =>
      make change permanent
   abort =>
      delete temp area
```

Figure 13-3: Skeleton of two-phase commit protocol

Several failure cases need to be addressed. The coordinator could fail before starting the protocol, during the first phase, while collecting replies, after collecting replies but before sending the second phase messages, or during the transmission of the second phase messages. The same is true for a participant. For each case we need to specify a recovery action that leads to successful termination of the protocol with the desired all-or-nothing semantics.

In addition to this, the protocol described above omits consideration of the storage of information associated with the run. In particular, it seems clear that the coordinator and participants should not need to keep any form of information "indefinitely" in a correctly specified protocol. Our protocol makes use of a protocol identifier, and we will see that the recovery mechanisms require that some information been saved for a period of time, indexed by protocol identifier. Thus, rules will be needed for garbage collection of information associated with terminated 2PC protocols. Otherwise, the information-base in which this data is stored might grow without limit, ultimately posing serious storage and management problems.

We start by focusing on participant failures, then turn to the issue of coordinator failure, and finally to this question of garbage collection.

Suppose that a process p_i fails during the execution of a 2PC protocol. With regard to the protocol, p_i may be any of several states. In its initial state, p_i will be "unaware" of the protocol. In this case, p_i will not receive the initial vote message, hence the coordinator aborts the protocol. The initial state ends when p_i has received the initial vote request and is prepared to send back a vote in favor of commit (if p_i doesn't vote for commit, or isn't yet prepared, the protocol will abort in any case). We will now say that p_i *is prepared to commit*. In the prepared to commit state, p_i is compelled to learn the outcome of the protocol even if it fails and later recovers. This is an important observation because the applications that use 2PC often must lock critical resources or limit processing of new requests by p_i while it is prepared to commit. This means that until p_i learns the outcome of the request, it may be unavailable for other types of processing. Such a state can result in denial of services. The next state entered by p_i is called the *commit* or *abort* state, in which it knows the outcome of the protocol. Failures that occur at this stage must not be allowed to disrupt the termination actions of p_i, such as the release of any resources that were tied up during the prepared state. Finally, p_i returns to its initial state, garbage collecting all information associated with the execution of the protocol and retaining only the effects of any committed actions.

From this discussion, we see that a process recovering from a failure will need to determine whether or not it was in a prepared to commit, commit, or abort state at the moment of the failure. In a prepared to commit state, the process will need to find out whether the 2PC protocol terminated in a commit or abort, so there must be some form of system service or protocol outcome file in which this information is logged. Having entered a commit or abort state, the process needs a way to complete the commit or abort action even if it is repeatedly disrupted by failures in the act of doing so. We say that the action must be *idempotent*, meaning that it can be performed repeatedly without ill effects. An example of an idempotent action would be copying a file from one location to another: provided that access to the target file is disallowed until the copying action completes, the process can copy the file once or many times with the same outcome. In particular, if a failure disrupts the copying action, it can be restarted after the process recovers.

Not surprisingly, many systems that use 2PC are structured to take advantage of this type of file copying. In the most common approach, information needed to perform the commit or abort action is saved in a *log* on the persistent storage area. The commit or abort state is represented by a bit in a table, also stored in the persistent area, describing pending 2PC protocols, indexed by protocol identifier. Upon recovery, a process first consults this table to determine the actions it should take, and then uses the log to carry out the action. Only after successfully completing the action does a process delete its knowledge of the protocol and garbage collect the log records that were needed to carry it out.

Up to now, we have not considered coordinator failure, hence it would be reasonable to assume that the coordinator itself plays the role of tracking the protocol outcome and saving this information until all participants are known to have completed their commit or abort actions. The 2PC protocol thus needs a final phase in which messages flow back from participants to the coordinator, which must retain information about the protocol until all such messages have been received.

```
Coordinator:
    multicast: ok to commit?
    collect replies
        all ok  => log "commit" to "outcomes" table
                   send commit
        else    => send abort
    collect acknowledgments
    garbage-collect protocol outcome information
```

```
Participant:
    ok to commit =>
        save to temp area, reply ok
    commit =>
        make change permanent
    abort =>
        delete temp area

After failure:
    for each pending protocol
        contact coordinator to learn outcome
```

Figure 13-4: 2PC extended to handle participant failures.

Consider next the case where the coordinator fails during a 2PC protocol. If we are willing to wait for the coordinator to recover, the protocol requires few changes to deal with this situation. The first change is to modify the coordinator to save its commit decision to persistent storage *before* sending commit or abort messages to the participants.[8] Upon recovery, the coordinator is now guaranteed to have available the information needed to terminate the protocol, which it can do by simply retransmitting the final commit or abort message. A participant that is not in the precommit state would acknowledge such a message but take no action; a participant waiting in the precommit state would terminate the protocol upon receipt of it.

[8] It is actually sufficient for the coordinator to save only commit decisions in persistent storage. After failure, a recovering coordinator can safely presume the protocol to have aborted if it finds no commit record; the advantage of such a change is to make the abort case less costly, by removing a disk I/O operation from the "critical path" before the abort can be acted upon. The elimination of a single disk I/O operation may seem like a minor optimization, but in fact can be quite significant, in light of the 10-fold latency difference between a typical disk I/O operation (10-25ms) and a typical network communication operation (perhaps 1-4ms latency). One doesn't often have an opportunity to obtain an order of magnitude performance improvement in a critical path, hence these are the sorts of engineering decisions that can have very important implications for overall system performance!

```
Coordinator:
    multicast: ok to commit?
    collect replies
        all ok  => log "commit" to "outcomes" table
                   wait until safe on persistent store
                   send commit
        else    => send abort
    collect acknowledgements
    garbage-collect protocol outcome information

After failure:
    for each pending protocol in outcomes table
        send outcome (commit or abort)
        wait for acknowledgements
        garbage-collect outcome information
```

```
Participant: first time message received
    ok to commit =>
        save to temp area, reply ok
    commit =>
        make change permanent
    abort =>
        delete temp area

Message is a duplicate (recovering coordinator)
    send acknowledgment

After failure:
    for each pending protocol
        contact coordinator to learn outcome
```

Figure 13-5: 2PC protocol extended to overcome coordinator failures

One major problem with this solution to 2PC is that if a coordinator failure occurs, the participants are blocked, waiting for the coordinator to recover. As noted earlier, precommit often ties down resources or involves holding locks, hence blocking in this manner can have serious implications for system availability. Suppose that we permit the participants to communicate among themselves. Could we increase the availability of the system so as to guarantee progress even if the coordinator crashes?

Again, there are three stages of the protocol to consider. If the coordinator crashes during its first phase of message transmissions, a state may result in which some participants are prepared to commit, others may be unable to commit (they have voted to abort, and know that the protocol will eventually do so), and still other processes may not know anything at all about the state of the protocol. If it crashes during its decision, or before sending out all the second-phase messages, there may be a mixture of processes left in the prepared state and processes that know the final outcome.

Suppose that we add a timeout mechanism to the participants: in the prepared state, a participant that does not learn the outcome of the protocol within some specified period of time will timeout and seek to complete the protocol on its own. Clearly, there will be some unavoidable risk of a timeout that occurs because of a transient network failure, much as in the case of RPC failure detection mechanisms discussed early in the text. Thus, a participant that takes over in this case cannot safely conclude that the coordinator has actually failed. Indeed, any mechanism for takeover will need to work even if the timeout is set to 0, and even if the participants try to run the protocol to completion starting from the instant that they receive the phase 1 message and enter a prepared to commit state!

Accordingly, let p_i be some process that has experienced a protocol timeout in the prepared to commit state. What are p_i's options? The most obvious would be for it to send out a first-phase message of its own, querying the state of the other p_j. From the information gathered in this phase, p_i may be able to deduce that the protocol committed or aborted. This would be the case if, for example, some process p_j had received a second-phase outcome message from the coordinator before it crashed. Having determined the outcome, p_i can simply repeat the second-phase of the original protocol. Although participants may receive as many as n copies of the outcome message (if all the participants time out simultaneously), this is clearly a safe way to terminate the protocol.

On the other hand, it is also possible that p_i would be unable to determine the outcome of the protocol. This would occur, for example, if all processes contacted by p_i, as well as p_i itself, were in the prepared state, with a single exception: process p_j, which does not respond to the inquiry message. Perhaps, p_j has failed, or perhaps the network is temporarily partitioned. The problem now is that only the coordinator and p_j can determine the outcome, which depends entirely on p_j's vote. If the coordinator is itself a participant, as is often the case, a single failure can thus leave the 2PC participants blocked until the failure is repaired! This risk is unavoidable in a 2PC solution to the commit problem.

Earlier, we discussed the garbage collection issue. Notice that in this extension to 2PC, participants must retain information about the outcome of the protocol until they are certain that all participants know the outcome. Otherwise, if a participant p_j were to commit but "forget" that it had done so, it would be unable to assist some other participant p_i in terminating the protocol after a coordinator failure.

Garbage collection can be done by adding a third phase of messages from the coordinator (or a participant who takes over from the coordinator) to the participants. This phase would start after all participants have acknowledged receipt of the second-phase commit or abort message, and would simply tell participants that it is safe to garbage collect the protocol information. The handling of coordinator failure can be similar to that during the pending state. A timer is set in each participant that has entered the final state but not yet seen the garbage collection message. Should the timer expire, such a participant can simply echo out the commit or abort message, which all other participants acknowledge. Once all participants have acknowledged the message, a garbage collection message can be sent out and the protocol state safely garbage collected.

Notice that the final round of communication, for purposes of garbage collection, can often be delayed for a period of time and then run once in a while, on behalf of many 2PC protocols at the same time. When this is done, the garbage collection protocol is itself best viewed as a 2PC protocol that executes perhaps once per hour. During its first round, a garbage collection protocol would solicit from each process in the system the set of protocols for which they have reached the final state. It is not difficult to see that if communication is FIFO in the system, then 2PC protocols — even if failures occur — will complete in FIFO order. This being the case, each process need only provide a single protocol identifier, per protocol coordinator, to in response to such an inquiry: the identifier of the last 2PC initiated by the coordinator to have reached its final state. The process running the garbage collection protocol can then compute the minimum over these values. For each coordinator, the minimum will be a 2PC protocol identifier which has fully terminated at all the participant processes, and hence which can be garbage-collected throughout the system.

```
Coordinator:
    multicast: ok to commit?
    collect replies
        all ok  => log "commit" to "outcomes" table
                   wait until safe on persistent store
                   send commit
        else    => send abort
    collect acknowledgements

After failure:
   for each pending protocol in outcomes table
        send outcome (commit or abort)
        wait for acknowledgements

Periodically:
    query each process: terminated protocols?
    for each coordinator: determine fully
      terminated protocols
    2PC to garbage collect outcomes information
```

```
Participant: first time message received
    ok to commit =>
        save to temp area, reply ok
    commit =>
        log outcome, make change permanent
    abort =>
        log outcome, delete temp area

Message is a duplicate (recovering coordinator)
    send acknowledgment

After failure:
    for each pending protocol
        contact coordinator to learn outcome

After timeout in prepare to commit state:
    query other participants about state
      outcome can be deduced =>
          run coordinator-recovery protocol
      outcome uncertain =>
          must wait
```

Figure 13-6: Final version of 2PC commit, participants attempt to terminate protocol without blocking, periodic 2PC protocol used to garbage collect outcomes information saved by participants and coordinators for recovery.

We thus arrive at the "final" version of the 2PC protocol shown in Figure 13-6. Notice that this protocol has a potential message complexity that grows as $O(n^2)$ with the worst case occurring if a network communication problem disrupts communication during the three basic stages of communication. Further, notice that although the protocol is commonly called a "two phase" commit, a true two-phase version will always block if the coordinator fails. The version of Figure 13-6 gains a higher degree of availability at the cost of additional communication for purposes of garbage collection. However, although this protocol may be more available than our initial attempt, it can still block if a failure occurs at a critical stage. In particular, participants will be unable to terminate the protocol if a failure of both the coordinator and a participant occurs during the decision stage of the protocol.

13.6.1.2 Three-Phase Commit

Skeen and Stonebraker studied the cases in which 2PC can block, in 1981 [Ske82b]. Their work resulted in a protocol called *three-phase commit* (3PC), which is guaranteed to be non-blocking provided that only failstop failures occur. Before we present this protocol, it is important to stress that the failstop model is not a very realistic one: this model requires that processes fail only by crashing and that such failures *be accurately detectable* by other processes that remain operational. Inaccurate failure detections and network partition failures continue to pose the threat of blocking in this protocol, as we shall see. In practice, these considerations limit the utility of the protocol (because we lack a way to accurately sense failures in most systems, and network partitions are a real threat in most distributed environments). Nonetheless, the protocol sheds light both on the issue of blocking and on the broader notion of consistency in distributed systems, hence we present it here.

As in the case of the 2PC protocol, 3PC really requires a fourth phase of messages for purposes of garbage collection. However, this problem is easily solved using the same method that was presented in

Figure 13-6 for the case of 2PC. For brevity, we therefore focus on the basic 3PC protocol and overlook the garbage collection issue.

Recall that 2PC blocks under conditions in which the coordinator crashes and one or more participants crash, such that the operational participants are unable to deduce the protocol outcome without information that is only available at the coordinator and/or these participants. The fundamental problem is that in a 2PC protocol, the coordinator can make a commit or abort decision that would be known to some participant p_j and even acted upon by p_j, but totally unknown to other processes in the system. The 3PC protocol prevents this from occurring by introducing an additional round of communication, and delaying the "prepared" state until processes receive this phase of messages. By doing so, the protocol ensures that the state of the system can always be deduced by a subset of the operational processes, provided that the operational processes can still communicate reliably among themselves.

```
Coordinator:
    multicast: ok to commit?
    collect replies
        all ok  => log "precommit"
                   send precommit
        else    => send abort
    collect acks from non-failed participants
        all ack  => log "commit"
                    send commit
    collect acknowledgements
    garbage-collect protocol outcome information
```

```
Participant: logs "state" on each message
    ok to commit =>
        save to temp area, reply ok
    precommit =>
        enter precommit state, acknowledge
    commit  =>
        make change permanent
    abort  =>
        delete temp area

After failure:
    collect participant state information
    all precommit, or any committed =>
        push forward to commit
    else =>
        push back to abort
```

Figure 13-7: Outline of a 3-phase commit protocol

A typical 3PC protocol operates as shown in Figure 13-7. As in the case of 2PC, the first round message solicits votes from the participants. However, instead of entering a prepared state, a participant that has voted for commit enters an *ok to commit* state. The coordinator collects votes and can immediately abort the protocol if some votes are negative, or if some votes are missing. Unlike for a 2PC, it does not immediately commit if the outcome is unanimously positive. Instead, the coordinator sends out a round of *prepare to commit* messages, receipt of which cases all participants to enter the prepare to commit state and to send an acknowledgment. After receiving acknowledgements from all participants, the coordinator sends *commit* messages and the participants commit. Notice that the *ok to commit* state is similar to the *prepared* state in the 2PC protocol, in that a participant is expected to remain capable of committing even if failures and recoveries occur after it has entered this state.

If the coordinator of a 3PC protocol detects failures of some participants (recall that in this model, failures are accurately detectable), and has not yet received their acknowledgements to its *prepare to commit* messages, the 3PC can still be committed. In this case, the unresponsive participants can be counted upon to run a recovery protocol when the cause of their failure is repaired, and that protocol will lead them to eventually commit. The protocol thus has the property that it will only commit if all operational participants are in the *prepared to commit* state. This observation permits any subset of operational participants to terminate the protocol safely after a crash of the coordinator and/or other participants.

The 3PC termination protocol is similar to the 2PC protocol, and starts by querying the state of the participants. If any participant knows the outcome of the protocol (commit or abort), the protocol can be terminated by disseminating that outcome. If the participants are all in a prepared to commit state, the protocol can safely be committed.

Suppose, however, that some mixture of states is found in the state vector. In this situation, the participating processes have the choice of driving the protocol forward to a commit or back to an abort. This is done by rounds of message exchange that either move the full set of participants to *prepared to commit* and thence to a *commit*, or that back them to *ok to commit* and then abort. Again, because of the failstop assumption, this algorithm runs no risk of errors. Indeed, the processes have a simple and natural way to select a new coordinator at their disposal: since the system membership is assumed to be static, and since failures are detectable crashes (the failstop assumption), the operational process with the lowest process identifier can be assigned this responsibility. It will eventually recognize the situation and will then take over, running the protocol to completion.

Notice also that even if additional failures occur, the requirement that the protocol only commit once all operational processes are in a *prepared to commit* state, and only abort when all operational processes have reached an *ok to commit* state (also called *prepared to abort*) eliminates many possible concerns. However, this is true only because failures are accurately detectable, and because processes that fail will always run a recovery protocol upon restarting.

It is not hard to see how this recovery protocol should work. A recovering process is compelled to track down some operational process that knows the outcome of the protocol, and to learn the outcome from that process. If all processes fail, the recovering process must identify the subset of processes that were the last to fail [Ske85], learning the protocol outcome from them. In the case where the protocol had not reached a commit or abort decision when all processes failed, it can be resumed using the states of the participants that were the last to fail, together with any other participants that have recovered in the interim.

Unfortunately, however, the news for 3PC is actually not quite so good as this protocol may make it seem, because real systems do not satisfy the failstop failure assumption. Although there may be some specific conditions under which failures are by detectable crashes, these most often depend upon special hardware. In a typical network, failures are only detectable using timeouts, and the same imprecision that makes reliable computing difficult over RPC and streams also limits the failure handling ability of the 3PC.

The problem that arises is most easily understood by considering a network partitioning scenario, in which two groups of participating processes are independently operational and trying to terminate the protocol. One group may see a state that is entirely *prepared to commit* and would want to terminate the protocol by commit. The other, however, could see a state that is entirely *ok to commit* and would consider abort to be the only safe outcome: after all, perhaps some unreachable process voted against commit! Clearly, 3PC will be unable to make progress in settings where partition failures can arise. We will return to this issue in Section 13.8, when we discuss a basic result by Fisher, Lynch and Paterson; the inability to terminate a 3PC protocol in settings that don't satisfy failstop-failure assumptions is one of many manifestations of the so-called "FLP impossibility" result [FLP85, Ric96]. For the moment, though, we find ourselves in the uncomfortable position of having a solution to a problem that is similar to, but not quite identical to, the one that arises in real systems. One consequence of this is that few systems make use of 3PC commit protocols today: given a situation in which 3PC is "less likely" to block than 2PC, but may nonetheless block when certain classes of failures occur, the extra cost of the 3PC is not generally seen as bringing a return commensurate with its cost.

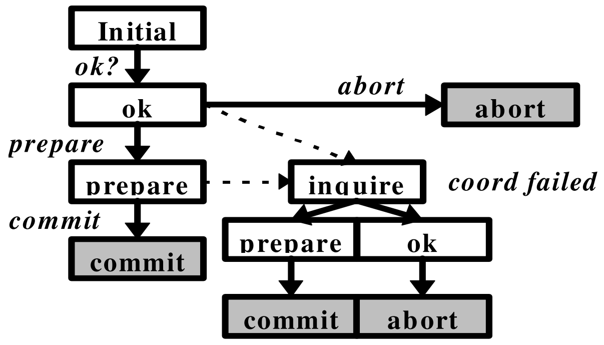


Figure 13-8: States for a non-faulty participant in 3PC protocol

13.6.2 Reading and Updating Replicated Data with Crash Failures

The 2PC protocol represents a powerful tool for solving end-user applications. In this section, we focus on the use of 2PC to implement a data replication algorithm in an environment where processes fail by crashing. Notice that we have returned to a realistic failure model here, hence the 3PC protocol would offer few advantages.

Accordingly, consider a system composed of a static set *S* containing processes $\{p_0, p_1, \ldots p_n\}$ that fail by crashing and that maintain volatile and persistent data. Assume that each process p_i maintains a local replica of some data object, which is updated by operation $update_i$ and read using operation $read_i$. Each operation, both local and distributed, returns a value for the replicated data object. Our goal is to define distributed operations *UPDATE* and *READ* that remain available even when $t<n$ processes have failed, and that return results indistinguishable from those that might be returned by a single, non-faulty process. Secondary goals are to understand the relationship between t and n and to determine the maximum level of availability that can be achieved without violating the "one copy" behavior of the distributed operations.

The best known solutions to the static replication problem are based on *quorum* methods Tho87, Ske82a, Gif79]. In these methods, both *UPDATE* and *READ* operations can be performed on less than the full number of replicas, provided however that there is a guarantee of overlap between the replicas at which any successful *UPDATE* is performed, and those at which any other *UPDATE* or any successful *READ* is performed. Let us denote the number of replicas that must be read to perform a *READ* operation by $q_{r,}$ and the number to perform an *UPDATE* by q_u . Our quorum overlap rule requires us that we need $q_r + q_u > n$ and that $q_u + q_u > n$.

An implementation of a quorum replication method associates a *version number* with each data item. The version number is just a counter that will be incremented by each attempted update. Each replica will include a copy of the data object, together with the version number corresponding to the update that wrote that value into the object.

To perform a *READ* operation, a process reads q_r replicas and discards any replicas with version numbers smaller than those of the others. The remaining values should all be identical, and the process treats any of these as the outcome of its *READ* operation.

To perform an *UPDATE* operation, the 2PC protocol must be used. The updating process first performs a *READ* operation to determine the current version number and, if desired, the value of the data item. It calculates the new value of the data object, increments the version number, and then initiates a 2PC protocol to write the value and version number to q_u or more replicas. In the first stage of this protocol, a replica votes to abort if the version number it already has stored is larger than the version number proposed in the update. Otherwise, it locks out read requests to the same item and waits in an *ok to commit* state. The coordinator will commit the protocol if it receives only *commit* votes, and if it is successful in contacting at least q_u or more replicas; otherwise, it aborts the protocol. If new read operations occur during the *ok to commit* state, they are delayed until the commit or abort decision is reached. On the other hand, if new updates arrive during the *ok to commit* state, the participant votes to abort them

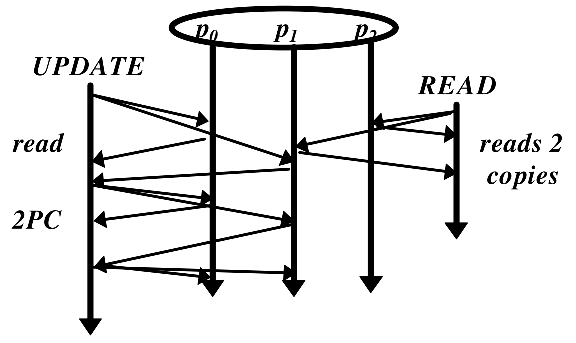


Figure 13-9: Quorum update algorithm uses a quorum-read followed by a 2PC protocol for updates

Our solution raises several issues. First, we need to be convinced that it is correct, and to understand how it would be used to build a replicated object tolerant of *t* failures. A second issue is to understand the behavior of the replicated object if recoveries occur. The last issue to be addressed concerns concurrent systems: as stated, the protocol may be prone to livelock (cycles in which one or more updates are repeatedly aborted).

With regard to correctness, notice that the use of 2PC ensures that an *UPDATE* operation either occurs at q_u replicas or at none. Moreover, *READ* operations are delayed while an *UPDATE* is in progress. Making use of the quorum overlap property, it is easy to see that if an *UPDATE* is successful, any subsequent *READ* operation must overlap with it at least one replica, and the *READ* will therefore reflect the value of that *UPDATE,* or of a subsequent one. If two *UPDATE* operations occur concurrently, one or both will abort. Finally, if two *UPDATE* operations occur in some order, then since the *UPDATE* starts with a *READ* operation, the later *UPDATE* will use a larger version number than the earlier one, and its value will be the one that persists.

To tolerate *t* failures, it will be necessary that the *UPDATE* quorum, q_u be no larger than n-t. It follows that the *READ* quorum, q_r, must have a value larger than *t*. For example, in the common case where we wish to guarantee availability despite a single failure, *t* will equal 1. The *READ* quorum will therefore need to be at least 2, implying that a minimum of 3 copies are needed to implement the replicated data object. If 3 copies are in fact used, the *UPDATE* quorum would also be set to 2. We could also use extra copies: with 4 copies, for example, the *READ* quorum could be left at 2 (one typically wants reads to be as fast as possible and hence would want to read as few copies as possible), and the *UPDATE* quorum increased to 3, guaranteeing that any *READ* will overlap with any prior *UPDATE* and that any pair of *UPDATE* operations will overlap with one another. Notice, however, that with 4 copies, 3 is the smallest possible *UPDATE* quorum.

Our replication algorithm places no special constraints on the recovery protocol, beyond those associated with the 2PC protocol itself. Thus, a recovering process simply terminates any pending 2PC protocols and can then resume participation in new *READ* and *UPDATE* algorithms.

Turning finally to the issue of concurrent *UPDATE* operations, it is evident that there may be a real problem here. If concurrent operations of this sort are required, they can easily force one another to abort. Presumably, an aborted *UPDATE* would simply be reissued, hence a livelock can arise. One solution to this problem is to protect the *UPDATE* operation using a locking mechanism, permitting concurrent *UPDATE* requests only if they access independent data items. Another possibility is employ some form of backoff mechanism, similar to the one used by an ethernet controller. Later, when we consider dynamic process groups and atomic multicast, we will see additional solutions to this problem.

What should the reader conclude about this replication protocol? One important conclusion is that the protocol does not represent a very good solution to the problem, and will perform very poorly in comparison with some of the dynamic methods introduced below, in Section 13.9. Limitations include the need to read multiple copies of data objects in order to ensure that the quorum overlap rule is satisfied despite failures, which makes read operations costly. A second limitation is the extensive use of 2PC, itself a costly protocol, when doing *UPDATE* operations. Even a modest application may issue large numbers of *READ* and *UPDATE* requests, leading to a tremendous volume of I/O. This is in contrast with dynamic membership solutions that will turn out to be extremely sparing in I/O, permitting completely local *READ* operations, *UPDATE* operations that cost as little as one message per replica, and yet able to guarantee very strong consistency properties. Perhaps for these reasons, quorum data management has seen relatively little use in commercial products and systems.

There is one setting in which quorum data management is found to be less costly: transactional replication schemes, typically as part of a replicated database. In these settings, database concurrency control eliminates the concerns raised earlier in regard to livelock or thrashing, and the overhead of the 2PC protocol can be amortized into a single 2PC protocol that executes at the end of the transaction. Moreover, *READ* operations can sometimes "cheat" in transactional settings, accessing a local copy and later confirming that the local copy was a valid one as part of the first phase of the 2PC protocol that terminates the transaction. Such a read can be understood as using a form of optimism, similar to that of an optimistic concurrency control scheme. The ability to abort thus makes possible significant optimizations in the solution.

On the other hand, few transactional systems have incorporated quorum replication. If one discusses the option with database companies, the message that emerges is clear: transactional replication is perceived as being extremely costly, and 2PC represents a huge burden when compared to transactions that run entirely locally on a single, non-replicated database. Transaction rates are approaching 10,000 per second for top of the line commercial database products on non-replicated high performance computers; rates of 100 per second would be impressive for a replicated transactional product. The two orders of magnitude performance loss is more than the commercial community can readily accept, even if it confers increased product availability. We will return to this point in Chapter 21.

13.7 Replicated Data with Non-Benign Failure Modes

The discussion of the previous sections assumed a crash-failure model that is approximated in most distributed systems, but may sometimes represent a risky simplification. Consider a situation in which the actions of a computing system have critical implications, such as the software responsible for adjusting the position of an aircraft wing in flight, or for opening the cargo-door of the Space Shuttle. In settings like these, the designer may hesitate to simply assume that the only failures that will occur will be benign ones.

There has been considerable work on protocols for coordinating actions under extremely pessimistic failure models, centering on what is called the Byzantine Generals problem, which explores a type of agreement protocol under the assumption that failures can produce arbitrarily incorrect behavior, but that the *number* of failures is known to be bounded. Although this assumption may seem "more realistic" than the assumption that processes fail by clean crashes, the model also includes a second type of assumption that some might view as unrealistically benign: it assumes that the processors participating in a system share perfectly synchronized clocks, permitting them to exchange messages in "rounds" that are triggered by the clocks (for example, once every second). Moreover, the model assumes that the latencies associated with message exchange between correct processors is accurately known.

Thus, the model permits failures of unlimited severity, but at the same time assumes that the *number* of failures is limited, and that operational processes share a very simple computing environment. Notice in particular that the round model would only be realistic for a very small class of modern parallel computers and is remote from the situation on distributed computing networks. The usual reasoning is that by endowing the operational computers with "extra power" (in the form of synchronized rounds), we can only make their task easier. Thus, understanding the minimum cost for solving a problem in this model will certainly teach us something about the minimum cost of overcoming failures in real-world settings.

The Byzantine Generals problem itself is as follows [Lyn96]. Suppose that an army has laid siege to a city and has the force to prevail in an overwhelming attack. However, if divided the army might lose the battle. Moreover, the commanding generals suspect that there are traitors in their midst. Under what conditions can the loyal generals coordinate their action so as to either attack in unison, or not attack at all? The assumption is that the generals start the protocol with individual opinions on the best strategy: to attack or to continue the siege. They exchange messages to execute the protocol, and if they "decide" to attack during the *i'th* round of communication, they will all attack at the start of round *i+1*. A traitorous general can send out any messages it likes and can lie about its own state, but can never forge the message of a loyal general. Finally, to avoid trivial solutions, it is required that if all the loyal generals favor attacking, an attack will result, and that if all favor maintaining the siege, no attack will occur.

To see why this is difficult, consider a simple case of the problem in which three generals surround the city. Assume that two are loyal, but that one favors attack and the other prefers to hold back. The third general is a traitor. Moreover, assume that it is known that there is at most one traitor. If the loyal generals exchange their "votes", they will both see a tie: one vote for attack, one opposed. Now suppose that the traitor sends an attack message to one general and tells the other to hold back. The loyal generals now see inconsistent states: one is likely to attack while the other holds back. The forces divided, they would be defeated in battle. The Byzantine Generals problem is thus seen to be impossible for $t=1$ and $n=3$.

With four generals and at most one failure, the problem is solvable, but not trivially so. Assume that two loyal generals favor attack, the third retreat, and the fourth is a traitor, and again that it is known that there is at most one traitor. The generals exchange messages, and the traitor sends retreat to one an attack to two others. One loyal general will now have a tied vote: two votes to attack, two to retreat. The other two generals will see three votes for attack, and one for retreat. A second round of communication will clearly be needed before this protocol can terminate! Accordingly, we now imagine a second round in which the generals circulate messages concerning their state in the first round. Two loyal generals will start this round knowing that it is "safe to attack:" on the basis of the messages received in the first round, they can deduce that even with the traitor's vote, the majority of loyal generals favored an attack. The remaining loyal general simply sends out a message that it is still undecided. At the end of this round, all the loyal generals will have one "undecided" vote, two votes that "it is safe to attack", and one message from the traitor. Clearly, no matter what the traitor votes during the second round, all three loyal generals can deduce that it is safe to attack. Thus, with four generals and at most one traitor, the protocol terminates after 2 rounds.

Using this model one can prove what are called lower-bounds and upper-bounds on the Byzantine Agreement problem. A lower bound would be a limit to the quality of a possible solution to the problem. For example, one can prove that any solution to the problem capable of overcoming *t* traitors requires a minimum of $3t+1$ participants (hence: $2t+1$ or more loyal generals). The intuition into such a bound is fairly clear: the loyal generals must somehow be able to deduce a common strategy even with *t* participants whose votes cannot be trusted. Within the remainder there needs to be a way to identify a majority decision. However, it is surprisingly difficult to prove that this must be the case. For our purposes in the present textbook, such a proof would represent a digression and hence is omitted, but interested readers are referred to the excellent treatment in [Merxx]. Another example of a lower bound concerns the minimum number of messages required to solve the problem: no protocol can overcome *t* faults with fewer than t+1 rounds of message exchange, and hence $O(t*n^2)$ messages, where *n* is the number of participating processes.

In practical terms, these represent costly findings: recall that our 2PC protocol is capable of solving a problem much like Byzantine agreement in two rounds of message exchange requiring only $3n$ messages, albeit for a simpler failure model. Moreover, the quorum methods permit data to be replicated using as few as $t+1$ copies to overcome *t* failures. And, we will be looking at even cheaper replication schemes below, albeit with slightly weaker guarantees. Thus, a Byzantine protocol is genuinely costly, and the best solutions are also fairly complex.

An upper bound on the problem would be a demonstration of a protocol that actually solves Byzantine agreement and an analysis of its complexity (number of rounds of communication required or messages required). Such a demonstration is an upper bound because it rules out the need for a more costly protocol to achieve the same objectives. Clearly, one hopes for upper bounds that are as close as possible to the lower bounds, but unfortunately no such protocols have been found for the Byzantine agreement problem. The simple protocol illustrated above can easily be generalized into a solution for *t* failures that achieves the lower bound for rounds of message exchange, although not for numbers of messages required.

Suppose that we wanted to use Byzantine Agreement to solve a static data replication problem in a very critical or hostile setting. To do so, it would be necessary that the setting somehow correspond to the setup of the Byzantine agreement problem itself. For example, one could imagine using Byzantine agreement to control an aircraft wing or the Space Shuttle cargo hold door by designing hardware that carries out voting through some form of physical process. The hardware would need to implement the mechanisms needed to write software that executes in rounds, and the programs would need to be carefully analyzed to be sure that when operational, all the computing they do in each round can be completed before that round terminates.

On the other hand, one would not want to use a Byzantine agreement protocol in a system where at the end of the protocol, some single program will take the output of the protocol and perform a critical action. In that sort of a setting (unfortunately, far more typical of "real" computer systems), all we will have done is to transfer complete trust in the set of servers within which the agreement protocol runs into a complete trust in the single program that carries out their decision.

The practical use of Byzantine agreement raises another concern: the timing assumptions built into the model are not realizable in most computing environments. While it is certainly possible to build a system with closely synchronized clocks and to approximate the synchronous rounds used in the model, the pragmatic reality is that few existing computer systems offer such a feature. Software clock synchronization, on the other hand, is subject to intrinsic limitations of its own, and for this reason is a poor alternative to the real thing. Moreover, the assumption that message exchanges can be completed within known, bounded latency is very hard to satisfy in general purpose computing environments.

Continuing in this vein, one could also question the extreme pessimism of the failure model. In a Byzantine setting the traitor can act as an adversary, seeking to force the correct processes to malfunction. For a worst-case analysis this makes a good deal of sense. But having understood the worst case, one can also ask whether real-world systems should be designed to routinely assume such a pessimistic view of the behavior of system components. After all, if one is this negative, shouldn't the hardware itself also be suspected of potential misbehavior, and the compiler, and the various prebuilt system components that implement message passing? In designing a security subsystem or implementing a firewall, such an analysis makes a lot of sense. But when designing a system that merely seeks to maintain availability despite failures, and is not expected to come under active and coordinated attack, an extremely pessimistic model would be both unwieldy and costly.

From these considerations, one sees that a Byzantine computing model may be applicable to certain types of special-purpose hardware, but will rarely be directly applicable to more general distributed computing environments where we might raise a reliability goal. As an aside, it should be noted that Rabin has introduced a set of probabilistic Byzantine protocols that are extremely efficient, but that accept a small risk of error (the risk diminishes exponentially with the number of rounds of agreement executed) [Rab83]. Developers who seek to implement Byzantine-based solutions to critical problems would be wise to consider using these elegant and efficient protocols.

13.8 Reliability in Asynchronous Environments

At the other side of the spectrum is what we call the *asynchronous* computing model, in which a set of processes cooperate by exchanging messages over communication links that are arbitrarily slow and balky. The assumption here is that the messages sent on the links eventually get through, but that there is no meaningful way to measure progress except by the reception of messages. Clearly such a model is overly pessimistic, but in a way that is different from the pessimism of the Byzantine model, which extended primarily to failures: here we are pessimistic about our ability to measure time or to predict the amount of time actions will take. A message that arrives after a century of delay would be processed no differently than a message received within milliseconds of being transmitted. At the same time, this model assumes that processes fail by crashing, taking no incorrect actions and simply halting silently.

One might wonder why the asynchronous system completely eliminates any physical notion of time. We have seen that real distributed computing systems lack ways to closely synchronize clocks and are unable to distinguish network partitioning failures from processor failures, so that there is a sense in which the asynchronous model isn't as unrealistic as it may initially appear. Real systems do have clocks and use these to establish timeouts, but generally lack a way to ensure that these timeouts will be "accurate", as we saw when we discussed RPC protocols and the associated reliability issues in Chapter 4. Indeed, if an asynchronous model can be criticized as specifically unrealistic, this is primarily in its assumption of reliable communication links: real systems tend to have limited memory resources, and a reliable communication link for a network subject to extended partitioning failures will require unlimited spooling of the messages sent. This represents an impractical design point, hence a better model would state that when a process is *reachable* messages will be exchanged reliably with it, but that if it becomes *inaccessible* messages to it will be lost and its state, faulty or operational, cannot be accurately determined. In Italy, Babaoglu and his colleagues are studying such a model, but this is recent work and the full implications of this design point are not yet fully understood [BDGB94]. Other researchers, such as Cristian, are looking at models that are partially asynchronous: they have time bounds, but the bounds are large compared to typical message passing latencies [Cri96]. Again, it is too early to say whether or not this model represents a good choice for research on realistic distributed systems.

Within the purely asynchronous model, a classical result limits what we can hope to accomplish. In 1985, Fischer, Lynch and Patterson proved that the asynchronous consensus problem (similar to the Byzantine agreement problem, but posed in an asynchronous setting) is impossible if even a single process can fail [FLP85]. Their proof revolves around the use of type of message scheduler that delays the progress of a consensus protocol, and holds regardless of the way that the protocol itself works. Basically, they demonstrate that any protocol that is guaranteed to only produce correct outcomes in an asynchronous system can be indefinitely delayed by a complex pattern of network partitioning failures. More recent work has extended this result to some of the communication protocols we will discuss in the remainder of this Chapter [CHTC96, Ric96].

The FLP proof is short but quite sophisticated, and it is common for practitioners to conclude that it does not correspond to any scenario that would be expected to arise in a real distributed system. For example, recall that 3PC is unable to make progress when failure detection is unreliable because of message loss or delays in the network. The FLP result predicts that if a protocol such as 3PC is capable of solving the consensus problem, can be prevented from terminating. However, if one studies the FLP proof, it turns out that the type of partitioning failure exploited by the proof is at least superficially very remote from the pattern of crashes and network partitioning that forces the 3PC to block.

Thus, it is a bit facile to say that FLP predicts that 3PC will block in this specific way, because the proof constructs a scenario that on its face seems to have relatively little to do with the one that causes problems in a protocol like 3PC. At the very least, one would be expected to relate the FLP scheduling pattern to the situation when 3PC blocks, and this author is not aware of any research which has made this connection concrete. Indeed, it is not entirely clear that 3PC *could* be used to solve the consensus problem: perhaps the latter is actually a harder problem, in which case the inability to solve consensus might not imply that 3PC cannot be solved in asynchronous systems.

As a matter of fact, although it is obvious that 3PC cannot be solved when the network is partitioned, if one studies the model used in FLP carefully one discovers that network partitioning is not actually a failure model admitted by this work: the FLP result assumes that every message sent will eventually be received, in FIFO order. Thus FLP essentially requires that every partition eventually be fixed, and that every message eventually get through. The tendency of 3PC to block during partitions, which concerned us above, is not captured by FLP because FLP is willing to wait until such a partition is repaired (and implicitly assumes that it will be), while we wanted 3PC to make progress even while the partition is present (whether or not it will eventually be repaired).

To be more precise, FLP tells us that any asynchronous consensus decision can be *indefinitely delayed*, not merely delayed until a problematic communication link is fixed. Moreover, it says that this is true even if every message sent in the system eventually reaches its destination. During this period of delay the processes may thus be quite active. Finally, and in some sense most surprising of all, the proof doesn’t require that any process fail at all: it is entirely based on a pattern of message delays. Thus, FLP not only predicts that we would be unable to develop a 3PC protocol that can guarantee progress despite failures, but in fact that there is no 3PC protocol that can terminate at all, even if no failures actually occur and the network is merely subject to unlimited numbers of network partitioning events. Above, we convinced ourselves that 3PC would need to block (wait) in a single situation; FLP tells us that if a protocol such as 3PC can be used to solve the consensus, then there is a sequence of communication failures that would it from reaching a commit or abort point regardless of how long it executes!

The Asynchronous Computing Model

Although we refer to our model as the "asynchronous one", it is in fact more constrained. In the asynchronous model, as used by distributed systems theoreticians, processes communicate entirely by message passing and there is no notion of time. Message passing is reliable but individual messages can be delayed indefinitely, and there is no meaningful notion of failure except that of a process that crashes, taking no further actions, or that violates its protocol by failing to send a message or discarding a received message. Even these two forms of communication failure are frequently ruled out.

The form of asynchronous computing environment used in this chapter, in contrast, is intended to be "realistic". This implies that there are in fact clocks on the processors and expectations regarding typical round-trip latencies for messages. Such temporal data can be used to define a notion of reachability, or to trigger a failure detection mechanism. The detected failure may not be attributable to a specific component (in particular, it will be impossible to know if a process failed, or just the link to it), but the fact that some sort of problem has occurred will be detected, perhaps very rapidly. Moreover, in practice, the frequency with which failures are erroneously suspected can be kept low.

Jointly, these properties make the asynchronous model used in this textbook "different" than the one used in most theoretical work. And this is a good thing, too: in the fully asynchronous model, it is known that the group membership problem cannot be solved, in the sense that any protocol capable of solving the problem may encounter situations in which it cannot make progress. In contrast, these problems are always solvable in asynchronous environments that satisfy sufficient constraints on the frequency of true or incorrectly detected failures and on the quality of communication.

To see that 3PC solves consensus, we should be able to show how to map one problem to the other, and back. For example, suppose that the inputs to the participants in a 3PC protocol are used to determine their vote, for or against commit, and that we pick one of the processes to run the protocol. Superficially, it may seem that this is a mapping from 3PC to consensus. But recall that consensus of the type considered by FLP is concerned with protocols that tolerate a single failure, which would presumably include the process that starts the protocol. Moreover, although we didn't get into this issue, consensus has a non-triviality requirement, which is that if all the inputs are '1' the decision will be '1', and if all the inputs are '0' the decision should be '0'. As stated, our mapping of 3PC to consensus might not satisfy non-triviality while also overcoming a single failure. This author is not aware of a detailed treatment of this issue. Thus, while it would not be surprising to find that 3PC is equivalent to consensus, neither is it obvious that the correspondence is an exact one.

But assume that 3PC is in fact equivalent to consensus. In a *theoretical* sense, FLP would represent a very strong limitation on 3PC. In a *practical* sense, though, it is unclear whether it has direct relevance to developers of reliable distributed software. Above, we commented that even the scenario that causes 2PC to block is extremely unlikely unless the coordinator is also a participant; thus 2PC (or 3PC when the coordinator actually is a participant) would seem to be an adequate protocol for most real systems. Perhaps we are saved from trying to develop some other very strange protocol to evade this limitation: FLP tells us that any such protocol will sometimes block. But once 2PC or 3PC has blocked, one could argue that it is of little practical consequence whether this was provoked by a complex sequence of network partitioning failures or by something simple and "blunt" like the simultaneous crash of a majority of the computers in the network. Indeed, we would consider that 3PC has failed to achieve its objectives as soon as the first partitioning failure occurs and it ceases to make *continuous* progress. Yet the FLP result, in some sense, hasn't even "kicked in" at this point: it relates to *ultimate* progress. In the FLP work, the issue of a protocol being blocked is not really modeled in the formalism at all, except in the sense that such a protocol has not yet reached a decision state.

We thus see that although FLP tells us that the asynchronous consensus problem cannot *always* be solved, it says nothing at all about when problems such as this actually *can* be solved. As we will see momentarily, more recent work answers this question for asynchronous consensus. However, unlike an impossibility result, to apply this new result one would need to be able to relate a given execution model to the asynchronous one, and a given problem to consensus.

FLP is frequently misunderstood having proved the impossibility of building fault-tolerant distributed software for realistic environments. This is not the case at all! FLP doesn't say that one cannot build a consensus protocol tolerant of one failure, or of many failures, but simply that if one does build such a protocol, and then runs it in a system with no notion of global time whatsoever, and no "timeouts", there will be a pattern of message delays that prevents it from terminating. The pattern in question may be extremely improbable, meaning that one might still be able to build an asynchronous protocol that would terminate with overwhelming probability. Moreover, realistic systems have many forms of time: timeouts, loosely synchronized global clocks, and (often) a good idea of how long messages should take to reach their destinations and to be acknowledged. This sort of information allows real systems to "evade" the limitations imposed by FLP, or at least creates a runtime environment that differs in fundamental ways from the FLP-style of asynchronous environment.

This brings us to the more recent work in the area, which presents a precise characterization of the conditions under which a consensus protocol can terminate in an asynchronous environment. Chandra and Toueg have shown how the consensus problem can be expressed using what they call "weak failure detectors", which are a mechanism for detecting that a process has failed without necessarily doing so accurately [CT91, CHT92]. A weak failure detector can make mistakes and change its mind; its behavior is similar to what might result by setting some arbitrary timeout, declaring a process faulty if no communication is received from it during the timeout period, and then declaring that it is actually operational after all if a message subsequently turns up (the communication channels are still assumed to be reliable and FIFO). Using this model, Chandra and Toueg prove that consensus can be solved provided that a period of execution arises during which all genuinely faulty processes are suspected as faulty, and during which at least one operational process is never suspected as faulty by any other operational process. One can think of this as a constraint on the quality of the communication channels and the timeout period: if communication works well enough, and timeouts are accurate enough, for a long enough period of time, a consensus decision can be reached. Interested readers should also look at [BDM95, FKMBD95, GS96, Ric96]. Two very recent papers in the area are [BBD96, Nei96].

Impossibility of Computing to Work

The following tongue-in-cheek story illustrates the sense in which a problem such as distributed consensus can be "impossible to solve." Suppose that you were discussing commuting to work with a colleague, who comments that because she owns two cars, she is able to reliably commute to work. In the rare mornings when one car won't start, she simply takes the other, and gets the non-functioning one repaired if it is still balky when the weekend comes around.

In a formal sense, you could argue that your colleague may be *lucky*, but is certainly not accurate in claiming that she can "reliably" commute to work. After all, both cars might fail at the same time. Indeed, even if neither car fails, if she uses a "fault-tolerant" algorithm, a clever adversary might easily prevent her from ever leaving her house.

This adversary would simply prevent the car from starting during a period that lasts a little longer than your colleague is willing to crank the motor before giving up and trying the other car. From her point of view, both cars will appear to have broken down. The adversary, however, can maintain that neither car was actually faulty, and that had she merely cranked the engine longer, either car would have started. Indeed, the adversary can argue that had she *not* tried to use a fault-tolerant algorithm, she could have started either car by merely not giving up on it "just before it was ready to start."

Obviously, the argument used to demonstrate the impossibility of solving problems in the general asynchronous model is quite a bit more sophisticated than this, but it has a similar flavor in a deeper sense. The adversary keeps delaying a message from being delivered just long enough to convince the protocol to "reconfigure itself" and look for a way of reaching concensus without waiting for the process that sent the message. In effect, the protocol gives up on one car and tries to start the other one. Eventually, this leads back to a state where some critical message will trigger a consensus decision ("start the car"). But the adversary now allows the old message through and delays messages from this new "critical" source.

What is odd about the model is that protocols are not supposed to be bothered by arbitrarily long delays in message deliver. In practice, if a message is delayed by a "real" network for longer than a small amount of time, the message is considered to have been lost and the link, or its sender, is treated as having crashed. Thus, the asynchronous model focuses on a type of behavior that is not actually typical of real distributed protocols.

For this reason, readers with an interest in theory are encouraged to look to the substantial literature on the theory of distributed computing, but to do so from a reasonably sophisticated perspective. The theoretical community has shed important light on some very fundamental issues, but the models used are not always realistic ones. One learns from these results, but must also be careful to appreciate the relevance of the results to the more realistic needs of practical systems.

What Chandra and Toueg have done has general implications for the developers of other forms of distributed systems that seek to guarantee reliability. We learn from this result that to guarantee progress, the developer may need to guarantee a higher quality of communication than in the classical asynchronous model, a degree of clock synchronization (lacking in the model), or some form of accurate failure detection. With any of these, the FLP limitations can be evaded (they no longer hold). In general, it will not be possible to say "my protocol always terminates" without also saying "when such and such a condition holds" on the communication channels, the timeouts used, or other properties of the environment.

This said, the FLP result does create a quandary for practitioners who hope to be rigorous about the reliability properties of their algorithms, by making it difficult to talk in rigorous terms about what protocols for asynchronous distributed systems actually guarantee. We would like to be able to talk about one protocol being more tolerant of failures than another, but now we see that such statements will apparently need to be made about protocols that one can only guarantee fault-tolerance in a conditional way, and where the conditions may not be simple to express or to validate.

What seems to have happened here is that we lack an appropriate notion of what it means for a protocol to be "live" in an asynchronous setting. The FLP notion of liveness is rigorously defined and not achievable, but in any case seems not to get at the more relative notion of liveness that we seek in developing a "non-blocking commit protocol". As it happens, even this more relative form of liveness is not always achievable, and this coincidence has sometimes lead practitioners and even theoreticians to conclude that the forms of liveness are "the same", since neither is always possible. This subtle but very important point has yet to be treated adequately by the theoretical community. We need a model within which we can talk about 3PC making progress "under conditions when 2PC would not do so" without getting snarled in the impossibility of guaranteeing progress for all possible runs in the asynchronous model.

Returning to our data replication problem, these theoretical results do have some practical implications. In particular, they suggest that there may not be much more that can be accomplished in a static computing model. The quorum methods give us a way to overcome failures or damage to limited numbers of data objects within a set of replicas; although expensive, such methods clearly work. While they would not work with a very serious type of failure in which processes behave maliciously, the Byzantine agreement and consensus literature suggests that one cannot always solve this problem in an asynchronous model, and the synchronous model is sufficiently specialized as to be largely inapplicable to standard distributed computing systems.

Our best hope, in light of these limitations, will be to focus on the poor performance of the style of replication algorithm arrived at above. Perhaps a less costly algorithm would represent a viable option for introducing tolerance to at least a useful class of failures in realistic distributed environments. Moreover, although the FLP result tells us that for certain categories of objectives that are as strong as consensus, availability must always be limited, the result does not speak directly to the sorts of tradeoffs between availability and cost seen in 2PC and 3PC. Perhaps we can talk about optimal progress and identify the protocol structures that result in the best possible availability without sacrificing consistency, even if we must accept that our protocols will (at least theoretically) remain exposed to scenarios in which they are unable to make progress.

13.9 The Dynamic Group Membership Problem

If we desire to move beyond the limitations of the protocols presented above, it will be necessary to identify some point of leverage offering a practical form of freedom that can be exploited to reduce the costs of reliability, measured in terms of messages exchanged to accomplish a goal like updating replicated data, or rounds of messages exchanged for such a purpose. What options are available to us?

Focusing on the model itself, it makes sense to examine the static nature of the membership model. In light of the fact that real programs are started, run for a while, and then terminate, one might ask if a static membership model really makes sense in a distributed setting. The most apparent response is that the hardware resources on which the programs execute will normally be static, but also that such a model actually may not be *required* for many applications. Moreover, the hardware base used in many systems does change over time, albeit slowly and fairly infrequently. If one assumes that certain platforms are always started with the same set of programs running on them, a static membership model is closely matched to the environment, but if one looks at the same system over a long enough time scale to encompass periods of hardware reconfiguration and upgrades, or looks at the applications themselves, the static model seems less natural.

Suppose that one grants that a given application is closely matched to the static model. Even in this case, what really makes that system static is the "memory" its component programs retain of their past actions; otherwise, even though the applications tend to run on the same platforms, this is of limited importance outside of its implications for the maximum number of applications that might be operational at any time. Suppose that a computer has an attached disk on it, and some program is the manager for a database object stored on that disk. Each time this program interacts with other programs in the system, its entire past history of interactions is evidenced by the state of the database at the time of the interaction. It makes sense to model such a system as having a set of processes that may be inaccessible for periods of time, but that are basically static and predefined, *because the data saved on disk lets them prove that this was the case*. Persistence, then, is at the core of the static model.

Another example of a setting in which the resources are fundamentally static would be a system in which some set of computers control a collection of external devices. The devices will presumably not represent a rapidly changing resource, hence it makes a lot of sense to model the programs that manage them as a static group of processes. Here, the external world is persistent, to the degree that it retains memory of the actions taken in the past. Absent such memory, though, one might just as well call this a dynamic system model.

In fact, there are a great many types of programs that are launched and execute with no particular memory of the past activities on the computer used to run them. These programs may tend to run repeatedly on the same set of platforms, and one certainly could apply a static system membership model to them. Yet in a deeper sense they are oblivious to the place at which they run. Such a perspective gives rise to what we call the *dynamic membership model*, in which the set of processes that compose the system at a given time varies. New processes are started and *join* the system, while active processes *leave* the system when they terminate, fail, or simply chose to disconnect themselves. While such a system may also have a static set of processes associated with it, such as a set of servers shared by the dynamically changing population of transient processes, it makes sense to adopt a dynamic membership model to deal with this collection of transient participants.

Even the process associated with a static resource such as a database can potentially be modeled as a dynamic component of the system to which it belongs. In this perspective there is a need for the system to behave in a manner consistent with the externally persistent actions taken by previous members, which may be recorded in databases or have had external effects on the environment that must be respected. On the other hand, the process that manages such a resource is treated as a dynamic component of the system that must be launched when the database server is booted and that terminates when the server crashes. Its state is volatile: the database persists, but not the memory contents of the server at the time it shuts down. Thus, even if one believes that a static system model is physically reasonable, it may still be acceptable to model this with a dynamic model, provided that attention is paid to externally persistent actions initiated by system members. The advantage of doing so is that the model will not be tied to the system configuration, so hardware upgrades and other configuration changes do not necessarily have to be treated outside of the model in which reliability was presented. To the degree that systems need to be self-managing, such freedom can be extremely useful.

The dynamic model poses new challenges, not the least of which is to develop mechanisms for tracking the current membership of the system, since this will no longer be a "well known" quantity. It also offers significant potential for improvements in our protocols, however, particularly if processes that leave the system are considered to have terminated. The advantage, as we will see shortly, is that agreement *within the operational group of system members* is an easier problem to solve than the sorts of consensus and replication problems examined above for a static set of possibility inaccessible members.

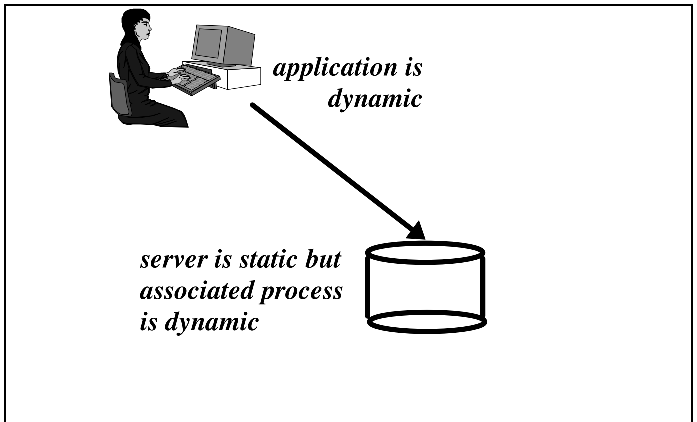


Figure 13-10: Trading system

In the remainder of this section, we explore some of the fundamental problems raised by dynamic membership. The treatment distinguishes two cases: those in which actions must be *dynamically uniform*, meaning that any action taken by a process must be consistent with subsequent actions by the operational part of the system [ADKM92a, SS93, MBRS94], and those in which dynamic uniformity is not required, meaning that the operational part of the system is taken to "define" the system, and the states and actions of processes that subsequently fail can be discarded. Dynamic uniformity captures the case of a process that, although executed within a dynamic system model, may perform externally visible actions. In our database server example, the server process can perform such actions, by modifying the contents of the database it manages.

Dynamic uniformity may seem similar to the property achieved by a commit protocol, but there are important differences. In a commit protocol, we require that *if any process commits some action, all processes will commit it*. This obligation holds within a statically defined set of processes: a process that fails may later recover, so the commit problem involves an indefinite obligation with regard to a set of participants that is specified at the outset. In fact, the obligation even holds if a process reaches a decision and then crashes without telling any other process what that decision was.

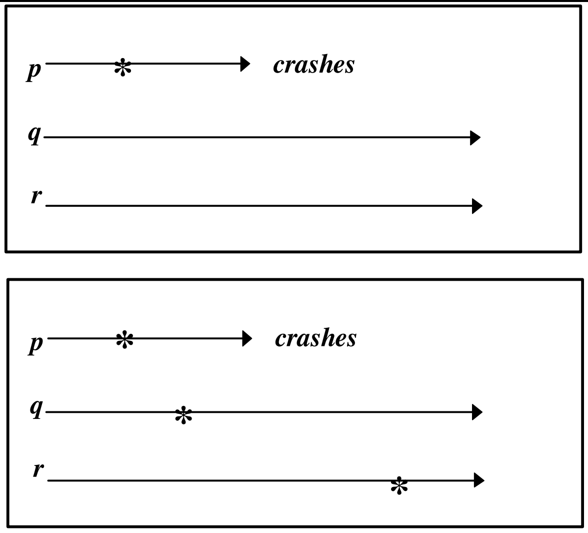


Figure 13-11: Non-uniform (above) and dynamically uniform (below) executions. In the non-uniform case, process p takes an action (denoted "") that no other process takes. This is acceptable because p crashes. In the dynamically uniform case, all must take the action if any does so. This implies that before p took the action, its intention to do so must have been communicated to the remainder of the system.*

Dynamic uniformity says that *if any process performs some action, all processes that remain operational will also perform it* (Figure 13-14). That is, the obligation to perform an action begins as soon as any process in the system performs that action, and then extends to processes that remain operational, but not to processes that fail. This is because, in a dynamic membership model, we adopt the view that if a process is excluded from the membership it has "failed", and that a process that fails never rejoins the system. In practice, a process that fails may reconnect as a "new" process, but when we consider join events, we normally say that a process that joins the system is only required to perform those actions that are initiated after its join event (in a causal sense). The idea is that a process that joins the system, whether it is new or some sort of old process that is reconnecting after a long delay, learns the state of the system as it first connects, and then is guaranteed to perform all the dynamically uniform actions that any process performs, until such time as it fails or (equivalently) is excluded from the system membership set.

Real-world applications often have a need for dynamic uniformity, but this is not invariably the case. Consider a desktop computing system used in a financial application. The role of the application program might be to display financial information of importance to the trader in a timely manner, to compute financial metrics such as theoretical prices for specified trading instruments, and to communicate new trades to the exchanges at which they will be carried out. Such systems are common and represent a leading-edge example of distributed computing used in support of day-to-day commerce. These systems are also quite critical to their users, so they are a good example of an application in which reliability is important.

Is such an application best viewed as static, or dynamic, and must the actions taken be dynamically uniform ones? If we assume that the servers providing trading information and updates are continuously operational, there is little need for the application on the trader's desktop to retain significant state. It may make sense to cache frequently used financial parameters or to keep a local log of submitted trading requests as part of the application, but such a system would be unlikely to provide services when disconnected from the network and hence is unlikely to have any significant form of local state information that is not also stored on the databases and other servers that maintain the firm's accounts.

Thus, while there are some forms of information that may be stored on disk, it is reasonable to view this as an example of a dynamic computing application. The trader sits down at his or her desk and begins to work, launching one or more programs that connect to the back-end computing system and start providing data and support services. At the end of the day, or when no longer needed, the programs shut down. And in the event of a network communication problem that disconnects the desktop system from the system servers, the trader is more likely to move to a different station than to try and continue using it while disconnected. Rapidly, any "state" that was associated with the disconnected station will become stale, and when the station is reconnected, it may make more sense to restart the application programs as if from a failure rather than try and reintegrate them into the system so as to exploit local information not available on the servers.

This type of dynamicism is common in client-server architectures. Recall the discussion of caching that arose when we considered web proxies in Part II of the textbook. The cached documents reside within a dynamically changing set of processes that surround a static set of servers on which the originals reside. The same could be said for the file systems we examined in Part I, which cache buffers or whole files.

It is not often required that actions taken by a user-interface program satisfy a dynamic uniformity property. Dynamic uniformity arises when a single action is logically taken at multiple places in a network, for example when a replicated database is updated, or when a set of traders are all asked to take some action and it is important that *all* of them do so. In our trading system, one could imagine the server sending information to multiple client systems, or a client entering a trading strategy that impacts multiple other clients and servers. More likely, however, the information sent by servers to the client systems will be pricing updates and similar data, and if the client system and server system fail just as the information is sent, there may not be any great concern that the same message be delivered to every other client in the network. Similarly, if the client sends a request for information to the server but fails at the instant the request is sent, it may not be terribly important if the request is seen by the server or not. These are examples of non-dynamically-uniform actions.

In contrast, although the process that handles the database can be viewed as a dynamic member of the system, its actions do need to satisfy dynamically uniform. If a set of such servers maintain replicated data or have any sort of consistency property that spans them as a set, it will be important that if an action is taken by one process it will also be taken by the others.

Thus, our example can be modeled as a dynamic system. Some parts of the system may require dynamic uniformity, but others can manage without this property. As we will see shortly, this choice has significant performance implications, hence we will work towards a family of protocols in which dynamic uniformity is available on an operation by operation basis, depending on the needs of the application and the costs they are prepared to pay for this guarantee.

13.10 The Group Membership Problem

The role of a group membership service (GMS) is to maintain the membership of a distributed system on behalf of the processes that compose it [BJ87b, Cri91b, MPS91, MSMA91, RB91, ADKM92b, Gol92, Ric92, Ric93, RVR93, Mal94, MMA94, Aga94, BDGB94, Rei94b BG95, CS95, ACBM95, BDM95, FKMBD95, CHTC96, GS96]. As described above, processes join and leave the system dynamically over its lifetime. We will adopt a model in which processes wishing to join do so by first contacting the GMS, which updates the list of system members and then grants the request. Once admitted to the system, a process may interact with other system members. Finally, if the process terminates, the GMS service will again update the list of system members.[9]

[9] The author's work in this area included an effort to break out the GMS problem from the context in which we use it, specifying it as a formal abstract question using temporal logic, and then using the temporal logic properties to reason about the correctness and optimality of the protocols used to implement GMS. Unfortunately, some of this work was later found to have flaws, specifically in the temporal logic formulas that were proposed as the rigorous specification of the GMS. Readers who become interested in this area should be aware that the GMS specification

The interface between the GMS and other system processes provides three operations, tabulated in Figure 13-12. The *join* operation is involved by a process that wishes to become a system member. The *monitor* operation is used by a process to register its interest in the status of some other process; should that process be dropped from the membership, a callback will occur to notify it of the event. Such a callback is treated as the equivalent of a failure notification in the failstop computing model: the process is considered to have crashed, all communication links with it are severed, and messages subsequently received from it are rejected. Finally, the *leave* operation is used by a process that wishes to disconnect itself from the system, or by some other system component that has detected a fault and wishes to signal that a particular process has failed. We assume throughout this section that failure detections are inaccurate in the sense that they may result from partitioning of the network, but are otherwise of sufficiently good quality as to rarely exclude an operational process as faulty.

The GMS itself will need to be highly available, hence it will typically be implemented by a set of processes that cooperate to implement the GMS abstraction. Although these processes would normally reside on a statically defined set of server computers, so that they can readily be located by a process wishing to join the system, the actual composition of the group may vary over time due to failures within the GMS service itself, and one can imagine other ways of tracking down representatives (files, name services, the use of hardware broadcast to poll for a member, etc). Notice that in order to implement the GMS abstraction on behalf of the remainder of the system, a GMS server needs to solve the GMS problem

---

used in papers by Ricciardi and Birman does have some fairly serious problems. Better specifications have subsequently been proposed by Malkhi (unpublished; 1995), Babaoglu [BBD96], and Neiger [Nei96].

The essential difficulty is that the behavior of a GMS implementation depends upon future events. Suppose that a process *p* suspects that process *q* is faulty. If *p* itself remains in the system, *q* will eventually be excluded from it. But there are cases in which *p* might itself be excluded from the system, in which both *p* and *q* might be excluded, and in which the system as a whole is prevented from making progress because less than a majority of the processes that participated in the previous system view have remained operational. Unfortunately, it is not clear which of these cases will apply until *later in the execution* when the system's future has become definite. In the specification of the GMS one wants to say that "if *p* suspects the failure of *q* than *q* will eventually be excluded from the system, unless *p* is excluded from the system." However, formalizing this type of uncertain action is extremely difficult in the most popular forms of temporal logics.

A further problem was caused by what is ultimately a misunderstanding of terminology. In the theory community, one says that a problem is "impossible" if the problem is "not always solvable" in the target environment. Thus, because there can be sequences of failures for which GMS is unable to make progress, the GMS problem is an "impossible one". The theory community deals with this by defining special types of failure detectors and proving that a problem like GMS is (always) solvable given such a failure detector. They do not necessarily tackle the issues associated with implementing such a failure detector.

In our specification of GMS Ricciardi and I used a similar approach, characterizing GMS as "live provided that the frequency of failures was sufficiently low", but using english rather than temporal logic formulas to express these conditions. Subsequent readers were critical of this approach, arguing that from the formulas alone it was impossible to know precisely what were the conditions for which GMS is guaranteed to be live. Moreover, they proved that in the absense of some form of restrictive condition (some form of failure detector), the GMS problem was "impossible" in the sense that it cannot always be solved in the general asynchronous model. This is not surprising, and reflects the same limitations we encountered when looking at the multi-phase commit protocol.

In summary, then, there exists an attempt to formalize the GMS problem, and there are some known and serious problems with the proposed formalization. However, the proposed protocols are generally recognized to work correctly, and the concerns that have been raised revolve around relatively subtle theoretical issues having to do with the way the formal specification of the problem was expressed in temporal logic. At the time of this writing, there has been encouraging progress on these issues, and we are closer than ever before to knowing precisely when GMS can be solved and precisely how this problem relates to the asynchronous concensus problem. These research topics, however, seem to have no practical implications at all for the design of distributed software systems such as the ones we present here: they warn us that we will not always be able to guarantee progress, consistency, and high availability, but do not imply that we will not "usually" be successful in all of these goals.

on its own behalf. We will say that it uses a *group membership protocol*, or GMP, for this purpose. Thus, the GMP deals with the membership of a small "service," the GMS, which the remainder of the system (a potentially large set of processes) employ to track the composition of the system as a whole.

Similar to the situation for other system processes that don't comprise the GMS, the GMP problem is defined in terms of *join* and *leave* events; the latter being triggered by inability of the GMS processes to communicate with one-another. Clearly, such an environment creates the threat of a partitioning failure, in which a single GMS might split into multiple GMS sub-instances, each of which considers the other to be faulty. What should be our goals when such a partitioned scenario arises?

Suppose that our distributed system is being used in a setting such as air-traffic control. If the output of the GMS is treated as being the logical equivalent of a failure notification, one would expect the system to reconfigure itself after such notifications to restore full air traffic control support within the remaining set of processes. For example, if some component of the air traffic control system is responsible for advising controllers as to the status of sectors of the airspace (free or occupied), and the associated process fails, the air traffic system would probably restart it by launching a new status manager process.

Now, a GMS partition would be the likely consequence of a network partition, raising the prospect that two air traffic sector services could find themselves simultaneously active, both trying to control access to the same portions of the airspace, and neither aware of the other! Such an inconsistency would have disastrous consequences. While the partitioning of the GMS might be permissible, it is clear that at most one of the resulting GMS components should be permitted to initiate new actions.

From this example we see that although one might want to allow a system to remain operational during partitionings of the GMS, one also needs a way to pick one component of the overall system as the "primary" one, within which authoritative decisions can be taken on behalf of the system as a whole [Ric93, Mal94]. Non-primary components might report information on the basis of their state as of the time when a partitioning occurred, but would not permit potentially conflicting actions (like routing a plane into an apparently free sector of airspace) to be initiated. Such an approach clearly generalizes: one can imagine a system in which some applications would be considered "primary" within a component that is considered non-primary for other purposes. Moreover, there may be classes of actions that are safe even within a non-primary component; an example would be the reallocation of air traffic within sectors of the air traffic service already owned by the partition at the time the network failed. But it is clear that any GMP solution should at least track primaryness so that actions can be appropriately limited.[10]

The key properties of the primary component of the GMS will be that its membership should overlap with the membership of a previous primary component of the GMS, and that there should only be one primary component of the GMS within any partitioning of the GMS as a whole. We will then say that the primaryness of a partition of the distributed system as a whole is determined by the primaryness of the GMP component to which its processes are connected.

At the outset of this chapter we discussed notions of time in distributed settings. In defining the primary component of a partitioned GMS we used temporal terms without making it clear exactly what

[10] Later, we will present some work by Idit Keidar and Danny Dolev [KD95], in which a partioned distributed system is able to make progress despite the fact that no primary component is ever present. However, in this work a static system membership model is used, and no action can be taken until a majority of the processes in the system as a whole are known to be "aware" of the action. This is a costly constraint and seems likely to limit the applicability of the approach.

form of time was intended. In what follows, we have *logical* time in mind. In particular, suppose that process *p* is a member of the primary component of the GMS, but then suddenly becomes partitioned away from the remainder of the GMS, executing for an arbitrarily long period of time without sending or receiving any additional messages, and finally shutting down. By the discussion up to now, it is clear that we would want the GMS to reconfigure itself to exclude *p*, if possible, forming a new primary GMS component that can permit further progress in the system as a whole. But now the question arises of whether *p* would be aware that this has occurred. If not, *p* might consider itself a member of the previous primary component of the GMS, and we would now have two primary components of the GMS active simultaneously.

There are two ways in which we will want to respond to this issue. The first involves a limited introduction of time into the model. Where clocks are available, it would be useful to have a mechanism whereby any process that ceases to be a member of a component of a partitioned GMS can detect this situation within a bounded period of time. For example, it would be helpful to know that within "2 seconds" of being excluded from the GMS, *p* knows that it is no longer a member of the primary component. If we assume that clocks are synchronized to a specified degree, we would ideally like to be able to compute the smallest time constant δ, such that it is meaningful to say that *p* will detect this condition within time $t+\delta$ of when it occurs.

In addition to this, we will need a way to capture the sense in which it is legal for *p* to lag the GMS in this manner, albeit for a limited period of time. Notice that because we wish to require that primary components of the GMS have overlapping membership, if we are given two different membership lists for the GMS, *a* and *b*, either $a \rightarrow b$, or $b \rightarrow a$. Thus, rather than say that there should be at most one primary component of the GMS active simultaneously, we will say that any two concurrently active membership lists for the GMS (in the sense that each is considered current by some process) should be ordered by causality. Equivalently, we could now say that there is at most a single sequence of GMS membership lists that are considered to represent the primary component of the GMS. We will use the term *view* of the GMS membership to denote the value of the membership list that holds for a given process within the GMS at a specified point in its execution.

If the GMS can experience a partitioning failure, it can also experience the *merging* of partitions [ADKM92b, Mal94, MMA94]. The GMP should therefore include a merge protocol. Finally, if all the members of the GMS fail, or if the primary partition is somehow lost, the GMP should provide for a restart from complete failure, or for identification of the primary partition when the merge of two non-primary partitions makes it possible to determine that there is no active primary partition within the system. We'll discuss this issue at some length in Chapter 15.

The protocol that we now present is based on one that was developed as part of the Isis system in 1987 [BJ87b], but was substantially extended by Ricciardi in 1991 as part of her Ph.D. dissertation [RB91, Ric92, Ric93]. A slightly more elaborate version of this protocol has been proved optimal, but we present a simpler version for clarity. The protocol has the interesting property that all GMS members see exactly the same sequence of join and leave events. The members use this property to obtain an unusually efficient protocol execution.

To avoid placing excessive trust in the correctness or fault-tolerance of the clients, our goal will be to implement a GMS for which all operations are invoked using a modified RPC protocol. Our solution should allow a process to issue requests to any member of the GMS server group with which it is able to establish contact. The protocol implemented by the group should have the property that a *join* operations are idempotent: if a joining process times out or otherwise fails to receive a reply it can reissue its request, perhaps to a different server. Having joined the system, clients that detect apparent failures merely report them to the GMS. The GMS itself will be responsible for all forms of failure notification, both for GMS members and other clients. Thus, actions that would normally be triggered by timeouts (such as reissuing an RPC or breaking a stream connection) will be triggered in our system by a GMS callback notifying the process doing the RPC or maintaining the stream that the party it is contacting has failed. Figure 13-12 summarizes this interface.

Operation	***Function***	***Failure Handling***
join(process-id, callback) returns (time,GMS-list)	***Calling process is added to membership list of system, returns logical time of the join event and a list giving the membership of the GMS service. The callback function is invoked whenever the core membership of the GMS changes.***	***Idempotent: can be reissued to any GMS process with same outcome***
leave(process-id) returns void	***Can be issued by any member of the system. GMS drops the specified process from the membership list and issues notification to all members of the system. If the process in question is really operational, it must rejoin under a new process-id***	***Idempotent. Fails only if the GMS process that was the target is dropped from the GMS membership list.***
monitor(process-id,callback) returns callback-id	***Can be issued by any member of the system, GMS registers a callback and will invoke callback(process-id) later if the designated process fails***	***Idempotent, as for leave.***

Figure 13-12: Table of GMS operations

13.10.1 Protocol used to track GMS Membership

We start by presenting the protocol used to track the core membership of the GMS service itself. These are the processes responsible for implementing the GMS abstraction, but not their clients. We assume that the processes all watch one-another using some form of network-level ping operation, detecting failures by timeout.

Both the addition of new GMS members and the deletion of apparently failed members is handled by the GMS coordinator, which is the GMS member that has been operational for the longest period of time. As we will see, although the GMS protocol permits more than one process to be added or deleted at a time, it orders all add and delete events so that this notion of “oldest” process is well defined and consistent throughout the GMS. If some process believes the GMS coordinator has failed, it treats the next highest ranked process (perhaps itself) as the new coordinator.

Our initial protocol will be such that any process suspected of having failed is subsequently *shunned* by the system members that learn of the suspected failure. Upon detection of an apparent failure, a GMS process immediately ceases to accept communication from the failed process. It also immediately sends a message to every other GMS process with which it is communicating, informing them of the apparent failure; they shun the faulty process as well. If a shunned process is actually operational, it will learn that it is being shunned when it next attempts to communicate with some GMS process that has heard of the fault, at which point it is expected to rejoin the GMS under a new process identifier. In this manner, a suspected failure can be treated as if it were a real one.

Having developed this initial protocol, we will discuss extensions that allow partitions to form and later merge, in Section 13.10.4, and than again in Chapter 15, where we present an execution model that makes use of this functionality.

Upon learning of a failure or an addition request, the GMS coordinator starts a protocol that will lead to the updating of the membership list, which is replicated among all GMS processes. The protocol requires two phases when the processes being added or deleted do not include the old GMS coordinator; a third phase is used if the coordinator has failed and a new coordinator is taking over. Any number of add operations can be combined into a single round of the protocol. A single round can also perform multiple delete operations, but here, there is a limit: at most a minority of the processes present in a given view can be dropped from the subsequent view (more precisely, a majority of the processes in a given view must acknowledge the next view; obviously, this implies that the processes in question must be alive!)

In the two-phase case, the first round of the protocol sends the list of add and delete events to the participants, including the coordinator itself, which all acknowledge receipt. The coordinator waits for as many replies as possible, but also requires a majority response from the current membership. If less than a majority of processes are reachable it waits until communication is restored before continuing. If processes have failed and only a minority are available, a special protocol described below is executed.

Unless additional failures occur at this point in the protocol, which would be very unlikely, a majority of processes acknowledge the first-round protocol. The GMS coordinator now commits the update in a second round, which also carrys with it notifications of any failures that were detected during the first round. Indeed, the second round protocol can be compacted with the first round of a new instance of the deletion protocol, if desired. The GMS members update their membership view upon reception of the second round protocol messages.

In what one hopes will be unusual conditions, it may be that a majority of the previous membership cannot be contacted because too many GMS processes have crashed. In this case, a GMS coordinator still must ensure that the failed processes did not acquiesce in a reconfiguration protocol of which it was not a part. In general, this problem may not be solvable: for example, it may be that a majority of GMS processes have crashed, and hence prior to crashing they could have admitted any number of new processes and deleted the ones now trying to run the protocol. Those new processes could now be anywhere in the system. In practice, however, this problem is often easy to solve: the GMS will most often execute within a static set of possible server hosts, and even if this set has some small degree of dynamicism, it is normally possible to track down any possible GMS server by checking some moderate number of nodes for a representative.

Both of these two cases were for the case where the coordinator did not fail. A three-phase protocol is required when the current coordinator is suspected as having failed, and some other coordinator must take over. The new coordinator starts by informing at least a majority of the GMS processes listed in the current membership that the coordinator has failed, and collecting their acknowledgements and current membership information. At the end of this first phase the new coordinator may have learned of pending add or delete events that were initiated by the prior coordinator before it was suspected as having failed. The first round protocol also has the effect of ensuring that a majority of GMS processes will start to shun the old coordinator. The second and third rounds of the protocol are exactly as for the normal case: the new coordinator proposes a new membership list, incorporating any add events of which it has learned, and all the delete events including those it learned about during the initial round of communication and that for the old coordinator. It waits for a majority to acknowledge this message and then commits it, piggybacking suspected failure information for any unresponsible processes.

Ricciardi has given a detailed proof that the above protocol results in a single, ordered sequence of process add and leave events for the GMS, and that it is immune to partitioning [Ric92]. The key to her proof is the observation that any new membership list installed successfully necessarily must be acknowledged by a majority of the previous one, and hence that any two concurrent protocols will be related by a causal path. One protocol will learn of the other, or both will learn of one another, and this is sufficient to prevent the GMS from partitioning. She shows that if the *i'th* round of the protocol starts with *n* processes in the GMS membership, an arbitrary number of processes can be added to the GMS and at most $\lfloor n/2 \rfloor$-1 processes can be excluded (this is because of the requirement that a majority of processes agree with each proposed new view). In addition, she shows that even if a steady stream of join and leave or failure events occurs, the GMS should be able to continuously output new GMS views provided that the number of failures never rises high enough to prevent majority agreement on the next view. In effect, although the protocol may be discussing the proposed *i*+2'nd view, it is still able to commit the *i*+1'st view.

13.10.2 GMS Protocol to Handle Client Add and Join Events

We now turn to the issues that arise if a GMS server is used to manage the membership of some larger number of client processes, which interact with it through the interface given earlier.

In this approach, a process wishing to join the system will locate an operational GMS member. It then issues a *join* RPC to that process. If the RPC times out the request can simply be reissued to some other member. When the join succeeds, it learns its ranking (the time at which the join took place), and also the current membership of the GMS service, which is useful in setting up subsequent monitor operations. Similarly, a process wishing to report a failure can invoke the *leave* operation in any operational GMS member. If that member fails before confirming that the operation has been successful, the caller can detect this by receiving a callback reporting the failure of the GMS member itself, and then can reissue the request.

To solve these problems, we could now develop a specialized protocol. Before doing so, however, it makes sense to ask if the GMS is not simply an instance of a service that manages replicated data on behalf of a set of clients; if so, we should instead develop the most general and efficient solutions possible for the replicated data problem, and then use them within the GMS to maintain this specific form of information. And indeed, it is very natural to adopt this point of view.

To transform the one problem into the other, we need to understand how an RPC interface to the GMS can be implemented such that the GMS would reliably offer the desired functionality to its clients, using data replication primitives internally for this purpose. Then we can focus on the data replication problem separately, and convince ourselves that the necessary primitives can be developed and can offer efficient performance.

The first problem that needs to be addressed concerns the case where a client issues a request to some representative of the GMS that fails before responding. This can be solved by ensuring that such requests are *idempotent*, meaning that the same operation can be issued repeatedly, and will repeatedly return the identical result. For example, an operation that assigns the value 3 to a variable *x* is idempotent, whereas an operation that increments *x* by adding 1 to it would not be. We can make the client join operation idempotent by having the client uniquely identify itself, and repeat the identifier each time the request must be reissued. Recall that the GMS returns the "time" of the join operation; this can be made idempotent by arranging that if a client join request is received from a client that is already listed as a system member, the time currently listed is returned and no other action is taken.

The remaining operations are all initiated by processes that belong to the system. These, too, might need to be reissued if the GMS process contacted to perform the operation fails before responding (the failure would be detected when a new GMS membership list is delivered to a process waiting for a response, and the GMS member it is waiting for is found to have been dropped from the list). It is clear that exactly the same approach can be used to solve this problem. Each request need only be uniquely identifiable, for example using the process identifier of the invoking process and some form of counter (request 17 from process *p* on host *h*).

The central issue is thus reduced to replication of data within the GMS, or within similar groups of processes. We will postpone this problem momentarily, returning below when we give a protocol for implementing replicated data within dynamically defined groups of processes.

13.10.3 GMS Notifications With Bounded Delay

If the processes within a system possess synchronized clocks, it is possible to bound the delay before a process becomes aware that it has been partitioned from the system. Consider a system in which the health of a process is monitored by the continued reception of some form of “still alive” messages received from it; if no such message is received after delay σ, any of the processes monitoring that process can report it as faulty to the GMS. (Normally, such a process would also cease to accept incoming messages from the faulty process, and would also gossip with other processes to ensure that if *p* considers *q* to have failed, than any process that receives a message from *p* will also begin to shun messages from *q*). Now, assume further that all processes which do receive a “still alive” message acknowledge it.

In this setting, *p* will become aware that it may have been partitioned from the system within a maximum delay of 2*ε+σ, where ε represents the maximum latency of the communication channels. More precisely, *p* will discover that it has been partitioned from the system 2*ε+σ time units after it last had contact with a majority of the previous primary component of the GMS. In such situations, it would be appropriate for *p* to break any channels it has to system members, and to cease taking actions on behalf of the system as a whole.

Thus, although the GMS may run its protocol to exclude *p* as early as 2*ε time units before *p* discovers that has been partitioned from the main system, there is a bound on this delay. The implication is that the new primary system component can safely break locks held by *p* or otherwise takeover actions for which *p* was responsible after 2*ε time units have elapsed.

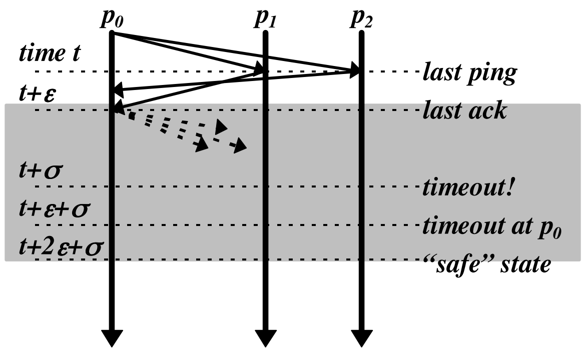


Figure 13-13: If channel delays are bounded, a process can detect that it has been partitioned from the primary component within a bounded time interval, making it safe for the primary component to take over actions from it even if where externally visible effects may be involved. Above, the gray region denotes a period during which the new primary process will be unable to take over because there is some possibility that the old primary process is still operational in a non-primary component and may still be initiating "authoritative actions". At the end of the gray period a new primary process can be appointed within the primary component. There may be a period of real-time during which no primary process was active, but there is no risk that two were simultaneously active. One can also bias a system in the other direction, so that there will always be at least one primary active provided that the rate of failures is limited.

Reasoning such as this is only possible in systems where clocks are synchronized to a known precision, and in which the delays associated with communication channels are also known. In practice, such values are rarely known with any accuracy, but coarse approximations may exist. Thus, in a system where message passing primitives provide expected latencies of a few milliseconds, one might take ε to be a much larger number, like one second or ten seconds. Although extremely conservative, such an approach would in practice be quite safe. Later we will examine real-time issues more closely, to ask how much better we can do, but it is useful to keep in mind that very coarse-grained real-time problems are often simple in distributed systems where the equivalent fine-grained real-time problems would be very difficult or provably impossible. At the same time, even a coarse-grained rule such as this one would only be safe if there was good reason to believe that the value of ε was a "safe" approximation. Some systems provide no guarantees of this sort at all, in which case incorrect behavior could result if a period of extreme overload or some other unusual condition caused the ε limit to be exceeded.

To summarize, the "core primary partition GMS" protocol must satisfy the following properties:

- *C-GMS-1: The system membership takes the form of "system views". There is an initial system view which is predetermined at the time the system starts. Subsequent views differ by the addition or deletion of processes.*
- *C-GMS-2: Only processes that request to be added to the system are added. Only processes that are suspected of failure, or that request to leave the system, are deleted.*
- *C-GMS-3: A majority of the processes in view i of the system must acquiesce in the composition of view i+1 of the system.*
- *C-GMS-4: Starting from an initial system view, subsequences of a single sequence of "system views" are reported to system members. Each system member observes such a subsequence starting with the view in which it was first added to the system, and continuing until it fails, leaves the system, or is excluded from the system.*

- *C-GMS-5: If process p suspects process q of being faulty, then if the core-GMS service is able to report new views, either q will be dropped from the system, or p will be dropped, or both.*
- *C-GMS-6: In a system with synchronized clocks and bounded message latencies, any process dropped from the system view will know that this has occurred within bounded time.*

As noted above, the core GMS protocol will not always be able to make progress: there are patterns of failures and communication problems that can prevent it from reporting new system views. For this reason, *C-GMS-5* is a conditional liveness property: *if* the core GMS is able to report new views, then it eventually acts upon process add or delete requests. It is not yet clear what conditions represent the weakest environment within which liveness of the GMS can always be guaranteed. For the protocol given above, the core GMS will make progress provided that at most a minority of processes from view *i* fail or are suspected as having failed during the period needed to execute the two- or three-phase commit protocol used to install new views. Such a characterization may seen evasive, since such a protocol may execute extremely rapidly in some settings and extremely slowly in others. However, unless the timing properties of the system are sufficiently strong to support estimation of the time needed to run the protocol, this seems to be as strong a statement as can be made.

We note that the failure detector called <>W in the work of Chandra and Toueg is characterized in terms somewhat similar to this [CT91, CHT92]. Very recent work by several researchers ([BDM95, Gue95, FKMB95) has shown that the <>W failure detector can be adapted to asynchronous systems in which messages can be lost during failures or processes can be "killed" because the majority of processes in the system consider them to be malfunctioning. Although fairly theoretical in nature, these studies are shedding light on the conditions under which problems such as membership agreement can always be solved, and those under which agreement may not always be possible (the theoreticians are fond of calling the latter settings in which the problem is "impossible"). To present this work here, however, would require a lengthy theoretical digression, which would be out of keeping with the generally practical tone of the remainder of the text. Accordingly, we cite this work and simply encourage the interested reader to turn to the papers for further detail.

13.10.4 Extending the GMS to Allow Partition and Merge Events

Research on the Transis system, at Hebrew University in Jerusalem, has yielded insights into the extension of protocols such as the one used to implement our primary component GMS so that it can permit continued operation during partitionings that leave no primary component, or allow activity in a non-primary component, reconciling the resulting system state when partitions later remerge [ADKM92b, Mal94]. Some of this work was done jointly with the Totem project at U. C. Santa Barbara [MAMA94].

Briefly, the approach is as follows. In Ricciardi's protocols, when the GMS is unable to obtain a majority vote in favor of a proposed new view, the protocol ceases to make progress. In the extended protocol, such a GMS can continue to produce new views, but no longer considers itself to be the primary partition of the system. Of course, there is also a complementary case in which the GMS encounters some other GMS and the two merge their membership views. It may now be the case that one GMS or the other was the primary component of the system, in which case the new merged GMS will also be primary for the system. On the other hand, perhaps a primary component fragmented in such a way that none of the surviving components considers itself to be the primary one. When this occurs, it may be that later, such components will remerge and primaryness can then be "deduced" by study of the joint histories of the two components. Thus, one can extend the GMS to make progress even when partitioning occurs.

Some recent work at the University of Bologna, on a system named Relacs, has refined this approach into one that is notable for its simplicity and clarity. Ozalp Babaoglu, working with Alberto Bartoli and Gianluca Dini, have demonstrated that a very small set of extensions to a view-synchronous environment suffice to support EVS-like functionality. They call their model Enriched View Synchrony, and describe it in a technical report that appeared shortly before this text went to press [BBD96]. Very briefly, Enriched View Synchrony arranges to deliver only *non-overlapping* group views within different components of a partitioned system. The reasoning behind this is that overlapping views can cause applications to briefly believe that the same process or site resides on both sides of a partition, leading to inconsistent behavior. Then, they provide a set of predicates by which a component can determine whether or not it has a quorum that would permit direct update of the global system state, and algorithmic tools for assisting in the state merge problem that arises when communication is reestablished. The author is not aware of any implementation of this model yet, but the primitives are simple and an implementation in a system such as Horus (Chapter 18) would not be difficult.

Having described these approaches, there remains an important question: whether or not it is *desirable* to allow a GMS to make progress in this manner. We defer this point until Chapter 16. In addition, as was noted in a footnote above, Keidar and Dolev have shown that there are cases in which no component is ever the primary one for the system, and yet dynamically unform actions can still be performed through a type of gossip that occurs whenever the network becomes reconnected and two non-minority components succeed in communicating [KD95]. Although interesting, this protocol is costly: prior to taking any action, a majority of all the processes in the system must be known to have seen the action. Indeed, Keidar and Dolev develop their solution for a static membership model, in which the GMS tracks subsets of a known maximum system membership. The majority requirement makes this protocol costly, hence although it is potentially useful in the context of wide-area systems that experience frequent partition failures, it is not likely that one would use it directly in the local-area communication layers of a system. We will return to this issue in Chapter 15, in conjunction with the model called *Extended Virtual Synchrony*.

13.11 Dynamic Process Groups and Group Communication

When the GMS is used to ensure system-wide agreement on failure and join events, the illusion of a failstop computing environment is created [SM94]. For example, if one were to implement the 3PC protocol of Section 13.6.1.2 using the notifications of the GMS service as a failure detection mechanism, the 3PC protocol would be non-blocking — provided, of course, that the GMS service itself is able to remain active and continue to output failure detections. The same power that the GMS brings to the 3PC problem can also be exploited to solve other problems, such as data replication, and offers us ways to do so that can be remarkably inexpensive relative to the quorum update solutions presented previously. Yet, we remain able to say that these systems are reliable in a strong sense: under the conditions when the GMS can make progress, such protocol will also make progress, and will maintain their consistency properties continuously, at least when permission to initiate new actions is limited to the primary component in the event that the system experiences a partitioning failure.

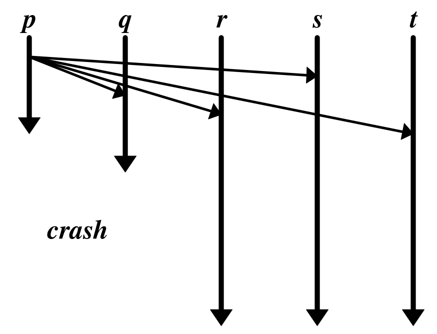


Figure 13-14: A dynamically uniform multicast involves a more costly protocol, but if the message is delivered to any destination, the system guarantees that the remaining destinations will also receive it. This is sometimes called a "safe" delivery, in the sense that it is safe to take actions that leave externally visible effects with respect to which the remainder of the system must be consistent. However, a non-uniform multicast is often safe for applications in which the action taken upon receipt of the message has only internal effects on the system state, or when consistency with respect to external actions can be established in other ways, for example from the semantics of the application.

In the subsections that follow, we develop this idea into an environment for computing with what are called *virtually synchronous process groups*. We begin by focusing on a simpler problem, closely related to 2PC, namely the reliable delivery of a message to a statically defined group of processes. Not surprisingly, our solution will be easily understood in terms of the 2PC protocol, delivering messages in the first phase if internal consistency is all that we require, and doing so in the second phase if dynamic uniformity (external consistency) is needed. We will then show how this solution can be extended to provide ordering on the delivery of messages; later, such ordered and reliable communication protocols will be used to implement replicated data and locking. Next, we show how the same protocols can also be used to implement dynamic groups of processes. In contrast to the dynamic membership protocols used in the GMS, however, these protocols will be quite a bit simpler and less costly. Next, we introduce a synchronization mechanism that allows us to characterize these protocols as failure-atomic with respect to group membership changes; this implements a model called the *view synchrony* model. Finally, we show how view synchrony can support a more extensive execution model called *virtually synchrony*, which supports a particularly simple and efficient style of fault-tolerant computing. Thus, step by step, we will show how to built up a reliable and consistent computing environment starting with the protocols embodied in the group membership service.

Up to the present we have focused on protocols in terms of a single group of processes at a time, but the introduction of sophisticated protocols and tools in support of process group computing also creates the likelihood that a system will need to support a great many process groups simultaneously and that a single distributed application may embody considerable numbers of groups — perhaps many groups per process that is present. Such developments have important performance implications, and will motivate us to reexamine our protocols.

Finally, we will turn to the software engineering issues associated with support for process group computing. This topic, which is addressed in the next chapter of the textbook, will center on a specific software system developed by the author and his colleagues, called Horus. The chapter also reviews a number of other systems, however, and in fact one of the key goals of Horus is to be able to support the features of some of these other systems within a common framework.

13.11.1 Group Communication Primitives

A *group communication primitive* is a procedure for sending a message to a set of processes that can be addressed without knowledge of the current membership of the set. Recall that we discussed the notion of a hardware *broadcast* capable of delivering a single message to every computer connected to some sort of communications device. Group communication primitives would normally transmit to subsets of the full membership of a computing system, so we use the term *multicast* to describe their behavior. A multicast is a protocol that sends a message from one *sender* process to multiple *destination* processes, which *deliver* it.

Suppose that we know that the current composition of some group *G* is $\{p_0, ... p_k\}$. What properties should a multicast to G satisfy?

The answer to this question will depend upon the application. As will become clear in Chapters 16 and 17, there are a great number of "reliable" applications for which a multicast with relatively weak properties would suffice. For example, an application that is simply seeking information that any of the members of *G* can provide might multicast an inquiry to *G* as part of an RPC-style protocol that requires a single reply, taking the first one that is received. Such a multicast would ideally avoid sending the message to the full membership of *G*, resorting instead to heuristics for selecting a member that is likely to respond quickly (like one on the same machine as the sender), and implementing the multicast as an RPC to this member that falls back to some other strategy if no local process is found, or if it fails to respond before a timeout elapses. One might argue that this is hardly a legitimate implementation of a multicast, since it often behaves like an RPC protocol, but there are systems that implement precisely this functionality and find it extremely useful.

A multimedia system might use a similar multicast, but with real-time rate-control or latency properties that correspond to the requirements of the display software [RS92]. As groupware uses of distributed systems become increasingly important, one can predict that vendors of web browsers will focus on offering such functions and that telecommunications service providers will provide the corresponding communications support. Such support would probably need to be aware of the video

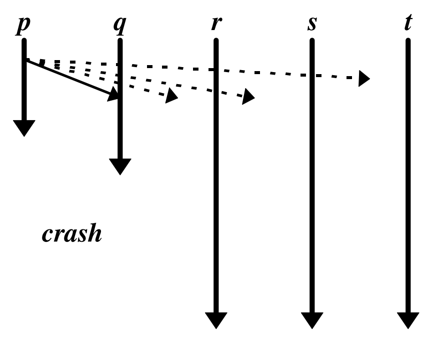


Figure 13-15: Non-dynamically uniform message delivery: the message is delivered to one destination, q, but then the sender and q crash and the message is delivered to none of the remaining destinations.

encoding that is used, for example MPEG, so that it can recognize and drop data selectively if a line becomes congested or a date frame is contaminated or will arrive too late to be displayed.

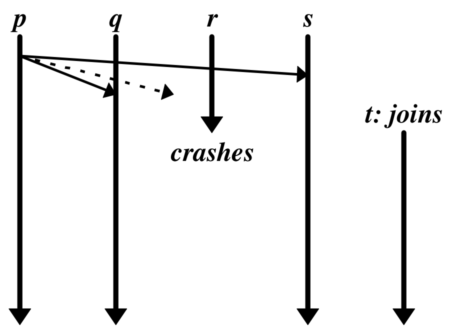


Figure 13-16: Neither form of atomicity guarantees that a message will actually be delivered to a destination that fails before having an opportunity to deliver the message, or that joins the system after the message is sent.

Distributed systems that use groups as a structuring construct may require guarantees of a different nature, and we now focus on those. A slightly more ambitious multicast primitive that could be useful in such a group-oriented application might work by sending the message to the full membership of the destination set, but without providing reliability, ordering, flow control, or any form of feedback in regard to the outcome of the operation. Such a multicast could be implemented by invoking the IP multicast (or UDP multicast) transport primitives that we discussed in Section 3.3.1. The user whose application requires any of these properties would implement some form of end-to-end protocol to achieve them.

A more elaborate form of multicast would be a *failure-atomic* multicast, which guarantees that for a specified class of failures, the multicast will either reach all of its destinations, or none of them. As we just observed, there are really two forms of failure atomicity that might both be interesting, depending on the circumstance. A failure-atomic multicast is *dynamically uniform* if it guarantees that if any process delivers the multicast, than all processes that remain operational will do so, regardless of whether or not the initial recipient remains operational subsequent to delivering the message. A failure-atomic multicast that is not dynamically uniform would guarantee only that if one waits long enough, one will find either that all the destinations that remained operational delivered the message, or that none did so. To avoid trivial outcomes, both primitives require that the message be delivered eventually if the sender doesn't fail.[11]

To reiterate a point made earlier, the key difference between a dynamically uniform protocol and one that is merely failure-atomic but non-uniform has to do with the obligation when the first delivery event occurs. From the perspective of a recipient process *p*, if *m* is sent using a protocol that provides dynamic uniformity, then when *p* delivers *m* it also knows that any future execution of the system in which a set of processes remains operational will also guarantee the delivery of *m* within its remaining destinations among that set of processes, as illustrated in Figure 13-14. (We state it this way because

[11] Such a definition leaves open the potential for another sort of trivial solution: one in which the act of invoking the multicast primitive causes the sender to be excluded from the system as faulty. A rigorous non-triviality requirement would also exclude this sort of behavior, and there may be other trivial cases that this author is not aware of. However, as was noted early in the textbook, our focus here is on reliability as a practical engineering discipline, and not on the development of a mathematics of reliability. The author is convinced that such a mathematics is urgently needed, and is concerned that minor problems such as this one could have subtle and undesired implications in correctness proofs. However, the logical formalism that would permit a problem such as this to be specified completely rigorously remain elusive, apparently because of the self-defined character of a system that maintains its own membership and seeks an internal consistency guarantee but not an external one. This author is hopeful that with further progress in the area of specification, limitations such as these can be overcome in the near future.

processes that join after *m* was sent are not required to deliver *m*). On the other hand, if process *p* receives a non-uniform multicast *m*, *p* knows that if both the sender of *m* and *p* crash or are excluded from the system membership, *m* may not reach its other destinations, as seen in Figure 13-15.

Dynamic uniformity is a costly property to provide, but *p* would want this guarantee if its actions upon receiving *m* will leave some externally visible trace that the system must know about, such as redirecting an airplane or issuing money from a automatic teller. A non-dynamic failure atomicity rule would be adequate for most internal actions, like updating a data structure maintained by *p,* and even for some external ones, like displaying a quote on a financial analyst's workstation, or updating an image in a collaborative work session. In these cases, one may want the highest possible performance and not be willing to pay a steep cost for the dynamic uniformity property because the guarantee it provides is not actually a necessary one. Notice that neither property ensures that a message will reach *all* of its destinations, because no protocol can be sure that a destination will not crash before having an opportunity to deliver the message, as seen in Figure 13-16.

In this section of the textbook, the word "failure" is understood to refer to the reporting of failure events by the GMS [SM94]. Problems that result in the detection of an apparent failure by a system member are reported to the GMS but do not directly trigger any actions (except that messages from apparently faulty processes are ignored), until the GMS officially notifies the full system that the failure has occurred. Thus, although one could develop failure atomic multicasts against a variety of failure models, we will not be doing so in this section.

13.12 Delivery Ordering Options

Turning now to multicast delivery ordering, let us start by considering a multicast that offers no guarantees whatsoever. Using such a multicast, a process that sends two messages m_0 and m_1 concurrently would have no assurances at all about their relative order of delivery or relative atomicity. That is, suppose that m_0 was the message sent first. Not only might m_1 reach any destinations that it shares with m_0 first, but a failure of the sender might result in a scenario where m_1 was delivered atomically to all its destinations but m_0 was not delivered to any process that remains operational (Figure 13-17). Such an outcome would be atomic on a per-multicast basis, but might not be a very useful primitive from the perspective of the application developer! Thus, we should ask what forms of order a multicast primitive can guarantee, but also ask how order is connected to atomicity in our failure-atomicity model.

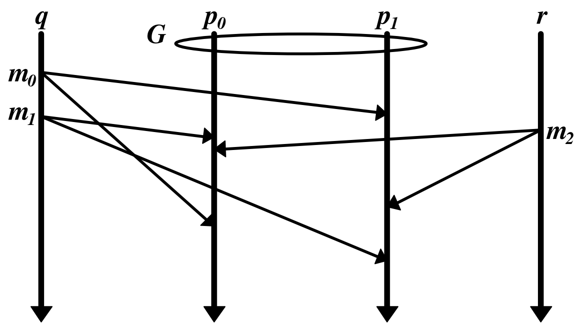


Figure 13-17: An unordered multicast provides no guarantees. Here, m_0 was sent before m_1, but is received after m_1 at destination p_0. The reception order for m_2, sent concurrently by process r, is different at each of its destinations.

We will be studying a hierarchy of increasingly ordered delivery properties. The weakest of these is usually called "sender order" or "FIFO order" and requires that if the same process sends m_0 and m_1 then m_0 will be delivered before m_1 at any destinations they have in common. A slightly stronger ordering property is called causal delivery order, and says that if $send(m_0) \rightarrow send(m_1)$, then m_0 will be delivered before m_1 at any destinations they have in common (Figure 13-19). Still stronger is an order whereby any processes that receive the same two messages receive them in the same order: if at process *p*, $deliv(m_0) \rightarrow deliv(m_1)$, then m_0 will be delivered before m_1 at all destinations they have in common. This is sometimes called a totally ordered delivery protocol, but to do so is something of a misnomer, since one can imagine a number of ordering properties that would be total in this respect without necessarily implying the existing of a single system-wide total ordering on all the messages sent in the system. The reason for this is that our definition focuses on delivery orders where messages overlap, but doesn't actually relate these orders to an acyclic system-wide ordering. The Transis project calls this type of locally ordered multicast an "agreed" order, and we like this term too: the destinations agree on the order, even for multicasts that may have been initiated concurrently and hence that may be unordered by their senders (Figure 13-20). However, the agreed order is more commonly called a "total" order or an "atomic" delivery order in the systems that support multicast communication and the papers in the literature.

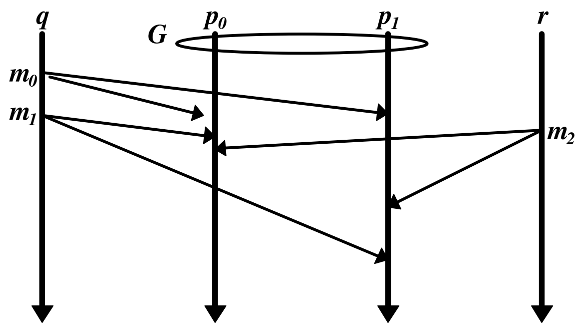


Figure 13-18: Sender ordered or "fifo" multicast. Notice that m_2, which is sent concurrently, is unordered with respect to m_0 and m_1.

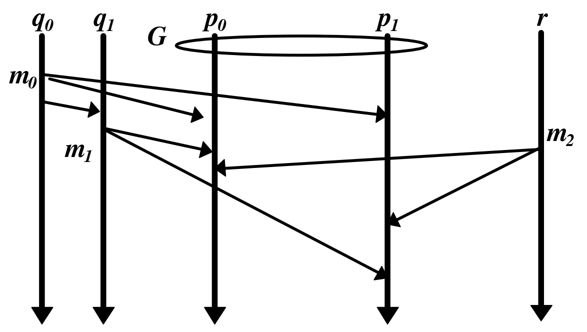


Figure 13-19: Causally ordered multicast delivery. Here m_0 is sent before m_1 in a causal sense, because a message is sent from q_0 to q_1 after m_0 was sent, and before q_1 sends m_1. Perhaps q_0 has requested that q_1 send m_1. m_0 is consequently delivered before m_1 at destinations that receive both messages. Multicast m_2 is sent concurrently and no ordering guarantees are provided. In this example, m_2 is delivered after m_1 by p_0 and before m_1 by p_1.

One can extend the agreed order into a causal agreed order (now one requires that if the sending events were ordered by causality, the delivery order will respect the causal send order), or into a system-wide agreed order (one requires that there exists a single system-wide total order on messages, such that the delivery ordering used at any individual process is consistent with the message ordering in this system total order). Later we will see why these are not identical orderings. Moreover, in systems that have multiple process groups, the issue will arise of how to extend ordering properties to span multiple process groups.

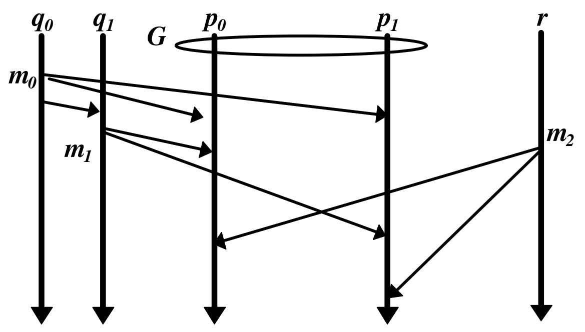


Figure 13-20: When using a totally ordered multicast primitive, p_0 and p_1 receive exactly the same multicasts, and the message are delivered in identical orders. Above the order happens to also be causal, but this is not a specific guarantee of the primitive.

Wilhelm and Schiper have proposed that total ordering be further classified as *weak* or *strong* in terms analogous to the dynamically uniform and non-uniform delivery properties. A weak total ordering property would be one guaranteed to hold only at *correct* processes, namely those that remain operational until the protocol terminates. A strong total ordering property would hold even at faulty processes, namely those that fail after delivering messages but before the protocol as a whole has terminated.

For example, suppose that a protocol fixes the delivery ordering for messages m_1 and m_2 at process p, delivering m_1 first. If p fails, a weak total ordering would permit the delivery of m_2 before m_1 at some other process q that survives the failure, even though this order is not the one seen by p. Like dynamic uniformity, the argument for strong total ordering is that this may be required if the ordering of messages may have externally visible consequences, which could be noticed by an external observer who interacts with a process that later fails, and then interacts with some other process that remained operational. Naturally, this guarantee has a price, though, and one would prefer to use a less costly weak protocol in settings where such a guarantee is not required.

Let us now return to the issue raised briefly above, concerning the connection between the ordering properties for a set of multicasts and their failure atomicity properties. To avoid creating an excessive number of possible multicast protocols, we will assume here that the developer of a reliable application will typically want the specified ordering property to extend into the failure atomicity properties of the primitives used, too. That is, in a situation where the ordering property of a multicast would imply that message m_0 should be delivered before m_1 if they have any destinations in common, we will require that if m_1 is delivered successfully, then m_0 must be too, whether or not they actually do have common destinations. This is sometimes called a *gap freedom* guarantee: it is the constraint that failures cannot leave holes or gaps in the ordered past of the system. Such a gap is seen in Figure 13-21.

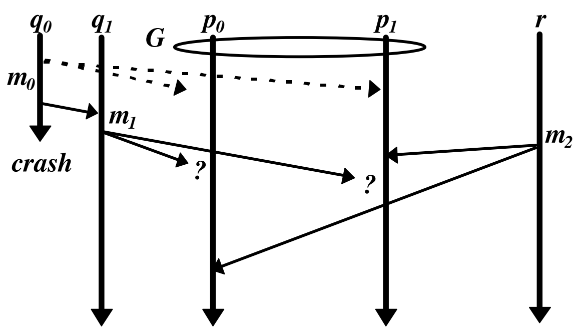


Figure 13-21: In this undesirable scenario, the failure of q_0 leaves a "causal gap" in the message delivery order, preventing q_1 from communicating with members of G. If m_1 is delivered, the causal ordering property would be violated, because send(m_0)→send(m_1). But m_0 will never be delivered. Thus q_1 is logically partitioned from G!

Notice that this rule is stated so that it would apply even if m_0 and m_1 have no destinations in common! The reason for this is that ordering requirements are normally transitive: if m_0 is before m_1 and m_1 is before m_2, then m_0 is also before m_2, and we would like both delivery ordering obligations and failure atomicity obligations to be guaranteed between m_0 and m_2. Had we instead required that "in a situation where the ordering property of a multicast implies that message m_0 should be delivered before m_1, then if they have any destinations in common, we will also require that if m_1 is delivered successfully, then m_0 must be too," the delivery atomicity requirement might not apply between m_0 and m_2.

Lacking a gap-freedom guarantee, one can imagine runs of a system that would leave orphaned processes that are technically prohibited from communicating with one-another. For example, in Figure 13-21, q_1 sends message m_1 to the members of group G causally after m_0 was sent by q_0 to G. The members of G are now required to deliver m_0 before delivering m_1. However, if the failure atomicity rule is such that the failure of q_0 could prevent m_0 from ever being delivered, this ordering obligation can only be satisfied by *never* delivering m_1. One could say that q_1 has been partitioned from G by the ordering obligations of the system! Thus, if a system provides ordering guarantees and failure atomicity guarantees, it should normally extend the latter to encompass the former.

Yet an additional question arises if a process sends multicasts to a group while processes are joining or leaving it. In these cases the membership of the group will be in flux at the time that the message is sent, and one can imagine a number of ways to interpret group atomicity that a system could implement. We will defer this problem for the present, returning to it in Section 13.12.2.

13.12.1.1 Non-Uniform Failure-Atomic Group Multicast

Consider the following simple, but inefficient group multicast protocol. The sender adds a header to its message listing the membership of the destination group at the time that it sends the message. It now transmits the message to the members of the group, perhaps taking advantage of a hardware multicast feature if one is available, and otherwise transmitting the message over stream-style reliable connections to the destinations. (However, unlike a conventional stream protocol, here we will assume that the connection is only broken if the GMS reports that one of the endpoints has left the system).

Upon reception of a message, the destination processes deliver it immediately, but then resend it to the remaining destinations. Again, each process uses reliable stream-style channels for this retransmission stage, breaking the channel only if the GMS reports the departure of an endpoint. A participant will now receive one copy of the message from the sender, and one from each non-failed participant other than itself. After delivery of the initial copy, it therefore discards any duplicates. We will now argue that this protocol is failure-atomic, although not dynamically uniform.

To see that it is failure-atomic, assume that some process p_i receives and delivers a copy of the message and remains operational. Failure atomicity tells us that all other destinations that remain operational must also receive and deliver the message. It is clear that this will occur, since the only condition under which p_i would fail to forward a message to p_j would be if the GMS reports that p_i has failed, or if it reports that p_j has failed. But we assumed that p_i does not fail, and the output of the GMS can be trusted in this environment. Thus, the protocol achieves failure-atomicity. To see that the protocol is not dynamically uniform, consider the situation if the sender sends a copy of the message only to process p_i and then both processes fail. In this case, p_i may have delivered the message and then executed for some extended period of time before crashing or detecting that it has been partitioned from the system. The message has thus been delivered to one of the destinations and that destination may well have acted on it in a visible way, and yet none of the processes that remain operational will ever receive it. As we noted earlier, this often will not pose a problem for the application, but it is a behavior that the developer must anticipate and treat appropriately in his or her application.

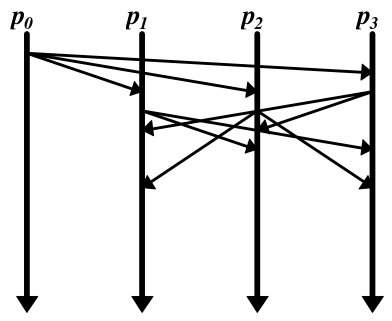


Figure 13-22: A very simple reliable multicast protocol. The initial round of messages triggers a second round of messages as each recipient echoes the incoming message to the other destinations.

As can be seen in Figure 13-22, this simple protocol is a costly one: to send a message to n destinations requires $O(n^2)$ messages. Of course, with hardware broadcast functions, or if the network is not a bottleneck, the cost will be lower, but the protocol still requires each process to send and receive each message approximately n times.

But now, suppose that we delay the "retransmission" stage of the protocol, doing this only if the GMS informs the participants that the sender has failed. This change yields we have a less costly protocol, which requires *n* messages (or just one, if hardware broadcast is an option), but in which the participants may need to save a copy of each message indefinitely. They would do this "just in case" the sender fails.

Recall that we are transmitting messages over a reliable stream. It follows that within the lower levels of the communication system, there is an occassional acknowledgment flowing from each participant back to the sender. If we tap into this information, the sender will "know" when the participants have all received copies of its message. It can now send a second phase message out, informing the participants that it is safe to delete the saved copy of each message, although they must still save the message "identification" information to reject duplicates if the sender happens to crash midway through this stage. At this stage the participants can disable their retransmission logic and discard the saved copy of the message (although not its identification information), since any retransmitted message would be a duplicate. Later, the sender could run still a third phase, telling the participants that they can safely delete even the message identification information, because after the second phase there will be no risk of a failure that would cause the message to be retransmitted by the participants.

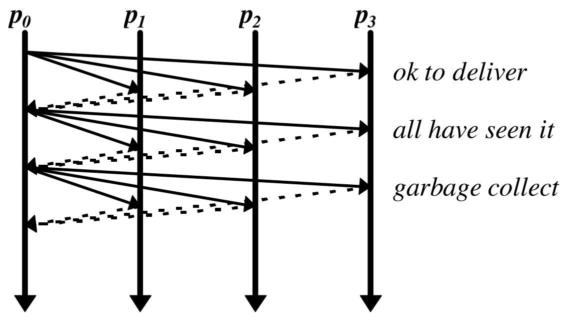


Figure 13-23: An improved 3-phase protocol. Ideally, the second and third phases would be piggybacked onto other multicasts from the same sender to the same set of destinations, and hence would not require "extra" messages.

But now a further optimization is possible. There is no real hurry to run the third phase of this protocol, and even the second phase can be delayed to some degree. Moreover, most processes that send a multicast will tend to send a subsequent one soon afterwards: this principle is well known from all forms of operating systems and database software, and can be summarized by the maxim that *the most likely action by any process is to repeat the same action it took most recently*. Accordingly, it makes sense to delay sending out messages for the second and third phase of the protocol, in the hope that a new multicast will be initiated and this information can be piggybacked onto the first-stage of an outgoing message associated with that subsequent protocol!

In this manner, we arrive at a solution, illustrated in Figure 13-23, that has an average cost of *n* messages per multicast, or just one if hardware broadcast can be exploited, plus some sort of background cost associated with the overhead to implement a reliable stream channel. When a failure does occur, any pending multicast will suddenly generate as many as n^2 additional messages, but even this effect can potentially be mitigated. For example, since the GMS provides the same membership list to all processes and the message itself carried the list of its destinations, the participants can delay briefly in the hope that some jointly identifiable "lowest ranked" participant will turn out to have received the message and will terminate the protocol on behalf of all. We omit the details of such a solution, but any serious system for reliable distributed computing would implement a variety of such mechanisms to keep costs down to an absolute minimum, and to maximize the value of each message actually transmitted using piggybacking, delaying tactics, and hardware broadcast.

13.12.1.2 Dynamically Uniform Failure-Atomic Group Multicast

We can extend the above protocol to one that is dynamically uniform, but doing so requires that no process deliver the message until it is known the processes in the destination group all have a copy. (In some cases it may be sufficient to know that a majority have a copy, but we will not concern ourselves with these sorts of special cases now, because they are typically limited to the processes that actually run the GMS protocol).

We could accomplish this in the original inefficient protocol of Figure 13-22, by modifying the original non-uniform protocol to delay the delivery of messages until a copy has been received from every destination that is still present in the membership list provided by the GMS. However, such a protocol would suffer from the inefficiencies that lead us to optimize the original protocol into the one in Figure 13-23. Accordingly, it makes more sense to focus on that improved protocol.

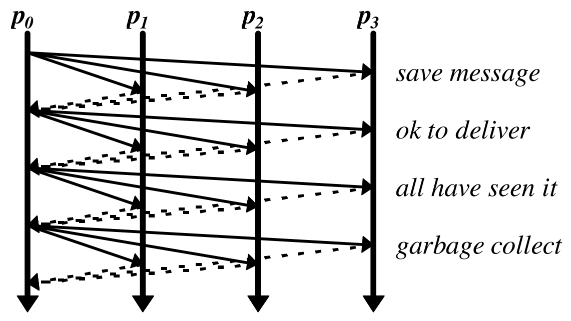


Figure 13-24: A dynamically uniform version of the optimized, reliable multicast protocol. Latency to delivery may be much higher, because no process can deliver the message until all processes have received and saved a copy. Here, the third and fourth phases can piggyback on other multicasts but the first two stages may need to be executed as promptly as possible, to avoid increasing the latency still further. Latency is often a key performance factor.

Here, it can be seen that an additional round of messages will be needed before the multicast can be delivered initially; the remainder of the protocol can then be used without change (Figure 13-24). Unfortunately, though, this initial round also delays the delivery of the messages to their destinations. In the original protocol, a message could be delivered as soon as it reached a destination for the first time, thus the latency to delivery is precisely the latency from the sender to a given destination for a single "hop". Now the latency might be substantially increased: for a dynamically uniform delivery, we will need to wait for a round-trip to the slowest process in the set of destinations, and then one more hop until the sender has time to inform the destinations that it is safe to deliver the messages. In practice, this may represent an increase in latency of a factor of ten or more. Thus, while dynamically uniform guarantees are sometimes needed, the developer of a distributed application should request this property only when it is genuinely necessary, or performance (to the degree that latency is a factor in performance) will suffer badly.

13.12.2 Dynamic Process Groups

When we introduced the GMS, our system became very dynamic, allowing processes to join and leave at will. But not all processes in the system will be part of the same application, and the protocols presented in the previous section are therefore assumed to be sent to groups of processes that represent subsets of the full system membership. This is seen in Figure 13-25, which illustrates the structure of a hypothetical trading system, in which services (replicated for improved performance or availability) implement theoretical pricing calculations. Here we have one big system, with many small groups in it. How should the membership of such a subgroup be managed?

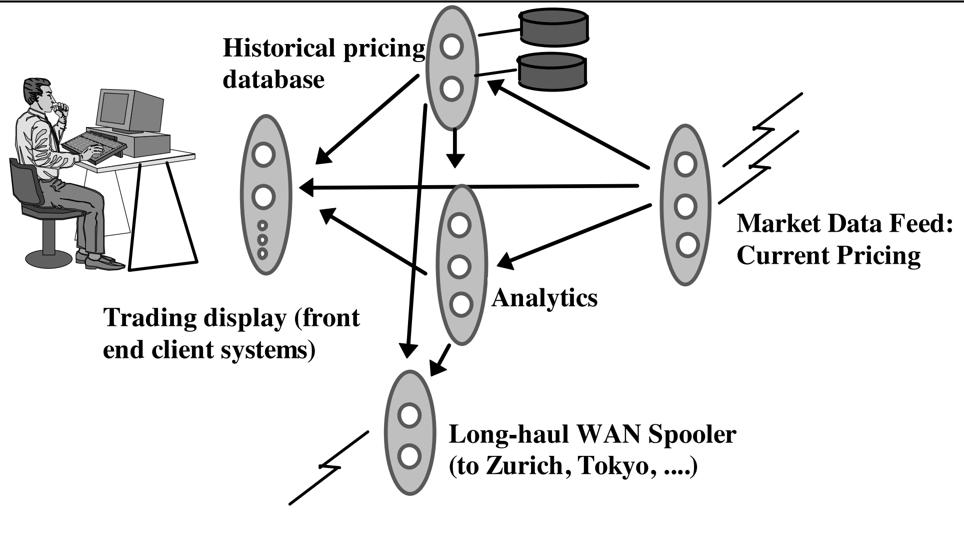

Historical pricing database
Market Data Feed: Current Pricing
Analytics
Trading display (front end client systems)
Long-haul WAN Spooler (to Zurich, Tokyo, ...)

Figure 13-25: Distributed trading system may have both "static" and "dynamic" uses for process groups. The historical database, replicated for load-balancing and availability, is tied to the databases themselves and hence can be viewed as static. This is also true of the market data feeds, which are often redundant for fault-tolerance. Other parts of the system, however, such as the analytics (replicated for parallelism) and the client interface processes (one or more per trader) are highly dynamic groups. For uniformity of the model, it makes sense to adopt a dynamic group model, but to keep in mind that some of these groups in fact manage physical resources.

In this section, we introduce a membership management protocol based on the idea that a single process within each group will serve as the "coordinator" for membership changes. If a process wishes to join the group, or voluntarily leaves the group, this coordinator will update the group membership accordingly. (The role of being coordinator will really be handled by the layer of software that implements groups, so this won't be visible to the application process itself.) Additionally, the coordinator will monitor the members (through the GMS, and by periodically pinging them to verify that they are still healthy), excluding any failed processes from the membership much as in the case of a process that leaves voluntarily.

In the approach we present here, all processes that belong to a group maintain a local copy of the current membership list. We call this the "view" of the group, and will say that each time the membership of the group changes, a "new view" of the group is reported to the members. Our protocol will have the property that all group members see the identical sequence of group views within any given component of a partitioned system. In practice, we will mostly be interested in primary-component partitions, and in these cases, we will simply say that all processes either see identical views for a group or, if excluded from the primary component, cease to see new views and eventually detect that they are partitioned, at which point a process may terminate or attempt to rejoin the system much as a new process would.

The members of a group depend upon their coordinator for the reporting of new views, and consequently monitor the liveness of the coordinator by periodically pinging it. If the coordinator appears to be faulty, the member or members that detect this report the situation to the GMS in the usual manner, simultaneously cutting off communication to the coordinator and starting to piggyback or "gossip" this information on messages to other members, which similarly cut their channels to the coordinator and, if necessary, relay this information to the GMS. The GMS will eventually report that the coordinator has failed, at which point the lowest ranked of the remaining members takes over as the new coordinator, and similarly if this process fails in its turn.

Interestingly, we have now solved our problem, because we can use the non-dynamically uniform multicast protocol to distribute new views within the group. In fact, this hides a subtle point, to which we will return momentarily, namely the way to deal with ordering properties of a reliable multicast, particularly in the case where the sender fails and the protocol must be terminated by other processes in the system. However, we will see below that the protocol has the necessary ordering properties when it operates over stream connections that guarantee FIFO delivery of messages, and when the failure handling mechanisms introduced earlier are executed in the same order that the messages themselves were initially seen (i.e. if process p_i first received multicast m_0 before multicast m_1, then p_i retransmits m_0 before m_1).

13.12.3 View-Synchronous Failure Atomicity

We have now created an environment within which a process that joins a process group will receive the membership view for that group as of the time it was added to the group, and will subsequently observe any changes that occur until it crashes or leaves the group, provided only that the GMS continues to report failure information. Such a process may now wish to initiate multicasts to the group using the reliable protocols presented above. But suppose that a process belonging to a group fails while some multicasts from it are pending? When can the other members be certain that they have seen "all" of its messages, so that they can take over from it if the application requires that they do so?

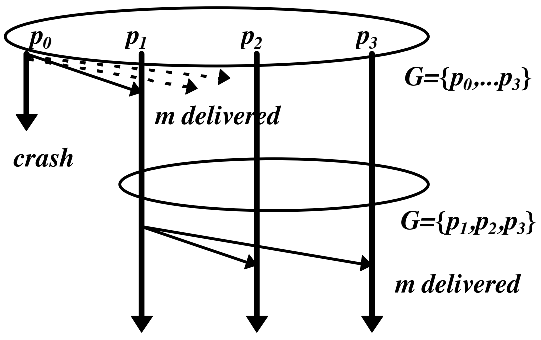


Figure 13-26: Although m *was sent when p_0 belonged to G, it reaches p_2 and p_3 after a view change reporting that p_0 has failed. The red and blue delivery events thus differ in that the recipients will observe a different view of the process group at the time the message arrives. This can result in inconsistency if, for example, the membership of the group is used to subdivide the incoming tasks among the group members.*

Up to now, our protocol structure would not provide this information to a group member. For example, it may be that process p_0 fails after sending a message to p_1 but to no other member. It is entirely possible that the failure of p_0 will be reported through a new process group view before this message is finally delivered to the remaining members. Such a situation would create difficult problems for the application developer, and we need a mechanism to avoid it. This is illustrated in Figure 13-26.

It makes sense to assume that the application developer will want failure notification to represent a "final" state with regard to the failed process. Thus, it would be preferable for all messages initiated by process p_0 to have been delivered to their destinations before the failure of p_0 is reported through the delivery of a new view. We will call the necessary protocol a *flush* protocol, meaning that it flushes partially completed multicasts out of the system, reporting the new view only after this has been done.

In the example illustrated by Figure 13-26, we did not include the exchange of messages required to multicast the new view of group G. Notice, however, that the figure is probably incorrect if the new view coordinator for group G is actually process p_1. To see this, recall that the communication channels are FIFO and that the termination of an interrupted multicast protocol requires only a single round of communication. Thus, if process p_1 simply runs the completion protocol for multicasts initiated by p_0 before it starts the new-view multicast protocol that will announce that p_0 has been dropped by the group, the pending multicast will be completed first. This is shown below.

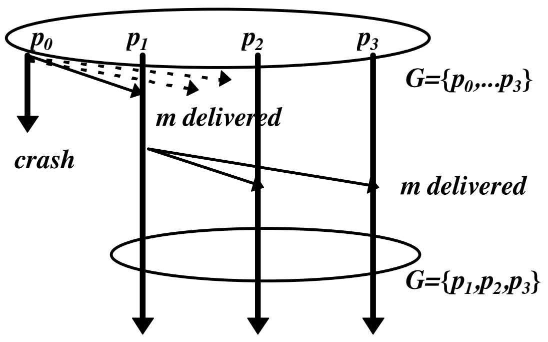


Figure 13-27: Process p_1 flushes pending multicasts before initiating the new-view protocol.

We can guarantee this behavior even if multicast *m* is dynamically uniform, simply by delaying the new view multicast until the outcome of the dynamically uniform protocol has been determined.

On the other hand, the problem becomes harder if p_1 (which is the only process to have received the multicast from p_0) is not the coordinator for the new view protocol. In this case, it will be necessary for the new-view protocol to operate with an additional round, in which the members of G are asked to flush any multicasts that are as yet unterminated, and the new-view protocol runs only when this flush phase has finished. Moreover, even if the new view protocol is being executed to drop p_0 from the group, it is possible that the system will soon discover that some other process, perhaps p_2, is also faulty and must also be dropped. Thus, a flush protocol should flush messages *regardless of their originating process* with the result that all multicasts will have been flushed out of the system before the new view is installed.

These observations lead to a communication property that Babaoglu and his colleagues have called *view synchronous communication*, which is one of several properties associated with the *virtual synchrony model* introduced by the author and Thomas Joseph in 1985-1987. A view-synchronous communication system ensures that any multicast initiated in a given view of some process group will be failure-atomic with respect to that view, and will be terminated before a new view of the process group is installed.

One might wonder how a view-synchronous communication system can prevent a process from initiating new multicasts while the view installation protocol is running. If such multicasts are locked out, there may be an extended delay during which no multicasts can be transmitted, causes performance problems for the application programs layered over the system. But if such multicasts are permitted, the first phase of the flush protocol will not have flushed *all* the necessary multicasts!

A solution for this problem was suggested independently by Ladin and Malki, working on systems called Harp and Transis, respectively. In these systems, if a multicast is initiated while a protocol to install view *i* of group G is running, the multicast destinations are taken to be the future membership of G when that new view has been installed. For example, in the figure above, a new multicast might be initiated by process p_2 while the protocol to exclude p_0 from G is still running. Such a new multicast would be addressed to $\{p_1, p_2, p_3\}$ (not to p_0), and would be delivered only after the new view is delivered to the remaining group members. The multicast can thus be initiated while the view change protocol is running, and would only be delayed if, when the system is ready to deliver a copy of the message to some group member, the corresponding view has not yet been reported. This approach will often avoid delays completely, since the new view protocol was already running and will often terminate in roughly the same amount of time as will be needed for the new multicast protocol to start delivering messages to destinations. Thus, at least in the most common case, the view change can be accomplished even as communication to the group continues unabated. Of course, if multiple failures occur, messages will still queue up on reception and will need to be delayed until the view flush protocol terminates, so this desirable behavior cannot always be guaranteed.

13.12.4 Summary of GMS Properties

The following is an informal (English-language) summary of the properties that a group membership service guarantees to members of subgroups of the full system membership. We use the term process group for such a subgroup. When we say "guarantees" the reader should keep in mind that a GMS service does not, and in fact cannot, guarantee that it will remain operational despite all possible patterns of failures and communication outages. Some patterns of failure or of network outages will prevent such a service from reporting new system views and will consequently prevent the reporting of new process group views. Thus, the guarantees of a GMS are relative to a constraint, namely that the system provide a sufficiently reliable transport of messages and that the rate of failures is sufficiently low.

• *GMS-1: Starting from an initial group view, the GMS reports new views that differ by addition and deletion of group members. The reporting of changes is by the two-stage interface described above, which gives protocols an opportunity to flush pending communication from a failed process before its failure is reported to application processes.*

•*GMS-2: The group view is not changed capriciously. A process is added only if it has started and is trying to join the system, and deleted only if it has failed or is suspected of having failed by some other member of the system.*

• *GMS-3: All group members observe continuous subsequences of the same sequence of group views, starting with the view during which the member was first added to the group, and ending either with a view that registers the voluntary departure of the member from the group, or with the failure of the member.*

• *GMS-4: The GMS is fair in the sense that it will not indefinitely delay a view change associated with one event while performing other view changes. That is, if the GMS service itself is live, join requests will eventually cause the requesting process to be added to the group, and leave or failure events will eventually cause a new group view to be formed that excludes the departing process.*

• *GMS-5: Either the GMS permits progress only in a primary component of a partitioned network, or, if it permits progress in non-primary components, all group views are delivered with an additional boolean flag indicating whether or not the group view resides in the primary component of the network. This single boolean flag is shared by all the groups in a given component: the flag doesn't indicate whether a given view of a group is primary for that group, but rather indicates whether a given view of the group resides in the primary component of the encompassing network.*

Although we will not pursue these points here, it should be noted that many networks have some form of critical resources on which the processes reside. Although the protocols given above are designed to make progress when a majority of the processes in the system remain alive after a partitioning failure, a more reasonable approach would also take into account the resulting resource pattern. In many settings, for example, one would want to define the primary partition of a network to be the one that retains the majority of the servers after a partitioning event. One can also imagine settings in which the primary should be the component within which access to some special piece of hardware remains possible, such as the radar in an air-traffic control application. These sorts of problems can generally be solved by associating weights with the processes in the system, and redefining the majority rule as a weighted majority rule. Such an approach recalls work in the 1970's and early 1980's by Bob Thomas of BBN on weighted majority voting schemes and weighted quorum replication algorithms [Tho79, Gif79].

13.12.5 Ordered Multicast

Earlier, we observed that our multicast protocol would preserve the sender's order if executed over FIFO channels, and if the algorithm used to terminate an active multicast was also FIFO. Of course, some systems may seek higher levels of concurrency by using non-FIFO reliable channels, or by concurrently executing the termination protocol for more than one multicast, but even so, such systems could potentially "number" multicasts to track the order in which they should be delivered. Freedom from gaps in the sender order is similarly straightforward to ensure.

This leads to a broader issue of what forms of multicast ordering are useful in distributed systems, and how such orderings can be guaranteed. In developing application programs that make use of process groups, it is common to employ what Leslie Lamport and Fred Schneider call a *state machine* style of distributed algorithm [Sch90]. Later, we will see reasons that one might want to relax this model, but the original idea is to run identical software at each member of a group of processes, and to use a failure-atomic multicast to deliver messages to the members in identical order. Lamport's proposal was that Byzantine Agreement protocols be used for this multicast, and in fact he also uses Byzantine Agreement on messages output by the group members. The result of this is that the group as a whole gives the behavior of a single ultra-reliable process, in which the operational members behave identically and the faulty behaviors of faulty members can be tolerated up to the limits of the Byzantine Agreement protocols. Clearly, the method requires deterministic programs, and thus could not be used in applications that are multi-threaded or that accept input through an interrupt-style of event notification. Both of these are common in modern software, so this restriction may be a serious one.

As we will use the concept, through, there is really only one aspect of the approach that is exploited, namely that of building applications that will remain in identical states if presented with identical inputs in identical orders. Here we may not require that the applications actually be deterministic, but merely that they be designed to maintain identically replicated states. This problem, as we will see below, is solvable even for programs that may be very non-deterministic in other ways, and very concurrent. Moreover, we will not be using Byzantine Agreement, but will substitute various weaker forms of multicast protocol. Nonetheless, it has become usual to refer to this as a variation on Lamport's state machine approach, and it is certainly the case that his work was the first to exploit process groups in this manner.

13.12.5.1 Fifo Order

The FIFO multicast protocol is sometimes called *fbcast* (the "b" comes from the early literature which tended to focus on static system membership and hence on "broadcasts" to the full membership; "fmcast" might make more sense here, but would be non-standard). Such a protocol can be developed using the methods discussed above, provided that the software used to implement the failure recovery algorithm is carefully designed to ensure that the sender's order will be preserved, or at least tracked to the point of message delivery.

There are two variants on the basic *fbcast*: a normal *fbcast*, which is non-uniform, and a "safe" *fbcast*, which guarantees the dynamic uniformity property at the cost of an extra round of communication.

The costs of a protocol are normally measured in terms of the latency before delivery can occur, the message load imposed on each individual participant (which corresponds to the CPU usage in most settings), the number of messages placed on the network as a function of group size (this may or may not be a limiting factor, depending on the properties of the network), and the overhead required to represent protocol-specific headers. When the sender of a multicast is also a group member, there are really two latency metrics that may be important: latency from when a message is sent to when it is delivered, which is usually expressed as a multiple of the communication latency of the network and transport software, and the latency from when the sender initiates the multicast to when it learns the delivery ordering for that multicast. During this period, some algorithms will be waiting — in the sender case, the sender may be unable to proceed until it knows "when" its own message will be delivered (in the sense of ordering with respect to other concurrent multicasts from other senders). And in the case of a destination process, it is clear that until the message is delivered, no actions can be taken.

In all of these regards, *fbcast* and *safe fbcast* are inexpensive protocols. The latency seen by the sender is minimal: in the case of *fbcast,* as soon as the multicast has been transmitted, the sender knows that the message will be delivered in an order consistent with its order of sending. Still focusing on *fbcast,* the latency between when the message is sent and when it is delivered to a destination is exactly that of the network itself: upon receipt, a message is immediately deliverable. (This cost is much higher if the sender fails while sending, of course). The protocol requires only a single round of communication, and other costs are hidden in the background and often can be piggybacked on other traffic. And the header used for *fbcast* needs only to identify the message uniquely and capture the sender's order, information that may be expressed in a few bytes of storage.

For the safe version of *fbcast*, of course, these costs would be quite a bit higher, because an extra round of communication is needed to know that all the intended recipients have a copy of the message. Thus *safe fbcast* has a latency at the sender of roughly twice the maximum network latency experienced in sending the message (to the slowest destination, and back), and a latency at the destinations of roughly three times this figure. Notice that even the fastest destinations are limited by the response times of the slowest destinations, although one can imagine "partially safe" implementations of the protocol in which a majority of replies would be adequate to permit progress, and the view change protocol would be changed correspondingly.

The *fbcast* and *safe fbcast* protocols can be used in a state-machine style of computing under conditions where the messages transmitted by different senders are independent of one another, and hence the actions taken by recipients will commute. For example, suppose that sender *p* is reporting trades on a stock exchange and sender *q* is reporting bond pricing information. Although this information may be sent to the same destinations, it may or may not be combined in a way that is order sensitive. When the recipients are insensitive to the order of messages that originate in different senders, *fbcast* is a "strong enough" ordering to ensure that a state machine style of computing can safely be used. However, many applications are more sensitive to ordering than this, and the ordering properties of *fbcast* would not be sufficient to ensure that group members remain consistent with one another in such cases.

13.12.5.2 Causal Order

An obvious question to ask concerns the maximum amount of order that can be provided in a protocol that has the same cost as *fbcast*. At the beginning of this chapter, we discussed the causal ordering relation, which is the transitive closure of the message send/receive relation and the internal ordering associated with processes. Working with Joseph in 1985, this author developed a causally ordered protocol with cost similar to that of *fbcast* and showed how it could be used to implement replicated data. We named the protocol *cbcast*. Soon thereafter, Schmuck was able to show that causal order is a form of maximal ordering relation among fbcast-like protocols. More precisely, he showed that any ordering property that can be implemented using an asynchronous protocol can be represented as a subset of the causal ordering relationship. This proves that causally ordered communication is the most powerful protocol possible with cost similar to that of *fbcast*.

The basic idea of a causally ordered multicast is easy to express. Recall that a FIFO multicast is required to respect the order in which any single sender sent a sequence of multicasts. If process *p* sends m_0 and then later sends m_1, a FIFO multicast must deliver m_0 before m_1 at any overlapping destinations. The ordering rule for a causally ordered multicast is almost identical: if $send(m_0) \rightarrow send(m_1)$, then a causally ordered delivery will ensure that m_0 is delivered before m_1 at any overlapping destinations. In some sense, causal order is just a generalization of the FIFO sender order. For a FIFO order, we focus on event that happen in some order at a single place in the system. For the causal order, we relax this to events that are ordered under the "happens before" relationship, which can span multiple processes but is otherwise essentially the same as the sender-order for a single process. In English, a causally ordered multicast simply guarantees that *if m_0 is sent before m_1, then m_9 will be delivered before m_1 at destinations they have in common*.

The first time one encounters the notion of causally ordered delivery, it can be confusing because the definition doesn't look at all like a definition of FIFO ordered delivery. In fact, however, the concept is extremely similar. Most readers will be comfortable with the idea of a thread of control that moves from process to process as RPC is used by a client process to ask a server to take some action on its behalf. We can think of the thread of computation in the server as being part of the thread of the client. In some sense, a single "computation" spans two address spaces. Causally ordered multicasts are simply multicasts ordered along such a thread of computation. When this perspective is adopted one sees that FIFO ordering is in some ways the less natural concept: it "artificially" tracks ordering of events only when they occur in the same address space. If process *p* sends message m_0 and then asks process *q* to send message m_1 it seems natural to say that m_1 was sent after m_0. Causal ordering expresses this relation, but FIFO ordering only does so if *p* and *q* are in the same address space.

There are several ways to implement multicast delivery orderings that are consistent with the causal order. We will now present two such schemes, both based on adding a timestamp to the message header before it is initially transmitted. The first scheme uses a logical clock; the resulting change in header size is very small but the protocol itself has high latency. The second scheme uses a vector timestamp and achieves much better performance. Finally, we discuss several ways of compressing these timestamps to minimize the overhead associated with the ordering property.

13.12.5.2.1 Causal ordering with logical timestamps

Suppose that we are interested in preserving causal order within process groups, and in doing so only during periods when the membership of the group is fixed (the flush protocol that implements view synchrony makes this a reasonable goal). Finally, assume that all multicasts are sent to the full membership of the group. By attaching a logical timestamp to each message, maintained using Lamport's logical clock algorithm, we can ensure that if $SEND(m_1) \rightarrow SEND(m_2)$, then m_1 will be delivered before m_2 at overlapping destinations. The approach is extremely simple: upon receipt of a message m_i a process p_i waits until it knows that there are no messages still in the channels to it from other group members, p_j that could have a timestamp smaller than $LT(m_i)$.

How can p_i be sure of this? In a setting where process group members continuously emit multicasts, it suffices to wait long enough. Knowing that m_i will eventually reach every other group member, p_i can reason that eventually, every group member will increase its logical clock to a value at least as large as $LT(m_i)$, and will subsequently send out a message with that larger timestamp value. Since we are assuming that the communication channels in our system preserve FIFO ordering, as soon as any message has been received with a timestamp greater than or equal to that of m_i from a process p_j, all future messages from p_j will have a timestamp strictly greater than that of m_i. Thus, p_i can wait long enough to have the full set of messages that have timestamps less than or equal to $LT(m_i)$, then deliver the delayed messages in timestamp order. If two messages have the same timestamp, they must have been sent concurrently, and p_i can either deliver them in an arbitrary order, or can use some agreed-upon rule (for example, by breaking ties using the process-id of the sender, or its ranking in the group view) to obtain a total order. With this approach, it is no harder to deliver messages in an order that is causal and total than to do so in an order that is only causal.

Of course, in many (if not most) settings, some group members will send to the group frequently while others send rarely or participate only as message recipients. In such environments, p_i might wait in vain for a message from p_j , preventing the delivery of m_i. There are two obvious solutions to this problem: group members can be modified to send a periodic multicast simply to keep the channels active, or p_i can ping p_j when necessary, in this manner flushing the communication channel between them.

Although simple, this causal ordering protocol is too costly for most settings. A single multicast will trigger a wave of n^2 messages within the group, and a long delay may elapse before it is safe to deliver a multicast. For many applications, latency is the key factor that limits performance, and this protocol is a potentially slow one because incoming messages must be delayed until a suitable message is received on every other incoming channel. Moreover, the number of messages that must be delayed can be very large in a large group, creating potential buffering problems.

13.12.5.2.2 Causal ordering with vector timestamps

If we are willing to accept a higher overhead, the inclusion of a vector timestamp in each message permits the implementation of a much more accurate message delaying policy. Using the vector timestamp, we can delay an incoming message m_i precisely until any missing causally prior messages have been received. This algorithm, like the previous one, assumes that all messages are multicast to the full set of group members.

Again, the idea is simple. Each message is labeled with the vector timestamp of the sender as of the time when the message was sent. This timestamp is essentially a count of the number of causally prior messages that have been delivered to the application at the sender process, broken down by source. Thus, the vector timestamp for process p_1 might contain the sequence [13,0,7,6] for a group G with membership $\{p_0, p_1, p_2, p_3\}$ at the time it creates and multicasts m_i. Process p_1 will increment the counter for its own vector entry (here we assume that the vector entries are ordered in the same way as the processes in the group view), labeling the message with timestamp [13,1,7,6]. The meaning of such a timestamp is that this is the first message sent by p_1, but that it has received and delivered 13 messages from p_0, 7 from p_2, and 6 from p_3. Presumably, these received messages created a context within which m_i makes sense, and if some process delivers m_i without having seen one or more of them, it may run the risk of misinterpreting m_i. A causal ordering avoids such problems.

Now, suppose that process p_3 receives m_i. It is possible that m_i would be the very first message that p_3 has received up to this point in its execution. In this case, p_3 might have a vector timestamp as small as [0,0,0,6], reflecting only the six messages it sent before m_i was transmitted. Of course, the vector timestamp at p_3 could also be much larger: the only really upper limit is that the entry for p_1 is necessarily 0, since m_i is the first message sent by p_1. The delivery rule for a recipient such as p_3 is now clear: it should delay message m_i until both of the following conditions are satisfied:

1. Message m_i is the *next* message, in sequence, from its sender.
2. Every "causally prior" message has been received and delivered to the application.

We can translate rule 2 into the following formula:

If message m_i sent by process p_i is received by process p_j, then we delay m_i until, for each value of k different from i and j, $VT(p_j)[k] \geq VT(m_i)[k]$

Thus, if p_3 has not yet received any messages from p_0, it will not delivery m_i until it has received at least 13 messages from p_0. Figure 13-28 illustrates this rule in a simpler case, involving only two messages.

We need to convince ourselves that this rule really ensures that messages will be delivered in a causal order. To see this, it suffices to observe that when m_i was sent, the sender had already received and delivered the messages identified by *VT*(m_i). Since these are precisely the messages causally ordered before m_i, the protocol only delivers messages in an order consistent with causality.

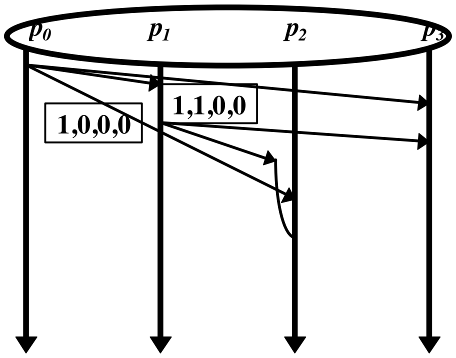


Figure 13-28: Upon receipt of a message with vector timestamp [1,1,0,0] from p_1, process p_2 detects that it is "too early" to deliver this message, and delays it until a message from p_0 has been received and delivered.

The causal ordering relationship is acyclic, hence one would be tempted to conclude that this protocol can never delay a message indefinitely. But in fact, it can do so if failures occur. Suppose that process p_0 crashes. Our flush protocol will now run, and the 13 messages that p_0 sent to p_1 will be retransmitted by p_1 on its behalf. But if p_1 also fails, we could have a situation in which m_i, sent by p_1 causally after having received 13 messages from p_0, will never be safely deliverable, because no record exists of one or more of these prior messages! The point here is that although the communication channels in the system are FIFO, p_1 is not expected to forward messages on behalf of other processes until a flush protocol starts because one or more processes have left or joined the system. Thus, a dual failure can leave a gap such that m_i is causally orphaned.

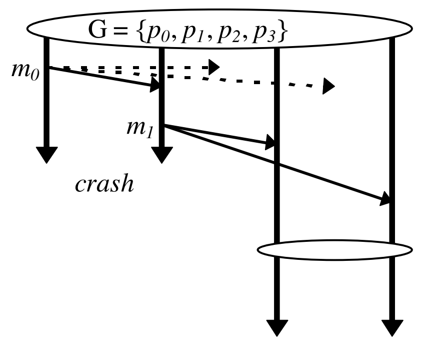


Figure 13-29: When processes p_0 and p_1 crash, message m_1 is causally orphaned. This would be detected during the flush protocol that installs the new group view. Although m_1 has been received by the surviving processes, it is not possible to deliver it while still satisfying the causal ordering constraint. However, this situation can only occur if the sender of the message is one of the failed processes. By discarding m_1 the system can avoid causal gaps. Surviving group members will never be logically partitioned (prevented from communicating with each other) in the sense that concerned us earlier.

The good news, however, is that this can only happen if the *sender of m_i* fails, as illustrated in Figure 13-29. Otherwise, the sender will have a buffered copy of any messages that it received and that are still unstable, and this information will be sufficient to fill in any causal gaps in the message history prior to when m_i was sent. Thus, our protocol can leave individual messages that are orphaned, but cannot partition group members away from one another in the sense that concerned us earlier.

Our system will eventually discover any such causal orphan when flushing the group prior to installing a new view that drops the sender of m_i. At this point, there are two options: m_i can be delivered to the application with some form of warning that it is an orphaned message preceded by missing causally prior messages, or m_i can simply be discarded. Either approach leaves the system in a self-consistent state, and surviving processes are never prevented from communicating with one another.

Causal ordering with vector timestamps is a very efficient way to obtain this delivery ordering property. The overhead is limited to the vector timestamp itself, and to the increased latency associated with executing the timestamp ordering algorithm and with delaying messages that genuinely arrive too early. Such situations are common if the machines involved are overloaded, channels are backlogged, or the network is congested and lossy, but otherwise would rarely be observed. In the best case, when none of these conditions is present, the causal ordering property can be assured with essentially no additional cost in latency or messages passed within the system! On the other hand, notice that the causal ordering obtained is definitely not a total ordering, as was the case in the algorithm based on logical timestamps. Here, we have a genuinely less costly ordering property, but it is also less ordered.

13.12.5.2.3 Timestamp compression

The major form of overhead associated with a vector-timestamp causality is that of the vectors themselves. This has stimulated interest in schemes for compressing the vector timestamp information transmitted in messages. Although an exhaustive treatment of this topic is well beyond the scope of the current textbook, there are some specific optimizations that are worth mentioning.

Suppose that a process sends a burst of multicasts — a common pattern in many applications. After the first vector timestamp, each subsequent message will contain a nearly identical timestamp, differing only in the timestamp associated with the sender itself, which will increment for each new multicast. In such a case, the algorithm could be modified to omit the timestamp: a missing timestamp would be interpreted as being "the previous timestamp, incremented in the sender's field only". This single optimization can eliminate most of the vector timestamp overhead seen in a system characterized by bursty communication! More accurately, what has happened here is that the sequence number used to implement the FIFO channel from source to destination makes the sender's own vector timestamp entry redundant. We can omit the vector timestamp because none of the other entries were changing and the sender's sequence number is represented elsewhere in the packets being transmitted.

An important case of this optimization arises if all the multicasts to some group are sent along a single causal path. For example, suppose that a group has some form of "token" that circulates within it, and only the token holder can initiate multicasts to the group. In this case, we can implement *cbcast* using a single sequence number: the *1'st cbcast,* the *2'nd,* and so forth. Later this form of *cbcast* will turn out to be important. Notice, however, that if there are concurrent multicasts from different senders (that is, if senders can transmit multicasts without waiting for the token), the optimization is no longer able to express the causal ordering relationships on messages sent within the group.

A second optimization is to reset the vector timestamp fields to zero each time the group changes its membership, and to sort the group members so that any passive receivers are listed last in the group view. With these steps, the vector timestamp for a message will tend to end in a series of zeros, corresponding to those processes that have not sent a message since the previous view change event. The vector timestamp can then be truncated: the reception of a short vector would imply that the missing fields are all zeros. Moreover, the numbers themselves will tend to stay smaller, and hence can be represented using shorter fields (if they threaten to overflow, a flush protocol can be run to reset them). Again, a single very simple optimization would be expected to greatly reduce overhead in typical systems that use this causal ordering scheme.

A third optimization involves sending only the difference vector, representing those fields that have changed since the previous message multicast by this sender. Such a vector would be more complex to represent (since we need to know which fields have changed and by how much), but much shorter (since, in a large system, one would expect few fields to change in any short period of time). This generalizes into a "run-length" encoding.

This third optimization can also be understood as an instance of an ordering scheme introduced originally in the Psync, Totem and Transis systems. Rather than representing messages by counters, a precedence relation is maintained for messages: a tree of the messages received and the causal relationships between them. When a message is sent, the leaves of the causal tree are transmitted. These leaves are a set of concurrent messages, all of which are causally prior to the message now being transmitted. Often, there will be very few such messages, because many groups would be expected to exhibit low levels of concurrency.

The receiver of a message will now delay it until those messages it lists as causally prior have been delivered. By transitivity, no message will be delivered until all the causally prior messages have been delivered. Moreover, the same scheme can be combined with one similar to the logical timestamp ordering scheme of the first causal multicast algorithm, to obtain a primitive that is both causally and totally ordered. However, doing so necessarily increases the latency of the protocol.

13.12.5.2.4 Causal multicast and consistent cuts

At the outset of this chapter we discussed notions of logical time, defining the causal relation and introducing, in Section 13.4, the definition of a consistent cut. Notice that the delivery events of a multicast protocol such as *cbcast* are concurrent and hence can be thought of as occurring "at the same time" in all the members of a process group. In a logical sense, *cbcast* delivers messages at what may look to the recipients like a single instant in time. Unfortunately, however, the delivery events for a single *cbcast* do not represent a consistent cut across the system, because communication that was concurrent with the *cbcast* could cross it. Thus one could easily encounter a system in which a *cbcast* is delivered at process *p* which has received message *m,* but where the same *cbcast* was delivered at process *q* (the eventual sender of *m*) before *m* had been transmitted.

With a second *cbcast* message, it actually possible to identify a true consistent cut, but to do so we need to either introduce a notion of an epoch number, or to inhibit communication briefly. The inhibition algorithm is easier to understand. It starts with a first *cbcast* message, which tells the recipients to inhibit the sending of new messages. The process group members receiving this message send back an acknowledgment to the process that initiated the *cbcast*. The initiator, having collected replies from all group members, now sends a second *cbcast* telling the group members that they can stop recording incoming messages and resume normal communication. It is easy to see that all messages that were in the communication channels when the first *cbcast* was received will now have been delivered and that the communication channels will be empty. The recipients now resume normal communication. (They should also monitor the state of the initiator, in case it fails!) The algorithm is very similar to the one for changing the membership of a process group, presented in Section 13.12.3.

Non-inhibitory algorithms for forming consistent cuts are also known. One way to solve this problem is to add *epoch numbers* to the multicasts in the system. Each process keeps an *epoch counter* and tags every message with the counter value. In the consistent cut protocol described above, the first phase message now tells processes to increment the epoch counters (and not to inhibit new messages). Thus, instead of delaying new messages, they are sent promptly but with epoch number *k+1* instead of epoch number *k*. The same algorithm described above now works to allow the system to reason about the consistent cut associated with its *k'th* epoch even as it exchanges new messages during epoch *k+1*. Another well known solution takes the form of what is called an *echo protocols* in which two messages traverse every communication link in the system [Chandy/Lamport]. For a system will all-to-all communication connectivity, such protocols will transmit $O(n^2)$ messages, in contrast with the $O(n)$ required for the inhibitory solution.

This *cbcast* provides a relatively inexpensive way of testing the distributed state of the system to detect a desired property. In particular, if the processes that receive a *cbcast* compute a predicate or write down some element of their states at the moment the message is received, these states will "fit together" cleanly and can be treated as a glimpse of the system as a whole at a single instant in time. For example, to count the number of processes for which some condition holds, it is sufficient to send a *cbcast* asking processes if the condition holds and to count the number that return *true*. The result is a value that could in fact have been valid for the group at a single instant in real-time. On the negative side, this guarantee only holds with respect to communication that uses causally ordered primitives. If processes communicate with other primitives, the delivery events of the *cbcast* will not necessarily be prefix-closed when the send and receive events for these messages are taken into account. Marzullo and Sabel have developed optimized versions of this algorithm.

Some examples of properties that could be checked using our consistent cut algorithm include the current holder of a token in a distributed locking algorithm (the token will never appear to be lost or duplicated), the current load on the processes in a group (the states of members will never be accidentally sampled at "different times" yielding an illusory load that is unrealistically high or low), the wait-for graph of a system subject to infrequent deadlocks (deadlock will never be detected when the system is in fact not deadlocked), or the contents of a database (the database will never be checked at a time when it has been updated at some locations but not others). On the other hand, because the basic algorithm inhibits the sending of new messages in the group, albeit briefly, there will be many systems for which the performance impact is too high and a solution that sends more messages but avoids inhibition states would be preferable. The epoch based scheme represents a reasonable alternative, but we have not treated fault-tolerance issues; in practice, such a scheme works best if all cuts are initiated by some single member of a group, such as the oldest process in it, and a group flush is known to occur if that process fails and some other takes over from it. We leave the details of this algorithm as a small problem for the reader.

13.12.5.2.5 Exploiting Topological Knowledge

Many networks have topological properties that can be exploited to optimize the representation of causal information within a process group that implements a protocol such as *cbcast*. Within the NavTech system, developed at INESC in Portugal, wide-area applications operate over a communications transport layer implemented as part of NavTech. This structure is programmed to know of the location of wide area network links and to make use of hardware multicast where possible [RVR93, RV95]. A consequence is that if a group is physically laid out with multiple subgroups interconnected over a wide area link, as seen in Figure 13-30.

In a geographically distributed system, it is frequently the case that all messages from some subset of the process group members will be relayed to the remaining members through a small number of relay points. Rodriguez exploits this observation to reduce the amount of information needed to represent causal ordering relationships within the process group. Suppose that message m_1 is causally dependent upon message m_0 and that both were sent over the same communications link. When these messages are relayed to processes on the other side of the link they will appear to have been "sent" by a single sender and hence the ordering relationship between them can be compressed into the form of a single vector-timestamp entry. In general, this observation permits any set of processes that route through a single point to be represented using a single sequence number on the other side of that point.

Stephenson explored the same question in a more general setting involving complex relationships between overlapping process groups (the "multi-group causality" problem) [Ste91]. His work identifies an optimization similar to this one, as well as others that take advantage of other regular "layouts" of overlapping groups, such as a series of groups organized into a tree or some other graph-like structure.

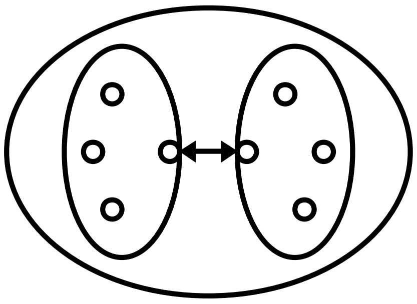


Figure 13-30: In a complex network, a single process group may be physically broken into multiple subgroups. With knowledge of the network topology, the NavTech system is able to reduce the information needed to implement causal ordering. Stephenson has looked at the equivalent problem in multigroup settings where independent process groups may overlap in arbitrary ways.

The reader may wonder about causal cycles, in which message m_2, sent on the "right" of a linkage point, becomes causally dependent on m_1, send on the "left", which was in turn dependent upon m_0, also sent on the left. Both Rodriguez and Stephenson made the observation that as m_2 is forwarded back through the link, it emerges with the old causal dependency upon m_1 reestablished. This method can be generalized to deal with cases where there are multiple links (overlap points) between the subgroups that implement a single process group in a complex environment.

13.12.5.3 Total Order

In developing our causally ordered communication primitive, we really ended up with a family of such primitives. Cheapest of these are purely causal in the sense that concurrently transmitted multicasts might be delivered in different orders to different members. The more costly ones combined causal order with mechanisms that resulted in a causal, total order. We saw two such primitives: one was the causal ordering algorithm based on logical timestamps, and the second (introduced very briefly) was the algorithm used for total order in the Totem and Transis systems, which extend the causal order into a total one using a canonical sorting procedure, but in which latency is increased by the need to wait until multicasts have been received from all potential sources of concurrent multicasts.[12] In this section we discuss totally ordered multicasts, known by the name *abcast* (for historical reasons), in more detail.

When causal ordering is not a specific requirement, there are some very simple ways to obtain total order. The most common of these is to use a sequencer process or token [CM84, Kaa92]. A sequencer process is a distinguished process that publishes an ordering on the messages of which it is aware; all other group members buffer multicasts until the ordering is known, and then deliver them in the appropriate order. A token is a way to move the sequencer around within a group: while holding the token, a process may put a sequence number on outgoing multicasts. Provided that the group only has a single token, the token ordering results in a total ordering for multicasts within the group. This approach was introduced in a very early protocol by Chang and Maxemchuck [CM84], and remains popular because of its simplicity and low overhead. Care must be taken, of course, to ensure that failures cannot cause the token to be lost, briefly duplicated, or result in gaps in the total ordering that orphan subsequent messages. We saw this solution above as an optimization to *cbcast* in the case where all the communication to a group originates along a single causal path within the group. From the perspective of the application, *cbcast* and *abcast* are indistinguishable in this case, which turns out to be a common and important one.

It is also possible to use the causally ordered multicast primitive to implement a causal and totally ordered token-based ordering scheme. Such a primitive would respect the delivery ordering property of *cbcast* when causally prior multicasts are pending in a group, and like *abcast* when two processes concurrently try to send a multicast. Rather than present this algorithm here, however, we defer it momentarily until Chapter 13.16, when we present it in the context of a method for implementing replicated data with locks on the data items. We do this because, in practice, token based total ordering algorithms are more common than the other methods. The most common use of causal ordering is in conjunction with the specific replication scheme presented in Chapter 13.16, hence it is more natural to treat the topic in that setting.

Yet an additional total ordering algorithm was introduced by Leslie Lamport in his very early work on logical time in distributed systems [Lam78b], and later adapted to group communication settings by Skeen during a period when he collaborated with this author on an early version of the Isis totally ordered communication primitive. The algorithm uses a two-phase protocol in which processes vote on the message ordering to use, expressing this vote as a logical timestamp.

[12] Most "ordered" of all is the flush protocol used to install new views: this delivers a type of message (the new view) in a way that is ordered with respect to all other types of messages . In the Isis Toolkit, there was actually a *gbcast* primitive that could be used to obtain this behavior at the desire of the user, but it was rarely used and more recent systems tend to use this protocol only to install new process group views.

The algorithm operates as follows. In a first phase of communication, the originator of the multicast (we'll call it the coordinator) sends the message to the members of the destination group. These processes save the message but do not yet deliver it to the application. Instead, each proposes a "delivery time" for the message using a logical clock, which is made unique by appending the process-id. The coordinator collects these proposed delivery times, sorts the vector, and designates the maximum time as the *committed* delivery time. It sends this time back to the participants. They update their logical clocks (and hence will never propose a smaller time) and reorder the messages in their pending queue. If a pending message has a committed delivery time, and the time is smallest among the proposed and committed times for other messages, it can be delivered to the application layer.

This solution can be seen to deliver messages in a total order, since all the processes base the delivery action on the same committed timestamp. It can be made fault-tolerant by electing a new coordinator if the original sender fails. One curious property of the algorithm, however, is that it has a non-uniform ordering guarantee. To see this, consider the case where a coordinator and a participant fail, and that participant also proposed the maximum timestamp value. The old coordinator may have committed a timestamp that could be used for delivery to the participant, but that will not be re-used by the remaining processes, which may therefore pick a different delivery order. Thus, just as dynamic uniformity is costly to achieve as an atomicity property, one sees that a dynamically uniform ordering property may be quite costly. It should be noted that dynamic uniformity and dynamically uniform ordering tend to go together: if delivery is delayed until it is known that all operational processes have a copy of a message, it is normally possibly to ensure that all processes will use identical delivery orderings

This two-phase ordering algorithm, and a protocol called the "born-order" protocol which was introduced by the Transis and Totem systems (messages are ordered using unique message identification numbers that are assigned when the messages are first created or "born"), have advantages in settings with multiple overlapping process groups, a topic to which we will return in Chapter 14. Both provide what is called "globally total order", which means that even *abcast* messages sent in different groups will be delivered in the same order at any overlapping destinations they may have.

The token based ordering algorithms provide "locally total order", which means that *abcast* messages sent in different groups may be received in different orders even at destinations that they share. This may seem to argue that one should use the globally total algorithms; such reasoning could be carried further to justify a decision to only consider gloablly total ordering schemes that also guarantee dynamic uniformity. However, this line of reasoning leads to more and more costly solutions. For most of the author's work, the token based algorithms have been adequate, and the author has never seen an application for which globally total dynamically uniform ordering was a requirement.

Unfortunately, the general rule seems to be that "stronger ordering is more costly". On the basis of the known protocols, the stronger ordering properties tend to require that more messages be exchanged within a group, and are subject to longer latencies before message delivery can be performed. We characterize this as unfortunate, because it suggests that in the effort to achieve greater efficiency, the designer of a reliable distributed system may be faced with a tradeoff between complexity and performance. Even more unfortunate is the discovery that the differences are extreme. When we look at Horus, we will find that its highest performance protocols (which include a locally total multicast that is non-uniform) are nearly *three orders of magnitude* faster than the best known dynamically uniform and globally total ordered protocols (measured in terms of latency between when a message is sent and when it is delivered).

By tailoring the choice of protocol to the specific needs of an application, far higher performance can be obtained. On the other hand, it is very appealing to use a single, very strong primitive system-wide in order to reduce the degree of domain-specific knowledge needed to arrive at a safe and correct implementation. The designer of a system in which multicasts are infrequent and far from the critical performance path should count him or herself as very fortunate indeed: such systems can be built on a strong, totally ordered, and hence dynamically uniform communication primitive, and the high cost will probably not be noticable. The rest of us are faced with a more challenging design problem.

13.13 Communication From Non-Members to a Group

Up to now, all of our protocols have focused on the case of group members communicating with one-another. However, in many systems there is an equally important need to provide reliable and ordered communication from non-members into a group. This section presents two solutions to the problem, one for a situation in which the non-member process has located a single member of the group but lacks detailed membership information about the remainder of the group, and one for the case of a non-member that nonetheless has cached group membership information.

In the first case, our algorithm will have the non-member process ask some group member to issue the multicast on its behalf, using an RPC for this purpose. In this approach, each such multicast is given a unique identifier by its originator, so that if the forwarding process fails before reporting on the outcome of the multicast, the same request can be reissued. The new forwarding process would check to see if the multicast was previously completed, issue it if not, and then return the outcome in either case. Various optimizations can then be introduced, so that a separate RPC will not be required for each multicast. The protocol is illustrated in Figure 13-31 for the normal case, when the contact process does not fail. Not shown is the eventual garbage collection phase needed to delete status information accumulated during the protocol and saved for use in the case where the contact eventually fails.

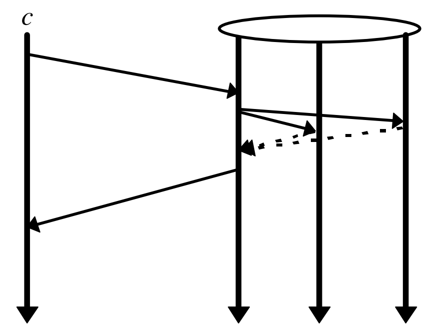


Figure 13-31: Non-member of a group uses a simple RPC-based protocol to request that a multicast be done on its behalf. Such a protocol becomes complex when ordering considerations are added, particularly because the forwarding process may fail during the protocol run.

Our second solution uses what is called an *iterated* approach, in which the non-member processes cache possibly inaccurate process group views. Specifically, each group view is given a unique identifier, and client processes use an RPC or some other mechanism to obtain a copy of the group view (for example, they may join a larger group within in which the group reports changes in its core membership to interested non-members). The client then includes the view identifier in its message and multicasts it directly to the group members. Again, the members will retain some limited history of prior interactions using a mechanism such as the one for the multiphase commit protocols.

There are now three cases that may arise. Such a multicast can arrive in the correct view, it can arrive partially in the correct view and partially "late" (after some members have installed a new group view), or it can arrive entirely late. In the first case, the protocol is considered successful. In the second case, the group flush algorithm will push the partially delivered multicast to a view-synchronous termination; when the late messages finally arrive, they will be ignored as duplicates by the group members that receive them, since these processes will have already delivered the message during the flush protocol. In the third case, all the group members will recognize the message as a late one that was not flushed by the system and all will reject it. Some or all should also send a message back to the non-member warning it that its message was not successfully delivered; the client can then retry its multicast with refreshed membership information. This last case is said to "iterate" the multicast. If it is practical to modify the underlying reliable transport protocol, a convenient way to return status information to the sender is by attaching it to the acknowledgment messages such protocols transmit.

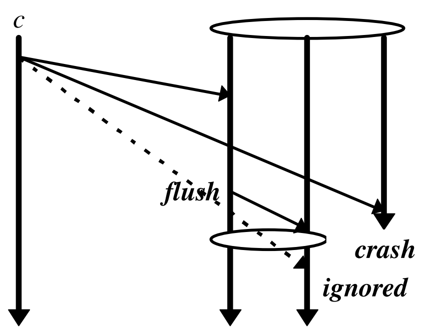


Figure 13-32: An iterated protocol. The client sends to the group as its membership is changing (to drop one member). Its multicast is terminated by the flush associated with the new view installation (message just prior to the new view), and when one of its messages arrives late (dashed line), the recipient detects it as a duplicate and ignores it. Had the multicast been so late that all the copies were rejected, the sender would have refreshed its estimate of group membership and retried the multicast. Doing this while also respecting ordering obligations can make the protocol complex, although the basic idea is quite simple. Notice that the protocol is cheaper than the RPC solution: the client sends directly to the actual group members, rather than indirectly sending through a proxy. However, while the figure may seem to suggest that there is no acknowledgment from the group to the client, this is not the case: the client communicates over a reliable FIFO channel to each member, hence acknowledgements are implicitly present. Indeed, some effort may be needed to avoid an implosion effect that would overwhelm the client of a large group with a huge number of acknowledgements.

This protocol is clearly quite simple, although its complexity grows when one considers the issues associated with preserving sender order or causality information in the case where iteration is required. To solve such a problem, a non-member that discovers itself to be using stale group view information should inhibit the transmission of new multicasts while refreshing the group view data. It should then retransmit, in the correct order, all multicasts that are not known to have been successfully delivered in while it was sending using the previous group view. Some care is required in this last step, however, because new members of the group may not have sufficient information to recognize and discard duplicate messages.

To overcome this problem, there are basically two options. The simplest case arises when the group members transfer information to joining processes that includes the record of multicasts successfully received from non-members prior to when the new member joined. Such a state transfer can be accomplished using a mechanism discussed in the next chapter. Knowing that the members will detect and discard duplicates, the non-member can safely retransmit any multicasts that are still pending, in the correct order, followed by any that may have been delayed while waiting to refresh the group membership. Such an approach minimizes the delay before normal communication is restored.

The second option is applicable when it is impractical to transfer state information to the joining member. In this case, the non-member will need to query the group, determining the status of pending multicasts by consulting with surviving members from the previous view. Having determined the precise set of multicasts that were "dropped" upon reception, the non-member can retransmit these messages and any buffered messages, and then resume normal communication. Such an approach is likely to have higher overhead than the first one, since the non-member (and there may be many of them) must query the group after each membership change. It would not be surprising if significant delays were introduced by such an algorithm.

13.13.1 Scalability

The preceeding discussion of techniques and costs did not address questions of scalability and limits. Yet clearly, the decision to make a communication protocol reliable will have an associated cost, which might be significant. We treat this topic in Section 18.8, and hence to avoid duplication of the material treated there, limit ourselves to a forward pointer here. However, the reader is cautioned to keep in mind that reliability does have a price, and hence that many of the most demanding distributed applications, which generate extremely high message-passing loads, must be "split" into a reliable subsystem that experiences lower loads and provides stronger guarantees for the subset of messages that pass through it, and a concurrently executed unreliable subsystem, which handles the bulk of communication but offers much weaker guarantees to its users. Reliability and properties can be extremely valuable, as we will see below and in the subsequent chapters, but one shouldn't make the mistake of assuming that reliability properties are *always* desirable or that such properties should be provided everywhere in a distributed system. Used selectively, these technologies are very powerful; used blindly, they may actually compromise reliability of the application by introducing undesired overheads and instability in those parts of the system that have strong performance requirements and weaker reliability requirements.

13.14 Communication from a Group to a Non-Member

The discussion of the preceding section did not consider the issues raised by transmission of replies from a group to a non-member. These replies, however, and other forms of communication outside of a group, raise many of the same reliability issues that motivated the ordering and gap-freedom protocols presented above. For example, suppose that a group is using a causally ordered multicast internally, and that one of its members sends a point-to-point message to some process outside the group. In a logical sense, that message may now be dependent upon the prior causal history of the group, and if that process now communicates with other members of the group, issues of causal ordering and freedom from causal gaps will arise.

This specific scenario was studied by Ladin and Liskov, who developed a system in which vector timestamps could be exported by a group to its clients; the client later presented the timestamp back to the group when issuing requests to other members, and in this way was protected against causal ordering violations. The protocol proposed in that work used stable storage to ensure that even if a failure occurred, no causal gaps could arise.

Other researchers have considered the same issues using different methods. Work by Schiper, for example, explored the use of a n x n matrix to encode point to point causality information [SES89], and the Isis Toolkit introduced mechanisms to preserve causal order when point to point communication was done in a system. We will present some of these methods below, in Chapter 13.16, and hence omit further discussion of them for the time being.

13.15 Summary

When we introduced the sender ordered multicast primitive, we noted that it is often called "fbcast" in systems that explicitly support it, the causally ordered multicast primitive "cbcast", and the totally ordered one, "abcast". These names are traditional ones, and are obviously somewhat at odds with terminology in this textbook. More natural names might be "fmcast", "cmcast" and "tmcast". However, a sufficiently large number of papers and systems have used the terminology of broadcasts, and have called the totally ordered primitive "atomic", that it would confuse many readers if we did not at least adopt the standard acronyms for these primitives.

The following table summarizes the most important terminology and primitives defined in this chapter.

Concept	Brief description
Process group	A set of processes that have joined the same group. The group has a *membership list* which is presented to group members in a data structure called the *process group view* which lists the members of the group and other information, such as their ranking.
View synchronous multicast	A way of sending a message to a process group such that all the group members that don't crash will receive the message between the same pair of group views. That is, a message is delivered entirely before or entirely after a given view of the group is delivered to the members. If a process sends a multicast when the group membership consists of $\{p_0, ... p_k\}$ and doesn't crash, the message will be delivered while the group view is still $\{p_0, ... p_k\}$.
safe multicast	A multicast having the property that if any group member delivers it, then all operational group members will also deliver it. This property is costly to guarantee and corresponds to a *dynamic uniformity* constraint. Most multicast primitives can be implemented in a safe or an unsafe version; the less costly one being preferable. In this text, we are somewhat hesitant to use the term "safe" because a protocol lacking this property is not necessarily "unsafe". Consequently, we will normally describe a protocol as being dynamically uniform (safe) or non-uniform (unsafe). If we do not specifically say that a protocol needs to be dynamically uniform, the reader should assume that we intend the non-uniform case.
fbcast	View-synchronous FIFO group communication. If the same process *p* sends m_1 prior to sending m_2 than processes that receive both messages deliver m_1 prior to m_2.
cbcast	View-synchronous causally ordered group communication. If $SEND(m_1) \rightarrow SEND(m_2)$, then processes that receive both messages deliver m_1 prior to m_2.
abcast	View-synchronous totally ordered group communication. If processes *p* and *q* both receive m_1 and m_2 then either both deliver m_1 prior to m_2, or both deliver m_2 prior to m_1. As noted earlier, *abcast* comes in several versions. Throughout the remainder of this text, we will assume that *abcast* is a locally total and non-dynamically uniform protocol. That is, we focus on the least costly of the possible *abcast* primitives, unless we specifically indicate otherwise.
cabcast	Causally and totally ordered group communication. The deliver order is as for abcast, but is also consistent with the causal sending order.
gbcast	A group communication primitive based upon the view-synchronous flush protocol. Supported as a user-callable API in the Isis Toolkit, but very costly and not widely used. *gbcast* delivers a message in a way that is totally ordered relative to all other communication in the same group.
gap freedom	The guarantee that if message m_i should be delivered before m_j and some process receives m_j and remains operational, m_i will also be delivered to its remaining destinations. A system that lacks this property can be exposed to a form of logical partitioning, where a process that has received m_j is prevented from (ever) communicating to some process that was supposed to receive m_i but will not because of a failure.
member	A process belonging to a process group

(of a group)	
group client	A non-member of a process group that communicates with it, and that may need to monitor the membership of that group as it changes dynamically over time.
virtual synchrony	A distributed communication system in which process groups are provided, supporting view-synchronous communication and gap-freedom, and in which algorithms are developed using a style of "closely synchronous" computing in which all group members see the same events in the same order, and consequently can closely coordinate their actions. Such synchronization becomes "virtual" when the ordering properties of the communication primitive are weakened in ways that do not change the correctness of the algorithm. By introducing such weaker orderings, a group can be made more likely to tolerate failure and can gain a significant performance improvement.

13.16 Related Readings

On logical notions of time: [Lam78b, Lam84]. Causal ordering in message delivery: [BJ87a, BJ87b]. Consistent cuts: [CL85, BM93]. Vector clocks: [Fid88, Mat89], used in message delivery: [SES89, BSS91, LLSG92]. Optimizing vector clock representations [Cha91, MM93], compression using topological information about groups of processes: [BSS91, RVR93, RV95]. Static groups and quorum replication: [Coo85, BHG87, BJ87a]. Two-phase commit: [Gra79, BHG87, GR93]. Three-phase commit: [Ske82b, Ske85]. Byzantine agreement: [Merxx, BE83, CASD85, COK86, CT90, Rab83, Sch84]. Asynchronous Consensus: [FLP85, CT91, CT92], but see also [BDM95, FKMBD95, GS96, Ric96]. The method of Chandra and Toueg: [CT91, CHT92, BDM95, Gue92, FKMB95, CHTC96]. Group membership: [BJ87a, BJ87b, Cri91b, MPS91, MSMA91, RB91, CHTC96], see also [Gol92, Ric92, Ric93, RVR93, Aga94, BDGB94, Rei94b, BG95, CS95, ACBM95, BDM95, FKMBD95, CHTC96, GS96, Ric96]. Partitionable membership [ADKM92b, MMA94]. Failstop illusion: [SM93]. Token based total order: [CM84, Kaa92]. Lamport's method: [Lam78b, BJ87b]. Communication from non-members ot a group: [BJ87b, Woo91]. Point-to-point causality [SES90].

14. Point-to-Point and Multigroup Considerations

Up to now, we have considered settings in which all communication occurs within a process group, and although we did discuss protocols by which a client can multicast into a group, we did not consider issues raised by replies from the group to the client. Primary among these is the question of preserving the causal order if a group member replies to a client, which we treat in Section 13.14. We then turn to issues involving multiple groups, including causal order, total order, causal and total ordering domains, and coordination of the view flush algorithms where more than one group is involved.

Even before starting to look at these topics, however, there arises a broader philosophical issue. When one develops an idea, such as the combination of "properties" with group communication, there is always a question concerning just how far one wants to take the resulting technology. Process groups, as treated in the previous chapter, are localized and self-contained entities. The directions treated in this chapter are concerned with extending this local model into an encompassing system-wide model. One can easily imagine a style of distributed system in which the fundamental communication abstraction was in fact the process group, with communication to a single process being viewed as a special case of the general one. In such a setting, one might well try and extend ordering properties so that they would apply system-wide, and in so doing, achieve an elegant and highly uniform programming abstraction.

There is a serious risk associated with this whole line of thinking, namely that it will result in system-wide costs and system-wide overhead, of a potentially unpredictable nature. Recall the end-to-end argument of Saltzer et. al. [SRC84]: in most systems, given a choice between paying a cost where and when it is needed, and paying that cost system-wide, one should favor the end-to-end solution, whereby the cost is incurred only when the associated property is desired. By and large, the techniques we present below should only be considered when there is a very clear and specific justification for using them. Any system that uses these methods casually is likely to perform poorly and to exhibit unpredictable.

14.1 Causal Communication Outside of a Process Group

Although there are sophisticated protocols in guaranteeing that causality will be respected for arbitrary communication patterns, the most practical solutions generally confine concurrency and associated causality issues to the interior of a process group. For example, at the end of Section 13.14, we briefly cited the replication protocol of Ladin and Liskov [LGGJ91, LLSG92]. This protocol transmits a timestamp to the client, and the client later includes the most recent of the timestamps it has received in any requests it issues to the group. The group members can detect causal ordering violations and delay such a request until causally prior multicasts have reached their destinations, as seen in Figure 14-1.

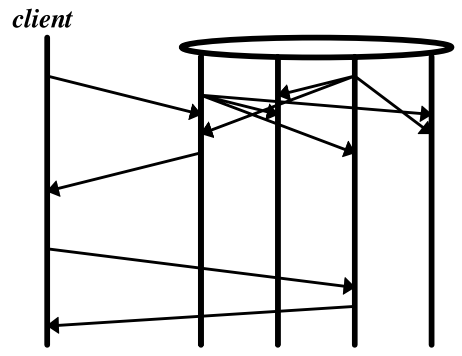


Figure 14-1: In the replication protocol used by Ladin and Liskov in the Harp system, vector timestamps are used to track causal multicasts within a server group. If a client interacts with a server in that group, it does so using a standard RPC protocol. However, the group timestamp is included with the reply, and can be presented with a subsequent request to the group. This permits the group members to detect missing prior multicasts and to appropriately delay a request, but doesn't go so far as to include the client's point-to-point messages in the causal state of the system. Such tradeoffs between properties and cost seem entirely appropriate, because an attempt to track causal order system-wide can result in significant overheads. Systems such as the Isis Toolkit, which enforce causal order even for point to point message passing, generally do so by delaying after sending point-to-point messages until they are known to be stable, a simple and conservative solution that avoids the need to "represent" ordering information for such messages.

An alternative is to simply delay messages sent out of a group until any causally prior multicasts sent within the group have become stable — have reached their destinations. Since there is no remaining causal ordering obligation in this case, the message need not carry causality information. Moreover, such an approach may not be as costly as it sounds, for the same reason that the *flush* protocol introduced earlier turns out not to be terribly costly in practice: most asynchronous *cbcast* or *fbcast* messages become stable shortly after they are issued, and long before any reply is sent to the client. Thus any latency is associated with the very last multicasts to have been initiated within the group, and will normally be small. We will see a similar phenomenon (in more detail) in Section 17.5, which discusses a replication protocol for stream protocols.

There has been some work on the use of causal order as a system-wide guarantee, applying to *point-to-point* communication as well as multicasts. Unfortunately, representing such ordering information requires a matrix of size O(n^2) in the size of the system. Moreover, this type of ordering information is only useful if messages are sent asynchronously (without waiting for replies). But, if this is done in systems that use point-to-point communication, there is no obvious way to recover if a message is lost (when its sender fails) after subsequent messages (to other destinations) have been delivered. Cheriton and Skeen discuss this form of all-out causal order in a well known paper and conclude that it is probably not desirable; this author agrees [SES89, CS93, Bir94, Coo94, Ren95]. If point-to-point messages are treated as being causally prior to other messages, it is best to wait until they have been received before sending causally dependent messages to other destinations.[13] (We'll have more to say about Cheriton and Skeen's paper in Chapter 16.)

[13] Notice that this issue doesn't arise for communication to the same destination as for the point-to-point message: one can send any number of point-to-point messages or "individual copies" of multicasts to a single process within a group without delaying. The requirement is that messages to *other* destinations be delayed, until these point-to-point messages are stable.

Early versions of the Isis Toolkit actually solved this problem without actually representing causal information at all, although later work replaced this scheme with one that waits for point-to-point messages to become stable [BJ87b, BSS91]. The approach was to piggyback pnding messages (those that are not known to have reached all their destinations) on *all* subsequent messages, regardless of their destination (Figure 14-2). That is, if process *p* has sent multicast m_1 to process group G and now wishes to send a message m_2 to any destination other than group G, a copy of m_1 is included with m_2. By applying this rule system-wide, *p* can be certain that if any route causes a message m_3, causally dependent upon m_1, to reach a destination of m_1, a copy of m_1 will be delivered too. A background garbage collection algorithm is used to delete these spare copies of messages when they do reach their destinations, and a simple duplicate supression scheme is employed to avoid delivering the same message more than once if it reaches a destination multiple times in the interim.

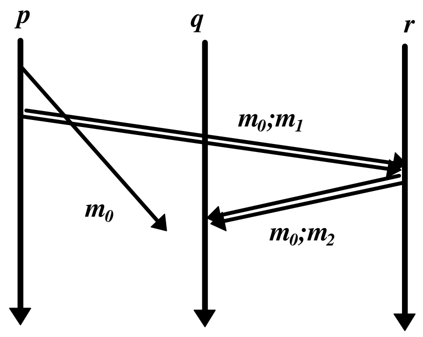


Figure 14-2: After sending m_0 asynchronously to q, p sends m_1 to r. To preserve causality, a copy of m_0 is piggybacked on this message, and similarly when r sends m_3 to q. This ensures that q will receive m_0 by the first causal path to reach it. A background garbage collection algorithm cleans up copies of messages that have become stable by reaching all of their destinations. To avoid excessive propogation of messages, the system always has the alternative of sending a message directly to its true destination and waiting for it to become stable, or simply waiting until the message reaches its destinations and becomes stable.

This scheme may seem wildly expensive, but in fact was rarely found to send a message more than once in applications that operate over Isis. One important reason for this was that Isis has other options available for use when the cost of piggybacking grew too high. For example, instead of sending m_0 piggybacked to some destination far from its true destination, *q*, any process can simply send m_0 to *q*, in this way making it stable. The system can also wait for stability to be detected by the original sender, at which point garbage collection will remove the obligation. Additionally, notice that m_0 only needs to be piggybacked once to any given destination. In Isis, which typically runs on a small set of servers, this meant that the worst case was just to piggyback the message once to each server. For all of thes reasons, the cost of piggybackbacking was never found to be extreme in Isis. The Isis algorithm also has the benefit of avoiding any potential gaps in the causal communication order: if *q* has received a message that was causally after m_1, then *q* will retain a copy of m_1 until m_1 is safe at its destinations.

Nonetheless, the author is not aware of any system that has used this approach other than Isis. Perhaps the strongest argument against the approach is that it has an *unpredictable* overhead: one can imagine patterns of communication for which its costs would be high, such as a client-server architecture in which the server replies to a high rate of incoming RPC's: in principle, each reply will carry copies of some large number of prior but unstable replies, and the garbage collection algorithm will have a great deal of work to do. Moreover, the actual overhead imposed on a given message is likely to vary depending on the amount of time since the garbage collection mechanism last was executed. Recent group communications systems, like Horus, seek to provide extremely predictable communication latency and bandwidth, and hence steer away from mechanisms that are difficult to analyze in any straightforward manner.

14.2 Extending Causal Order to Multigroup Settings

Additional issues arise when groups can overlap. Suppose that a process sends or receives multicasts in more than one group, a pattern that is commonly observed in complex systems that make heavy use of group computing. Just as we asked how causal order can be guaranteed when a causal path includes point-to-point messages, one can ask how causal and total order can be extended to apply to multicasts sent in a series of groups.

Consider first the issue of causal ordering. If process *p* belongs to groups g_1 and g_2, one can imagine a chain of multicasts that include messages sent asynchronously in both groups. For example, perhaps we will have $m_1 \rightarrow m_2 \rightarrow m_3$, where m_1 and m_3 are sent asynchronously in g_1 and m_2 in g_2. Upon receipt of a copy of m_3, a process may need to check for and detect causal ordering violations, delaying m_3 if necessary until m_1 has been received. In fact, this example illustrates two problems, because we also need to be sure that the delivery atomicity properties of the system extend to sequences of multicasts sent in different group. Otherwise, scenarios can arise whereby m_3 becomes causally orphaned and can never be delivered.

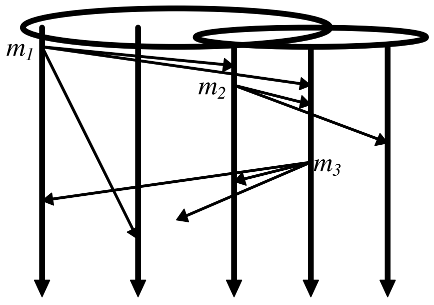


Figure 14-3: Message m_3 is causally ordered after m_1, and hence may need to be delayed upon reception.

In Figure 14-3, for example, if a failure causes m_1 to be lost, m_3 can never be delivered. There are several possibilities for solving the atomicity problem, which lead to different possibilities for dealing with causal order. A simple option is to delay a multicast to group g_2 while there are causally prior multicasts pending in group g_1. In the example, m_2 would be delayed until m_1 becomes stable. Most existing process group systems use this solution, which is called the *conservative* scheme. It is simple to implement and offers acceptable performance for most applications. To the degree that overhead is introduced, it occurs within the process group itself and hence is both localized and readily measured.

Less conservative schemes are both riskier in the sense that safety can be compromised when certain types of failures occur, that they require more overhead, and that this overhead is less localized and consequently harder to quantify. For example, a *k-stability* solution might wait until m_1 is known to have been received at $k+1$ destinations. The multicast will now be atomic provided that no more than k simultaneous failures occur in the group. However, we now need a way to detect causal ordering violations and to delay a message that arrives prematurely to overcome them.

One option is to annotate each multicast with multiple vector timestamps. The approach requires a form of piggybacking; each multicast carries with it only timestamps that have changed, or (if timestamp compression is used), only those that fields that have changed. Stephenson has explored this scheme and related ones, and shown that they offer general enforcement of causality at low average overhead. In practice, however, the author is not aware of any systems that implement this method, apparently because the conservative scheme is so simple and because of the risk of a safety violation if a failure in fact causes *k* processes to fail simultaneously.

Another option is to use the Isis style of piggybacking cbcast implementation. Early versions of the Isis Toolkit employed this approach, and as noted earlier; the associated overhead turns out to be fairly low. The details are essentially identical to the method presented in Section 14.1. This approach has the advantage of also providing atomicity, but the disadvantage of having unpredictable costs.

In summary, there are several possibilities for enforcing causal ordering in multigroup settings. One should ask whether the costs associated with doing so are reasonable ones to pay. The consensus of the community has tended to accept costs that are limited to within a single group (i.e. the conservative mode delays) but not costs that are paid system-wide (such as those associated with piggybacking vector timestamps or copies of messages). Even the conservative scheme, however, can be avoided if the application doesn't actually *need* the guarantee that this provides. Thus, the application designer should start with an analysis of the use and importance of multigroup causality before deciding to assume this property in a given setting.

14.3 Extending Total Order to Multigroup Settings

The total ordering protocols presented in Section 13.12.5.3 guarantee that messages sent in any one group will be totally ordered with respect to one-another. However, even if the conservative stability rule is used, this guarantee does not extend to messages sent in different groups but received at processes that belong to both. Moreover, the local versions of total ordering permit some surprising global ordering problems. Consider, for example, multicasts sent to a set of processes that form overlapping groups as shown in Figure 14-4. If one multicast is sent to each group, we could easily have process *p* receive m_1 followed by m_2, process *q* receive m_2 followed by m_3, process *r* receive m_3 followed by m_4, and process *s* receive m_1 followed by m_4. Since only a single multicast was sent in each group, such an order is total if only the perspective of the individual group is considered. Yet this ordering is clearly a cyclic one in a

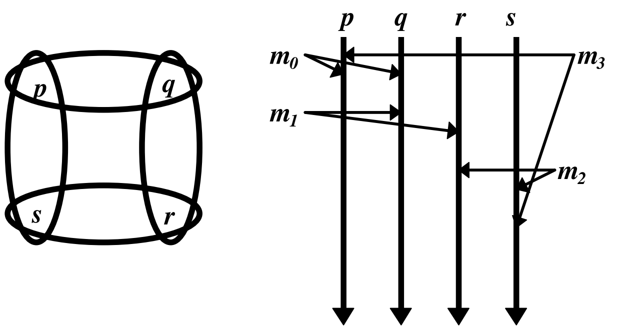


Figure 14-4: Overlapping process groups, seen from "above" and in a time-space diagram. Here, m_0 was sent to {p,q}, m_1 to {q,r} and so forth, and since each group received only one message, there is no ordering requirement within the individual groups. Thus an abcast protocol would never delay any of these messages. But one can deduce a global ordering for the multicasts: process p sees m_0 after m_3, q sees m_0 before m_1, r sees m_1 before m_2, and s sees m_2 before m_3. This global ordering thus cyclic, illustrating that many of our abcast ordering algorithms provide locally total ordering but not globally total ordering.

global sense.

A number of schemes for generating a globally acyclic total ordering are known, and indeed one could express qualms with the use of the term total for an ordering that now turns out to sometimes admit cycles. Perhaps it would be best to say that previously we identified a number of methods for obtaining *locally total* multicast ordering whereas now we consider the issue of *globally total* multicast ordering.

The essential feature of the globally total schemes is that the groups within which ordering is desired must share some resource that is used to obtain the ordering property. For example, if a set of groups shares the same ordering token, the ordering of messages assigned using the token can be made globally as well as locally total. Clearly, however, such a protocol could be costly, since the token will now be a single bottleneck for ordered multicast delivery.

In the Psync system an ordering scheme that uses multicast labels was first introduced [Pet87, PBS89]; soon after, variations of this were proposed by the Transis and Totem systems ADKM92a, MM89]. All of these methods work by using some form of unique label to place the multicasts in a total order determined by their labels. Before delivering such a multicast, a process must be sure it has received all other multicasts that could have smaller labels. The latency of this protocol is thus prone to rise with the number of processes in the aggregated membership of groups to which the receiving process belongs.

Each of these methods, and in fact all methods known to the author, have performance that degrades as a function of scale. The larger the set of processes over which a total ordering property will apply, the more costly the ordering protocol. When deciding if globally total ordering is warranted, it is therefore useful to ask what sort of applications might be expected to notice the cycles that a local ordering protocol would allow. The reasoning is that if a cheaper protocol is still adequate for the purposes of the application, most developers would favor the cheaper protocol. In the case of globally total ordering, few applications that really need this property are known.

Indeed, the following may be the only widely cited example of a problem for which locally total order is inadequate and globally total order is consequently needed. Suppose that we wish to solve the Dining Philosopher's problem. In this problem, which is a classical synchronization problem well known to the distributed systems community, a collection of philosophers gather around a table. Between each pair of philosophers is a single shared fork, and at the center of the table is a plate of pasta. To eat, a philosopher must have one fork in each hand. The life of a philosopher is an infinite repetition of the sequence: *think, pick up forks, eat, put down forks*. Our challenge is to implement a protocol solving this problem that avoids deadlock.

Suppose that the processes in our example are the forks, and that the multicasts originate in philosopher processes that are arrayed around the table. The philosophers can now request their forks by sending totally ordered multicasts to the process group of forks to their left and right. It is easy to see that if forks are granted in the order that the requests arrive, a globally total order avoids deadlock, but a locally total order is deadlock prone. Presumably, there is a family of multi-group locking and synchronization protocols for which similar results would hold. However, to repeat the point made above, this author has never encountered a real-world application in which globally total order is needed. This being the case, such strong ordering should perhaps be held in reserve as an option for applications that specifically request it, but not a default. If globally total order were as cheap as locally total order, of course; the conclusion would be reversed.

14.4 Causal and Total Ordering Domains

We have seen that when ordering properties are extended to apply to multiple heavyweight groups, the costs of achieving ordering can rise substantially. Sometimes, however, such properties really are needed, at least in subsets of an application. If this occurs, one option may be to provide the application with control over these costs by introducing what are called *causal and total ordering domains*. Such a domain would be an attribute of a process group: at the time a group is created, it would be bound to an ordering domain identifier, which remains constant thereafter. We can then implement the rule that when two groups are in different domains, multicast ordering properties need not hold across them. For example, if group g_1 and group g_2 are members of different ordering domains, the system could ignore causal ordering between multicasts sent in g_1 and multicasts sent in g_2. More general still would be a scheme in which a domain is provided for each type of ordering: two groups could then be in the same causal domain but different total ordering domains, for example. Implementation of ordering domains is trivial if the corresponding multigroup ordering property is available within a system. For example, if group g_1 and group g_2 are members of different causal ordering domains, the conservative rule would be overlooked when a process switched from sending or receiving in one group and starts to do send in the other. Delays would only arise when two groups are explicitly placed in the same ordering domain, presumably because the application actually requires multigroup ordering in this case.

It can be argued that the benefits associated with preserving causal order system-wide are significantly greater than those for supporting globally total order. The reasoning is that causal order is needed to implement asynchronous data replication algorithms, and that these have such a large performance advantage over other schemes that the benefits outweigh the costs of now needing to enforce causal order across group boundaries. However, the conservative causality scheme is an adequate solution to this particular problem, and has the benefit of *providing a system-wide guarantee with a local method*. When combined with causal domains, such a mechanism has a highly selective cost. This said, however, it should also be noted that the *flush* primitive proposed earlier offers the same benefits and is quite easy to use. Thus, many real systems opt for causal ordering, do not delay when sending messages outside of a group, and provide a *flush* primitive for use by the application itself when causal ordering is needed over group boundaries. Such a compromise is visible to the user, but easily understood.

Similar reasoning seems to argue against globally total order: the primitive has a significant cost (mostly in terms of latency) and limited benefit. Thus, the author's work has ceased to provide this property, after initially doing so in the early versions of the Isis Toolkit. The costs were simply too high to make globally total ordering the default, and the complexity of supporting a very rarely used mechanism then argued against having the property at all.

14.5 Multicasts to Multiple Groups

An additional multigroup issue concerns the sending of a single multicast to a set of process groups in a single atomic operation. Up to now, such an action would require that the multicast be sent to one group at a time, raising issues of non-atomic delivery if the sender fails midway. One can imagine solving this problem by implementing a multigroup multicast as a form of non-blocking commit protocol; Schiper and Raynal have proposed such a protocol in conjunction with their work on the Phoenix system [SR96]. However, there is another option, which is to create a new process group superimposed on the multiple destination groups, and to send the multicast in that group. Interestingly, the best implementations of a group create protocol require a single *fbcast*, hence if one creates a group, issues a single multicast in it, and then deletes the group, this has comparable cost to doing a multi-phase commit over the same set of processes and then garbage collecting after the protocol has terminated!

This last observation argues against explicit support for sending a multicast to several groups at the same time, except in settings where the set of groups to be used cannot be predicted in advance and is very unlikely to be "reused" for subsequently communication. That is, although the application process can be presented with an interface that allows multicasts to be sent to sets of groups, it may be best to implement such a mechanism by creating a group in the manner described above. On the belief that most group communication patterns will be reused shortly after they are first employed, such a group could then be retained for a period of time in the hope that a subsequent multicast to the same destinations will reuse its membership information. The group can then be torn down after some period during which no new multicasts are transmitted. Only if such a scheme is impractical would one need a multicast primitive capable of sending to many groups at the same time, and the author of this text is not familiar with any setting in which such a scheme is clearly not viable.

14.6 Multigroup View Management Protocols

A final issue that arises in systems where groups overlap heavily is that our view management and flush protocol will run once for each group when a failure or join occurs, and our state transfer protocol only handles the case of a process that joins a single group at a time. Clearly, these will be sources of inefficiency (in the first case) and inconvenience (in the second case) if group overlap is common. This observation, combined with the delays associated with conservative ordering algorithms and the concerns raised above in regard to globally total order, has motivated research on ways of collapsing heavily overlapped groups into smaller numbers of larger groups. Such approaches are often described as resulting in *lightweight* groups, because the groups seen by the application typically map onto some enclosing set of *heavyweight* groups.

Glade has explored this approach in Isis and Horus [GBCS92]. His work supports the same process group interfaces as for a normal process group, but maps multicasts on lightweight groups into multicasts to the enclosing heavyweight groups. Such multicasts are filtered on arrival, so that an individual process will only be given copies of messages actually destined to for it. The approach essentially maps the fine-grained membership of the lightweight groups to a coarser-grained membership in a much smaller number of heavyweight groups.

The benefit of Glade's approach is that it avoids both the costs of maintaining large numbers of groups (the membership protocols run just once if a process joins or leaves the system, updating multiple lightweight groups in one operation. Moreover, the causal and total ordering guarantees of our single-group solutions will now give the illusion of multigroup causal and total ordering, with no changes to the protocols themselves. Glade argues that when a system produces very large numbers of overlapping process groups there are likely to be underlying patterns that can be exploited to efficiently map the groups to a small number of heavyweight ones. In applications with which the author is familiar, Glade's point seems to hold. Object oriented uses of Isis (for example, Orbix+Isis) can generate substantial overlap when a single application uses multiple object groups. But this is also precisely the case where a heavyweight group will turn out to have the least overhead, because it precisely matches the membership of the lightweight groups.

Glade's algorithms in support of lightweight process groups are relatively simple. A multicast to such a group is, as noted before, mapped to a multicast to the corresponding heavyweight group and filtered on arrival. Membership changes can be implemented either by using an *abcast* to the lightweight group or, in more extreme cases, by using a flushed multicast, similar to the one used to install view changes in the heavyweight group. For most purposes, the *abcast* solution is sufficient.

14.7 Related Reading

On the timestamp technique used in Harp: [LGGJ91, LLSG92]. Preserving causality in point-to-point message passing systems [SES90]. On the associated controversy [CS93] and the responses [Bir94, Coo94, Ren94]. Multiple groups in Isis: [BJ87b, BSS81] . On communication from a non-member of a group to a group: [Woo93, BJ87b]. Graph representations of message dependencies [Pet87, PBS89, ADKM92a, MM89]. Lightweight process groups [GBCS92].

15. The Virtually Synchronous Execution Model

The process group communication primitives introduced in the previous chapters create a powerful framework for algorithmic development. When the properties and primitives are combined for this purpose, we will say that a *virtually synchronous* execution environment results [BJ87a, BJ87b, BR94]. However, although we built our primitives "up" from basic message passing, it is probably easier to understand the idea behind virtual synchrony in a top-down treatment. We'll then use the approach to develop an extremely high performance replicated data algorithm, as well as several other tools for consistent distributed computing.

15.1 Virtual Synchrony

Suppose that we wish to use a process group (or a set of process groups) as a building block in a distributed application. The group members will join that group for the purpose of cooperation, perhaps to replicate data or to perform some operation fault-tolerantly. The issue now arises of how to design such algorithms with a high degree of confidence that they will operate correctly.

Recall the discussion of transactional serializability from Section 7.5. In that context, we encountered a similar problem: a set of concurrently executed programs that share files or a database and wish to avoid interference with one-another. The basic idea was to allow the developer to code these applications as if they would run in isolation from one-another, one by one. The database itself is permitted to interleave operations for greater efficiency, but only in ways that preserve the illusion that each "transaction" executes without interruption. The results of a transaction are visible only after it commits; a transaction that aborts is automatically and completely erased from the memory of the system. As we noted at the time, transactional serializability allows the developer to use a simple programming model, while offering the system an opportunity to benefit from high levels of concurrency and asynchronous communication.

Virtual synchrony is not based on transactions, but introduces a similar approach to programming with process groups. In the virtual synchrony model, the simplifying abstraction seen by the developer is that of a set of processes (the group members) which *all see the same events in the same order*. These events are incoming messages to the group and group membership changes. The key insight, which is not a very deep one, is that since all the processes see the same inputs, they can execute the same algorithm and in this manner stay in consistent states. This is seen in Figure 15-1, which illustrates a process group that receives messages from several non-members, has a new member join and transfers the "state" of the group to it, and then experiences a failure of one of the old members. Notice that the group members see identical sequences of events while they belong to the group. The members differ, however, in their relative "ranking" within the group membership. There are many possible ways to rank the members of a group, but the most common one, which is used in this Chapter, assumes that the rank is based on when members joined — the oldest member having the lowest ranking, and so forth.

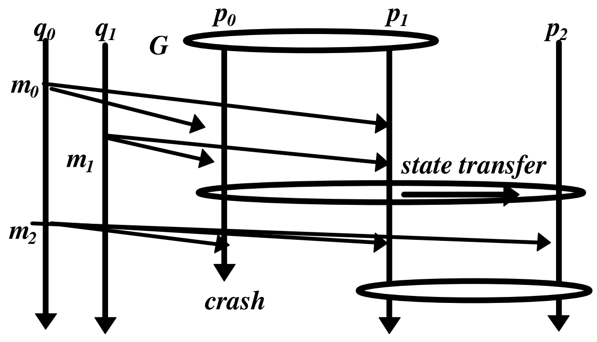


Figure 15-1: Closely synchronous execution: all group members see the same events (incoming messages and new group views) in the same order. In this example only the non-members of the group send multicasts to it, but the group members can also multicast to one-another, and can send point-to-point messages to the non-members. For example, a group RPC could be performed by sending a multicast to the group, to which one or more members reply.

The State Machine approach of Lamport and Schneider first introduced this approach as part of a proposal for replicating objects in settings subject to Byzantine failures [Sch88b, Sch90]. Their work made a group of identical replicas of the object in question, and used Byzantine Agreement for all interactions with the group and to implement its interactions with the external world. However, the State Machine approach saw little use when it was first proposed for the same reason that Byzantine Agreement sees little practical use: few computing environments satisfy the necessary synchronous computing and communications requirements, and it is difficult to employ a service that employs a Byzantine fault model without extending the same approach to other aspects of the environment, such as any external objects with which the service interactions, and the operating systems software used to implement the communication layer.

A further concern about the State Machine approach is that all copies of the program see identical inputs in the identical order. If one program crashes because of a software bug, so will all of the replicas of that program. Unfortunately, as we saw earlier, studies of real-world computing installations reveal that even in mature software systems, bugs that crash the application remain a proportionately important cause of failures. Thus, by requiring correct processes that operate deterministically and in lock-step, the State Machine approach is unable to offer protection against software faults.

Virtual synchrony is similar to the State Machine abstraction, but moves outside of the original Byzantine setting, while also introducing optimizations that overcome the concerns raised above. The idea is to view the State Machine as a sort of reference model but to allow a variety of optimizations so that the true execution of a process group may be very far from lock-step synchrony. In effect, just as transactional executions allow operations to be interleaved, provided that the behavior is indistinguishable from some serial execution, a virtual synchrony execution allows operations to be interleaved, provided that the result is indistinguishable from some closely synchronous (State Machine) execution. Nonetheless, the executions of the different replicas will be different enough to offer some hope that software bugs will not be propagated across the entire group.

To take a very simple example, suppose that we wish to support a process group whose members replicate some form of database and perform load-balanced queries upon it. The operations on the service will be queries and updates, and we will overlook failures (for the time being) to keep the problem as simple as possible.

Next, suppose that we implement both queries and database updates as totally ordered multicasts to the group membership. Every member will have the same view of the membership of the group, and each will see the same updates and queries in the same order. By simply applying the updates in the order they were received, the members can maintain identically replicated copies of the database. As for the queries, an approximation of load-balancing can be had using the ranking of processes in the group view.

Suppose that the process-group view ranks the members in [0...*n-1*]. Then the *i'th* incoming query can be assigned to the process whose rank is *(i mod n)*. Each query will be handled by exactly one process.

We'll call this a *closely synchronous* execution. Frank Schmuck was the first to propose this term, observing that the actions of the receiving processes were closely synchronized but might be spread over a significant period of real-time. The synchronous model, as discussed previously in this text, would normally require real-time bounds on the time period over which an action is performed by the different processes that perform it. Notice that a closely synchronous execution does not require identical actions by identical processes: if we use the load-balancing idea outlined above, actions will be quite different at the different copies. Thus, a closely synchronous group is similar to a group that uses State Machine replication, but not identical.

Having developed this solution, however, there will often be ways to weaken the ordering properties of the protocols it uses. For example, it may be the case that updates are only initiated by a single source, in which case an *fbcast* protocol would be sufficient to provide the desired ordering. Updates will no longer be ordered with respect to queries if such a change is made, but in an application where a single process issues an update and follows it with a query, the update would always be received first and hence the query will reflect the result of doing the update. In a slightly fancier setting, *cbcast* might be needed to ensure that the algorithm will operate correctly. For example, with *cbcast* one would know that if an application issues an update and then tells some other process to do a query, that second process will see the effects of the causally prior updates. Often, an analysis such as this one can be carried very far.

Having substituted *fbcast* or *cbcast* for the original *abcast*, however, the execution will no longer be closely synchronous, since different processes may see different sequences of updates and queries and hence perform the same actions but in different orders. The significant point is that if the original analysis was performed correctly, the actions will *produce an effect indistinguishable from that which might have resulted from a closely synchronous execution*. Thus, the execution "looks" closely synchronous, even though it is not. It is *virtually synchronous* in much the same sense that a transactional system creates the illusion of a serial execution even though the database server is interleaving operations from different transactions to increase concurrency.

Our transformation has the advantage of delivering inputs to the process group members in different orders, at least some of the time. Moreover, as we saw earlier, the process groups themselves are dynamically constructed, with processes joining them at different times. And, the ranking of the processes within the group differs. Thus, there is substantial room for processes to execute in slightly different ways, affording a degree of protection against software bugs that crash some of the members.

Recall the Gray/Lindsey characterization of Bohrbugs and Heisenbugs, from Chapter 12. It is interesting to realize that virtually synchronous replication can protect against Heisenbugs [BR94, BR96]. If a replica crashes because such a bug has been exercised, the probability that other group members will crash simultaneously is reduced by the many aspects of the execution that differ from replica to replica. Our transformation from a closely synchronous system to a virtually synchronous one increases the natural resiliency of the group, assuming that its constituent members are mature, well debugged code. Nonetheless, some exposure to correlated failures is unavoidable, and the designer of a critical system should keep this in mind.

Additionally, notice that the *cbcast* primitive can be used asynchronously. That is, there is no good reason for a process that issues a *cbcast* to perform an update to wait until the update has been completed by the full membership of the group. The properties of the *cbcast* protocol ensure that these asynchronously transmitted messages will reliably reach their destinations, and that any causally subsequent actions by the same or different processes will see the effects of the prior *cbcast's*. In an intuitive sense, one could say that these *cbcast* protocols look as if they were performed instantly, even when they actually execute over an extended period of time.

In practice, the most common transformation that we will make is precisely this one: the replacement of a totally ordered *abcast* primitive with an asynchronous, causally ordered *cbcast* primitive.

In the subsections that follow, this pattern will occur repeatedly. Such transformations have been studied by Schmuck in his doctoral dissertation, and shown to be extremely general: what we do in the subsections below can be done in many settings.

Thus, we have transformed a closely synchronous group application, in which the members operate largely in lock-step, into a very asynchronous implementation in which the members can pull ahead and others can lag behind, in which communication can occur concurrently with other execution, and in which the group may be able to tolerate software bugs that crash some of its members! These are important advantages that account for the appeal of the approach.

When an application has multiple process groups in it, an additional level of analysis is often required. As we saw in the previous chapter, multigroup causal (and total) ordering is expensive. When one considers real systems, it also turns out that multigroup ordering is often unecessary: many applications that need multiple groups use them for purposes that are fairly independent of one-another. Operations on such independent groups can be thought of as commutative, and hence it may be possible to use *cbcast* to optimize such groups independently without taking the next step of enforcing causal orderings across groups. Where multi-group ordering is needed, it will often be confined to small sets of groups, which can be treated as an ordering domain. In this manner, an all-around solution results that can scale to large numbers of groups while still preserving the benefits of the one-round asynchronous communication pattern seen in the *cbcast* protocol, and absent in the *abcast* protocols.

Our overall approach, it should be noted, is considerably less effective when the dynamic uniformity guarantees of the "safe" multicast protocols are required. The problem is that whereas asynchronous *cbcast* is a very fast protocol that delivers messages during its first phase of communication, any dynamically uniform protocol will delay delivery until a second phase. The benefit of replacing *abcast* with *cbcast* in such a case is lost. Thus, one begins to see a major split between the algorithms that run fairly synchronously, requiring more than a single phase of message passing before delivery can occur, and those that operate asynchronously, allowing the sender of a multicast to continue computing while multicasts that update the remainder of a group or that inform the remainder of the system of some event propagate concurrently to their destinations.

The following is a summary of the key elements of the virtual synchrony model:

- *Support for process groups*. Processes can join groups dynamically, and are automatically excluded from a group if they crash.
- *Identical process group views and mutually consistent rankings*. Members of a process group are presented with identical sequences of group membership, which we call *views* of that process group. If a non-primary component of the system forns after a failure, any process group views reported to processes in that component are identified as non-primary, and the view sequence properties will otherwise hold for all the processes in a given components. The view ranks the components, and all group members see identical rankings for identical group views.
- *State transfer to the joining process*. A process that joins a group can obtain the group's current state from some prior member, or from a set of members.
- *A family of reliable, ordered multicast protocols*. We have seen a number of these, including *fbcast, cbcast, abcast, cabcast,* the *safe* (dynamically uniform) versions of these, and the group flush protocol, which is sometimes given the name *gbcast*.
- *Gap-freedom guarantees*. After a failure, if some message m_j is delivered to its destinations, than any message m_i that the system is obliged to deliver prior to m_j will also have been delivered to its destinations.

- *View synchronous multicast delivery*. Any pair of processes that are both members of two consecutive group views receive the same set of multicasts during the period between those views.[14]
- *Use of asynchronous, causal or fifo multicast*. Although algorithms will often be developed using a closely synchronous computing model, a systematic effort is made to replace synchronous, totally ordered, and dynamically uniform (safe) multicasts with less costly alternatives, notably the asynchronous *cbcast* primitive in its non-dynamically-uniform (unsafe) mode.

15.2 Extended Virtual Synchrony

As presented above, the virtual synchrony model is inherently intolerant of partitioning failures: the model is defined in terms of a single "system" component within which process groups reside. In this primary component approach, if a network partitioning failure occurs and splits a group into fragments, only the fragment that resides in the primary component of the system is able to continue operation. Fragments that find themselves in the non-primary component(s) of the system are typically forced to shut down, and the processes within them must reconnect to the primary component when communication is restored.

The basis of the primary component approach lies in a subtle issue that we first saw when discussing commit protocols. In a dynamic distributed environment there can be symmetric failure modes that result from communication problems that mimic process failures. In such a situation perhaps process *p* will consider that process *q* has failed while process *q* believes the converse to be true. To make progress, one or the other (or perhaps both) of these events must become "official". In a partitioned run of the system, only one of these conflicting states can become official.

At the core of the problem is that observation that, if a system experiences a partitioning failure, it is impossible to guarantee that multiple components can remain operational (in the sense of initiating new actions, delivering messages and new group views, etc) with guarantees that also span both sides of the partition. To obtain strong system-wide guarantees a protocol *must always wait for communication to be reestablished under at least some executions in at least one side of the partition*. When we resolve this problem in the manner of the protocols of the previous chapters, the primary component is permitted to make progress at the expense of inconsistency relative to other components: within them, the sets of messages delivered may be different from the set in the primary component, and the order may also be different. In the case of the dynamically uniform protocols the guarantees are stronger but non-primary components may be left in a state where some dynamically uniform multicasts are still undelivered and where new dynamically uniform ones are completely blocked. The primary component, in contrast, can make progress so long as its GMS protocol is able to make progress.

Some researchers, notably in the Transis and Totem projects, have pointed out that there are applications that can tolerate inconsistency of the sort that would potentially arise if progress was permitted in a non-primary component of a partitioned system [Mal94, Aga94, DMS95]. In these systems, any component that can reach internal agreement on its membership is permitted to continue operation. However, only a single component of the system is designated as the primary one. An application that is safe only in the primary component would simply shut down in non-primary components. Other applications, however, might continue to be available in non-primary components, merging their states back into the primary component when the partitioning failure ends.

Carrying this observation even further, the Transis group has shown that there are distributed systems in which no component ever can be identified as the primary one, and yet every action initiated

[14] In some systems this is weakened so that if a process fails but its failure is not reported promptly, it is considered to have received multicasts that would have been delivered to it had it still been operational.

within the system can eventually be performed in a globally consistent manner [KD95, DMS95]. However, this work involves both a static system model and a very costly protocol, which delays peforming an action until a majority of the processes in the system as a whole have acknowledge receipt of it. The idea is that actions can be initiated within dynamically defined components that represent subsets of the true "maximal" system configuration, but that they remain in a pending state until a sufficient number of processes are known to have seen them, which occurs when communication is restored between components. Eventually, knowledge of the actions reaches enough processes so that it becomes safe to perform them. But the protocol is clearly intended for systems that operate in a partitioned mode over very long periods of time, and where there is no special hurry to perform actions. Yair Amir has extended this approach to deal with more urgent actions, but his approach involves weakening the global consistency properties [Ami95]. Thus, one is faced with a basic tradeoff between ensuring that actions will occur quickly and providing consistency between the primary component of the system and other components. We can have one or the other, but not both at once.

Although the author's own system, Horus, supports an extended model of the former sort [Mal94]. (In fact, this part of Horus was actually implemented by Malki, who ported the associated code from Transis into Horus). However, it is quite a bit harder to work with than the primary partition model. The merge of states when an arbitrary application resumes contact between a non-primary and a primary component can very difficult and cannot, in general, be solved automatically. In practice, such an application would normally remember any "update" actions it has taken and save these on a queue. When a merge becomes possible it would replace its state with that of the primary component and then reapply these updates, if any. But it is not clear how large a class of applications can operate this way. Moreover, unless dynamically uniform protocols are employed for updates, the non-primary component's state may be inconsistent with the primary one in significant ways.

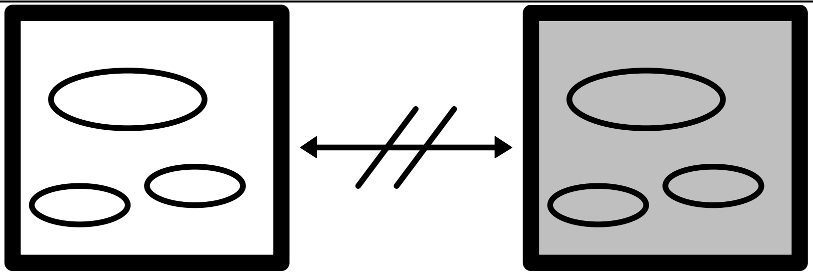


Figure 15-2: When a partitioning failure occurs, an application may be split into two or more fragments, each complete in the sense that it may potentially have a full set of processes and groups. In the primary component model, however, only one "set" is permitted to remain operational — hopefully one that has a full complement of process and groups. Above, the white component might thus be alive after the link breaks while the members of the gray component are prevented from continuing execution. The rationale underlying this model is that it is impossible to guarantee consistency if both sides of a partitioning failure are also permitted to remain available while a communication failure is pending. Thus we could allow both to run if we sacrifice consistency, but then we face a number of hard problems: which side "owns" critical resources? How can the two sides overcome potential inconsistencies in their states as of the time of the partition failure event? There are no good general answers to these questions.

On the other hand, the primary component model is awkward in wide-area networks where partitioning events occur frequently. Here, the model will in effect shut down parts of the overall system that are physically remote from the "main" part of the system. Each time they manage to restart after a communication failure, a new communication problem will soon cut them off again.

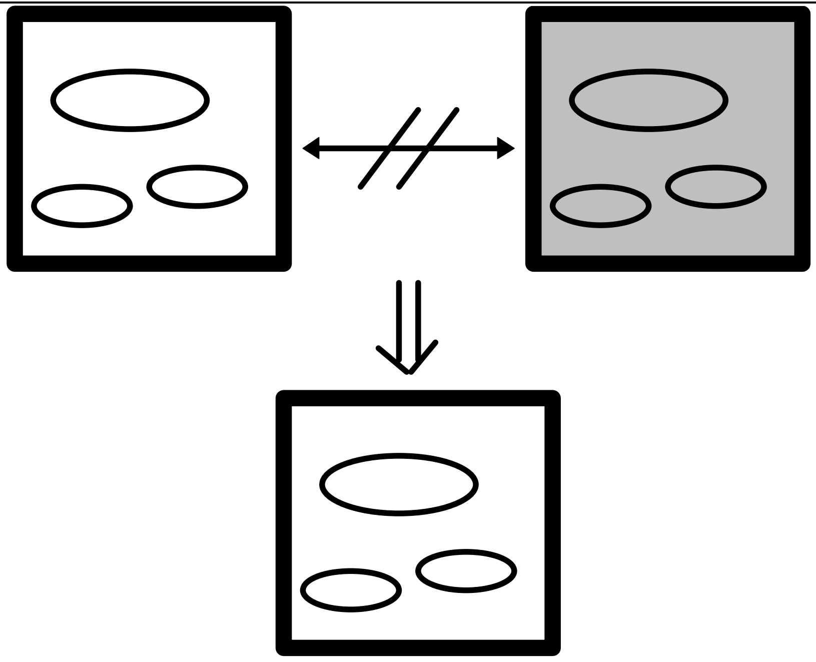


Figure 15-3: The extended virtual synchrony model allows both white gray partitions to continue progress despite the inconsistencies that may arise between their states. However, only one of the components is considered to be the primary one. Thus the white partition might be considered to be "authoritative" for the system while the gray one is permitted to remain alive but is known to be potentially stale. Later when the communication between the components is restored, the various process group components merge, resulting in a single larger system component with a single instance of each process group (below). The problem, however, is that merging must somehow overcome the inconsistencies that may have arisen during the original partitioning failure, and this may not always be possible. Working with such a model is potentially challenging for the developer. Moreover, one must ask what sorts of applications would be able to continue operating in the red partition knowing that the state of the system may at that point be inconsistent, for example reflecting the delivery of messages in orders that differ from the order in the main partition, having atomicity errors, or gaps in the message delivery ordering.

Recent work, which we will not have time to discuss in detail, points to yet a third possible mode of operation. In this mode, a computing system would be viewed as a wide-area-network composed of interconnected local area networks, as was first proposed in the Transis project. Within each of the LAN systems one would run a "local" subsystem: a complete primary-component system with its own sets of process groups and a self-sufficient collection of services and applications. The WAN layer of the system would be built up by superimposing a second communication structure on the aggregate of LAN's and would support its own set of WAN services. At this higher level of the system, one would use a true asynchronous communication model: if a partitioning event does occur, such a WAN system would wait until the problem is resolved. The WAN system would then be in a position to make use of protocols that don't attempt to make progress while communication is disrupted, but rather wait as long as necessary until the exchange of messages resumes and hence the protocol can be pushed forward. The consensus protocol of Chandra and Toueg is a good example of a protocol that one could use at the WAN level of a system structured in this manner, while the virtual synchrony model would be instantiated multiple times separately: once for each LAN subsystem.

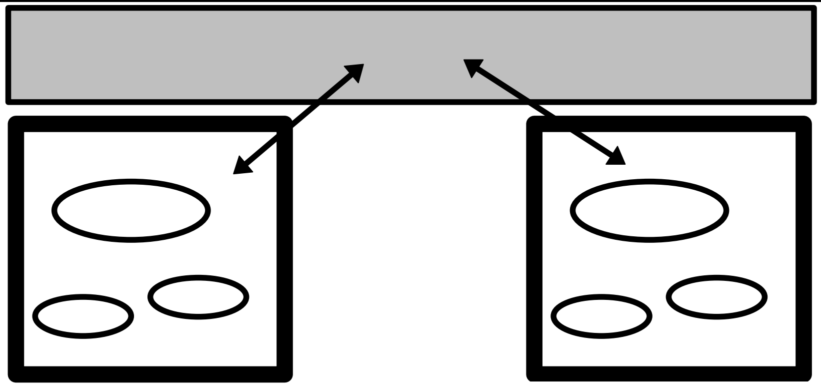


Figure 15-4: In a two-tiered model, each LAN has its own complete subsystem and runs using its own copy of the primary-component virtual synchrony model. A WAN system (gray) spans the LANs and is responsible for distributing global updates. The WAN layer may block while a partitioning failure prevents it from establishing the degree of consensus needed to safely deliver updates, but the local systems continue running even if global updates are delayed. Such a mixed approach splits the concerns of the application into local ones, where the focus is on local consistency and high availability, and global ones, where the focus is on global consistency even if partitioning failures can introduce long delays. The author favors this approach, which is used in the Isis Toolkit's "long-haul" subsystem, and has been applied successfully in such Isis applications as its wide-area "news" facility.

In this two-tiered model, an application would typically be implemented as a "local part" designed to remain available in the local component and to reconfigure itself to continue progress despite local failures. The primary component virtual synchrony model is ideal for this purpose. When an action is taken that has "global implications" the local part would initiate a global action by asking the WAN architecture to communicate this message through the WAN system. The WAN system would use potentially slow protocols that offer strong global ordering and atomicity properties at the expense of reduced progress when partitioning failures occur, delivering the resulting messages back into the various local subsystems. The local subsystems would then apply these updates to their "global states".

Danny Dolev has suggested the following simple way to understand such a two-tier system. In his view, the LAN subsystems run applications that are either entirely confined to the LAN (and have no interest in global state), or that operate by reading the *global* state but updating their *local* state. These applications do not directly update the global system state. Rather, if an action requires that the global state be updated, the LAN subsystem places the associated information on a WAN action queue, perhaps replicating this for higher availability. From that queue, the WAN protocols will eventually propagate the action into the WAN level of the system, out of which it will filter back down into the LAN level in the form of an update to the global system state. The LAN layer will then update its local state to reflect the fact that the requested action has finally been completed. The LAN layer of such a system would use the primary-component virtual synchrony model, while the WAN layer employs protocols based on the method in [KD95].

First introduced in the Isis system's "long-haul" service by Makpangou, and then extended through Dolev and Malki's work on the Transis architecture (which has a "lansis" and a "wansis" subsystem), two-tier architectures such as this have received attention in many projects and systems. They are now used in Transis, Horus, NavTech, Phoenix and Relacs. By splitting the application into the part that can be done locally with higher availability and the part that must be performed globally even if availability is limited, they don't force a black or white decision on the developer. Moreover, a great many applications seem to fit well with this model. Looking to the future, it seems likely that we will soon begin to see programming tools that encapsulate this architecture into a simple to use, object-oriented framework, making it readily accessible to a wide community of potential developers.

15.3 Virtually Synchronous Algorithms and Tools

In the subsections that follow, we develop a set of simple algorithms that illustrate the power, and limitations, of reliable multicast within dynamic process groups. These algorithms are just a small subset of the ones that can be developed using the primitives, and the sophisticated system designer may sometimes find a need for a causal and total multicast primitive ("cabcast"), or one with some other slight variation on the properties on which we have focused here. Happily, the protocols we have presented are easily modified for special needs, and modern group communication systems, like the Horus system, are designed precisely to accommodate such flexibility and fine-tuning. The algorithms that follow, then, should be viewed as a form of template upon which other solutions might be developed through a process of gradual refinement and adaptation.

15.3.1 Replicated Data and Synchronization

When discussing the static process group model, we put it to the test by using it to implement replicated data. The reader will recall from Section 13.7 that this approach was found to have a number of performance problems. The algorithm that resulted would have forced group members to execute nearly in lock-step, and the protocols themselves were costly in terms both of latency and messages required. Virtual synchrony, on the other hand, offers a solution to this problem that is inexpensive in all of these aspects, provided that dynamic uniformity is not required. When dynamic uniformity is required, the cost is still lower than for the static, quorum-replication methods although the advantage is less pronounced.

As suggested above, we start by describing our replication and synchronization algorithm in terms of a closely synchronous execution model. We will initially focus on the non dynamically-uniform case. Suppose that we wish to support *READ, UPDATE,* and *LOCK* operations on data replicated within a process group. As a first approximation to a solution would use *abcast* to implement the *UPDATE* and *LOCK* operations, while allowing any group member to perform *READ* operations using its local replica of the data maintained by the group.

Specifically, we will require that each group member maintain a private replica of the group data. When joining a group, the state transfer algorithm (developed below) must be used to initialize the replica associated with the joining process. Subsequently, all members will apply the same sequence of updates by tracking the order in which *UPDATE* messages are delivered, and respecting this order when actually performing the updates. *READ* operations, as suggested above, are performed using the local replica (this is in contrast to the quorum methods, where a read must access multiple copies).

An *UPDATE* operation can be performed without waiting for the group members to actually complete the individual update actions. Instead, an *abcast* is issued asynchronously (without waiting for the message to be delivered), and the individual replicas perform the update when the message arrives.

Many systems make use of non-exclusive "read" locks. If desired, these can also be implemented locally. The requesting process will be granted the lock immediately unless an exclusive (write) lock is registered at this copy. Finally, exclusive (write) *LOCK* operations are performed by issuing an *abcast* to request the lock and then waiting for each group member to grant it. A recipient of such a request waits until there are no pending read locks and then grants the request in the order it was received. The lock will later be released either with another *abcast* message, or upon reception of a new view of the process group reporting the failure of the process that holds the lock.

One would hope that it is obvious that this implementation of replicated data will be tolerant of failures and guarantee the consistency of the copies. The individual replicas start in identical states because the state transfer to a joining member copies the state of the replicated data object from some existing member. Subsequent updates and lock operations behave identically at all copies. Thus, all see the same events in the same order, and remain in identical states.

Now, let us ask how many of these *abcast* operations can be replaced with asynchronous *cbcast* operations. In particular, suppose that we replace *all* of the *abcast's* with asynchronous *cbcast*'*s*. Remarkably, with just two small changes, the modified algorithm will be correct. The first change is that *all updates must be guarded by a lock with appropriate granularity*. That is, if any update might be in conflict with a concurrent update, we will require that the application ensure that the update provide for some form of mutual exclusion. On the other hand, updates that are known to be independent commute, and hence can be issued concurrently. For example, updates to two different variables maintained by the same process group can be issued concurrently, and in groups where all the updates for a specific type of data originate with a single process, no locks may be required at all.

The second change is a bit more subtle: it has to do with the way that ordering is established when a series of write locks are requested within the group. The change is as follows. We will say that the first process to join the group, when the group is first created, is its *initial writer*. This process is considered to control write access to all the data items managed by the group.

Now, before doing an update, a process will typically request a lock, sending a *cbcast* to inform the group members of its *LOCK* request and waiting for the members to grant the request. In our original closely synchronous algorithm, a recipient of such a request granted it in first-come first-served order when no local read-lock was pending. Our modified algorithm, however, will wait before granting lock requests. They simply pile up in a queue, ordered in whatever order the *cbcast* messages were delivered.

When the writer for a given lock no longer needs that lock, we will say that it becomes *prepared to pass the lock*. This process will react to incoming lock requests by sending out a *cbcast* that *grants* the lock request. The grant message will be delivered to all group members. Once the grant message is received, a member dequeues the corresponding lock request (the causal ordering properties ensure that the request will indeed be found on the pending lock-request queue!) and then grants it when any read-locks that may be present for the same item have been released. A writer grants the lock to the process that issued that oldest of the pending lock requests on its version of the lock queue.

Having obtained a grant message for its lock request and individual confirmation messages from each group member that the lock has been acquired locally to it, the writer may begin issuing updates. In many systems the local read-lock mechanism will not be required in which case the members need not confirm write-lock acquisition, and the writer need not wait for these messages. The members simply dequeue the pending lock request when the grant message arrives, and the writer proceeds to issue updates as soon as it receives the grant message itself.

It may at first seem surprising that this algorithm can work: why should it ensure that the group members will perform the same sequence of updates on their replicas? To see this, start by noticing that the members actually might not perform identical sequences of updates (Figure 15-5). However, any sequence of *conflicting updates* will be identical at all replicas, for the following reason. Within the group, there can be only one writer that holds a given lock. That writer uses *cbcast* (asynchronously) to issue updates, and uses *cbcast* (again asynchronously) to grant the write lock to the subsequent writer. This establishes a total order on the updates: one can imagine a causal path traced through the group, from writer to writer, with the updates neatly numbered along it: the first update, the second, the granting of the lock to a new writer, the third update, the granting of the lock to a new writer, the fourth update, and so forth. Thus, when *cbcast* enforces the delivery order, any set of updates covered by the same lock will be delivered in the same order to all group members.

As for the non-conflicting updates: these commute with the others, and hence would have had the same effect regardless of how they were ordered. The ordering of updates is thus significant only with respect to other conflicting updates.

Finally, the reader may have noticed that lock requests are not actually seen in the same order at each participant. This is not a problem, however, because the lock request order isn't actually used in the algorithm. As long as the grant operation is reasonably fair and grants a request that is genuinely pending, the algorithm will work. These properties do hold in the algorithm as presented above.

The remarkable thing about this new algorithm is that it is almost entirely asynchronous. Recall that our *cbcast* algorithm is delivered in the same view of the process group as the one that was in place when the *cbcast* was initiated. This implies that the sender of a *cbcast* can always deliver "its own copy" of the multicast as soon as it initiates the message. After all, by definition, any causally prior *cbcasts* will already have been delivered at the sender, and the flush protocol enforces the view-synchrony and causal-gap freedom guarantees. This means that a process that wants to issue an update can perform the update locally, sending off a *cbcast* that will update the other replicas without delaying local computation. Clearly, a lock request will block the process that issues it — unless that process happens to hold the write lock already, as may often be the case in systems with bursty communication patterns. But it is clear that this minimal delay — the time needed to request permission to write, and for the grant message to travel

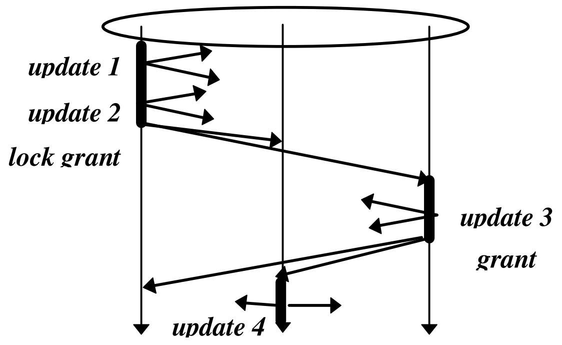


Figure 15-5: A set of conflicting updates are ordered because only one process can write at a time. Each update, and the lock-granting message, is an asynchronous cbcast. Because the causal order is in fact a total one along this causal path (shown in bold), all group members see the same updates in the same order. Lock requests are not shown, but they too would be issued using asynchronous cbcasts. Notice that lock requests will not be seen in the same order by all processes, but this is not required for the algorithm to behave correctly. All that matters is that the grant operation grant a currently pending lock request, and in this algorithm, all processes do have a way to track the pending requests, even though they may learn about those requests in different orders.

back to the requesting process, is necessary in any system.

Moreover, the algorithm can be simplified further. Although we used *cbcast* here, one could potentially replace these with *fbcast* by employing a sequence number: the *i'th* update would be so labeled, and all group members would simply apply updates in sequence order. The token would now represent permission to initiate new updates (and the guarantee that the values a process reads are the most current ones). Such a change eliminates the vector timestamp overhead associated with *cbcast,* and is also recognizable as an implementation of one of the *abcast* protocols that we developed earlier!

From the perspective of an application, this asynchronous replication and locking scheme may seem astonishingly fast. The only delays imposed upon the application are when it requests a new write lock. During periods when it holds the lock, or if it is lucky enough to find the lock already available, the application is never delayed at all. Read operations can be performed locally, and write operations respond as soon as the local update has been completed and the *cbcast* or *fbcast* (we'll just call it a *cbcast* for simplicity) to the remaining members handed off to the communications subsystem. Later, we will see that the Horus system achieves performance that can reach 85,000 such updates per second. Reads are essentially free, hence millions could be done per second. When this is compared with a quorum read and update technology, in which it would be surprising to exceed 100 reads and updates (combined) in one second, the benefits of an asynchronous *cbcast* become clear. In practice, quorum schemes are often considerably slower than this because of the overheads built into the algorithm. Moreover, a quorum read or update forces the group members into lock-step, while our asynchronous replication algorithm encourages them to leap ahead of one-another, buffering messages to be transmitted in the background.

However, we must not delude ourselves into viewing this algorithm as identical to the quorum replication scheme, because that scheme provides the equivalent of a dynamic uniformity guarantee and of a strong total ordering. The algorithm described above could be modified to provide such a guarantee by using a *safe cbcast* in place of the standard *cbcast* used above. But such a change will make the protocol dramatically slower, because each *UPDATE* will now be delayed until at least a majority of the group members acknowledge receipt of the update message. Thus, although the algorithm would continue to perform *READ* operations from local replicas, *UPDATE* operations will now be subject to the same performance limits as for a quorum update. The benefit may still be considerable, but the advantage of this scheme over a quorum one would be much reduced.

In the experience of the author, dynamic uniformity is needed quite rarely. If an application is about to take an externally visible action and it is important that, in the event of a failure, the other replicas of the application be in a consistent state with that of the application taking the action, this guarantee becomes important. In such cases, it can be useful to have a way to *flush* communication within the group, so that any prior asynchronous multicasts are forced out of the communication channels and delivered to their destinations. A *cbcast* followed by a *flush* is thus the equivalent of a *safe cbcast* (stronger, really, since the *flush* will flush all prior cbcasts, while a *safe cbcast* might not do provide this guarantee). Many process group systems, including Horus, adopt this approach rather than one based on a *safe cbcast*. The application developer is unlikely to use *flush* very frequently, hence the average performance may approximate that of our fully asynchronous algorithm, with occassional short delays when a *flush* pushes a few messages through the channels to their destinations. Unless large backlogs develop within the system, long delays are unlikely to arise. Thus, such a compromise can be very reasonable from the perspective of the application designer.

By way of analogy, many system developers are familiar with the behavior of operating systems that buffer disk I/O. In such settings, to increase performance, it is common to permit the application to continue operation as soon as a disk write is reflected in the cache contents — without waiting for the data to be flushed to the disk itself. When a stronger guarantee is required, the application explicitly requests that the disk buffer be flushed by invoking an appropriate primitive, such as the UNIX *fsync* system call. The situation created by the asynchronous *cbcast* is entirely analogous, and the role of the *flush* primitive is precisely the same as that of *fsync*. Thus, even if inconvenient, there is a sense in which this problem should be familiar!

What about a comparison with the closely synchronous algorithm from which ours was derived? Interestingly, the story here is not so clear. Suppose that we adopt the same approach to the dynamic uniformity issue, by using a *flush* primitive with this property is required. Now, the performance of the closely synchronous *abcast* algorithm will depend entirely on the way that *abcast* is implemented. In particular, one could implement *abcast* using the *cbcast*-based lock and update scheme of this section, or using a rotating token (with very similar results). Such an *abcast* solution would push the logic of our "algorithm" into the communication primitive itself. In principle, performance could come be the same as for an algorithm that uses *cbcast* explicitly. And, indeed, this level of performance has been achieved in experiments with the Horus system. The major issues is that to do so, one needs to use an *abcast* algorithm well matched to the communication pattern of the user, and this is not always possible: many systems lack the knowledge to predict such patterns accurately.

15.3.2 State transfer to a joining process

As we noted several times in the above discussion, there often arises a need to transfer information about the current state of a process group to a joining member at the instant it joins. In the iterated protocol by which a non-member of a group communicates with a group, for example, it turned out to be necessary to ensure that the group members learn how many messages have been successfully delivered from each client prior to the join event. In a replicated data algorithm, there is clearly a need to transfer a current copy of the data in question to the joining process.

The most appropriate representation of the state of the group, however, will be highly dependent on the application. Some forms of state may be amenable to extreme compression, or may be reconstructable from information stored on files or logs using relatively small amounts of information at the time the process joins. Accordingly, we adopt the view that a state transfer should be done by the application itself. Such a transfer is requested at the time of the join, and operates much like a remote procedure call.

Given this point of view, it is easy to introduce state transfer into the protocol for group flush that was described above. At the time a process first requests that it be added to the group, it should signal its intention to solicit state from the members. The associated information is passed in the form a message to the group members, and carried along with the join protocol to be reported with the new group view after the members perform the flush operation.

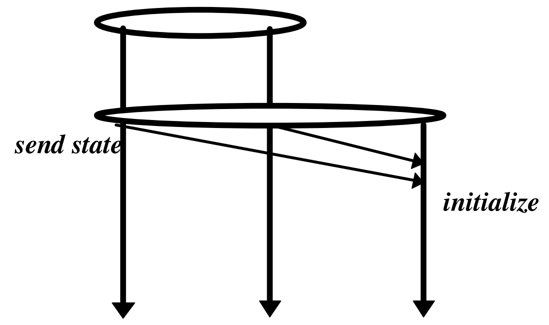


Figure 15-6: One of several state transfer mechanisms. In this very simple scheme, the group members all send their copies of the state to the joining member, and then resume computing. The method may be a good choice if the state is known to be small, since it minimizes delay, is fault-tolerant (albeit sending redundant information), and very easy to implement. If the state may be large, however, the overhead could be substantial.

Each member now faces a choice: it can stop processing new requests at the instant of the flush, or can make a copy of its state as of the time of the flush for possible future use, in which case it can resume processing. The joining process will solicit state information RPC-style, pulling it from one or more of the prior members. If state information is needed from all members, they can also send it without waiting for it to be solicited (Figure 15-6), although this can create a burst of communication load just at the moment when the flush protocol is still running, with the risk of momentarily overloading some processes or the network. At the other extreme, if a transfer is needed from just a single member, the joining process should transmit an asynchronous multicast *terminating* the transfer after it has successfully pulled the state from some member. The remaining members can now resume processing requests or discard any information that had saved for use during the state transfer protocol.

Perhaps the best among these options, if one single approach is desired as a default, is for the joining process to pull state from a single existing member, switching to a second member if a failure disrupts the transfer. The members should save the state in a buffer for later transfer, and should use some form of "out of band" transfer (for example over a specially created TCP channel) to avoid sending large state objects over the same channels used for other forms of group communication and request processing. The transfer being completed, the joining process should send a multicast that tells the other members it is safe to delete their saved state copies. This is illustrated in Figure 15-7.

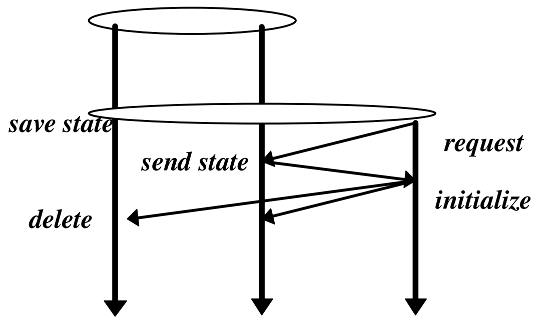


Figure 15-7: A good state transfer mechanism for cases where the state is of unknown size: the joining member solicits state from some existing member, and then tells the group as a whole when it is safe to delete the saved data.

Having had considerable experience with state transfer mechanisms, the author should warn developers of a frequently encountered problem. In many systems, the group state can be so large that transferring it represents a potentially slow operation. For example, in a file system application, the state transferred to a joining process might need to contain the full contents of every file that was modified since that process was last operational. Clearly, it would be a *terrible* idea to shut down the request processing by existing group members during this extended period of time!

Such considerations lead to three broad recommendations. First, if the state may be very large, it is advisable to transfer as much of it as possible before initiating the join request. A mechanism can then be implemented by which any "last minute changes" are transferred to the joining process, but without extended delays. Secondly, the state transfer should be done asynchronously, and in a manner that will not lead to congestion or flow control problems that impede the normal processing of requests by the service. A service that technically remains available, but actually is inaccessible because its communication channels are crammed with data to a joining process may seem very unavailable to other users. Finally, where possible, the approach of "jotting down the state" is preferable to one that shuts down a server even briefly during the transfer. Again, this reflects a philosophy whereby every effort is made to avoid delaying the response by the server to ongoing requests during the period while the join is still in progress.

15.3.3 Load-Balancing

One of the more common uses of process groups is to implement some form of load-balancing algorithm, whereby the members of a group share the workload presented to them so as to obtain a speedup from parallelism. It is no exaggeration to say that parallelism of this sort may represent the most important single property of process group computing systems: the opportunity to gain performance while also obtaining fault-tolerance benefits on relatively inexpensive cluster-style computing platforms is obviously of tremendous potential importance to the size of the market for process group server architectures.

There are two broad styles of load-balancing algorithms. The first style involves multicasting the client's request to the full membership of the group; the decision as to how the request should be processed is left for the group members to resolve. This approach has the advantage of requiring little trust in the client, but the disadvantage of communicating the full request (which may involve a large amount of data) to more processes than really need to see this information. In the second style, the client either makes a choice among the group members, or is assigned a preferred group member to which its requests are issued. Here, some degree of trust in the behavior of the clients is accepted in order to reduce the communication load on the system. In this second style, the client may also need to implement a *fail-over policy* by which it reissues a request if the server to which it was originally issued turns out to be faulty or fails while processing it.

Load-balancing algorithms of the first sort require some form of deterministic rule by which incoming requests can be assigned within the server group. As an example, if incoming requests are issued using an *abcast* protocol, the group can take advantage of the fact that all members see the requests in the same order. The *i'th* request can now be assigned to the server whose rank within the group is *i mod n,* or the servers can use some other deterministic algorithm for assigning the incoming work.

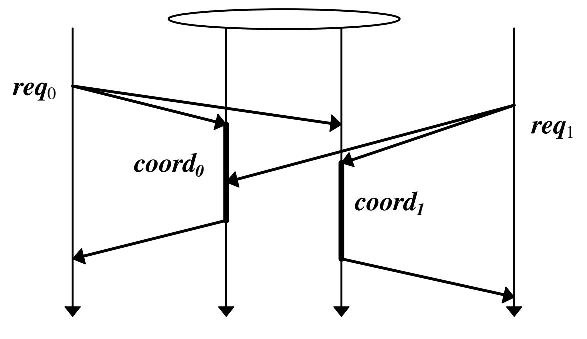


Figure 15-8: Load balancing based on a coordinator scheme using ranking. Ideally, the load on the members will be fairly uniform.

If group members periodically send out load reports to one-another, also using *abcast*, these load measures can be used to balance work in the following manner. Suppose that the servers in the group measure their load on an simple numeric scale, with 0 representing an unloaded server, 1 representing a server that is currently handling a single request, and so forth. The load on a group of *n* servers now can be represented as a vector [l_0, ... l_n]. Think of these load values as intervals within a line segment of total length $L = (l_0 + ... + l_n)$ and assume that the group members employ an algorithm for independently but identically generating pseudo-random numbers, the seed of which is transferred as part of the state passed to a joining process when it joins the group. Then as each new request is received, the group members can independently pick the same random number on the segment [*0,L*], assigning the request to the process corresponding to the interval within which that number falls. Such an approach will tend to equalize load by randomly spreading it within the group, and has the benefit of working well even if the load values are approximate and may be somewhat inaccurate.

The same methods can be used as the basis for client affinity load-balancing schemes. In these, the group members provide the client with information that they use to select the server to which requests will be sent. For example, the group can statically assign clients to particular members at the time the client first interacts with the group. Such an approach risks overloading a server whose clients happen to be unusually active, but can also be advantageous if caching is a key determinate of request processing performance, since this server is more likely to benefit from the use of a caching algorithm. Alternatively, the client can randomly select a server for each new request within the group membership, or can use the same load-balancing scheme outlined above to spread requests over the group membership using approximate load information, which the members would periodically broadcast to the clients. Any of these represents a viable option for distributing work, and the best choice for a given setting will depend on other information available only to the application designer, such as the likely size of the data associated with each request, fault-tolerance considerations (discussed in the next section), or issues such as the balance between queries (which can often be load-balanced) and update requests (which generally cannot).

In the Isis system, some use was made of load-balancing methods within which the members of a group subdivide the processing of individual requests among themselves (*e.g.* one process does "half" the work of processing a request and a second does the other "half", as might be done to speed the search of a large database). In practice, the author is not aware that much use was ever made of these methods, and consequently they are not included in the present textbook.

15.3.4 Primary-Backup Fault Tolerance

Earlier, we illustrated the concept of primary-backup fault-tolerance, in which a pair of servers are used to implement a critical service. Virtually synchronous process groups offer a good setting within which such an approach can be used [BBMS93].

Primary-backup fault-tolerance is most easily understood if one assumes that the application is completely deterministic. That is, the behavior of the server program will be completely determined by the order of inputs to it, and is therefore reproducible by simply replaying the same inputs in the same order to a second copy. Under this (admittedly unrealistic!) assumption, a backup server can track the actions of a primary server by simply arranging that a totally ordered broadcast be used to transmit incoming requests to the primary-backup group. The client processes should be designed to detect and ignore duplicate replies to requests (by numbering requests and including the number in the reply). The primary can simply compute results for incoming requests and reply normally, periodically informing the backup of the most recent replies that are known to have been received safely. The backup mimics the primary, buffering replies and garbage collecting them when such a status message is received. If the primary fails, the backup resends any replies in its buffer.

Most primary-backup schemes employ some form of checkpoint method to launch a new replica if the primary process actually does fail. At some convenient point soon after the failure, the backup turned primary makes a checkpoint of its state[15], and simultaneously launches a new backup process. The new process loads its initial state from the checkpoint and joins a process group with the primary. State transfer can also be used to initialize the backup, but this is often harder to implement because many primary-backup schemes must operate with old code that is not amenable to change, and in which the most appropriate form of "state" is hard to identify. Fortunately, it is just this class of server that is most likely to support a checkpoint mechanism.

The same approach can be extended to work with non-deterministic primary servers, but doing so is potentially much harder. The basic idea is to find a way to *trace* (keep a record of) the non-deterministic actions of the primary, so that the backup can be forced to repeat those actions in a trace-driven mode. For example, suppose that the only non-deterministic actions taken by the primary are to request the time of day from the operating system. This system call can be modified to record the value so obtained, sending it in a message to the backup. If the backup pauses each time it encounters a time-of-day system call, it will either see a copy of the value used by the primary (in which case it should use that value and ignore the value of its local clock), or see the primary fail (in which case it takes over as primary and begins to run off its local clock). Unfortunately, there can be a great many sources of non-determinism in a typical program, and some will be very hard to deal with: lightweight thread scheduling, delivery of interrupts, shared memory algorithms, I/O ready notifications through system calls such as "select", and so forth. Moreover, it is easily seen that to operate a primary-backup scheme efficiently, both the incoming requests, the corresponding replies, and these internal trace messages will need to be transmitted as asynchronously as possible, while respecting causality. Our causal ordering algorithms were oriented towards group multicast, and this particular case would demand non-trivial analysis and optimizations. Thus, in practice, primary-backup replication can be very hard to implement when using arbitrary servers.

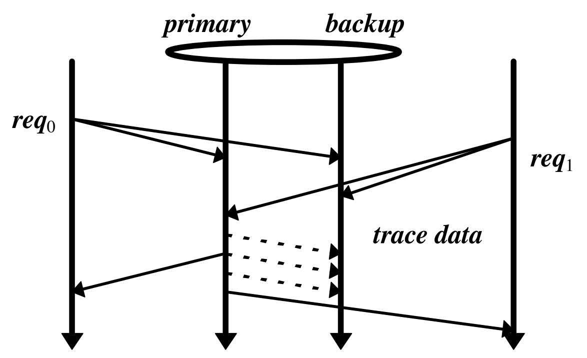


Figure 15-9: Primary-backup scheme for non-deterministic servers requires that trace information reach the backup. The fundamental requirement is a causal gap-freedom property: if a reply or some other visible consequence of a primary's actions is visible to a client or the outside world, all causally prior inputs to the primary and trace information must also be delivered to the backup. The trace data contains information about how non-deterministic actions were performed in the primary. The ordering obligation is ultimately a fairly weak one, and the primary could run "far ahead" of the backup, giving good performance and masking the costs of replication for fault-tolerance. The complexity of the scheme is fairly high because it can be hard to generate and use trace information, hence it is rare to see primary-backup fault-tolerance in non-deterministic applications.

[15] Interested readers may also want to read about log-based recovery techniques, which we do not cover in this book because these techniques have not been applied in many real systems. Alvisi gives a very general log-based recovery algorithm and reviews other work in the area in his doctoral dissertation and a paper with Marzullo.

Yet an additional drawback to the approach is that it may fail to overcome software bugs. As we can see, primary-backup replication is primarily appealing for deterministic applications. But these are just the ones in which Heisenbugs would be carefully repeated by a primary-backup solution, unless the fact of starting the backup from a state checkpoint introduces some degree of tolerance to this class of failures! Thus, the approach is likely to be exposed to correlated failures of the primary and backup in the case where it can be most readily applied.

15.3.5 Coordinator-Cohort Fault-Tolerance

The coordinator-cohort approach to fault-tolerance generalizes the primary-backup approach in ways that can help overcome the limitations cited above. In this fault-tolerance method, the work of handling requests is load-shared within the group members. (The same load-sharing mechanisms discussed before are used to balance load). The handler for a given request is said to be the *coordinator* for processing that request, and is responsible for sending any updates or necessary trace information to the other members, which are termed the *cohorts* for that request. As in the primary-backup scheme, if the coordinator fails, one of the cohorts takes over.

Unlike the primary-backup method, there may be many coordinators active in the same group, for many different requests. Moreover, the trace information in a primary backup scheme normally contains the information needed for the backup to duplicate the actions of the primary, whereas the trace data of a coordinator-cohort scheme will often consist of a "log" of updates that the coordinator applied to the group state. In this approach, the cohorts do not actively replicate the actions of the coordinator, but merely update their states to reflect its updates. Locking must be used for concurrency control. In addition, the coordinator will normally send some form of copy of the reply to its cohorts, so that they can garbage collect information associated with the pending requests for which they are backups. The approach is illustrated in Figure 15-10.

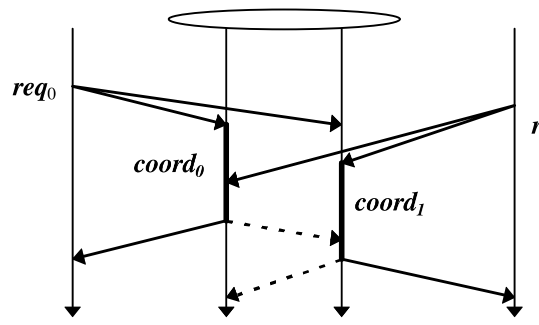


Figure 15-10: Coordinator-cohort scheme. The work of handling requests is divided among the processes in the group. Notice that as each coordinator replies, it also (atomically) informs the other group members that is has terminated. This permits them to garbage collect information about pending requests that other group members are handling. In the scheme, each process group member is active handling some requests, and is simultaneously passively backup up other members on other requests. The approach is best suited well for deterministic applications, but can also be adapted to non-deterministic ones.

Some practical cautions limit the flexibility of this style of load-balanced and fault-tolerant computing (which is quite popular among user's of systems like the Isis Toolkit and Horus, we should add!). First, it is important that the coordinator selection algorithm do a good job of load-balancing, or some single group member may become overloaded with the lion's share of the requests. In addition to this, the method can be very complex for requests that involve non-trivial updates to the group state, or that involve non-deterministic processing which the cohort may be expected to reproduce. In such cases, it can be necessary to use an atomic protocol for sending the reply to the requesting client and the trace information or termination information to the cohorts. Isis implements a protocol for this purpose: it is atomic and can send to the members of a group plus one additional member. However, such protocols are not common in most systems for reliable distributed computing. Given appropriate protocol support, however, and a reasonably simple server (for example, one processing requests that are primarily queries that don't change the server state), the approach can be highly successful, offering scaleable parallelism and fault-tolerance for the same "price".

15.4 Related Readings

On virtual synchrony: [BR94, Pow96, BR96], but see also [BJ87a, BJ87b, DM96, BR96, SR96]. Extended virtual synchrony: [Mal94], but also see [Ami95, KD95, Aga95, MMABL96]. On uses of the virtual synchrony model, see [BR94, BJ87a]. Primary-backup schemes, [BBMS93]. Discussion of other approaches to the same problems can be found in [Cri96].

16. Consistency in Distributed Systems

In the previous sections, we examined options for implementing replicated data in various group membership models and looked at protocols for ordering conflicting actions under various ordering goals. We then showed how these protocols could be used as the basis of a computational model, virtual synchrony, in which members of distributed process groups see events that occur within those groups in consistent orders and with failure atomicity guarantees, and are consequently able to behave in consistent ways.

Lacking is a more general synthesis, which we seek to provide in the present chapter. Key ideas underlying virtual synchrony are seen to be:

- *Self-defining system and process group membership, in which processes are excluded from a system if necessary to permit continued progress. Tools for joining a group, state transfer, communication, and reporting new membership views.*
- *Depending on the model, a notion of primary component of the system.*
- *Algorithms that seek to achieve internal (as opposed to dynamically uniform) consistency.*
- *Distributed consistency achieved by ordering conflicting replicated events in consistent ways at the processes that observe those events.*

The remainder of this chapter reviews these points relative to the alternatives we touched upon in developing our protocols and tools.

16.1 Consistency in the Static and Dynamic Membership Models

In the static model, the system is understood to be the set of places at which processes that act on behalf of the system execute. Here, “the system” is a relatively fixed collection of resources, but which experience dynamic disruptions of communication connectivity, process failures, and restarts. Obviously, a static system may not be static over very long periods of time, but the time scale on which membership of the full set of places where members run is understood to be long compared to the time scale at which these other events occur, and the protocols for adding new members to the static set or dropping them are treated as being outside of the normal execution model. In cases where the system is symmetric, meaning that any correct execution of the system would also have been correct if the process identifiers were permuted, static systems rely on agreement protocols within which the majority of the statically defined composition of the full system must participate, directly or indirectly.

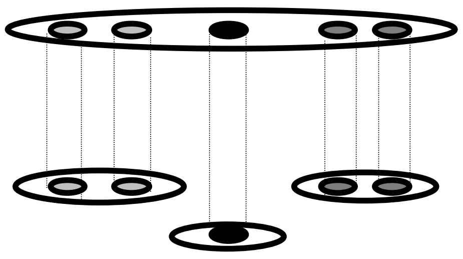


Figure 16-1: Static and dynamic views of a single set of sites. From a static perspective, the set has fixed membership but changing connectivity and availability properties (above). For example, the black nodes may be available and the gray ones treated as not available. Depending upon how such a system is implemented, it may be impossible to perform certain types of operations (notably, updates) unless a majority of the nodes are available. The dynamic perspective, shown below, treats the system as if it were partitioned into a set of components whose membership is self-defined (below). Here, the black component might be the primary one and the gray components non-primary. In contrast to the static approach, the primary component remains available, if primaryness can be deduced within the system. If communication is possible between two components, they are expected to merge their states in this model. Neither perspective is "more correct" than the other: the most appropriate way to view a system will typically depend upon the application, and different parts of the same application may sometimes require different approaches to membership. However, in the dynamic model, it is frequently important to track one of the components as being "primary" for the system, restricting certain classes of actions to occur only in this component (or not at all, if the primaryness attribute cannot be tracked after a complex series of failures).

The dynamic model employs a notion of system membership that is self-defined and for this reason less difficult to support than the static one. Dynamic systems add and lose members on a very short time scale compared to static ones. In the case where the system is symmetric, the set of processes that must participate in decisions is based on a majority of a dynamically defined group; this is a weaker requirement than for the static model and hence permits progress under conditions when a static system would not make progress.

These points are already significant when one considers what it means to say that a protocol is "live" in the two settings. However, before focusing on liveness, we review the question of consistency.

Consistency in a static model is typically defined with regard to an external observer who may be capable of comparing the state and actions of a process that has become partitioned from the other processes in the system with the states and actions of the processes that remained connected. Such an external observer could be a disk that contains a database that will eventually have to be reintegrated and reconciled with other databases maintained by the processes remaining in the connected portion of the system, an external device or physical process with which the system processes interact, or some form of external communication technology that lacks the flexibility of message passing but may still transfer information in some way between system processes.

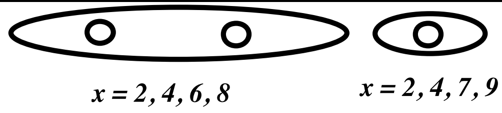


Figure 16-2: Dynamic(or "interactive") consistency is the guarantee that the members of a given system component will maintain mutually consistent states (here, by agreeing upon the sequence of values that a variable x has taken). If a protocol is not dynamically uniform, it may allow a process that becomes partitioned away from a component of the system to observe events in a way that is inconsistent with the event ordering observed within that component. Thus, in this example, the component on the right (consisting of a single process) observes x to take on the values 7 and 9, while the larger component on the right sees x pass through only even values. By pronouncing at most one of these components to be the primary one for the system, we can impose a sensible interpretation on this scenario. Alternatives are to use dynamically uniform protocols with external consistency guarantees. Such protocols can be supported both in the dynamic membership model and the static one, where this guarantee is almost always required. However, they are far more costly than protocols that do not provide dynamic uniformity.

Consistency in a dynamic system is a much more "internal" notion. In essence, a dynamic form of consistency requires that processes permitted to interact with one another will never observe contradictions in their states, detectable by comparing the contents of messages that they exchange. Obviously, process states and the system state evolve through time, but the idea here is that if process *p* sends a message to process *q* that in some way reflects state information shared by them, process *q* should never conclude that the message sent by process *p* is "impossible" on the basis of what *q* itself has seen in regard to this shared state. For example, if the state shared by *p* and *q* is a replicated variable and *q* has observed that variable to increment only by 2's from 0 to its current value of 40, it would be inconsistent if *p* sent a message, ostensibly reflecting a past state, in which the variable's value was *7*. For *q* such a state would not merely be stale, it would be impossible, since *q* believes itself to have seen the identical sequence of events, and the variable never had the value 7 in *q's* history.

Although this example is a silly one, it corresponds to more realistic scenarios in which dynamic consistency is precisely what one wants. For example, when a set of processes divide the work of performing some operation using a coordinator-cohort rule, or by exploiting a mutually perceived ranking to partition a database, dynamic consistency is required for the partitioning to make sense. Dynamic consistency is also what one might desire from the web proxies and servers that maintain copies of some document: they should agree on the version of the document that is the most current one, and provide guarantees to the user that the most current document is returned in response to a request.

The significance of the specific example seen above is thus not that applications often care about the past state of a replicated variable, but rather that "cooperation" or "coordination" or "synchronization" in distributed settings all involve cases in which a process *p* may need to reason about the state and actions of some other process *q*. When this occurs, *p* can be understood to be using a form of replicated system state that it believes itself to share with *q*. Our shared variable has now become the shared notion of the state of a lock, or the shared list of members and ranking of members for a process group to which both belong. Inconsistency in these cases means that the system is visibly misbehaving: two processes both think they have locked the same variable, or each thinks the other holds the lock when neither in fact holds it. Perhaps both processes consider themselves primary for some request, or neither does. Perhaps both search the first half of a database, each thinking the other is searching the second half. And these same issues only get worse if we move to larger numbers of processes.

Of course, as the system evolves through time, it may be that *p* once held a lock but no longer does. So the issue is not so much one of being continuously consistent, but of seeing mutually consistent and mutually evolving histories of the system state. In effect, if the processes in a system see the same events in the same order, they can remain consistent with one another. This extremely general notion is at the heart of all forms of distributed consistency.

In the purest sense, the dynamic system model is entirely concerned with freedom from detectable inconsistencies in the logically derivable system state. This notion is well defined in part because of the rule that when a dynamic system considers some process to have failed, communication to that process is permanently severed. Under such a rule, *p* cannot communicate to *q* unless both are still within the same component of the possibly partitioned system, and the protocols for dynamic systems operate in a manner that maintains consistency within subsets of processes that reside in the same component. That is, the system may allow a process to be inconsistent with the state of the system as a whole, but it does so only when that process is considered to have failed, and will never be allowed to rejoin the system until it has done something to correct its (presumably inconsistent) state.

The ability to take such an action permits dynamic systems to make progress when a static system might have to wait for a disconnected process to reconnect itself, or a failed process to be restarted. Thus, a process in the dynamic model can sometimes (often, in fact) make progress while a process in the static one would not be able to do so.

The static model, on the other hand, is in many ways a more intuitive and simpler one than the dynamic one. It is easy to draw an analogy between a static set of resources and a statically defined set of system processes, and external consistency constraints, being very strong, are also easy to understand. The dynamic model is in some sense superficially easy to understand, but much harder to understand upon close study. Suppose that we are told that "process *p* is a member of a dynamically defined system component, and sets a replicated variable *x* to 7." In a static system we would have concluded that, if the system is guaranteeing the consistency of this action, *p* was safe in taking it. In a dynamic system, it may be that it is too early to know if *p* is a valid member of the system, and hence that setting *x* to 7 is a safe action in the broader sense. The problem is that future events may cause the system to reconfigure itself in a way that excludes *p* and leads to an evolution of system state in which *x* never does take on the value 7. Moreover, the asynchronous nature of communication means that even if in real-time *p* sets *x* to 7 before being excluded by the other system members as if it was faulty, in the logical system model, *p's* action occurs *after* it has been excluded from the system!

Where external actions are to be taken, the introduction of time offers us a way to work around this dilemma. Recall our air traffic control example. Sharing a clock with the remainder of the system, *p* can be warned with adequate time to avoid a situation where two processes ever own the air traffic space at the same time. Of course, this does not eliminate the problem that during the period after it became disconnected and before the remainder of the system "took over", *p* may have initiated actions. We can resolve this issue by acknowledging that it is impossible to improve on the solution, and asking the application program to take an appropriate action. In this specific example, *p* would warn the air traffic controller that actions taken within the past δ seconds may not have been properly recorded by the main system, and that connection to it has now been lost. With a human in the loop, such a solution would seem adequate. In fact, there is little choice, for no system that takes actions at multiple locations can ever be precisely sure of its state if a failure occurs while such an action might be underway.

Faced with such seemingly troubling scenarios, one asks why we consider the dynamic model at all. Part of the answer is that the guarantees it offers are almost as strong as those for the static case, and yet it can often make progress when a static solution would be unable to do so. Moreover, the static model sometimes it just doesn't fit a problem. Web proxies, for example, are a very dynamic and unpredictable set: "the truth is out there", but a server will not be able to predict in advance just where copies of its documents may end up (imagine the case where one web proxy obtains a copy of a document from some other web proxy!). But perhaps the best answer is that, as we saw in previous chapters, the weaker model permits dramatically improved performance, perhaps by a factor of hundreds if our goal is to replicate data.

Both the static and dynamic system models offer a strong form of consistency whereby the state of the system is guaranteed to be consistent and coordinated over large numbers of components. But while taking an action in the static model can require a fairly slow, multiphase, protocol, the dynamic system is often able to exploit asynchronous single-phase protocols such as the non-uniform *fbcast* and *cbcast* primitives for similar purposes. It is no exaggeration to say that these asynchronous protocols may result in levels of performance that are hundreds of times superior to those achievable when subjected to static consistency and membership constraints! For example, the Horus system is able to send nearly 75,000 small multicasts per second to update a variable replicated between two processes. This figure drops to about 50 updates per second when using a quorum style replication scheme, and perhaps 1500 per second when using an RPC scheme that is disconnected from any notion of consistency at all. The latency improvements can be even larger: in Horus, there are latency differences of as much as three orders of magnitude between typical figures for the dynamic case and typical protocols for taking actions in a static or dynamically uniform manner.

In practical work with dynamic system models, we typically need to assume that the system is basically well behaved, despite experiencing some infrequent rate of failures. Under such an assumption, the model is easy to work with and makes sense. If a system experiences frequent failures (relative to the time it takes to reconfigure itself or otherwise repair the failures), the static model becomes more and more appealing and the dynamic one less and less predictable. Fortunately, most real systems are build with extremely reliable components that, indeed, experience infrequent failures. This pragmatic consideration explains why dynamically consistent distributed systems have become popular: the model behaves reasonably in real environments, and the performance is extremely good compared to what can be achieved in the static model.

Indeed, one way to understand the performance advantage of the dynamic model is that, by "precomputing" membership information, the dynamic algorithms represent optimizations of the static algorithms. As one looks closely at the algorithms, they seem more and more similar in a basic way, and perhaps this explains why that should be the case. In effect, the static and dynamic models are very similar, but the static algorithms (such as quorum data replication) tend to compute the membership information they needed on each operation, while the dynamic ones precompute this information and are built using a much simpler failstop model. However, although this perspective is intuitively appealing, the author has never seen it elaborated in any sort of detailed formal treatment

Moreover, it is important to realize that the external notion of consistency associated with static models is in some ways much stronger, and consequently more restrictive, than is necessary for realistic applications. This can translate to periods of "mandatory unavailability" where a static system model forces us to stop and wait, while a dynamic consistency model permits reconfiguration and progress. Many distributed systems contain services of various kinds that have small server states (which can therefore be transferred to a new server when it joins the system), and that are only of interest when they are operational and connected to "the system" as a whole. Mutual consistency between the servers and the states of the applications using them is all that one desires in such internal uses of a consistency preserving technology. If a dynamic approach is dramatically faster than a static one, so much the better for the dynamic approach!

These comments should not be taken to suggest that a dynamic system can *always* make progress even when a static one must wait. In recent work, Chandra and his colleagues have established that a result similar to the FLP result holds for group-membership protocols [CHTC96], and hence that there are conditions under which an asynchronous system can be prevented from reaching consensus upon its own membership, and hence prevented from making progress. Other researchers (including the author) have pinned down precise conditions (in various models) under which dynamic membership consensus protocols are guaranteed to make progress [BDM95, FKMBD95, GS96, Nei96], and the good news is that for most practical settings the answer is that such protocols make progress with overwhelmingly high probability if the probability of failures and message loss are uniform and independent over the processes and messages sent in the system. In effect, only partitioning failures or a very intelligent adversary (one that in practice could never be implemented) can prevent these systems from making progress.

Thus, we know that *all* of these models face conditions under which progress is not possible. Research is still underway on pinning down the precise conditions when progress *is* possible in each approach: the maximum rates of failures that dynamic systems can sustain. But as a practical matter, the evidence is that all of these models are perfectly reasonable for building reliable distributed systems. The theoretical impossibility results do not appear to represent practical impediments to implementing reliable distributed software; they simply tell us that there will be conditions that these reliability approaches cannot overcome. The choice, in a practical sense, is to match the performance and consistency properties of the solution to the performance and consistency requirements of the application. The weaker the requirements, the better the performance we can achieve.

Our study also revealed two other issues that deserve comment: the need, or lack thereof, for a *primary component* in a partitioned membership model, and the broader but related question of how consistency is tied to ordering properties in distributed environments.

The question of a primary component is readily understood in terms of the air-traffic control example we looked at earlier. In that example, there was a need to take "authoritative action" within a service on behalf of the system as a whole. In effect, a representative of a service needed to be sure that it could safely allow an air traffic control to take a certain action, meaning that it runs no risk of being contradicted by any other process (or, in the case of a possible partitioning failure, that before any other process could start taking potentially conflicting actions, a timeout would elapse and the air traffic controller warned that this representative of the service was now out of touch with the primary partition).

In the static system model, there is only a single notion of the system as a whole, and actions are taken upon the authority of the full system membership. Naturally, it can take time to obtain majority acquiescence in an action [KD95], hence this is a model in which some actions may be delayed for a considerable period of time. However, when an action is actually taken, it is taken on behalf of the full system.

In the dynamic model we lose this guarantee and face the prospect that our notion of consistency can become trivial because of system partitioning failures. In the limit, a dynamic system could partition arbitrarily, with each component having its own notion of authoritative action. For purely *internal* purposes, such a notion of consistency may be adequate, in the sense that it still permits work to be shared among the processes that compose the system, and (as noted above), is sufficient to avoid the risk that the states of processes will be directly inconsistent in a way that is readily detectable. The state merge problem [Mal94, BBD96], which arises when two components of a partitioned system reestablish communication connectivity and must reconcile their states, is where such problems are normally resolved (and the normal resolution is to simply take the state of one partition as being the official system state, abandoning the other). As noted in Chapter 13, this challenge has lead researchers working on the Relacs system in Bologna to propose a set of tools, combined with a set of guarantees that relate to view installation, which simplify the development of applications that can operate in this manner [BBD96].

The weakness of allowing simultaneous progress in multiple components of a partitioned dynamic system, however, is that there is no meaningful form of consistency that can be guaranteed between the components, unless one is prepared to pay the high cost of using only dynamically uniform message delivery protocols. In particular, the impossibility of guaranteeing progress among the participants in a consensus protocol implies that when a system partitions, there will be situations in which we can define the membership of both components but cannot decide how to terminate protocols that were underway at the time of the partitioning event. Consequences of this observation include the implication that when non-uniform protocols are employed, it will be impossible to ensure that the components have consistent histories (in terms of the events that occurred and the ordering of events) for their past prior to the partitioning event. In practice, one component, or both, may be irreconcilably inconsistent with the other!

There is no obvious way to "merge" states in such a situation: the only real option is to arbitrarily pick one component's state as the official one and to replace the other component's state with this state, perhaps reapplying any updates that occurred in the "unofficial" partition. Such an approach, however, can be understood as one in which the primary component is simply selected when the network partition is corrected rather than when it forms. If there is a reasonable basis on which to make the decision, why delay it?

As we saw in the previous chapter, there are two broad ways to deal with this problem. The one favored in the author's own work is to define a notion of *primary component* of a partitioned system, and to track primaryness when the partitioning event first occurs. The system can then enforce the rule that non-primary components must not trust their own histories of the past state of the system and certainly should not undertake authoritative actions on behalf of the system as a whole. A non-primary component may, for example, continue to operate a device that it "owns", but is not safe in instructing an air traffic controller about the status of air space sectors or other global forms of state-sensitive data unless they were updated using dynamically uniform protocols.

Of course, a dynamic distributed system can lose its primary component, and making matters still more difficult, there may be patterns of partial communication connectivity within which a static distributed system model can make progress but no primary partition can be formed, and hence a dynamic model must block! For example, suppose that a system partitions so that all of its members are disconnected from one another. Now we can selectively reenable connections so that over time, a majority of a static system membership set are able to vote in favor of some action. Such a pattern of communication could allow progress. For example, there is the protocol of Keidar and Dolev, cited several times above, in which an action can be terminated entirely on the basis of point to point connections [KD95]. However, as we commented, this protocol delays actions until a majority of the processes in the whole system knows about them, which will often be a very long time.

The author's work has not needed to directly engage these issues because of the underlying assumption that rates of failure are relatively low and that partitioning failures are infrequent and rapidly repaired. Such assumptions let us conclude that these types of partitioning scenarios just don't arise in typical local-area networks and typical distributed systems.

On the other hand, frequent periods of partitioned operation *could* arise in very mobile situations, such as when units are active on a battlefield. They are simply less likely to arise in applications like air traffic control systems or other "conventional" distributed environments. Thus, there are probably systems that should use a static model with partial communications connectivity as their basic model, systems that should use a primary component consistency model, and perhaps still other systems for which a virtual synchrony model that doesn't track primaryness would suffice. These represent successively higher levels of availability, and even the weakest retains a meaningful notion of distributed consistency. At the same time, they represent diminishing notions of consistency in any absolute sense. This suggests that there are unavoidable tradeoffs in the design of reliable distributed systems for critical applications.

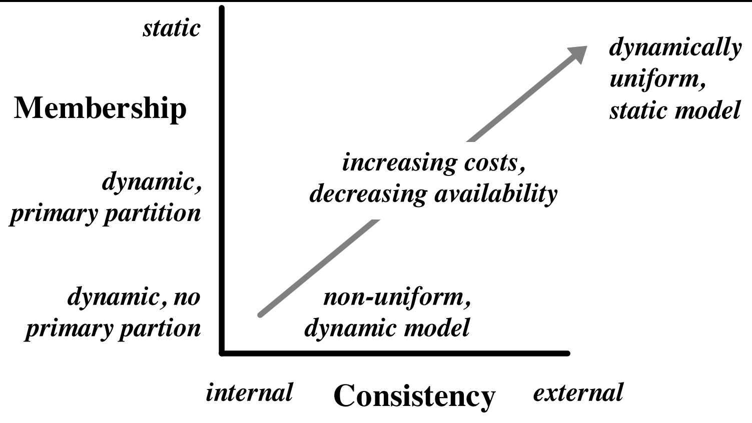

static
Membership
dynamic, primary partition
dynamic, no primary partion
dynamically uniform, static model
increasing costs, decreasing availability
non-uniform, dynamic model
internal
Consistency
external

Figure 16-3: Conceptual options for the distributed systems designer. Even when one seeks "consistency" there are choices concerning how strong the consistency desired should be, and which membership model to use. The least costly and highest availability solution for replicating data, for example, looks only for internal consistency within dynamically defined partitions of a system, and does not limit progress to the primary partition. This model, we have suggested, may be too weak for practical purposes. A slightly less available approach that maintains the same high level of performance allows progress only in the primary partition. As one introduces further constraints, such as dynamic uniformity or a static system model, costs rise and availability falls, but the system model becomes simpler and simpler to understand. The most costly and restrictive model sacrifices nearly three orders of magnitude of performance in some studies relative to the least costly one. Within any given model, the degree of ordering required for multicasts introduces further fine-grained cost/benefit tradeoffs.

The two-tiered architecture of the previous section can be recognized as a response to this impossibility result. Such an approach explicitly trades higher availability for weaker consistency in the LAN subsystems while favoring strong consistency at the expense of reduced availability in the WAN layer (which might run a protocol based on the Chandra-Toueg consensus algorithm). For example, the LAN level of a system might use non-uniform protocols for speed, while the WAN level uses tools and protocols similar to the ones proposed bythe Transis effort, or by Babaoglu's group in their work on Relacs [BBD96].

We alluded briefly to the connection between consistency and order. This topic is perhaps an appropriate one on which to end our review of the models. Starting with Lamport's earliest work on distributed computing systems, it was already clear that consistency and the ordering of distributed events are closely linked. Over time, it has become apparent that distributed systems contain what are essentially two forms of knowledge or information. Static knowledge is that information which is well known to all of the processes in the system, at the outset. For example, the membership of a static system is a form of static knowledge. Being well known, it can be exploited in a decentralized but consistent manner. Other forms of static knowledge can include knowledge of the protocol that processes use, knowledge that some processes are more important than others, or knowledge that certain classes of events can only occur in certain places within the system as a whole.

Dynamic knowledge is that which stems from unpredicted events that arise within the system either as a consequence of non-determinism of the members, failures or event orderings that are determined by external physical processes, or inputs from external users of the system. The events that occur within a distributed system are frequently associated with the need to update the system state in response to dynamic events. To the degree that system state is replicated, or is reflected in the states of multiple system processes, these dynamic updates of the state will need to occur at multiple places. In the work we presented above, process groups are the places where such state resides, and multicasts are used to update such state.

Viewed from this perspective, it becomes apparent that *consistency is order*, in the sense that the distributed aspects of the system state are entirely defined by process groups and multicasts to those groups, and these abstractions, in turn, are defined entirely in terms of ordering and atomicity. Moreover, to the degree that the system membership is self-defined, as in the dynamic models, atomicity is also an order-based abstraction!

This reasoning leads to the conclusion that the deepest of the properties in a distributed system concerned with consistency may be the *ordering in which distributed events are scheduled to occur*. As we have seen, there are many ways to order events, but the schemes all depend upon either explicit participation by a majority of the system processes, or upon dynamically changing membership, managed by a group membership protocol. These protocols, in turn, depend upon majority action (by a dynamically defined majority). Moreover, when examined closely, all the dynamic protocols depend upon some notion of token or special permission that enables the process holding that permission to take actions on behalf of the system as a whole. One is strongly inclined to speculate that in this observation lies the grain of a general theory of distributed computing, in which all forms of consistency and all forms of progress could be related to membership, and in which dynamic membership could be related to the liveness of token passing or "leader election" protocols. At the time of this writing, the author is not aware of any clear presentation of this theory of all possible behaviors for asynchronous distributed systems, but perhaps it will emerge in the not distant future.

Our goals in this textbook remain practical, however, and we now have powerful practical tools to bring to bear on the problems of reliability and robustness in critical applications. Even knowing that our solutions will not be able to guarantee progress under all possible asynchronous conditions, we have seen enough to know how to guarantee that *when* progress is made, consistency will be preserved. There are promising signs of emerging understanding of the conditions under which progress can be made, and the evidence is that the prognosis is really quite good: if a system rarely loses messages and rarely experiences real failures (or mistakenly detects failures), the system will be able to reconfigure itself dynamically and make progress while maintaining consistency.

As to the tradeoffs between the static and dynamic model, it may be that real applications should employ mixtures of the two. The static model is more costly in most settings (perhaps not in heavily partitioned ones), and may be drastically more expensive if the goal is merely to update the state of a distributed server or a set of web pages managed on a collection of web proxies. The dynamic primary component model, while overcoming these problems, lacks external safety guarantees that may sometimes be needed. And the non-primary component model lacks consistency and the ability to initiate authoritative actions at all, but perhaps this ability is not always needed. Complex distributed systems of the future may well incorporate multiple levels of consistency, using the cheapest one that suffices for a given purpose.

16.2 General remarks Concerning Causal and Total Ordering

The entire notion of providing ordered message delivery has been a source of considerable controversy within the community that develops distributed software [Ren93]. Causal ordering has been especially controversial, but even total ordering is opposed by some researchers [CS93], although others have been critical of the arguments advanced in this area [Bir94, Coo94, Ren94]. The CATOCS controversy came to a head in 1993, and although it seems no longer to interest the research community, it would also be hard to claim that there is a generally accepted resolution of the question.

Underlying the debate are tradeoffs between consistency, ordering, and cost. As we have seen, ordering is an important form of "consistency". In the next chapter we will develop a variety of powerful tools for exploiting ordering, especially to implement replicated data efficiently. Thus, since the first work on consistency and replication with process groups, there has been an emphasis on ordering. Some systems, like the Isis Toolkit developed by this author in the mid 1980's, made extensive use of causal ordering because of its relatively high performance and low latency. Isis, in fact, enforces causally delivered ordering as a system-wide default, although as we saw in Chapter 14, such a design point is in some ways risky. The Isis approach makes certain types of asynchronous algorithm very easy to implement, but has important cost implications; developers of sophisticated Isis applications sometimes need to disable the causal ordering mechanism to avoid these costs. Other systems, such as Ameoba, looked at the same issues but concluded that causal ordering is rarely needed if total ordering can be made fast enough. Writing this text, today, this author tends to agree with the Ameoba project except in certain special cases.

Above, we have seen a sampling of the sorts of uses to which ordered group communication can be put. Moreover, earlier sections of this book have established the potential value of these sorts of solutions in settings such as the Web, financial trading systems, and highly available database or file servers.

Nonetheless, there is a third community of researchers (Cheriton and Skeen are best known within this group) who have concluded that ordered communication is almost never matched with the needs of the application [CS93]. These researchers cite their success in developing distributed support for equity trading in financial settings and work in factory automation, both settings in which developers have reported good results using distributed message-bus technologies (TIB is the one used by Cheriton and Skeen) that offer little in the sense of distributed consistency or fault-tolerance guarantees. To the degree that the need arises for consistency within these applications, Cheriton and Skeen have found ways to reduce the consistency requirements of the *application* rather than providing stronger consistency within a system to respond to a strong application-level consistency requirement (the NFS example from Section 7.3 comes to mind). Broadly, this leads them to a mindset that favors the use of stateless architectures, non-replicated data, and simple fault-tolerance solutions in which one restarts a failed server and leaves it to the clients to reconnect. Cheriton and Skeen suggest that such a point of view is the logical extension of the end-to-end argument [SRC84], which they interpret as an argument that each application must take direct responsibility for guaranteeing its own behavior.

Cheriton and Skeen also make some very specific points. They are critical of system-level support for causal or total ordering guarantees. The argue that communication ordering properties are better left to customized application-level protocols, which can also incorporate other sorts of application-specific properties. In support of this view, they present applications that need stronger ordering guarantees and applications that need weaker ones, arguing that in the former case, causal or total ordering will be inadequate, and in the latter that it will be overkill (we won't repeat these examples here). Their analysis leads them to conclude that in *almost all cases*, causal order is more than the application needs (and more costly), or less than the application needs (in which case the application must add some higher level ordering protocol of its own in any case), and similarly for total ordering [CS93].

Unfortunately, while making some good points, this paper also includes a number of questionable claims, including some outright errors that were refuted in other papers including one written by the author of this text [Bir94, Coo94, Ren94]. For example, they claim that causal ordering algorithms have an overhead on messages that grows as n^2 where n is the number of processes in the system as a whole. Yet we have seen that causal ordering for group multicasts, the case Cheriton and Skeen claim to be discussing, can easily be provided with a vector clock whose length is linear in the number of active senders in a group (rarely more than two or three processes), and that in more complex settings, compression techniques can often be used to bound the vector timestamp to a small size. This particular claim is thus incorrect. The example is just one of several specific points on which Cheriton and Skeen make statements that could be disputed purely on technical grounds.

Also curious is the entire approach to causal ordering adopted by Cheriton and Skeen. In this chapter, we have seen that causal order is often needed when one seeks to *optimize* an algorithm expressed originally in terms of totally ordered communication, and that total ordering is useful because, in a state-machine style of distributed system, by presenting the same inputs to the various processes in a group in the same order, their states can be kept consistent. Cheriton and Skeen never address this use of ordering, focusing instead on causal and total order in the context of a publish-subscribe architecture in which a small number of data publishers send data that a large number of consumers receive and process, and in which there are no consistency requirements that span the consumer processes. This example somewhat misses the point of the preceedings chapters, where we made extensive use of total ordering primarily for consistent replication of data, and of causal ordering as a relaxation of total ordering where the sender has some form of mutual exclusion within the group.

To this author, Cheriton and Skeen's most effective argument is one based on the end-to-end philosophy. They suggest, in effect, that although many applications will benefit from properties such as fault-tolerance, ordering, or other communication guarantees, no single primitive is capable of capturing all possible properties without imposing absurdly high costs for the applications that required weaker guarantees. Our observation about the cost of dynamically uniform strong ordering bears this out: here we see a very strong property, but it is also thousands of times more costly than rather similar but weaker property! If one makes the weaker version of a primitive the default, the application programmer will need to be careful not to be surprised by its non-uniform behavior; the stronger version may just be too costly for many applications. Cheriton and Skeen generalize from similar observations based on their own examples and conclude that the application should implement its own ordering protocols.

Yet we have seen that these protocols are not trivial, and implementing them would not be an easy undertaking. It also seems unreasonable to expect the average application designer to implement a special-purpose, hand-crafted protocol for each specific need. In practice, if ordering and atomicity properties are not provided by the computing system, it seems unlikely that applications will be able to make any use of these concepts at all. Thus, even if one agrees with the end-to-end philosophy, one might disagree that it implies that each application programmer should implement nearly identical and rather complex ordering and consistency protocols, because no single protocol will suffice for all uses.

Current systems, including the Horus system which was developed by the author and his colleagues at Cornell, usually adopt a middle ground, in which the ordering and atomicity properties of the communication system are viewed as options that can be selectively enabled (Chapter 18). The designer can in this way match the ordering property of a communication primitive to the intended use. If Cheriton and Skeen were using Horus, their arguments would warn us not to enable such-and-such a property for a particular application because the application doesn't need the property and the property is costly. Other parts of their work would be seen to argue in favor of additional properties beyond the ones normally provided by Horus. As it happens, Horus is easily extended to accomodate such special needs. Thus the reasoning of Cheriton and Skeen can be seen as critical of systems that adopt a single all-or-nothing approach to ordering or atomicity, but perhaps not of systems such as Horus that seek to be more general and flexible.

The benefits of providing stronger communication tools in a "system", in the eyes of the author, are that the resulting protocols can be highly optimized and refined, giving much better performance than could be achieved by a typical application developer working over a very general but very "weak" communications infrastructure. To the degree that Cheriton and Skeen are correct and application developers will need to implement special-purpose ordering properties, such a system can also provide powerful support for the necessary protocol development tasks. In either case, the effort required from the developer is reduced and the reliability and performance of the resulting applications improved.

We mentioned that the community has been particularly uncomfortable with the causal ordering property. Within a system such as Horus, causal order is normally used *as an optimization* of total order, in settings where the algorithm was designed to use a totally ordered communication primitive but exhibits a pattern communication for which the causal order is also a total one. We will return to this point below, but we mention it now simply to stress that the "explicit" use of casually ordered communication, much criticized by Cheriton and Skeen, is actually quite uncommon. More typical is a process of refinement whereby an application is gradually extended to use less and less costly communication primitives in order to optimize performance. The enforcement of causal ordering, system wide, is not likely to become standard in future distributed systems. When *cbcast* is substituted for *abcast* communication may cease to be totally ordered but any situation in which messages arrive in different orders at different members will be due to events that commute. Thus their *effect* on the group state will be as if the messages had been received in a total order even if the actual sequence of events is different.

In contrast, much of the discussion and controversy surrounding causal order arises when causal order is considered not as an optimization, but rather as an ordering property that one might employ by default, just as a stream provides FIFO ordering by default. Indeed, the analogy is a very good one, because causal ordering is an extention of FIFO ordering. Additionally, much of the argument over causal order uses examples in which point-to-point messages are sent asynchronously, with system-wide causal order used to to ensure that "later" messages arrive after "earlier" ones. There some merit in this view of things, because the assumption of system-wide causal ordering permits some very asynchronous algorithms to be expressed extremely elegantly and simply. It would be a shame to lose the option of exploiting such algorithms. However, system-wide causal order is not really the main use of causal order, and one could easily live without such a guarantee. Point-to-point messages can also be sent using a fast RPC protocol, and saving a few hundred microseconds at the cost of a substantial system-wide overhead seems like a very questionable design choice; systems like Horus obtain system-wide causality, if desired, by waiting for asynchronously transmitted messages to become stable in many situations.

On the other hand, when causal order is used as an optimization of atomic or total order, the performance benefits can be huge. So we face a performance argument, in fact, in which the rejection of causal order involves an acceptance of higher than necessary latencies, particularly for replicated data.

Notice that if asynchronous *cbcast* is only used to replace *abcast* in settings where the resulting delivery order will be unchanged, the associated process group can still be programmed under the assumption that all group members will see the same events in the same order. As it turns out, there are cases in which the handling of messages commute and the members may not even need to see messages in identical ordering in order to behave as if they did. There are major advantages to exploiting these cases: doing so potentially reduces idle time (because the latency to message delivery is lower, hence a member can start work on a request sooner, if the *cbcast* encodes a request that will cause the recipient to perform a computation). Moreover, the risk that a Heisenbug will cause all group members to fail simultaneously is reduced because the members do not process the requests in identical orders, and Heisenbugs are likely to be very sensitive to the detailed ordering of events within a process. Yet one still presents the algorithm in the group and thinks of the group as if all the communication within it was totally ordered.

16.3 Summary and Conclusion

There has been a great deal of debate over the notions of consistency and reliability in distributed systems (which are sometimes seen as violating end-to-end principles), and of causal or total ordering (which are sometimes too weak or too strong for the needs of a specific application that does need ordering). Finally, although we have not focused on this here, there is the criticism that technologies such as the ones we have reviewed do not "fit" with standard styles of distributed systems development.

As to the first concern, the best argument for consistency and reliability is to simply exhibit classes of critical distributed computing systems that will not be sufficiently available unless data is replicated, and will not be trustworthy unless the data is replicated consistency. We have done so throughout this textbook; if the reader is unconvinced, there is little that will convince him or her. On the other hand, one would not want to conclude that *most* distributed applications need these properties: today, the ones that do remain a fairly small subset of the total. However, this subset is rapidly growing. Moreover, even if one believed that consistency and reliability are extremely important in a great many applications, one would not want to impose potentially costly communication properties system-wide, especially in applications with very large numbers of overlapping process groups. To do so is to invite poor performance, although there may be specific situations where the enforcement of strong properties within small sets of groups is desirable or necessary.

Turning to the second issue, it is clearly true that different applications have different ordering needs. The best solution to this problem is to offer systems that permit the ordering and consistency properties of a communications primitive or process group to be tailored to their need. If the designer is concerned about paying the minimum price for the properties an application really requires, such a system can then be configured to only offer the properties desired. Below, will see that the Horus system implements just such an approach.

Finally, as to the last issue, it is true that we have presented a distributed computing model that, so far, may not seem very closely tied to the software engineering tools normally used to implement distributed systems. In the next chapter we study this practical issue, looking at how group communication tools and virtual synchrony can be applied to real systems that may have been implemented using other technologies.

16.4 Related Reading

On notions of consistency in distributed systems: [BR94, BR96]; in the case of partitionable systems, [Mal94, KD95, MMABL96, Ami95]. On the Causal Controversy, [Ren93]. The dispute over CATOCS: [CS93], with responses in [Bir94, Coo94, Ren94]. The end-to-end argument was first put forward in [SRC84]. Regarding recent theoretical work on tradeoffs between consistency and availability: [FLP85, CHTC96, BDM95, FKMBD95, CS96].

17. Retrofitting Reliability into Complex Systems

This chapter is concerned with options for presenting group computing tools to the application developer. Two broad approaches are considered: those involving wrappers that encapsulate an existing piece of software in an environment that transparently extends its properties, for example by introducing fault-tolerance through replication or security, and those based upon toolkits which provide explicit procedure-call interfaces. We will not examine specific examples of such systems now, but instead focus on the advantages and disadvantages of each approach, and on their limitations. In the next chapter and beyond, however, we turn to a real system on which the author has worked and present substantial detail, and in Chapter 26 we review a number of other systems in the same area.

17.1 Wrappers and Toolkits

The introduction of reliability technologies into a complex application raises two sorts of issues. One is that many applications contain substantial amounts of preexisting software, or make use of off-the-shelf components (the military and government favors the acronym COTS for this, meaning "components off the shelf"; presumably because OTSC is hard to pronounce!) In these cases, the developer is extremely limited in terms of the ways that the old technology can be modified. A *wrapper* is a technology that overcomes this problem by intercepting events at some interface between the unmodifiable technology and the external environment [Jon93], replacing the original behavior of that interface with an extended behavior that confers a desired property on the wrapped component, extends the interface itself with new functionality, or otherwise offers a virtualized environment within which the old component executes. Wrapping is a powerful technical option for hardening existing software, although it also has some practical limitations that we will need to understand. In this section, we'll review a number of approaches to performing the wrapping operation itself, as well as a number of types of interventions that wrappers can enable.

An alternative to wrapping is to explicitly develop a new application program that is designed from the outset with the reliability technology in mind. For example, we might set out to build an authentication service for a distributed environment that implements a particular encryption technology, and that uses replication to avoid denial of service when some of its server processes fail. Such a program would be said to use a *toolkit* style of distributed computing, in which the sorts of algorithms developed in the previous chapter are explicitly invoked to accomplish a desired task. A toolkit approach packages potentially complex mechanisms, such as replicated data with locking, behind simple to use interfaces (in the case of replicated data, *LOCK, READ* and *UPDATE* operations). The disadvantage of such an approach is that it can be hard to glue a reliability tool into an arbitrary piece of code, and the tools themselves will often reflect design tradeoffs that limit generality. Thus, toolkits can be very powerful but are in some sense inflexible: they adopt a programming paradigm, and having done so, it is potentially difficult to use the functionality encapsulated within the toolkit in a setting other than the one envisioned by the tool designer.

Toolkits can also take other forms. For example, one could view a firewall, which filters messages entering and exiting a distributed application, as a tool for enforcing a limited security policy. When one uses this broader interpretation of the term, toolkits include quite a variety of presentations of reliability technologies. In addition to the case of firewalls, a toolkit could package a reliable communication technology as a message bus, a system monitoring and management technology, a fault-tolerant file system or database system, a wide-area name service, or in some other form (Figure 17-1). Moreover, one can view a programming language that offers primitives for reliable computing as a form of toolkit.

Server replication	Tools and techniques for replicating data to achieve high availability, load-balancing, scalable parallelism, very large memory-mapped caches, etc. Cluster API's for management and exploitation of clusters
Video server	Technologies for striping video data across multiple servers, isochronous replay, single replay when multiple clients request the same data
WAN replication	Technologies for data diffusion among servers that make up a corporate network.
Client groupware	Integration of group conferencing and cooperative work tools into Java agents, Tcl/Tk, or other GUI-builders and client-side applications.
Client reliability	Mechanisms for transparently fault-tolerant RPC to servers, consistent data subscription for sets of clients that monitor the same data source, etc.
System management	Tools for instrumenting a distributed system and performing reactive control. Different solutions might be needed when instrumenting the network itself, cluster-style servers, and user-developed applications.
Firewalls and containment tools	Tools for restricting the behavior of an application or for protecting it against a potentially hostile environment. For example, such a toolkit might provide a bank with a way to install a "partially trusted" client-server application so as to permit its normal operations while prevening unauthorized ones.

Figure 17-1: Some types of toolkits that might be useful in building or hardening distributed systems. Each toolkit would address a set of application-specific problems, presenting an API specialized to the programming language or environment within which the toolkit will be used, and to the task at hand. While it is also possible to develop extremely general toolkits that seek to address a great variety of possible types of users, doing so can result in a presentation of the technology that is architecturally weak and hence doesn't guide the user to the best system structure for solving their problems. In contrast, application-oriented toolkits often reflect strong structural assumptions that are known to result in solutions that perform well and achieve high reliability.

In practice, many realistic distributed applications require a mixture of toolkit solutions and wrappers. To the degree that a system has new functionality which can be developed with a reliability technology in mind, the designer is afforded a great deal of flexibility and power through the execution model supported (for example, transactional serializability or virtual synchrony), and may be able to provide sophisticated functionality that would not otherwise be feasible. On the other hand, in any system that reuses large amounts of old code, wrappers can be invaluable by shielding the previously developed functionality from the programming model and assumptions of the toolkit.

17.1.1 Wrapper Technologies

In our usage, a wrapper is any technology that intercepts an existing execution path in a manner transparent to the wrapped application or component. By wrapping a component, the developer is able to virtualize the wrapped interface, introducing an extended version with new functionality or other desirable properties. In particular, wrappers can be used to introduce various robustness mechanisms, such as replication for fault-tolerance, or message encryption for security.

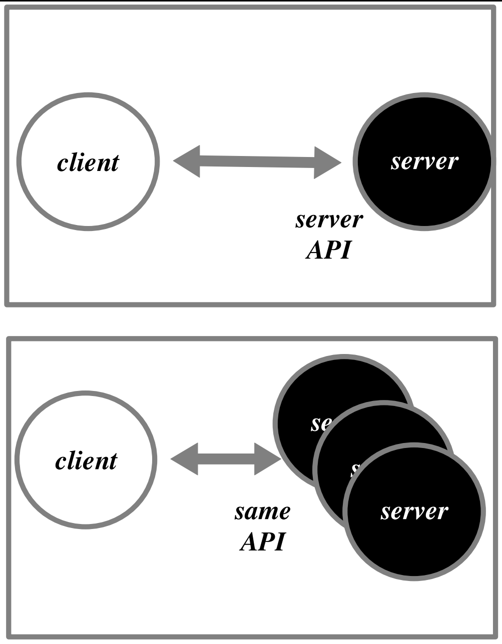


Figure 17-2: Object oriented interfaces permit the easy substitution of a reliable service for a less reliable one. They represent a simple example of a "wrapper" technology. However, one can often wrap a system component even if it was not built using object-oriented tools.

17.1.1.1 Wrapping at Object Interfaces

Object oriented interfaces are the best example of a wrapping technology (Figure 17-2), and systems built using Corba or OLE-2 are, in effect, "pre-wrapped" in a manner that makes it easy to introduce new technologies or to substitute a hardened implementation of a service for a non-robust one. Suppose, for example, that a Corba implementation of a client-server system turns out to be unavailable because the server has sometimes crashed. Earlier, when discussing Corba, we pointed out that the Corba architectural features in support of dynamic reconfiguration or "fail-over" are difficult to use. If, however, a Corba service could be replaced with a process group ("object group") implementing the same functionality, the problem becomes trivial. Technologies like Orbix+Isis and Electra, described in Chapter 18, provide precisely this ability. In effect, the Corba interface "wraps" the service in such a manner that any other service providing a compatible interface can be substituted for the original one transparently.

17.1.1.2 Wrapping by Library Replacement

Even when we lack an object-oriented architecture, similar ideas can often be employed to achieve these sorts of objectives. As an example, one can potentially wrap a program by relinking it with a modified version of a library procedure that it calls. In the relinked program, the code will still issue the same procedure calls as it did in the past. But control will now pass to the wrapper procedures which can take actions other than those taken by the original versions.

In practice, this specific wrapping method would only work on older operating systems, because of the way that libraries are implemented on typical modern operating systems. Until fairly recently, it was typical for linkers to operate by making a single pass over the application program, building a *symbol table* and a list of *unresolved external references*. The linker would then make a single pass over the library (which would typically be represented as a directory containing object files, or as an archive of object files), examining the symbol table for each contained object and linking it to the application program if the symbols it declares include any of the remaining unresolved external references. This process causes the size of the program object to grow, and results in extensions both to the symbol table and, potentially, to the list of unresolved external references. As the linking process continues, these references will in turn be resolved, until there are no remaining external references. At that point, the linker assigns addresses to the various object modules and builds a single program file which it writes out. In some systems, the actual object files are not copied into the program, but are instead loaded dynamically when first referenced at runtime.

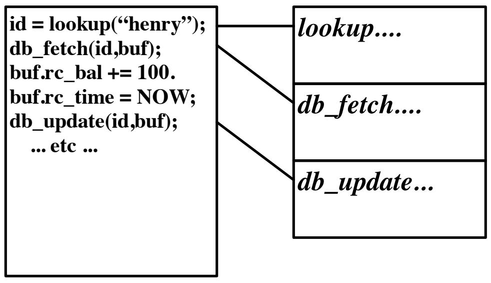


Figure 17-3: A linker establishes the correspondence between procedure calls in the application and procedure definitions in libraries, which may be shared in some settings.

Operating systems and linkers have evolved, however, in response to pressure for more efficient use of computer memory. Most modern operating systems support some form of shared libraries. In the shared library schemes, it would be impossible to replace just one procedure in the shared library. Any wrapper technology for a shared library environment would then involve reimplementing all the procedures defined by the shared library, a daunting prospect.

17.1.1.3 Wrapping by Object Code Editing

Object code editing is an example of a recent wrapping technology that has been exploited in a number of recent research and commercial application settings. The approach was originally developed by Wahbe, Lucco, Anderson and Graham [WLAG93], and involves analysis of the object code files before or during the linking process. A variety of object code transformations are possible. Lucco, for example, uses object code editing to enforce type safety and to eliminate the risk of address boundary violations in modules that will run without memory protection: a software fault isolation technique.

For purposes of wrapping, object code editing would permit the selective remapping of certain procedure calls into calls to wrapper functions, which could then issue calls to the original procedures if desired. In this manner, an application that uses the UNIX *sendto* system call to transmit a message could be transformed into one that calls *filter_sendto* (perhaps even passing additional arguments). This procedure, presumably after filtering outgoing messages, could then call *sendto* if a message survives its output filtering criteria. Notice that an approximation to this result can be obtained by simply reading in the symbol table of the application's object file and modifying entries prior to the linking stage.

One important application of object code editing, cited earlier, involves importing untrustworthy code into a client's Web browser. When we discussed this option in Section 10.9, we described it simply as a security enhancement tool. Clearly, however, the same idea could be useful in many other settings. Thus it makes sense to understand object code editing as a wrapping technology, and the specific use of it in Web browser applications as an example of how such a wrapper might permit us to increase our level of trust in applications that would otherwise represent a serious security threat.

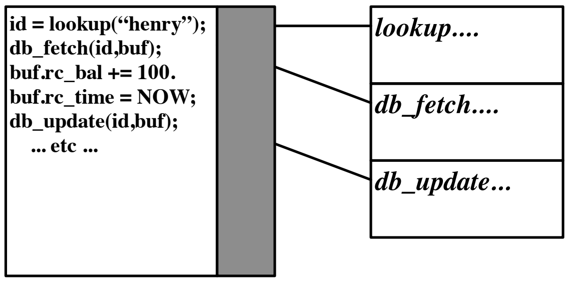


Figure 17-4: A wrapper (gray) intercepts selected procedure calls or interface invocations, permitting the introduction of new functionality transparently to the application or library. The wrapper may itself forward the calls to the library, but can also perform other operations. Wrappers are an important option for introducing reliability into an existing application, which may be too complex to rewrite or to modify easily with explicit procedure calls to a reliability toolkit or some other new technology.

17.1.1.4 Wrapping With Interposition Agents and Buddy Processes

Up to now, we have focused on wrappers that operate directly upon the application process and that live in its address space. However, wrappers need not be so intrusive.

Interposition involves placing some sort of object or process in between an existing object or process and its users. An interposition architecture based on what are called "coprocesses" or "buddy" processes is a simple way to implement this approach, particularly for developers familiar with UNIX "pipes" (Figure 17-5). Such an architecture involves replacing the connections from an existing process to the outside world with an interface to a buddy process that has a much more sophisticated view of the external environment. For example, perhaps the existing program is basically designed to process a pipeline of data, record by record, or to process batch-style files containing large numbers of records. The buddy process might employ a pipe or file system interface to the original application, which will often continue to execute as if it were still reading batch files or commands typed by a user at a terminal, and hence may not need to be modified. To the outside world, however, the interface seen is the one presented by the buddy process, which may now exploit sophisticated technologies such as CORBA, DCE, the Isis Toolkit or Horus, a message bus, and so forth. (One can also imagine imbedding the buddy process directly into the address space of the original application, coroutine style, but this is likely to be much more complex and the benefit may be small unless the connection from the buddy process to the older application is known to represent a bottleneck). The pair of processes would be treated as a single entity for purposes of system management and reliability: they would run on the same platform, and be set up so that if one fails, the other is automatically killed too.

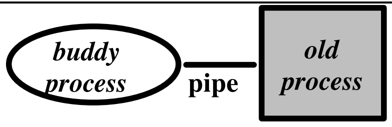


Figure 17-5: A simple way to wrap an old program may be to build a new program that controls the old one through a pipe. The "buddy" process now acts as a proxy for the old process. Performance of pipes is sufficiently high in modern systems to make this approach surprisingly inexpensive. The buddy process is typically very simple and hence is likely to be very reliable; a consequence is that the reliability of the pair (if both run on the same processor) is typically the same as that of the old process.

Interposition wrappers may also be supported by the operating system. Many operating systems provide some form of packet filter capability, which would permit a user-supplied procedure to examine incoming or outgoing messages, selectively operating on them in various ways. Clearly, a packet filter can implement wrapping. The streams communication abstraction in UNIX, discussed in Chapter 5, supports a related form of wrapping, in which streams modules are pushed and popped from a protocol stack. Pushing a streams module onto the stack is a way of "wrapping" the stream with some new functionality implemented in the module. The stream still looks the same to its users, but its behavior changes.

Interposition wrappers have been elevated to a real art form in the Chorus operating system [RAAB88, RAAH88], which is object oriented and uses object invocation for procedure and system calls. In Chorus, an object invocation is done by specifying a procedure to invoke and providing a handle referencing the target object. If a different handle is specified for the original one, and the object referenced has the same or a superset of the interface of the original object, the same call will pass control to a new object. This object now represents a wrapper. Chorus uses this technique extensively for a great variety of purposes, including the sorts of security and reliability objectives cited above.

17.1.1.5 Wrapping Communication Infrastructures: Virtual Private Networks

Sometime in the near future, it may become possible to wrap an application by replacing the communications infrastructure it uses with a virtual infrastructure. Much work on the internet and on telecommunications information architectures is concerned with developing a technology base that can support virtual private networks, having special security or quality of service guarantees. A virtual network could also wrap an application, for example by imposing a firewall interface between certain classes of components, or by encrypting data so that intruders can be prevented from eavesdropping.

The concept of a virtual private network runs along the following lines. In Section 10.8 we saw how agent languages such as Java permit a server to download special purpose display software into a client's browser. One could also imagine doing this into the network communication infrastructure itself, so that the network routing and switching nodes would be in a position to provide customized behavior on behalf of specialized applications that need particular, non-standard, communication features. We call the resulting structure a virtual private network because, from the perspective of each individual user, the network seems to be a dedicated one with precisely the properties needed by the application. This is a virtual behavior, however, in the sense that it is superimposed on the a physical network of a more general nature. Uses to which a virtual private network (VPN) could be put include the following:

- Support for a security infrastructure within which only legitimate users can send or receive messages. This behavior might be accomplished by requiring that messages be signed using some form of VPN key, which the VPN itself would validate.
- Communication links with special video-transmission properties, such as guarantees of limited loss rate or real-time delivery (so-called "isochronous" communication).
- Tools for stepping down data rates when a slow participant conferences to a set of individuals who all share much higher speed video systems. Here, the VPN would filter the video data, sending through only a small percentage of the frames to reduce load on the slow link.
- Concealing link-level redundancy from the user. In current networks, although it is possible to build a redundant communications infrastructure that will remain conected even if a link fails, one often must assign two IP addresses to each process in the network, and the application itself must sense that problems have developed and switch from one to the other explicitly. A VPN could hide this mechanism, providing protection against link failures in a manner transparent to the user.

17.1.1.6 Wrappers: Some Final Thoughts

Wrappers will be familiar to the systems engineering community, which has long employed these sorts of "hacks" to attach an old piece of code to a new system component. By giving the approach an appealing name, we are not trying to suggest that it represents a breakthrough in technology. On the contrary, the point is simply that there can be many ways to introduce new technologies into a distributed system and not all of them require that the system be rebuilt from scratch.

Given the option, it is certainly desirable to build with the robustness goals and tools that will be used in mind. But lacking that option, one is not necessarily forced to abandon the use of a robustness enhancing tool. There are often back-door mechanisms by which such tools can be slipped under the covers or otherwise introduced in a largely transparent, non-intrusive manner. Doing so will preserve the large investment that an organization may have made in its existing infrastructure and applications, and hence should be viewed as a positive option, not a setback for the developer who seeks to harden a system. Preservation of the existing technology base must be given a high priority in any distributed systems development effort, and wrappers represent an important tool in trying to accomplish this goal.

17.1.2 Introducing Robustness in Wrapped Applications

Our purpose in this textbook is to understand how reliability can be enhanced through the appropriate use of distributed computing technologies. How do wrappers help in this undertaking? Examples of robustness properties that wrappers can be used to introduce into an application include the following:

- *Fault-tolerance*. Here, the role of the wrapper is to replace the existing I/O interface between an application and its external environment with one that replicates inputs so that each of a set of replicas of the application will see the same inputs. The wrapper also plays a role in "collating" the outputs, so that a replicated application will appear to produce a single output, albeit more reliably than if it were not replicated. To this author's knowledge, the first such use was in a protocol proposed by Borg as part of a system called Aurogen [BBG83, BBGH85], and the approach was later generalized by Eric Cooper in his work on a system called Circus at Berkeley [Coo87], and in the Isis system developed by the author at Cornell University [BJ87a]. Generally, these techniques assume that the wrapped application is completely deterministic, although later we will see an example in which a wrapper can deal with non-determinism by carefully tracing the non-deterministic actions of a primary process and then replaying those actions in a replica.

- *Caching*. Many applications use remote services in a client-server manner, through some form of RPC interface. Such interfaces can potentially be wrapped to extend their functionality. For example, a database system might evolve over time to support caching of data within its clients, to take advantage of patterns of repeated access to the same data items, which are common in most distributed applications. To avoid changing the client programs, the database system could wrap an existing interface with a wrapper that manages the cached data, satisfying requests out of the cache when possible and otherwise forwarding them to the server. Notice that the set of clients managing the same cached data item represent a form of process group, within which the cached data can be viewed as a form of replicated data.

- *Security and authentication*. A wrapper that intercepts incoming and outgoing messages can secure communication by, for example, encrypting those messages or adding a signature field as they depart, and decrypting incoming messages or validating the signature field. Invalid messages can either be discarded silently, or some form of I/O failure can be reported to the application program. This type of wrapper needs access to a cryptographic subsystem for performing encryption or generating signatures. Notice that in this case, a single application may constitute a form of *security enclave* having the property that all components of the application share certain classes of cryptographic secrets. It follows that the set of wrappers associated with the application can be considered as a form of process group, despite the fact that it may not be necessary to explicitly represent that group at runtime or communicate to it as a group.

- *Firewall protection*. A wrapper can perform the same sort of actions as a firewall, intercepting incoming or outgoing messages and applying some form of filtering to them, passing only those messages that satisfy the filtering criteria. Such a wrapper would be placed at each of the I/O boundaries between the application and its external environment. As in the case of the security enclave just mentioned, a firewall can be viewed as a set of processes that ring a protected application, or that encircle an application to protect the remainder of the system from its potentially unauthorized behavior. If the ring contains multiple members — multiple firewall processes — the structure of a process group is again present, even if the group is not explicitly represented by the system. For example, all firewall processes need to use consistent filtering policies if a firewall is to behave correctly in a distributed setting.

- *Monitoring and tracing or logging*. A wrapper can monitor the use of a specific interface or set of interfaces, and triggering actions under conditions that depend on the flow of data through those interfaces. For example, a wrapper could be used to log the actions of an application for purposes of tracing the overall performance and efficiency of a system, or in a more active role, could be used to enforce a security policy under which an application has an associated behavioral profile, and in which deviation from that profile of expected behavior potentially triggers interventions by an oversight mechanism. Such a security policy would be called an *in-depth* security mechanism, meaning that unlike a security policy applied merely at the perimeter of the system, it would continue to be applied in an active way throughout the lifetime of an application or access to the system.

- *Quality of service negotiation*. A wrapper could be placed around a communication connection for which the application has implicit behavioral requirements, such as minimum performance, throughput, or loss rate requirements, or maximum latency limits. The wrapper could then play a role either in negotiation with the underlying network infrastructure to ensure that the required quality of service is provided, or in triggering reconfiguration of an application if the necessary quality of service cannot be obtained. Since many applications are build with *implicit* requirements of this sort, such a wrapper would really play the role of making *explicit* an existing (but not expressed) aspect of the application. One reason that such a wrapper might make sense would be that future networks may be able to offer guarantees of quality of service even when current networks do not. Thus, an existing application might in the future be "wrapped" to take advantage of those new properties with little or no change to the underlying application software itself.

- *Language level wrappers*. Wrappers can also operate at the level of a programming language, or an interpreted runtime environment. In Chapter 18, for example, we will describe a case in which the Tcl/Tk programming language was extended to introduce fault-tolerance by wrapping some of its standard interfaces with extended ones. Similarly, we will see that fault-tolerance and load-balancing can often be introduced into object-oriented programming languages, such as C++, Ada, or SmallTalk, by introducing new object classes that are transparently replicated or that use other transparent extensions of their normal functionality. An existing application can then benefit from replication by simply using these objects in place of the ones previously used.

The above is at best a very partial list. What it illustrates is that given the idea of using wrappers to reach into a system and manage or modify it, one can imagine a great variety of possible interventions that would have the effect of introducing fault-tolerance or other forms of robustness, such as security, system management, or explicit declaration of requirements that the application places on its environment.

These examples also illustrate another point: when wrappers are used to introduce a robustness property, it is often the case that some form of distributed process group structure will be present in the resulting system. As noted above, the system may not need to actually represent such a structure and may not try to take advantage of it *per-se*. However, it is also clear that the ability to represent such structures and to program using them explicitly could confer important benefits on a distributed environment. The wrappers could, for example, use consistently replicated and dynamically updated data to vary some sort of security policy. Thus, a firewall could be made dynamic, capable of varying its filtering behavior in response to changing requirements on the part of the application or environment. A monitoring mechanism could communicate information among its representatives in an attempt to detect correlated behaviors or attacks on a system. A caching mechanism can ensure the consistency of its cached data by updating it dynamically.

Wrappers do not always require process group support, but the two technologies are well matched to one-another. Where a process group technology is available, the developer of a wrapper can potentially benefit from it to provide sophisticated functionality that would otherwise be difficult to implement. Moreover, some types of wrappers are only meaningful if process group communication is available.

17.1.3 Toolkit Technologies

In the introduction to this chapter, we noted that wrappers will often have limitations. For example, although it is fairly easy to use wrappers to replicate a completely deterministic application to make it fault-tolerant, it is much harder to do so if an application is not deterministic. And, unfortunately, many applications are non-deterministic for obvious reasons. For example, an application that is sensitive to time (e.g. timestamps on files or messages, clock values, timeouts) will be non-deterministic to the degree that it is difficult to guarantee that the behavior of a replica will be the same without ensuring that the replica sees the same time values and receives timer interrupts at the same point in its execution. The UNIX *select* system call is a source of non-determinism, as are interactions with devices. Any time an application uses *ftell* to measure the amount of data available in an incoming communication connection, this introduces a form of non-determinism. Asynchronous I/O mechanisms, common in many systems, are also potentially non-deterministic. And parallel or preemptive multithreaded applications are potentially the most nondeterministic of all.

In cases such as these, there may be no obvious way that a wrapper could be introduced to transparently confer some desired reliability property. Alternatively, it may be possible to do so but impractically costly or complex. In such cases, it is sometimes hard to avoid building a new version of the application in question, in which explicit use is made of the desired reliability technology. Generally, such approaches involve what is called a *toolkit* methodology.

In a toolkit, the desired technology is prepackaged, usually in the form of procedure calls (Figure 17-6). These provide the functionality needed by the application, but without requiring that the user understand the reasoning that lead the toolkit developer to decide that in one situation, *cbcast* was a good choice of communication primitive, but that in another, *abcast* is a better option, and so forth. A toolkit for managing replicated data might offer an abstract data type called a replicated data item, perhaps with some form of "name" and some sort of representation, such as a vector or an *n*-dimensional array. Operations appropriate to the data type would then be offered: *UPDATE, READ,* and *LOCK* being the obvious ones for a replicated data item (in addition to such additional operations as may be needed to initialize the object, detach from it when no longer using it, etc). Other examples of typical toolkit functionality might include transactional interfaces, mechanisms for performing distributed load-balancing or fault-tolerant request execution, tools for publish/subscribe styles of communication, tuple-space tools implementing an abstraction similar to the one in the Linda tuple-oriented parallel programming environment, etc. The potential list of tools is really unlimited, particularly if such issues as distributed systems security are also considered.

Tool	Description
Load-balancing	Provides mechanisms for building a load-balanced server, which can handle more work as the number of group members increases.
Guaranteed execution	Provides fault-tolerance in RPC-style request execution, normally in a manner that is transparent to the client
Locking	Provides synchronization or some form of "token passing"
Replicated data	Provides for data replication, with interfaces to read and write data, and selectable properties such as data persistence, dynamic uniformity, and the type of data integrity guarantees supported
Logging	Maintains logs and checkpoints and provides playback
Wide-area spooling	Provides tools for integrating LAN systems into a WAN solution
Membership ranking	Within a process group, provides a ranking on the members that can be used to subdivide tasks or load-balance work
Monitoring and control	Provides interfaces for instrumenting communication into and out of a group and for controlling some aspects of communication
State transfer	Supports the transfer of group "state" to a joining process
Bulk transfer	Supports out of band transfer of very large blocks of data
Shared memory	Tools for managing shared memory regions within a process group, which the members can then use for communication that is difficult or expensive to represent in terms of message passing

Figure 17-6: Typical interfaces that one might find in a toolkit for process group computing. In typical practice, a set of toolkits would be needed, each aimed at a different class of problems. The interfaces listed above would be typical for a server replication toolkit, but might not be appropriate for building a cluster-style multimedia video server or a caching web proxy with dynamic update and document consistency guarantees.

Toolkits often include other elements of a distributed environment, such as a name space for managing names of objects, a notion of a communications endpoint object, process group communication support, message data structures and message manipulation functionality, lightweight threads or other event notification interfaces, and so forth. Alternatively, a toolkit may assume that that the user is already working with a distributed computing environment, such as the DCE environment or SUN Microsystem's ONC environment. The advantage of such an assumption is that it reduces the scope of the toolkit itself to those issues explicitly associated with its model; the disadvantage being that it compels the toolkit user to also use the environment in question, reducing portability.

17.1.4 Distributed Programming Languages

The reader may recall the discussion of agent programming languages and other "*Fourth generation languages"* (4GL's), which package powerful computing tools in the form of special-purpose programming environments. Java is the best known example of such a language, albeit aimed at a setting in which reliability is taken primarily to mean "security of the user's system against viruses, worms, and other forms of intrusion." Power Builder and Visual Basic will soon emerge as important alternatives to Java. Other sorts of agent oriented programming languages include Tcl/Tk [Ous94] and TACOMA [JvRS95].

Although existing distributed programming languages lack group communication features and few make provisions for reliability or fault-tolerance, one can extend many such languages without difficult. The resulting enhanced language can be viewed as a form of distributed computing toolkit in which the tools are tightly integrated with the language. For example, in Chapter 18, we will see how the Tcl/Tk GUI development environment was converted into a distributed groupware system by integrating it with Horus. The resulting system is a powerful protyping tool, but in fact could actually support "production" applications as well; Brian Smith at Cornell University is using this infrastructure in support of a new video conferencing system, and it could also be employed as a groupware and computer-supported cooperative work CSCW programming tool.

Similarly, one can integrate a technology such as Horus into a web browser such as the Hot Java browser, in this way providing the option of group communication support directly to Java applets and applications. We'll discuss this type of functionality and the opportunities it might create in Section 17.4.

17.2 Wrapping a Simple RPC server

To illustrate the idea of wrapping for reliability, consider a simple RPC server designed for a financial setting. A common problem that arises in banking is to compute the theoretical price for a bond; this involves a calculation that potentially reflects current and projected interest rates, market conditions and volatility (expected price fluctuations), dependency of the priced bond on other securities, and myriad other factors. Typically, the necessary model and input data is represented in the form of a server, which clients access using RPC requests. Each RPC can be reissued as often as necessary: the results may not be identical (because the server is continuously updating the parameters to its model) but any particular result should be valid for at least a brief period of time.

Now, suppose that we have developed such a server, but that only after putting it into operation began to be concerned about its availability. A typical scenario might be that the server has evolved over time, so that although it was really quite simple and easy to restart after crashes when first introduced, it can now require an hour or more to restart itself after failures. The result is that if the server does fail, the disruption could be extremely costly.

An analysis of the causes of failure is likely to reveal that the server itself is fairly stable, although a low residual rate of crashes is observed. Perhaps there is a lingering suspicion that some changes recently introduced to handle the possible unification of European currencies after 1997 are buggy, and are causing crashes. The development team is working on this problem and expects to have a new version in a few months, but management, being pragmatic, doubts that this will be the end of the software reliability issues for this server. Meanwhile, however, routine maintenance and communication link problems are believed to be at least as serious a source of downtime. Finally, although the server hardware is relatively robust, it has definitely caused at least two major outages during the past year, and loss of power associated with a fire triggered additional downtime recently.

In such a situation, it may be extremely important to take steps to improve server reliability. But clearly, rebuilding the server from scratch would be an impractical step given the evolutionary nature of the software that it uses. Such an effort could take months or years, and when traders perceive a problem, they are rarely prepared to wait years for a solution.

The introduction of reliable hardware and networks could improve matters substantially. A dual network connection to the server, for example, would permit messages to route around problematic network components such as faulty routers or damaged bridges. But the software and management failures would remain an issue. Upgrading to a fault-tolerant hardware platform on which to run the server would clearly improve reliability but only to a degree. If the software is in fact responsible for many of the failures that are being observed, all of these steps will only eliminate some fraction of the outages.

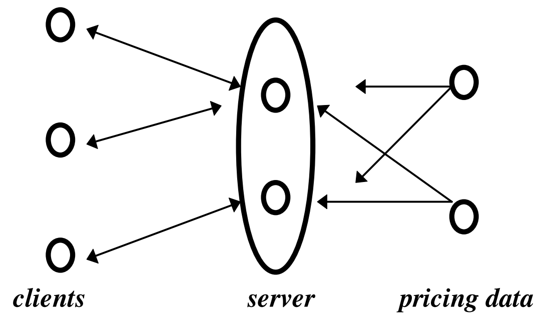


Figure 17-7: A client-server application can be wrapped to introduce fault-tolerance and load-balancing with few or no changes to the existing code.

An approach that replicates the server using wrappers, however, might be very appealing in this setting. As stated, the server state seems to be dependent on pricing inputs to it, but not on queries. Thus, a solution such as the one in Figure 17-7 can be considered. Here, the inputs that determine server behavior are replicated using broadcasts to a process group. The queries are load-balanced by directing the queries for any given client to one or another member of the server process group. The architecture has substantial design flexibility in this regard: the clients can be managed as a group, with their queries carefully programmed to match each client to a different, optimally selected, server. Alternatively, the clients can use a random policy to issue requests to the servers. If a server is unreasonably slow to respond, or has clearly failed, the same request could be reissued to some other server (or, if the request itself may have caused the failure, a slightly modified version of the request could be issued to some other server). Moreover, the use of wrappers makes it easy to see how such an approach can be introduced transparently (without changing existing server or client code). Perhaps the only really difficult problem would be to restart a server while the system is already active.

In fact, even this problem may not be so difficult to solve. The same wrappers that are used to replace the connection from the data sources to the server with a broadcast to the replicated server group can potentially be set up to log input to the server group members in the order that they are delivered. To start a new server, this information can be transferred to it using a state transfer from the old members, after which any new inputs can be delivered. When the new server is fully initialized, a message can then be sent to the client wrappers informing them that the new server is able to accept requests. To optimize this process, it may be possible to launch the server using a checkpoint, replaying only those logged events that changed the server state after the checkpoint was created. These steps would have the effect of minimizing the impact of the slow server restart on perceived system performance.

This discussion is not entirely hypothetical. The author is aware of a number of settings in which problems such as this were solved precisely in this manner. The use of wrappers is clearly an effective way to introduce reliability or other properties (such as load-balancing) transparently, or nearly so, in complex settings characterized by substantial preexisting applications.

17.3 Wrapping a Web Server

The techniques of the preceding section could also be used to develop a fault-tolerant version of a web server. However, whereas the example presented above concerned a database server that was used only for queries, many web servers also offer applications that become active in response to data submitted by the user through a form-fill or similar interface. To wrap such a server for fault-tolerance, one would need to first confirm that its implementation is deterministic if these sorts of operations are invoked in the same order at the replicas. Given such information, the *abcast* protocol could be used to ensure that the replicas all see the same inputs in the same order. Since the replicas would now take the same actions against the same state, the first response received could be passed back to the user; subsequent duplicate responses can be ignored.

A slightly more elaborate approach is commonly used to introduce load-balancing within a set of replicated web servers for query accesses, while fully replicating update accesses to keep the copies in consistent states. The HTTP protocol is sufficiently sophisticated to make this an easy task: for each retrieval (*get)* request received, a front-end web server simply returns a different server's address from which that retrieval request should be satisfied, using a "temporary redirection" error code. This requires no changes to the http protocol, web browsers, or web servers, and although purists might consider it to be a form of "hack", the benefits of introducing load-balancing without having to redesign HTTP are so substantial that within the Web development community, the approach is viewed as an important design paradigm. In the terminology of this chapter, the front-end server "wraps" the cluster of back-end machines.

17.4 Hardening Other Aspects of the Web

A wrapped Web server just hints at the potential that group communication tools may have in future enterprise uses of the Web. As seen in

Application domain	Uses of process groups
Server replication	• High availability, fault-tolerance • State transfer to restarted process • Scalable parallelism and automatic load balancing • Coherent caching for local data access • Database replication for high availability
Data dissemination	• Dynamic update of documents in the Web, or of fields in documents • Video data transmission to group conference browser's with video viewers • Updates to parameters of a parallel program • Updates to spread-sheet values displayed to browsers showing financial data • Database updates to database GUI viewers • Publish/subscribe applications
System management	• Propagate management information base (MIB) updates to visualization systems • Propogate knowlwdge of the set of servers that compose a service • Rank the members of a server set for subdividing the work • Detecting failures and recoveries and triggering consistent, coordinated action • Coordination of actions when multiple processes can all handle some event • Rebalancing of load when a server becomes overloaded, fails, or recovers
Security applications	• Dynamically updating firewall profiles • Updating security keys and authorization information • Replicating authorization servers or directories for high availability • Splitting secrets to raise the barrier faced by potential intruders • Wrapping components to enforce behavior limitations (a form of firewall that is placed close to the component and monitors the behavior of the application as a whole)

Figure 17-8: Potential uses of groups in Internet Systems

Application domain	**Uses of process groups**
Server replication	• High availability, fault-tolerance • State transfer to restarted process • Scalable parallelism and automatic load balancing • Coherent caching for local data access • Database replication for high availability
Data dissemination	• Dynamic update of documents in the Web, or of fields in documents • Video data transmission to group conference browser's with video viewers • Updates to parameters of a parallel program • Updates to spread-sheet values displayed to browsers showing financial data • Database updates to database GUI viewers • Publish/subscribe applications
System management	• Propagate management information base (MIB) updates to visualization systems • Propogate knowlwdge of the set of servers that compose a service • Rank the members of a server set for subdividing the work • Detecting failures and recoveries and triggering consistent, coordinated action • Coordination of actions when multiple processes can all handle some event • Rebalancing of load when a server becomes overloaded, fails, or recovers
Security applications	• Dynamically updating firewall profiles • Updating security keys and authorization information • Replicating authorization servers or directories for high availability • Splitting secrets to raise the barrier faced by potential intruders • Wrapping components to enforce behavior limitations (a form of firewall that is placed close to the component and monitors the behavior of the application as a whole)

Figure 17-8, Figure 17-9 and Figure 17-10, the expansion of the Web into groupware applications and environments, computer-aided cooperative work (CSCW), and dynamic information publication applications, all create challenges that the sorts of tools we developed in Chapters 13-16 could be used to solve.

Today, a typical enterprise that makes use of a number of Web servers treats each server as an independently managed platform, and has little control over the cache coherency policies of the Web proxy servers that reside between the end-user and the Web servers. With group replication and load-balancing, we could transform these Web servers into fault-tolerant, parallel processing systems. Such a step would bring benefits such as high availability and scalable performance, enabling the enterprise to reduce the risk of server overload when a popular document is under heavy demand. Looking to the future, Web servers will increasingly be used as video servers, capturing video input (such as conferences and short presentation by company experts on topics of near-term interest, news stories off the wire, etc), in which case such scalable parallelism may be critical to both data archiving (which often involves computationally costly techniques such as compression) and playback.

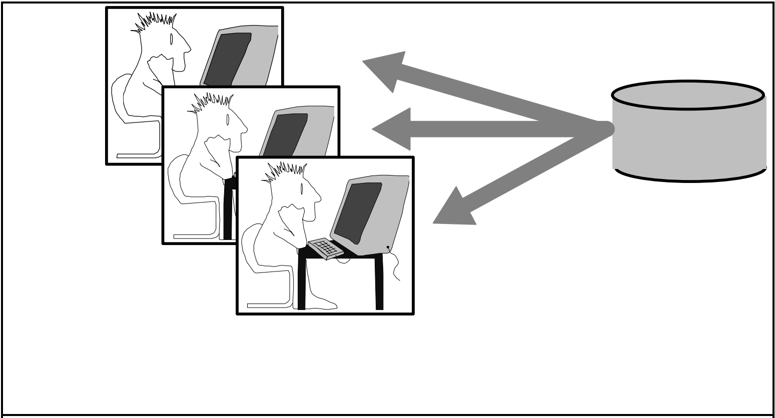


Figure 17-9: Web server transmits continuous updates to documents or video feeds to a group of users. Depending upon the properties of the group-communication technology employed, the user's may be guaranteed to see identical sequences of input, to see data synchronously, security from external intrusion or interference, and so forth. Such a capability is most conveniently packaged by integrating group communication directly into a web agent language such as Java or Visual Basic, for example by extending the Hot Java browser with group communication protocols that could then be used through a groupware API.

Wide-area group tools could also be used to integrate these servers into a wide-area architecture that would be seamless, presenting users with the abstraction of a single, highly consistent, high availability Web service, and yet internally self-managed and structured. Such a multi-server system might implement data migration policies, moving data to keep it close to the users that demand it most often, and wide-area replication of critical information that is widely requested, while also providing guarantees of rapid update anc consistency. Later, we will be looking at security technologies that could also be provided through such an enterprise architecture, permitting a company to limit access to its critical data to just those users who have been authorized, for example through provision of a Fortezza card (see Section 19.3.4).

Turning to the caching Web proxies, group communication tools would permit us to replace the standard caching policy with a stateful coherent caching mechanism. In contrast with the typical situation today, where a Web page may be stale, such an approach would allow a server to reliably send out a message that would invalidate or refresh any cached data that has changed since it was copied. Moreover, by drawing on CORBA functionality, one could begin to deal with document groups (sets of documents with hyperlinks to one-another) and over multi-document structures in a more sophisticated manner.

Group communication tools can also play a role in the delivery of data to end-users. Consider, for example, the idea of treating a message within a group as a Java-style self-displaying object, a topic we touched upon above. In effect, the server could manufactor and broadcast to a set of users an actively self-constructed entity. Now, if group tools are available within the browsers themselves, these applets could cooperate with one-another to animate a scene in a way that all participants in the group conferencing session can observe, or to mediate among a set of concurrent actions initiated by different users. User's would download the current state of such an applet and then receive (or generate) updates, observing these in a consistent order with respect to other concurrent users. Indeed, the applet itself could be made self-modifying, for example by sending out new code if actions taken by the users demand it (zooming for higher resolution, for example, might cause an applet to replace itself with one suited for accurate display of fine grained detail).

Thus, one could imagine a world of active multi-documents in which the objects retrieved by different users would be mutually consistent, dynamically updated, able to communicate with one another, and in which updates originating on the Web servers would be automatically and rapidly propagated to the documents themselves. Such a technology would permit a major step forward in conferencing tools, and is likely to be needed in some settings, such as telemedicine (remote surgery or consultations), military strategic/tactical analysis, and remote teleoperation of devices. It would enable a new generation of interactive multiparticipant network games or simulations, and could support the sorts of cooperation needed in commercial or financial transactions that require simultaneous actions in multiple markets or multiple countries. The potential seems nearly unlimited. Moreover, all of these are applications that would appear very difficult to realize in the absense of a consistent group communication architecture, and that demand a high level of reliability in order to be useful within the intended community.

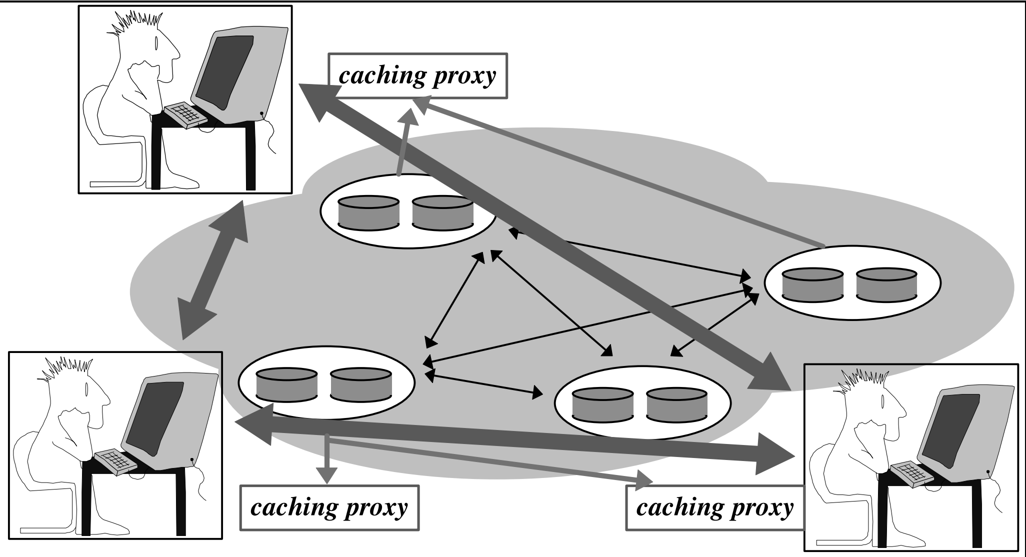


Figure 17-10: Potential group communication uses in Web applications arise at several levels. Web servers can be replicated for fault-tolerance and load-balancing, or integrated into wide-area structures that might span large corporations with many sites. Caching web proxies could be "fixed" to provide guarantees of data consistenyc, and digital encryption or signatures used to protect the overall enterprise against intusion or attack. Moreover, one can forsee integrating group communication directly into agent languages like Java, thereby creating a natural tool for building cooperative groupware applications. A key to successfully realizing this vision will be to design wrappers or toolkit API's that are both natural and easy to use for the different levels of abstraction and purposes seen here: clearly, the tools one would want to use in building an interactive multimedia groupware object would be very different from those one would use to replicate a Web server.

Obviously, our wrapped Web server represents just the tip of potentially large application domain. While it is difficult to say with any certainty that this type of system will ever be of commercial importance, or to predict the timeframe in which it might become real, it seems plausible that the pressures that today are pushing more and more organizations and cooperations onto the Web will tomorrow translate into pressure for consistent, predictable, and rapidly updated groupware tools and objects. The match of the technologies we have presented with this likely need is good, although the *packaging* of group communication tools to work naturally and easily within such applications will certainly demand additional research and development. In particular, notice that the tools and API's that one might desire at the level of a replicated Web server will look completely different from those that would make sense in a multimedia groupware conferencing system. This is one reason that systems like Horus need flexibility, both at the level of how they behave and how they look. Nonetheless, the development of appropriate API's ultimately seems like a small obstacle. The author is confident that group communication tools will come to play a large role in the enterprise Web computing systems of the coming decades.

17.5 Unbreakable Stream Connections

Motivated by Section 17.4, we now consider a more complex example. In Chapter 5 we discussed unreliability issues associated with stream style communication. Above, we discussed extensions to web servers that might make them reliable. However, consider the client browser: it will typically connect to such a server through a stream (a TCP connection, to be specific). Thus it makes sense to ask how group communication tools can help us overcome some of the problems we noted in our original discussion of streams and their behavior when failures occur. After all, if we want our web technology to be *completely* reliable and to handle failures in a completely transparent manner, we will need to solve this problem.

Our analysis will lead to a mixed conclusion, and indeed one reason for including this section in the textbook is to illustrate the challenges created by "real world" considerations, and the sort of tradeoffs that result. A constraint underlying the discussion will be the assumption that we are concerned with a client and a server, and that the server (but not the client) is to be replicated for increased availability. The client, on the other hand, uses a completely standard and unmodified implementation of some stream-style reliable protocol. Below, we will use TCP as our example for such a protocol, although the same discussion would make sense for other stream-style protocols. This constraint prevents us from using a solution such as the protocol of Section 13.13.

Notice, however, that these constraints are somewhat arbitrary. While there may be important benefits in avoiding modification of the client systems, these benefits are unlikely to appeal to a developer who will need to pay a high cost, in complexity or performance, for the transparency afforded by such a solution. Moreover, in a world where servers can download agents to the client, it may be quite simple to download a special purpose applet that causes the client's system to simply talk to the server through some new, special-purpose protocol. This alternative will underlie much of the discussion of this chapter. We will see that under certain conditions, a very transparent stream protocol from the client to the server can be made reliable at low cost, and this class of solutions will be discussed in some detail. Under other conditions, we will encounter dead-ends in which either the complexity or performance overheads exceed the likely threshold of pain at which the non-member to group protocols would make more sense. Such of solutions are consequently of limited practical interest, and we will discuss them only superficially.

17.5.1 Reliability Options for Stream Communication

What would it mean to say that a stream connection is "more reliable" than the ones considered in Chapter 5? Two types of answers make sense to this author. A sensible starting point would be to overcome the failure reporting problems of stream connections, by "rewiring" the failure mechanisms of some standard stream protocol to the GMS input and outs. More precisely, we can introduce wrappers for this purpose. Depending on how the streams package was implemented, this might be very easy (i.e. if the streams module is implemented using source code available to the developer and that can easily be modified), but would more often represent a tremendously difficult undertaking. The problem is that standard computer systems generally place such code inside the O/S kernel and protect it against modification by users.

In light of our constraint that the client be unmodified, this rewiring will only occur within the server. Nonetheless, it can have the effect of avoiding inconsistent failure scenarios, if a client is connected to multiple servers. In such cases, we will now be sure that if one server concludes that a client has failed, all servers will react consistently.

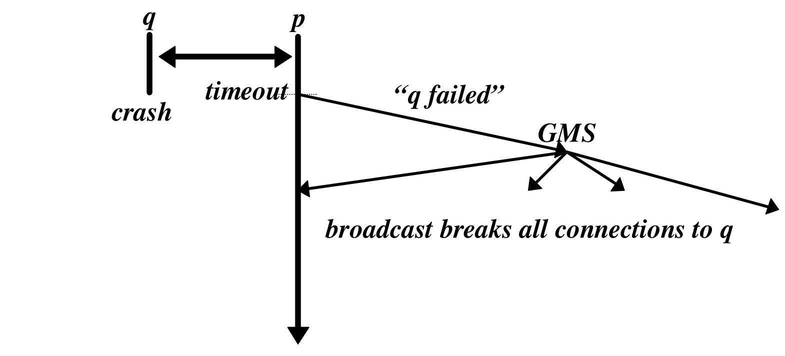


Figure 17-11: Modified streams protocol reports detected failures to the GMS, which breaks all connections to a failed or excluded process simultaneously. To introduce this behavior, the original interfaces by which the protocol detects and reports failures, within itself, would be wrapped to interconnect the detection mechanism with the GMS, and to connect the GMS output back to the stream protocol.

The use of wrappers to provide consistent failure reporting requires that code be added to "intercept" failure detections in the streams package, modifying the reporting of such events so that they become upcalls to the GMS service. To do this, one would first modify the channel protocol so that each process using the protocol registers itself with the GMS, and so that any process *p* connected to some other process *q* asks the GMS to monitor *q* and to report failures. Next, suppose that the original code implementing the connection had a procedure called *break_connection* that gets called when the number of retransmission attempts for some packet exceeds a threshold. The developer would modify these parts of the cold to instead issue upcalls to the GMS service, informing it that the endpoint has apparently failed. This will cause the GMS to run a protocol excluding the endpoint, as discussed above, and eventually to issue downcalls *to all the processes monitoring the endpoint that has been excluded*. When the GMS reports that the endpoint of the connection has "failed" the old code associated with *break_connection* would be executed. Notice, however, that in the original implementation, each process independently detects (apparent) failures and immediately executes *break_connection*. With this change, each process continues to independently detect failures, but *all* processes execute *break_connection* if any does so. This interaction is illustrated in Figure 17-11. Moreover, the example isn't completely hypothetical: there are public-domain implementations of the TCP protocol stack that run in user-space, and the author's students have carried out this transformation successfully, and demonstrated that the resulting technology indeed exhibits consistent failure reporting semantics. On the other hand, one could question whether the benefits of this change justify the effort.

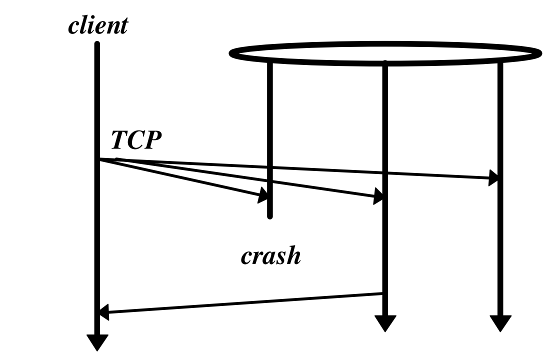


Figure 17-12: A more elaborate solution to the reliable streams problem. In the desired protocol, the client uses a completely standard streams protocol, such as TCP, but the messages are delivered as reliable broadcasts within the group. Unless all members fail, the client sees an unbroken TCP connection to what appears to be a continuously available server. Only if all members fail does the channel break from the perspective of the client.

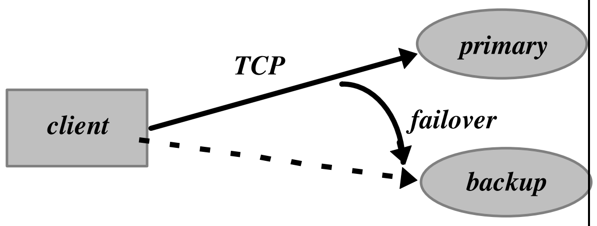


Figure 17-13: A successful implementation of the reliable streams protocol would provide clients with the illusion of a completely transparent "failover", under which their TCP connection to a primary server would automatically and seamlessly switch from a primary server to the backup if a failure occurs..

A more ambitious goal would be to support a stream connection to a group of processes, having the property that the members can now emulate the behavior of a single very reliable, non-replicated, service. Solving this problem potentially involves much more effort than for our initial intervention. Here, we need a way to ensure that the stream connection (say, TCP) survives the failure of subsets of the members, and that the members can stay in consistent states even when failures occur. We will want our solution to be easy to use (in particular, it would be best if the clients of such a connection could employ standard versions of the streams protocol, with all the changes being made on the server side). And, we will want the solution to be as efficient as possible, so that the cost of using a reliable service through such a reliable connection is as close as possible to the cost of using an unreliable service through a conventional stream connection. Such an arrangement is illustrated in Figure 17-12 and Figure 17-13; the former figure shows how this might work, while the latter shows how the resulting structure appears to the client system.

Recall that this problem would be straightforward to solve if we had the freedom to modify the application on the client side of the connection. In that case, it would suffice to implement an interface that looks like the standard streams interface for the client computing platform, but operates using one of the client to group protocols developed in Section 13.13. Thus, in the specific case of a Java-enabled web browser, where there is a realistic option for downloading an agent that can use such a non-standard protocol to talk to the server, there is a relatively simple solution to this problem.

The same problem is quite a bit harder if our goal is to fool a standard streams protocol like TCP into believing that it is communicating with a single non-faulty process using that protocol, when in fact the destination is group. As we will see below, a completely general solution may be so costly that implementation of a "direct" client to group broadcast would be highly advantageous. However, for a somewhat constrained class of stream protocols and applications, a very transparent, very general solution is achievable.

17.5.2 An Unbreakable Stream That Mimics TCP

To address this issue, we will need to assume that there is a version of the stream protocol that has been "isolated" in the form of a protocol module with a well-defined interface. To simplify the discussion, assume that we are talking about a TCP protocol (other stream protocols could be treated the same way; only the details would change). At the bottom, the protocol accepts incoming IP packets from the network, and sends back IP packets containing acknowledgements, retransmission requests, and outgoing TCP data (outgoing "segments"). Internally, the module has an interface to the timer subsystem of the machine on which it is running, using this to read the time and to schedule timer interrupts. Fortunately, not many protocol implementations of this sort make use of threaded concurrency, but if the module in question does so, the interface from it to the subsystem that implements lightweight threads would also have to be considered as part of its interface to the outside world. Finally, there is the interface by which the module interacts with the application process: this consists of its read and write interface, perhaps a control interface (this would implement the UNIX *ioctl, ftell,* and *select* interfaces, or the equivalent operations on other operating systems). This environment is seen in Figure 17-14.

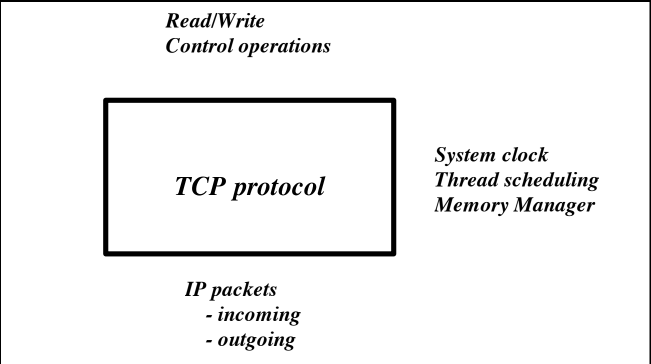


Figure 17-14: The TCP protocol can be viewed as a black box with interfaces to its environment. Although these interfaces are non-trivial, they are not overwhelmingly so.

17.5.3 Non-Determinism and Its Consequences

To use the load-balancing or primary-backup replication techniques presented earlier, together with a fault-tolerance scheme based on wrappers, we need to understand how to control any non-determinism associated with this protocol. Specifically, let's assume that we intercept incoming events by replacing the various interfaces that connect the TCP protocol to the outside world with modified interfaces that will try and keep a set of backup processes in sync with a primary. How hard would it be to make such a solution work?

Given a TCP protocol that is accurately modeled as above, this problem is not as hard as one might expect. Our enumeration of interfaces has reduced the TCP protocol itself to a state machine that can be thought of as receiving incoming "events" from its varied interfaces, computing, and then performing output events. Even access by the protocol to the clock can be thought of as an output event (sent to the clock) followed by an input event (a response from the clock to the protocol). It follows that we can arrange for a primary copy of the TCP protocol to broadcast a script of the full set of its interactions with the outside world. Such a script would list the events that occurred to the protocol and its actions: first, it received an IP data packet containing the following byte sequence. Then it issued a request to read the local clock. Next, a read request was received from the application; 18 bytes of data were returned.

If a copy of the TCP protocol module has access to such a script — earlier we called this a "trace" — it can precisely emulate the actions of the primary merely by replaying the same input events in the same order. If our interface specification was complete and accurate, the backup will faithfully perform the exact same actions in the same order. This approach can be extended to encapsulate the application process as well: given a complete characterization of the application process' interface to the external environment, the actions taken by the primary copy can be traced in such a manner that the actions of the replicas will emulate it in an accurate manner. An analysis of the protocol itself will generally be needed to convince ourselves that we know precisely how it can be non-deterministic, and that the required information can be encoded into trace messages.

Obviously, there are sources of non-determinism that can be very hard to deal with. Interrupt driven behavior and thread scheduling are two instances of such problems, and any sort of direct access by the driver to hardware properties of the computer or attached peripherals runs the risk of introducing similar problems. One can imagine noting the "time" at which an interrupt occurs, and forcing the backup to replay interrupts at the right time, and similarly noting the time at which thread scheduling actions occur, replaying these in the same manner. Other kinds of non-determinism, on the other hand, may be much easier to deal with. For example, if the application program itself is deterministic, and will write back the identical data if given identical inputs, the main source of non-determinism seen by the protocol stack may be that associated with the relative ordering of timeouts and *write* operations. This ordering information, and the associated clock values when timeouts occur, can be encoded very concisely. Notice, though, that even the knowledge that no timeout occurred before the application sent a message to the client may be significant to the state of the protocol.

17.5.4 Dealing With Arbitrary Non-Determinism

A paper by Bressoud and Schneider recently suggested a way to extend this trace-driven approach to software fault-tolerance to make entire machines fault-tolerant, including the operating system and all the applications running on the machine [BS95]. They do this using special hardware properties of certain classes of modern microprocessors. Their work operates at the level of the CPU itself, and involves noting the time at which interrupts occur. Specifically, the method requires a special hardware register that measures time in machine cycles and is saved as part of the interrupt sequence.

The same register can also be set to a value, in which case an interrupt is generated at the desired "time". Using uses this feature on the backup, the Bressoud and Schneider solution operates by repeatedly setting the cycle counter to the time at which the next interrupt should occur. All other types of interrupts are disabled, and the machine is allowed to execute up to the point when the counter fires. Then, an interrupt indistinguishable from the one that occurred on the primary is generated. The method is most readily applicable to machines with very few I/O connections: ideally, just a communications interface and an interface to the clock. Unfortunately, the hardware required is available only on HP's PA-RISC microprocessors.

Returning to our problem, it is easy to see that the key factor limiting a solution will be the degree of non-determinism present in the TCP protocol. Motivated by Bressoud and Schneider's work, it may sometimes be possible to modify a TCP protocol that includes non-determinism (such as concurrent threads or interrupts) into a protocol that is deterministic and hence describable by a trace (for example, by replacing the threads and interrupts with a non-threaded polling method). As noted in the introduction to this setting, some forms of design complexity are best viewed as an argument for the "non-member to a group" protocols of Section 13.13, and any substantial change to the TCP protocol itself to eliminate non-determinism probably falls into this class of complex interventions that should be viewed with skepticism. In the remainder of this section, we will assume that it is reasonably easy to "trace" the actions of our TCP protocol, and that the volume of trace information is reasonably low; otherwise, the method simply should not be used.

17.5.5 Replicating the IP Address

Our transformation leaves two questions open. First, we need to resolve a simple matter, which is to ensure that the backup will actually receive incoming IP packets after taking over from the primary in the event of a failure. The specific issue is as follows. Normally, the receive side of a stream connection is identifiable by the address (the IP address) to which packets are sent by TCP. This address would normally consist of the IP address of the machine itself, and is "locked" into the TCP protocol of the client system. As a consequence, IP packets sent by the client are only received at one site, which represents a single point of failure for the protocol. We need a way to shift the adress to a different location to enable a backup to take over after a crash.

This problem can be solved by manufacturing "virtual" IP addresses, which don't correspond to any real machine on the network. It turns out that the IP address of a machine is assigned during the boot sequence, and that there is typical some form of protected system call by which a new address can be assigned. Indeed, if a machine resides on multiple networks, it may have multiple IP addresses, since the IP address is typically used to index into routing tables. Thus, it is perfectly practical to assign a single machine a "true" address and one or more virtual addresses. We can use this feature to assign the TCP endpoint such an address, and to reassign that address to a backup after a failure. In UNIX, this is done using the *ifconfig* system command.

17.5.6 Maximizing Concurrency by Relaxing Multicast Ordering

The other lingering problem is concerned with maximizing performance. Had we not intervened to replicate the TCP state machine, it would reside on the critical path that determines I/O latency and throughput from client to server and back. Suddenly, we have modified this path to insert what may turn out to be a large number of multicasts to the replicas. How costly will these be?

Specifically, we need to understand the conditions under which a replicated TCP stack can perform as well, or nearly as well, as a non-replicated one. The rationale is similar to the one we encountered in discussing sources of non-determinism: if the performance hit associated with a reliable stream is high, it makes more sense to simply modify the client to use a protocol that is knowledgeable about the presence of a group.

The cheapest case for our protocol arises when the stream connection is used as a pipe, with uni-directional communication from the client to the server. In this case, notice that all the multicasts are initiated by the primary copy of the TCP protocol stack until a failure occurs. Only then will multicasts begin to be initiated by a backup process, namely the new primary. For this purpose, a sender-ordered protocol would be sufficient, the primitive we called *fbcast*. Now, the costs of *fbcast* arise in several ways. There is the fixed overhead of creating the message and passing it to the local multicast subsystem, which may be as small as a few tens of instructions for a very small message and an efficient implementation of *fbcast*. There is a background cost associated with the protocol, but this would normally not impact the latency seen in the primary server, which is the one measurable by the client. Next, there is a bandwidth cost: every byte that reaches the primary will need to be forwarded to the clients; in some cases this may represent a problem, although it will often be of minor importance because most communication devices are capable of sustaining much higher loads than the TCP protocol itself can produce. Finally, however, there is an issue of waiting for *fbcast* stability that we will discuss momentarily.

To the degree that bandwidth proves to be a problem, one can imagine developing a protocol in which the full group of servers would present the same IP address to the network, much as in the IP multicast protocol discussed earlier. However, with such an approach, there is the risk that some data segments may not reach some of the clients, and there will be a need to retransmit data that any client misses. This starts to sound like a complex and costly undertaking, so in keeping with our initial constraints, we will assume that bandwidth is not a problem. If it is expected to be a problem, a protocol knowledgeable about the presence of a group should be used.

The stability issue to which we alluded is the following. Consider a TCP-level acknowledgment or some other message sent by the TCP protocol, from the primary server to the client. When such a message is received, the client's outgoing window will be updated, clearing frames associated with any data that was acknowledged. If the primary now crashes, there will be no possibility of reconstructing the data that was garbage collected. Thus, we see that before sending any TCP-level message from the primary to the client, the primary should wait until any causally prior messages have reached the backups. This constraint applies both to incoming TCP-level messages from the client to the server, and to trace messages that the primary may have generated in the course of handling incoming data. In the terminology of Chapter 13, we need to be sure that these causally prior messages are "stable" at the backup.

For one-directional streams, this problem does not represent a serious source of delay, because the only messages affected are acknowledgements generated within TCP, and the use of a larger TCP window suffices to hide the higher latency. In effect, there is no situation in which the *application* would be forced to wait for stability of the *fbcast's* that convey trace information to the backup processes. In pipelines with multiple stages, each successive write may need to delay for stability of the prior *fbcast's*, but such delays are likely to be hidden by concurrency. Moreover, one-directional streams are easy to detect, because the protocol can simply assume itself to be in a uni-directional mode until the server attempts to send information back to the client.

In the more general case, however, such as an RPC or object invocation that runs over a stream, it is likely that a single, very small *fbcast* will need to be sent from the primary server to its backups immediately after each *write* operation by the application to the TCP stream. This *fbcast* becomes stable when it has reached its destinations; round-trip times in typical modern multicast systems, like Horus, are in the range of .7ms to 1.4ms for such events. Thus, responses from the server to the client may be delayed by about 1 ms to achieve fault-tolerance. Such a cost may seem small to some users and large to others. It can be viewed as the "price of transparency", since comparable delays would not have arisen in applications where complete transparency on the client side was not an objective. This is illustrated in Figure 17-15.

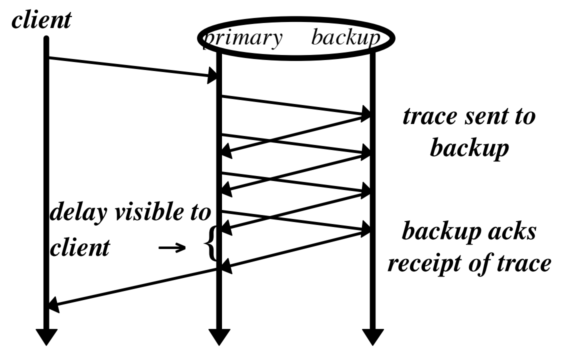


Figure 17-15: The latency introduced by replication is largely invisible to the client. As seen here, most trace information reaches the backup while the primary is still computing. Only the last trace message, sent after the primary issues its reply and before that reply can be sent from primary to client, introduces noticeable latency in the critical path that limits RPC round-trip times over the replicated TCP channel. The problem is that if the trace information causally prior to the reply is lost in a crash, the primary's state cannot be reproduced by the backup. Thus, this information must be stable at the backup before the reply is transmitted. Unfortunately, the use of timeouts in the TCP protocol stack implies that such information will be generated. One can imagine other protocols, however, or optimizations to the TCP protocol, in which the primary would have extremely little or no trace information to send to the backup; replication of such protocols for fault-tolerance would introduce minimal additional latency.

Thus, we find ourselves back at the same limitation cited earlier for the TCP protocol itself. In the simple case of a largely one-way communication channel, and to the degree that the protocol and the application are deterministic, the replication method will have minimal impact on system performance. As we move away from this simple case into more complex ones, the protocol becomes much more complex and imposes increasingly visible overheads that would not have been incurred if the client were simply modified to use a protocol knowledgeable about the presence of a group of servers.

When our modifications are feasible, notice also that the role of the virtual synchrony model is fairly limited. The model lets us overlook issues of agreement on membership in the server group, and lets us implement an *fbcast* protocol that will be delivered atomically before failure notifications occur for the primary server, if it fails. These guarantees greatly simplify the protocol, which isn't all that simple in any case. One could argue that without them, the solution would be impractically complex. The model does not, however, introduce any particularly complex reasoning of its own.

17.5.7 State Transfer Issues

Our discussion overlooked the issues relating to launching of new servers, and of transferring state information to bring them up to date. For applications and protocols in which state is easily represented, the solution presented here can easily be modified to accommodate joins with state transfer to the joining process. Otherwise, it may be best to launch a sufficient set of replicas at the time the connection is first made, so that even if failures occur, the group will still have an adequate number of servers to continue providing response.

17.5.8 Discussion

Although we presented the unbreakable streams problem as a hypothetical one, the author has supervised several research projects that pursued precisely such an analysis and ultimately implemented unbreakable TCP connections for various purposes. One of these projects focused on the case of TCP channels to mobile users whose hand-held computers might need to connect to a succession of base stations as the computer was moved around [CB94], while another looked at TCP in a more standard LAN setting where the focus was on transparent failover of the sort described above [Won95]. Thus, with modest effort, the problem can be solved in the manner outlined above.

Our analysis suggests that in situations where we are not bandwidth limited and where the streams protocol to be modified is available to the developer and has modest non-determinism, a reliability transformation that uses wrappers to introduce fault-tolerance through replication might be worthwhile. The impact on performance and complexity would be reasonably low, and the performance costs may actually be hidden by concurrency. However, if any of these conditions does not hold, the introduced complexity and performance overhead may begin to seem excessive for the degree of transparency such a reliable protocol can afford. Finally, we have observed that there will probably be a small latency impact associated with our transformation in RPC-style interactions, but that but that this cost would probably be hidden in pipeline-style uses of streams, again because of the concurrency achieved between the protocol used to transmit trace information to a replica and the computation occurring in the application itself.

In the specific case of the web browser that connects to an enhanced web server, one might well look at these tradeoffs and these costs, and conclude that the benefit of providing reliability of this sort would not be worth the additional complexity and development effort. First, the possibility of downloading a Java applet that contains a protocol such as the ones developed for "non-member to group communication" in Chapter 13 may represent the easiest path to a solution. Moreover, even in situations where downloading such an applet is unrealistic, the decision to use an unreliable stream might have relatively minor consequences. Such a decision would mean that the web server group would remain available even if some of its members fail, but that a client actually using a web server at the instant it crashes might see the failure much as, today, web accesses often fail for any of a number of reasons. However, the actual frequency of such events will surely be very low, and perhaps the impact when they occur sufficiently minor to make the design point an acceptable one. This is especially likely to be the case if some of the other reasons that a web operation can fail remain in our "hardened" design, for example a DNS timeout. After all, if the hardened system can still fail from time to time (although, one hopes, infrequently!), making a large investment to eliminate what may be a statistically small percentage of the remaining failure cases might not make a lot of sense.

These sorts of tradeoffs are inevitable in complex distributed systems, and it important for the developer to keep one eye on the balance. It is very appealing to imagine a technology that would let us replicate a server, making it fault-tolerant, in a manner that would be completely transparent to its clients. An unbreakable TCP stream connecting the client to the server seems like a natural and inevitably desirable feature. Yet the alternative of building a protocol whereby the client would "know" it was communicating to the group, or that would conceal such interactions beneath a layer of software presenting a streams-like interface, must be weighed against the presumed disadvantages of needing to modify (or at least recompile) the client program.

Pro	• Totally transparent failover • Uninterrupted service to client • Client system not changed at all
Con	• With Java, might have a reasonably easy way to modify the client system • Solution is complex • Performance penalty may be substantial for some patterns of use • Doesn't address "other" causes of failure, such as the ones discussed in Part II of the text

Figure 17-16: Tradeoffs to be considered when looking at the decision to implement a transparently fault-tolerant TCP stream protocol. In many settings, the arguments <u>against</u> doing so would dominate.

In the author's experience, one often encounters such tradeoffs between performance and transparency, or complexity and transparency. Transparency is a good thing, and the use of wrappers can provide a route for very transparent hardening of a distributed system. However, transparency should generally not be elevated to the level of a religion. In effect, a technology should be as transparent as possible, consistent with the need to keep the software used simple and the overheads associated with it low. When these properties fall into question, a less transparent solution should be seriously considered.

17.6 Building a Replicated TCP Protocol Using a Toolkit

The above analysis may leave the reader with the sense that even if one can wrap a TCP protocol for fault-tolerance, the benefits of doing so are outweighed by the complexity of dealing with "black-box" non-determinism. But it is important to keep in mind that the limitations associated with the solution we developed stemmed specifically from the attempt to use a wrapper to avoid modifying the TCP protocol itself. If, in contrast, we were in a position to implement a TCP protocol of our own, the same issues could be circumvented.

In particular, the question of non-determinism underlies most of the performance concerns raised above. Were we to design a TCP implementation *specifically for the purpose of replicating it* we could probably eliminate most or all of this non-determinism through a design that cleverly hides non-deterministic events behind other sorts of communication. For example, our TCP protocol could be designed to check for timeouts and to send acknowledgment messages using timestamps placed on messages by the primary copy of the protocol stack. That is, each time the primary process receives a message or any other form of input, it could timestamp the outgoing copies of that message with the time at which it saw the event. If the protocol used these inputs to trigger timeout-related events, we could avoid the need to send much of the trace information mentioned above.

Conversely, whereas our wrapper would be forced to trace interrupt events and thread switching events in order to overcome non-determinism, an explicitly replicated approach might use concurrent threads only for logically independent tasks, ensuring that in any situation where there is a sensitivity to thread scheduling order, that order is deterministically fixed by the external sequence of events received by the TCP protocol stack.

Another concern of ours was that the application program might sometimes request a full buffer of data and yet be passed a partial buffer by the TCP implementation. Knowing that such a situation creates an overhead in a replicated TCP stream, one might implement the protocol to never return a partial result from a *read* operation unless the remote end of the stream has been closed. Knowing that all reads block until the stream closes or 8kbytes are available for the reader, the non-determinism associated with read requests can be greatly reduced or even eliminated.

These observations having been made, however, it should also be commented that our analysis in this section has been superficial. Moreover, if a toolkit approach is used within the TCP stack on the server side, it may be reasonable to extend the approach to encompass the client side as well. Such an effort clearly represents a potential research (or product) opportunity, and goes beyond any investigation of this topic of which the author is aware. In the interest of brevity, we will not develop this discussion at the present time.

17.7 Reliable Distributed Shared Memory

Distributed shared memories are a “hot topic” in the distributed systems research community. In this section we look at the idea of implementing a wrapper for the UNIX *mmap* (or *shrmem*) functions, which are normally used to map files and memory regions into the address space of user applications and shared between concurrently executing processes. The extension we consider here provides for the sharing of memory-mapped objects over a virtually synchronous communications architecture running on a high speed communications network. One might use such a system as a repository for rapidly changing visual information in the form of web pages: the provider of the information would update a local mapped copy directly in memory, while the subscribers could map the region directly onto the memory of a display device and in this way obtain a direct I/O path between the data source and the remote display. Other uses might include parallel scientific computations in which the shared memory represents the shared state of the parallel computation, a collaborative workplace or virtual reality environment shared between a number of users, a simulation of a conference or meeting room populated by the participants in a teleconference, or some other abstraction.

In studying this problem, we should comment at the outset that the topic is an area of active research by several operating systems groups world-wide [LH89, GLLG90, ABHN91, FZ91, Car93, FMPK95, JKW95] , but that author is not aware of any effort that has looked at the implementation of a *reliable* shared memory using process group technology. To the degree that there has been work on this subject, the emphasis has tended to be on settings in which reliability issues are secondary to questions of functionality and performance. Our goal in this section, then, is to look at another non-trivial example of how group communication might be used to solve a challenging contemporary problem, but not to claim that our solution is a “real one” with known performance and latency properties.

17.7.1 The shared memory wrapper abstraction

As for the case of the unbreakable TCP connection, our solution will start with an appropriate wrapper technology. In many UNIX-like operating systems there is a mechanism available for mapping a file into the memory of a process, sharing memory between concurrently executing processes, or doing both at the same time. The UNIX system calls supporting this functionality are called *shrmem* or *mmap* depending on the version of UNIX one is using; a related interface called *semctl* provides access to a semaphore-based mutual exclusion mechanism. By wrapping these interfaces (for example, by intercepting calls to them, checking the arguments and special-casing certain calls using new code, and passing other calls to the operating system itself), the functionality of the shared memory subsystem can potentially be extended. Our design makes use of such a wrapper.

In particular, if we assume that there will be a *distributed shared memory daemon* process (DSMD) running on each node where our extended memory mapping functionality will be used, we can adopt an approach whereby certain mapped-memory operations are recognized as being operations on the DSM and are handled through cooperation with the DSMD. The recognition that an operation is remote can be supported in either of two ways. One simple option is to introduce a new file-system object called a DSM object and recognizable through a special file type, filename extension (such as .dsm), or some other attribute. The file contents can then be treated as a handle on the DSM object itself by the DSM subsystem. A second option is to extend the "options" field supported by the existing shared memory system calls with extra bits, one of which could indicate that the request refers to a region of the DSM. In a similar manner, we can extend the notion of semaphore "names" (which are normally positive integers in UNIX) to include a DSM semaphore namespace for which operations are recognizable as being distributed synchronization requests.

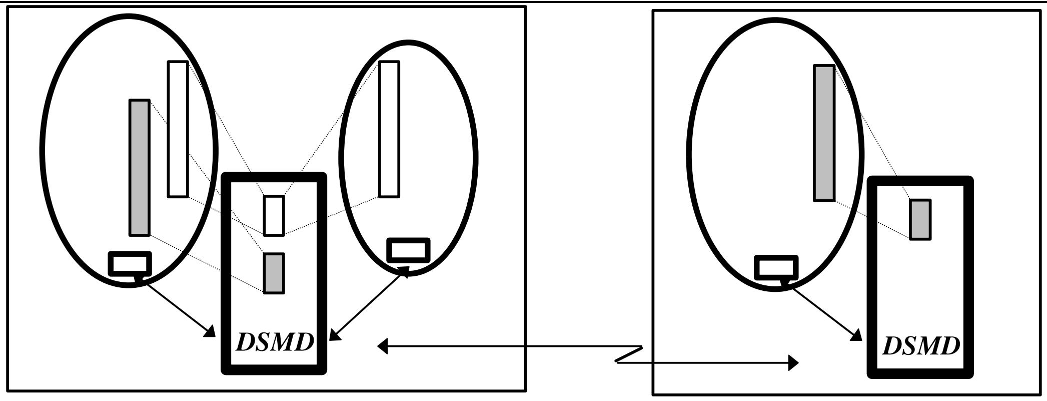


Figure 17-17: Two machines shared memory through the intermediary of a distributed shared memory daemon that runs on each. A wrapper (shown as a small box) intercepts memory mapping and semaphore system calls, redirecting DSM operations to the DSMD. The DMSD processes sharing a given region of memory belong to a process group and cooperate to provide coherent, fault-tolerant behavior. The best implementation of the abstraction depends upon the expected pattern of sharing and the origin of updates.

Having identified a DSM request, that request can then be handled through a protocol with the DSMD process. In particular, we can adopt the rule that all distributed shared memory is implemented as locally shared memory between the application process and the DSMD process, which the DSMD process arranges to maintain in a coherent manner with regard to other processes mapping the same region of memory. The DSMD process thus functions as a type of server that handles requests associated with semaphore operations or events that involve the mapped memory, and that also manages the mapped regions themselves as parts of its own address space. It will be the role of the DSMD servers as a group to cooperate to implement the DSM abstractions in a correct manner; the system call wrappers are thereby kept extremely small and simple, functioning mainly by passing requests through to the DSMD or to the local copy of the operating system, depending on the nature of the system call that was intercepted. This is illustrated in Figure 17-17.

For simplicity of the design, it will be helpful to consider the DSM architecture as being volatile: DSM regions exist only while one or more processes are mapping them, and there is no persistent disk storage associated with them, except perhaps for purposes of paging if the region is too large to maintain in memory. We can view the DSM as a whole as being a collection of objects or *regions*, each having a base address within the DSM, a size, and perhaps access restrictions and security properties. A region might be associated with a file system name, or could be allocated using some form of DSM region manager server; we will not address this issue here.

Notice that our design has reduced the issue to one of maintaining replicated data and performing synchronization with a collection of superimposed process groups (one on behalf of each shared memory region). The DMSD processes that map a given region would also belong to the corresponding process group. The properties of that process group and the algorithms used to maintain the data in it can now be tuned to match well with the patterns of access expected from the application processes using it.

17.7.2 Memory coherency options for distributed shared memory

In any distributed memory architecture, memory coherence is one of the hardest issues to address. Abstractly, the coherence properties of a memory characterize the degree to which that memory is guaranteed to behave like a single non-shared memory that handles every memory access directly. Because our memory is not resident at any single location, but is shared among the processes that happen to be mapping it at a given time, there are a number of options in regard to the degree to which these copies should be coherent. The major choices correspond to the options for shared memory on parallel processors, and consist of the following:

1. *Strong consistency*. In this model, the DSM is expected to behave precisely as a single non-replicated memory might have behaved. In effect, there is a single global serialization order for all read and write operations.

2. *Weak consistency*. In this model, the DSM can be highly inconsistent. Updates propagate after an unspecified and possibly long delay, and copies of the mapped region may differ significantly for this reason.

3. *Release consistency (DASH project).* This model assumes that conflicting read or update accesses to memory are always serialized (protected) using mutual exclusion locks, such as the semaphore system calls that our wrapper intercepts. The model requires that if process *p* obtains a lock associated with a region from process *q,* then *p* will also observe the results of any update that *q* has performed. However, if *p* tries to access the DSM without properly locking the memory, the outcome can be unpredictable.

4. *Causal consistency (Neiger and Hutto).* In this model, the causal relationship between reads and updates is tracked; the memory must provide the property that if access *b* occurs after access *b* in a causal sense, then *b* will observe the results of access *a*.

There are additional models, but these four already represent a sufficient variety of options to present us with some reasonable design choices. To implement strong consistency, it will be necessary to order all update operations, raising the question of how this can be accomplished. The memory protection mechanisms of a virtual memory system offer the needed flexibility: if we imagine the DSMD processes for a given region to have all locked it as read-only, then the memory will refuse updates and will trivially achieve the strong consistency property. Suppose now that each time an update occurs, we intercept the resulting page fault in our wrapper and request that the local DSMD process enable the memory region for write-access. The DSMD process can do this by obtaining a token, perhaps using the *cbcast* based token passing algorithm we developed earlier. It can then unlock the region for local writes and permit the local process to continue. After a suitable period of time, or when it next learns of an update access attempt by some other process, it can relock the local copy of the region, *cbcast* the changed portions, and then pass the token. Notice that although *cbcast* is used to implement this policy, the desired behavior could also have been obtained using *abcast* for all the operations, or even using *fbcast* and sequencing the update and token passing messages at the sender (the latter would have the advantage of requiring just a single field to represent the sequence number). With all of these approaches, the resulting behavior is that of the strongly consistent memory. The strongly consistent memory will also be causally consistency in the case of the implementation that uses *cbcast;* for the alternative implementations this will depend upon the details of the scheme that is used.

The release consistency model can be implemented in a similar manner, except that in this case, the token passing is associated with semaphore operations, and there is no need to communicate changes to a page until the corresponding semaphore is released. Of course, there may be performance reasons that would favor transmitting updates before the semaphore is released, but the release consistency model itself does not require us to do so.

Consider now the degree of match between these design options and the expected patterns of use for a DSM. It is likely that a DSM will either be updated primarily from one source at a time, or in a random way by the processes that use it, simply because this is the pattern seen for other types of distributed applications that maintain replicated data. For the case where there is a primary data source, both the strong and release consistency models will work equally well: the update "lock" will tend to remain at the site where the updates are done, and other copies of the DSM will passively receive incoming updates. If the update source moves around, however, there may be advantages to the release consistency implementation: although the programmer is compelled to include extra code (to lock objects in a way that guarantees determinism), these locks may be obtained more efficiently than in the case of strong consistency, where the implementation we proposed might move the update lock around more frequently than necessary, incurring a high overhead in the process. Further, the release consistency implementation avoids the need to trap page faults in the application, and in this manner avoids a potentially high overhead for updates.

These considerations make release consistency an appealing model for our DSM, despite its dependence on the use of semaphore-style locking. Of course, should an application desire a weak

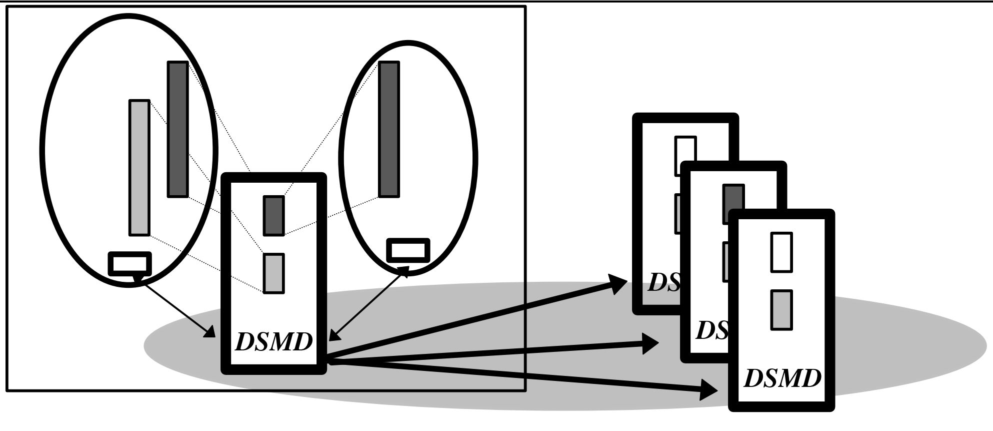


Figure 17-18: The proposed solution maps the DSM problem to a more familiar one: replicated data with locking within a virtually synchronous process group. Only one of several overlapped groups is shown; another group would be used for the dark gray memory region, yet another for the white one, etc. Virtual synchrony provides us with simple solutions for what would otherwise be tricky problems, like ensuring the coherence of the distributed memory, handling failures and dynamic join events, and dealing with protection. Actually implementing such an architecture using the Horus system over an ATM network would be an interesting research project and would raise interesting performance challenges, but the basic problem is clearly very closely matched to the model for which virtual synchrony and Horus were developed.

consistency model or need strong consistency, we now know how both models can be implemented.

However, there are also issues that the consistency model overlooks, and yet that could be quite important in a practical DSM. Many applications that operate on shared memory will be sensitive to the latency with which updates are propagated, and there will be a subset in which other communication patterns and properties are needed: for example, video algorithms will want to send a full frame at a time, and will need guarantees of throughput and latency from the underlying communications architecture. Accordingly, our design should include one additional interface by which a knowledgeable application can specify the desired update properties to the DSM. This *dsmctl* system call would be used to specify both the pattern of updates that the application will generate (random, page based, isochronous) and also the maximum latency and other special requirements for acceptable performance. The DSMD can then use this information to schedule its communication appropriately. If available, the *page dirty* bit provided by the virtual memory hardware can be checked periodically by the DSMD; if not, shared regions that are mapped for update can be transmitted in their entirety at the frequency requested by the user.

17.7.3 False sharing

False sharing is a phenomenon seen on parallel shared memory machines that corresponds to thrashing in a virtual memory architecture. The problem arises when multiple logically unrelated objects are mapped to the same shared memory region or page by an accident of storage allocation. When these objects are updated in parallel, the memory subsystem is unable to detect that the updates are independent ones and treats the situation as one in which the processes doing the updates are contending for the same object. In our implementation of strong consistency, the update token would bounce around in this case, resulting in a huge overhead for token passing and page fault handing on the client systems. Yet the problem also points to an issue in our proposed release consistency scheme, namely the *granularity of locking*. In particular, it becomes clear that the semaphores used for locking must have the same granularity as the objects the DSMD transmits for updates, most likely a page. Otherwise, because the DSMD lacks a fine-grained notion of data access, when an object is updated on a page and the semaphore locking that object is released, the entire page will be transmitted to other processes mapping the page, potentially overwriting parts of the page that the semaphore was not considered to “lock” and which are in fact not even up to date on the node that held the lock.

Our DSM architecture can only work if the granularity of locking is at the page level or region level, and in either case, false sharing could now arise as a visible problem for the developer. Rather than trying to overcome this problem, it may be best to simply caution the user: the DSM architecture we have proposed here will perform poorly if an application is subject to false sharing, hence such applications may need to be redesigned to arrange for concurrently updated but logically unrelated objects to reside in different regions or at least on different pages, and in the case of release consistency, must be locked by separate semaphores.

17.7.4 Demand paging and intelligent prefetching

We cited the case of frequent and time-critical updates, but there is another style of DSM use which will require more or less the opposite treatment. Suppose that the DSM region is extremely large and most applications access it in a sparse manner. Then even if a region is “mapped” by some process, it may not be necessary or even desirable to actively update that region each time some process updates some part of the data area. In such cases, a demand-paging model makes more sense, whereby a portion of the DSM is maintained as current only if the process holding that region is actually accessing it.

Although we will not tackle the problem here, for reasons of brevity, it would be desirable for such large regions to be managed as multiple subregions, shrinking the process group for a given subregion to include only those processes that are actively updating it or reading it. With such an approach, one arrives at a form of *demand paging* in which a process, upon attempting to access a subregion that is not currently mapped into its address space, experiences a page fault. To resolve the fault the DSMD would join the process group for that subregion, transferring the current state of the subregion (or just those updates that have occurred since the process was last a memory), and then enabling read or update access to the subregion and resuming local computation.

Notice that the virtual synchrony properties of the state transfer make it easy to describe a solution to what would otherwise be a tricky synchronization problem! Lacking the virtual synchrony model, it would not be at all simple to coordinate the addition of a new memory to a subregion group and to integrate the state transfer operation with updates that may be occurring dynamically. The model makes it easy to do so and to still be able to guarantee that release consistency or strong consistency will be observed by the DSM user. On the other hand, recall that virtual synchrony comes with no guarantees of real-time performance (a topic to which we will return in Chapter 20), and hence support for dynamically adjusting the members of a process group that maps a given region or subregion may be incompatible with providing real-time performance and latency guarantees. For situations in which such guarantees are desired, it may be wise to disable this form of dynamicism unless the requirements are fairly weak ones.

Demand paging systems perform best if the relatively costly operations involved in fetching a page are performed shortly before the page fault actually takes place, so as to overlap useful computation with the paging-in activity, and minimizing the delay associated with actually servicing the page fault when it occurs. Accordingly, it would be advisable to implement some form of prefetching policy whereby the DSMD, recognizing a pattern of access (such as sequential access to a series of subregions) would assume that this pattern will continue into the future and join subregion groups in anticipation of the future need. Our architecture creates a convenient context within which to implement such a policy.

17.7.5 Fault-tolerance issues

Our DSM will have a natural form of fault-tolerance that arises directly from the fault-tolerance of the virtual synchrony model used by the DSMD processes to form process groups and propagate updates. The issues that arise are primarily ones associated with the possibility of a failure by a process while it is doing an update. Such an event might leave the DSM corrupted and a semaphore in the locked state (the token for the group would be at the process that failed).

A good way to solve this problem would be to introduce a new kind of page-fault exception into the DSM model, which could be called a *page corruption* exception. In such an approach, when a process holding an update lock or semaphore for a page or region fails, any subsequent access by some other process mapping that region would result in a corruption trap. The handler for such a trap would be granted the update lock or semaphore and would be required to restore the page to a consistent state. The next update would be understood to clear the corruption bit, so that processes that don't attempt to access the page during the period of corruption might be completely y unaware that a problem had occurred.

17.7.6 Security and protection considerations

The reliability of a DSM should extend beyond issues of fault-tolerance and detecting potential corruption to also include guarantees of protection and security or privacy if desired. We have not yet treated security issues in this text, and hence defer discussion of the options until Chapter 19. In brief, one could arrange for the data on the wire to be encrypted so that eavesdroppers lacking an appropriate key would be unable to map a protected segment and unable to make sense of any intercepted updates. Depending on the degree to which the system implementing virtual synchrony is trusted, weaker security options might include some form of user-id based access control in which unauthorized users are prevented from joining the group. Because the DSMD must join a process group to gain access to a DSM segment, the group join operation can include authorization keys for use in determining whether or not access should be granted. Alternatively, if the DSMD process itself can be trusted, it can perform a mapping from local user id's on the host machine where it is running to global user id's in a protection domain associated with the DSM, permitting access under UNIX-style restrictions.

17.7.7 Summary and discussion

The previous examples in this chapter illustrated some of the challenges that can be encountered when attempting to exploit group structures in implementing a distributed system. In contrast, the DSM example shows how simple and elegant solutions can be (and how easy it can be to understand them) when the match of problem and tool turns out to be close. Our architecture would, in principle, be a highly efficient one: the costs and overheads are predominately those of the virtual synchrony communication architecture, and in the case of the release consistency model, no additional overhead beyond this is imposed. As we will see below, the model can perform extremely well over an appropriate software implementation architecture and with high speed hardware. The "video mapped" shared memory suggested at the start of this section is not an unreasonable prospect. Moreover, by wrapping the standard shared memory mechanisms for a setting, the DSM abstraction can be made extremely transparent.

Of course, the feasibility of this architecture depends upon having a suitable shared memory subsystem available for use between the DSMD and its clients; our solutions have in the end required that we be able to manipulate memory protection bits from the DSMD, trap page faults by the client processes and restart them after servicing these, and sense the state of a page (dirty or clean) to avoid undesired excess communication. Some operating systems, such as Mach or the commercial OSF/1 system, provide interfaces by which this would be possible; others do not, or offer only part of the support that might be needed. However, the problem is clearly an approachable one and indeed is an appealing research topic; perhaps the next edition of this textbook will report on one or more systems that really use this approach, and will be able to compare the results with one or more systems offering equivalent functionality using other methods.

17.8 Related Readings

On wrappers and technologies that can support them: [Jon93, RAAB88, RAAH88, WLAG93]. On the Isis Toolkit: [BR94, BJ87a]. (Information on the most current API's should be obtained directly from the company that markets the Isis product line; their web page is http://www.isis.com). On agents: [GM95, Ous94, JvRS95]. Virtual fault-tolerance: [BS95]. On shared memory: [LH89, GLLG90, ABHN91, FZ91, Car93, FMPK95, JKW95]. Tanenbaum also discusses shared memory in [Tan88], and Coulouris treats the topic in [CDK94].

18. Reliable Distributed Computing Systems

The purpose of this chapter is to shift our attention away from protocol issues to architectural considerations associated with the implementation of process group computing solutions. Although there has been a great deal of work in this area, we focus on the Horus system[16], because that system is well matched to the presentation of this textbook and, having been developed by the author and his colleagues, is well known to the author. Horus is available for researchers in academic or industrial settings (at no fee), and may be used in conjunction with this textbook as a platform on which to base experiments and to gain some hands-on experience with reliable distributed computing.

18.1 Architectural Considerations in Reliable Systems

It may strike the reader that Part II of this text and the first chapters of Part III have lost one of the important themes of the first part of the book, namely the growing importance of architectural structure and modularity in reliable distributed systems, and indeed in structuring distributed systems of all types. Our goal in this chapter, in part, is to reestablish some of these principles in the context of the group computing constructs introduced in Chapters 13-17. Specifically, we will explore the embedding of group communication support into a modular systems architecture.

Historically, group computing and data replication tools have tended to overlook the importance of architectural structure. These technologies have traditionally been presented in what might be called a "flat" architecture: one in which the API's provided by the system are fixed, correspond closely to the group construct and associated communication primitives, and accessible more or less uniformly from any application that makes use of the group communication environment, anywhere in the system.

In practice, however, the use of group communication will vary considerably depending upon what one is attempting to do. Consider the examples that arose in Chapter 17 when we discussed group computing in the context of enterprise Web applications:

- Groups used to replicate a Web server for load-balancing, fault-tolerance, or scalable performance through parallelism.
- Groups used to interconnect a set of Web servers so as to create the illusion of a single, corporate-wide server within which objects might migrate or be replicated to varying degrees depending upon usage patterns.
- Groups corresponding to the set of Web proxy servers that cache a given data item, and used to invalidate those cached copies or to refresh them when they change.
- Groups used to distribute Java applets to user's cooperating in conferencing applications or other groupware applications (we gave a number of examples in Chapter 17 and won't repeat them here).

[16] The ancient Egyptian religion teaches that after the world was created, the Gods Osiris and Seth engaged in an epic battle for control of the earth. Osiris, who is associated with good, was defeated and hacked to pieces by the evil Seth, and his body scattered over the Nile Delta. The Goddess Isis, gathered the fragments and magically restored Osiris to life. He descended to rule the Underworld, and with Isis fathered a child, Horus, who went on to defeat Seth. When we developed the Isis Toolkit, the image of a system that puts the pieces together after a failure appealed to us, and we named the system accordingly, although the failures that the toolkit can handle are a little less extreme than the one that Osiris experienced!. Later, when we developed Horus, it seemed appropriate to again make allusion to the Egyptian myth. However, it may take some time to determine whether the Horus system will go on to banish unreliability and inconsistency from the Information Superhighway).

- Groups used to distribute updates to documents, or other forms of updates, to Java applets running close to the client browsers.
- Groups formed among the set of Java applets running on behalf of clients, for the purpose of multicasting updates or other changes to the state of the group session among the participants.
- Groups associated with security keys employed in a virtual private network.

Clearly, these uses correspond to applications that would be implemented at very different levels of programming abstraction, and for which the most appropriate presentation of the group technology would vary dramatically. Several of these represent potential uses of wrappers, but others would match better with toolkit interfaces and still others with special-purpose high level programming languages. Even within those subclasses, one would expect considerable variation in terms of what is wrapped, the context in which those tools or languages are provided, and the nature of the tools themselves. No single solution could possibly satisfy all of these potential types of developers and uses. On the contrary, any system that offers just a single interface to all of its users is likely to confuse its users and to be perceived as complex and difficult to learn. Returning to our historical observation, the tendency to offer group communication tools through a flat interface (one that looks the same to all applications and that offers identical capabilities no matter where it is used in the system) has proved to be an obstacle to the adoption of these technologies, because the resulting tools tend to be “conceptually mismatched” with the developer’s goals and mindset.

Indeed, the lesson goes further than this. Although we have presented group communication as an obvious and elegant step, the experience of programming with groups can be quite a bit more challenging than the developer might expect. Obtaining good performance is not always an easy thing, and the challenge of doing so rises steeply if groups are deployed in an unstructured way, creating complex patterns of overlap within which the loads placed on individual group members may vary widely from process to process. Thus what may seem obvious and elegant to the reader, can start to seem clumsy and complex to the developer who is struggling to obtain predictable performance and graceful scalability.

These observations argue for a more structured presentation of group computing technologies: one in which the tools and API’s provided are aimed at a specific class of users, and will guide those users to a harmonious and simple solution to the problems anticipated for that class of users. If the same technology will also support some other community of users, a second set of tools and API’s should be offered to them. Thus, the tools provided for Web server replication might look very different from those available to the developer of a Java display applet, even if both the applet and the Web server turn out to offer functionality that arises out of a group communication subsystem. The author believes that far too little attention has been given to this issue up to the present, and that this has emerged as a significant obstacle to the widespread use of reliability technologies.

Functionality of a "client-level" API:

- Fault-tolerant remote procedure call
- Reliable, unbreakable streams to servers
- Consistent or reliable subscriptions to data published by servers
- Tools for forming groupware sessions involving other client systems

Functionality of a "WAN" server API:

- Tools for consistently replicating data within wide-area or corporate networks
- Technology for updating global state and for merging after a partitioning failure is corrected
- Security tools for creating virtual private networks
- Management tools for control and supervision

Functionality of a cluster server API:

- Tools for building fault-tolerant servers (ideally, as transparently as is feasible)
- Load-balancing and scalable parallelism support
- Management tools for system servicing and automatic reconfiguration
- Facilities for online upgrade

Other cases that may require specialized API's:

- Multimedia data transport protocols (special quality-of-service or real-time properties)
- Security (key management and authentication API's)
- Debugging and instrumentation
- Very large scale data diffusion

Figure 18-1: Different levels of a system may require different styles of group computing support. A simple client-server architecture gives rise to three levels of API, shown above. Further structure might be introduced in a multimedia setting (where special protocols may be needed for video data movement or to provide time-synchronous functionality), in a transactional database setting (where client's may expect an SQL-oriented interface), and so forth, or in a security setting (where API's will focus on authentication and key management).

At a minimum, focusing only on issues associated with replication (as opposed to security, system management, or real-time), it would appear that three "layers" of API's are needed (Figure 18-19). The lowest such layer is the one aimed at uses within servers; the middle layer focuses on interconnection and management of servers within a WAN setting, and the third layer on client-side issues and interfaces. Such layers may be further subdivided: perhaps the client layer offers a collection of transactional database tools and a collection of Java groupware interfaces, while the server layer offers tools for multimedia data transmission, tools for consistent replication and coordinated control, and tools for fault-tolerance through active replication. This view of the matter now places unusual demands upon the underlying communication system: not only must it potentially "look" different for different classes of users, but it may also need to offer very different properties for different classes of users. Security and management subsystems would introduce additional API's, which may well be further structured. And real-time subsystems are likely to require still further structure and interfaces.

18.2 Horus: A Flexible Group Communications System

The observations of the preceeding section may seem to yield an ambigious situation. On the one hand, we have seen in previous chapters that process-group environments for distributed computing represent a promising step towards robustness for mission-critical distributed applications. Process groups have a "natural" correspondence with data or services that have been replicated for availability, or as part of a coherent cache such as might be used to ensure the consistency of documents managed by a set of Web proxies. They can been used to support highly available security domains. And, group mechanisms fit well with an emerging generation of intelligent network and collaborative work applications.

Yet we have also seen that there are many options concerning how process groups should look and behave. The requirements that applications place on a group infrastructure can vary tremendously, and there may be fundamental htradeoffs between semantics and performance. Even the most appropriate way to present the group abstraction to the application depends on the setting.

The Horus system responds to this observation by providing an unusually flexible group communication model to application-developers. This flexibility extends to system interfaces, the properties provided by a protocol stack, and even the configuration of Horus itself, which can run in user space, in an operating system kernel or microkernel, or be split between them. Horus can be used through any of several application interfaces. These include toolkit-styled interfaces, but also wrappers that hide group functionality behind Unix communication system-calls, the Tcl/Tk programming language, and other distributed computing constructs. The intent is that it be possible to slide Horus beneath an existing system as transparently as possible, for example to introduce fault-olerance or security without requiring substantial changes to the system being hardened [BS95].

A basic goal of Horus is to provide efficient support for the virtually synchronous execution model. However, although often desirable, properties like virtual synchrony may sometimes be unwanted, introduce unnecessary overheads, or conflict with other objectives such as real-time guarantees. Moreover, the optimal implementation of a desired group communication property sometimes depends on the runtime environment. In an insecure environment, one might accept the overhead of data encryption, but wish to avoid this cost when running inside a firewall. On a platform like the IBM SP2, which has reliable message transmission, protocols for message retransmission would be superfluous.

Accordingly, Horus provides an architecture whereby the protocol supporting a group can be varied, at runtime, to match the specific requirements of its application and environment. Virtual synchrony is only one of the options available, and even when it is selected, the specific ordering properties that messages will respect, the flow control policies used, and other details can be fine-tuned. Horus obtains this flexibility using a structured framework for protocol composition, which incorporates ideas from systems such as the UNIX streams framework and the x-Kernel, but replaces point-to-point communication with group communication as the fundamental abstraction. In Horus, group communication support is provided by stacking protocol modules that have a regular architecture, and in which each module has a separate responsibility. A process group can be optimized by dynamically including or excluding particular modules from its protocol stack.

18.2.1 A layered process group architecture

It is useful to think of Horus' central protocol abstraction as resembling a Lego block; the Horus "system" is thus like a box of Lego blocks. Each type of block implements a microprotocol that provides a different communication feature. To promote the combination of these blocks into macroprotocols with desired properties, the blocks have standardized top and bottom interfaces that allows them to be stacked on top of each other at run time in a variety of ways (see Figure 18-2). Obviously, not every sort of protocol block makes sense above or below every other sort. But the conceptual value of the architecture is that where it makes sense to create a new protocol by restacking existing blocks in a new way, doing so is straightforward.

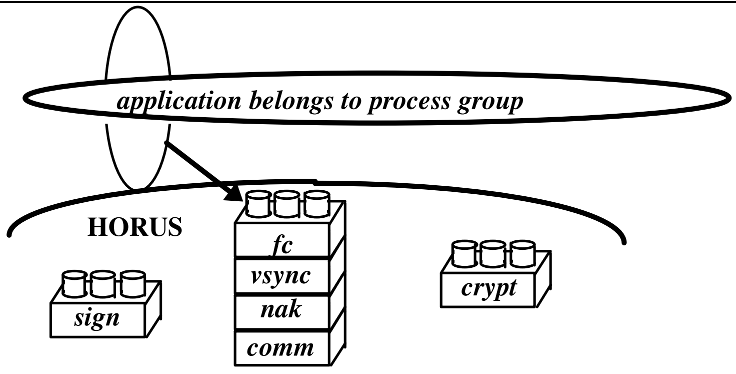


Figure 18-2: Group protocol layers can be stacked at run-time like Lego blocks, and support applications through one of several application programmer interfaces. Shown is an application program that belongs to a single process group and is supported by a Horus protocol stack of four layers: "fc", the flow-control layer, "vsync", a layer that implements virtually-synchronous process group views, "nak", a layer that uses negative acknowledgements to overcome communication failures, and "comm", which interfaces Horus to a network. The application would often use Horus through a wrapper that might conceal this group functionality, but can also do so using a toolkit. The layers illustrated here are fanciful ones; some real layers are shown in Figure 18-3. Horus supports many layers but not all need be used in any particular stack: shown here are two security layers (one for signing messages and one for encrypting their contents), which were not used for this particular application.

COM	**The COM layer provides the Horus group interface to such low-level protocols as IP, UDP, and some ATM interfaces.**
NAK	**This layer implements a negative acknowledgement based message retransmission protocol.**
CYCLE	**Multi-media message dissemination using Smith's "cyclic UDP" protocol**
PARCLD	**Hierarchical message dissemination (parent-child layer)**
FRAG	**Fragmentation and reassembly of large messages**
MBRSHIP	**This layer provides each member with a list of endpoints that are believed to be accessible. It runs a group membership consensus protocol to provide its users with a virtually synchronous execution model.**
FC	**Flow control layer**
TOTAL	**Totally ordered message delivery**
STABLE	**This layer detects when a message has been delivered to all destination endpoints, and can consequently be garbage collected.**
CRYPT	**Encryption and decryption of message body**
MERGE	**Location and merging of multiple group instances**

Figure 18-3: ***Horus provides a large collection of microprotocols. Some of the most important ones are shown above, but there are more than 60 specialized layers that have been developed for special purposes, such as real-time communication. This illustrates the sense in which Horus is both an architecture that can be used to support special-purpose protocols as well as an implementation of virtual synchrony, if this is what the application desires.***

Technically, each Horus protocol block is a software module with a set of entry points for downcall and upcall procedures. For example, there is a downcall to send a message, and an upcall to receive a message. Each layer is identified by an ASCII name, and registers its upcall and downcall-handlers at initialization time. There is a strong similarity between Horus protocol blocks and object classes in an object-oriented inheritance scheme, and readers may wish to think of protocol blocks as members of a class hierarchy.

To see how this works, consider the Horus *message_send* operation. It looks up the message send entry in the topmost block, and invokes that function. This function may add a header to the message, and will then typically invoke *message_send* again. This time, control passes to the message send function in the layer below it. This repeats itself recursively until the bottommost block is reached and invokes a driver to actually send the message.

The specific layers currently supported by Horus solve such problems as interfacing the system to varied communication transport mechanisms, overcoming lost packets, encryption and decryption, maintaining group membership, helping a process that joins a group obtain the state of the group, merging a group that has partitioned, flow control, etc. (see sidebar). Horus also includes tools to assist in the development and debugging of new layers.

Each stack of blocks is carefully shielded from other stacks. It has its own prioritized threads, and has controlled access to available memory through a mechanism called *memory channels*. Horus has a memory scheduler that dynamically assigns the rate at which each stack can allocate memory, depending on availability and priority, so that no stack can monopolize the available memory. This is particularly important inside a kernel, or if one of the stacks has soft real-time requirements.

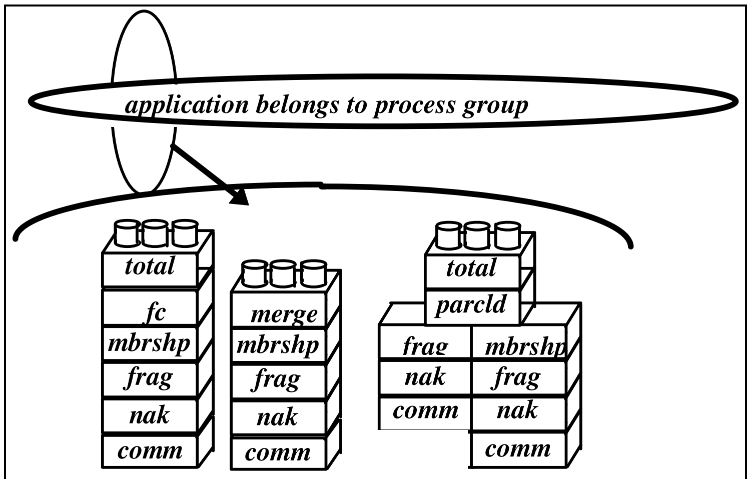


Figure 18-4: The Horus stacks are shielded from each other, and have their own threads and memory, each of which is provided through a scheduler. Each stack can be thought of as a small program that executes inside Horus. Although this feature is not shown above, a stack can be split between the user's address space and the kernel, permitting the user to add customized features to a stack while benefitting from the performance of a kernel-based protocol implementation.

Besides threads and memory channels, each stack deals with three other types of objects: endpoints, groups, and messages. The endpoint object models the communicating entity. Depending on the application, it may correspond to a machine, a process, a thread, a socket, a port, and so forth. An endpoint has an address, and can send and receive messages. However, as we will see later, messages are not addressed to endpoints, but to groups. The endpoint address is used for membership purposes.

A *group object* is used to maintain the local protocol state on an endpoint. Associated with each group object is the *group address* to which messages are sent, and a *view*: a list of destination endpoint addresses that are believed to be accessible group members. Since a group object is purely local, Horus technically allows different endpoints to have different views of the same group. An endpoint may have multiple group objects, allowing it to communicate with different groups and views. A user can install new views when processes crash or recover, and can use one of several membership protocols to reach some form of agreement on views between multiple group objects in the same group.

The message object is a local storage structure. Its interface includes operations to push and pop protocol headers. Messages are passed from layer to layer by passing a pointer, and never need be copied.

A thread at the bottommost layer waits for messages arriving on the network interface. When a message arrives, the bottommost layer (typically COM) pops off its header, and passes the message on to the layer above it. This repeats itself recursively. If necessary, a layer may drop a message, or buffer it for delayed delivery. When multiple messages arrive simultaneously, it may be important to enforce an order on the delivery of the messages. However, since each message is delivered using its own thread, this ordering may be lost depending on the scheduling policies used by the thread scheduler. Therefore, Horus numbers the messages, and uses *event count* synchronization variables[RK79] to reconstruct the order where necessary.

18.3 Protocol stacks

The microprotocol architecture of Horus would not be of great value unless the various classes of process group protocols that we might wish to support can be simplified by being expressed as stacks of layers, perform well, and share significant functionality. The experience with Horus in this regard has been very positive.

For example, the stacks shown in Figure 18-4 all implement virtually synchronous process groups. The left-most stack provides totally ordered, flow-controlled communication over the group membership abstraction. The layers FRAG, NAK and COM respectively break large messages into smaller ones, overcome packet loss using negative acknowledgements, and interface Horus to the underlying transport protocols. The adjacent stack is similar, but provides weaker ordering and includes a layer that supports "state transfer" to a process joining a group, or when groups merge after a network partition. To the right is a stack that supports scaling through a hierarchical structure, in which each "parent" process is responsible for a set of "child" processes. The dual stack illustrated in this case represents a feature whereby a message can be routed down one of several stacks, depending on the type of processing required. Additional protocol blocks provide functionality such as data encryption, packing small messages for efficient communication, isochronous communication (useful in multimedia systems), etc.

For Horus layers to fit like Lego blocks, they each must provide the same downcall and upcall interfaces. A lesson learned from thex-Kernel is that if the interface is not rich enough, extensive use will be made of general purpose control operations (similar to *ioctl*), which reduces configuration flexibility. (Since the control operations are unique to a layer, the Lego blocks would not "fit" as easily.) The *Horus Common Protocol Interface* (HCPI) therefore supports an extensive interface that supports all common operations in group communication systems, going beyond the functionality of earlier layered systems such as the x-Kernel, Furthermore, the HCPI is designed for multiprocessing, and is completely asynchronous and reentrant.

Broadly, the HCPI interfaces fall into two categories. Those in the first group are concerned with sending and receiving messages, and the stability of messages[17]. The second category of Horus operations are concerned with membership. In the down direction, they let an application or layer control the group membership used by layers below it. As upcalls, these report membership changes, communication problems, and other related events to the application.

While supporting the same HCPI, each Horus layer runs a different protocol, each implementing a different property. Although Horus allows layers to be stacked in any order (and even multiple times), most layers require certain semantics from layers below it, imposing a partial order on the stacking. These constraints have been tabulated. Given information about the properties of the network transport service, and the properties provided by the application, it is often possible to automatically generate the minimal protocol stack that achieves a desired property.

Layered protocol architectures sometimes perform poorly. Traditional layered systems impose an order on which protocols process messages, limiting opportunities for optimization, and imposing excessive overhead. Clark and Tennenhouse have suggested that the key to good performance rests in *Integrated Layer Processing* (ILP) [CT87, AP93, BD95, KP93, KC94]. Systems based on the ILP principle avoid inter-layer ordering constraints, and can perform as well as monolithically structured systems. Horus is consistent with ILP: there are no intrinsic ordering constraints on processing, so unnecessary synchronization delays are avoided. Moreover, as we will see below, Horus supports an optional protocol accelerator which greatly improves the performance of those layered protocols that make use of it.

18.4 Using Horus to Build a Robust Groupware Application

Earlier, we commented that Horus can be hidden behind standard application programmer interfaces. A good illustration of how this is done arose when we interfaced the Tcl/TK graphical programming

[17] It is common to say that a message is *stable* when processing has completed and associated information can be garbage collected. Horus standardizes the handling of stability information, but leaves the actual semantics of stability to the user. Thus, an application for which stability means "logged to disk" can share this Horus functionality with an application for which stability means "displayed on the screen."

language to Horus. A challenge posed by running systems like Horus side by side with a package like X-windows or Tcl/TK is that such packages are rarely designed with threads or Horus communication stacks in mind. To avoid a complex integration task, we therefore chose to run Tcl/TK as a separate thread in an address space shared with Horus. Horus intercepts certain system calls issued by Tcl/TK (see Figure 3), such as the Unix *open* and *socket* system calls. We call this the resulting mechanism an *intercept proxy;* it is a special type of wrapper oriented towards intercepting this type of system call. The proxy redirects he system calls, invoking Horus functions which will create Horus process groups and register appropriate protocol stacks at run time. Subsequent I/O operations on these group I/O sockets are mapped to Horus communication functions.

To make Horus accessible within Tcl applications, two new functions were registered with the Tcl interpreter. One creates endpoint objects, and the other creates group addresses. The endpoint object itself can create a group object using a group address. Group objects are used to send and receive messages. Received messages result in calls to Tcl code, which typically interpret the message as a Tcl command. This yields a powerful framework: a distributed, fault-tolerant, whiteboard application can be built using only eight short lines of Tcl code, over a Horus stack of seven protocols.

To validate our approach, we ported a sophisticated Tcl/TK application to Horus. The Continuous Media Toolkit (CMT) [RS92] is a Tcl/TK extension that provides objects that read or output audio and video data. These objects can be linked together in pipelines, and are synchronized by a *logical timestamp* object. This object may be set to run slower or faster than the real clock, or even backwards. This allows stop, slow motion, fast forward, and rewind functions to be implemented.

Architecturally, CMT consists of a multi-media server process that multicasts video and audio to a set of clients. We decided to replicate the server using a primary-backup approach, where the backup servers stand by to back up failed or slow primaries.

The original CMT implementation depends on extensions to Tcl/TK. These implement a master/slave relationship between the machines, provide for a form of logical timestamp synchronization between them, and support a real-time communication protocol called Cyclic UDP. The Cyclic UDP implementation consists of two halves, a sink object that accepts multi-media data from another CMT object, and a source object that produces multi-media data and passes it on to another CMT object (see *Figure 18-5a*). The resulting system is distributed but intolerant of failures, and does not allow for multicast.

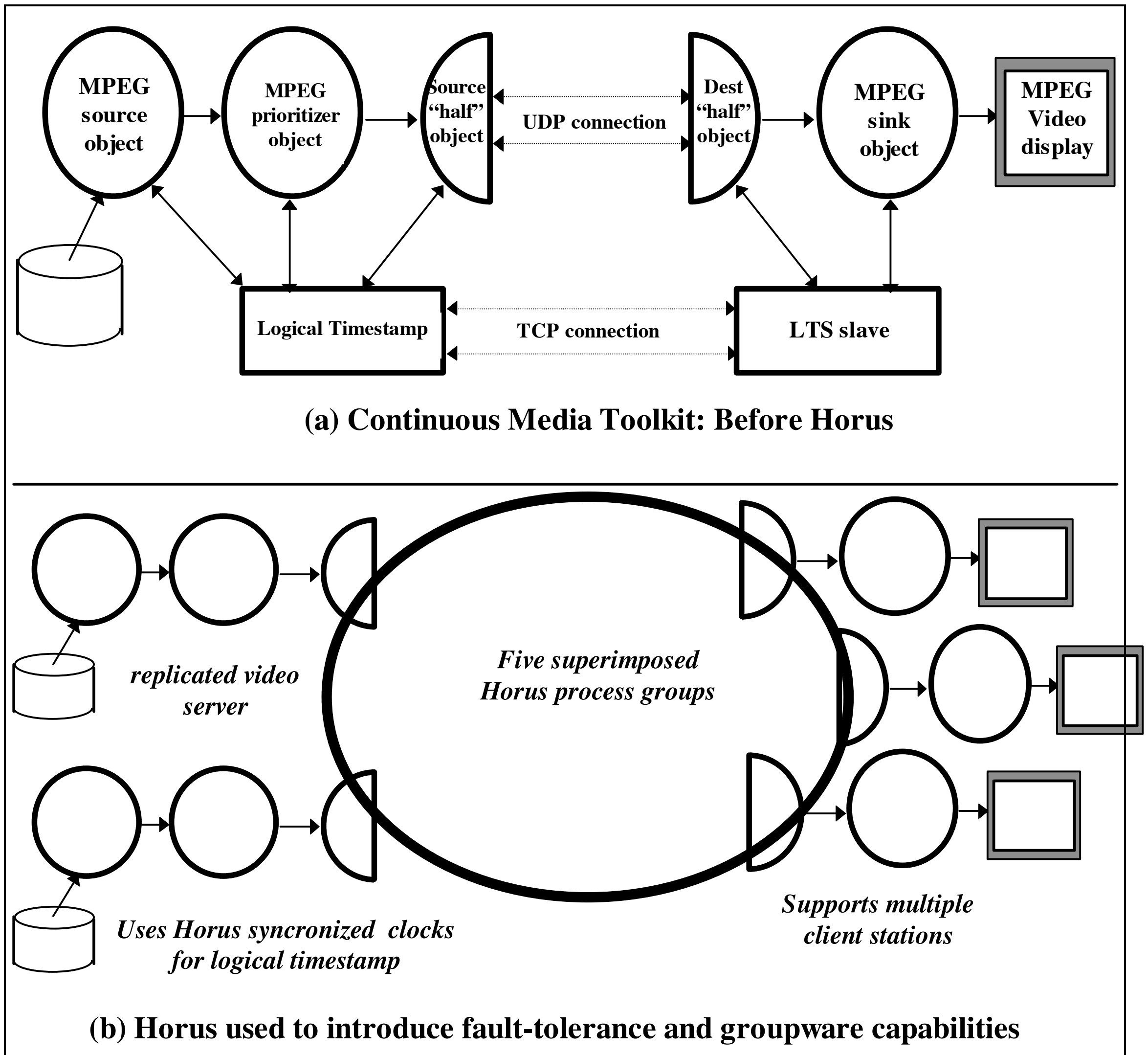

MPEG source object
MPEG prioritizer object
Source "half" object
UDP connection
Dest "half" object
MPEG sink object
MPEG Video display
Logical Timestamp
TCP connection
LTS slave
(a) Continuous Media Toolkit: Before Horus
replicated video server
Five superimposed Horus process groups
Uses Horus syncronized clocks for logical timestamp
Supports multiple client stations
(b) Horus used to introduce fault-tolerance and groupware capabilities

Figure 18-5: These figures show an example of a video service implemented using the Continuous Media Toolkit. MPEG is a video compression standard. In (a), a standard, fault intolerant set-up is depicted. In (b), Horus was used to implement a fault-tolerant version that is also able to multicast to a set of clients.

Using Horus, it was straightforward to extend CMT with fault-tolerance and multicast capabilities. Five Horus stacks were required. One of these is hidden from the application, and implements a clock synchronization protocol [Cri89]. It uses a Horus layer called MERGE to ensure that the different machines will find each other automatically (even after network partitions), and employs the virtual synchrony property to rank the processes, assigning the lowest ranked machine to maintain a master clock on behalf of the others. The second stack synchronizes the speeds and offsets with respect to real-time of the logical timestamp objects. To keep these values consistent, it is necessary that they be updated in the same order. Therefore, this stack is similar to the previous one, but includes a Horus protocol block that places a total order on multicast messages delivered within the group. [18] The third tracks the list of servers and clients. Using a deterministic rule based on the process ranking maintained by the virtual synchrony layer, one server decides to multicast the video, and one server, usually the same, decides to multicast the audio. This set-up is shown in Figure 18-5b.

To disseminate the multi-media data, we used two identical stacks, one for audio and one for video. The key component in these is a protocol block that implements a multi-media generalization of the Cyclic UDP protocol. The algorithm is similar to FRAG, but will reassemble messages that arrive out of order, and drop messages with missing.

One might expect that a huge amount of recoding would have been required to accomplish these changes. However, all of the necessary work was completed using 42 lines of Tcl code. An additional 160 lines of C code supports the CMT frame buffers in Horus. Two new Horus layers were needed, but were developed by adapting existing layers; they consist of 1800 lines of C code and 300 lines, respectively (ignoring the comments and lines common to all layers). Thus, with relatively little effort and little code, a complex application written with no expectation that process group computing might later be valuable was modified to exploit Horus functionality.

18.5 Using Horus to Harden CORBA applications

The introduction of process groups into CMT required sophistication with Horus and its intercept proxies. Many potential users would lack the sophistication and knowledge of Horus required to do this, hence we recognized a need for a way to introduce Horus functionality in a more transparent way. This goal evokes an image of “plug and play” robustness, and leads one to think in terms of an object-oriented approach to group computing.

Early in this text we looked at CORBA, noting that object-oriented distributed applications that comply with the CORBA ORB specification and support the IOP protocol can invoke one-another's methods with relative ease. Our work resulted in a CORBA compliant interface to Horus, which we call Electra [Maf95]. Electra can be used without Horus, and vice versa, but the combination represents a more complete system.

In Electra, applications are provided with ways to build Horus process groups, and to directly exploit the virtual synchrony model. Moreover, Electra objects can be aggregated to form “object groups,” and object references can be bound to both singleton objects and object groups. An implication of the interoperability of CORBA implementations is that Electra object groups can be invoked from *any* CORBA-compliant distributed application, regardless of the CORBA platform on which it is running, without special provisions for group communication. This means that a service can be made fault-tolerant without changing its clients.

[18] This protocol differs from the *Total* protocol in the Trans/Total[MMABL96] project in that the Horus protocol only rotates the token among the current set of senders, while the Trans/Total protocol rotates the token among all members.

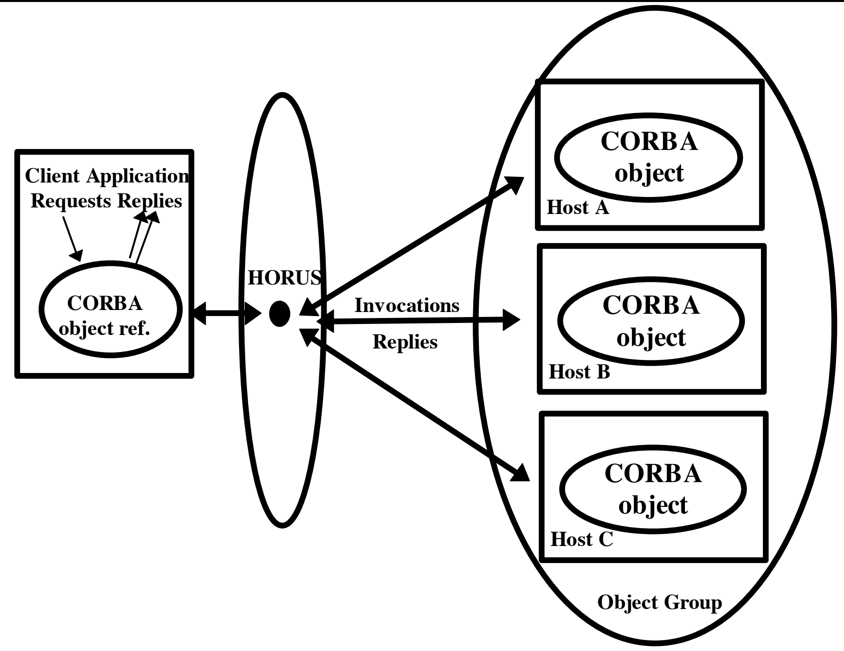


Figure 18-6: Object-group communication in Electra, a CORBA-compliant ORB that uses Horus to implement group multicast. The invocation method can be changed depending on the intended use. Orbix+Isis and the COOL-ORB are examples of commercial products that support object groups..

When a method invocation occurs within Electra, object-group references are detected and transformed into multicasts to the member objects (see Figure 18-6). Requests can be issued either in transparent mode, where only the first arriving member reply is returned to the client application, or in non-transparent mode, permitting the client to access the full set of responses from individual group members. The transparent mode is used by clients to communicate with replicated CORBA objects, while non-transparent mode is employed with object groups whose members perform different tasks. Clients submit a request either in a synchronous, asynchronous, or deferred-synchronous way.

The integration of Horus into Electra shows that group programming can be provided in a natural, transparent way with popular programming methodologies. The resulting technology permits the use to "plug in" group communication tools anywhere that a CORBA application has a suitable interface. To the degree that process-group computing interfaces and abstractions represent an impediment to their use in commercial software, technologies such as Electra suggest a possible middle ground, in which fault-tolerance, security, and other group-based mechanisms can be introduced late in the design cycle of a sophisticated distributed application.

18.6 Basic Performance of Horus

A major concern of the Horus architecture is the overhead of layering, hence we now focus on this issue. This section present the overall per formance of Horus on a system of SUN Sparc10 workstations running SunOS 4.1.3, communicating through a loaded Ethernet. We used two network transport protocols: normal UDP, and UDP with the Deering IP multicast extensions [Dee88] (shown as "Deering").

To highlight some of the performance numbers: Horus achieves a one-way latency of 1.2 msecs over an unordered virtual synchrony stack (over ATM, it is currently 0.7 msecs), and, using a totally ordered layer over the same stack, 7,500 1-byte messages per second. Given an application that can accept lists of messages in a single receive operation, we can drive up the total number of messages per second to over 75,000 using the FC Flow-Control layer, which buffers heavily using the "message list" capabilities of Horus [FR95a]. Horus easily reached the Ethernet 1007 Kbytes/second maximum bandwidth with a message size smaller than 1 kilobyte.

The performance test program has each member do exactly the same thing: send *k* messages and wait for *k (n -1)* messages of size *s*, where *s* is the number of members. This way we simulate an application that imposes a high load on the system while occasionally synchronizing on intermediate results.

Obtain Data from CACM paper

Figure 18-7: The left figure compares the one-way latency of 1-byte FIFO Horus messages over straight UDP and UDP with the Deering IP multicast extensions. The right figure compares the performance of total and FIFO order of Horus, both over UDP multicast.

Figure 18-7 depicts the one-way communication latency of 1-byte Horus messages. As can be seen in the top graph, hardware multicast is a big win, especially when the message size goes up. In the bottom graph, we compare FIFO to totally ordered communication. For small messages we get a FIFO one-way latency of about 1.5 milliseconds and a totally ordered one-way latency of about 6.7 milliseconds. A problem with the totally ordered layer is that it can be inefficient when senders send single messages at random, and with a high degree of concurrent sending by different group members. With just one sender, the one-way latency drops to 1.6 milliseconds.

Obtain Data from CACM paper

Figure 18-8: These graphs depict the message throughput for virtually synchronous, FIFO ordered communication over normal UDP and Deering UDP, and for totally ordering communication over Deering UDP.

Figure 18-8 shows the number of 1-byte messages per second that can be achieved for three cases. For normal UDP and Deering UDP the throughput is fairly constant. For totally ordered communication we see that the throughput becomes better if we send more messages per round (because of increased concurrency). Perhaps surprisingly, the throughput also becomes better as the number of members in the group goes up. The reason for this is threefold. First, with more members there are more senders. Second, with more members it takes longer to order messages, and thus more messages can be packed together and sent out in single network packets. Last, the ordering protocol allows only one sender on the network at a time, thus introducing flow control and reducing collisions.

18.7 Masking the Overhead of Protocol Layering

Although layering of protocols can be advocated as a way of dealing with the complexity of computer communication, it is also criticized for its performance overhead. Recent work by Van Renesse has yielded considerable insight regarding the design of protocols, which van Renesse uses to mask the overhead of layering in Horus. The fundamental idea is very similar to client caching in a file system. With these new techniques, he achieves an order of magnitude improvement in end-to-end message latency in the Horus communication framework, compared to the best latency possible using Horus without these optimizations. Over an ATM network, the approach permits applications to send and deliver messages of varying levels of semantics in about 85us, using a protocol stack that is written in ML, an interpreted functional language. In contrast, the performance figures shown in the previous section were for a version of Horus coded in C, and carefully optimzed by hand but without use of the protocol accelerator.

Having presented this material in seminars, the author has noticed that the systems community seems to respond to the very mention of the ML language with skepticsm, and it is perhaps appropriate to comment on this before continuing. First, the reader should keep in mind that a technology such as Horus is simply a tool that one uses to harden a system. It makes little difference whether such a tool is internally coded in C, assembler language, Lisp, or ML if it works well for the desired purpose. The decision to work with a version of Horus coded in ML is not one that would impact the *use* of Horus in applications that work with the technology through wrappers or toolkit interfaces. However, as we will see here and in Chapter 25, it does bring some important benefits for Horus itself, notably the potential for us to harden the system using formal software analysis tools. Moreover, although ML is often viewed as obscure and of academic interest, the version of ML used in our work on Horus is not really so different from Lisp or C++ once one becomes accustomed to the syntax. Finally, as we will see here, the performance of Horus coded in ML is actually better than that of Horus coded in C, at least for certain patterns of communication. Thus we would hope that the reader will recognize that the work reported here is in fact very practical.

As we saw in earlier chapters, modern network technology allows for very low latency communication. For example, the U-Net [EBBV95] interface to ATM achieves 75 microsecond round-trip communication as long as the message is 40 bytes or smaller. On the other hand, if a message is larger, it will not fit in a single ATM cell, significantly increasing the latency. This points to two basic concerns: first, that systems like Horus need to be designed to take full advantage of the potential performance of current communications technology, and secondly that to do so, it will be important that Horus protocols use small headers, and introduce minimal processing overhead.

Unfortunately, these properties are not typical of the protocol layers needed to implement virtual synchrony. Many of these protocols are complex, and layering introduces additional overhead of its own. One source of overhead is interfacing: crossing a layer costs some CPU cycles. The other is header overhead. Each layer uses its own header, which is prepended to every message and usually padded so that each header is aligned on a 4 or 8 byte boundary. Combining this with a trend to very large addresses (of which at least two per message are needed), it is impossible to have the total amount of header space be less than 40 bytes.

The Horus Protocol Accelerator (Horus PA) eliminates these overheads almost entirely, and offers the potential of a one to three orders of magnitude of latency improvement over the protocol implementations described in the previous subsection. For example, we looked at the impact of the Horus PA on an ML [MTH90] implementation of a protocol stack with five layers. The ML code is interpreted (although in the future it will be compiled), and therefore relatively slow compared to compiled C code. Nevertheless, between two SunOS user processes on two Sparc 20s connected by a 155 Mbit/sec ATM network, the Horus PA permits these layers to achieve a roundtrip latency of 175 microseconds, down from about 1.5 milliseconds in the original Horus system (written in C).

The Horus PA achieves its results using three techniques. First, message header fields that never change are only sent once. Second, the rest of the header information is carefully packed, ignoring layer boundaries, typically leading to headers that are much smaller than 40 bytes, and thus leaving room to fit a small message within a single U-Net packet. Third, a semi-automatic transformation is done on the send and delivery operations, splitting them into two parts: one that updates or checks the header but not the protocol state, and the other vice versa. The first part is then executed by a special packet filter (both in the send and the delivery path) to circumvent the actual protocol layers whenever possible. The second part is executed, as much as possible, when the application is idle or blocked.

18.7.1 Reducing Header Overhead

In traditional layered protocol systems, each protocol layer designs its own header data structure. The headers are concatenated and prepended to each user message. For convenience, each header is aligned to a 4 or 8 byte boundary to allow easy access. In systems like the x-Kernel or Horus, where many simple protocols may be stacked on top of each other, this may lead to extensive padding overhead.

Some fields in the headers, such as the source and destination addresses, never change from message to message. Yet, instead of agreeing on these values, they are frequently included in every message, and used as the identifier of the connection to the peer. Since addresses tend to be large (and getting larger to deal with the rapid growth the Internet), this results in significant use of space for what are essentially constants of the connection. Moreover, notice that the connection itself may already be identifiable from other information. On an ATM network, connections are "named" by a small 4 byte VPI/VCI pair, and every packet carries this information. Thus, constants such as sender and destination addresses are implied by the connection identifier and including them in the header is superfluous.

The Horus PA exploits these observations to reduce header sizes to a bare minimum. The approach starts by dividing header fields into four *classes:*

- *Connection Identification* — fields that never change during the period of a connection, such as sender and destination.
- *Protocol-specific Information* — fields that are important for the correct delivery of the particular message frame. Examples are the sequence number of a message, or the message type (Horus messages have types, such as "data", "ack", or "nack"). These fields must be deterministically implied by the protocol "state", and not on the message contents or the time at which it was sent.
- *Message-specific information* — fields that need to accompany the message, such as the message length and checksum, or a timestamp. Typically, such information depends only on the message, and not on the protocol state.
- *Gossip* — fields that technically do not need to accompany the message, but are included for efficiency.

Each layer is expected to declare the header fields that it will use during initialization, and subsequently accesses fields using a collection of highly optimized functions implemented by the Horus PA. These functions extract values directly from headers if they are present, and otherwise compute the appropriate field value and return that instead. This permits the Horus PA to precompute header templates that have optimized layouts, with a minumum of wasted space.

Horus includes the Protocol-specific and Message-specific information in every message. Currently, although not technically necessary, Gossip information is also always included, since it is usually small. However, since the Connection Identification fields never change, they are only included occasionally because they tend to be large.

A 64-bit “mini-header” is placed on each message to indicate which headers it actually includes. Two bits of this are used to indicate whether or not the connection identification is present in the message and to destinate the byte-ordering for bytes in the message. The remaining 62-bits are a *connection cookie,* which is a magic number established in the connection identification header and selected randomly, to identify the connection.

The idea is that the first message sent over a connection will a connection identifier, specifying the cookie to use, and providing an initial copy of the connection identification fields. Subsequent messages need only contain the identification field if it has changed. Since the Connection Identification tend to include very large identifiers, this mechanism reduces the amount of header space in the normal case significantly. For example, in the version of Horus that Van Renesse used in his tests, the connection identification typically occupies about 76 bytes.

18.7.2 Eliminating Layered Protocol Processing Overhead

In most protocol implementations, layered or not, a great deal of processing must be done between the application's send operation, and the time that the message is actually sent out onto the network. The same is true between the arrival of a message and the delivery to the application. The Horus PA reduces the length of the critical path by updating the protocol state only after a message has been sent or delivered, and by precomputing any statically predictable protocol-specific header fields, so that the necessary values will be known *before* the application generates the next message (Figure 18-9). These methods work because the protocol-specific information for most messages can be predicted (calculated) before the message is sent or delivered. (Recall that, as noted above, such information must not depend on the message contents or the time on which it was sent). Each connection maintains a predicted protocol-specific header for the next send operation, and another for the next delivery (much like a read-ahead strategy in a file system). For sending, the gossip information can be predicted as well, since this does not depend on the message contents. The idea is a bit like that of prefetching in a file system.

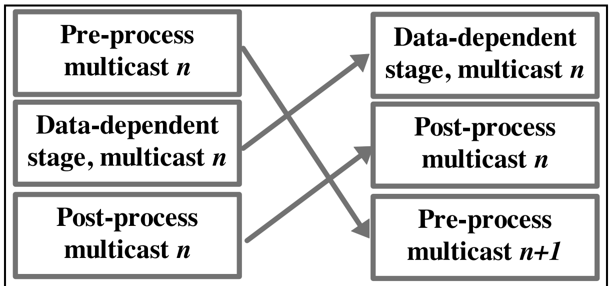

Pre-process multicast n
Data-dependent stage, multicast n
Post-process multicast n
Data-dependent stage, multicast n
Post-process multicast n
Pre-process multicast n+1

Figure 18-9: Restructuring a protocol layer to reduce the critical path. By moving data-dependent code to the front, delays for sending the next message are minimized. Post-processing of the current multicast and preprocessing of the next multicast (all computation that can be done before seeing the actual contents of the message) are shifted to occur after the current multicast has been sent, and hence concurrently with application-level computing.

Thus, when a message is actually sent, only the message-specific header will need to be generated. This is done using a *packet filter* [MRA87], which is constructed at the time of layer initialization. Packet filters are programmed using a simple programming language (a dialect of ML), and operate by extracting information from the message needed to form the message-specific header. A filter can also hand off a message to the associated layer for special handling, for example if a message fails to satisfy some assumption that was used in predicting the protocol-specific header. In the usual case, the message-specific header will be computed, other headers are prepended from the precomputed versions, and the message is transmitted with no additional delay. Because the header fields have fixed and precomputed sizes, a header template can be filled in with no copying, and scatter-send/scatter-gather hardware used to transmit the header and message as a single packet without copying them first to a single place. This reduces the computational cost of sending or delivering a message to a bare minimum, although it leaves some background costs in the form of prediction code that must be executed before the next message is sent or delivered

18.7.3 Message Packing

The Horus PA as described so far will reduce the latency of individual messages significantly, but only if they are spaced out far enough to allow time for post-processing. If not, messages will have to wait until the post-processing of every previous message completes (somewhat like a process that reads file system records faster than they can be prefetched). To reduce this overhead, the Horus PA uses *message packing* [FR95] to deal with backlogs. The idea is a very simple one. After the post-processing of a send operation completes, the PA checks to see if there are messages waiting. If there are more than one, the PA will pack these messages together into a single message. The single message is now processed in the usual way, which takes only one pre-processing and post-processing phase. When the packed message is ready for delivery, it is unpacked and the messages are individually delivered to the application.

Returning to our file system analogy, the approach is similar to one in which the application could indicate that it plans to read three 1k data blocks. Rather than fetching them one by one, the file system can now fetch them all at the same time. Doing so amortizes the overhead associated with fetching the blocks, permitting better utilization of network bandwidth.

18.7.4 Performance of Horus with the Protocol Accelerator

The Horus PA dramatically improved the performance of the system over the base figures described earlier (which were themselves comparable to the best performance figures cited for other systems). With the accelerator, one-way latencies dropped to as little as 85us (compared to 35us for the U-Net implementation over which the accelerator was tested). As many as 85,000 one-byte messages could be sent and delivered per second, over a protocol stack of five layers implementing the virtual synchrony model within a group of two members. For RPC-style interactions, 2,600 round-trips per second were achieved. These latency figures, however, represent a best-case scenario in which the frequency of messages was low enough to permit the predictive mechanisms to operate; when they become overloaded, latency increases to about 425us for the same test pattern. This points to a strong dependency of the method on the speed of the code used to implement layers.

Van Renesse's work on the Horus PA made use of a version of the ML programming language which was interpreted, not compiled. ML turns out to be a very useful language for specifying Horus layers: it lends itself to formal analysis and permits packet filters to actually be constructed at runtime; moreover, the programming model is well matched to the functional style of programming used to implement Horus layers. ML compiler technology is rapidly evolving, and when the Horus PA is moved to a compiled version of ML the sustainable load should rise and these maximum latency figures drop.

The Horus PA does suffer from some limitations. Message fragmentation and reassembly is not supported by the PA, hence the pre-processing of large messages must be handled explicitly by the protocol stack. Some technical complications result from this design decision, but it reduces the complexity of the PA and hence improves the maximum performance achievable using it. A second limitation is that the PA must be used by all parties to a communication stack. However, this is not an unreasonable restriction, since Horus has the same sort of limitation with regard to the stacks themselves (all members of a group must use identical or at least compatible protocol stacks).

18.8 Scalability

Up to the present, this text as largely overlooked issues associated with protocol scalability. Although a serious treatment of scalability in the general sense might require a whole textbook in itself, the purpose of this section is to set out some general remarks on the subject, as we have approached it in the Horus project. It is perhaps worthwhile to comment that, overall, surprisingly little is known about scaling reliable distributed systems.

If one looks at the scalability of Horus protocols, as we did earlier in presenting some basic Horus performance figures, it is clear that Horus performs well for groups with small numbers of members, and for moderately large groups when IP multicast is available as a hardware tool to reduce the cost of moving large volumes of data to large numbers of destinations. Yet although these graphs are honest, they may be misleading. In fact, as systems like Horus are scaled to larger and larger numbers of participating processes, they experience steadily growing overheads, in the form of acknowldgements and negative acknowledgements from the recipient processes to the senders. A consequence is that if these systems are used with very large numbers of participating processes, the "backflow" associated with these types of messages and with flow control becomes a serious problem.

A simple thought experiment suffices to illustrate that there are probably fundamental limits on reliability in very large networks. Suppose that a communication network is extremely reliable, but that the processes using it are designed to distrust that network, and to assume that it may actually malfunction by losing messages. Moreover, assume that these processes are in fact closely rate-matched (the consumers of data keep up with the producers), but again that the system is designed to deal with individual processes that lag far behind. Now, were it not for the backflow of messages to the senders, this hypothetical system might perform very well near the limits of the hardware. It could potentially be scaled just by adding new recipient processes, and with no changes at all, continue to provide a high observed level of reliability.

However, the backflow messages will substantially impact this simple and rosy scenario. They represent a source of overhead, and in the case of flow control messages, if they are not received, the sender may be forced to stop and wait for them. Now, the performance of the sender side is coupled to the timely and reliable reception of backflow messages, and as we scale the number of recipients connected to the system, we can anticipate a traffic jam phenomenon at the sender's interface (protocol designers call this an acknowledgement "implosion") that will cause traffic to get increasingly bursty and performance to drop. In effect, the attempt to protect against the mere risk of data loss or flow control mismatches is likely to slash the maximum achievable performance of the system. Now, obtaining a stable delivery of data near the limits of our technology will become a tremendously difficult juggling problem, in which the protocol developer must trade the transmission of backflow messages against their performance impact.

Graduate students Guerney Hunt and Michael Kalantar have studied aspects of this problem in their doctoral dissertations at Cornell University, both using special purpose experimental tools (that is, neither actually experimented on Horus or a similar system; Kalantar, in fact, worked mostly with a simulator). Hunt's work was on flow control in very large scale system. He concluded that most forms of backflow were unworkable on a large scale, and ultimately proposed a rate-based flow control scheme in which the sender limits the transmission rate for data to match what the receivers can accomodate [Hunt95]. Kalantar looked at the impact of multicast ordering on latency, asking how frequently an ordering property such as causal or total ordering would significantly impact the latency of message delivery [Kal95]. He found that although ordering had a fairly small impact on latency, there were other much important phenomena that represented serious potential concerns.

In particular, Kalantar discovered that as he scaled the size of his simulation, message latencies tended to become unstable and bursty. He hypothesized that in large-scale protocols, the domain of stable performance becomes smaller and smaller. In such situations, a slight perturbation of the overall system, for example because of a lost message, could cause much of the remainder of the system to block because of reliability or ordering constraints. Now, the system would shift into what is sometimes called a *convoy* behavior, in which long message backlogs build up and are never really eliminated; they may shift from place to place, but stable, smooth delivery is generally not restored. In effect, a bursty scheduling behavior represents a more stable configuration of the overall system than one in which message delivery is extremely regular and smooth, at least if the number of recipients is large and the presented load is a substantial percentage of the maximum achievable (so that there is little slack bandwidth with which the system can catch up after an overload develops).

Hunt's and Kalantar's observations are not really surprising ones. It makes sense that it should be easy to provide reliability or ordering when far from the saturation point of the hardware, and much harder to do so as the communication or processor speed limits are approached.

Over many years of working with Isis and Horus, the author has gained considerable experience with these sorts of scaling and flow control problems. Realistically, the conclusion can only be called a mixed one. On the positive side, it seems that one can fairly easily build a reliable system if the communication load won't exceed, perhaps, 20% of the capacity of the hardware. With a little luck, one can even push as high as perhaps 40% of the hardware. (Happily, hardware is becoming so fast that this may still represent a very satisfactory level of perfomance long into the future!)

However, as the load presented to the system rises beyond this threshold, or if the number of destinations for a typical message becomes very large (hundreds), it becomes increasingly difficult to guarantee reliability and flow control. A fundamental tradeoff seems to be present: one can send the data and hope that it will usually arrive, and by doing so, may be able to operate quite reliably near the limits of the hardware. But, of course, if a process falls behind, it may lose large numbers of messages before it recovers, and no mechanism is provided to let it recover these from any form of backup storage. On the other hand, one can operate in a less demanding performance range, and in this case provide reliability, ordering, and performance guarantees. In between the two, however, lies a domain that is extremely difficult in an engineering sense and often requires a very high level of software complexity, which will necessarily reduce reliability. Moreover, one can raise serious questions about the stability of message passing systems that operate in this intermediate domain, where the load presented is near the limits of what can be accomplished. The typical experience in such systems is that they perform well, most of the time, but that once something fails, the system falls so far behind that it can never again catch up: in effect, any perturbation can shift such a system into the domain of overloads and hopeless backlogs.

Where does Horus position itself in this spectrum? Although the performance data shown earlier may suggest that the system seeks to provide scalable reliability, it is more likely that successful Horus applications will seek one property or the other, but not both at once, or at least not both when performance is demanding. In Horus, this is done by using multiple protocol stacks, in which the protocol stacks providing strong properties are used much less frequently, while the protocol stacks providing weaker reliability properties may be used for high volume communication.

As an example, suppose that Horus were to be used to build a stock trading system. It might be very important to ensure that certain clases of trading information will reach all clients, and for this sort of information, a stack with strong reliability properties could be used. But as a general rule, the majority of communication in such systems will be in the form of bid/offered pricing, which may not need to be delivered quite so reliably: if a price quote is dropped, the loss won't be serious so long as the next quote has a good probability of getting through. Thus, one can visualize such a system as having two superimposed architectures: one, which has much less traffic, and much stronger reliability requirements, and a second one with much greater traffic but weaker properties. We saw a similar structure in the Horus application to the CMT system: here, the stronger logical properites were reserved for coordination, timestamp generation, and agreement on such data as system membership. The actual flow of video data was through a protocol stack with very different properties: stronger temporal guarantees, but weaker reliability properties. In building scalable reliable systems, such tradeoffs may be intrinsic.

In general, this leads to a number of interesting problems, having to do with the synchronization and ordering of data when multiple communication streams are involved. Researchers at the Hebrew University in Jerusalem, working with a system similar to Horus called Transis (and with Horus itself), have begun to investigate this issue. Their work, on providing strong communication semantics in applications that mix multiple "quality of service" properties at the transport level, promises to make such multi-protocol systems more and more manageable and controlled [Iditxx].

More broadly, it seems likely that one could develop a theoretical argument to the effect that reliability properties are fundamentally at odds with high performance. While one can scale reliable systems, they appear to be intrinsically unstable if the result of the scaling is to push the overall system anywhere close to the maximum performance of the technology used. Perhaps some future effort to model these classes of systems will reveal the basic reasons for this relationship and point to classes of protocols that degrade gracefully while remaining stable under steadily increasing scale and load. Until then, however, the heuristic recommended by this writer is to scale systems, by all means, but to be extremely careful not to expect the highest levels of reliabilty, performance and scale simultaneously. To do so is simply to move beyond the limits of problems that we know how to solve, and may be to expect the impossible. Instead, the most demanding systems must somehow be split into subsystems that demand high performance but can manage with weaker reliability properties, and subsystems that need reliabilty, but will not be subjected to extreme performance demands.

18.9 Related Readings

Chapter 26 includes a review of related research activities, which we will not duplicate here. On the Horus system: [BR96, RBM96, FR95]. Horus used in a real-time telephone switching application: Section 20.3 [FB96]. Virtual fault-tolerance: [BS95]. Layered protocols: [CT87, AP93, BD95, KP93, KC94]. Event counters: [RK79]. The Continuous Media Toolkit: [RS92]. U-Net [EBBV95]. Packet filters (in Mach) [MRA87]. Chapter 25 discusses verification of the Horus protocols in more detail; this work focuses on the same ML implementation of Horus to which the Protocol Accelerator was applied.

19. Security Options for Distributed Settings

The use of distributed computing systems for storage of sensitive data and in commercial applications has created significant pressure to improve the security options available to software developers. Yet distributed systems security has many possible interpretations, corresponding to very different forms of guarantees, and even the contemporary distributed systems that claim to be secure often suffer from basic security weaknesses. In this chapter we will review some of the major security technologies, look at the nature of their guarantee and of their limitations, and discuss some of the issues raised when one asks that a security system also guarantee high availability.

The technologies we consider here span a range of approaches. At the weak end of the spectrum are firewall technologies and other *perimeter defense mechanisms* that operate by restricting access or communication across specified system boundaries. These technologies are extremely popular but very limited in their capabilities. In particular, once an intruder has found a way to work around the firewall or log into the system, the protection benefit is lost.

Internal to a distributed system one typically finds *access control mechanisms* that are often based on the UNIX model of user and group id's, which are employed to limit access to shared resources such as file systems. When these are used in stateless settings, serious problems arise, which we will discuss here and will return to later, in Chapter 23. Access control mechanisms rarely extend to communication, and this is perhaps their most serious security exposure. In fact, many communication systems are open to attack by a clever intruder who is able to guess what port numbers will be used by the protocols within the system: secrecy of port numbers is a common security dependency in modern distributed software.

Stateful protection mechanisms operate by maintaining strong notions of session and channel state, and authenticating use at the time that communication sessions are established. These schemes adopt the approach that after a user has been validated the difficulty of breaking into the user's session will represent an obstacle to intrusion.

Authentication based security systems employ some scheme to authenticate the user who is running each application; the method may be highly reliable or less so depending upon the setting [NS78, Den84]. Individual communication sessions are then protected using some form of key that is negotiated using a trusted agent. Messages may be encrypted or signed in this approach, resulting in very strong security guarantees. However, the costs of the overall approach can also be high, because of the intrinsically high costs of data encryption and signature schemes. Moreover, such methods may involve non-trivial modifications of the application programs that are used, and may be unsuitable for embedded settings in which no human user would be available to periodically enter passwords or other authentication data. The best known system of this sort is Kerberos, developed by MIT's project Athena, and our review will focus on the approaches used in that system [SNS88, Sch94].

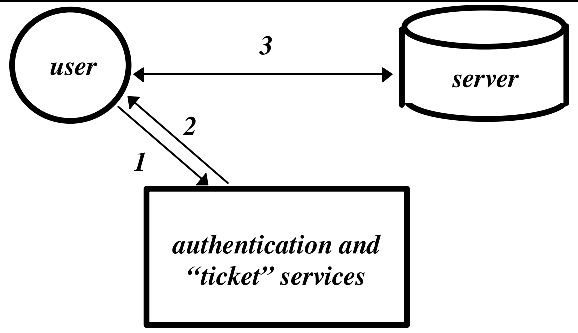


Figure 19-1: MIT's Project Athena developed the Kerberos security architecture. Kerberos or a similar mechanism is found at the core of many distributed systems security technologies today. In this approach, an authentication service is used as a trusted intermediary to create secure channels, using DES encryption for security. During step (1), the user employs a password as a DES key to request that a connection be established to the remote server. The authentication server, which knows the user's password, constructs a session key which is sent back in duplicated form, one copy readable to the user and one encrypted with the server's secret key (2). The session key is now used between the user and server (3), providing the server with trusted information about user identification and whereabouts. In practice, Kerberos avoids the need to keep user passwords around by trading the user's password for a session to the "ticket granting service", which then acts as the user's proxy in establishing connections to necessary servers, but the idea is unchanged. Kerberos session keys expire and must be periodically refreshed, hence even if an intruder gains physical access to the user's machine, the period during which illicit actions are possible is limited.

Multi-level distributed systems security architectures are based on a government security standard that was developed in the mid 1980's. The security model here is very strong, but has proved to be difficult to implement and to require extensive effort on the part of application developers. Perhaps for these reasons, this approach has not been widely successful. Moreover, the pressure to use off the shelf technologies has made it difficult even for the government to build systems that enforce multi-level security.

Traditional security technologies have not considered availability when failures occur, creating a exposure to attacks whereby critical system components are shut down, overloaded, or partitioned away from application programs that depend upon them. Recent research has begun to address these concerns, resulting in a new generation of highly available security technologies. However, when one considers failures in the context of a security subsystem, the benign failure models of earlier chapters must be called into question. Thus, work in this area has included a reexamination of Byzantine failure models, asking if extremely robust authentication servers can be built that will remain available even if Byzantine failures occur. Progress in this direction has been encouraging, as has work on using process groups to provide security guarantees that go beyond those available in a single server.

Looking to the future, technologies supporting digital cash and digital commerce are likely to be of increasing importance, and will often depend upon the use of trusted "banking" agents and strong forms of encryption, such as the RSA or DES standards [DH79, RSA78, DES88]. Progress in this area has been very rapid and we will review some of the major approaches.

Yet, if the progress in distributed systems security has been impressive, the limitations on such systems remain quite serious. On the whole, it remains difficult to secure a distributed system and very hard to add security to a technology that already exists and must be treated as a form of black box. The best known technologies, such as Kerberos, are still used only sporadically. This makes it hard to implement customized security mechanisms, and leaves the average distributed system quite open to attack. Break-ins and security violations are extremely common in the most standard distributed computing environments, and there seems to be at best a shallow commitment by the major software vendors to improving the security of their basic product lines. These observations raise troubling questions about the security to be expected from the emerging generation of extremely critical distributed systems, many of which will be implemented using standard software solutions on standard platforms. Until distributed systems security is difficult to *disable*, as opposed to being difficult to enable, we may continue to read about intrusions of increasingly serious natures, and will continue to be at risk of serious intrusions into our personal medical records, banking and financial systems, and personal computing environments.

19.1 Perimeter Defense Technologies

It is common to protect a distributed system by erecting barriers around it. Examples include the password control associated with dial-in ports, dial-back mechanisms that some systems use to restrict access to a set of predesignated telephone numbers, and firewalls through which incoming and outgoing messages must pass. Each of these technologies has important limitations.

Password control systems are subject to attack by password guessing mechanisms, and by intruders who find ways to capture packets containing passwords as they are transmitted over the internet or some other external networking technology. So-called password "sniffers" became a serious threat to systems security in the mid 1990's, and illustrate that the general internet is not the benign environment that was in the early days of distributed computing, when most internet users knew each other by name. Typical sniffers operate by exhibiting an IP address for some other legitimate machine on the network, or by placing their network interfaces into promiscuous mode, in which all passing packets will be accepted. They then scan the traffic captured for packets that might have originated in a login sequence. With a bit of knowledge about how such packets normally look, it is not hard to reliably capture passwords as they are routed through the internet. Sniffers have also been used to capture credit card information and to intrude into email correspondence.

Dialup systems are often perceived as being more secure than direct network connections, but this is not necessarily this is the case. The major problem is that many systems use their dialup connections for data and file transfer and as a sending and receiving point for fax communications, and hence the corresponding telephone numbers are stored in various standard data files, often with connection information. An intruder who breaks into one system may in this manner learn dialup numbers for other systems, and may even find logins and passwords that will make it easy to break in. Moreover, the telephone system itself is increasingly complex and, as an unavoidable side-effect, increasingly vulnerable to intrusions. This creates the threat that a telephone connection over which communication protocols are running may be increasingly open to attack by a clever hacker who breaks into the telephone system itself.

Dialback mechanisms, whereby the system calls the user back, clearly increase the hurdle that an intruder must cross to penetrate a system relative to one in which the caller is assumed to be a potentially legitimate user. However, such systems depend for their security upon the integrity of the telephone system, which, a we have noted, can be subverted. In particular, the emergence of mobile telephones and the introduction of mobility mechanisms into telephone switching systems creates a path by which an intruder can potentially redirect a telephone dialback to a telephone number other than the intended one. Such a mechanism is a good example of a security technology that can protect against benign attacks but would be considerably more exposed to well-organized malicious ones.

Firewalls have become popular as a form of protection against communication-level attacks on distributed systems. Many of these technologies operate using *packet filters* and must be instantiated at all the access points to a distributed network. Each copy of the firewall will have a *filtering control policy* in the form of a set of rules for deciding which packets to reject and which to pass through; although firewalls that can check packet content have been proposed, typical filtering is on the basis of protocol type, sender and destination addresses, and port numbers. Thus, for example, packets can be allowed through if they are addressed to the email or ftp server on a particular node, and otherwise rejected. Often, firewalls are combined with *proxy* mechanisms that permit file transfer and remote log in through an intermediary system which enforces further restrictions. The use of proxies for the transfer of public web pages and ftp areas has also become common: in these cases, the proxy is configured as a mirror of some protected internal file system area, copying changed files to the less secure external area periodically.

Other technologies that are commonly used to implement firewalls include application-level proxies and routers. With these approaches, small fragments of user-supplied code (or programs obtained from the firewall vendor) are permitted to examine the incoming and outgoing packet streams. These programs run in a loop, waiting for the next incoming or outgoing message, performing an acceptance test upon it, and then either discarding the message or permitting it to continue. The possibility of logging the message and maintaining additional statistics on traffic is also commonly supported.

The major problem associated with firewall technologies is that they represent a single point of failure: if the firewall is breached, the intruder may gain essentially free run of the enclosed system. Intruders may know of ways to attack specific firewalls, perhaps learned through study of the code used to implement the firewall, secret backdoor mechanisms included by the original firewall developers for reasons of their own, or by compromising some of the software components included into the application itself. Having broken in, it may be possible to establish connections to servers that will be fooled into trusting the intruder or to otherwise act to attack the system from within. Reiterating the point made above, an increasingly serious exposure is created by the explosive growth of telecommunications. In the past, a dedicated "leased line" could safely be treated as an internal technology that links components of a distributed system within its firewall. As we move into the future, such a line must be viewed as a potential point of intrusion.

These considerations are increasingly leading corporations to implement what are called *virtual private networks* in which communication is authenticated (typically using a hardware signature scheme) so that all messages originating outside of the legitimately accepted sources will be rejected. In settings where security is vital, these sorts of measures are likely to considerably increase the robustness of the network to attack. However, the cost remains high, and a consequence it seems unlikely that the "average" network will offer this sort of cryptographic protection for the forseeable future. Thus, while the prospects for strong security may be promising in certain settings, such as military systems or electronic banking systems, the more routine computing environments on which the great majority of sensitive applications in fact run remain open to a great variety of attacks and are likely to continue to have such exposures well into the next decade.

This situation may seem pessimistic, and yet in many respects, the story is far from over. Although it may seem extremely negative to think in such terms, it is probable that future information terrorists and warfare tactics will include some of these forms of attack and perhaps others that are hard to anticipate until they have first been experienced. Short of a major shift in mindset on the part of vendors, the situation is very likely to improve, and even then, we may need to wait until a generation of new technologies has displaced the majority of the existing infrastructure, a process that takes some 10 to 15 years at the time of this writing. Thus, information security is likely to remain a serious problem at least until the year 2010 or later.

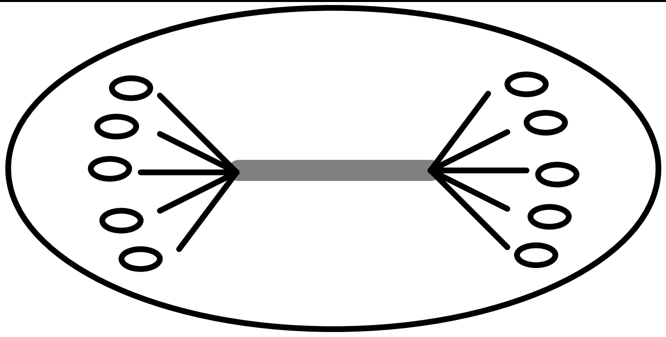


Figure 19-2: A long-haul connection internal to a distributed system (gray) represents a potential point of attack. Developers often protect systems with firewalls on the periphery but overlook the risk that the communications infrastructure may itself be compromised, offering the intruder a back-door into the protected environment. Although some corporations are protecting themselves against such threats using encryption techniques to create virtual private networks, most "mundane" communication systems are increasingly at risk.

Although we will now move on to other topics in security, we note that defensive management techniques can be coupled with security-oriented wrappers to raise the barriers in systems that use firewall technologies for protection. We will return to this subject in Chapter 23.

19.2 Access Control Technologies

Access control techniques operate by restricting use of system resources on the basis of user or group identifiers that are typically fixed at login time, for example by validation of a password. It is typical that these policies trust the operating system, its key services, and the network. In particular, the login program is trusted to obtain the password and correctly check it against the database of system passwords, granting the user permission to work under the desired user-id or group-id only if a match is detected, the login system trusts the file server or Network Information Server to respond correctly with database entries that can be safely used in this authentication process, and the resource manager (typically, an NFS server or database server) trusts the ensemble, believing that all packets presented to it as "valid NFS packets" or "valid XYZbase requests" in fact originated at a trusted source.[19]

These many dependencies are only rarely enforced in a rigorous way. Thus, one could potentially attack an access control system by taking over a computer, rebooting it as the "root" or "superuser", directing the system to change the user id to any desired value, and then starting to work as the specified user. An intruder could replace the standard login program with a modified one, introduce a fake NIS that would emulate the NIS protocol but substitute faked password records. One could even code one's own version of the NFS client protocol which, operating from user space as a normal RPC application, could misrepresent itself as a trusted source of NFS requests. All of these attacks on the NFS have been used successfully at one time or another, and many of the loopholes have been closed by one or more of the major vendors. Yet the fact remains that file and database servers continue to be largely trusting of the major operating system components on the nodes where they run and where their clients run.

Perhaps the most serious limitation associated with access control mechanisms is that they generally do not extend to the communication subsystem: typically, any process can issue an RPC message

[19] Not all file systems are exposed to such problems. For example, the AFS file system has a sophisticated stateful client-server architecture that is also much more robust to attack. AFS has become popular, but remains much less widely used than NFS.

to any address it wishes to place in a message, and can attempt to connect to any stream endpoint for which it possesses an address. In practice, these exposures are hard to exploit because a process that undertakes to do so will need to guess the addresses being used by the applications is attacks. Precisely to reduce this risk, many applications exploit *randomly generated* endpoint addresses, so that an intruder would be forced to guess a large pseudo-random number to break into a critical server. However, pseudo-random numbers may be less random than intended, particularly if an intruder has access to the pseudo-random number generation scheme and samples of the values recently produced.

Such break-ins are more common than one might expect. For example, in 1994 an attack on X11 servers was discovered in which an intruder found a way to deduce the connection port number that would be used. Sending a message that would cause the X11 server to prepare to accept a new connection to a shell command window, the intruder instead managed to connect to the server and to send a few commands to it. Not surprisingly, this proved sufficient to open the door to a full-fledged penetration. Moreover, the attack was orchestrated in such a manner as to trick typical firewalls into forwarding these poisoned messages even through the normal firewall protection policy should have required that they be rejected. Until the nature of the attack was understood, the approach permitted intrusion into a wide variety of firewall protected systems.

To give some sense of how exposed typical distributed systems currently are, the following table presents some of the assumptions made by the NFS file server technology when it is run without the security technology available from some vendors (in practice, NFS security is rarely enabled in systems that are protected by firewalls; the security mechanisms are hard to administer in heterogeneous environments and can slow the NFS system down significantly). We have listed typical assumptions of the NFS, the normal reason that this assumption holds, and one or more attacks that operate by emulation of the normal NFS environment in a way that the server is unable to detect. The statelessness of the NFS server makes it particularly easy to attack, but most client-server systems have similar dependencies and hence are similarly exposed.

NFS assumption	***Dependent on...***
O/S integrity	***NFS protocol messages originate only in trusted subsystems or the kernel***
Attacks: introduce a computer running an "open" operating system, modify the NFS subsystem. Develop a user-level program that implements the NFS client protocol, use it to emulate a legitimate NFS client issuing requests under any desired user id.	
Authentication	***Assumes that user and group ID information is valid***
Attacks: Spoof the Network Information Server or NFS response packets so that authentication will be done against a falsified password database. Compromise the login program. Reboot the system or login using the "root" or "superuser" account; then change the user id or group id to the desired one and issue NFS requests.	
Network integrity	***Assumes that communication over the network is secure***
Attacks: Intercept network packets, reading file system data and modifying data written. Replay NFS commands, perhaps with modifications.	

Figure 19-3: When the NFS security mechanisms are not explicitly enabled, many attacks become possible. Other client-server technologies, including database technologies, often have similar security exposures.

One can only feel serious concern when these security exposures are contemplated against the backdrop of increasingly critical applications that trust client-server technologies such as NFS. For example, it is very common to store sensitive files on unprotected NFS servers. As we noted, there is an NFS security standard, but it is vendor-specific, and hence may be impractical to use in heterogeneous environments. A hospital system, for example, is necessarily heterogeneous: the workstations used in such systems must interoperate with a great variety of special purpose devices and peripherals, produced by many vendors. Thus, in precisely the setting one might hope would use strong data protection, one typically finds priorietary solutions or unprotected use of standard file servers! Indeed, many hospitals might be prevented from using a strong security policy because so many individuals potentially need access to a patient record that any form of restriction would effectively be nullified.

Thus, in a setting where protection of data is not just important but is actually legally mandated, it may be very easy for an intruder to break in. While such an individual might find it hard to walk up to a typical hospital computing station and break through its password protection, by connecting a portable laptop computer to the hospital ethernet (potentially a much easier task), it would often be trivial to gain access to the protected files stored on the hospitals servers. Such security exposures are already a potentially serious issue, and the problem will only grow more serious with time.

When we first discussed the NFS security issues, we pointed out that there are other file systems that do quite a bit better in this regard, such as the AFS system developed originally at Carnegie Mellon University, and now commercialized by Transarc. AFS, however, is not considered to be standard and many vendors provide NFS as part of their basic product line, while AFS is a commercial product from a third party. Thus, the emergence of more secure file system technologies faces formidable practical barriers. It is unfortunate but entirely likely that the same is true for other reliability and security technologies.

19.3 Authentication Schemes and Kerberos

The weak points of typical computing environments are readily seen to be their authentication mechanisms and their blind trust in the security of the communication subsystem. Best known among the technologies that respond to these issues is MIT's Kerberos system, developed as part of Project Athena.

Kerberos makes use of encryption, hence it will be useful to start by reviewing the existing encryption technologies and their limitations. Although a number of encryption schemes have been proposed, the most popular ones at the time of this writing are the RSA public key algorithms and the DES encryption standard.

19.3.1 RSA and DES

RSA [RSA78] is an implementation of a public key cryptosystem [DH79] that exploits properties of modular exponentiation. In practice, the method operates by generating pairs of *keys* that are distributed to the users and programs within a distributed system. One key within each pair is the *private key* and is kept secret. The other key is *public*, as is an encryption function *crypt(key, object)*. The encryption function has a number of useful properties. Suppose that we denote the public key of some user as K and the private key of that user as K^{-1}. Then $crypt(K, crypt(K^{-1}, M)) = crypt(K^{-1}, crypt(K, M)) = M$. That is, encryption by the public key will decrypt an object encrypted previously with the private key, and vice versa. Moreover, even if keys A and B are unrelated, encryption is commutative: $crypt(A, crypt(B, M)) = crypt(B, crypt(A, M))$.

In typical use, public keys are published in some form of trusted directory service [Bir85, For95]. If process A wants to send a secure message to process B, that could only have originated in process A and can only be read by process B, A sends $crypt(A^{-1}, crypt(B, M))$ to B, and B computes $crypt(B^{-1}, crypt(A, M))$. to extract the message. Here, we have used A and A^{-1} as shorthand's for the public and private keys of process A, and similarly for B. A can send a message that only B can read by computing the simpler $crypt(B, M)$, and can sign a message to prove that the message was seen by A by attaching $crypt(A^{-1}, digest(M))$ to the message, where $digest(M)$ is a function that computes some sort of small number that reflects the contents of M, perhaps using an error-correcting code for this purpose. Upon reception, a process B can compute the digest of the received message and compare this with the result of decrypting the signature sent by A using A's public key. The message can be validated by verifying that these values match [Den84].

A process can also be asked to encrypt or sign a blinded message when using the RSA scheme. To solve the former problem, process A is presented with M' = *crypt*(B, M). If A computes M'' = *crypt*(A^{-1}, M') than *crypt*(B^{-1},M'') will yield *crypt*(A^{-1}, M) without A having ever seen M. Given an appropriate message digest function, the same approach also allows a process to sign a message without being able to read that message.

In contrast, the DES standard [DES77, DH79] is based on shared secret keys, in which two users or processes that exchange a message will both have a copy of the key for messages sent between them. Separate functions are provided for encryption and decryption of a message. Like the RSA scheme, DES can also be used to encrypt a digest of a message as a proof that the message has not been tampered with. Blinding mechanisms for DES are, however, not available at the present time.

DES is the basis of a government standard which specifies a standard key size and can be implemented in hardware. Although the standard key size is large enough to provide security for most applications, the key is still small enough to permit it to be broken using a supercomputing system or a large number of powerful workstations in a distributed environment. This is viewed by the government as a virtue of the scheme, because the possibility is thereby created of decrypting messages for purposes of criminal investigation or national security. When using DES, it is possible to convert plain text (such as a password) into a DES key; in effect, a password can be used to encrypt information so that it can only be decrypted by a process that also has a copy of that password. As will be seen below, this is the central feature that makes possible DES-based authentication architectures such as the Kerberos one [SNS88, Sch94].

More recently, a security standard has been proposed for use in telecommunications environments. This standard, Capstone, was designed for telephone communication but is not specific to telephony, and involves a form of key for each user and supports what is called *key escrow* whereby the government is able to reconstruct the key by combining two portions of it, which are stored in secure and independent locations [Den96]. The objective of this work is to permit secure and private use of telephones while preserving the government's right to wiretap with appropriate court orders. The Clipper chip, which implements Capstone in hardware, is also used in the Fortezza PCMCIA card, described further in Section 19.3.4.

Both DES and the Capstone security standard are the subjects of vigorous debate. On the one hand, such methods limit privacy and personal security, because the government is able to break both schemes and indeed may have taken steps to make them easier to break than is widely appreciated. On the other hand, the growing use of information systems by criminal organizations clearly poses a serious threat to security and privacy as well, and it is obviously desirable for the government to be able to combat such organizations. Meanwhile, the fundamental security of methods such as RSA and DES is not known. For example, although it is conjectured that RSA is very difficult to break, in 1995 it was shown that in some cases, information about the amount of time needed to compute the *crypt* function could provide data that substantially reduces the difficulty of breaking the encryption scheme. Meanwhile, clever uses of large numbers of computers have made it possible to break DES encryption unexpectedly rapidly. These ongoing tensions between social obligations of privacy and security and the public obligation of the government to oppose criminality, and between the strength of cryptographic systems and the attacks upon them, can be expected to continue into the coming decades.

19.3.2 Kerberos

The Kerberos system is a widely used implementation of secure communication channels, based on the DES encryption scheme [SNS88, Sch94]. Integrated into the DCE environment, Kerberos is currently a de-facto standard in the UNIX community. The approach genuinely offers a major improvement in security over that which is traditionally available within UNIX. Its primary limitation is that applications using Kerberos must be modified to create communication channels using the Kerberos secure channel facilities. Although this may seem to be a minor point, it represents a surprisingly serious one for potential Kerberos users, since application software that makes use of Kerberos is not yet common. Nonetheless, Kerberos has had some important successed; one of these is its use in the AFS system, discussed earlier [Sat89].

The basic Kerberos protocols revolve around the use of a trusted authentication server which creates session keys between clients and servers upon demand. The basic scheme is as follows. At the time the user logs in, he presents a name and password to a login agent that runs in a trusted mode on the user's machine. The user can now create sessions with the various servers that he or she accesses. For example, to communicate with an AFS server, the user requests that the authentication server create a new unique session key and send it back in two forms, one for use by the user's machine, and one for use by the file server.

The authentication server, which has a copy of the user's password and also the secret key of the server itself, creates a new DES session key and encrypts it using the user's password. A copy of the session key encrypted with the server's secret key is also included. The resulting information is sent back to the user, where it is decrypted.

The user now sends a message to the remote server asking it to open a session. The server can easily validate that the session key is legitimate, since it has been encrypted with its own secret key, which could only have been done by the authentication server. The session key also contains trustworthy information concerning the user id, workstation id, and the expiration time of the key itself. Thus, the server knows with certainty who is using it, where they are working, and how long the session can remain open without a refreshed session key.

It can be seen that there is a risk associated with the method described above, which is that it uses the user's password as an encryption key and hence must keep it in memory for a long period of time. Perhaps the user trusts the login agent, but does not wish to trust the entire runtime environment over long periods. A clever intruder might be able to simply walk up to a temporarily unused workstation and steal the key from it, reusing it later at will.

Accordingly, Kerberos actually works by exchanging the user's password for a type of one-time password that has a limited lifetime and is stored only at a *ticket granting service* with which a session is established as soon as the user logs in. The user sends requests to make new connections to this ticket granting service instead of to the original authentication service during the normal course of work, and it encrypts them not with the user's password, but with this one-time session key. The only threat is now that an intruder might somehow manage to execute commands while the user is logged in (e.g. by sitting down at a machine while the normal user is getting a cup of coffee). This threat is a real one, but minor compared to the others that concern us. Moreover, since all the keys actually stored on the system have limited validity, even if one is stolen, it can only be used briefly before it expires. In particular, if the session key to the ticket granting service expires, the user is required to type in his or her password again, and an intruder would have no way to obtain the password in this model without grabbing it during the initial protocol to create a session with the ticket granting service, or by breaking into the authentication server itself.

Once a session exists, communication to and from the file server can be done "in the clear", in which case the file server can use the user id information established during the connection setup to authenticate file access, or can be signed, giving a somewhat stronger guarantee that the channel protocol has not been compromised in some way, or even encrypted, in which case data exchanged is only accessible by the user and the server. In practice, the initial channel authentication, which also provides strong authentication guarantees for the user id and group id information that will be employed in restricting file access, suffices for most purposes. An overview of the protocol is seen in Figure 19-1.

The Kerberos protocol has been proved secure against most forms of attack [LABW92]; one of the few dependencies being its trust in the system time servers, which are used to detect expiration of session keys [BM90]. Moreover, the technology has been shown to scale to large installations using an approach whereby authentication servers for multiple protection domains can be linked to create session keys spanning wide areas. Perhaps the most serious exposure of the technology is that associated with partitioned operation. If a portion of the network is cut off from the authentication server for its part of the network, Kerberos session keys will begin expire and yet it will be impossible to refresh them with new keys. Gradually, such a component of the network will lose the ability to operate, even between applications and servers that reside entirely within the partitioned component. In future applications that require support for mobility, with links forming and being cut very dynamically, the Kerberos design would require additional thought.

A less obvious exposure to the Kerberos approach is that associated with active attacks on its authentication and ticket-granting server. The server is a software system that operates on standard computing platforms, and those platforms are often subject to attack over the network. For example, a knowledgeable user might be able to concoct poison pill, by building a message that will look sufficiently legitimate to be passed to some standard service on the node, but will then provoke the node into crashing by exploiting some known intolerance to incorrect input. The fragility of contemporary systems to this sort of attack is well known to protocol developers, many of whom have the experience of repeatedly crashing the machines with which they work during the debugging stages of a development effort. Thus, one could imagine an attack on Kerberos or a similar system aimed not at breaking through its security architecture, but rather at repeatedly crashing the authentication server, with the effect of denying service to legitimate users.

Kerberos supports the ability to prefabricate and cache session keys (tickets) for current users, and this mechanism would offer a period of respite to a system subjected to a denial of service attach. However, after a sufficient period of time, such an attack would effectively shut down the system.

Within military circles, there is an old story (perhaps not true) about an admiral who used a new generation of information-based battle management system in a training exercise. Unfortunately, the story goes, the system had an absolute requirement that all accesses to sensitive data be logged on an "audit trail", which for that system was printed on a protected lineprinter. At some point during the exercise the line printer jammed or ran low on paper, hence the audit capability shut down. The system, now unable to record the required audit records, therefore denied the admiral access to his databases of troup movements and enemy positions. Moreover, the same problem rippled through the system, preventing all forms of legitimate but sensitive data access.

The developer of a secure system often thinks of his or her task as being to protect critical data from the "bad guys". But any distributed system has a more immediate obligation which is to make data and critical services available to the "good guys". Denial of service in the name of security may be as serious a problem as providing service to an unauthorized user. Indeed, the admiral in the story is now said to have a profound distrust of computing systems. Having no choice but to use computers, in his command the security mechanisms are disabled. (The military phrase is that "he runs all his computers at system high"). This illustrates a fundamental point which is overlooked by most security technologies today: security cannot be treated independent of other aspects of reliability.

19.3.3 ONC security and NFS

SUN Microsystems Inc. has developed an RPC standard around the protocols used to communicate with NFS servers and similar systems, which it calls Open Network Computing (ONC). ONC includes an authentication technology that can protect against most of the spoofing attacks described above. Similar to a Kerberos system, this technology operates by obtaining unforgable authorization information at the time a user logs into a network. The NFS is able to use this information to validate accesses as being from legitimate workstations and to strengthen its access control policies. If desired, the technology can also encrypt data to protect against network intruders who monitor passing messages.

Much like Kerberos, the NFS security technology is considered by many users to have limitations and to be subject to indirect forms of attack. Perhaps the most serious limitations are those associated with export of the technology: companies such as SUN export their products and US government restrictions prevent the export of encryption technologies. As a result, it is impractical for SUN to enable the NFS protection mechanisms by default, and in fact impractical to envision an open standard that would allow complete interoperability between client and server systems from multiple vendors (the major benefit of NFS), while also being secure through this technology. The problem here is the obvious one: not all client and server systems are manufactured in the United States!

Beyond the heterogeneity issue is the problem of management of a security technology in complex settings. Although ONC security works well for NFS systems in fairly simple systems based entirely on SUN products, serious management challenges arise in complex system configurations with users spread over a large physical area, or in systems that use heterogeneous hardware and software sources. With security disabled, these problems vanish. Finally, the same availability issues raised in our discussion of Kerberos pose a potential problem for ONC security. Thus it is perhaps not surprising that these technologies have not been adopted on a widespread basis. Such considerations raise the question of how one might "wrap" a technology such as NFS that was not developed with security in mind, so that security can be superimposed without changing the underlying software. One can also ask about monitoring a system to detect intrusions as an pro-active alternative to hardening a system against intrusions and then betting that the security scheme will in fact provide the desired protection. We discuss these issues in Chapter 23, below.

19.3.4 Fortezza

Fortezza is a recently introduced hardware-based security technology oriented towards users of portable computers and other PC-compatible computing systems [Fort95, Den96]. Fortezza can be understood both as an architecture and as an implementation of that architecture. In this section, we briefly described both perspectives on the technology.

Viewed as an architecture, Fortezza represents a standard way to attach a public-key cryptographic protocol to a computer system. Fortezza consists of a set of software interfaces which standardize the interface to its cryptographic engine, which is itself implemented as a hardware device that plugs into the PCMCIA slot of a standard personal computer. The idea is that a variety of hardware devices might eventually exist that are compatible with this standard. Some, such as a military security technology, might be highly restricted and not suitable for export; others, such as an internally accepted security standard for commercial transactions might be less restricted and safe for export. By designing software systems to use the Fortezza interfaces, the distributed application becomes independent of its security technology and very general. Depending upon the Fortezza card that is actually used in a given setting, the security properties of the resulting system may be strengthened or weakened. When no security is desired at all, the Fortezza functions become no-ops: calls to them take no action and are extremely inexpensive.

Viewed as an implementation, Fortezza is an initial version of a credit-card sized PCMCIA card compatible with the standard, and of the associated software interfaces implementing the architecture. The initial Fortezza cards use the Clipper chip, which implements a cryptographic protocol called Capstone. For example, the interfaces define a function *CI_Encrypt* and a function *CI_Decrypt* that respectively convert a data record provided by the user into and out of its encrypted form. The initial version of the card implements the "Capstone" cryptographic integrated circuit. It stores the private key information needed for each of its possible users, and public keys needed for cryptography. The card performs the digital signature and hash functions needed to sign messages, provides public and private key functions, and supports block data encryption and decryption at high speeds. Other cards could be produced that would implement other encryption technologies using the same interfaces, but different methods.

Although we will not discuss this point in the present text, readers should be aware that Fortezza supports what is called *key escrow* [Den96], meaning that the underlying technology permits a third party to assemble the private key of a Fortezza user from information stored at one or more trusted locations (two, in the specific case of the Capstone protocol). Key escrow is controversial because of public concerns about the degree to which the law enforcement authorities who maintain these locations can themselves be trusted, and about the security of the escrow databases. On the one hand, it can be argued that in the absense of such an escrow mechanism, it will be easy for criminals to exploit secure communications for illegal purposes such as money laundering and drug transactions. Key escrow permits law enforcement organizations to wiretap such communication. But on the other side of the coin, one can argue that the freedom of speech should extend to the freedom to encrypt data for privacy. The issue is an active topic of public debate.

Described coarsely, many authentication schemes are secure either because of something the user "knows", which is used to establish authorization, or something the user "has". Fortezza is designed to have both properties: each user is expected to remember a personal identification code (PIN), and the card cannot be used unless the PIN has been entered reasonably recently. At the same time, the card itself is required to perform secure functions, and stores the user's private keys in a trustworthy manner. When a user correctly enters his or her PIN, Fortezza behaves according to a standard public key encryption scheme, as described earlier. (As an aside, it should be noted that the Clipper-based Fortezza PCMCIA card does not implement this PIN functionality).

To authenticate a message as coming from user A, such a scheme requires a way to determine the public key associated with user A. For this purpose, Fortezza uses a secured X.500-compatible directory, in which user identifications are saved with what are called "certificates". A certificate consists of: a version number, a serial number, the issuer's signature algorithm, the issuer's distinguished name validity period (after which the name is considered to have expired), the subject's distinguished name, the subject's public key, and the issuer's signature for the certificate as a whole. The "issuer" of a certificate will typically be an X.500 server administered by a trusted agency or entity on behalf of the Fortezza authentication domain.

In a typical use, Fortezza is designed with built-in knowledge of the public keys associated with the trusted directory services that are appropriate for use in a given domain. A standard protocol is supported by which these keys can be refreshed prior to the expiration of the "distinguished name" on behalf of which they were issued. In this manner, the card itself knows whether or not it can trust a given X.500 directory agent, because the certificates issued by that agent are either correctly and hence securely signed, or are not and hence are invalid. Thus, although an intruder could potentially mascarade as an X.500 directory server, without the private key information of the server it will be impossible to issue valid certficates and hence to forge public key information. Short of breaking the cryptographic system itself, the intruder's only option is to seek to deny service by somehow preventing the Fortezza user from obtaining needed public keys. If successful, such an attack could in principle last long enough for the "names" involved to expire, at which point the card must be reprogrammed or replaced. However, secured information will never be revealed even if the system is attacked in this manner, and incorrect authentication will never occur.

Although Fortezza is designed as a PCMCIA card, the same technology could be implemented in a true credit card with a microprocessor embedded into it. Such a system would then be a very suitable basis for commercial transactions over the Internet. The primary risk would be one in which the computer itself becomes compromised and takes advantage of the user's card and PIN during the period when both are present and valid to perform undesired actions on behalf of that user. Such a risk is essentially unavoidable, however, in any system that uses software as an intermediary between the human user and the services that he or she requests. With Fortezza or a similar technology, the period of vulnerability is kept to a minimum: it holds only for as long as the card is in the machine, the PIN entered, and the associated timeout has not yet occured. Although this still represents an exposure, it is difficult to see how the risk could be further reduced.

19.4 Availability and Security

Recent research on the introduction of availability into Kerberos-like architectures has revealed considerable potential for overcoming the availability limitations of the basic Kerberos approach. As we saw above, Kerberos is dependent upon the availability of its authentication server for the generation of new protection keys. Should the server fail or become partitioned away from the applications that depend up it, the establishment of new channels and the renewal of keys for old channels will cease to be possible, eventually shutting down the system.

In a doctoral dissertation based on an early version of the Horus system, Reiter showed that process groups could be used to build highly available authentication servers [RBG92, RBR95, Rei93, Rei94a, Rei94b]. His work included a secure join protocol for adding new processes to such a group, methods for securely replicating data and for securing the ordering properties of a group communication primitive (including the causal property), and an analysis of availability issues that arise in key distribution when such a server is employed. Interestingly, Reiter's approach does not require that the time service used in a system like Kerberos be replicated: his techniques have a very weak dependency on time.

Process group technologies permit Reiter to propose a number of exotic new security options as well. Still working with Horus, he explored the use of "split secret" mechanisms to ensure that in a group of *n* processes [HT87, Des88, Fra89, LH91, DFY92, FD92], the availability of any *n-k* members would suffice to maintain secure and available access to that group. In this work, Reiter uses a state machine approach: the individual members have identical states and respond to incoming requests in identical manner. Accordingly, his focus was on implementing state machines in environments with intruders, and on signing reponses in such a way that *n-k* signatures by members would be recognizable as a "group signature" carrying the authority of the group as a whole.

A related approach can be developed in which the servers split a secret in such a manner that none of the servers in the group has access to the full data, and yet clients can reconstruct the data provided that *n-k* or more of the servers are correct. Such a split secret scheme might be useful if the group needs to maintain a secret that none of its individual members can be trusted to manage appropriately.

Techniques such as these can be carried in many directions. Reiter, after leaving the Horus project, started work on a system called Rampart at AT&T [Rei96]. Rampart provides secure group functionality under assumptions of Byzantine failures, and would be used to build extremely secure group-based mechanisms for use by less stringently secured applications in a more general setting. For example, Rampart could be the basis of an authentication service, a service used to maintain billing information in a shared environment, a digital cash technology, or a strongly secured firewall technology.

Cooper, also working with Horus, has explored the use of process groups as a "blinding mechanism." The concept here originated with work by Chaum, who showed how privacy can be enforced in distributed systems by mixing information from many sources in a manner that prevents an intruder from matching an individual data item to its source or tracing a data item from source to destination [Cha81]. Cooper's work shows how a replicated service can actually mix up the contents of messages from multiple sources to create a private and secure email repository [Coo94]. In his approach, the process-group based mail repository service stores mail on behalf of many users. A protocol is given for placing mail into the service, retrieving mail from it, and for dealing with "vacations"; the scheme offers privacy (intruders cannot determine sources and destinations of messages) and security (intruders cannot see the contents of messages) under a variety of attacks, and can also be made fault-tolerant through replication.

Intended for large-scale mobile applications, Cooper's work would permit exchanging messages between processes in a large office complex or a city without revealing the physical location of the principals. Such services might be popular among celebrities who need to arrange romantic liaisons using portable computing and telephone devices; today, this type of communication is notoriously insecure. More seriously, the emergence of digital commerce may exposure technology users to very serious intrusions on their privacy and finances. Work such as Reiter's, Chaum's and Cooper's suggests that security and privacy should be possible even with the levels of availability that will be needed when initiating commercial transactions from mobile devices.

19.5 Related Readings

On Kerberos: [SNS88, Sch94]. Associated theory [LABW92, BM90]. RSA and DES: [DH79, RSA78, DES88, Den84]. Fortezza: most information is online, but [Den96] includes a brief review. Rampart: [RBG92, RBR95, Rei93, Rei94a, Rei94b]. Split-key cryptographic techniques and associated theory: [HT87, Des88, Fra89, LH91, DFY92, FD92]. Mixing techniques [Cha81, Coo94, CB95].

20. Clock Synchronization and Synchronous Systems

Previous sections of this book have made a number of uses of clocks or time in distributed protocols. In this chapter, we look more closely at the underlying issues. Our focus is on aspects of "real-time computing" that are specific to distributed protocols and systems, since a full treatment of the topic would merit a book of its own.

20.1 Clock Synchronization

Clock synchronization is an example of a topic that until the recent past represented an important area for distributed systems research [LAM84, LM85, Mar84, LM85, KO87, ST87, Cri89, CF94, VR92, CM96], overviews of the field can be found in [SWL90] and [Lis93]. The introduction of the global positioning system, in the early 1990's, greatly changed the situation. As recently as five years ago, a textbook such as this would have treated the problem in considerable detail, to the benefit of the reader because the topic is an elegant one and the clock-based protocols that have been proposed are interesting to read and analyze. Today, however, it seems more appropriate to touch only briefly on the subject.

The general problem of clock synchronization arises because the computers in a distributed system typically use internal clocks as their primary time source. On most systems, these clocks are accurate to within a few seconds per day, but there can be surprising exceptions to the rule. PC's, for example, may operate in "power-saving" modes in which even the clock is slowed down or stopped, making it impossible for the system to gauge real-time reliably. At the other end of the spectrum, the global positioning system (GPS) has introduced an inexpensive way to obtain accurate timing information using a radio receiver; time obtained in this manner is accurate to within a few milliseconds unless the GPS signal itself is distorted by unusual atmospheric conditions or problems with the antenna used to receive the signal.

MARS: A Distributed Systems for Real-Time Control

The MARS system uses clock synchronization as the basis of an efficient fault-tolerance method, implemented using pairs of processing components interconnected by redundant communication links. The basic approach is as follows [DRSK89, KO87, KV93].

A very high quality of clock synchronization is achieved using a synchronization method that resides close to the hardware (a broadcast-style bus). Implemented in part using a special purpose device controller, clocks can be synchronized to well under a millisecond, and if a source of accurate timing information is available, can be both precise and accurate to within this degree of precision.

Applications of MARS consist of directly controlled hardware, such as robotic units or components of a vehicle. Each processor is duplicated as is the program that runs on it, and each action is taken redundantly. Normally, every message will be sent four times: once by each processor on each message bus. The architecture is completely deterministic in the sense that all processes see the same events in the same order and base actions on synchronized temporal information in such a way that even clock readings will be identical when identical tasks are performed. Software tools for scheduling periodic actions and for performing actions after a timer expires are provided by the MARS operating system, which is a very simple execution environment concerns primarily with scheduling and message passing.

MARS is designed for very simple control programs, and assumes that these programs fail by halting (the programs are expected to self-check their actions for "sanity" and shut down if an error is detected). In the event that a component does fail, this can be detected by the absense of messages from it, or their late arrival. Such a failed component is taken off line for replacement, and reintegrated back into the system the next time it is restarted from scratch. These assumptions are typical of in-flight systems for aircraft and of factory-floor process control systems.

Although MARS is not a particularly elaborate or general technology, it is extremely effective within its domain of "intended use." The assumptions made are felt to be reasonable ones for this class of applications, and although there are limitations on the classes of failures that MARS can tolerate, the system is also remarkably simple and modular, benefiting precisely from those limitations and assumptions. The performance of the system is extremely good for the same reasons.

Traditionally, clock synchronization was treated in the context of a group of peers, each possessing an equivalent local clock, with known accuracy and drift properties. The goal in such a system was typically to design an agreement protocol by which the clocks could be kept as close as possible to "real-time" and with which the tendency of individual clocks to drift (from one another and to do so with respect to real-time) could be controlled. To accomplish this, processes in such a system would periodically exchange time readings, running a protocol by which a software clock could be constructed having substantially better properties than that of any of the individual participating programs, and with the potential to overcome outright failures whereby a clock might drift at an excessive rate or return completely erroneous values.

Key parameters to such a protocol are the expected and maximum communication latencies of the system. It can be shown that these values limit the quality of clock synchronization achievable in a system by introducing uncertainty in the values exchanged between processes. For example, if the latency of the communication system between p and q is known to vary in the range $[0,\varepsilon]$, any clock reading that p sends to q will potentially be aged by ε time units by the time q receives it. When latency is also bounded below, a method developed by Verissimo (briefly presented below) can achieve clock precisions bounded by the *variation* in latency. In light of the high speed of modern communication systems, these limits represent a remarkably high degree of synchronization: it is rarely necessary to time events to within accuracy's of a millisecond or less, and these limits tell us that it should be possible to synchronize clocks to that degree if desired.

Accuracy is a characterization of the degree to which a correct clock can differ from an external clock that gives the "true" time. A clock synchronization protocol that guarantees highly accurate clocks thus provides the assurance that a correct clock will return a value within some known maximum error of the value that the external clock would return. In some settings, accuracy is expressed as an absolute bound; in others, accuracy is expressed as a maximum rate of drift: in this latter case, the accuracy of the clock at a given time is a function of how long the clock has been free-running since the last round of the synchronization protocol.

Skew is a measure of the difference between clock readings for a pair of processes whose clocks have been sampled at the same instant in real-time.

Precision is a characterization of the degree to which any pair of correct clocks can differ over time. Like accuracy, a precision may be given as a constant upper bound on the skew, or as a maximum rate of drift of the skews for pairs of correct clocks.

Figure 20-2: Definitions of accuracy, skew, and precision for synchronized clocks in distributed settings.

Modern computing systems face a form of clock synchronization problem which is easier to solve than the most general version of the problem. If such systems make use of time at all, it is common to introduce two or more GPS receivers, in this manner creating some number of system time sources. Devices consisting of nothing more than a GPS receiver and a network interface can, for example, be placed directly on a shared communication bus. The machines sharing that bus will now receive time packets at some frequency, observing identical values at nearly identical time.

If the device driver associated with the network device is able to identify these incoming time packets, it can be used to set the local clock of the host machine to extremely high precision; even if not, an application should be able to do so with reasonable accuracy. Given data for the average access and propagation delays for packets sent over the communications hardware, the associated latency can be added to the incoming time value, producing an even more accurate result. In such a manner, systems in which real-time is important can synchronize processor clocks to within milliseconds, obviating the need for any more sophisticated application-level synchronization algorithm. After all, the delays associated with passing a message through an operating system up to the application, with scheduling the application process if it was in a blocked state, and with paging in the event of a possible page fault, are all large compared with the clock accuracy achievable in this manner. Moreover, it is very unlikely that a GPS time source would fail other than by crashing. Were non-crash failures a concern, a simple solution would be to collect sets of readings from three GPS sources, exclude the outlying values, and take the remaining value as the correct one.

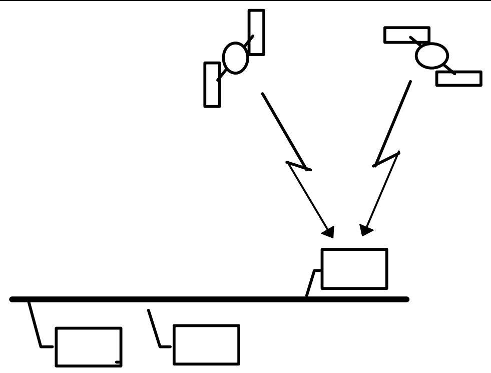


Figure 20-1: The global positioning system is a satellite network that broadcasts highly accurate time values worldwide. Although intended for accurate position location, GPS systems are also making accurate real-time information available at low cost.

In light of this development, it has become desirable to consider distributed computing systems as falling into two classes. Systems in which time is important for reliability can readily include accurate time sources and should do so. Systems in which time is not important for reliability should be designed to avoid all use of workstation clock values, using elapsed time on a local clock to trigger timer based events such as retransmission of messages or timeout, but not exchanging time values between processes or making "spurious" use of time. For the purposes of such elapsed timers, the clocks on typical processors are more than adequate: a clock that is accurate to a few seconds per day will measure a 100ms timeout with impressive accuracy.

Where clocks are known to drift, Verissimo and Rodriguez have suggested an elegant method for maintaining very precise clocks [VR92, *see also* CM96]. This protocol, called *a-posteriori clock synchronization*, operates roughly as follows. A process other than the GPS receiver initiates clock synchronization periodically (for fault-tolerance, two or more processes can run the algorithm concurrently). Upon deciding to synchronize clocks, this process sends out a *resynchronize* message, including its own clock value in the message, and setting this value as close as possible to when the message is transmitted on the wire. For example, the device driver can set the clock field in the header of an outgoing message just before setting up the DMA transfer to the network.

Upon arrival in destination machines, each recipient notes its local clock value, again doing this as close as possible to the wire. The recipients send back messages containing their clock values at the time of the receipt. The difference between these measured clock values and that of the initiator will be latency from the initiator to the receivers plus the drift of the recipient's clock relative to the clock of the initiator. Thus, if the initiator believes it to be three o'clock and the latency of communication is 1ms, the value -31ms would correspond to a recipient whose clock was showing 32ms before three o'clock when the initiator sent the message, while the value 121ms would be returned by a process whose clock read three o'clock plus 120ms at the time the initiator sent the message. Variations in latency will cause these values to be slightly higher or lower than in this example: perhaps -31.050ms or 120.980ms.

The synchronization algorithm now selects one of the participant's as the "official clock" of the system. It does so either by selecting a value returned from a process with a GPS receiver, if one is included, or by sorting the returned differences and selecting the median. It subtracts this value from the other differences. The vector will now have small numbers in it if, as assumed, the latency from initiator to participants is fairly constant over the set. The values in the vector will represent the distance that the corresponding participant's clock has drifted with respect to the reference clock. Given an estimate of the message latency between the reference process and the initiator, the initiator can also compute the drift of its own clock. For example, a process may learn that its clock has drifted by -32ms since the last synchronization event. Any sort of reliable multicast protocol can be used to return the correction factors to the participants.

To actually correct a clock that has drifted, it is common to use an idea introduced by Srikanth and Toueg. The approach involves gradually compensating for the drift under the assumption that the rate of drift is constant. Thus, if a process has drifted 120ms over a 1 minute period, the clock might be modified in software to introduce a compensating drift rate of -240ms over the next minute, in this manner correcting both the original 120ms and overcoming the continuing 120ms drift of its clock during the period. Such an adjustment occurs gradually, avoiding noticeable jumps in the clock value that might confuse an application program.

The above discussion has oversimplified the protocol: the method is actually more complicated because it needs to account for a variety of possible failure modes; this is done by running several rounds of the protocol and selecting, from among the "candidate clocks" that appears best in each round, that round and clock for which the overall expected precision and accuracy is likely to be best.

Verissimo and Rodriguez's algorithm is optimally precise but not necessarily the best for obtaining optimal accuracy: the best known solution to that problem is the protocol of Srikanth and Toueg mentioned above. However, when a GPS receiver is present in a distributed system that has a standard broadcast-style LAN architecture, the a-posteriori method will be optimal in both respects: accuracy and precision, with clock accuracies that are comparable in magnitude to the variation in message latencies from initiator to recipients. These variations can be extremely small: numbers in the tens of microseconds are typical. Thus, in a world-wide environment with GPS receivers one can imagine an inexpensive software and hardware combination that permits processes, anywhere in the world, to measure time accurately to a few tens of microseconds. Accuracies such as this are adequate for even very demanding real-time uses.

Unfortunately, neither of these methods is actually employed by typical commercial computing systems. At the time of this writing, the situation is best characterized as a transitional one. There are well known and relatively standard software clock synchronization solutions available for most networks, but the standards rarely span multiple vendor systems. Heterogeneous networks are thus likely to exhibit considerable time drift from processor to processor. On the other hand, most workstations have time sources available that can be trusted in a crude way, at a resolution of seconds or tens of seconds. This is fortunate, because the use of time is growing in applications such as security systems, where session keys and other generated secrets are normally viewed as having limited lifetimes. Thus, security systems may depend upon their time sources for correct behavior. Meanwhile, vendors seem to be closer and closer to including GPS recievers as a standard component of network servers, making cheap and accurate time more and more standard. The a-posteriori protocol, if used widely, could result in a situation where all computers world-wide would share clocks synchronized to within a few hundred microseconds, a development that would facilitate major advances in communication support for video-on-demand and group conferencing systems.

20.2 Timed-asynchronous Protocols

Given a network of computers that share an accurate time source, it is possible to design broadcast protocols that guarantee real-time properties as well as other properties, such as failure atomicity or totally ordered delivery. The best known work in this area is that of Cristian, Arghili, Strong and Dolev and is widely cited as the CASD protocol suite or the Δ-T atomic broadcast protocols [CASD85, CSDA90]. These protocols are designed for a static membership model, although Cristian later extended the network model to dynamically track the formation and merging of components in the event of network partitioning failures, again with real-time guarantees on the resulting protocols. In the remainder of this section, we present these protocols in the simple case where processes fail only by crashing or by having clocks that lie outside of the acceptable range for correct clocks, and where messages are lost but not corrupted. The protocols have often been called "synchronous" ones in the literature, but Cristian currently favors the term *timed asynchronous* [CS95], and this is the one we use here.

The CASD protocols seek to guarantee that during a period of time during which some set of processes are *continuously operational* and *connected*, they will deliver the same messages at the same time and in the same order. There is a subtlety here, to which we will return below: a process may not be able to detect that it has been non-operational for a period of time, and hence that it may not be guaranteed to see correct behavior from the protocols. But we start by considering the simple scenario of a network consisting of a collection of *n* processes, *k* of which may be faulty. Moreover, "same time" must be understood to be limited by the clock skew: because processor clocks may differ by as much as ε, two correct processors that undertake to perform the same action at the same time may in fact do so as much as ε time-units apart.

The CASD protocol is designed for a network in which packets must be routed; the network diameter, *d,* is the maximum number of hops that a packet may have to take to reach a destination node from a source node. It is assumed that failures will not cause the network to become disconnected. Although individual packets can be lost in the network, it is assumed that there is a known limit on the number of packets that will actually be lost in any single run of the protocol. Finally, multicast networks are not modeled as such: an ethernet or FDDI is treated as a set of point to point links.

The CASD protocol operates as follows. A process (which may itself be faulty) creates a message and labels it with a timestamp t (from its local clock) and with its process identifier. It then forwards the message to all processors reachable over communication links directly connected to it. These processes accept incoming messages. A message is *discarded* if it is a duplicate of a message that has been seen previously or if the timestamp on the message falls outside a range of currently feasible valid timestamps. Otherwise, the incoming message is *relayed* over all communication links except the one on which it was received. This results in the exchange of $O(n^2)$ messages, as illustrated in Figure 20-3.

A process holding a message waits until time $t+\Delta$ on its local clock (here, t is the time when the message was sent), and then delivers it in the order determined by the sender's timestamp, breaking ties using the processor id of the sender. For suitable validity limits and Δ, this protocol can be shown to overcome crash failures, limited numbers of communication failures, incorrect clock values on the part of the sender or intermediary relay processes.

The calculation of this parameter is based on the following reasoning: for the range of behaviors possible in the system, there corresponds a maximum latency after which a message that originates at a faulty process and has been forwarded only by faulty processes finally reaches a correct process and is accepted as valid. From this point forward, there is an additional maximum latency before the message has reached all correct processes, limited by the maximum number of network packet losses that can occur. Finally, any specific recipient may consider itself to be the "earliest" of the correct processes to have received the message, and will assume that other correct processes will be the "last" to receive a copy. From this reasoning, a value can be asssigned to Δ such that at time $t+\Delta$, every correct process will have a copy of the message and will know that all other correct processes also have a copy. It is therefore safe to deliver the message at time $t+\Delta$: the other processes will do so as well, within a time skew of ε corresponding to the maximum difference in clock values for any two correct processes. This is illustrated in Figure 20-3, where time $t+b$ corresponds to $t+\Delta-\varepsilon/2$ and $t+c$ to $t+\Delta+\varepsilon/2$.

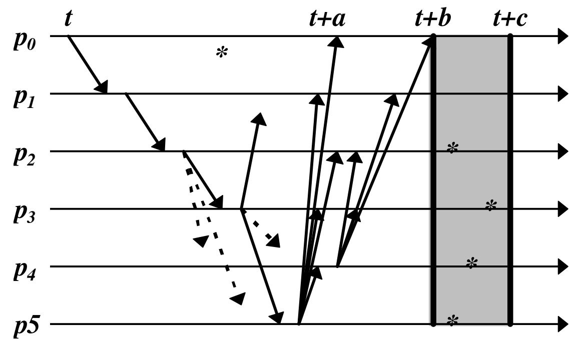


Figure 20-3: In the CASD protocol, messages are delivered with real-time guarantees despite a variety of possible failures. In this example for a fully connected network (d=1), processes p_0 and p_1 are faulty and send the message only to one destination each. p_2 and p_3 are correct, but experience communication failures that prevent the message from being forwarded to the full set of correct processors. Eventually, however, the full set of possible failures has been exhausted and the message reaches all correct destinations even if the execution is a worst-case one. In this example, the message finally reaches its last destination at time t+a. The processors now delay delivery of the message under a best case/worst case analysis whereby each process reasons that it may have received the message in the minimum possible time but that others may receive it after the maximum possible time, and yet assume that they too had received the message after a minimal delay. When this delay has elapsed, all correct processes know that all other correct processes have the message and are prepared to deliver it; delivery then takes place during a period bounded above and below by the clock synchronization constant ε (shown as [t+b,t+c] in the figure). Incorrect processes may fail to deliver the message, as in the case of p_1, may deliver outside of the window, as does p_0, or may deliver messages rejected by all correct processes.

Although we will not develop the actual formulas here, because the analysis would be fairly long, it is not hard to develop a basic intuition into the reasoning behind this protocol. If we are safe in assuming that there are at most *f* faulty process in the network, and that the network itself loses no more than *k* packets during a run of the protocol, it must follow that a broadcast will be reach at least one operational processes which will forward it successfully to every other operational process within *f*+*k* rounds. A process using the protocol simply waits long enough to be able to deduce that every other process must have a copy of the message, after which it delivers the message in timestamp order.

Because all the operational processes will have received the same messages and use the same timestamp values when ordering them for delivery, the delivered messages are the same and in the same order at all correct processes. However, this may not be the case at *incorrect* processes, namely those for which the various temporal limits and constants of the analysis do not hold, or those that failed to send or receive messages that the protocol requires them to send or receive.

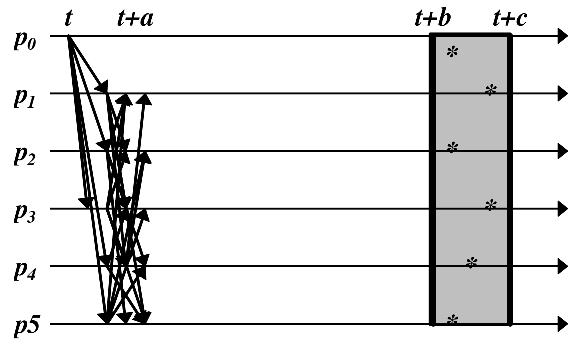


Figure 20-4: A run of the CASD protocol in which no failures occur. After a flurry of message exchanges during which $O(n^2)$ messages are sent and received, the protocol lies quiescent until delivery occurs. The delay to delivery is unaffected by the "good fortune" of the protocol in having reached all the participants so rapidly. Notice that as normally presented, the protocol makes no use of broadcast hardware.

Clearly, when a protocol such as this one is used in a practical setting, it will be advantageous to reduce the value of Δ as much as possible, since Δ is essentially a minimum latency for the protocol. For this reason, the CASD protocol is usually considered in a broadcast network for which the network diameter, *d,* is 1, processes and communication are assumed to be quite reliable (hence, these failure limits are reduced to numbers like 1), and clocks are assumed to be very closely synchronized for the operational processes in the network. With these sorts of assumptions, Δ, which would have a value of about 3 seconds in the local area network used by the Computer Science Department at Cornell, can be reduced into the range of 100 to 150ms. Such a "squeezing" of the protocol leads to runs like the one seen in Figure 20-5.

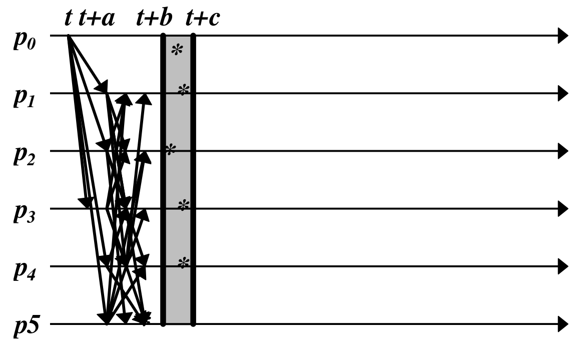


Figure 20-5: More aggressive parameter settings and assumptions can substantially reduce the delay before delivery occurs.

We noted that there is a subtle issue associated with the definition of "operational" in the goals of the CASD protocol. The problem is arises when we consider a process that is technically faulty because its clock has drifted outside the limits assumed for a correct process; with the clock synchronization methods reviewed above, this is an unavoidable risk, which grows as the assumed limits become tighter (this is also true when using Cristian's recommended clock synchronization protocol [Cri89]). That is, the same actions that we took to reduce Δ also have the side-effect of making it more likely that a process will be considered faulty.

Such a process is only faulty in a technical sense. Viewed from "above", we can see that its clock is slightly too fast or too slow, perhaps only 5 or 10 milliseconds from the admissible range. Internally, the process considers itself quite operational, and would be unable to detect this type of "fault" even if it tries to do so. Yet, because it is faulty in the formal sense of violating our conditions on correct processes, the guarantees of the protocol may no longer hold for such a process: it may deliver messages that no other process delivered, or fail to deliver messages that every other process delivered successfully, or deliver messages outside of the normal time range within which delivery should have occurred. Even worse, the process may then drift back into the range considered normal and hence recover to an operational state immediately after this condition arises. The outcome might be a run more like the one in Figure 20-6.

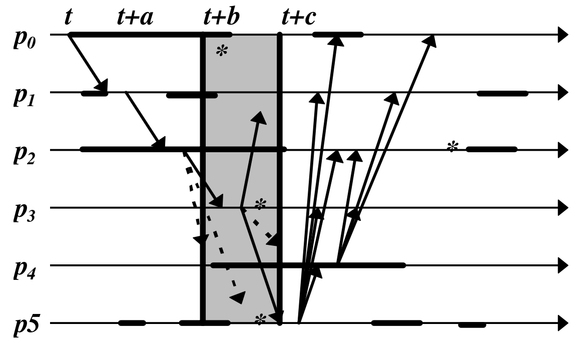


Figure 20-6: In this case, overly aggressive parameter settings have caused many processes to be "incorrect" in the eyes of the protocol, illustrated by bold intervals on the process time lines (each process is considered "incorrect" during a bold interval, for example because its clock has drifted too far from the global mean). The real-time and atomicity properties are considerably weakened; moreover, participating processes have no way to determine if they were "correct" or "incorrect" on a given run of the protocol. Here, the blue messages are considered as valid by the protocol; the red ones arrive "too late" and are ignored by correct processes.

Thus, although the CASD protocol offers strong temporal and fault-tolerance properties to correct processes, the guarantees of these protocols may appear weaker to a process that uses them, because such a process has no way to know, or to learn, whether or not it is one of the correct ones. In some sense, the protocol has a notion of system membership built into it, but this information is not available to the processes in the system themselves. The effect is to relax all of the properties of the protocol suite, which is perhaps best understood as being probabilistically reliable for this reason.

A stronger statement could be made if failures were detectable so that such a process could later learn that its state was potentially inconsistent with that of other processes. There has been some encouraging work on strengthening the properties of this protocol by layering additional mechanisms over it. Gopal *et. al.* for example, have shown how the CASD protocols can be extended to guarantee causal ordering and to overcome some forms of inconsistency [GSTC90]. In Chapter 22 we will see how a CASD-like protocol can be made to offer stronger guarantees under a slightly different system model, in which probabilistic information is available in regard to the properties of communication channels.

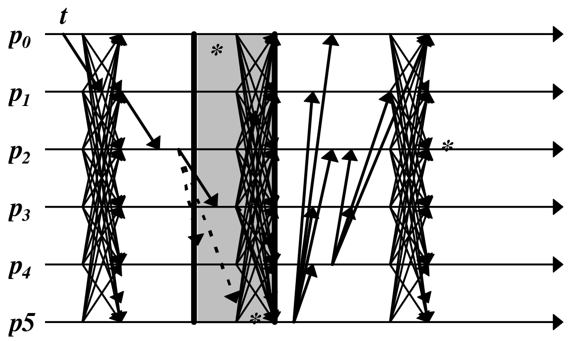


Figure 20-7: In the NavTech protocol suite developed by Almeida and Verissimo, periodic background exchanges of state (dark intervals) cut through the normal message traffic, permitting such optimizations as early message delivery and offering information for use in overcoming inconsistency. However, short of running a group membership protocol in the background communication channel, there are limits to the forms of inconsistency that this method can actually detect and correct.

Within the Portuguese NavTech project, Almeida and Verissimo have explored a class of protocols that superimpose a background "state exchange" mechanism on a CASD-like protocol structure. In this approach, processes within the system periodically send snapshots of aspects of their state to one-another use unreliable all-to-all message exchanges over dedicated but low bandwidth links. The resulting n^2 message exchange leaves the correct processes with accurate information about one-another's states prior to the last message exchange, and partially accurate information as of the current exchange (the limitation is due to the possibility that messages may be lost by the communication subsystem). In particular, the sender of a CASD-style broadcast may now learn that it has reached all its destinations. During the subsequent exchange of messages, information gained in the previous one can be exploited, for example to initiate an early delivery of a timed broadcast protocol. Unfortunately, however, the mechanism does not offer an obvious way to assist the correct processes in maintaining mutually consistent knowledge concerning which processes are correct and which are not: to accomplish that goal, one would need to go further by implementing a process group membership service "superimposed" on the real-time processes in the system. This limitation is apparent when one looks at possible uses for information that can be gathered though such a message exchange: it can be used to adjust protocol parameters in limited ways, but generally cannot be used to solve problems in which the correct processes must have mutually consistent views of shared parameters or other forms of replicated state.

It would be interesting to explore an architecture in which real-time protocols are knowingly superimposed on virtually synchronous process groups, using a high priority background channel such as the one introduced in Almeida's work to support the virtually synchronous group. With such a hybrid approach, it would be possible to exclude faulty processes from a system within a known delay after the fault occurs, adjust protocol parameters such as the delay to delivery by correct processes, so that the system will adaptively seek out the best possible delay for a given configuration, or combine the use of coherently replicated data and state with real-time updates to other forms of data and state. An approach that uses reserved-capacity high priority channels, such as the ones introduced by Almeida, could be used to support such a solution. At the time of this writing, however, the author is not aware of any project that has implemented such an architecture.

This brings us back to the normal implementation of the CASD protocol suite. The user of such a protocol must expect that the distributed system as a whole may contain processes that are contaminated by having updated their states on the basis of messages that were handled differently than at the correct processes. Such processes are not in any way prevented from initiating new messages that will be received by all processes, and which will presumably reflect this inconsistent state in direct or subtle ways. Indeed, over time, almost any process may be viewed as incorrect for one or another run of the protocol, hence such contamination is likely to be pervasive and capable of spreading. Mechanisms for ensuring that such a system will converge back into a mutually consistent state should a divergence of states occur are needed when these protocols are used. Alternatively, one can restrict the use of the protocols to forms of information that need not be absolutely correct or use them only as input to algorithms that are tolerant of a certain level of "noise" in their inputs. One should not, for example, use them as the basis for a coherently replicated cache or as the basis of a safety critical decision that must be made consistently at multiple locations in a system.

The CASD protocols represent an interesting contrast with the virtual synchrony protocols we discussed earlier in this text. Those protocols tolerate similar types of failures, but lack any notion of time and offer no temporal delivery guarantees. On the other hand, they do offer strong logical guarantees: the "consistency" properties that we stressed at the time we discussed them. CASD, as we have now seen, lacks this notion of consistency, but has a very strong temporal guarantee when used by processes that are operational within its model. CASD is weakened by the sorts of optimizations that improve its temporal responsiveness, but would be very unlikely to "misbehave" if a large value of Δ were considered acceptable. Thus, we have what appears to be a basic tradeoff between logical guarantees and temporal ones. It is intriguing to speculate that such tradeoffs may be fundamental ones.

The tradeoff is also noticeable in the delay of the protocol. For large values of Δ the CASD protocol provides very strong guarantees, but also has a very large latency to delivery. This is the converse of the situation for the virtually synchronous *fbcast* or *cbcast* protocol, which has a very low latency to delivery in the usual case and very strong *logical* guarantees, but no meaningful real-time guarantees. Indeed, the installation of new views of a process group can delay a *cbcast* for a period of time that grows with the size of the group, and that also will be affected by the level of communication traffic in the group at the time of the view installation. Thus, *cbcast* can be understood as rushing to deliver its messages and often doing so in far less time and with far fewer messages than a protocol such as CASD. However, if correct temporal behavior by the correct processes is critical to the application, *cbcast* is not able to offer this guarantee in better than a probabilistic sense (and even a probabilistic argument would involve an analysis of *cbcast* protocols outcomes and their relative likelihood's that we are not prepared to undertake in this textbook).

Stepping back one might characterize the basic difference here as one of pessimism versus optimistism. The *cbcast* style of protocols are generally optimistic in their expectations from the system: they expect that failures will be relatively uncommon events and are optimized for the earliest possible delivery when this case in fact arises. These protocols can give extremely low latency (two or more orders of magnitude better than the CASD style of protocol) and can be extremely predictable in their behavior provided that the network load is light, paging and other delays do not occur, and failures are genuinely infrequent. Indeed, if one could be *certain* that these conditions held, a protocol such as *cbcast* could be the basis of a real-time system, and it would perform perhaps thousands of times better than the timed-asynchronous style of system. But hoping that a condition holds and proving that it holds are two different matters.

The CASD suite of protocols and other work by Cristian's group on the timed asynchronous model can be viewed as relatively pessimistic, in the sense that for a given set of assumptions, these protocols are designed to expect and to overcome a worst-case execution. If CASD is used in a setting where it is known that the number of failures will be low, the protocol can be optimized to benefit from this. As we have seen, however, the protocol will only work to the degree that the assumptions are valid and that most operational processes will be considered as correct. When this ceases to be the case, the CASD protocols break down and will appear to behave incorrectly from the point of view of processes that, in the eyes of the system model, are now considered to dance in and out of the zone of "correct behavior." But the merit of this protocol suite is that if the assumptions are valid ones, the protocols are *guaranteed* to satisfy their real-time properties.

As noted above, Cristian has also worked on group membership in the timed asynchronous model. Researchers in the Delta-4 project in Europe have also proposed integrated models in which temporal guarantees and logical guarantees were integrated into a single protocol suite [Pow91, RV89, RVR93, Ver93, Ver94]. For brevity, however, we will not present these protocols here.

20.3 Adapting Virtual Synchrony for Real-Time Settings

Friedman has developed a real-time protocol suite for Horus that works by trying to improve the expected behavior of the virtual synchrony group protocols rather than by starting with temporal assumptions and deriving provable protocol behaviors as in the case of CASD [FR95b]. Although in a preliminary state, this work has yielded some interesting results. Among these, Friedman has developed a view installation and message delivery architecture for Horus that draws on the Transis idea of distinguishing "safe" from "unsafe" message delivery states [FR95b]. In Freidman's protocols, "safe" states are those for which the virtual synchrony properties hold while "unsafe" ones are states for which real-time guarantees can be offered but in which weaker properties than the usual virtual synchrony properties hold.

One way to understand Friedman's approach is to think of a system in which each message and view is delivered twice (he implements this behavior, however, with a more efficient upcall mechanism). The initial delivery occurs with real-time guarantees of bounded latency from sending to reception, or bounded delay from when an event that will change the group view occurs to when that view is delivered. However, the initial delivery may occur before the "virtually synchronous" one. The second delivery has the virtual synchrony properties and may report a group view that is different from the initial one, albeit in limited ways (specifically, such a view can be smaller than the original one but never larger — processes can fail but not join). The ideas is that the application can now select between virtual synchrony properties and real-time ones, using the real-time delivery event for time-critical tasks and the virtually synchronous event for tasks in which logical consistency of the actions by group members are critical. Notice that a similar behavior could be had by placing a Horus protocol stack running a real-time protocol side by side in the same processes as a Horus protocol stack supporting virtual synchrony, and sending all events through both stacks. Friedman's scheme also guarantees that event orderings in the two stacks will be the same unless the time constraints make this impossible; two side-by-side stacks might differ in their event orderings or other aspects of the execution.

A major goal of the research community is to develop reliable distributed systems for real-time applications. Such a system would have the ability to perform time-critical tasks with guarantees that deadlines will be respected, and that other required levels of performance will be achieved, even if failures of various sorts occur while the system is running.

A distributed real-time system would be built from real-time components: operating systems that support task priorities, preallocation of resources, special-purpose scheduling, and time-driven mechanisms permitting the user to implement real-time applications and to be sure that there is nothing in the operating system that can prevent critical tasks from occuring on time.

Over this, one would layer real-time communications protocols coupled to a failure detection mechanism of known maximum delay. Since messages may need to be buffered while the failure detection mechanism has yet to "kick in", an analysis of worst-case buffering requirements would also be required, so that if a failure does occur, the system doesn't become overloaded due to the accumulation of data being sent to the failed node before reconfiguring itself to exclude that node.

Finally, such a system would need to demonstrate a viable methodology for actually building real-time distributed applications that demonstrably tolerate failures while continuing to preserve response guarantees within the envelope specified by the designer.

A system like Horus, with real-time protocols and synchronized clocks, may be suitable for solving problems such as these if they are not excessively demanding. If such a system is used "far" from its maximum capacity and with very large amounts of memory, and if the real-time limits are relatively weak ones, Horus will predictably operate within the desired bounds. An open problem of considerable interest to the author is to pin down just what those bounds might be, so that we can strengthen such a statement and say that Horus can *guarantee* real-time behavior while also providing fault-tolerance and consistency.

In support of this work, Vogels and Mosse have investigated the addition of real-time scheduling features to Horus, message and thread priorities, and pre-allocation mechanisms whereby resources needed for a computation can be pinned down in advance to avoid risk of delaying if a needed resource is not available during a time-critical task.

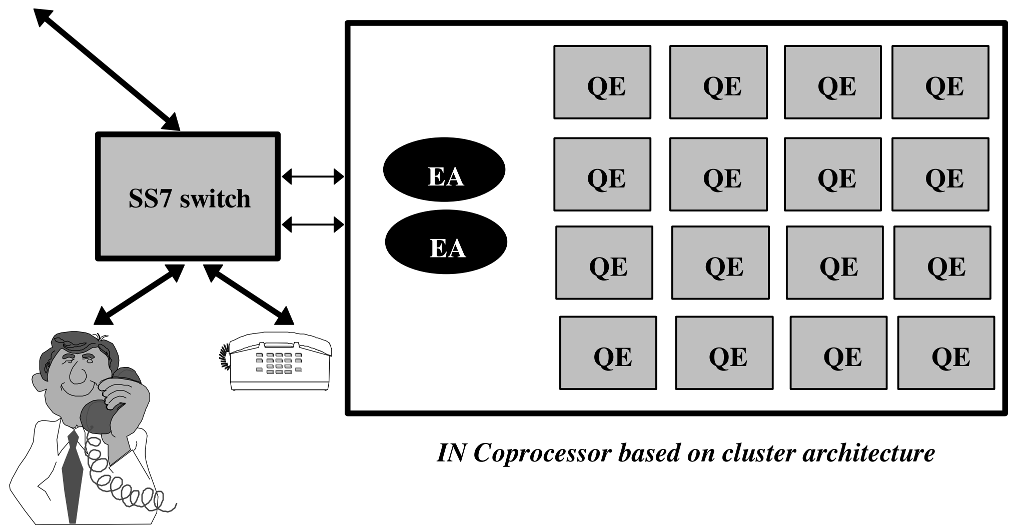


Figure 20-8: Friedman has experimented with the use of a cluster of computing systems in support of a demanding real-time telecommunications application. On the left is a single switch that handles telephone calls in the SS7 switching architecture. Somewhat simplifying the actual setup, we see local telephones connected to the switch from below, and lines connecting to other switches above. SS7-compatible switches can be connected to adjunct processors, called IN coprocessors, which provide intelligent routing functionality and implement advanced services on behalf of the switch itself. For example, if an 800-number call is received, the coprocessor would determine which line to route the call on, and if call forwarding was in use, the coprocessor would reroute forwarded calls. Friedman's architecture uses Horus to support a cluster configuration within the IN coprocessor, an approach that provides very large scalable memory for the query element's (which would typically map a telephone directory into memory), load-balancing, and fault-tolerance.

An early application of this real-time fault-tolerance technology is to the problem of building a telecommunications switch in which a cluster of computers control the actions taken as telephone calls are received (Figure 20-8). Such an application has a very simple architecture: the switch itself (based on the SS7 architecture) sees the incoming call and recognizes the class of telephone numbers as one requiring special treatment, as for the case of an 800 or 900 number in the United States. The switch creates a small descriptive message giving the caller's telephone number, the destination, billing information, and a call identification number, and forwards this to a what is called an *intelligent network coprocessor* or IN coprocessor. The coprocessor (traditionally implemented using a fault-tolerant computer system) is expected to perform a database query based on the telephone numbers and to determine the appropriate routing for the call, responding within a limited amount of time (typically, 100ms). Typically, the switch will need to handle as many as 10,000 to 20,000 calls per second, dropping no more than some small percentage, and doing this randomly even during periods when a failure is being serviced. That is, the switch must never be "down" for more than a few seconds per year, although individual calls may sometimes have a small probability of not going through and needing to be redialed.

The argument in favor of using a cluster of computers for this purpose is that such a system potentially has greater computing power (and much aggregate main memory) than any single processor could have. This may translate to the ability to keep a very large database in memory for rapid access (spread among the nodes), or of executing a more sophisticated query strategy. Moreover, whereas the upgrading of a fault-tolerant coprocessor may require that the switch be shut down, one can potentially upgrade a cluster style computer one node or one program at a time.

Without getting into the details, Friedman has demonstrated that systems like Horus can indeed be used to support such a model. In [FB96], he describes a system that emulates this configuration of telephone switch, servicing 22,000 calls per second while dropping no more than 1-3% even when a failure or recovery is actually being serviced. Friedman's design involves a pair of external adaptor nodes (EA's) which sense incoming calls and dispatch the corresponding query onto pairs of query processing nodes (QE's). Friedman batches requests and uses an innovative real-time fault-tolerance protocol to optimize for the very high processing loads that characterize the application.

To solve this problem, Friedman's work combines the real-time mechanisms cited above with a number of other innovations, and it is fair to say that the application is not a straightforward one. However, the benefits of being to use a cluster-style computing system in this manner could be dramatic: such systems are quite inexpensive, and yet they may bring a great deal of performance and flexibility to the application, which would otherwise be very constrained by the physical limitations typical of any single-processor solution. Friedman's initial work focuses on memory scalability, but he is now extending the approach to seek a performance benefit from scale as well as the possibility of benefit from memory-mapping the database.

Although cast in the context of a telephone switching application, it should be noted that the type of real-time client-server architecture being studied in Friedman's work is much more general. We have seen in earlier sections of this text that the great majority of distributed systems have a client-server architecture, and this is also true for real-time systems, which typically look like client-server systems with time-critical response deadlines superimposed upon an otherwise conventional architecture. Thus, Friedman's work on telephone switching could also be applicable to process control systems, air-traffic control systems, and other demanding applications that combine fault-tolerance and real-time constraints.

Other work in this area includes Marzullo's research on the CORTO system, which includes such features as *periodic process groups*. These are process groups whose members periodically and within a bounded period of real-time initiate synchronized actions. Marzullo has studied minimizing the communication overhead required in support of this periodic model, integrating real-time communication with other periodic or real-time actions, priority inversion in communication environments, and other topics in the area.

20.4 Related Readings

On clock synchronization see the review in [SWL90], other references include [Lam84, Mar84, LM85, KO87, ST87, Cri89]. On the a-posteriori method: [VR92, CM96]. On the CASD protocol: [CASD85, CSDA90, CS95, Cri96, CSTC90]. On the MARS system: [KO87, DRSK89, KV93]. On Delta-4: [Pow91, Pow94, RV89, RVR93, Ver93, Ver94]. On real-time work with Horus: [FR95b, FB96].

21. Transactional Systems

We first encountered the transactional execution model in Chapter 7, in conjunction with client-server architectures. As noted at that time, the model draws on a series of assumptions to arrive at a style of computing that is uniquely well suited to applications that operate on databases. In this chapter we consider some of the details that Chapter 7 did not cover, notably the issues involved in implementing transactional storage mechanisms, and the problems that arise when transactional architectures are extended to encompass transactional access to distributed objects in a reliable distributed system.

Without repeating the material covered earlier, it may be useful to start by reviewing the transactional model in light of what we have subsequently learned about other styles of distributed computing and distributed state. Notice first that the assumptions underlying the transactional approach are quite different from those that underly the virtual synchrony model. Transactional applications are expected to be structured in terms of the basic transactional constructs: *begin, read, update,* and *commit* or *abort*. They are assumed to have been written in isolation, so that they will operate correctly when applied to an idle database system in an initially consistent state. Each transaction, in effect, is a function that transforms the database from a consistent state into a new consistent state. The database, for its part, is a well defined entity that manages data objects, has a limited interface by which transactions operate on it, and manages information using operations with well understood semantics.

General purpose distributed systems, and many client-server applications, match such a model only to a limited degree. The computations performed may or may not act upon saved data in a database, and even when they are, it will be difficult to isolate “data access operations” from other types of message-based interactions and operations.

Additionally, the basic reliability goals of the transactional model are tied closely to its programming model. The transactional reliability guarantees are basically that if a server or client crashes, prior to the commit point of a transaction, a complete rollback of the server state will occur: it is as if the transaction had never been executed. There is a strong emphasis on recoverability of the database contents after a crash: any committed transaction will have effects that survive repeated server crashes and restarts. This strong separation of computation from data, coupled with an emphasis on recoverability (as opposed, for example, to continuous availability), distinguishes the transactional approach from the process group replication schemes we have studied in the preceding chapters of this text.

One could ask whether general purpose distributed programs couldn’t be considered as transactional programs, in this manner mapping the general case to the transactional one. This turns out to be very hard to do. General purpose distributed programs lack a well-defined *begin* or *commit* point, and it would not always be practical to introduce such a structure — sometimes one could do so, but often it would be difficult. They lack a well defined separation of program (transactional client) from persistent state (database); again, some applications could be represented this way, but many could not. Indeed, it is not unreasonable to remark that because of the powerful support that exists for database programming on modern computer systems, most database applications are in fact implemented using database systems. The applications that are left over are the ones where a database model either seems unnatural, fails to match some sort of external constraint, or would lead to extremely inefficient execution. This perspective argues that the distributed applications of interest to us will probably split into the transactional ones, and others that are unlikely to match the transactional model even if one tries to force them into it.

Nonetheless, the virtual synchrony model shares some elements of the transactional one: the serialization ordering of the transactional model is similar to the view-synchronous addressing and ordered delivery properties of a multicast to a process group[20]. Virtual synchrony can be considered as having substituted the notion of a multicast for the concept of the transaction itself: in virtual synchrony one talks about a single operation that affects multiple processes, while in transaction systems one talks about a sequence of *read* and *update* operations that are treated as a single atomic unit. The big difference is thus that whereas explicit data semantics are natural in the context of a database, they are absent in the communication-oriented world that we considered in studying the virtual synchrony protocols.

As we examine the transactional approach in more detail, it is important to keep these similarities and differences in mind. One could imagine using process groups and group multicast to implement replicated databases, and in fact there have been several research projects that have done just this. A great many distributed systems combine transactional aspects with non-transactional ones, using transactions where a database or persistent data structure is present, and using virtual synchrony to maintain consistently replicated in-memory structures, to coordinate the actions of groups of processes, and so forth. The models are different in their assumptions and goals, but not incompatible. Indeed, there has been work on merging the execution models themselves, although we will not present that work here.

Perhaps the most important point is the one stated at the start of this chapter: transactions focus primarily on recoverability and serializability, while virtual synchrony focuses primarily on order based consistency guarantees. This shift in emphasis has pervasive implications, and even if one could somehow merge the models, it is likely that they would still be used in different ways. Indeed, it is not uncommon for distributed systems engineers to try and simplify their lives by using transactions throughout a complex distributed system, as its sole source of reliability, or by using virtual synchrony throughout, exploiting dynamically uniform protocols as the sole source of external consistency. Such approaches, in this author's experience, are rarely successful.

In the former direction, it can be difficult to use transactions as the basic model for communications based systems because the interactions between transactional applications are necessarily structured into invocations of operations and manipulation of persistent data. To send a message, a process will potentially need to write the message into a database, from which the destination process is expected to read it. Moreover, there is the need to arrange that the transaction begin, that the operations issued by identifiable as part of the same transaction (they need some form of unique transaction id), and that it commit in a well defined way. Finally, the application itself needs a sensible way to deal with abort.

While this matches well with database access, it is hard to map such a model to an air traffic application in which a controller works with a continuously updated screen, interacting with pilots, other controllers, and various services. These applications will typically involve running more than one program at a time; logically some of the actions taken are part of the "same transaction", but for long running programs it may not be clear which actions belong to which transaction, nor may it be clear when the transaction should begin or end. For example, it would make sense that operations associated with different flights be treated as different transactions, but it is not clear how the database system (if one is

[20] For example, one can imagine doing a multicast by *reading the view of the group and then writing to the group members,* and updating the view of the group by *writing to the group view.* Such a transactional implementation of virtual synchrony would address some aspects of the model, such as view synchronous addressing, although it would not deal with others, such as the ordered gap-freedom requirement. More to the point, it would result in an extremely inefficient style of distributed computing, because every multicast to a process group would now require a database update. The analogy, then, is useful because it suggests that the fundamental approaches are closely related and differ more at the level of how one engineers such systems to maximize performance than in any more basic way. However, it is not an architecture that one would want to implement!

present) would recognize this distinction, unless the operations were entirely structured as transactional accesses to a shared database server.

Transactions pay a high price to guarantee the recoverability of persistent data. But in a distributed setting that replicates critical data, recovering a very old and stale copy of the state of some server may be pointless; only very large databases would be worth the trouble of recovering into a known state. The idea that transactions are coded to execute against an idle system and to run in isolation is fundamentally at odds with the notion of a system in which long-running services cooperate explicitly both to load-balance work internally and to coordinate actions with one-another. And then there is the issue of aborts: precisely what should an air-traffic controller do if a necessary action is aborted for reasons internal to the execution model? Worse, if an abort occurs "deep within" a system, how should the program recover: many systems have no obvious notion of an operator and must automatically recover from all conditions that can arise during execution.

Similar objections can be raised for the case where virtual synchrony is applied to what are basically transactional databases. Here, the issue is perhaps less one of a constraining programming style than that the specific programming tools we have discussed seem disconnected from the specific needs of a distributed database system. In fact, there has been considerable interest in applying process group concepts to replication and other aspects of distributed data management, notably through the Newtop protocol of the Arjuna system [EMS95], and we will point to some of the special requirements that arise from such explorations. To the application developer, though, the point is that if an application contains what are logically "databases", transactional access to them may be highly appropriate, while if it contains groups of dynamically adaptive, highly available, processes and servers, those may be more appropriately treated using virtual synchrony. Virtual synchrony is also a more natural way to *manage* complex distributed systems; transactions seem poorly matched to this goal. Very elaborate reliable distributed systems will probably need both technologies.

In the remainder of this chapter we move beyond these philosophical issues to more detailed technical ones, concerning the implementation of transactional systems for distributed environments.

21.1 Implementation of a Transactional Storage System

In this section we briefly review some of the more important techniques used in implementing transactional storage systems. Our purpose is not to be exhaustive or even to try and present the "best" techniques known, as this area is such an important one for database searchers that to cover it in detail would require another textbook! Moreover, that textbook has already been written [GR93, Gra79, BHG87]. Rather, we focus on basic techniques with the purpose of building insight into the reliability mechanisms needed when implementing transactional systems.

21.1.1 Write-ahead logging

A *write ahead log* is a data structure used by a transactional system as a form of backup for the basic data structures that compose the database itself. Transactional systems *append* to the log by writing *log records* to it. These records can record the operations that were performed on the database, their outcome (commit or abort), and can include "before" or "after" images of data that an operation updated. The specific content of the log will depend upon the transactional system itself.

We say that a log satisfies a *write ahead* property if there is a mechanism by which records associated with a particular transaction can be safely and persistently flushed to disk before (ahead of) updates to data records being done by that transaction. In a typical use of this property, the log will record before images (old values) of the records a transaction updates, and commit records for that transaction. When the transaction does an update, the database system will first log the old value of the record being updated, then update the database record itself on disk. Provided that the write-ahead property is respected, the actual order of I/O operations done can potentially be changed to optimize use of the disk. Should the server crash, it can recover by reviewing the uncommitted transactions in the log and reinstalling the original values of any data records that these had modified. The transactions themselves will now be forced to abort, if they have not already done so. Such a process rolls back the transactions that have not committed, leaving the committed ones in place. Later, the log can be garbage collected by cleaning out records for committed transactions (which will never need to be rolled back) and those for uncommitted transactions that have been successfully aborted (and hence need not be rolled back again).

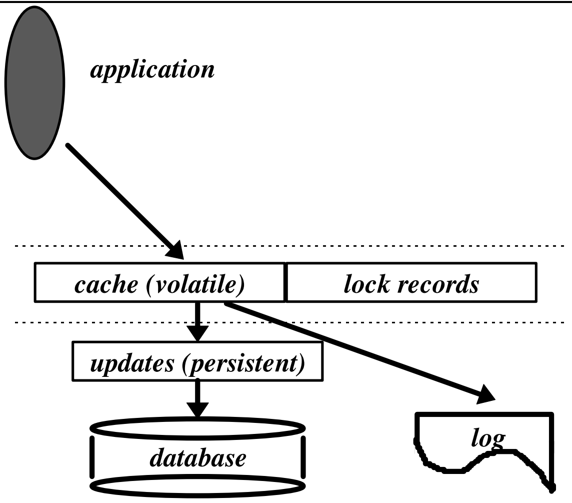


Figure 21-1: Overview of a transactional database server. Volatile data is used to maintain a high speed cache of database records and for storage of lock records for uncommitted transactions. An updates list and the database itself store the data, while a write-ahead log is used to enable transactional rollback if an abort occurs, and to ensure that updates done by committed transactions will be atomic and persistent. The log saves before- or after-images of updated data and lock records associated with a transaction that is running its commit protocol. Log records can be garbage collected after a transaction commits or aborts and the necessary updates to the database have been applied or rolled out.

Although a write-ahead log is traditionally managed on the disk itself, there has been recent research on the use of non-volatile RAM memory or active replication techniques to replace the log with some form of less expensive structure [LGGJ91]. Such trends are likely to continue as the relative performance gap between disks (which seem to have reached a performance limit at approximately 10ms per disk access for a fast disk, and as much as 40 to 50ms per access for a slow one) and communication continues to grow.

21.1.2 Persistent data seen "through" an updates list

Not all transactional systems perform updates to the persistent database at the time they are first issued. The decision to do updates directly depends on several factors, among which are the frequency with which transactions are expected to abort, and the likelihood that the transaction will rewrite the same record repeatedly. The major alternative to performing direct updates on the database itself are to maintain some form of *updates list* in which database records that have been updated are saved. Each access to the database is first filtered through this updates storage object, and if the record being accessed has changed, the changed version is returned. The database itself is only accessed if the updates list does not contain the desired item, and any update made to the database is instead applied to this updates list.

The advantage of such a structure is that the database itself can be maintained in a very efficient search and access structure without requiring costly structural updates as each operation occurs. Periodically, the database can be updated to merge the committed updates from the update list into the persistent part of the database, but this need not be done until a convenient time, perhaps while the database as a whole is under very light load. Moreover, as we will see shortly, the updates list can be generalized to deal with the "nested" transactions that arise when transactional databases are constructed using abstract data types.

The updates list data structure, if present, should not be confused with a cache or "buffer pool". A database cache is a volatile data structure used to accelerate access to frequently used data items by maintaining them in high speed memory. The updates list is a persistent data structure that is logically part of the database itself. Its role is provide the database system with a way of doing database updates without reorganizing the secondary index and other access structures needed to rapidly access items in the main portion of the database.

21.1.3 Non-distributed commit actions

To commit a transaction, it is necessary to ensure that its effects will be atomic even if the database server or client program fails during the commit procedure. In the non-distributed case, the required actions are as follows. First, all log records associated with updates done by the transaction are forced to the disk, as are *lock records* recording the locks currently held by the transaction. Once these actions are taken the transaction is *prepared to commit*. A log record containing the *commit bit* is now written to disk; once it is recorded in a persistent manner in the log the transaction is said to have *committed*.

Next, updates done by the transaction are applied to the updates list or database. In many transactional system this updating is done while the transaction is running, in which case this step (and the forcing of log records to disk, may have already occurred before the transaction reached the commit point.

Finally, when the updates have all been performed, the locks associated with the transaction are released and any log records associated with the transaction are freed for reuse by other transactions. The transaction is now said to be *stable*.

To abort a transaction, the log records associated with it are scanned and used to roll back any updates that may have been performed. All locks associated with the transaction are released, and the log records for the transaction are freed.

In the event that the client process should crash before requesting that the transaction commit or abort, the database server may *unilaterally abort* the transaction. This is done by executing the abort algorithm and later, if the client ever presents additional requests to the server, refusing them and returning an *already aborted* exception code.

Finally, in the event that the database server should crash, when it recovers it must execute a log recovery procedure before re-enabling access to the database. During this process, any transactions that are not shown as committed are aborted, and any updates that may have been done are backed out. Notice that if the log stored before-images, backing out updates can be done by simply reinstalling the previous values of any records that were written by the transaction, and this operation can be done as many times as necessary if the database server crashes repeatedly before recovering (that is, the recovery operation is *idempotent,* meaning that it can be performed repeatedly with the same effect as if it were performed only once).

For transactions shown as committed in the log, the database server recovers by completing the commit procedure and then freeing the log records. Abstractly, the database server can be thought of as recovering in a state where the committed transactions continue to hold any locks that they held at the time of the commit; this will be useful in the case of a distributed transaction on multiple databases.

21.2 Distributed Transactions and Multi-Phase Commit

When a transaction operates on multiple database, it is said to be a *distributed transaction*. The commit problem now becomes the multiphase commit problem we studied in Section 13.6.1. To commit, each participating database server is first asked to *prepare to commit*. If the server is unable to enter this state, it votes for abort, and otherwise flushes log records and agrees that it is prepared. The transaction commits only if all the participating servers are prepared to commit, and otherwise aborts. For this purpose, the transactional commit protocols presented earlier can be used without any modifications at all.

In the case of a database server recovery into the prepared state of a transaction, it is important for the server to act as if that transaction continues to hold any locks that it held at the time that it first became prepared to commit (including read locks, and even if the transactions was a read-only one from the perspective of the database server in question). These locks should continue to be held until the outcome of the commit protocol is known and the transaction can complete by committing or aborting. When a transaction has read data at a server that subsequently crashes and recovers, having lost its read locks, before the transaction is prepared to commit, the transaction must be aborted: otherwise, these "broken" read locks can permit some other transaction to acquire update locks and to commit that should have been serialized after the transaction that first held the read locks. Such a behavior is easily seen to result in non-serializable executions. Thus, a distributed transaction must include all database servers it has accessed in its commit protocol, not just the ones at which it performed updates.

21.3 Transactions on Replicated Data

A transactional system can replicate data by applying updates to all copies of the database, while load-balancing queries across the available copies (in a way that will not change the update serialization order that is used!). In the most standard approach, each database server is treated as a separate database, and each update is performed by updating at least a quorum of replicas. The transaction aborts if fewer than a quorum of replicas are operational. It should be noted that this method of replication, although much better known than other methods, performs poorly in comparison with a more sophisticated method described in Section 21.6.

The reality is that few existing database servers make use of replication for high availability, and hence the topic is primarily of academic interest. Transactional systems that are concerned with availability more often use primary-backup schemes in which a backup server periodically is passed a log of committed actions that were performed on a primary server. Such a scheme is faster (because the backup is not included in the commit protocol) but also has a window during which updates by committed transactions can be temporarily lost (e.g. if the log records for a committed transaction have not yet reached the backup when the primary crashes). When this occurs, the lost updates are rediscovered later, after the primary recovers, and are either merged into the database or, if this would be inconsistent with the database state, human intervention is requested.

Another option is to use a spare computer connected by a dual ported disk controller to a highly reliable RAID style disk subsystem. If the primary computer on which the database is running fails, it can be restarted on the backup computer with little delay. The RAID disk system provides a degree of protection against hardware failures of the stored database in this case.

Although database replication for availability remains uncommon, there is a small but growing commercial market for systems that support distributed transactions. The limiting factor for widespread acceptance of these technologies remains performance. Whereas a non-replicated, non-distributed transactional system may be able to achieve thousands or tens of thousands of short update transactions and short read transactions per second, distributed transactional protocols and replication slows such systems to perhaps hundreds of updates per second. Although such performance levels are adequate to sustain a moderately large market of customers who value high availability or distributed consistency more than performance, the majority of the database marketplace remains focused on scaleable high performance systems. Such customers are apparently prepared to accept the risk of downtown because of hardware or software crashes to gain an extra factor of ten to one hundred in performance. These things said, however, it should again be noted that process group technology may offer a compromise that combines high performance with replication for increased availability or scaleable parallelism. We will return to this issue below, in Section 21.5.5.

21.4 Nested Transactions

Recall that at the outset of this textbook, we suggested that object oriented distributed systems architectures are a natural match with client-server distributed systems structures. This raises the question of how transactional reliability can be adapted to object-oriented distributed systems.

As we saw in Chapter 6, object oriented distributed systems are typically treated as being composed of *active objects* that invoke operations on *passive objects*. To some degree, of course, the distinction is an artificial one, because some passive objects have active computations associated with them, for example to rearrange a data structure for better access behavior. However, to keep this section simple, we will accept the division. We can now ask if the active objects cannot be treated as transactions, and the passive ones as small database servers.

Such a step leads to what are called *nested transactions* [Mos82]. The sense in which these transactions are nested is that when an active object invokes an operation on an abstract object stored within an object-oriented database, that object may implement the operation by performing a series of operations on some other, more primitive, database object. For example, an operation that inserts a name into a list of names maintained in a name server may be implemented by performing a series of updates on a file server in which the name list and associated values are actually stored. One now will have a tree-structured perspective on the transactions themselves, in which each "level" of object performs a transaction on the objects below it.

In such a tree, only the top-most level corresponds to an active object or "program" in the conventional sense. The intermediate levels of code correspond to the execution of methods (procedures) defined by the passive objects in the database. For these passive objects, transactions begin with the operation invocation by the invoking object, and end when a result is returned. That is, procedure executions (operation invocations) are treated as starting with an implicit *begin* and ending with an implicit *commit* in the normal return case. Error conditions can be mapped to an *abort* outcome. The active object at the very top of the tree, in contrast, is said to *begin* a *top-level transaction* when it is started, and to *commit* when it terminates normally. A nested transaction is shown in Figure 21-2.

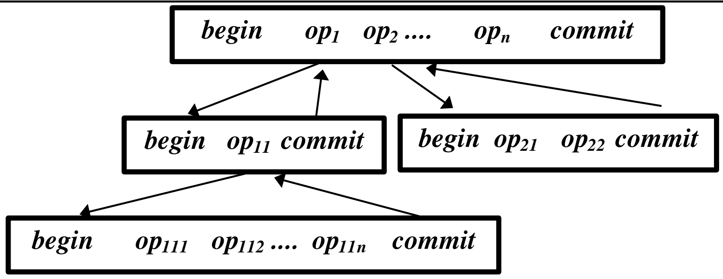

begin op_1 op_2 .... op_n commit
begin op_{11} commit
begin op_{21} op_{22} commit
begin op_{111} op_{112} .... op_{11n} commit

Figure 21-2: Nested transaction. The operations are numbered hierarchically: op_{ijk} thus represents the k'th suboperation initiated by the j'th suboperation initiated by operation i at the "top level". Commit and abort becomes relative in this model, which is due to Moss and Liskov.

The nested transaction model can be used for objects that are co-located on a single object repository, or for objects distributed among multiple repositories. In both cases, the basic elements of the resulting system architecture are similar to the approach used for a single-level transaction system. The details differ, however, because of the need to extend the concurrency control mechanisms to deal with nesting.

The easiest way to understand nested transactions is to view each subtransaction as a transaction that runs in a context created by its parent transaction and any committed sibling subtransactions that the parent executed prior to it. Thus, operation op_{21} in Figure 21-2 should see a database state that corresponds to having executed the subtransaction below op_1 and committing it, even though the effects of that subtransaction will not in fact become permanent and “globally visible” until the main transaction commits. This approach can be extended to deal with internal concurrency, for example if op_1 were executed in parallel with op_2, but this issue lies outside of the scope of topics of this text.

Moss proposed a notion of lock and data version inheritance that accomplishes this goal. In his approach, each subtransaction operates by creating new versions of data items and acquiring locks which are *inherited* by the subtransactions’s immediate parent when the subtransaction commits, or return to the state prior to when the subtransaction began if it aborts. These inherited locks and data values are accessible to other subtransactions of the parent that now retains them, but remain inaccessible to transactions outside of its scope. Moss’ doctoral dissertation includes a proof that this approach yields a nested version of 2-phase locking that guarantees serializable executions.

To implement a nested transaction system, it is usual to start by extending the update list and locking subsystems of the database so that it will know about transactional nesting. Abstracting, the resulting architecture is one in which each lock and each data item is represented as a *stack* of locks or data items. When a new subtransaction is spawned, the abstract effect is to push a new copy of each lock or data item onto the top of the stack. Later, as the subtransaction acquires locks or updates these data items, the copy at the top of the stack is changed. Finally, when the subtransaction aborts, the top-most stack element is discarded, whereas if it commits, the top-most stack item is popped, and the one below it is too, and then the top-most item is pushed back onto the stack. In a similar manner, the stack of lock records is maintained; the one difference being that if a subtransaction obtains a different class of lock than was held by the parent transaction, the lock is left in the more restrictive of the lock modes.

In practice, nested transactional systems are designed to be lazy, so that the creation of new versions of data items or new lock records is delayed until absolutely necessary. Thus, the stack of data items and lock records is not actually generated unless it is needed to perform operations.

A similar abstraction is used to handle the commit and abort mechanisms. Abstractly, as a nested transaction executes, each level of the transaction tracks the data servers that it visits, maintaining a list of *commit participants*. To commit or abort, the transaction will interact with the servers on this list. In practice, however, such an approach would require repeated execution of the of multiphase commit protocols, which will have to run once for each internal node in the transaction tree and once more time for the root! Clearly, this would be unacceptably costly.

To avoid this problem, Liskov's ARGUS group proposed an approach in which commit decisions are *deferred*, so that only the top-level commit protocol is actually executed as a multiphase protocol [LS83, LLSG90, LCJS87]. Intermediate commits are optimistically assumed successful, while aborts are executed directly by informing the commit participants of the outcome. Now, the issue arises of how to handle an access by a subtransaction to a lock held by a sibling subtransaction or to a data item updated by a sibling. When this occurs, a protocol is executed by which the server tracks down a mutual parent and interrogates it about the outcomes, commit or abort, of the full transaction stack separating the two subtransactions. It then updates the stacks of data items and locks accordingly and allows the operation to proceed. In the case where a transaction rarely revisits data items, such a strategy reduces the cost of the nest transactional abstraction to the cost of a flat one-level transaction; the benefit is smaller as the degree of interference increases.

The reader may recall that Liskov's group also pioneered in the use of optimistic (or "lazy") concurrency control schemes. Such approaches, which can be recognized as analogous to the use of asynchronous communication in a process group environment, allow a system to achieve high levels of internal concurrency, improving performance and processor utilization time by eliminating unneeded wait states, much as an asynchronous multicast eliminates delay when a multicast is sent in favor of later delays if a message arrives out of order at some destination. In the limit, these approaches converge upon one in which transactions on non-replicated objects incur little overhead beyond that of the commit protocol run at the end of the top-level transaction, while transactions on replicated objects can be done largely asynchronously, but with a similar overhead when the commit point is reached. These costs are low enough to be tolerable in many distributed settings, and it is likely that at some future time, a commercially viable high performance object-oriented transaction technology will emerge as a serious design option for reliable data storage in distributed computing systems.

21.4.1 Comments on the nested transaction model

Nested transactions were first introduced by the ARGUS project at MIT [Mos82] and rapidly adopted by several other research projects, such as CLOUDS at Georgia Institute of Technology and CMU's TABS and CAMELOT systems [Spe85] (predecessors of ENCINA, the commercial product marketed by Transarc). The model proved elegant but also difficult to implement efficiently, and sometimes quirky. The current view of this technology is that it works best on object-oriented databases which reside mostly on a single storage server, but that it is less effective for general purpose computing in which objects may be widely distributed and in which the distinction between active and passing objects can become blurred.

It is worthy of note that the same conclusions have been reached about database systems. During the mid 1980's, there was a push to develop database operating systems, in which the database would take responsibility for more and more of the tasks traditionally handled by a general purpose operating system. This trend culminated in systems like IBM's AS/400 database server products, which achieve an extremely high level of integration between database and operation systems functionality. Yet there are many communications applications which suffer a heavy performance penalty in these architectures, because direct point to point messages must be largely replaced by database updates followed by a read. While commercial products that take this approach offer optimizations that can equal the performance of general purpose operating systems, users may require special training to understand how and when to exploit them. The trend at the time of this writing seems to be to integrate database servers into general purpose distributed systems by including them on the network, but running non-database operating systems on the general purpose computing nodes that support application programs. This is sometimes called the "open systems" approach to networked computing.

The following example illustrates the sort of problems that can arise when transactions are applied to objects that fit poorly with the database computing model. Consider a file system directory service implemented as an object-oriented data structure: in such an approach, the directory would be a linked list of "named" objects, associating a name with some sort of abstract object corresponding to what would be a file in a conventional file system. Operations on a directory include searching it, scanning it sequentially, deleting and inserting entries, and updating the object nodes. Such a structure is illustrated by Figure 21-3.

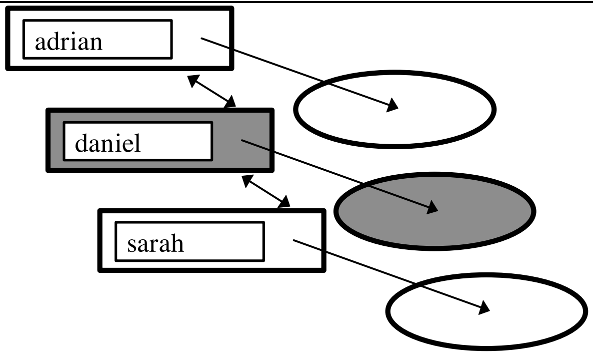


Figure 21-3: While a directory is being updated (in this case, the entry corresponding to "daniel"), other transactions may be prevented from scanning the associated directory node by locks upon it, even if they are searching for some other record such as the one corresponding to "sarah" or "adrian." Although a number of schemes can be used to work around such problems, they require sophistication by the developer, who must consider cases that can arise because of concurrency and arrange for concurrent transactions to cooperate implicitly to avoid inefficient patterns of execution. Such meta-design considerations, the author argues, run counter to the principle of independent design on which transactions are based, and make the overall approach hard to use in general purpose operating system settings.

A typical transaction in such a system might be a program that displays a graphical interface by which the user enters a name, and then looks up the corresponding object. The contents of the object could then be displayed for the user to edit, and the changes, if any, saved into the object, when the user finishes. Interfaces such as this are common in modern operating systems, such as Microsoft's Windows 95 or some of the more advanced versions of UNIX.

Viewed as an instance of a nested transaction, this program begins a transaction and then reads a series of directory records looking for the one that matches the name the user entered. The corresponding node would then be locked for update while the user scrutinizes its contents and updates it. The transaction commit would occur when the record is saved in its changed state. An example of such a locked record is highlighted in gray in Figure 21-3.

But now consider the situation if the system has any concurrency at all. While this process is occurring, the entire data structure may potentially be locked against operations by other transactions, even if they are not interested in the same record as the user is preparing to update! The problem is that any simplistic application of the nested transaction concurrency control rules will leave the top-level records that bind names to objects locked for either read or update, and will leave all the directory records scanned while searching for the name entered by the user locked for reads. Other transactions will be unable to acquire conflicting forms of locks on these records and may thus be delayed until the user (who is perhaps heading down the hall for a cup of coffee!) terminates the interaction.

Many extensions to the nested transaction model have been proposed to cope with this sort of problem. ARGUS, for example, offers a way to perform operations outside of the scope of a transaction, and includes a way for a transaction to spawn new "top level" transactions from deep within a nested execution. Weihl argues for a relaxation of the semantics of objects such as directory servers: in his view of the problem, overspecification of the interface of the directory service is the cause of this sort of problem, and he suggests extensions such as unordered queues and non-deterministic interfaces that correspond to implementations that give better performance. In his approach one would declare the directory to be an unordered semi-queue (an unordered set) and would implement a non-transactional search mechanism in which the search order is non-deterministic and hence need not involve an access to the locked record until all other records have been scanned [Weixx]. Shasha has developed families of concurrent data structures, in which semantic information is exploited to obtain highly concurrent transactional implementations of operations specific to the data type [Shaxx]. Still other researchers have proposed that such problems be addressed by mixing transactional and non-transactional objects, and offered various rules to adapt the ACID properties to such an environment.

The example we gave above arises in a data structure of unsurpassed simplicity. Similar issues would also be encountered in other data structures, such as doubly linked lists where order *does* matter, trees, hash tables, stacks, and so forth. In each case, a separate set of optimizations are needed to achieve optimal levels of concurrency.

This author and many others who have worked with transactions have concluded that although the model works very well for databases, there are simply problems for which the transaction model is poorly matched. The argument is basically that although the various solutions suggested in the literature do work, they have complicated side-effects (interested readers may want to track down the literature concerned with terminating what are called "orphans of an aborted nested transaction", a problem that arises when a nested transaction that has active subransactions aborts, eliminating the database state in which those subtransactions were spawned and exposing them to various forms of inconsistency). The resulting mechanisms are complex to work with, and many users would have problems using them correctly; some developers of nested transaction systems have suggested that only experts would be likely to actually build transactional objects, while most "real" users would work with libraries of preconstructed objects. Thus, even if mechanisms for overcoming these issues do exist, it seems clear that nested transactions do not represent an appropriate general purpose reliability solution for non-database applications.

The commercial marketplace seems to have reached a similar decision. Transactional systems consist largely of relational databases (which may be used to store abstract data types, but in which the relationships between the objects are represented in the transactional tables), or transactional file-structured systems. Although many distributed, object-oriented, transactional systems have been developed, few seem to have made the transition from research prototype to the wider commercial use.

In particular, many of the problems that are most easily solved using process groups are quite hard to solve using transactional solutions. The isolation property of transactions runs counter to the idea of load-balancing in a service replicated at several nodes, or of passing a token within a group of cooperating processes. Conversely, however, notice that transactional mechanisms bring a considerable infrastructure to the problem of implementing the ACID properties for applications that act upon persistent data stored in complex data structures, and this infrastructure is utterly lacking in the virtual synchrony model.

To this author, the implication is that while both models introduce reliability into distributed systems, they deal with very different reliability goals: recoverability on the one hand, and availability on the other. While the models can be integrated so that one could use transactions within a virtual synchrony context and vice versa, there seems to be little hope that the could be merged into a single model that would provide all forms of reliability in a single, highly transparent environment. Integration and co-existence is for this reason a more promising direction, and seems to be the one favored by industry and research groups.

21.5 Weak Consistency Models

There are some applications in which one desires most aspects of the transactional model, but where serializability in the strict sense is not practical to implement. Important among these are distributed systems in which a database must be accessed from a remote node that is sometimes partitioned away from the system. In this situation, even if the remote node has a full copy of the database, it is potentially limited to read-only access. Even worse, the impossibility of building a non-blocking commit protocol for partitioned settings potentially prevents these read-only transactions from executing on the most current state of the database, since a network partitioning failure can leave a commit protocol in the "prepared" state at the remote site.

In practice, many distributed systems treat remote copies of databases as a form of second-class citizen. Such databases are often updated by periodic transfer of the log of recent committed transactions, and used only for read-only queries. Update transactions execute on a *primary copy* of the database. This approach avoids the need for a multi-phase commit but has limited opportunity to benefit from the parallelism inherent in a distributed architecture. Moreover, the delay before updates reach the remote copies may be substantial, so that remote transactions will often execute against a stale copy of the database, with outcomes that may be inconsistent with the external environment in obvious ways. For example, a remote banking system may fail to reflect a recent deposit for hours or days.

In the subsections that follow we briefly present some of the mechanisms that have been proposed as extensions to the transactional model to improve its usefulness in settings such as these.

21.5.1 Epsilon serializability

Originally proposed by Pu, this is a model in which a pre-agreed strategy is used to limit the possible divergence between a primary database and its remote replicas [Pu93]. The "epsilon" refers to the case where the database contains numeric data, and it is agreed that any value read by a transaction is within ε of the correct one.

For example, suppose that a remote transaction is executed to determine the current value of a bank balance, and the result obtained is \$500. If ε=\$100, we can conclude that the actual balance in the database (in the primary version) is no less than \$400 and no more than \$600. The benefit of this approach is that it relaxes the need to run costly synchronization protocols between remote copies of a database and the primary: such protocols are only needed if an update might violate the constraint.

Continuing our example, suppose that we know that there are two replicas and one primary copy of the database. We can now allocate ranges within which these copies can independently perform update operations without interacting with one another to confirm that it is safe to do so. Thus, the primary copy and each replica might be limited to a maximum cumulative update of \$50 (larger updates would require a standard locking protocol). Even if the primary and one replica perform maximum increments to the balance of \$50 respectively, the remaining replica still sees a value that is within \$100 of the true value, and this remains true for any update that the third replica might undertake. In general, the rule for this model is that the minimum and maximum cumulative updates done by “other copies” must be bounded by ε to ensure that a given copy will see a value within ε of the true one.

21.5.2 Weak and strong consistency in partitioned database systems

During periods when a database system may be completely disconnected from other replicas of the same database, we will in general be unable to determine a safe serialization order for transactions originating at that disconnected copy.

Suppose that we want to implement a database system for use by soldiers in the field, where communication may be severely disrupted. The database could be a map showing troop positions, depots, the state of roads and bridges, and major targets. In such a situation, one can imagine transactions of varying ranges of urgency. A fairly routine transaction might be to update the record showing where an enemy outpost is located, indicating that there has been “no change” in the status of the outpost. At the other extreme would be an emergency query seeking to locate the closest medic or supply depot capable of servicing a given vehicle.

Serializability considerations underlie the consistency and correctness of the real database, but one would not necessarily want to wait for serializability to be guaranteed before making an “informed guess” about the location of a medical team. Thus, even if a transactional system requires time to achieve a completely stable ordering on transactions, there may be cases in which one would want it to process at least certain classes of transactions against the information presently available to it.

In his doctoral thesis, Amir addressed this problem using the Transis system as a framework within which he constructed a working solution [Ami95; *see also* CGS85, AKA93, TTPD95]. His basic approach was to consider only transactions that can be represented as a single multicast to the database, which is understood to be managed by a process group of servers. (This is a fairly common assumption in transactional systems, and in fact most transactional applications indeed originate with a single database operation that can be represented in a multicast or remote procedure call). Amir’s approach was to use *abcast* (the dynamically uniform or “safe” form) to distribute update transactions among the servers, which were designed to use a serialization order that is deterministically related to the incoming *abcast* order. Queries were implemented as local transactions requiring no interaction with remote database servers.

As we saw earlier, dynamically uniform *abcast* protocols must wait during partitioning failures in all but the primary component of the partitioned system. Thus, Amir’s approach is subject to blocking in a site that has become partitioned away from the main system. Such a site may, in the general case, have a queue of undeliverable and partially ordered *abcasts* that are waiting either for a final determination of their relative ordering, or for a guarantee that dynamic uniformity will be achieved. Each such *abcast* corresponds to an update transaction that could change the database state, perhaps in an order-sensitive way, and which cannot be safely applied until this information is known.

What Amir does next depends on the type of request presented to the system. If a request is urgent, it can be executed either against the last known completely safe state (ignoring these incomplete transactions), or against an approximation to the correct and current state (by applying these transactions, evaluating the database query, and then aborting the entire transaction). Finally, a normal update can simply wait until the safe and global ordering for the corresponding transaction is known, which may not occur until communication has been reestablished with remote sites.

Amir’s work is not the only effort to have arrived at this solution to the problem. Working independently, a group at Xerox Parc developed a very similar approach to disconnected availability in the

Bayou system [TTPD95]. There work is not expressed in terms of process groups and totally ordered, dynamically uniform, multicast, but the key ideas are the same. In other ways, the Bayou system is more sophisticated than the Transis-based one: it includes a substantial amount of constraint checking and automatic correction of inconsistencies that can creep into a database if urgent updates are permitted in a disconnected mode. Bayou is designed to support distributed management of calendars and scheduling of meetings in large organizations, a time-consuming activity that often requires approximate decision making because some desired participants may be on the road or otherwise unavailable at the time a meeting must be scheduled.

21.5.3 Transactions on multi-database systems

The Phoenix system [Mal96], developed by Malloth, Guerraoui, Raynal, Schiper and Wilhelm, adopts a similar philosophy but considers a different aspect of the problem. Starting with the same model as is used in Amir's work and in Bayou, where each transaction is initiated from a single multicast to the database servers, which form a process group, this effort asked how transactions that operate upon multiple objects could be accommodated. Such considerations lead them to propose a generalized multi-group atomic broadcast that is totally ordered, dynamically uniform, and failure atomic over multiple process groups to which it is sent [SR96]. The point of using this approach is that if a database is represented in fragments that are managed by separate servers, each of which is implemented in a process group, a single multicast would not otherwise suffice to do the desired updates. The Phoenix protocol used for this purpose is similar to the extended three-phase commit developed by Keidar for the Transis system, and is considerably more efficient than sending multiple concurrent and asynchronous multicasts to the process groups and then running a multi-phase commit on the full set of participants. Moreover, whereas such as multi-step protocol would leave serious unresolved questions insofar as the view-synchronous addressing aspects of the virtual synchrony model are considered, the Phoenix protocol can be proved to guarantee this property within all of the destination groups.

21.5.4 Linearizability

Herlihy and Wing studied consistency issues from a more theoretical perspective [HW90]. In a paper on the *linearizability* model of database consistency, they suggested that object oriented systems may find the full nested serializability model overly constraining, but still benefit from some forms of ordering guarantee. A nested execution is linearizable if the invocations of *each object, considered independently of other objects, leave that object in a state that could have been reached by some sequential execution of the same operations, in an order consistent with the causal ordering on the original invocation sequence*. In other words, this model says that an object may reorder the operations upon it and interleave their execution provided that it behaves as if it had executed operations one by one, in some order consistent with the (causal) order in which the invocations were presented to it.

Linearizability may seem like a very simple and obvious idea, but there are many distributed systems in which servers might not be guaranteed to respect this property. Such servers can allow concurrent transactions to interfere with one another, or may reorder operations in ways that violate intuition (for example by executing a read-only operation on a state that is sufficiently old to be lacking some updates that were issued before the read by the same source). At the same time, notice that traditional serializability can be viewed as an extention of linearizability (although serializability does not require that the causal order of invocations be respected, few database systems intentionally violate this property). Herlihy and Wing argue that if designers of concurrent objects at least prove them to achieve linearizability, the objects will behave in an intuitive and consistent way when used in a complex distributed system; should one then wish to go further and superimpose a transactional structure over such a system, doing so simply requires stronger concurrency control. This author is inclined to agree: linearizability seems like an appropriate "weakest" consistency guarantee for the objects used in a distributed environment. The Herlihy and Wing paper develops this idea by presenting proof rules for demonstrating that an object implementation achieves linearizability; however, we will not discuss this issue here.

21.5.5 Transactions in Real-Time Systems

The option of using transactional reliability in real-time systems has been considered by a number of researchers, but the resulting techniques have apparently seen relatively little use in commercial products. There are a number of approaches that can be taken to this problem. Davidson is known for work on transactional concurrency control subject to real-time constraints; her approach involves extending the scheduling mechanisms used in transactional systems (notably, timestamped transactional systems) to seek to satisfy the additional constraints associated with the need to perform operations before a deadline expires.

Amir, in work on the Transis project, has looked at transactional architectures in which data is replicated and it may be necessary to perform a transaction with weaker consistency than the normal serializability model because of temporal or communication constraints [Ami95]. For example, Amir considers the case of a mobile and disconnected user who urgently requires the results of a query, even at the risk that the database replica on which it will be executed is somewhat out of date. The Bayou project, descibed below, uses a similar approach. These methods can be considered "real-time" to the degree to which they might be used to return a result for a query that has a temporal constraint inconsistent with the need to run a normal concurrency control algorithm, which can delay a transaction for an unpredictable period of time.

Broadly, however, the transactional model is fairly complex and consequently ill-suited for use in settings where the temporal constraints have fine granularity with regard to the time needed to execute a typical transaction. In environments where there is substantial "breathing room" transactions may be a useful technique even if there are real-time constraints that should be taken into account, but as the temporal demands on the system rise, more and more deviation from the pure serializability model is typically needed in order to continue to guarantee timely response.

21.6 Advanced Replication Techniques

Looking to the future, one of the more exciting research directions of which the author is aware involves the use of process groups as a form of coherently replicated cache to accelerate access to a database. The idea can be understood as a synthesis of Liskov's work on the Harp file system [LGGJ91], the author's work on Isis [BR94], and research by Seltzer and others on log-structured database systems [Sel93]. However, this author is not aware of any publication in which the contributions of these disparate systems are unified.

To understand the motivation for this work, it may help to briefly review the normal approach to replication in database systems. As was noted earlier, one can replicate a data item by maintaining multiple copies of that item on servers that will fail independently, and updating the item using a transaction that either writes all copies or at least writes to a majority of copies of the item. However, such transactions are slowed by the quorum read and commit operations: the former will now be a distributed operation and hence subject to high overhead, while the latter is a cost not paid in the non-distributed or non-replicated case.

For this reason, most commercial database systems are operated in a non-distributed manner, even in the case of technologies such as Tuxedo or Encina that were developed specifically to support distributed transactional applications. Moreover, many commercial database systems provide a weak form of replication for high availability applications, in which the absolute guarantees of a traditional serializability model are reduced to improve performance. The specific approach is often as follows. The database system is replicated between a primary and backup server, whose roles will be interchanged if the primary fails and later is repaired and recovers. The primary server will, while running, maintain a log of committed transactions, periodically transmitting it to the backup, which applies the corresponding updates.

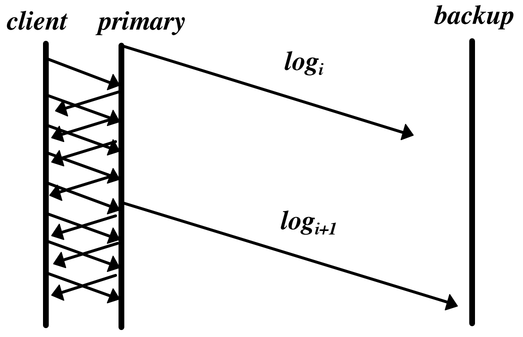


Figure 21-4: Many commercial database products achieve high availability using a weak replication policy that can have a window of vulnerability. In this example, the red transactions have not been logged to the backup and hence can be lost if the primary fails; the green transactions, on the other hand, are "stable" and will not be lost even if the primary fails. Although lost transactions will be recovered when the primary restarts, it may not be possible to reapply the updates automatically. A human operator intervenes in such cases.

Notice that this protocol has a "window of vulnerability". If a primary server is performing transactions rapidly, perhaps hundreds of them per second, the backup may lag by hundreds or thousands of transactions because of the delay associated with preparing and sending the log records. Should the primary server now crash, these transactions will be trapped in the log records: they are committed and the client has potentially seen the result, but the backup will take over in a state that does not yet reflect the corresponding updates. Later, when the primary restarts, the lost transactions will be recovered and, hopefully, can still be applied without invalidating other actions that occurred in the interim; otherwise, a human operator is asked to intervene and correct the problem. The benefit of the architecture is that it gives higher availability without loss of performance; the hidden cost, however, is the risk that transactions will be "rolled back" by a failure, creating noticeable inconsistencies and a potentially difficult repair problem.

As it happens, we can do a less costly job of replicating a database using process groups, and may actually *gain* performance by doing so!

The idea is the following. Suppose that one were to consider a database as being represented by a checkpoint and a log of subsequent updates. At any point in time, the state of the database could be constructed by loading the checkpoint and then applying the updates to it; if the log were to grow too long, it could be truncated by forming a new checkpoint. This isn't an unusual way to actually view database systems: Seltzer's work on log-structured databases [Sel93] in fact implemented a database this way and demonstrated some performance benefits by doing so, and Liskov's research on Harp (a non-transactional file store that was implemented using a log-based architecture) employed a similar idea, albeit in a system with non-volatile RAM memory. Indeed, within the file system community, Rosenblum's work on LFS (a log-structured file system) revolutionized the architecture of many file system products [RO91]. So, it is entirely reasonable to adopt a similar approach to database systems.

Now, given a a checkpoint and log representation of the database, a database server can be viewed as a process that caches the database contents in high speed volatile memory. Each time the server is launched, it reconstructs this cached state from the most recent checkpoint and log of updates, and subsequently transactions are executed out of the cache. To commit a transaction in this model, it suffices to force a description of the transaction to the log (perhaps as little as the transactional request itself and the serialization order that was used). The database state maintained in volatile memory by the server can safely be discarded after a failure, hence the costly disk access associated with the standard database server architecture are avoided. Meanwhile, the log itself becomes an append-only structure that is almost never reread, permitting a very efficient storage on disk. This is precisely the sort of logging studied by Rosenblum for file systems and Seltzer for database systems, and is known to be very cost effective for small to moderate sized databases it can work well. Subsequent research has suggested that this approach can also be applied to very large databases.

But now our process group technology offers a path to further performance improvements through parallelism. What we can do is to use the lightweight replication methods of Chapter 15 to replicate the volatile, "cached" database state within a group of database servers, which can now use one of the load-balancing techniques of Section 15.3.3 to subdivide the work of performing transactions. Within this process group, there is *no need to run a multi-phase commit protocol!* To see this, notice that just as the non-replicated volatile server is merely a cache of the database log, so the replicated group is merely a volatile, cached database state.

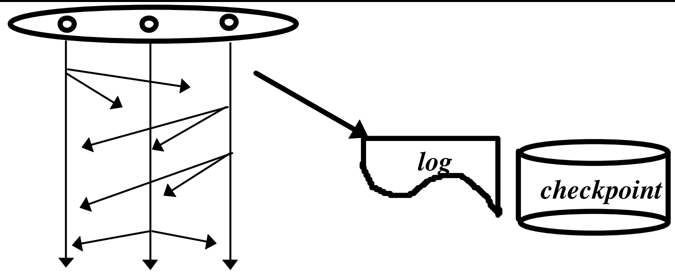


Figure 21-5: Future database systems may gain performance benefits by exploiting process groups as scaleable "parallel" front-ends that cache the database in volatile memory and run transactions against this coherently cached state in a load-balanced manner. A persistent log is used to provide failure atomicity; because the log is a write-only structure it can be optimized to give very high performance. The log would record a description of each update transaction and the serialization order that was used; only committed transactions need be logged in this model.

When we claim that there is no need to run a multiphase commit protocol here, it may at first seem that such a claim is incorrect, since the log records associated with the transaction do need to be forced to disk (or to NVRAM if we use the Harp approach), and if there is more than one log, there will need to be a coordination of this activity to ensure that either all logs reflect the committed transaction, or that none does. For availability, it may actually be necessary to replicate the log, and if this is done, a multi-phase commit would be unavoidable. However, in many settings it might make sense to use just a single log server; for example, if the loging device is itself a RAID disk, then the intrinsic fault-tolerance of the RAID technology could be adequate to provide the degree of availability desired for our purposes. Thus, it may be better to say that there is no *inherent* reason for a multiphase commit protocol here, although in specific cases one may be needed.

The primary challenge associated with this approach is to implement a suitable concurrency control scheme in support of it. While optimistic methods are favored in the traditional work on distributed databases, it is not clear that they represent the best solution for this style of group-structured replication married to a log-structured database architecture. In the case of pessimistic locking, a solution is known from the work of Joseph and this author in the mid 1980'. In the approach developed by Joseph, data is replicated within a process group [Jos86]. Reads are done from any single local copy and writes are done by issuing an asynchronous *cbcast* to the full group. Locking is done by obtaining local read locks and replicated write locks, the latter using a token-based scheme. The issue now arises of read locks that can be broken by a failure; this is addressed by *re-registering* read locks at an operational database server if one of the group members fails. In the scheme Joseph explored, such re-registration occurs during the flush protocol used to reconfigure the group membership.

Next, Joseph introduced a rule whereby a write lock must be granted in the same process group view in which it was requested. If a write lock is requested in a group view where process *p* belongs to the group, and *p* fails before the lock is granted (perhaps because of a read lock some transaction held at process *p*), this forces the transaction requesting the write lock to release any locks it successfully acquired and to repeat its request. The repeated request will occur after the read-lock has been re-registered, avoiding the need to abort a transaction because its read locks were broken by a failure. In such an approach, the need to support unilateral transaction abort is eliminated, because the log now provides persistency, and locks can never be lost within the process group (unless all its members fail, which is a special case). Transaction commit becomes an asynchronous *cbcast*, with the same low cost as the protocol used to do writes.

Readers familiar with transactional concurrency control may be puzzled by the similarity of this scheme to what is called the *available copies* replication method, an approach that is known to yield non-serializable executions [BHG87]. In fact, however, there is a subtle difference between Joseph's scheme and the available copies scheme, namely that Joseph's approach depends on group membership changes to trigger lock reregistration, whereas the available copies scheme does not. Since group membership, in the virtual synchrony model, involves a consensus protocol that provides consistent failure notification throughout the operational part of a system, the inconsistent failure detections that arise in the available copies approach do not occur. This somewhat obscure observation does not seem to be widely known within the database community.

Using Joseph's pessimistic locking scheme, a transaction that does not experience any failures will be able to do local reads at any copy of the replicated data objects on which it operates. The update and commit protocols both permit immediate local action at the group member where the transaction is active, together with an asynchronous *cbcast* to inform other members of the event. Only the acquisition of a write lock and the need to force the transaction description and commit record (including the serialization order that was used) involve a potential delay. This overhead however is counterbalanced by the performance benefits that come with scaleable parallelism.

The result of this effort represents an interesting mixture of process group replication and database persistence properties. On the one hand, we get the benefit of high-speed memory-mapped database access, and can use the very lightweight non-uniform replication techniques that achieved such good performance in previous chapters. Moreover, we can potentially do load-balancing or other sorts of parallel processing within the group. Yet the logging method also gives us the persistence properties normally associated with transactions, and the concurrency control scheme provides for traditional transactional serializability. Moreover, this benefit is available without special hardware (such as NVRAM), although NVRAM would clearly be beneficial if one wanted to replicate the log itself for higher available. To the author, the approach seems to offer the best of both worlds.

The integration of transactional constructs and process groups thus represents fertile territory for additional research, particularly of an experimental nature. As noted earlier, it is clear that developers of reliable distributed systems need group mechanisms for high availability and transactional ones for persistence and recoverability of critical data. Integrated solutions that offer both options in a clean way could lead to a much more complete and effective programming environment for developing the sorts of robust distributed applications that will be needed in complex environments.

21.7 Related Readings

Chapter 26 includes a review of some of the major research projects in this area, which we will not attempt to duplicate here. For a general treatment of transactions, this author favors [GR93, BHG87]. On the nested transaction model, [Mos82]. Disconnected operation in transactional systems [Ami95, CGS85, AKA93, TTPD95]. Log-based transactional architectures [LGGJ91, Jos86, Sel93, BR94].

22. Probabilistic Protocols

The protocols considered in previous chapters of this textbook share certain basic assumptions concerning the way that a distributed behavior or a notion of distributed consistency is derived from the local behaviors of system components. Although we have explored a number of styles of protocol, the general pattern involves reasoning about the possible system states observable by a correct process, and generalizing from this to properties that are shared by sets of correct processes. This approach could be characterized as a "deductive" style of distributed computing, in which the causal history prior to an event is used to deduce system properties, and the possible deductions by different processes are shown to be consistent in the sense that, through exchanges of messages, they will not encounter direct contradictions in the deduced distributed state.

In support of this style of computing we have reviewed a type of distributed system architecture that is hierarchical in structure, or perhaps (as in Transis) composed of a set of hierarchical structures linked by some form of wide-area protocol. There is little doubt that this leads to an effective technology for building very complex, highly reliable distributed systems. One might wonder, however, if there are *other* ways to achieve meaningful forms of consistent distributed behavior, and if so, whether the corresponding protocols might have advantages that would favor their use under conditions where the protocols we have seen up to now, for whatever reason, encounter limitations.

This line of reasoning has motivated some researchers to explore other styles of reliable distributed protocol, in which weaker assumptions are made about the behavior of the component programs but stronger ones are made about the network. Such an approach results in a form of protection against misbehavior whereby a process fails to respect the rules of the protocol but is not detected as having failed. In this chapter we discuss the use of probabilistic techniques to implement reliable broadcast protocols and replicated data objects. Although we will see that there are important limitations on the resulting protocols, they also represent an interesting design point that may be of practical value in important classes of distributed computing systems. Probabilistic protocols are not likely to replace the more traditional deductive protocols anytime soon, but they can be a useful addition to our repertoire of "tools" for constructing reliable distributed systems, particularly in setting where the load and timing properties of the system components are extremely predictable.

22.1 Probabilistic Protocols

The protocols we will be looking at in this section are scaleable and *probabilistically reliable*. Unlike the protocols presented previously, they are based on a probabilistic system model somewhat similar to the "synchronous" model that we considered in our discussion of real-time protocols. In contrast to the asynchronous model, no mechanism for detecting failure is required.

These protocols are scaleable in two senses. First, the message costs and latencies of the protocols grow slowly with the system size. Second, the reliability of the protocols, expressed in terms of the probability of a failed run of a protocol, approaches 0 exponentially fast as the number of processes is increased. This scaleable reliability is achieved through a form of gossip protocol which is strongly self-stabilizing. Such a system has the property that if it is disrupted into an inconsistent state, it will automatically restore itself to a consistent one given a sufficient period of time without failures. Our protocols (particularly for handling replicated data) also have this property.

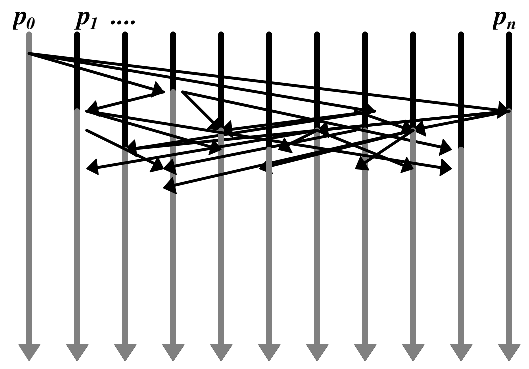


Figure 22-1: A "push" gossip protocol. Each process that receives a message picks some number of destinations (the "fanout", 3 in the example shown) randomly among the processes not known to have received it. Within a few rounds, the message has reached all destinations (above, a process that has received a message is shown in gray). A "pull" protocol can be used to complement this behavior: a process periodically selects a few processes and solicits messages from them. Both approaches exhibit exponential convergence typical of epidemics in densely populated biological systems.

The basic idea with which we will work is illustrated in Figure 22-1, which shows a possible execution for a form of *gossip* protocol developed by Demers and others at Xerox Parc [DGHI87]. In this example of a "push" gossip protocol, messages are diffused through a randomized flooding mechanism. The first time a process receives a message, it selects some fixed percentage of destinations from the set of processes that have not yet received it. The number of such destinations is said to be the *fanout* of the protocol, and the processes selected are picked randomly (a bit vector, carried on the messages, indicates which processes have received them). As these processes receive the message, they relay it in the same manner. Subsequently, if a process receives a duplicate copy of a message it has seen before, it discards the message silently.

Gossip protocols will typically flood the network within a logarithmic number of rounds. This behavior is very similar to that of a biological epidemic, hence such protocols are also known as *epidemic* ones [Bai75]. Notice that although each process may receive a message many times, the computational cost of detecting duplicates and discarding them is likely to be low. On the other hand, the cost of relaying them is a fixed function of the fanout regardless of the size of the network; this is cited as one of the benefits of the approach. The randomness of the protocols has the benefit of overcoming failures of individual processes, in contrast with protocols where each process has a specific role to play and must play it correctly, or fail detectably, for the protocol itself to terminate correctly. Our figure illustrates a push protocol, in the sense that processes with data push it to other processes that lack data by gossiping. A "pull" style of gossip can also be defined: in this approach, a process periodically solicits messages from some set of randomly selected processes. Moreover, the two schemes can be combined.

Demers and his colleagues have provided an analysis of the convergence and scaling properties of gossip protocols based on pushing, pulling, and combined mechanisms, and shown how these can overcome failures [DGHI89]. They prove that both classes of protocols converge towards flooding exponentially quickly, and demonstrate that they can be applied to real problems. The motivation for their work was a scaling problem that arose in the wide-area mail system that was developed at Parc in the 1980's. As this system was used on a larger and larger scale, it began to exhibit consistency problems and had difficulties in accommodating mobile users. Demers and his colleagues showed that by reimplementing the email system to use a gossip broadcast protocol they could overcome these problems, helping ensure timely and consistent email services that were location independent and inexpensive.

22.2 Other applications of gossip protocols

The protocol of Demers is not the first or the only to explore gossip-style information dissemination as a tool for communication in distributed systems. Other relevant work in this area includes [ABM87], an information diffusion protocol that uses a technique similar to the one presented above, and [Gol91a, GT92], which uses gossip as a mechanism underlying a group membership algorithm for wide-area applications. For reasons of brevity, however, we will not treat these papers in detail in the current chapter.

22.3 Hayden's pbcast primitive

In the style of protocol explored at Xerox, the actual rate with which messages will flood the network is not guaranteed because of failures. Instead, these protocols guarantee that, given enough time, eventually either all or no correct processes will deliver a message. This property is called *eventual convergence*. Although eventual convergence is sufficient for many uses, the property is weaker than the guarantees of the protocols we used earlier to replicate data and perform synchronization, because eventual convergence does not provide bounds on message latency or ordering properties. Hayden has shown how gossip protocols can be extended to have these properties [HB93], and in this section we present the protocol he developed for this purpose. Hayden calls his protocol *pbcast,* characterizing it as a probabilistic analog of the *abcast* protocol for process groups.

The *pbcast* protocol is based on a number of assumptions about the environment, which may not hold in typical distributed systems. Thus, after presenting the protocol, we will need to ask ourselves when the protocol could appropriately be applied. If used in a setting where these assumptions are not valid, *pbcast* might not perform as well as the analysis would otherwise suggest.

Specifically, *pbcast* is designed for a static set of processes that communicate synchronously over a fully connected, point-to-point network. The processes have unique, totally ordered identifiers, and can toss weighted, independent random coins. Runs of the system proceed in a sequence of rounds in which messages sent in the current round are delivered in the next.

There are two types of failures, both probabilistic in nature. The first are process failures. There is an independent, per-process probability of at most f_p that a process has a crash failure during the finite duration of a protocol. Such processes are called faulty. The second type of failures are message omission failures. There is an independent, per-message probability of at most f_m that a message between non-faulty processes experiences a send omission failure. The union of all message omission failure events and process failure events are mutually independent. In this model, there are no malicious faults, spurious messages, or corruption of messages. We expect that both f_p and f_m are small probabilities. (For example, unless otherwise stated, the values used in the graphs in this section are $f_m = 0.05$ and $f_p = 0.001$.)

The impact of the failure model above can be visualized by thinking of the power that would be available to an adversary who seeks to cause a run of the protocol to fail by manipulating the system within the bounds of the model. Such an adversary has these capabilities and restrictions:

- An adversary cannot use knowledge of future probabilistic outcomes, interfere with random coin tosses made by processes, cause correlated (non-independent) failures to occur, or do anything not enumerated below.
- An adversary has complete knowledge of the history of the current run of the protocol.
- At the beginning of a run of the protocol, the adversary has the ability to individually set process failure rates, within the bounds $[0..f_p]$.
- For faulty processes, the adversary can choose an arbitrary point of failure.

- For messages, the adversary has the ability to individually set send omission failure probabilities within the bounds of $[0..f_m]$.

Note that although probabilities may be manipulated by the adversary, doing so can only make the system "more reliable" than the bounds, f_p and f_m.

The probabilistic analysis of the properties of the *pbcast* protocol are only valid in runs of the protocol in which the system obeys the model. In particular, the independence properties of the system model are quite strong and are not likely to be continuously realizable in an actual system. For example, partition failures in the sense of correlated communication failures do not occur in this model. Partitions can be "simulated" by the independent failures of several processes, but are of vanishingly low probability. However, the protocols we develop using *pbcast*, such as our replicated data protocol, remain safe even when the system degrades from the model. In addition, *pbcast*-based algorithms can be made self-healing. For instance, our replicated data protocol has guaranteed eventual convergence properties similar to normal gossip protocols: so if the system recovers into a state that respects the model and remains in that state for sufficiently long, the protocol will eventually recover from the "failure" and reconverge to a consistent state.

22.3.1 Unordered pbcast protocol

We begin with an unordered version of *pbcast* with static membership (see Figure 22-2). The protocol itself extends a basic gossip protocol with a quorum-based ordering algorithm inspired by the ordering scheme in CASD [CASD85, CT90]. What makes the protocol interesting is that it is tolerant of failures and that, under the assumptions of the model, it can be analyzed formally.

The protocol consists of a fixed number of rounds, in which each process participates in at most one round. A process initiates a *pbcast* by sending a message to a random set of other processes. When other processes receive a message for the first time, they gossip the message to some other randomly chosen members. Each process only gossips once: the first process does nothing after sending the initial messages and the other processes do nothing after sending their set of gossip messages. Processes choose the destinations for their gossip by tossing a weighted random coin for each other process to determine whether to send a gossip message to that process. Thus, the parameters of the protocol are:

- *P:* the set of processes in the system. $n = |P|$.
- *k,* the number of rounds of gossip to run
- *r,* the probability that a process gossips to each other process (the "weighting" of the coin mentioned above).

The behavior of the gossip protocol mirrors a class of disease epidemics which nearly always infect either almost all of a population or almost none of it. Below, we will show that *pbcast* has a bimodal delivery distribution stems from the "epidemic" behavior of the gossip protocol. The normal behavior of the protocol is for the gossip to flood the network in a random but exponential fashion. If *r* is sufficiently large, most processes will usually receive the gossip within a logarithmic number of rounds.

```
(* State kept per pbcast:  have I  received a message regarding this pbcast yet? *)
let received_already = false

(* Initiate a pbcast. *)
to pbcast(msg):
  deliver_and_gossip(msg,k)

(* Handle message receipt. *)
on receive Gossip(msg,round):
  deliver_and_gossip(msg,round)

(* Auxiliary function. *)
to deliver_and_gossip(msg,round):
  (* Do nothing if already received  it. *)
  if received_already then return

  (* Mark the message as being seen and deliver. *)
  received_already := true
  deliver(msg)

  (* If last round, don't gossip. *)
  if round = 0 then return

  foreach p in P:
    do with probability r:
      sendto p Gossip(msg,round-1)
```

Figure 22-2: Unordered pbcast protocol. The function time() returns the current time expressed in rounds since the first round. Message receipt and pbcast are executed as atomic actions.

22.3.2 Adding Total Ordering

In the protocol shown above, the *pbcast* messages are unordered. However, because the protocol runs in a fixed number of rounds of fixed length, it is trivial to extend it using the same method as was proposed in the CASD protocols. By delaying the delivery of a message until it is known that all correct processes have a copy of that message, totally ordered delivery can be guaranteed. This yields a protocol similar to *abcast* in that it has totally ordered message delivery and reliability within the fixed membership of the process group that invokes the primitive. It would not be difficult to introduce a further extention of the protocol for use in dynamic process groups, but we will not address that issue here.

```
(* Local state: message buffer and  counter for generating unique identifiers. *)
let buffer = {}
let id_counter = 0

(* Initiate a pbcast. *)
to pbcast(msg):
  (* Create unique id for each message. *)
  let id = (my_id, id_counter)
  id_counter := id_counter + 1

  do_gossip(time(),id,msg,k)

(* Handle message receipt. *)
on receive Gossip(timesent,id,msg,round):
  do_gossip(timesent,id,msg,round)

(* Handle timeouts. *)
on timeout(time):
  (* Check for messages ready for delivery.  Assumes buffer is
   * scanned in lexicographic order of (sent,id). *)
  foreach (sent,id,msg) in buffer:
    if sent + k + 1 = time then
      buffer := buffer \ (sent,id,msg)
      deliver(msg)

(* Auxiliary function. *)
to do_gossip(timesent,id,msg,rnd):
  (* If have seen message already, do nothing. *)
  if (timesent,id,msg) in buffer then
    return

  (* Buffer the message for later delivery, and then gossip. *)
  buffer := buffer ∪(timesent,id,msg)
  set_timer timesent + k + 1

  (* If last round, do nothing more. *)
  if rnd = 0 then return

  foreach p in P
    with probability r
      send p Gossip(timesent,id,msg,rnd-1)
```

Figure 22-3: Ordered pbcast protocol, using the method of CASD.

22.3.3 Probabilistic Reliability and the Bimodal Delivery Distribution

Hayden has demonstrated that when the system respects the model, a pbcast is almost always delivered to "most" or to "few" processes, and almost never to "some" processes. Such a delivery distribution is called a "bimodal" one, and is depicted in Figure 22-4. The graph shows that varying numbers of processes will deliver a *pbcast*. For instance the probability that 26 out of the 50 processes deliver a *pbcast* is around 10^{-28}. Such a probabilistic guarantee is, for most practical purposes, a guarantee that the outcome cannot occur. This bimodal distribution property is presented here informally, but later we discuss the method

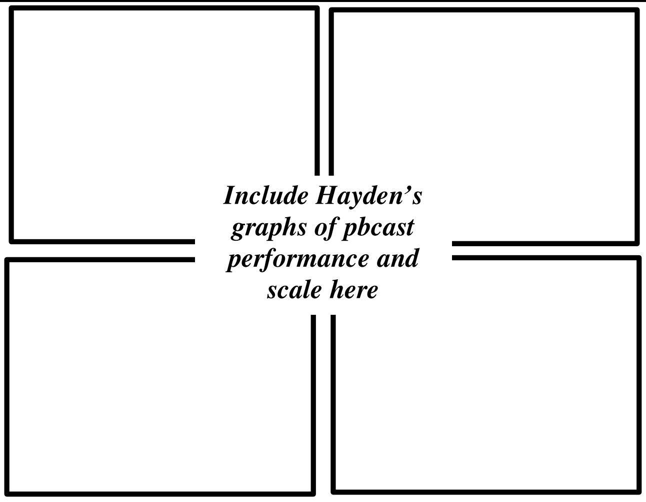

Include Hayden's graphs of pbcast performance and scale here

Figure 22-4: Graphs showing pbcast reliability, performance and scaling.

used by Hayden to calculate the actual probability distributions for a particular configuration of pbcast. In keeping with the generally informal tone of this textbook, we omit the detailed analysis he employed.

A bimodal distribution is useful for voting-style protocols where, as an example, updates must be made at a majority of the processes to be valid; we saw examples of such protocols when discussing quorum replication. Problems occur in these sorts of protocols when failures cause a large number of processes to carry out an update, but not a majority. *Pbcast* overcomes this difficulty through its bimodal delivery distribution by ensuring that votes will almost always be weighted strongly for or against an update, and very rarely be evenly divided. By counting votes, it can almost always be determined whether an update was valid or not, even in the presence of some failed processes.

With *pbcast*, the "bad" cases are when "some" processes deliver the *pbcast* and these are the cases that pbcast makes unlikely to occur. We will call *pbcasts* that are delivered to "some" processes *failed* pbcasts, and pbcasts that are delivered to "few" processes *invalid pbcasts*. The distinction anticipates the replicated data protocol presented below, in which invalid *pbcasts* are inexpensive events, whereas failed ones are potentially costly.

To establish that *pbcast* indeed has a bimodel delivery distribution, Hayden used a mixture of symbolic and computational methods. First, he computed a recurrence relation that expresses the probability that a *pbcast* will be received by *a* processes at the end of round *j* given that the message had been received by *b* processes at the end of round *j-1, c* of these for the first time. In the terminology of a biological infection, *b* denotes the number of processes that were infected during round *j-1* and hence are infectious; the difference between *a* and *b* thus represents the number of *susceptible* processes that had not yet received a gossip message and that are successfully infected during round *j*.

The challenging aspect of this analysis is to deal with the impact of failures, which has the effect of making the variables in the recurrence relation random ones with binomial distributions. Hayden arrives at a recursive formula but not a closed form solution. However, such a formula is amenable to computational solutions, and by writing a program to calculate the various probabilities involved, he is able to arrive at the delivery distributions shown in the figures.

A potential risk in the analysis of pbcast is to assume, as may be done for many other protocols, that the worst case occurs when message loss is maximized. Pbcast's failure mode occurs when there is a partial delivery of a pbcast. A pessimistic analysis must consider the case where local increases in the message delivery probability decrease the reliability of *the overall pbcast protocol*. This makes the analysis quite a bit more difficult than the style of worst-case analysis that can be used in protocols like the CASD one, where the worst case is the one in which the maximum number of failures occur.

22.3.4 An Extension to Pbcast

When the process which initiates a pbcast is not faulty, it is possible to provide stronger guarantees for the pbcast delivery distribution. By having the process which starts a pbcast send more messages, an analysis can be given that shows that if the sender is not faulty the pbcast will almost always be delivered at "most" of the processes in the system. This is useful because an application can potentially take some actions knowing that its previous pbcast is almost certainly going to reach most of the processes in the system. The number of messages can be increased by having the process that initiates a pbcast use a higher value for *r* for just the first round of the pbcast. This extension is not used in the computations that are presented below. Had the extention been included, the distributions would have favored bimodal delivery with even higher probabilities.

22.3.5 Evaluation and Scalability

The evaluation of pbcast is framed in the context of its scalability. As the number of processes increases, pbcast scales according to several metrics. First, the reliability of pbcast grows with system size. Second, the cost per participant, measured by number of messages sent or received, remains at or near constant as the system grows. Having made these claims, it must be said that the version of pbcast presented and analyzed makes assumptions about a network that become less and less realizable for large systems. In practice, this issue could be addressed with a more hierarchically structured protocol, but Hayden's analysis has not been extended to such a protocol. In this section, we will address the scaling characteristics according to the metrics listed above, and then discuss informally how pbcast can be adapted for large systems.

22.3.5.1 Reliability

Pbcast has the property that as the number of processes participating in a pbcast grows, the protocol becomes more reliable. In order to demonstrate this, we present a graph (Figure 22-4(b)) of pbcast reliability as the number of processes are varied between 10 and 60, fixing fanout and failure rates. For instance, the graph shows that with 20 processes the reliability is around 10^{-13}. The graph almost fits a straight line with slope =- 0.45, thus the reliability of pbcast increases almost ten-fold with every two processes added to the system.

22.3.5.2 Message cost and fanout.

Although not immediately clear from the protocol, the message cost of the pbcast protocol is roughly a constant multiple of the number of processes in the system. In the worst cast, all processes can gossip to all other processes, causing $O(n^2)$ messages per pbcast. *r* will be set to cause some expected *fanout* of messages, so that on average a process should gossip to about *fanout* other processes, where *fanout* is some constant, in practice at most 10 (unless otherwise stated, *fanout=7* in the graphs presented in this section). Figure 22-4(c) shows a graph of reliability verses *fanout* when the number of processes and other parameters is held constant. For instance, the graph shows that with a fanout of 7.0, pbcast's reliability is around $10^{-13.}$ In general, the graph shows that the fanout can be increased to increase reliability, but eventually there are diminishing returns for the increased message cost.

On the other hand, *fanout* (and hence cost) can be decreased as the system grows, keeping the reliability at fixed level. In Figure 22-4(d), reliability of at least "twelve nines" (i.e. the probability of a *failed pbcast* is less than or equal to 10^{-12}) is maintained, while the number of processes is increased. The graph shows that with 20 processes a *fanout* of 6.63 achieves twelve-nines reliability, while with 50 processes a *fanout* of 4.32 is sufficient.

22.4 An Unscalable System Model

Although the protocol operating over the system model is scaleable, the system model is not. The model assumes a flat network in which the cost of sending a message is the same between all pairs of processes. In reality, as a real system scales and the network loses the appearance of being flat, this assumption breaks down. There are two possible answers to this problem. The first is to consider pbcast suitable for scaling only to mid-sized systems (perhaps with as many as 100 processes). Certainly, at this size of system, pbcast provides levels of reliability that are adequate for most applications. The second possible solution may be to structure pbcast's message propagation hierarchically, so that a weaker system model can be used which scales to larger sizes. The structure of such a protocol, however, would likely complicate the analysis. Investigating the problem of scaling pbcast to be suitable for larger numbers of processes is an area of future work.

More broadly, the *pbcast* system model is one that would only be reasonable in certain settings. General purpose distributed systems are not likely to guarantee clock synchronization and independence of failures and message delays, as assumed in the model. On the other hand, many dedicated networks would have the desired properties and hence could support *pbcast* if desired. For example, the networks that control telecommunications switches and satellite systems are often designed with guarantees of capacity and known maximum load; in settings such as these the assumptions required by *pbcast* would hold. An interesting issue concerns the use of *pbcast* for high priority, infrequent tasks in a network that uses general purpose computing technologies but supports a notion of message priority. In such settings the *pbcast* protocol messages might be infrequent enough to appear as independent events, permitting the use of the protocol for special purposes although not for heavier loads or more frequent activities.

22.5 Replicated Data using Pbcast

In presenting other reliable broadcast protocols, we used replicated data as our primary “quality” metric. How would a system that replicates data using *pbcast* be expected to behave, and how might such a replicated data object be used? In this section we briefly present a replication and synchronization protocol developed by Hayden and explore the associated issues.

22.5.1 Representation of replicated data

It is easiest to understand Hayden’s scheme if the replicated data managed by the system is stored in the form of a history of values that the replicated data took on, linked by the updates that transformed the system data from each value to the successive one. In what follows, we will assume that the system contains a single replicated data object, although the generalization to a multi-object system is trivial.

22.5.2 Update protocol

Hayden’s scheme uses *pbcast* to transmit updates. Each process applies these updates to its local data copy as they are successfully delivered, in the order determined by the protocol. Were it not for the small but non-zero probability of a *failed pbcast*, this would represent an adequate solution to our problem. However, we know that *pbcast* is very sensitive to the assumptions made in developing the model, hence there is some risk that if the system experiences a brief overload or some other condition that pushes it away from its basic model, failed *pbcasts* may occur with unexpected high frequency, leaving the processes in the system with different update sequences: the updates will be ordered in the same way throughout, but some may be missing updates and others may have an update that in fact reflects a failed pbcast and hence is not visible to many processes in the system..

To deal with this, we will use a *read* protocol that can stabilize the system if it has returned to its normal operational mode (the resulting algorithm will be said to be *self-stabilizing* for this reason). Specifically, associated with each update in the data queue we will also have a distribution of the probability that the update and all previous ones is stable, meaning that the history of the queue prior to that update is believed to be complete and identical to the update queues maintained by other processes. For an incoming update, this distribution will be just the same as the basic *pbcast* reliability distribution; for older updates, it can change as the *read* algorithm is executed.

22.5.3 Read protocol

Hayden's read algorithm distinguishes two types of *read* operation. A local read operation returns the current value of the data item, on the basis of the updates received up to the present. Such an operation has a probability of returning the correct value that can be calculated using the reliability distributions of the updates. If each update rewrites the value of the data item, the probability will be just that of the last update; if updates are in some way state sensitive (such as an increment or decrement operation), this computation involves a recurrence relation.

Hayden also supports a *safe* read, which operates by using a gossip *pull* protocol to randomly compare the state of the process at which the read was performed with that of some number of other processes within the system. As the number of sampled states grows, it is possible to identify failed updates with higher and higher probability, permitting the processes that have used a safe read to clean these from their data histories. The result is that a read can be performed with any desired level of confidence at the cost of sampling a greater number of remote process states. Moreover, each time a *safe read* is performed, the confidences associated with updates that remain on the queue will rise. In practice, Hayden finds that by sampling even a very small number of process states, a read can be done with what is effectively perfect safety: the probability of correctness soars to such a high level that it essentially converges to unity.

Conceptually, the application process that uses an unsafe "local" read should do so only under circumstances where the implications of an erroneous result are not all that serious for the end-user. Perhaps these relate to current positions of aircraft "remote" from the one doing the read operations and hence are of interest but not a direct threat to safe navigation. In contrast, a process would use a safe read for operations that have an external consistency requirement. A safe read is only risky if the network model is severely perturbed, and even then, in ways that may be very unlikely. Thus a safe read might be used to obtain information about positions and trajectories of flights "close" to an aircraft of interest, so as to gain strong confidence that the resulting aircraft routing decisions are safe in a physical sense. However, the safe read may need to sample other process states, and hence would be a slower operation.

One could question whether a probabilistic decision is ever "safe". Hayden reasons that even a normal, non-probabilistic distributed system is ultimately probabilistic because of the impracticality of proving complex systems correct. The many layers of software involved (compiler, operating system, protocols, applications) and the many services and servers involved introduce a probabilistic element to any distributed or non-distributed computing application. In *pbcast* these probabilisties merely become part of the protocol properties and model, but this is not to say that they were not previously "present" in any case, even if unknown to the developer.

22.5.4 Locking protocol

Finally, Hayden presents a *pbcast* based locking protocol that is always safe and is probabilistically live. His protocol works by using *pbcast* to send out locking requests. A locking request that is known to have reached more than a majority of processes in the system is considered to be granted in the order given by the *pbcast* ordering algorithm. If the *pbcast* is unsuccessful or has an uncertain outcome, the lock is released (using any reliable point to point protocol, or another *pbcast*), and then requested again (this may mean that a lock is successfully acquired but, because the requesting process is not able to confirm that a majority of processes granted the lock, released without having been used). It is easy to see that this scheme will be safe, but also that it is only probabilistically live. After use of the lock, the process that requested it releases it. If desired, a timeout can be introduced to deal with the possibility that a lock will be requested by a process that subsequently fails.

22.6 Related Readings

Probabilistic protocols are an esoteric area of research, on which relatively little has been published. For gossip protocols, [DGHI87, ABM87, Gol91a, GT92]. The underlying theory, [Bai75]. Hayden's work [HB93], which draws on [CASD85, CT90].

23. Distributed System Management

In distributed systems that are expected to overcome failures, reconfigure themselves to accommodate overloading or underloading, or to modify their behavior in response to environmental phenomena, it is helpful to consider the system at two levels. At one level, the system is treated in terms of its behavior while the configuration is static. At the second level, issues of transition from configuration to configuration are addressed: these include sensing the conditions under which a reconfiguration is needed, taking the actions needed to reconfigure the physical layout of the system, and informing the components of the new structure.

Most of the material in this chapter is based loosely on work by Wood and Marzullo on a management and monitoring system called *Meta,* and on related work by Marzullo on issues of clock and sensor synchronization and fault-tolerance [Mar84, Mar90, MCBW91, Woo91]. The Meta system is not the only one to have been developed for this purpose, and in fact has been superseded by other systems and technologies since it was first introduced. However, Meta remains unusual for the elegance with which it treats the problem, and for that reason is particularly well suited for presentation in this text. Moreover, Meta is one of the only systems to have dealt with reliability issues.

Marzullo and Wood treat the management issue as a form of programming problem, in which the inputs are events affecting the system configuration and the outputs are *actions* that may be applied to the environment, sets of components, or individual components. *Meta programming* is the problem of developing the control rules used by the meta system to manage the underlying controlled system.

In developing a system management structure, the following tasks must be undertaken:

- *Creation of a system and environment model.* This establishes the conventions for naming the objects in the system, identifies the events that can occur for each type of object, and the actions that can be performed upon it.
- *Linking the model to the real system.* The process of instrumenting the managed system so that the various control points will be accessible to the meta program.
- *Developing the meta programs.* The step during which control rules are specified, giving the conditions under which actions should be taken and the actions required.
- *Interpreting the control rules.* Developing a meta-program that acts upon the control rules, with the degree of reliability required by the application. A focus of this chapter will be on *fault-tolerance of the interpretation mechanisms* and on *consistency of the actions taken.*
- *Visualizing the resulting meta environment.* A powerful benefit of using a meta description and meta control language is that the controlled system can potentially be visualized using graphical tools that show system states in an intuitive manner and permit operators to intervene when necessary.

The state of the art for network management and state visualization is extremely advanced, and tools for this purpose represent an important and growing software market. Less well understood is the problem of managing reliable applications in a consistent and fault-tolerant manner, and this is the issue on which our discussion will focus in the sections that follow.

23.1 A Relational System Model

Marzullo uses a relational database both to model the system itself and the environment in which it runs. In this approach, the goal of the model is to establish the conventions by which the controlled entities and their environment can be referenced, to provide definitions of the relationships between them, and to provide definitions of the sensors and actuators associated with each type of component.

We assume that most readers are familiar with relational databases: they have been ubiquitous in settings ranging from personal finance to library computing systems. Such systems represent the basic entities of the database in the form of *tabular relations* whose entries are called *tuples* each of which contains a unique identifier. Relationships between these entities are expressed by additional relations giving the identifiers for related entities.

For example, suppose that we want to manage a system containing two types of servers: *file_servers* and *database_servers*. These servers execute on *server_nodes*. A varying number of *client_programs* execute on *client_nodes*. For simplicity, we will assume that there are only these three types of programs in the system and these two types of nodes. However, for reliability purposes, it may be that the file servers and database servers are replicated within process groups, and the collection of client programs may vary dynamically.

clid	*uid*	*nid*	*sz*	*lreq*
1	13/7	102	2201	READ
3	15/7	106	1840	READ
4	22/8	106	3103	WRITE

client_program

nid	*load*	*memused*	*vmemused*	*memavail*	*IP addr.*	*protocol*
102	3.5	4574	18544	642	128.13.71.2	SNMP
106	4.7	6620	24321	0	128.13.71.11	SNMP

client_nodes

fsid	*load*	*nid*	*sz*	*uptime*
13	12.2	67	1702	16:20:03
6	.30	33	620	12:22:11
27	3.5	25	980	1:02:19

file_servers

dbid	*load*	*nid*	*sz*	*uptime*
1	7.5	67	1888	16:21:02
2	6.2	25	9590	12:11:09
5	3.1	33	2890	1:21:02

database_servers

nid	*load*	*memused*	*vmemused*	*memavail*	*IP addr*	*protocol*
67	18.1	6541	16187	6151	128.13.67.1	SNMP
25	9.6	6791	21981	6151	128.13.67.2	SNMP
33	10.7	5618	17566	4371	128.13.67.5	SNMP

server_nodes

Figure 23-1: Relational database used to represent system configuration.

Such a system can be represented by a collection of relations, or tables, whose contents change dynamically as the system configuration changes. If we temporarily defer dynamic aspects, such a system state may resemble the one shown in Figure 23-1. The relations that describe client systems are:

- *client_programs*. This relation has an entry (tuple) for each client program. The fields of the relation specify the unique identifier of the client (clid), its user-id (uid), the last request issued by the client program (last_req), the current size of the client program process (sz), and so forth. The field called nid gives the client node on which the client is running. That is, this field "relates" the client_program entity to the client_node entity having that node id.

- *client_nodes*. This relation has an entry for each node on which a client program might be running and, for that node, gives the current load on the node (load), physical memory in use (memused), virtual memory used (vmemused), and physical memory available (memavail).
- *File_servers*. A relation describing the file server processes, similar to that for client processes.
- *Database_servers*. A relation describing the database server processes, similar to that for client processes.
- *Server_nodes*. A relation describing the nodes on which file server and database server processes execute.

Notice that the dependency relationships between the entities are encoded directly into the tuples of the entity relations in this example. Thus, it is possible to query the "load of the compute node on which a given server process is running" in a simple way.

Additionally, it is useful to notice that there are "natural" process group relationships represented in the table. Although we may not chose to represent the clients of a system as a process group in the explicit sense of our protocols from earlier in the text, such tables can encode groups in several ways. The entities shown in any given table can be treated as a group, as can the subsets of entities sharing some value in a field of their tuples, such as the processes residing on a given node. Marzullo uses the term *aggregate* to describe these sorts of process groups, recalling similar use of this term in the field of database research.

23.2 Instrumentation Issues: Sensors, Actuators

The instrumentation problem involves obtaining values to fill in the fields of our modeled distributed system. For example, our server nodes are shown as having "loads", as are the servers themselves. One aspect of the instrumentation problem is to define a procedure for sampling these loads. A second consideration concerns the specific properties of each sensor. Notice that these different "load" sensors might not have the same units or be computed in the same way: perhaps the load on a server is the average length of its queue of pending requests during a period of time, whereas a load on a server node is the average number of runnable programs on that node during some other period of time. Accordingly, we adopt the perspective that values that can be obtained from a system are accessed through *sensors,* which have *types* and *properties*. Examples of sensor types include numeric sensors, sensors that return strings, and set-valued sensors. Properties include the continuity properties of a numeric sensor (e.g. whether or not it changes continuously), the precision of the sensor, and its possible modes of values.

23.3 Management Information Bases, SNMP and CMIP

A management system will require a way to obtain sensor values from the instrumented entities. It is increasingly popular to do this using a standard called the Simple Network Management Protocol (SNMP) which defines procedure calls for accessing information in a Management Information Base or MIB. SNMP is an IP-oriented protocol and uses a form of extended IP address to identify the values in the MIB: if a node has IP address 128.16.77.12, its load might, for example, be available as 128.16.77.12:45.71. A mapping from ascii names to these IP address extensions is typically stored in the Domain Name Service (DNS) so that such a value can also be accessed as gunnlod.cs.cornell.edu:cpu/load. A trivial RPC protocol is used to query such values. Application programs on a node, with suitable permissions, use system calls to update the local MIB; the operating system is also instrumented and maintains the validity of standard system-level values.

The SNMP standard has become widely popular but is not the only such standard in current use. CMIP is a similar standard developed by the telecommunications industry; it differs in the details but is basically similar in its ability to represent values. SNMP and CMIP both standardize a great variety of sensors as well as the protocol used to access them: at the time of this writing, the SNMP standard included more than 4000 sensor values that might be found in a MIB. However, any particular platform will only export some small subset of these sensors and any particular management application will only make use of a small collection of sensors, often permitting the user to reconfigure these aspects. Thus, of the 4000 standard sensors, a typical system may in fact be instrumented using perhaps a dozen sensors of which only 2 or 3 are in fact critical to the management layer.

A monitoring system may also need a way to obtain sensor values directly from application processes, because both SNMP and CMIP have limitations on the nature of data that they can represent and both lack synchronization constructs that may be necessary if a set of sensors must be updated atomically. In such cases, it is common to use RPC-oriented protocols to talk to special *monitoring interfaces* that the application itself supports; these interfaces could provide an SNMP-like behavior, or a special-purpose solution. However, such approaches to monitoring are invasive and entail the development use of wrappers with monitoring interfaces or other modifications to the application. In light of such considerations, it is likely that to the degree that SNMP and CMIP information bases will continue to be the more practical option for representing sensor values in distributed settings.

23.3.1 Sensors and events

The ability to obtain a sensor's value is only a first step in dealing with dynamic sensors in a distributed system. The problem of computing with sensors also involves dealing with inaccuracy and possible sensor failures, developing a model for dealing with aggregates of sensors, and dealing with the issue of time and clock synchronization. Moreover, there are issues of dynamicism that arise if the group of instrumented entities changes of time. We need to understand how these issues can be addressed so that, given an instrumented system, we can define a meaningful notion of *events* which occur when a condition expressed over one or multiple sensors becomes true after having been false or becomes false after having been true.

For example, suppose that Figure 23-2 represents the loads on a group of database servers. We might wish to define an event called "database overloaded" that will occur if more than 2/3 of the servers in the group have loads in excess of 15. It can be seen that the servers in this group briefly satisfied this condition. Yet the sensor samples were taken in such a manner that this condition cannot be detected.

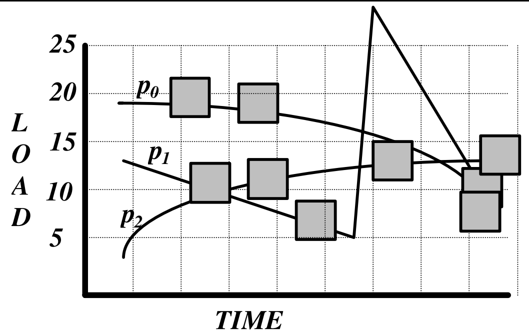


Figure 23-2: Imprecision in time and sensor values can confuse a monitoring system. Here, sensor readings (shaded boxes) are obtained from a distributed system. Not only is there inaccuracy in the values and time of the reading, but they are not sampled simultaneously. To ensure that its actions will be reasonable, a management system must address these issues.

Notice that the sensor readings are depicted as boxes. This is intended to reflect the concept of uncertainty in measurements: temporal uncertainty yields a box with horizontal extend and value uncertainty yields a box with horizontal extent. Also visible here is the lack of temporal synchronization between the samples taken from different processes: unless we have a real-time protocol and the associated infrastructure needed to sample a sensor accurately at a precise time, there is no obvious way to ensure that a set of data points represent a simultaneous system state. Simply sending messages to the database servers asking that they sample their states is a poor strategy for detecting overloads, since they may tend to process such requests primarily when lightly loaded (simply because a heavily loaded program may be too busy to update the database of sensor values or to notice incoming polling requests). Thus we might obtain an artificially low measurement of load, or one in which the servers are all sampled at different times and hence the different values cannot really be combined.

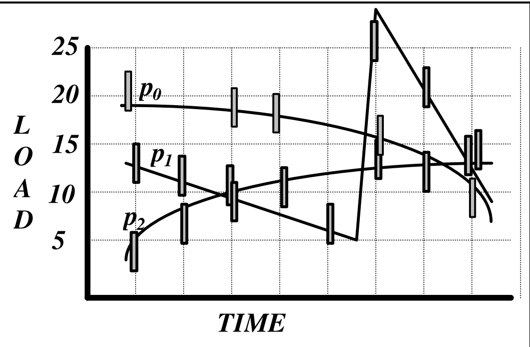


Figure 23-3: Sampling using a periodic process group. Here we assume that a wrapper or some form of process-group oriented real-time mechanism has been introduced to coordinate sampling times. Some samples are missing, corresponding to times at which the load for the corresponding process was not well defined or in which that process was unable to meet the deadlines associated with the real-time mechanism. In this example the samples are well synchronized but there are often missing values, raising the question of how a system can calculate an average given just two out of three or even one out of three values..

This point is illustrated in Figure 23-3, where we have postulated the use of a high-precision clock synchronization algorithm and some form of periodic process group mechanism that arranges for a high-priority load-checking procedure to be executed periodically. The sampling boxes are reduced in size and tend now to occur at the same point in time. But notice that some samples are missing. This illustrates yet another limitation associated with monitoring a complex system: certain types of measurements may not always be possible. For example, if the "load" on our servers is computed by calculating the length of a request queue data structure, there may be periods of time during which the queue is locked against access because it is being updated and is temporarily in an inconsistent state. If a sampling period happens to fall during such a lock-out period, we would be prevented from sampling the load for the corresponding server during that sampling period. Thus, we could improve on our situation by introducing a high priority monitoring subsystem (perhaps in the form of a Horus-based wrapper, which could then take advantage of real-time protocols to coordinate and synchronize its sampling), but we would still be confronted with certain fundamental sources of uncertainty. In particular, we may now need to compute "average load" with one or even two missing values. As the desired accuracy of sampling rises the probability that data will be missing will also rise; a similar phenomenon was observed when the CASD protocols were operated with smaller and smaller values of Δ corresponding to stronger assumptions on the execution environment.

In the view of the monitoring subsystem, these factors blur the actual events that took place. As seen in Figure 23-4, the monitoring system is limited to an approximate notion of the range of values that the load sensor may have had, and this approximation may be quite poor (this figure is based on the samples from Figure 23-2). A higher sampling rate and more accurate sensors would improve upon the quality of this estimate, but missing values would creep in to limit the degree to which we can guarantee "accuracy".

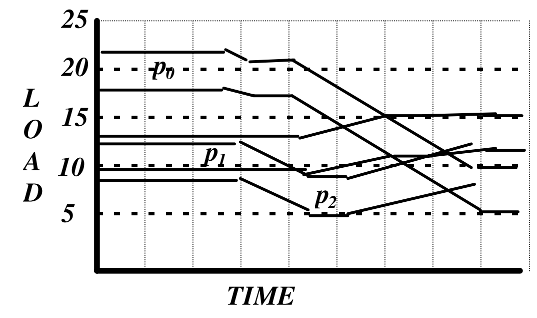


Figure 23-4: By interpolating between samples a monitoring system can approximate the true behavior of a monitored application. But important detail can be lost if the sensor values are not sufficiently accurate and frequent. From this interpolated version of Figure 23-2 it is impossible to determine that the database system briefly became "overloaded" by having two servers that both exceeded loads of 15.

With these limitations in mind, we now move on to the question of events. To convert sensor values into events, a monitoring system will define a set of *event trigger conditions*. For example, a very simple overload condition might specify:

trigger overload when avg(s∈ db_servers: s.load) > 15

Here, we have used an informal notation to specify the aggregate consisting of all server processes and then computed the average values of the corresponding "load" fields. If this average exceeds 15, the overload "event" will occur; it will then remain disabled until the average load falls back below 15, and can then be triggered by the next increase beyond the threshold. Under what conditions should this event be triggered?

In our example, there was a brief period during which 2/3 of the database servers exceeded a load of 15 and during this period, the "true" average load may well have also crossed the threshold. A system sampled as erratically as this one, however, would need to sustain an overload for a considerable period of time before such a condition would definitely be detectable. Moreover, even when the condition is detected, uncertainty in the sensor readings makes it difficult to know if the average actually exceeded the limit: one can in fact distinguish three cases: definitely below the limit, possibly above the limit, and definitely above the limit. Thus there may be conditions under which the monitoring system will be uncertain of whether or not to trigger the overload event.

Circumstances may require that the interpretation of a condition be done in a particular manner. If an "overload" might trigger a catastrophic failure, the developer would probably prefer an aggressive solution: should the load reach a point where the threshold might have been exceeded, the event should be raised. On the other hand, it may be that the more serious error would be to trigger the overload event when the load might actually be below the limits, or might fall below soon. Such a scenario would argue for the more conservative approach.

To a limited degree, one could address such considerations by simply adjusting the limits. Thus if we seek an aggressive solution but are working with a system that operates conservatively, we could reduce the threshold value by the expected imprecision in the sensors. By signaling an overload if the average definitely exceeds 12, one can address the possibility that the sensor readings were too low by 3 and that the true values averaged 15. However, a correct solution to this problem should also account for the possibility that the value might change more rapidly than the frequency of sampling, as in Figure 23-2. Knowing the maximum possible rate of change and "assuming the worst", one might arrive at a system model more like the one in Figure 23-5. Here, the possible rate of change of the various sensor values permits the system to extrapolate the possible envelope within which the "true" value may lie. These curves are discontinuous because when a new reading is made, the resulting concrete data immediately narrows the envelope to the uncertainty built into the sensors themselves.

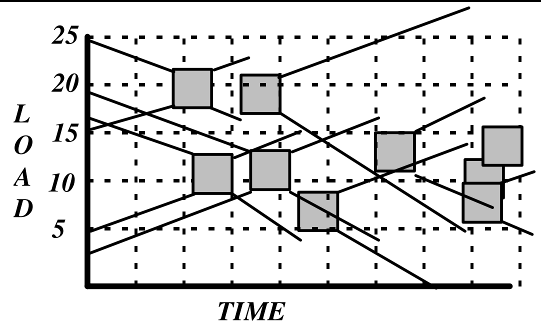


Figure 23-5: By factoring in the possible rate of change of a continuous sensor, the system can estimate possible sensor values under "worst case" conditions. This permits a very conservative interpretation of trigger conditions.

Marzullo and Wood have developed a comprehensive theoretical treatment of these issues, dealing both with estimation of values and performing imprecise comparisons [Woo93]. Their work results both in algorithms for combining and comparing sensor values, and the suggestion that comparison operators be supported in two modes: one for the "possible" case and one for the "definite" one. They also provide some assistance on the problem of selecting an appropriate sampling rate to ensure that critical events will be detected correctly. Because some applications require rapid polling of sensors, they develop algorithms for transforming a distributed condition over a sensor aggregate into a set of local conditions that can be evaluated close to the monitored objects, where polling is comparatively inexpensive and can be done frequently. For purposes of this text we will not cover their work in detail, but interested readers will find discussion of these topics in [Woo91, Mar90, MCWB91, BM93].

23.3.2 Actuators

An actuator is the converse of a sensor: the management system assigns a value to it, and this causes some action to be taken. Actuators may be physical (for example a controller for a robot arm), logical (a parameter of a software system), or may be connection to abstract actions (an actuator could cause a program to be executed on behalf of the management system). In the context of SNMP, an actuator can be approximated by having the external management program set a value in the MIB that is periodically polled by the application program. More commonly, a monitoring program will place a remote agent at the locations where it may take actions, and trigger those actions by RPC to it.

Thus, an actuator is the logical abstraction corresponding to any of the *actions* that a control policy can take. The policy will determine what action is desired, and then the action is performed by placing an appropriate value into the appropriate actuator or actuators. Actuators can be visualized as buttons that can be pushed and formfill menus that can be filled in and executed. Whereas a human might do these things through a GUI, a system control rule does so by "actuating" an actuator or a set of actuators.

Whereas the handling of faulty sensors is a fairly simple matter, dealing with potentially faulty actuators is quite a bit more complex. Marzullo and Wood studied this issue as part of a general treatment of aggregated actuators: groups of actuators having some type. For example, one could define the group of *run a program* actuators associated with a set of computers (in practice, such an actuator would be a form of RPC interface to a remote execution facility: by placing a value into it, the remote execution facility could be asked to run the program corresponding to that value -- e.g. the program with the same name that was written to the actuator, or a program identified by an index into a table). One could then imagine a rule whereby, if one machine was unable to run the desired program, some other machine would be asked to do so: "run on any one of these machines", in effect. Rules for load-balanced execution could be superimposed on such an actuator aggregate. However, although the Meta system implemented some simple mechanisms along these lines, the author is not aware of any use of the idea in commercial system.

Marzullo and Wood have noted that in the limit, fault-tolerant actuator aggregates will need to run a protocol much like Byzantine consensus, an analogy that brings to mind the large body of research that has been conducted on what are called *embedded systems*, in which a control program is placed close to a hardware subsystem and used to manage or control some form of external process. However, although this is a rich area of literature, brevity will prevent us from treating it in the current text.

In most commercial monitoring and management systems, actuators are limited to very simple tasks, such as executing a program, changing a priority for a scheduling algorithm, and so-forth. In effect, the actuator model is used to link the management policy to the external world, but not to superimpose any sort of more sophisticated abstractions upon it. The topic thus remains an intriguing area for future study.

23.4 Reactive control in Distributed Settings

Having addressed the issues associated with modeling a distributed system and with interpreting its sensors, we turn the problem of *reactive control* that arises when triggered events are used to drive management policies.

Marzullo and Wood recommend that management policies be viewed as a database of control rules, which are "bound" to system components and become active when those components are active. Thus, a policy for managing a database server would be instantiated once for each database server in the system, and each rule would manage its own server as long as that server keeps running. A policy for the aggregate of database servers would be instantiated the first time a database server is started, and would remain active as long as there are one or more servers in the system.

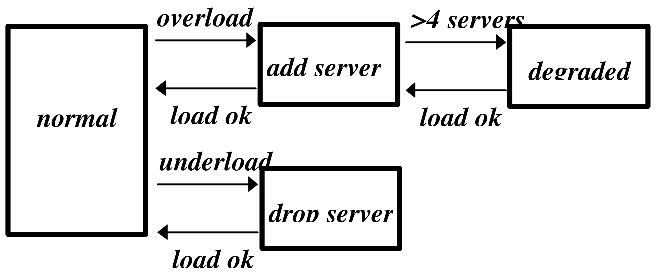


Figure 23-6: State machine for database server management. In the normal mode of operation, if overload is detected a server is added. However, if this would cause the server group to exceed 4 servers, the system degrades its quality of service until normal loading is restored. If the system is "underloaded" for a period of time, a server is dropped.

Each of these policies is described in the form of a script giving the control rules to use for the corresponding component and the conditions under which those rules should apply. Policies are in this sense similar to a *state machine*. Each state defines a set of events for which it is monitoring, and if the event occurs, the machine transitions to a new state and takes some management action.

Thus, for our database server example, we might define a policy for managing the aggregate of database servers whereby an *overload* event, detected in the *normal* state, causes the system to add servers up to a maximum of 4, after which it might move to a *degraded operation* state, remaining in this state until a *load ok* event is detected (Figure 23-6). As it moves from state to state, such a policy might write values to actuators. For example, the rule shown here requires a means of launching and shutting down servers, and for placing the group in a degraded mode. The actuators for the first of these cases would be an *execute process* actuator on the least loaded machine, which can be picked by a simple operation on the *server_nodes* relation. To shut down a process one would probably send that process a termination message, or perhaps even sending it a SIGTERM signal. To switch the servers into a *degraded query* mode, one would probably want an out-of-band signaling mechanism, since the message queues may be congested when this condition arises. Thus, a degraded query actuator might simply be a bit that can be set in the MIB associated with the database server; the server itself would check this bit periodically and switch in and out of degraded query mode accordingly. This illustrates that the concept of an actuator needs to be interpreted in a very flexible way.

Approaches such as this raise a number of hard problems. One major issue is to pick an appropriate language in which to specify these rules. Marzullo and Wood proposed two such languages: a high-level language that looked similar to a database query language, and a low-level one into which these high level language rules could be compiled; the low-level language was similar to the postscript language used for printers. Other work in this area has focused on popular command script languages, such as Perl. Questions of the appropriate execution environment for a rule, and methods by which a rule can interact with the system configuration, are all potentially difficult. For the purposes of this chapter, however, we will not delve into the details of these language proposals, viewing the topic as one that remains open for further research, and that is somewhat outside of the primary scope of this textbook.

23.5 Fault-tolerance by State Machine Replication

Having reduced our problem to one of interpreting a state machine, two classes of issues arise. The first is associated with the efficiency of our monitoring mechanism. If rules are evaluated at some computer which is remote from the place where the managed components are running, polling the sensors may become a costly source of overhead. Marzullo and Wood solve this problem in their Meta system using an elegant technique: they compile conditions into "local triggers" that can be evaluated entirely local to the monitored or managed component. Only potentially significant events need to be communicated to the state machine that is actually interpreting the rule. Thus, if we are indeed concerned about load, we might find that the current average load is 10. If we ask the components to report their load to us if it rises by more than 1.333, we will certain learn of any condition in which the load average has reached 15. Yet we can do so without reporting every intermediate value through which the servers pass, and indeed no communication may be needed at all in the normal case.

A second concern is that of ensuring the availability of the management environment. Using a process group (Meta was implemented over the Isis Toolkit), it was possible to implement this approach fault-tolerantly. Events reported by a sensor to the policy were sent as *abcast* messages to a process group running replicas of the policy state machine. In this manner, the control policies needed to manage the system could remain operational even if some of the platforms on which the control software was running failed. Marzullo and Wood argue that availability is important in management systems: the control policies often tell a system how to reconfigure after a failure, and hence must be running if the system itself is to reconfigure after a failure. One implication of this observation is that embedding monitoring and management functionality directly into the application itself may be a good idea, when it is practical to do so. In this manner, if any portion of the application survives a failure, so will the corresponding portion of the replicated management framework; one can then design the system so that if a sufficient amount of it remains operational, the management policies needed to recover will also be available and can be executed to reconfigure the system. Such a system can be said to be "self managed."

23.6 Visualization of Distributed System States

A benefit of system management is that the existence of a system model and of a database of sensor values can support elegant visualization tools. Such tools go beyond the scope of this text, hence we will not discuss the challenges of building them here. However, it worthwhile to pause and stress the value of such a management interface. When building a reliable distributed system, one sets out to identify the potential causes of unreliability and to counter them systematically. A management infrastructure with suitable visualization tools will let an operator quickly and effectively understand the cause of such problems that may slip through the original hardening process, intervening to correct problems that arise at runtime and providing invaluable information for improvements of the systems self-management policies and technologies.

23.7 Correlated Events

It is common to assume that failures, recoveries, and other events that require management actions are relatively infrequent and independent within a system. This avoids a number of thorny issues that would be raised by the potential for a sudden "storm" of events triggered when several things go wrong at the same time in a distributed system. We haven't talked about the concurrent execution of a set of rules (linearizability or serializability of the actions taken might be a sensible objective), and there is a broader issue of whether or not one even wants the same policy to be used while a system is operated in a "routine" manner as in the case where multiple events occur simultaneously.

Unfortunately, in a complex system, this assumption of independent failures and events is rather unlikely to be satisfied. A power outage, for example, may cause half the nodes in a cluster computer to fail, and the management rules for the applications on that cluster will now be triggered simultaneously for many components of the surviving parts of the system. Moreover, if a system has complex dependencies between its components, the failure of a single component can cascade, resulting in a storm of secondary events and secondary failures that are in fact symptoms of the original event. Attempting to correct these secondary problems will probably not be successful unless the core problem is identified and resolved.

For example, suppose that our database servers make use of the file servers. If a file server crashes, the database servers may hang until the file server subsystem has reconfigured itself to reallocate workload for which the failed machine was originally responsible. During this period it is likely that we will detect an "overload", but the overload is in fact purely a consequence of the file server reconfiguration. The correct management policy is to *inhibit* the overload condition for database servers while file server reconfiguration is underway. Yet, to understand this one needs to start by characterizing the dependencies of components upon one another, and then to pass from such a characterization to one giving dependent failure modes of the system as a whole. From this, it becomes clear that the correct way to deal with *independent* overload of the database server may be to add servers or to move to a degraded query mode, but that the best way to deal with *dependent* overload is entirely different; perhaps such a condition should be addressed by telling the client systems to stop submitting queries!

Traditional hardware engineering involves the development of *fault trees* by which such conditions can be characterized and appropriate solutions for diagnosis of events that occur and intervening to respond can be charted and implemented. At the time of this writing, there has been little research on the use of analogous techniques in complex distributed systems, and the issues that would arise if one were to provide software support for a fault-tree analysis and problem resolution remain unknown. Looking to the future, it is likely that the development of increasingly critical distributed systems will require that we develop increasingly sophisticated fault models, including techniques by which fault trees can be "compiled" into policies. The sorts of management policies described above would then be typical of those that might be used in a normal mode of operation, while other mechanisms design to deal with correlated failures or failures triggered by complex dependencies within the system would be addressed through other policies that might operate in very different ways. This area appears to be a potentially fertile one for future study.

23.8 Information Warfare and Defensive Tactics

In the introduction of this textbook we discussed the potential emergence of aggressive threats that could include outright attack on the critical information assets on which the military or major segments of industry depend. One can confront such problems through security measures such as firewalls and access authorization. However, there is always the possibility that the security techniques will themselves depend upon an underlying technology that can be compromised, leaving the system defenseless.

To take the most obvious example, many distributed systems use firewalls for protection, mounting their primary defense against attack at the points where communication interfaces from the system to the outside world are placed. Yet such systems are rarely located at a single physical site, and may have complex internal network topologies that include communication lines provided by telecommunications or internet service providers. These lines are not perceived as connections to the external world and hence are often unprotected.

In practice, however, such a perspective places considerable trust in at least one external technology: that used to implement the dedicated lines themselves. If the internet provider or telecommunications company is itself compromised, it may be that these dedicated lines can be tapped or even accessed freely by intruders. Such an intruder will now have circumvented the firewall protection and will find him or herself free to act within the protected system itself. The consequences of such a break-in can be very serious. Early in this textbook we noted that the complexity of telecommunications systems and internet architectures is risely sharply. Thus the opportunities to compromise the underlying architectures used to support these dedicated lines are also rising. Should the service provider itself be compromised, breaking into the network of a large organization may be much easier than doing so through the firewalls it has placed at its connections to the external world.

Recall the example of an NFS system used within a firewall, from Section 19.1. When an NFS is operated within a firewall, we saw that it is typical to disable its security mechanisms to improve performance. But an intruder may now be able to spoof in a manner that would trick the NFS system into granting requests by pretending to be a legitimate user on a legitimate machine. We saw that, once through the firewall, there is little that an intruder into an unprotected NFS environment would be prevented from doing: files can potentially be read, rewritten, and dates and times even reset with relative ease. Our challenge is to erect barriers that would convincingly detect and protect against such events.

The steps to repelling aggressive attacks, whether they threaten critical military assets or merely seek to falsify financial transfers from a large bank, will be to *detect* the intrusion, *quantify* the event by identifying the methods used and the system components under attack, and *respond* by modifying the behavior of the system, shutting down components, or disabling access points until the system itself can be made tolerate to the form of attack in question.

For purposes of *detection*, a "DIW" (defensive information warfare) monitoring system will typically rely upon audit trails that trace sensitive operations and permit comparison between patterns of access and authorized or "typical" patterns of activity. Such trails monitor classes of events that are of potential concern, such as file open operations, and are then filtered through programs that maintain historical information regarding permissible and typical patterns of access. ("Vice President Smith is permitted to access all account records but in practice very rarely accesses any records except those for customers with whom she works directly", or "Admiral Walker normally reads and sends memos associated with naval operations in the southern Atlantic"). When violations of these normal patterns are detected, responses can be initiated, and if an intruder successfully penetrates a system, the audit trail will later permit the penetration to be localized and further problems to be prevented.

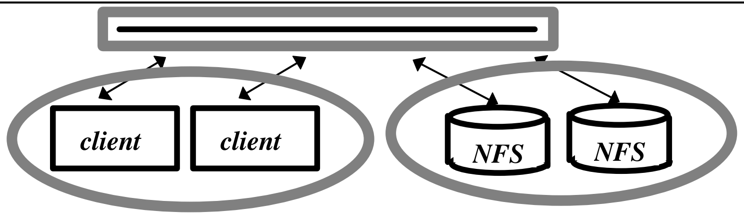


Figure 23-8: Application-specific wrappers surround various classes of system components in this hardened environment. The wrappers (gray) can intervene in lightweight ways that are potentially less costly than strong security (encryption and authentication technologies), and yet can represent a significant barrier to intrusion. One can imagine a great variety of possible techniques that could be incorporated into such wrappers; for the NFS case they might maintain profiles of typical behavior which would constitute a database exploited by the wrapper group to detect unusual access patterns and to intervene. Here we have wrapped the clients, database servers and network itself. Each wrapper might represent an intervention at a different place in the technology (the router of a network, the file I/O library on a client workstation, the network interface of an NFS server), and each might function in a different way.

In the case of our NFS example, such a policy would involve intercepting file *open* requests somewhere on the path to the NFS server and comparing them with normal file access profiles. Suspicious patterns could be signaled to an operator, or simply discarded. Vice President Smith would merely issue some sort of command to disable the protection system and then resume her unusual pattern of activity; an attacker would be much less likely to surmount this obstacle. These sorts of *application specific* wrappers would substantially raise the barrier to an attack on the system, without necessarily paying the cost of general purpose encryption or authentication, which can be high.

Wrappers can be a valuable tool for compiling audit databases and for filtering messages in this manner. In general, such technologies are added *after the fact* to a system that will have been built *without knowledge that it would later be wrapped*. Thus even if the system is compromised, it may not be able to anticipate and compensate for the protective mechanisms that will be used against it. The wrapper enjoys the advantage of “stealth”, provided of course that its performance impact is minimal. A wrapper can be used to control the actions of the wrapped component, to oversee those actions non-intrusively, or

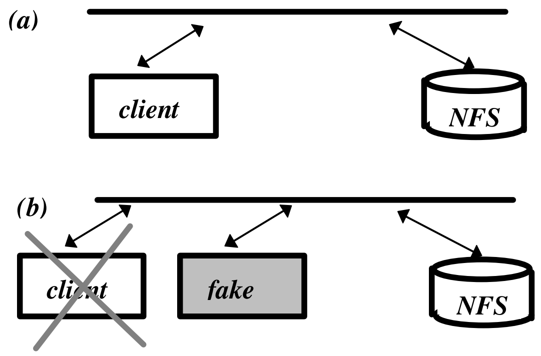


Figure 23-7: (a) The NFS protocol, when security is not enabled, involves trust in the IP address of client systems and the user/group id's presented by clients as part of each request. (b) Should an intruder break into the network it may be possible to mascarade as a legitimate client by killing some existing node with a poison pill or some other tactic, and then sending faked NFS packets to the serve. In this scenario, the NFS is unable to protect itself and may provide essentially unrestricted access to the files it manage. Wrappers around the client, the network, or the NFS unit could filter messages against a behavioral profile that would permit such behavior to be detected or prevented.

to build up profiles of typical behavior with which a new behavior can be compared. Moreover, a single system can support more than one form of wrapper for the same purpose, selecting the wrapper used randomly, and arranging that a countermeasure to one wrapper will be detected as unusual behavior by another wrapper. Even an insider might have trouble overcoming such an approach. Work on wrappers for this purpose could be of great value in DIW management applications.

With regard to *quantifying attacks*, the clear challenge is to find ways of synthesizing patterns out of information gathered from physically dispersed components, and to do so without excessively perturbing the performance or correctness of the monitored subsystems. One approach is to timestamp audit records and to send them to a central site for analysis; such a technology will depend upon bandwidth to the central site and heavy computing power to build up a centralized model of the decentralized behavior of the system from which patterns can be extracted. A second, distributed, approach would draw upon the natural process-group structures present in reliable distributed systems and visible through the system model. In this approach the wrappers for a set of components would be programmed to detect unusual patterns of behavior in a distributed manner, operating close to the components themselves and basically "compiling" a pattern into its local "sub pattern". With the growing availability of high performance process group support and clock synchronization technologies it may be increasingly practical to use such approaches, which have the benefit of imposing much reduced loads on the communication system and exploiting decentralized and hence coarsely parallel computing power. Again, much research will be needed in this area.

Turning to the problem of *response* when an attack has been detected, it is appealing to consider the introduction of wrappers for the purpose of placing selective firewalls close to groups of components. Suppose for example that every meaningful group of components was ringed by a firewall specifically tuned to the patterns of behavior seen in members of that group, and capable of filtering incoming and outgoing messages under criteria associated with the message source and destination. Such a system could be visualized as having a potentially large number of superimposed firewalls intersecting in various ways.

Any given computation would now generate communication that traverses one or many of these firewall boundaries, all implemented by wrappers in a lightweight manner. Now, if a threat was detected, the firewalls could be dynamically reprogrammed to inhibit communication to and from the compromised system components, or to restrict such communication in a way that is believed sufficient to filter out the compromised traffic. Whereas the intruder in a conventional system needs only to violate a single firewall to gain free run of the system, the intruder in such a multi-level security containment environment would confront obstacle after obstacle, with each layer of firewalls implementing its own containment policy based on the normal patterns of communication for that subsystem or application. It seems likely that this would result in significant improvements in the robustness of critical systems against attacks.

Recalling our NFS example, one could imagine a solution in which the NFS traffic passes through filters that do simple sanity checks on the origin and destination of packets and the degree to which such packets correspond to the expected network topology (Figure 23-8). By embedding such functionality into a packet router or the low levels of the network communication software used on the NFS itself, one could protect against many forms of NFS attacks even when the NFS security policy is not in place. Of course, protection against an attack requires that the attack be anticipated as a possible threat and that a suitable response be formulated, but at least the tools are available to potentially protect against such threats if the will exists to begin to exploit them. With few changes to the methodology by which distributed systems are typically constructed, it may be possible to introduce monitoring and protection mechanisms after the fact that would make such systems far more robust against a great variety of possible threats and attacks. Moreover, by customizing the behavior of a technology, such as a firewall, to match it closely to the normal mode of operation of the protected system component or components, the complexity of breaking through that firewall can be greatly increased.

To summarize, if we face a growing threat of outright attacks on critical technologies and systems, the good news is that we also have a growing technology base on which to draw for solutions to the problems of detecting those attacks (or explaining them in a post-mortum analysis) and for developing flexible and powerful responses. Wrapper technologies may permit the stealthy introduction of the resulting solutions into applications and subsystems that were not designed with such defensive abilities in mind, and whose designers may not even be trustworthy: to the degree that the wrappers can be trusted, the resulting wrapped application may be trusted for specific purposes. While much work remains to be done before such a vision can be called a practical reality, it is clear that we have a growing number of powerful tools for use in the task. More research is needed, but we could already harden systems against a great number of simple attacks that many systems are now unable to defend against.

23.9 Related Readings

This chapter drew primarily from [Mar84, Mar90, MCBW91, Woo91, BM93].

24. Cluster Computer Architectures

A new generation of hardware is emerging in support of client-server applications: the so-called *cluster server* architectures, in which a collection of relatively standard compute nodes and storage nodes are interconnected using a high speed communications device, often similar to a fast CSMA or ATM interconnect. Such cluster computers can be viewed as smaller cousins of the massively parallel supercomputers that emerged in the 1980's and now dominate the supercomputing market. However, they are more similar in many respects to distributed systems. In this chapter we look at the similarities and differences and ask what impact these have on the solutions needed for reliability.

The specific definition of a "cluster architecture" remains elusive at the time of this writing: many vendors offer computers that might be considered clusters, yet the term is sometimes applied to what might more property be termed a multiprocessor, while some cluster computing systems are characterized by the press as coarse-grained parallel machines. Some vendors have applied the term cluster to primary-backup systems that employ trivial (and potentially incorrect) failover technologies and that lack any tools for maintaining the consistency of the backup and the primary. Thus the term is not merely disputed, but is also potentially misused.

For the purposes of this chapter, a cluster architecture is one based on:

- *Standard components*. A basic idea of cluster architectures is to take advantage of the price advantage associated with mass production of PC's and workstations, hence clusters are typically built from completely standard components of these sorts, albeit without displays and perhaps packaged in a non-standard chassis. Typically cluster computers support at least two sorts of components: compute nodes and disk nodes. Some may also support network interface nodes and other sorts of special-purpose attachments.
- *High speed interconnect*. The internal communications "bus" that connects the components is typically message-oriented and may be based on a standard ATM or high speed CSMA connection device.
- *Cluster infrastructure*. This includes the chassis and cabling connecting components, the power supplies and communication adapters, and so forth.
- *Management subsystem*. A cluster is more than a rack-mounted pile of components, and the management subsystem plays what is often the dominant role in creating the abstraction of an integrated entity.
- *Cluster API*. This is a debatable term, but refers to the collection of system interfaces, available to programs and system administrators, by which applications can take advantage of the clustered nature of the system. The API will typically include ways of determining the set of nodes on the cluster and monitoring their states, launching applications on nodes and monitoring them while the run, accessing the management functions of the cluster, and so forth.

Perhaps the most important attribute of cluster is that a separate copy of the operating system runs on each node. This is in contrast to more integrated multi-node computers in which a single copy of the operating system controls many nodes at the same time: such systems are best understood as true multiprocessors, and will typically include special purpose hardware supporting shared memory and permitting the operating system to control the multiple processors comprising the system. At the other end of the spectrum, a cluster is distinguished from a parallel computing system by having fewer nodes (normally 4-32, with configurations of 8 to 16 processors being common), and running a full operating system on each node, unlike the very stripped "slave" operating systems often used on nodes of parallel computers.

Our interest in this text is in reliability, and hence it makes sense to focus on cluster computing systems intended for reliable or mission critical applications. These include the Stratus RADIO computer (the author played a role in developing that system and hence knows it best), Compaq's Proliant line of cluster PC systems, the DECsafe Available Server Environment [DEC95], SUN Microsystem's Solaris MC system [KBMS95], other products from HP, Microsoft, Tandem, and certain configurations of IBM's SP product line. Cluster computers that have a reliability concern must address many of the same replication and consistency issues we have considered in previous chapters; it makes sense to ask if we cannot adapt our solutions to a cluster computing environment. A more complete treatment of cluster computing can be found in [Pfi95].

24.1 Inside a High Availability Cluster Product: The Stratus Radio

The Stratus Radio (Reliable Architecture for Distributed I/O) product is a good example of a cluster computer designed for high availability applications. Radio consists of a rack-mounted collection of between 6 and 24 compute and storage nodes. An individual "rack" of Radio nodes can contain up to 6 of these nodes, and will also have 2 communications nodes, its own power supplies, and so forth.[21]

The compute nodes are dual processors based on a standard PC architecture, with substantial on-board memory, small swap disks, and a fast clock cycle time. Each runs a standard PC operating system: UNIX or NT, and communicates with the other nodes over the internal communications network using completely standard communication protocols such as TCP/IP or the ISO protocol stack. Each node has its own IP address and runs its own copy of the operating system. These nodes are thus free-standing computers that share a hardware environment with one-another but are capable of independent operation.

RADIO disk nodes are based on standard high speed disk technology, but run a non-standard operating system that functions as a form of front-end to the network. This permits Radio nodes to share disks (much like *n-way* multi-ported disks on a more conventional computing system), and software functionality for mirroring segments of disks is supported for high availability. The possibility of attaching other types of highly available disk technologies, such as RAID or SSA disk clusters, is also present. Any compute node can access any disk node within the cluster.

[21] We describe this system with some hesitation; as one of the developers of a commercial product, this author is sensitive to the perception of a possible conflict of interest. To avoid such issues, the discussion is limited to the RADIO product on which the author worked, and no attempt is made to compare this product with other products. Similar considerations entered into the review of distributed programming environments and transactional systems in previous chapters.

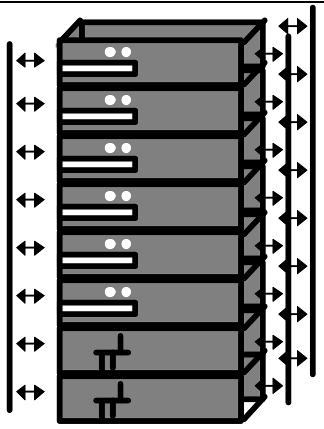


Figure 24-1: The Radio cluster marketed by Stratus computer was designed for high availability. All components are duplicated for redundancy, and the compute and storage nodes are hot-pluggable without disrupting availability of critical applications. Illustrated is a 6-node cluster of compute or storage nodes, the management network (black, on the left), and the dual internal networks (red and green, right). The bottom 2 nodes contain the network hardware, interfaces to external networks, power supplies, and other cluster management technology. Slots on the compute and storage node are for high capacity floppy disks.

Radio communication is over an internal bus consisting of a fast ethernet interface or an ATM switching interconnect. The network adapter nodes handle interconnection of multiple Radio systems into a large, scaleable system, as well as interconnection of the Radio system with external networks connected to client systems. For availability reasons, the internal network is a dual one and the system can function transparently with either network out of service; similarly, dual connections to the client systems are assumed. Radio also has a redundant power supply, and can function normally with one or both power supplies in operation, and has an internal management network which is used to monitor the status of the nodes and to provide the system administrator with control of the "operator console" interfaces to the PC's that comprise the cluster. Such access can be from a local console or remotely over a communications interconnect from a central management site that can be physically far from the Radio as a whole.

An interesting property of the Radio management network is that it can detect a node going off-line (because of a self-diagnosed failure or because it was removed) or that has just gone online within milliseconds. This is in striking contrast to the situation in a general distributed environment where the same event may not be safely detectable until an extended period of pinging has occurred, because of the relatively high frequency of communication outages and minor partitioning failures. Thus a type of failure detection that can take many seconds in a general setting is reduced to perhaps one hundredth of a second in this architecture.

All aspects of the Radio architecture were designed with the avoidance of single-points of failure in mind. Every component can be "hot swapped", meaning that each module is capable of being removed for service without disrupting normal operation of the remainder of the cluster, and plugged back in after servicing or upgrading is complete. Moreover, the technology includes provisions for upgrades to new versions of the software or hardware, also without disrupting continuous operations.

The software used with a cluster product such as Radio depends a great deal upon the goals of the application designer. Radio itself is designed to be self-managed and includes a software infrastructure that exploits the Isis Distributed Computing Toolkit for this purpose. The application manager uses the general approach we discussed in Chapter 23, and operates by executing scripts written in the Perl command language using a fault-tolerant state-machine approach.

Applications can be run on the cluster without change (in which case they benefit from the hardware availability and management aspects of the platform but must be "placed" on the nodes by hand and will fail if the node on which they are running fails). Alternatively, an application can be managed by the Radio application management technology, which involves developing management scripts indicating the aspects of the application state to monitor and what actions to take if a problem is detected. A Radio system can also manage applications running external to it on client workstations or PC's.

A second way to benefit from a continuously available cluster is to use special purpose applications designed specifically to take advantage of it. The Radio system can run a variety of actively replicated software systems based upon the Isis Toolkit technology, including a continuously available database system, a continuously available version of the UNIX NFS server technology, a message bus product supporting a publish/subscribe interface; all of these technologies are designed to gain performance as the size of the cluster is increased ("scaleable parallelism").

Of course, there will always be applications that require the development of new continuously available servers. For this purpose, the Radio user can draw upon the same active replication techniques presented earlier in this text. In particular, Orbix+Isis is available for the platform, offering a simple way to exploit replication and load-balancing in new applications, and Horus has been ported to the platform, providing a flexible and reconfigurable process group technology base for it.

The author is less familiar with the details of some of the other cluster products available from other vendors, and will not attempt to present any form of comparison of these products here. The experience of participating in the Radio design, however, made it clear that there are many ways a cluster server can fail to be highly available, adequately manageable, or adequately functional. Striking the correct balance is considerable challenge. There are no standard cluster architectures at this time, and existing cluster products employ very varied hardware and software availability strategies and management technologies: some are clearly deficient, while others are very impressive while also reflecting different design tradeoffs than the ones that were made in developing Radio. The developer of a critical application must therefore approach the selection of a cluster server architecture with considerable care, evaluating all aspects of the system design relative to the reliability goals of the application before selecting an appropriate server.

24.2 Reliability Goals for Cluster Servers

Cluster servers, if appropriately designed, have special properties that can be exploited in the communication and management software of the system. The motivation for these properties is best understood by looking at some of the goals that typical application developers would be expected to have. These include the following:

- *Continuous availability*. Not all clusters are intended for critical applications, but in those that are, continuous availability is typically an important design goal. This objective has implications at all levels of the cluster architecture, including the hardware and interconnection technologies used, the cluster management technologies, the operating systems functions offered to users through the "cluster API", and the technologies used to implement applications on the cluster. Indeed, availability goals extend from the cluster into the surrounding network, because of considerations stemming from the environmental dependencies we identified as early as Figure 1-6.
- *Rapid failure detection*. This property is required if applications are to "failover" quickly when a node crashes or is removed for servicing.
- *Ease of management*. In critical settings, it is important to be able to rapidly localize sources of failures in order to fix the faulty system component or components, and to have easily used tools for upgrading system components and performing routine service. Cluster management environments must therefore include automated "self-management" solutions and must provide for remote management access to the cluster, over a modem or computer network from which a remote operator is working.
- *Single system image*. Although this goal is debatable, many clusters seek to provide the application developer with the illusion that the cluster is "really" a traditional single-processor running a traditional operating system such as UNIX. The danger in such an approach is that unless some form of transparent availability technology is provided to the application designer, the application itself

may not wish to view the cluster as a single system: placement of replicas of critical services on failure-independent nodes, for example, requires that it be possible to distinguish the nodes in the clusters and to be able to separate them based on hardware properties of the machine. Yet at the same time, the application developer would clearly prefer not to deal with artificial barriers to programming the cluster as a single machine. Cluster designers struggle with the resulting dual goals: transparency and a single system image on the one hand, and yet easy of developing an extremely reliable application on the other hand. It sometimes comes as a surprise to the vendor that the first may not imply the second!

Some aspects of the single-system image concept considered in modern cluster designs are task migration and automated load balancing, support for a coherent disk buffer cache on each node, direct paging and disk block access from the online memory of a different node, shared memory for processes running on different nodes, UNIX process id and "signal group" functionality across nodes, and a single IP address for the cluster as a whole. Each of these can be beneficial in some settings and yet undesirable in others.

- *Load balancing tools*. The major benefits of a cluster are the redundancy of hardware, permitting high availability, but also the possibility of exploiting scaleable parallelism. The application designer will be looking for tools that can assist in this process, by helping place new tasks on lightly loaded nodes.

- *Task migration tools*. Another debatable aspect of the cluster debate. Some cluster technologies seek to automate the migration of application programs from node to node: a desirable property from the perspective of load balancing and performance management, but potentially problematic if there are availability concerns that govern task placement, or if the functionality associated with this feature is extremely complex to support. Task migration support can vary from a process group mechanism whereby the user can migrate a task (by adding a group member and doing a state transfer, and then killing the old memory) to completely automated solutions that transfer the virtual memory pages of a running application and in this way migrate its full state from node to node. The bias of the author favors simpler solutions: full fledged transparent migration seems costly, non-standard, and poses the risk that application programs intended to operate independently for increased reliability will instead find themselves on the same node and hence exposed to correlated failures.

- *Scaleable parallelism tools*. Many applications will benefit from cluster architectures without requiring any changes to the code itself at all: for example, it may be that the application involves running large numbers of small processes (perhaps each command by a client on a workstation results in the execution of a process for that client on the server), and the placement policy for these processes is already sufficient to provide a benefit from the parallelism of the cluster. However, some servers will need to exploit parallelism in an explicit way, using tools like the process group computing tools we discussed earlier. For users who need such tools, the cluster should provide well integrated and highly "tuned" solutions that take advantage of any special properties of the hardware.

- *Online upgrades*. Many clusters are designed to permit "hot plugging" of hardware and software components, permitting repair of problems without disruption of the remainder of the cluster and offering a path whereby software revisions and bug fixes or entirely new versions of the operating system or other major components can be installed non-disruptively. Technology permitting such upgrades within software used by the application programs can be a difficult challenge.

- *Clock synchronization*. Many clusters have the capability of supporting highly synchronized clocks; not all take the step of actually providing this support.

- *Reliable communication*. Many clusters use communications hardware that can guarantee reliability if appropriately configured. Not all clusters offer this functionality to the application, however, even if the hardware is capable of supporting it: modern operating systems are designed to discard messages when they become heavily loaded.

- *Security features*. Within the cluster, nodes may consider themselves to be in a shared security enclave, trusting one-another in ways that would be unwise for applications on widely separated nodes in a general distributed environment.
- *No single point of failure*. Developers of reliable applications need to be sure that their system will not have any single point of failure. Cluster designers who develop highly available cluster products must adopt the same point of view to ensure that their products will be robust against all possible single failures, and that this property extends from the lowest level hardware components up to the software services that operate each node and that provide management functionality.

These properties make cluster architectures extremely interesting for a number of applications. As an example, recall Friedman's work on cluster support for a telephone system that uses a centralized switch to handle some classes of calls, such as "800" number calls. Potentially, there may be as many as 10^7 telephone numbers in the associated database, each record of which will encode the correct action to take if the corresponding number is called. If these records require 100 each to store, the database will contain perhaps 1 gigabyte of data. To guarantee rapid response time for incoming calls (typically a switch must route a call in no more than 100ms), one would like this data to reside in memory. However, the amount of memory here considerably exceeds what a typical computer can accommodate. A cluster architecture could easily provide memory-mapped access to a multi-gigabyte database, together with the fault-tolerance properties required for critical applications like telephone switching.

This review has focused on positive aspects of the cluster approach, but there are also negatives. A cluster offers the *potential* for improved performance and availability, but actually achieving scaleable parallelism in a specific application or implementing a highly available solution to a specific problem may be a very difficult undertaking. Moreover, a cluster will be physically located at a single site, and if the roof leaks or the power supply to the building is cut, the availability features of the cluster may be of little benefit. A distributed system that replicates critical state between sites that are widely separated could well survive such disruptions with little or no interruptions in service. Thus while it is safe to say that a cluster offers exciting options to the developer of a critical application, it is also clear that many issues must be addressed to achieve critical reliability, and clusters may or may not match the needs of a specific system.

24.3 Comparison with Fault-Tolerant Hardware

The reliability goals of a cluster computer are in many ways similar to those traditionally addressed through special purpose fault-tolerant hardware. In practice, however, the properties of cluster solutions are in some ways very different from what hardware fault-tolerance typically provides. Although this text has not considered hardware fault-tolerance up to now, it may be useful to point out some of these differences.

Hardware fault-tolerance is usually achieved by building some form of self-checking logic into the basic hardware modules of a computer and off-lining a module that fails. A backup module continues operations on behalf of the failed one. For example, the same company that developed the Radio system also manufactures a line of fault-tolerant computers called "Continuum". The machines in this product line use a paired architecture in which each processor or memory component is duplicated twice. Pairs of CPU's and memories form a single module: by comparing the memory contents and CPU actions continuously, a module can be taken off-line instantly if any problem is detected. The remaining pair of processors remains operational until the damaged module is repaired, at which point the system is restored to its fully symmetric fault-tolerance mode.

Hardware fault-tolerance of this sort offers protection against data integrity errors, such as undetectable multi-bit memory corruption, as well as protecting against other forms of hardware failure. Application programs are continuously available without even a "flicker" when a failure does occur. But if a software failure occurs or a component of the system must be upgraded, there is no alternative but to take the system off-line to accomplish the repair. A cluster system will lack data integrity checks and may not detect hardware errors, permitting serious data corruption to occur and to spread. Failures that *are* detectable may not be detected very quickly: the software techniques used to detect some classes of failures and to reconfigure to recover from them may take many seconds to complete. However, cluster systems d offer a plausible story on the issues of software failure and upgrade, through the use of process group technologies and state transfer. Moreover, in a redundant hardware system, one pays for the extra hardware needed to gain reliability, but gains no improvement in performance; a cluster will be less reliable but is also more cost-effective, since the processors can be kept busy doing non-identical tasks while they also back one-another up.

One sees from this that there is no single story in fault-tolerance. Fault-tolerant hardware has found important roles in process-control settings where a failure that results in system downtime will have immediate and costly implications. Telephone switching is a case of this sort, and represents one of the largest applications of modern fault-tolerant computers; other examples arise in air traffic control, avionics, machine control, power systems, and so forth. These "applications" can't tolerate any downtime at all, even briefly. Clusters are a compromise offering scalability at a price: weaker fault-tolerance potential and slower reaction time. But they are also inexpensive and very flexible. It is likely that each class of computers will play a major role in future reliable systems, but that the associated markets will be different ones.

24.4 Protocol Optimizations

We noted earlier that is appealing to consider the use of process-group and transactional technologies on cluster computing systems. When this is done, a number of optimizations can potentially be introduced that greatly simplify these technologies and also hold the potential for improved performance.

In particular, a cluster system may have an *accurate means of sensing failures*. For example, the Stratus Radio cluster has a management network that can rapidly sense hardware failures of nodes and can rapidly sense the introduction of a new node. A failure can thus be detected within a few milliseconds and the necessary reconfiguration actions triggered almost immediately. The operating system itself will detect most forms of application failures. Thus, with the exception of failures that leave the operating system alive but unable to do useful work (hung), and similar failures in the application programs, a cluster system may be able to use a greatly simplified system membership management protocol. Even if the GMS protocol is used without change, its performance will improve because false failure detections are not a concern, and the risk of partitioning is completely eliminated, permitting progress even when a small set of nodes are the only ones to survive a failure.

A cluster may have a *lossless message capability*. If present, such a capability can eliminate the need for positive and negative acknowledgements. In an architecture like that of Horus, this allows a major protocol layer to be omitted and offers potential performance benefits. Of course, the ability to exploit such a feature will depend upon the nature of flow control done in the application program: if data sources can outrun data sinks, message loss will be unavoidable.

A cluster may have *accurately synchronized clocks*. Such functionality would permit the introduction of real-time features and protocols, and enable processes to cooperate in performing time-critical or periodic tasks.

A cluster will typically have *extremely uniform hardware properties*. These permit protocols to be tuned for better performance. For example, with knowledge of the true point to point latency, the decision to send acknowledgment and flow control messages can be finely tuned, and timeouts can be closely matched to the true expected response time of a critical service. Notice, however, that the ability to exploit this feature will be limited by the properties of the operating system.

A cluster will have powerful *management capabilities*. These permit the development of rich and sophisticated software for system and application management, going well beyond what can be done in a distributed environment where the heterogeneity of platforms used forces a least-common-denominator approach in the management subsystem.

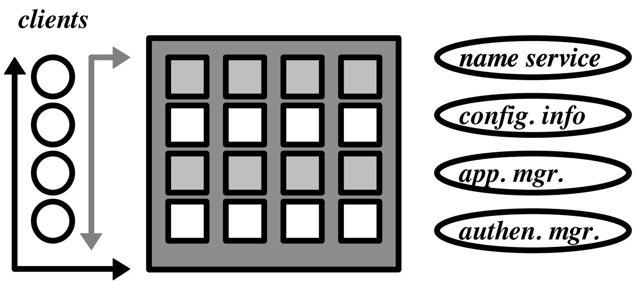


Figure 24-2: When a cluster is used to implement a highly available client-server system, information will be needed regarding independent failure properties of the component nodes (white and gray here), connections to clients (again red and green), and of dependencies that the application may have on other services in the enclosing network environment. There may be substantial simplification possible by assuming that a technology such as Horus is available on all nodes in the cluster and the client nodes: the resulting uniformity of infrastructure could go a long way towards creating the context within which powerful applications can be supported. However, additional services would also be needed, such as services for determining the configuration of the network and cluster, for programming management policies, and so forth. A "cluster API" would standardize these services and the interfaces by which they can be accessed, while also standardizing access to internal cluster communication and management mechanisms.

We thus see that although the same abstractions we considered in discussing reliability issues for general distributed computing systems may be extremely valuable in a cluster computing system, their implementation could be considerably affected by these properties. When using a layered protocol architecture such as the one favored in Horus, the implications may be as minor as the need or lack thereof for a particular Horus layer. More broadly, however, clusters offer the potential to exploit the uniformity of the system architecture to achieve better performance, improved management, and quicker response times.

The bad news, of course, is that the cluster is not a isolated entity. Most cluster systems are integrated into heterogeneous distributed environments, and the same software that runs on the cluster will often include components that reside outside of it, in client workstations or other components of the surrounding environment. We are now reminded of Figure 1-6, where some of the many technology dependencies present in a typical network were illustrated: the same dependencies will limit the effectiveness of our network as a whole, even if they have a somewhat reduced impact on the availability and reliability properties of the cluster itself. The designer who includes a cluster server into a large distributed system has not necessarily guaranteed high availability: a broader technology solution is needed. However, if a technology such as Horus can be used both on and off the cluster, a satisfactory response to this concern may emerge from the resulting software environment. The view will be one of software that is able to benefit from the local environment within the cluster, but that offers uniform features and a uniform programming model both on the cluster and off of it.

24.5 Cluster API Goals and Implementation

At the time of this writing in 1995, there has been considerable interest in the formation of a standards organization to develop and prototype potential standards for a cluster API. Such an API would provide the developer with a predictable set of primitives, available on clusters from multiple vendors, that could be exploited in building highly portable cluster applications. Such a solution would free the developers of "layered products" from dependencies on any single cluster vendor.

The most appropriate content for a cluster API is clearly debatable. From the review of technologies we have undertaken in this text, one can quickly see that if the cluster includes availability and reliability as an important goal, the API could potentially include every technology we have studied up to now. At a minimum, a cluster API should address the following:

- It should provide software access to system configuration and management interfaces. That is, the API should permit an application to learn the topology of the cluster, failure and performance characteristics of components, and access to the various status monitoring features provided by the hardware. Standards should be specified for such properties as load and status (operational or failed) of cluster components, for naming the most standard components (compute and storage nodes, communication paths), and for specifying performance properties (speed, latency, etc). Notice that some aspects of this could be addressed by mandating the use of SNMP or CMIP management information bases (MIB's) with standardized content; other aspects would not be resolved by such an approach.
- It should provide a simple architecture for exploiting the hardware features of the cluster. For example, it should address such issues as notification when system membership changes, communication among applications on nodes, and so forth.
- It should provide a uniform means of addressing applications. This is likely to require a two-level solution, in which the cluster as a whole can be accessed using a single IP address but its component systems have secondary "true" IP addresses.
- It should provide O/S level primitives for such aspects as launching an application on a node, monitoring the application as it runs, task migration (if supported), remote disk access (if supported), and so forth. The API should standardize these aspects even if they are considered "optional" depending upon the particular architecture used by a given vendor. Thus, there might be an API for shared memory across cluster nodes but also a way to determine whether or not the vendor actually supports this feature.
- It should deal easily with off-cluster applications and not be restricted to issues internal to the cluster.

This brief review points to what is clearly a very large topic. A system like Horus might make a good starting point on a cluster API for dealing with system management and communication issues. However, doing so would still leave important open topics. For example, existing distributed systems lack any way to interrogate the network about its topology and performance properties. Absent standards for this type of information, cluster applications would need to have their own proprietary configuration schemes, making them less easily portable and less standard. The problem is thus similar in many ways to the one tackled by the OMG group that developed CORBA or the OSF group that developed DCE, and it may require a similar scale of effort to resolve it.

24.6 Related Readings

Most of the information in this chapter was drawn from materials available from Stratus Computer Inc. For more general treatment of cluster computing, see [Pfi95]. Other clusters cited here include [DEC95, KBMS95].

25. Reasoning About Distributed Systems

The preceding sections have established a technology base with which existing distributed systems can be hardened, by selectively replacing components with versions that have been modified to withstand some class of anticipated threats, and with which new distributed systems can be constructed to be robust by design. Yet these tools are purely practical, in the sense that they represent software solutions to specific potential problems. Lacking is a broader conceptual framework that would let the designer characterize the behavior of a given system in a particular environment, reason about how that system might fail under various conditions, and predict the impact of a given robustness intervention on properties such as performance, security, and availability.

Early in the text we noted that it was not our intention to delve into the theory of distributed computing here, and this remains true now. Yet the ability to specify the behavior of a system and to reason about it, while representing a fairly “formal” activity, is not necessarily a theoretical one. In developing non-distributed systems, it is common practice to develop specifications that characterize the behavior of critical software modules, and to use these specifications as an input to large scale system design and visualization technologies.

In the remainder of this chapter, we review some of the options for describing a complex distributed system and specifying the technologies it uses to achieve robust behavior. Such specifications necessarily extend to the environment (upon which the system may impose requirements) and the application (which may be assumed to respect various constraints or to behave in specified ways). We then look at the issue of exploiting these specifications to reason about the large-scale behavior of a distributed system by using its small-scale behavior as an input to an overall system model. We also explore some recent work on automatic synthesis of robust distributed systems by plugging application-specific data handling methods into general purpose frameworks; such steps can be viewed as offering methods for compiling robustness into a system that is specified using a high level language. All of these areas are in need of additional study, and the sober-minded conclusion of our review will be that the technology base for reasoning about distributed systems is very tentative at the time of this writing. However, the longer term potential for the area is considerable.

25.1 Dimensions of the Systems Validation Problem

The distributed systems properties treated by this text can be understood as operating in a multi-dimensioned “space” of potential systems properties. Viewing a system in this manner can be a userful step in sorting out the critical properties of a specific application and beginning to demonstrate rigorously that they hold.

Consider, for example, a process group system such as Horus. Horus operates over the lower levels of the ISO hierarchy, and hence draws upon point-to-point properties of the routing subsystem and data-link layers of the transport subsystem. The correct function of Horus ultimately depends upon the correct function of these lower layers: if packets cease to be routed to their destinations, or corrupted packets are passed up to the application layer without detection by the data-link layer, Horus will begin to malfunction. Taking just one instance, the protocol used by UDP to reassemble fragmented packets into larger arbitrary sized packets, and the methods used by the device driver and device controller to manage packet chains. These are non-trivial software procedures, and are entirely capable of malfunctioning in subtle ways.

In the author’s direct experience (just to emphasize that there is already a non-trivial problem here), protocols and systems have been observed to fail because of hardware that experienced an extremely high error rate for certain sizes of data packet (multiples of 204 bytes, if memory serves), because of an accidental configuration error that could cause an network to partition in a partial way, delivering only packets associated with certain protocols and losing others, and operating systems interfaces have failed, for example by rejecting all UDP packets. (Since UDP is not guaranteed to be reliable, the vendor may have actually considered this behavior to be a "correct" one!). Othr researchers have shown that network routing can malfunction, so that process *a* can communicate normally with process *b* and *b* with both *a* and *c*, but so that *a* and *c* are incapable of direct communication. Such a behavior directly violates what we are told to assume about routing, and yet it arises with disturbing frequency in the modern internet! In one memorable event, the compiler malfunctioned in a way that caused a low-level Isis protocol to deadlock, despite the fact that the protocol as coded was actually correct. And now the question arises: how can we possibly build reliable systems, if we can't trust our own assumptions?

One could argue that the point-to-point properties of the communication environment represents what is just a first dimension for potential analysis and formal verification. Moreover, actually writing down the properties of a communication environment will clearly represent an extremely difficult task. To first approximation, we expect a message passing subsystem to route packets safely and correctly to their destinations, discarding corrupted ones, to be non-generative (i.e. to deliver only packets that were legitimately produced by the application), and to be free of very delayed replays. In practice, we often make a further assumption that the network has short, bounded delivery delays. But to what degree are such assumptions actually valid? Like the routing assumption, it may be that these assumptions sometimes will break down.

Moving on, the point-to-point properties of a network will now be extended by higher level protocols into end-to-end properties. For the ISO hierarchy, these correspond to stream protocols. Were we to verify such a protocol, we would need to first express the properties of the underlying system, and then to characterize the desired protocol properties. For example, a stream protocol typically will detect and reject duplicated packets, and place out-of-order packets back into the order of generation. Such protocols also handle retransmission, detection of failures, and may dynamically vary such parameters are window size or data flow rate to match the properties of the sender, receiver and network. Again, although the problem of actually doing this may seem simple at first glance, both the properties themselves and the task of validating them grow much more difficult when the true nature of the assumptions made and of the guarantees offered are really pinned down. TCP, for example, uses a finite counter that wraps when the field is full. If packet numbers can be reused, the protocol will not be correct in a network that may replay packets after long delays. The assumption that the network doesn't behave in this way is implicit in the typical TCP implementation, but would need to become explicit if we wanted to prove that TCP really works as desired. And then one would need to ask whether or not we really have grounds to trust this assumption. For example, what would happen if a router node somehow jammed up for a few hours but then suddenly resumed operation. Might it then replay packets hours after they were first sent?

In fact, there has been research on proving TCP correct under various assumptions, but to do so under realistic assumptions would be a daunting undertaking. This illustrates one of the points we will want to reconsider below: correctness analyses of protocols should probably not undertake to model the environment and protocol in excessive detail. Doing so is likely to load up the proof with a tremendous amount of detail, and yet may ultimately yield only limited insight. Moreover, when the proof is finished, the developer may still be left facing a number of assumptions in which one can have no better than probabilistic confidence.

The problem grows still more complex as we consider group communication protocols. Not only do these protocols depend upon point-to-point and end-to-end assumptions at the level of communication channels over which they run, they also depend upon the abstraction created by the group membership protocol (which may operate in a partitionable environment), and there may be further assumptions about the application itself. It is helpful to consider these protocols as having system-wide properties, such as the group abstraction itself, or the dynamic uniformity guarantee of a protocol having this property: another "dimension" to the spectrum of properties. It turns out that even if we assume a very simple environmental model, capturing this system-wide perspective can be tremendously challenging. What does it "mean" to say that a group manages a replicated data object, or that its members behave in a mutually consistent manner? Questions such as these remain active research topics today. Researchers work on them because there is considerable reason to hope that they can be solved, and because we gain considerable insight even from partition solutions. However, it may be some time before solutions begin to take on the character of useful tools that practitioners might exploit. We'll have more to say about work in this area below.

We observed that properties can be thought of as existing in a multidimensional space: multidimensional because there is a sense in which, for example, group properties seem to be orthogonal from the channel properties over which the group is implemented. Similarly, the real-time assumptions and guarantees of a real-time protocol suite represent a different "dimension" along which a system may seek to provide guarantees, as do the security properties of the environment and system, and its self-management abilities and guarantees.

If it is extremely hard provide a formal proof that a TCP protocol guarantees the stream properties, it will be neatly impossible to do so for a complex system that seeks to "guarantee" a range of reliability properties such as these. Of course, by isolating a question about a protocol, we can often provide a formal proof that under some set of conditions, the protocol provides a desired property or that it will always make progress. When we talk about "reasoning about a distributed system", this is typically what we have in mind: the systematic analysis of some aspect of the system, under what may seem like extreme simplifying assumptions, to establish that when those assumptions hold, the system has the desired property.

Such a point of view can now lead us in any of several directions. Below, we will review the state of affairs in some of the major research areas that take this sort of approach. Quite a bit is known, and the rate of progress is extremely encouraging. Over time, there is reason to expect that formal tools may emerge to assist the developer in putting together a system that has a desired property or behavior, using the sort of narrow and highly targetted analysis of properties that emerges from this approach.

It should be stressed, however, that one can also adopt a more skeptical view of the whole area. If it will ultimately be impossible to prove that systems have desired properties because of the complexity of the assumptions involved and the risk that, ultimately, these assumptions will not hold, we should seriously consider the possibility that, ultimately, reliable systems are "fated" to be unreliable -- but that, hopefully, this will not occur frequently. In the experience of the author, this is not an unrealistic perspective.

There is a community that takes this skepticism to an extreme degree: feeling that reliability guarantees are ultimately meaningless, these developers make no effort to achieve reliability at all. More specifically, they resort to completely ad-hoc methods, test their systems heavily, and accept as inevitable the reliability and security loopholes that may remain. To this author, such an approach seems to go too far: in effect, this community abandons all hope for any meaningful form of reliability at all because perfection is not attainable.

In particular, the sorts of techniques we have presented in the preceeding 24 chapters really do lead to systems that have useful properties and that can automatically adapt themselves to such accidents as infrequent failures and restarts, communication lines that break and later are reconnected, or limited security intrusions. They can guarantee real-time behavior, as long as the clocks are working well enough, the communication lines are responding the way they are expected to respond, and the applications themselves are not overloaded. And, most of the time, such assumptions are reasonable and the systems build this way can be considered as very reliable ones. It seems a bit extreme to throw out these benefits simply because we doubt that the reliability of the resulting applications will achieve some absolute standard of perfection!

Sometimes, however, we must expect that our assumptions will prove to be incorrect and that the correctness of the system as a whole will indeed be compromised. Thus, while on the one hand it makes sense to invest using these tools to build systems, and to invest effort in proving (to the extent that we can) that our techniques work, we should also design our systems to heal themselves when the unexpected occurs and the system state becomes corrupted as a result. That is, we can derive considerable benefit from reasoning about systems and using powerful tools, provided that we take the whole approach with a grain of salt.

Indeed, even if our assumptions were completely correct, we would still face this problem for other reasons. A system like Horus is fairly complex: a typical protocol stack and its runtime support involves perhaps 25,000 lines of active code, not considering the code embedded into the operating system, compiler, and the issues raised by communication with other machines that may be running other systems or other revision levels of the same system. Thus, even if one were ready to "bet the bank" that a certain data replication protocol provides consistency and fault-tolerance, it wouldn't hurt to design the system to either periodically reset the group members to have some known value for that replicated data, or to at least periodically have the group check itself to detect errors.

Such a view argues, ultimately, that reliable distributed systems should be constructed to be as modular as possible: not only will module boundaries serve as a natural place at which to specify behavior and undertake to prove that it will hold, but they will also serve as a natural firewall within which inconsistency or unexpected failures can be contained and repaired, and within which self-checking mechanisms can operate. Recall that we discussed the controversy associated with extending group properties across group boundaries. Here, we now see a concrete reason not to do so casually: by attempting to extend properties across a broad system, we link the correctness of the groups involved together. In effect, we break down what might otherwise have been a form of protection boundary for properties. The same groups, with properties limited to communication that remains inside them, would be much more isolated from one-another and hence much less likely to experience a system-wide failure if one of our assumptions turns out to have been invalid in some isolated corner of the system.

25.2 Process and Message-Oriented Models

In discussing the protocols presented in previous chapters, we developed execution models (the asynchronous model, the transactional model, the virtual synchrony model) which were expressed at the level of processes, messages, time, data objects, and relationships over these primitive elements. Today, such models are the ones most commonly used to formalize and reason about distributed systems. They are seen to be deficient, however, when one tries to apply them to a complex distributed application, composed of multiple subsystems having non-trivial properties and relationships among the components.

Most work on modelling distributed systems and protocols has focused on the use of very simplified formal descriptions of the network, the protocol itself, and the application, and using those descriptions to build up what might be called a "formal system" with provable properties. Although, as we commented above, it appears to be impractical to treat every aspect of a system in this manner, this type of formal systems research can still increase the confidence of the developer that a protocol does what it is supposed to do, does it when it is supposed to do it, and that it isn't subject to deadlock or other undesired problems in the normal course of events.

Speaking broadly, work in this area can be understood as falling into three categories. There is a community of researchers who are focused on developing "distributed algorithms", which generally take the form of simple and very high level techniques for solving specific problems that are expressed in terms of a distributed computing model. We have brushed against some of this work in the text, but in fact have presented very little of what is is a dynamic and extremely productive body of research. Problems that have been studied by this community include the detection of deadlock and other stable conditions in distributed systems, distributed discovery of graphs and computation of subgraphs (for example, to identify optimal routes in a routed network), consensus and leader election, various forms of load-balanced execution, evaluation of predicates whose variables make reference to the states of processes in a distributed setting, and so forth. Recently, the group membership problem and virtual synchrony model has become a focus of attention for this community.

In building reliable distributed systems, one sometimes encounters problems such as these, and it can be very useful to gain background and insight by following the literature of the distributed algorithms community. Although we haven't covered much of this work in tis textbook, there are other texts on the market that do this in a very effective way. Books that the reader might want to look at include Lynch's recent text [Lyn96], Andrew's text [And91], and Schneider forthcoming book [Sch97]. In addition, the interested reader should consider following academic conferences such as the *ACM Proceedings on Distributed Systems* (PODC) and the *Workshop on Distributed Algorithms* (WDAG). Many papers from this community appear in Spinger-Verlag's journal, *Distributed Computing*.

A second category of research in this area is concerned with the logical foundations of the systems and languages that are used to reason about distributed computing systems. The problem here is that we are only just beginning to find ways to express the properties of the sorts of reliability technologies this book has treated in some depth. For example, although the virtual synchrony model can be described in terms of a set of rules that such systems should follow, there is always a significant risk that such rules are somehow imprecise, leave open trivial solutions that don't have the properties desired in the formal specification, or otherwise flawed. What one would ideally want is a highly expressive "language" in which such rules could be expressed rigorously -- a logic for distributed computing systems.

In fact, there has been a great deal of research on logic used to reason about computing systems in general, and distributed systems in particular. Leslie Lamport, for example, whose work was cited in connection with notions of time and consistency, is in fact best known for his research on temporal logics which he uses to express concurrent algorithms and to reason about complex distributed protocols and systems. Other researchers have looked at notions of "knowledge" as they arise in distributed systems. For example, Halpern and Moses proposed what have become known as "knowledge logics" for distributed computing several years ago, and showed that one can sometimes understand distributed protocols better by thinking in terms of facts that the system is able to deduce on the basis of messages that the processes making up the system have exchanged. Lynch has suggested that a type of logic automata might be used to understand the behavior of certain protocols and algorithms. And these are just a few instances from a very broad and vigorous field of study.

Overall, it seems clear that we have yet to see the definitive language or formal approach for describing properties of the sorts of reliable distributed systems treated in this text. Existing languages continue to be fairly far from the actual software that one writes, so that protocols must be translated into the formalism (with the risk of mistakes or mistranslations), and it can be very difficult to reason about failures using formal tools. For example, the author has tried to formalize the properties of the group membership component of the virtual synchrony model, with mixed results (see Chapter 13 and 14). To a large extent, the difficulty here seems to be that the temporal logic used to write down properties of the GMS has difficulty dealing with what might be called "branching" future executions.

Suppose that one were to write down the rule that a process communicating with the GMS sees the same sequence of events as do other processes in the system. This sounds simple until one considers that such a rule needs to be evaluated "at" a time and "on" a set of process states. But the purpose of the GMS is to maintain the set of processes for the system, and this set evolves dynamically through time. Morover, since we lack accurate ways of detecting failures, any given process may suffer the misfortune of being partitioned away from the system and classified as having failed. The system as a whole may partition into two or more component subsystems that experience different sequences of events. Thus, what looked like a very simple goal suddenly becomes extremely complex: our "rule" must somehow be written in a way that can express all of these possible events. Worse still, at any given point in time for a process one may wish to say something about the state of that process, and yet the actual status of that process relative to the other processes in the system may not be known until a complex consensus protocol has run long in the future. Perhaps, at time *t,* process *p* believes that replicated variable *x* has value 17. But at time *t*+10 the system may conclude that process *p* was actually faulty at time *t* and that *x* never had this value.

In effect, the formal interpretation of an event that has already occured in the past will not be decided until some sort of protocol runs. How should one deal with this form of uncertainty? One could write down a very complex property: *either p will turn out to be operational, in which case x is truely 17, or p will later prove to be partitioned away from the system, in which the official value of x may not be 17 after all*. But this will clearly be awkward; moreover, once the "fate" of *p* is finally decided, the correct explanation of its behavior will suddenly become much simplified. Existing systems for formalizing the behavior of distributed algorithms lack a good way to deal with this issue.

Thus, while research is making valuable inroads into the tools for formalizing the behavior of distributed systems, we still seem to be fairly far from having the right set of tools for dealing with some of the more complex mechanisms that were considered in this text. A consequence is that those of us who work with reliability are forced to use relatively informal methods to specify our goals, and hence at constant risk of mishap. On the one hand, it would be desirable to be more formal, but one finds that the languages and logics themselves are awkward for our purposes. On the other hand, one hesitates to abandon formality, because of the very big risk that mistakes will now enter our work.

Finally, there is a third major direction for research into distributed systems specifications and formalisms. Were one to approach a logician with the comments seen above, the obvious first questiont o be posed would concern our ultimate goals: do we actually know what sort of systems we are trying to model? And ironically, the answer would have to be negative. At least at the time of this writing, the distributed systems community has yet to reach real agreement on the most appropriate system models to use. For example, as we saw in the text, Cristian works in a timed asynchronous model [Cri96], while Babaoglu favors a model based on "reachibility" [BDGB94, BDM95], and the author's own work has focused increasingly on Chandra and Toueg's approach of augementing an asynchronous system with a failure detector [CT91, FKMBD95]. Clearly, the language won't do us a great deal of good unless we have some idea of what we want to do with it. The good news here is that all of these approaches seem to work, in the sense that all three can be used to express the properties of the protocols and systems build and to show that they hold under various realistic conditions. But these models are also quite different from one-another and without some degree of consensus, we may not learn enough about any single model to begin to have a handle on what might be called a "distributed theory of everything." Indeed, the model we know the most about is the pure asynchronous model, and yet what we know is that this model is too weak to support the types of reliability that interest us here, and also that real systems are quite different from the ones described by the model.

This discussion has focused on logical properties of distributed systems, but could equally well have looked at security properties or temporal ones, with largely the same conclusions. In the area of security, for example, there has been some very important work on security logics [LABW92], which have been used to reason about the protocol used in Kerberos and other similar authentication subsystems. But, this work doesn't consider issues such as failures and partitioning, and focuses on the simplest non-replicated case of the authentication servers that are considered. So on the one hand we have some encouraging success, but on the other, we face a major challenge in extending the same work to the more complex systems that are used in the "real world".

For the case of systems with temporal properties, we have seen that the style of reasoning used in the CASD protocol was ultimately too conservative for its intended users [CASD85]. Yet we know very little about this sort of protocol, in a formal sense, if we try to run it "fast" enough so that its properties start to become probabalistic ones in the manner seen in Chapter 20. Friedman's work has shown how Horus can be used to obtain very predictable response time in a scalable, load-balanced server. But it seems impractical to *prove* that this approach should work in the manner that it is observed to work! By and large, the formal tools available for reasoning about temporal properties of fault-tolerant systems focus on systems that can be described conservatively. When we try to push predictability and performance close to their joint limits, we appear to enter a less conservative world in which those formal tools no longer will serve us.

Traditionally, when formal models and tools fail us, computer scientists have turned to simulation studies. Yet reliable distributed systems can be too complex to model using simulations. Kalantar, whose work was reported in connection with scalability [Kal95], encountered just this problem in his doctoral research. Although he was able to gain significant insight into the fine-grained behavior of a complex protocol using a simulation study, he was not able to simplify the model enough to let him simulate the large-scale behavior of such systems: the simulation became swamped with detail and too slow. This is one reason that researchers like Friedman have tended to build mock-ups of real applications and to study those mock-ups in great detail: because they draw on the real technology, at least one can be sure they are not simplistic in some basic way. But one is also limited by such an approach, because it can be hard to study the impact of specific parameters on real systems. How can one vary the latencies or bandwidths of a network used in a real system, for example, or arrange for a process to fail at a particularly awkward stage of a protocol? Real systems are notoriously difficult to test, and their use in this type of analysis represents a problem of at least comparable difficulty.

25.3 System Definition Languages

The first part of this textbook can be understood, in retrospect, as a study of client-server computing. Readers will recall that we started by looking at some of the underlying technical issues, such as the mechanisms by which messages can be addressed and transported, associated reliability issues, and computing models. But we then argued that the successful use of such technologies requires the development of computing environments within which the interfaces between the "objects" distributed over the network are published and standardized. This lead to a review of CORBA, which has become widely standard as the best available technology for documenting interfaces and managing systems that are modeled as collections of cooperating objects.

As we moved beyond these technologies into a broader collection of tools and protocols for replicating data, load-balancing, fault-tolerance, and for guaranteeing low latency and high data throughput, we lost the connection to specifications. It is clear how one can specify that a service manages 2-dimensional tables, but not at all clear how to specify that the service is replicated on three sites which must be selected to fail independently, accessible over two communication paths which similarly fail independently, load-balanced, and capable of providing 100ms response times to queries. This form of information, although potentially necessary for the purpose of describing a system correctly, is nowhere represented in a typical system.

To a limited degree, we saw that one can model a distributed system using a form of relational model similar to that used in relational database systems. Were this done, the kinds of properties listed above could be written down in the relational calculus much as one expresses consistency constraints in a database setting. Doing so would represent initial steps towards *system definition languages* within which the necessary, desired, or typical characteristics of a computing system could be specified and reasoned about.

How might one use such information, if a system were to make it explicit? One option would be to construct tools for automatically generating management policies permitting the system to be monitored and controlled so as to enforce the desired behavior through appropriate interventions. Another possibility would be to use such information to preprogram the communication subsystem so that it can allocate the resources needed to assure that a necessary quality of service will be provided and, if that level of service cannot be guaranteed, that an upcall will be issued to warn the system that it is operating in a degraded mode. One could use this information to protect the system against attacks that seek to create an environment that violates one or another of the basic assumptions to force the system itself to fail. And, information of this sort could be used to simulate and model the system, permitting intelligent comparison of alternative hardware configurations, management policies, and calculation of likely peak loading.

For example, in his work on Horus, Robbert van Renesse was able to chacterize the virtual synchrony stack in terms of a fairly small set of properties that the application might or might not require. One could imagine wiring Horus to some form of system definition language, so that the user would merely specify the expectations of the system, and the actual construction of an appropriate protocol stack could then be completely automated. A similar approach might be possible with the RMP system which, as will be observed below, also makes use of a formal specification language and hence could potential associate specific properties with specific needs of the user.

At the time of this writing, the author is not aware of any work on systems definition languages, but such a direction seems like a logical path for the CORBA community to pursue as the basic elements of CORBA become more form and their practical implications better understood. This topic, then, would appear to be a good one for near-term research within the language community or distributed systems management community.

25.4 High Level Languages and Logics

In our review of the Horus system, the reader may have been puzzled by the repeated references to the ML programming language [MMH90], which is one of the options for specifying Horus layers. ML is a Lisp-like high level language that has traditionally been interpreted, although compilers for ML are now entering general use, and mechanisms for translation from ML to C exist (indeed, Horus layers coded in ML are currently translated to C and then compiled from that form). Nonetheless, given that C and C++ offer greater control over performance, why would we consider implementing large parts of Horus in ML?

We have seen that the protocols implemented within Horus layers are potentially complex, both in their own terms, and also in terms of their interactions. As a consequence, it may be difficult to reason about the expected properties of a Horus stack that combines several layers in a non-trivial way. When programming languages such as C or C++ are used to implement a Horus stack, our ability to reason about such layerings is potentially limited by the lack of automated tools for doing so. We are forced to express the behavior of the layers in English or in some form of temporal logic, and may make mistakes in doing so (we cited, for example, some serious problems that arose when the author and a colleague tried to express the GMS service using a temporal logic language that turned out to be ill-suited to the purpose). An argument can be made that such an approach will lead to unconvincing correctness and behavioral arguments.

When a Horus layer is specified in ML, on the other hand, the situation is somewhat brighter in just these respects. ML is a very high level language, and a substantial body of applied mathematics and logic has evolved around the language. In particular, there exist sophisticated proof tools for ML: programming environments that assist the user in proving things about programs written in ML, or in deriving ML programs that "illustrate" the technique used to prove a purely mathematical property.
By coding Horus in ML, we can benefit from both of these automated theorem proving techniques.

In the first direction, if a Horus layer is expressed in ML, we can potentially claim something about that layer (for example, that its synchronization mechanism is deadlock-free) and establish the precise assumptions that must be made about layers above and below it for these condition to be true. The NuPrl (pronounced "New Pearl") environment is a particularly powerful tool for this sort of application, and is designed to accept ML as an input. At Cornell, the Horus project has already been successful in using NuPrl this way to establish simple properties of Horus layers. The potential of taking this work further and reasoning about composition of layers or complex properties of layers is very exciting. In particular, because NuPrl is automated, it can track very complex properties and automatically complete tedious aspects of such proofs. The degree of confidence one can express in the result is consequently much higher than for a hand-developed proof, and the nature of the questions one can undertake to ask is much expanded.

At the same time, one can imagine going from the bottom up and developing a formal logic within which a Horus protocol suite would be proved "possible" under a set of assumptions about the environment. With this direction, it would (at least in theory) be practical to extract the Horus layers that implement the proofs directly from the proofs. Such a process is similar to one whereby the proof that, given a pair of positive integers, one can find their least common multiple, embodies an algorithm for doing so. NuPrl is designed to extract algorithms for this form of constructive proof, and hence could potentially extract Horus layers that are intrinsically trustworthy because they would be backed by a rigorous mathematical foundation extending down to the first principles upon which the system is based.

While ambitious, we believe that such a goal is also eventually achievable, albeit in limited ways. More broadly, such a direction points to the potential for compiling protocols from descriptions of the system behavior that is desired. For applications requiring, for example, security and trust, one would be much more likely to trust the resulting protocols than protocols that were hand coded and proved correct after the fact, because such proofs are only as good as the developer's ability to formalize the behavior of the layer.

There are, unfortunately, serious obstacles to both of these directions. Earlier, we cited the difficulty of expressing a problem such as the group membership problem in the most widely used temporal logic for distributed systems. As we saw at the time, the core difficulty lies in the fact that at time an event occurs within a process, one may not yet know if that process will remain a member of the system or be excluded from it as faulty. Indeed, one may not even know if the system as a whole will continue to make progress. Thus, one is forced to make the claim that *if* the system makes progress, either this process will be considered as *faulty,* in which case some set of conditions holds upon the events that took place before it failed, or it will be considered as *correct,* in which case some other set of conditions holds. This is a very clumsy way to express the behavior of a distributed system, but seems to be the only possible way to do so in the usual temporal logic. Lacking progress on this problem, it is unlikely that NuPrl would be able to fully model the process group mechanisms implemented by Horus, and hence unlikely that we could fully verify Horus.

Yet there may be considerable benefit from less ambitious uses of NuPrl. For example, we have begun to make use of security layers that integrate Horus with the Fortezza standard. It may be entirely practical to formalize the resulting authentication and trust properties of the system, and in this manner to use NuPrl to convince ourselves that certain styles of distributed service are safe against attack. There appears to be considerable potential for such "lightweight" uses of NuPrl, even if the larger challenge of using such a system to verify all of Horus remains far in the future.

It should be noted that Horus is not at all the first to explore the use of ML for improving the clarity of distributed protocols. Earlier work includes [Kru91, BHL93, HL94, Bia94]. Also, although the author is more familiar with the work on hardening Horus, interested readers should also look at Nasa's work on formalizing the protocols and properties of the RMP (*Reliable Multicast Protocol*) system. As reported in [Mon94, MW94, CM96a, Wu95], the RMP group has achieved considerable success both in expressing properties of their protocol and in formally verifying that these properties actually hold.

26. Other Distributed and Transactional Systems

In this chapter we review some of the advanced research efforts in the areas covered by the text. The first section focuses on message-passing and group-communication systems and the second on transactional systems. The review is not intended to be exhaustive, but we do try to include the major activities that contributed to the technology areas stressed in the text itself.

26.1 Related Work in Distributed Computing

There there have been many distributed systems in which group communication played a role. We now review some of these systems, providing a brief description of the features of each, and citing sources of additional information. Our focus is on distributed computing systems and environments with support for some form of process group computing. However, we do not limit ourselves to those systems implementing virtually synchronous process groups or a variation on the model. Our review presents these systems in alphabetical order. Were we to discuss them chronologically, we would start by considering V, then the Isis Toolkit and Delta-4, and then we would turn to the others in a roughly alphabetical ordering. However, it is important to understand that these systems are the output of a vigorous research community, and that each of the systems cited included significant research innovations at the time it was developed. It would be simplistic to say that any one of these systems came first and that the remainder are somehow secondary. More accurate would be to say that each system innovated in some areas and borrowed ideas from prior systems in other areas.

Readers interested in learning more about this area may want to start by consulting the papers that appeared in *Communications of the ACM* in a special section of the April 1996 issue (Vol. 39, No. 4). David Powell's introduction to this special section is both witty and informative [Pow96], and there are papers on several of the systems touched upon in this text [MMABL96, DM96, R96, RBM96, SR96, Cri96].

26.1.1 Ameoba

During the early 1990's, Ameoba [RST88, RST89, MRTR90] was one of a few micro-kernel based operating systems proposed for distributed computing (others include V [CZ85], Mach [Ras86], Chorus [RAAB88, RAAH88] and QNX [Hil92]). The focus of the project when it was first launched was to develop a distributed system around a nucleus supporting extremely high performance communication, with the remaining system services being implemented using a client-server architecture. In our area of emphasis, process group protocols, Ameoba supports a subsystem developed by Frans Kaashoek that provides group communication using total ordering [Kaa92]. Message delivery is atomic and totally ordered, and implements a form of virtually synchronous addressing. During the early 1990's, Ameoba's sequencer protocols set performance records for throughput and latency, although other systems subsequently bypassed these using a mixture of protocol refinements and new generations of hardware and software.

26.1.2 Chorus

Chorus is an object-oriented operating system for distributed computing [RAAB88, RAAH88]. Developed at INRIA during the 1980's, the technology shifted to a commercial track in the early 1990's and has become one of the major vehicles for commercial UNIX development and for real-time computing products. The system is notable for its modularity and comprehensive use of object-oriented programming techniques. Chorus was one of the first systems to embrace these ideas, and is extremely sophisticated in its support for modular application programming and for reconfiguration of the operating system itself.

Chorus implements a process group communication primitive which is used to assist applications in dealing with services that are replicated for higher availability. When an RPC is issued to such a replicated service, Chorus picks a single member and issues an invocation to it. A feature is also available for sending an unreliable multicast the members of a process group (no ordering or atomicity guarantees are provided).

In its present commercial incarnation, the Chorus operating system is used primarily in real-time settings, for applications that arise in telecommunications systems. Running over Chorus is an object-request broker technology called Cool-ORB. This system includes a variety of distributed computing services including a replication service capable of being interconnected to a process group technology, such as that used in the Horus system.

26.1.3 Delta-4

Delta-4 was one of the first systematic efforts to address reliability and fault-tolerance concerns [Pow94]. Launched in Europe during the late 1980's, Delta-4 was developed by a multinational team of companies and academic researchers [Pow91, RV89]. The focus of the project was on factory floor applications, which combine real-time and fault-tolerance requirements. Delta-4 took an approach in which a trusted module was added to each host computer, and used to run fault-tolerance protocols. These modules were implemented in software but could be included onto a specially designed hardware interface to a shared communication bus. The protocols used in the system included process group mechanisms similar to the ones now employed to support virtual synchrony, although Delta-4 did not employ the virtual synchrony computing model.

The project was extremely successful as a research effort and resulting in working prototypes that were indeed fault-tolerant and capable of coordinated real-time control in distributed automation settings. Unfortunately, however, this stage was reached as Europe entered a period of economic difficulties, and none of the participating companies was able to pursue the technology base after the research funding of the project ended. Ideas from Delta-4 can now be found in a number of other group-oriented and real-time distributed systems, including Horus.

26.1.4 Harp

The "gossip" protocols of Ladin and Liskov were mentioned in conjunction with our discussion of communication from a non-member of a process group into that group [LGGJ91, LLSG92]. These protocols were originally introduced in a replicated file system project undertaken at MIT in the early 1990's. The key idea of the Harp system was to use a lazy update mechanism as a way of obtaining high performance and tolerance to partitioning failures in a replicated file system. The system was structured as a collection of file servers, consisting of multiple processes each of which maintained a full copy of the file system, and a set of clients that issue requests to the servers, switching from server to server to balance load or to overcome failures of the network or of a server process. Clients issue read operations, which the system handled locally at which ever server received the request, and update operations, which were performed using a quorum algorithm. Any updates destined for a faulty or unavailable process were spooled for later transmission when the process recovered or communication to it was reestablished. To ensure that when a client issues a series of requests, the file servers perform them at consistent (e.g. logically advancing) times, each response from a file server process to a client included a timestamp, which the client could present on subsequent requests. The timestamp was represented as a vector clock, and could be used to delay a client's request if it was sent to a server that had not yet seen some updates on which the request might be dependent.

Harp made extensive use of a hardware feature not widely used in modern workstations, despite its low cost and off-the-shelf availability. A so-called non-volatile or battery-backed RAM (NVRAM) is a small memory that preserves its contents even if the host computer crashes and later restarts. Finding that the performance of HARP was dominated by the latency associated with forced log writes to the disk, Ladin and Liskov purchased these inexpensive devices for the machines on which HARP runs and modified the HARP software to use the NVRAM area as a persistent data structure that could hold commit records, locking information, and a small amount of additional commit-related data. Performance of HARP increased sharply, leading these researchers to argue that greater use should be made of NVRAM in reliable systems of all sorts. However, NVRAM is not found on typical workstations or computing systems, and vendors of the major transactional and database products are under great pressure to offer the best possible performance on completely standard platforms, making the use of NVRAM problematic in commercial products. The technology used in HARP, on the other hand, would not perform well without NVRAM storage.

26.1.5 The Highly Available System (HAS)

The Highly Available System was developed by IBM's Almaden research laboratory under the direction of Cristian and Strong, with involvement by Skeen and Schmuck, in the late 1980's and subsequently contributed technology to a number of IBM products, including the ill-fated Advanced Automation System (AAS) development that IBM undertook for the American Federal Aviation Agency (FAA) in the early 1990's [CD90, Cri91a]. Unfortunately, relatively little of what was apparently a substantial body of work was published on this system. The most widely known results include the *timed asynchronous communication model,* proposed by Cristian and Schmuck [CS95] and used to provide a precise semantics for their reliable protocols. Protocols were proposed for synchronizing the clocks in a distributed system [Cri89], managing group membership in real-time settings [Cri91b] and for atomic communication to groups [CASD85, CDSA90], subject to timing bounds, and achieving totally ordered delivery guarantees at the operational members of groups. Details of these protocols were presented in Chapter 20. A shared memory model called *Delta-Common Storage* was proposed as a part of this project, and consisted of a tool by which process group members could communicate using a shared memory abstraction, with guarantees that updates would be seen by all operational group members (if by any) within a limited period of time.

26.1.6 The Isis Toolkit

Developed by the author of this textbook and his colleagues during the period 1985-1990, the Isis Toolkit was the first process group communication system to use the virtual synchrony model [BJ87a, BJ87b, BR94]. As its name suggests, Isis is a collection of procedural tools that are linked directly to the application program, providing it with functionality for creating and joining process groups dynamically, multicasting to process groups with various ordering guarantees, replicating data and synchronizing the actions of group members as they access that data, performing operations in a load-balanced or fault-tolerant manner, and so forth [BR96]. Over time, a number of applications were developed using Isis, and it became widely used through a public software distribution. These developments lead to the commercialization of Isis through a company, which today operates as a wholly owned subsidiary of Stratus Computer Inc. The company continues to extend and sell the Isis Toolkit itself, as well as an object-oriented embedding of Isis called Orbix+Isis (it extends Iona's popular Orbix product with Isis group functionality and fault-tolerance [O+I95]), products for database and file system replication, a message bus technology supporting a reliable post/subscribe interface, and a system management technology for supervising a system and controlling the actions of its components.

Isis introduced the primary partition virtual synchrony model, and the *cbcast* primitive. These steps enabled it to support a variety of reliable programming tools, which was unusual for process group systems at the time Isis was developed. Late in the "life cycle" of the system it was one of the first (along with the Harp system of Ladin and Liskov) to use vector timestamps to enforce causal ordering. In a practical sense, the system represented an advance merely by being a genuinely useable packaging of a reliable computing technology into a form that could be used by a large community.

Successful applications of Isis include components of the New York and Swiss stock exchanges, distributed control in AMD's FAB-25 VLSI fabrication facility, distributed financial databases such as one developed by the World Bank, a number of telecommunications applications involving mobility, distributed switch management and control, billing and fraud detection, several applications in air-traffic control and space data collection, and many others. The major markets into which the technology is currently sold are financial, telecommunications, and factory automation.

26.1.7 Locus

Locus is a distributed operating system developed by Popek's group at UCLA in the mid 1980's [WPEK93]. Known for such features as transparent process migration and a uniform distributed shared memory abstraction, Locus was extremely influential in the early development of parallel and cluster-style computing systems. Locus was eventually commercialized and is now a product of Locus Computing Corporation. The file system component of Locus was later extended into the Ficus system, which we discussed earlier in conjunction with other "stateful" file systems.

26.1.8 Sender-Based Logging and Manetho

In writing this text, the author was forced to make certain tradeoffs in terms of the coverage of topics. One topic that was not included is that of log-based recovery, whereby applications create checkpoints periodically and log messages sent or received. Recovery is by rollback into a consistent state, after which log replay is used to regain the state as of the instant when the failure occured.

Manetho [EZ92] is perhaps the best known of the log-based recovery systems, although the idea of using logging for fault-tolerance is quite a bit older [BBG83, KT87, JZ90]. In Manetho, a library of communication procedures automates the creation of logs that include all messages sent from application to application. An assumption is made that application programs are deterministic and will reenter the same state if the same sequence of messages is played into them. In the event of a failure, a rollback protocol is triggered that will roll back one or more programs until the system state is globally consistent, meaning that the set of logs and checkpoints represents a state that the system could have entered at some instant in logical time. Manetho then rolls the system forward by redelivery of the logged messages.. Because the messages are logged at the sender, the technique is called *sender-based logging* [JZ87]. Experiments with Manetho have confirmed that the overhead of the technique is extremely small. Moreover, working independently, Alvisi has demonstrated that sender-based logging is just one of a very general spectrum of logging methods that can store messages close to the sender, close to the recipient, or even mix these options [AM93].

Although conceptually simple, logging has never played a major role in reliable distributed systems in the field, most likely because of the determinism constraint and the need to use the logging and recovery technique system-wide. This issue, which also makes it difficult to transparently replicate a program to make it fault-tolerant, seems to be one of the fundamental obstacles to software-based reliability technologies. Unfortunately, non-determinism can creep into a system through a great many interfaces. Use of shared memory or semaphore-style synchronization can cause a system to be non-deterministic, as can any dependency on the order of message reception, the amount of data in a pipe or the time in the execution when the data arrives, the system clock, or the thread scheduling order. This implies that the class of applications for which one can legitimately make a determinism assumption is very small.

For example, suppose that the servers used in some system are a mixture of deterministic and non-deterministic programs. Active replication could be used to replicate the deterministic programs transparently, and the sorts of techniques discussed in previous chapters employed in the remainder. However, to use a sender-based logging technique (or any logging technique), the entire group of application programs needs to satisfy this assumption, hence one would need to recode the non-deterministic servers before any benefit of any kind could be obtained. This obstacle is apparently sufficient to deter most potential users of the technique.

The author is aware, however, of some successes with log-based recovery in specific applications that happen to have a very simple structure. For example, a popular approach to factoring very large numbers involves running very large numbers of completely independent factoring processes that deal with small ranges of potential factors, and such systems are very well suited to a log-based recovery technique because the computations are deterministic and there is little communication between the participating processes. Broadly, log-based recovery seems to be more applicable to scientific computing systems or problems like the factoring problem than to general purpose distributed computing of the sort seen in corporate environments or the Web.

26.1.9 NavTech

NavTech is a distributed computing environment built using Horus [BR96, RBM96], but with its own protocols and specialized distributed services [VR92, RV93, Ver93, Ver94, RV95, Ver96]. The group responsible for the system is headed by Verissimo, who was one of the major contributors to Delta-4, and the system reflects many ideas that originated in that earlier effort. NavTech is aimed at wide-area applications with real-time constraints, such as banking systems that involve a large number of "branches" and factory-floor applications in which control must be done close to a factory component or device. The issues that arise when real-time and fault-tolerance problems are considered in a single setting thus represent a particular focus of the effort. Future emphasis by the group will be on the integration of graphical user interfaces, security, and distributed fault-tolerance within a single setting. Such a mixture of technologies would result in an appropriate technology base for applications such as home banking and distributed game playing, both expected to be popular early uses of the new generation of internet technologies.

26.1.10 Phoenix

Phoenix is a recent distributed computing effort that was launched by C. Malloth and Andre Schiper of the Ecole Polytechnique de Lausanne jointly with Ozalp Babaoglu and Paulo Verissimo [Mal96, *see also* SR96]. Most work on the project is currently occurring at EPFL. The emphasis of this system is on issues that arise when process group techniques are used to implement wide-area transactional systems or database systems. Phoenix has a Horus-like architecture, but uses protocols specialized to the needs of transactional applications, and has developed an extention of the virtual synchrony model within which transactional serializability can be treated elegantly.

26.1.11 Psync

Psync is a distributed computing system that was developed by Peterson at the University of Arizona in the late 1980's and early 1990's [Pet87, PBS89, MPS91]. The focus of the effort was to identify a suitable set of tools with which to implement protocols such as the ones we have presented in the last few chapters. In effect, Psync sets out to solve the same problem as the Express Transfer Protocol, but where XTP focuses on point to point datagrams and streaming style protocols, Psync was more oriented towards group communication and protocols with distributed ordering properties. A basic set of primitives was provided for identifying messages and for reasoning about their ordering relationships. Over these primitives, Psync provided implementations of a variety of ordered and atomic multicast protocols.

26.1.12 Relacs

The Relacs system is the product of a research effort headed by Ozalp Babaoglu at the University of Bologna [BDGB94, BDM95]. The activity includes a strong theoretical component, but has also developed an experimental software testbed within which protocols developed by the project can be implemented and validated. The focus of Relacs is on the extention of virtual synchrony to wide-area networks in which partial connectivity disrupts communication. Basic results of this effort include a theory that links *reachability* to consistency in distributed protocols, and a proposed extention of the view synchrony properties of a virtually synchronous group model that permits safe operation for certain classes of algorithms despite partitioning failures. At the time of this writing, the project was working to identify the most appropriate primitives and design techniques for implementing wide-area distributed applications that offer strong fault-tolerance and consistency guarantees, and to formalize the models and correctness proofs for such primitives [BBD96].

26.1.13 Rampart

Rampart is a distributed system that uses virtually synchronous process groups in settings where security is desired even if components fail in arbitrary (Byzantine) ways [Rei96]. The activity is headed by Reiter at AT&T Bell Laboratories, and has resulted in a number of protocols for implementing process groups despite Byzantine failures as well as a prototype of a security architecture that employs these protocols [RBG92, Rei93, RB94, Rei94a, Rei94b, RBR95]. We discuss this system in more detail in Chapter 19. Rampart's protocols are more costly than those we have presented above, but the system would probably not be used to support a complete distributed application. Instead, Rampart's mechanisms could be employed to implement a very secure subsystem, such as a digital cash server or an authentication server in a distributed setting, while other less costly mechanisms were employed to implement the applications that make use of these very secure services.

26.1.14 RMP

The RMP system is a public-domain process group environment implementing virtual synchrony, with a focus on extremely high performance and simplicity. The majority of the development on this system occurred at U.C. Berkeley, where graduate student Brian Whetten needed such a technology for his work on distributed multimedia applications [MW94, Mon94, Whe95, CM96a]. Over time, the project became much broader, as West Virginia University / Nasa researchers Jack Callahan and Todd Montgomery became involved. Broadly speaking, RMP is similar to the Horus system, although less extensively layered.

The major focus of the RMP project has been on embedded systems applications that might arise in future space platforms or ground-based computing support for space systems. Early RMP users have been drawn from this community, and the long term goals of the effort are to develop technologies suitable for use by Nasa. As a result, the verification of RMP has become particularly important, since systems of this sort cannot easily be upgraded or services while in flight. RMP has pioneered the use of formal verification and software design tools in protocol verification [CM96a, Wu95], and the project is increasingly focused on robustness through formal methods, a notable shift from its early emphasis on setting new performance records.

26.1.15 StormCast

Researchers at the University of Tromso, within the Arctic circle, launched this effort, which seeks to implement a wide area weather and environmental monitoring system for Norway. StormCast is not a group communication system per-se, but rather is one of the most visible and best documented of the major group communication applications [AJ95, JH94, Joh94; *see also* BR96 *and* JvRS95a, JvRS95b, JvRS96]. Process group technologies are employed within this system for parallelism, fault-tolerance, and system management.

The basic architecture of StormCast consists of a set of data archiving sites, located throughout the far north. At the time of this writing, StormCast had roughly a half-dozen such sites, with more coming on line each year. Many of these sites simply gather and log weather data, but some collect radar and satellite imagery, and others maintain extensive datasets associated with short and long-term weather modeling and predictions. StormCast application programs typically draw on this varied data set for purposes such as local weather prediction, tracking of environmental problems such as oil spills (or radioactive discharges from within the ex-Soviet block to the east), research into weather modelling, and other similar applications.

StormCast is interesting for many reasons. The architecture of the system has received intense scrutiny [Joh94, JH94], and evolved over a series of iterations into one in which the application developer is guided to a solution using tools appropriate to the application, and by following templates that worked successfully for other similar applications. This notion of architecture driving the solution is one that has been lost in many distributed computing environments, which tend to be architecturally "flat" (presenting the same tools, services and API's system-wide even if the applications themselves have some very clear architecture, like a client-server structure, in which different parts of the system need different forms of support). It is interesting to note that early versions of StormCast, which lacked such a strong notion of system architecture, were much more difficult to use than the current one, in which the developer actually has less "freedom" but much stronger guidence towards solutions.

StormCast has encountered some difficult technical challenges. The very large amounts of data gathered by weather monitoring systems necessarily must be "visited" on the servers where they reside; it is impractical to move the data to the place where the user who requests a service, such as a local weather forecast, may be working. Thus, StormCast has pioneered in the development of techniques for sending computations to data: the so-called *agent* architecture [Rei94] we discussed in Section 10.8 in conjunction with the Tacoma system [JvRS95a, JvRS95b, JvRS96].

In a typical case, an airport weather prediction for Tromso might involve checking for incoming storms in the 500-km radius around Tromso, and then visiting one of several other data archives depending upon the prevailing winds and the locations of incoming weather systems. The severe and unpredictable nature of arctic weather makes these computations equally unpredictable: the data needed for one prediction may be primarily archives in the south of Norway while that needed for some other prediction is archived in the north, or on a system that collects data from trawlers along the coast. Such problems are solved by designing Tacoma agents that travel to the data, preprocess it to extract needed information, and then return to the end-user for display or further processing. Although such an approach raises challenging software design and management problems, it also seems to be the only viable option for working with such large quantities of data and supporting such a varied and unpredictable community of users and applications.

It should be noted that StormCast maintains an unusually interesting web page, **http://www.cs.uit.no**. Readers who have a web browser will find interactive remote controlled cameras focused on the ski trails near the University, current environmental monitoring information including data on small oil spills and the responsible vessels, 3-dimensional weather predictions intended to aid air-traffic controllers in recommending the best approach paths to airports in the region, and other examples of the use of the system. One can also download a version of Tacoma and use it to develop new weather or environmental applications that can be submitted directly to the StormCast system, load permitting.

26.1.16 Totem

The Totem system is the result of a multi-year project at U.C. Santa Barbara, focusing on process groups in settings that require extremely high performance and real-time guarantees [MMABL96, *see also* MM89, MMA90a, MMA90b, MM93, AMMA93, Aga94, MMA94]. The computing model used is the extended virtual synchrony one, and was originally developed by this group in collaboration with the Transis project in Isreal. Totem has contributed a number of high performance protocols, including a innovative causal and total ordering algorithm based on transitive ordering relationships between messages and a totally ordered protocol with extremely predictable real-time properties. The system differs from a technology like Horus in focusing on a type of distributed system that would result from the interconnection of clusters of workstations using broadcast media within these clusters and some form of bridging technology between them. Most of the protocols are optimized for applications within which communication loads are high and either uniformly distributed over the processes in the system, or in which messages originate primarily at a single source. The resulting protocols are very efficient in their use of messages but sometimes exhibit higher latency than the protocols we presented in earlier chapters of this textbook. Intended applications include parallel computing on clusters of workstations and industrial control problems.

26.1.17 Transis

The Transis system [DM96] is one of the best known and most successful process group-based research at the time of this writing. The group has contributed extensively to the theory of process group systems and virtual synchrony, repeatedly set performance records with its protocols and flow-control algorithms, and developed a remarkable variety of protocols and algorithms in support of such systems [ADKM92a, ADKM92b, AMMA93, AAD93, Mal94, KD95, FKMBD95]. Many of the ideas from Transis were eventually ported into the Horus system. Transis was, for example, the first system to show that by exploiting hardware multicast, a reliable group multicast protocol could scale with almost no growth in cost or latency. The "primary" focus of this effort was initially partitionable environments, and much of what is known about consistent distributed computing in such settings originated either directly or indirectly from this group. The project is also known for its work on transactional applications that preserve consistency in partitionable settings.

Recently, the project has begun to look at security issues that arise in systems subject to partitioning failures. The effort seeks to provide secure autonomous communication even while subsystems of a distributed system are partitioned away from a central authentication server. As we will see in the next Chapter, the most widely used security architectures would not allow secure operations to be initiated in such a partitioned system component and would not be able to deal with the revalidation of such a component if it later reconnected to the system and wanted to merge its groups into others that remained in the primary component. Mobility is likely to create a need for security of this sort, for example in financial applications and in military settings, where a team of soldiers may need to operate without direct communication to the central system from time to time.

As noted earlier, another interesting direction under study by the Transis group is that of building systems that combine multiple protocol stacks in which different reliability or quality-of-service properties apply to each stack [Idixx]. In this work, one assumes that a complex distributed system will give rise to a variety of types of reliability requirement: virtual synchrony for its control and coordination logic, isochronous communication for voice and video, and perhaps special encryption requirements for certain sensitive data, each provided through a corresponding protocol stack. However, rather than treating these protocol stacks as completely independent, the Transis work (which should port easily into Horus) deals with the synchronization of streams across multiple stacks. Such a step will greatly simplify the imlementation of demanding applications that need to present a unified appearance and yet cannot readily be implemented within a single protocol stack.

26.1.18 The V System

In the alphabetic ordering of this chapter, it is ironic that the first system to have used process groups is the last that we review. The V System was the first of the micro-kernel operating systems intended specifically for distributed environments, and pioneered the "RISC" style of operating systems developed that later swept the research community in this area. V is known primarily for innovations in the virtual memory and message passing architecture used within the system, which achieved early performance records for its RPC protocol. However, the system also included a process group mechanism, which was used to support distributed services capable of providing a service at multiple locations in a distributed setting [CZ85, Dee88].

Although the V system lacked any strong process group computing model or reliability guarantees, its process group tools were considered quite powerful. In particular, this system was the first to support a publish/subscribe paradigm, in which messages to a "subject" were transmitted to a process group whose named corresponded to that subject. As we saw earlier, such an approach provides a useful separation between the source and destination of messages: the publisher can send to the group without worrying about its current membership, and a subscriber can simply join the group to begin receiving messages published within it.

The V style of process group was not intended for process-group computing of the sorts we explored in this textbook; reliability in the system was purely on a "best effort" basis, meaning that the group communication primitives made an effort to track current group membership and to avoid high rates of message loss, but without providing real guarantees. When Isis introduced the virtual synchrony model, the purpose was precisely to show that with such a model, a V-style of process group could be used to replicate data, balance workload, or provide fault-tolerance. None of these problems were believed solvable in the V system itself. V set the early performance standards against which other group communication systems tended to be evaluated, however, and it was not until a second generation of process group computing systems emerged (the commercial version of Isis, the Transis and Totem systems, Horus and RMP) that these levels of performance were matched and exceeded by systems that also provided reliability and ordering guarantees.

26.2 Systems That Implement Transactions

We end this chapter with a brief review of some of the major research efforts that have explored the use of transactions in distributed settings. As in the case of our review of distributed communications systems, we present these in alphabetical order.

26.2.1 Argus

The Argus system was an early leader among transactional computing systems that considered transactions on abstract objects. Developed by a team lead by Liskov at MIT, the Argus system consists of a programming language and an implementation that was used primarily as a research and experimentation vehicle [LS83, LCJS87, LLSG90]. Many credit the idea of achieving distributed reliability through transactions on distributed objects to this project, and it was a prolific source of publications on all aspects of transactional computing, theoretical as well as practical, during the decade or so of peak activity,

The basic Argus data type is the *guardian:* a software module that defines and implements some form of persistent storage, using transactions to protect against concurrent access and to ensure recoverability and persistence. Similar to a CORBA object, each guardian exports an interface that defines the forms of access and operations possible on the object. Through these interfaces, Argus programs *(actors)* invoke operations on the guarded data. Argus treats all such invocations as transactions and also provides explicit transactional constructs in its programming language, including commit and abort mechanisms, a concurrent execution construct, top-level transactions, and mechanisms for exception handling.

The Argus system implements this model in a transparently distributed manner, with full nested transactions and mechanisms to optimize the more costly aspects, such as nested transaction commit. A sophisticated *orphan termination* protocol is used to track down and abort orphaned subtransactions, which can be created when the parent transaction that initiated some action fails and hence aborts, but leaves active child transactions which may now be at risk of observing system states inconsistent with the conditions under which the child transaction was spawned. For example, a parent transaction might store a record in some object and then spawn a child subtransaction that will eventually read this record. If the parent aborts and the orphaned child is permitted to continue executing, it may read the object in its prior state, leading to seriously inconsistent or erroneous actions.

Although Argus never entered into widespread practical use, the system was extremely influential. Not all aspects of system were successful, in the sense that many commercial transactional systems have rejected distributed and nested transactions is requiring an infrastructure that is relatively more complex, costly, and difficult to use than flat transactions in standard client-server architecture. Other commercial products, however, have adopted parts of this model successfully. The principle of issuing transactions to abstract data types remains debatable. As we saw above, transactional data types can be very difficult to construct, and expert knowledge of the system will often be necessary to achieve high performance. The Argus effort ended in the early 1990's and the MIT group that built the system began work on Thor, a second-generation technology in this area. The author is not sufficiently familiar with Thor, however, to treat it within the current text.

26.2.2 Arjuna

Whereas Argus explores the idea of transactions on objects, Arjuna is a system that focuses on the use of object-oriented techniques to customize a transactional system. Developed by Shrivistava at Newcastle, Arjuna is an extensible and reconfigurable transactional system, in which the developer can replace a standard object-oriented framework for transactional access to persistent objects with type-specific locking or data management objects that exploit semantic knowledge of the application to achieve high performance or special flexibility. The system was one of the first to focus on C++ as a programming language for managing persistent data, an approach that later became widely popular. Recent development of the system has explored the use of replication for increased availability during periods of failure using a protocol called *Newtop*; the underlying methodology used for this purpose draws on the sorts of process group mechanisms discussed in previous chapters [MES93, EMS95].

26.2.3 Avalon

Avalon was a transactional system developed at Carnegie Mellon University by Herlihy and Wing during the late 1980's. The system is best known for its theoretical contributions. This project proposed the *linearizability model,* which weakens serializability in object-oriented settings where full nested serializability may excessively restrict concurrency [HW90]. As noted briefly earlier in the chapter, linearizability has considerable appeal as a model potentially capable of integrating virtual synchrony with serializability. A research project, work on Avalon ended in the early 1990's.

26.2.4 Bayou

Bayou is a recent effort at Xerox Parc that uses transactions with weakened semantics in partially connected settings, such as for the management of distributed calendars for mobile users who may need to make appointments and schedule meetings or read electronic mail while in a disconnected or partially connected environment [TTPD95]. The system provides weak serialization guarantees by allowing the user to schedule meetings even when the full state of the calendar is inaccessible due to a partition. Later, when communication is reestablished, such a transaction is completed with normal serializability semantics.

Bayou makes the observation that transactional consistency may not guarantee that user-specific consistency constraints will be satisfied. For example, if a meeting is scheduled while disconnected form some of the key participants, it may later be discovered that the time conflicts with some other meeting. Bayou provides mechanisms by which the designer can automate both the detection and resolution of these sorts of problems. In this particular example, Bayou will automatically attempt to shift one or the other rather than requiring that a user become directly involved in resolving all such conflicts. The focus of Bayou is very practical: rather than seeking extreme generality, the technology is designed to solve the specific problems encountered in paperless offices with mobile employees. This domain-specific approach permits Bayou to solve a number of distributed consistency problems that, in the most general sense, are not even tractable. This reconfirms an emerging theme of the textbook: theoretical impossibility results often need to be reexamined in specific contexts; what cannot be solved in the most general sense or setting may be entirely tractable in a particular application where more is known about the semantics of operations and data.

26.2.5 Camelot and Encina

This system was developed at Carnegie Mellon University in the late 1980's, and was designed to provide transactional access to user-developed data structures stored in files [Spe85]. The programming model was one in which application programs perform RPC's on servers. Such transactions become nested if these servers are clients of other servers. The ultimate goal is to support transactional semantics for applications that update persistent storage. Camelot introduced a variety of operating system enhancements for maximizing the performance of such applications, and was eventually commercialized in the form of the Encina product from Transarc Corporation. Subsequent to this transition, considerable investment in Encina occurred at Transarc and the system is now one of the leaders in the market for OLTP products. Encina provides both non-distributed and distributed transactions, nested transactions if desired, a variety of tools for balancing load and increasing concurrency, prebuilt data structures for common uses, and management tools for system administration. The distributed data mechanisms can also be used to replicate information for high availability.

Industry analysts have commented that although many Encina users select the system in part for its distributed and nested capabilities, in actual practice most applications of Encina make little or no use of these features. If accurate, this observation raises interesting questions about the true characteristics of the distributed transactional market. Unfortunately, however, the author is not aware of any systematic study of this question.

Readers interested in Encina should also look at IBM's CICS technology, perhaps the world's most widely used transactional system, and at the Tuxedo system, an OLTP product developed originally at AT&T, which became an industry leader in the UNIX OLTP market. Similar to Encina, CICS and Tuxedo provide powerful and complete environments for client-server styled applications that require transactional guarantees, and Tuxedo includes real-time features required in telecommunications settings. This text, however, has generally avoided treatment of commercial technologies with which the author is not extremely familiar, and hence we will not discuss CICS or Tuxedo in any detail here.

Appendix: Problems

This text is intended for use by professionals or advanced students, and the material presented is at a level for which simple problems are not entirely appropriate. Accordingly, most of the problems in this section are intended as the basis for essay-style responses or for programming projects that might build upon the technologies we have treated up to now. Some of these projects are best undertaken as group exercises for a group of three or four students, others could be undertaken by individuals.

Professionals may find these problems interesting from a different perspective. Many of them are the sorts of questions that one would want to ask about a proposed distributed solution, and hence could be useful as a tool for individuals responsible for the development of a complex system. The author of this text is sometimes asked to comment on proposed systems designs, and like many others, has found that it can be difficult to know where to start when the time for questions finally arrives after a two-hour technical presentation. A reasonable suggestion is to begin to pose simple questions aimed at exposing the reliability properties and non-properties of the proposed system, the assumptions it makes, the dependencies embodied in it, and the cost/benefit tradeoffs reflected in the architecture. Such questions may not lead to a drastically changed system, but they do represent a path towards understanding the mentality of the designer and the philosophical structure of the proposed system. Many of the questions below are of the same nature that might be used in such a situation.

1. Write a program to experimentally characterize the packet loss rate, frequency of out-of-order delivery, send-to-receive latency, and byte throughput of the UDP and TCP transport protocols available on your computer system. Evaluate both the local case (source and destination on the same machine) and the remote case (source and destination on different machines).
2. We discussed the concept of a "broadcast storm" in conjunction with ethernet technologies. Devise an experiment that will permit you to quantify the conditions under which such a storm might arise on the equipment in your laboratory. Use your findings to arrive at a set of recommendations that should, if followed, minimize the likelihood of a broadcast storm even in applications that make heavy use of broadcast.
3. Devise a method for rapidly detecting the failure of a process on a remote machine and implement it. How rapidly can your solution detect a failure without risk of inaccuracy. Your work should consider one or more of the following cases: program that runs a protocol of your own devising implemented over UDP, program that is monitored by a parent program, program on a machine that fails or becomes partitioned from the network. For each case, you may use any system calls or standard communication protocols that are available to you.
4. Suppose that it is your goal to develop a network "radio" service that transmits identical data to a large set of listeners, and that you need to pick the best communication transport protocol for this purpose. Evaluate and compare the UDP, TCP and IP multicast transport protocols on your computer (you may omit IP multicast if this is not available in your testing environment). Your evaluation should look at throughput and latency (focusing on variability of these as a function of throughput presented to the transport). Can you characterize a range of performance within which one protocol is superior to the others in terms of loss rate, achievable throughput, and consistently low latency? Your results will take the form of graphs showing how these attributes scale with increasing numbers of destinations.

5. Develop a simple ping-pong program that bounces a UDP packet back and forth between a source and destination machine. One would expect such a program to give extremely consistent latency measurements when run on idle workstations. In practice however, your test is likely to reveal considerable variation in latency. Track down the causes of these variations and suggest strategies for developing applications with highly predictable and stable performance properties.

6. One challenge to timing events in a distributed system is that the workstations in that system may be running some form of clock synchronization algorithm that is adjusting clock values even as your test runs, leading to potentially confusing measurements. From product literature for the computers in your environment or by running a suitable experiment, determine the extent to which this phenomenon occurs in your testing environment. Can you propose ways of measuring performance that are immune to distortions of this nature?

7. Suppose that you wish to develop a *topology service* for a local area network, using *only* two kinds of information as "input" with which to deduce the network topology: IP addresses for machines, and measured point-to-point latency (for lightly loaded conditions, measured to a high degree of accuracy). How practical would it be to solve this problem? Ideally, a topology service should be able to produce a map showing how your local area network is interconnected, including bridges, individual ethernet segments, and so forth.

8. (Moderately difficult). If you concluded that you should be able to do a good job on the previous problem, implement such a topology service using your local area network. What practical problems limit the accuracy of your solution? What forms of use could you imagine for your service?

9. In Chapter 5, we saw that streams protocols could fail in inconsistent ways. Develop an application that demonstrates this problem by connecting two programs with multiple TCP streams, running them on multiple platforms, and provoking a failure in which some of the streams break and some remain connected. To do this test you may need to briefly disconnect one of the workstations from the network, hence you should obtain the permission of your network administration staff.

10. Propose a method for passing pointers to servers in an RPC environment, assuming that the source and destination programs are coded in C++ and that pointers are an abstract data type. What costs would a user of your scheme incur? Can you recommend programming styles or new programming constructs to minimize the impact of these costs on the running application? Contrast your solutions with those in Culler and Von Eicken's Split C programming environment.

11. (Requires sophistication in C++). Suppose that a CORBA implementation of the UNIX compression and decompression utilities is needed, and you have been asked to build it. Your utility needs to operate on arbitrary C++ objects of varied types. The types are not known in advance. Some of these objects will have a *compress_self* and a *decompress_self* interface but others will not. How could this problem be solved?

12. Can a CORBA application see a difference between CORBA remote invocations implemented directly over UDP and CORBA remote invocations implemented over a TCP-style reliable stream?

13. Suppose one were building a CORBA-based object oriented system for very long lived applications. The system needs to remain *continuously operational* for years at a time. Yet it is also expected that it will sometimes be necessary to upgrade software components of the system. Could such a problem be solved in software? That is, can a general purpose "upgrade" mechanism be designed as part of an application so that objects can be dynamically upgraded? To make this concrete, you can focus on a system of *k* objects, O_1, ... O_k and consider the case where we want to replace O_i with O_i' while the remaining objects remain unchanged. Express your solution by describing a proposed upgrade mechanism and the constraints it imposes on applications that use it.

14. Suppose that a CORBA system is designed to cache information at the clients of a server. The clients would be bound to *client objects* which would handle the interaction with the remote server. Now, consider the case where the data being cached can be dynamically updated on the server. What

options exist for maintaining the coherency of the cached data within the clients? What practical problems might need to be overcome in order to solve such a problem reliably? Does the possibility that the clients, the server, or the communication system might fail complicate your solution?

15. In CORBA we saw that it is possible to trap error conditions, such as server failure. Presumably, one would want to standardize the handling of such conditions. Suppose that you are designing a general purpose mechanism to handle "fail over" whereby a client connected to a server *S* will automatically and transparently rebind itself to server *S'* in the event that *S* fails. Under what conditions would this be easy? How would you deal with the possibility that the state of *S'* might not be identical to that of *S?* Could one detect such a problem and recover from it transparently?

16. Propose a set of extensions to the C++ IDL used in CORBA for the purpose of specifying reliability properties of a distributed server, such as fault-tolerance, real-time guarantees, or security.

17. Discuss options for handling the case where a transactional CORBA application performs operations on a non-transactional CORBA server.

18. (Moderately difficult; term project for a group). Build a CORBA-based web server and browser. What benefits or disadvantages might result from using a replication technology such as Orbix+Isis to replicate the server state and load-share clients among the servers in a process group? Experimentally test your expectations.

19. Each of the following is a potential reliability exposure for CORBA-based applications. Discuss the nature of the problem and the possible remedies. Do you feel that any of these is a "show stopper" for a typical large potential user, such as a bank with world-wide operations or a telecommunications company managing millions of lines of code and application programs?

    - Operator overloading and "unexpected consequences" of simple operations, like **a := b**
    - Exception handling when communicating with remote objects
    - The need to use CORBA "throughout" the distributed environment in order to benefit from the technology in a system-wide manner. Here, the implication might be that large amounts of old or commercially obtained code (some of which may not be well documented or even easily recompiled) may have to be modified to support CORBA IDL-style interface declarations and remotely accessible operations.

20. Suppose that a CORBA rebinding mechanism is to be used to automatically rebind CORBA applications to a working server if the server being used fails. What constraints on the application would make this a "safe" thing to do without notifying the application when rebinding occurs? Would this form of complete transparency make sense, or are the constraints too severe to use such an approach in practice?

21. A protocol that introduces tolerance to failures will also make the application that uses it more complex than one that makes no attempt to tolerate failures. Presumably, this complexity carries with it a cost in decreased application reliability. Discuss the pros and cons of building systems to be robust, in light of the likelihood that doing so will increase the cost of developing the application, the complexity of the resulting system, and the challenge of testing it. Can you suggest a principled way to reach a decision on the appropriateness of hardening a system to provide a desired property?

22. Suppose that you are using a conventional client-server application for a banking environment, and the bank requires that there be *absolutely no risk* of authorizing a client to withdraw funds beyond the limit of the account. Considering the possibility that the client systems may sometimes crash and need to be repaired before they restart, what are the practical implications of such a policy? Can you suggest other policies that might be less irritating to the customer while bounding the risk to the bank?

23. Suppose that you are developing a medical computing system using a client-server architecture, in which the client systems control the infusion of medication directly into an IV line to the patient. Physicians will sometimes change medication orders by interacting with the server systems. It is *absolutely imperative* that the physician be confident that an order he or she has given will be carried out, or that an alarm will be sounded if there is *any uncertainty whatsoever* about the state of the system. Provide an analysis of possible failure modes (client system crashes, server crashes) and the way that they should be handled to satisfy this reliability goal. Assume that the software used in the system is correct and that the only failures experienced are due to hardware failures of the machines on which the client and server systems run, or communication failures in the network.

24. Consider an air-traffic control system in which each flight is under the control of a specific individual at any given point in time. Suppose that the system takes the form of a collection of client-server distributed networks, one for each of a number of air traffic control centers. Design a protocol for handing off a flight from one controller to another, considering first the case of a single center and then the case of a multicenter system. Now, analyze the possible failure modes of your protocol under the assumption that client systems, server systems, and the communications network may be subject to failures.

25. (Term project) Using the Web, locate the specifications of the web server protocol (HTTP) over the network. Make a list of the *critical dependencies* of a typical web browser application. That is, list the technologies and servers that the browser "trusts" in its normal mode of operation. Now, suppose that you were concerned with possible *punning* attacks, in which a trusted server is replaced with a non-trustworthy server that mimics the behavior of the true one but in fact sets out to compromise the user. What methods could be used to reduce the exposure of your browsers to such attacks?

26. (Term project; team of two or more) Copy one of the public-domain web server sources to your system. In this textbook we have explored technologies for increasing distributed systems reliability using replication, fault-tolerance in servers, security tools, and coherent caching. Using protocols of your own, or Cornell's public Horus distribution, extend the web server to implement one or more of these features. Evaluate the result of your effort by comparing the before and after behavior of the server in the areas that you modified.

27. (Term project; team of two or more) Design a wide-area service for maintaining directory-style information in very large environments. Such systems implement a mapping from *name* to *value* for potentially large numbers of names. Implement your architecture using existing distributing computing tools. Now evaluate the quality of your solution in terms of performance, scaling, and reliability attributes. To what degree can your system be "trusted" in critical settings, and what technology dependencies does it have? Note: the X.500 standard specifies a directory service interface and might be a good basis for your design.

28. Use Horus to implement layers based on two or more of the best known *abcast* ordering protocols. Compare the performance of the resulting implementations as a function of load presented and the number of processes in the group receiving the message.

29. Suppose that a Horus protocol stack implementing Cristian's real-time atomic broadcast protocol will be used side-by-side with one implementing virtual synchronous process groups with *abcast,* both in the same application. To what degree might inconsistency be visible to the application when group membership changes because of failures of some group members? Can you suggest ways that the two protocol stacks might be "linked" to limit the time period during which such inconsistencies can occur? (Hard problem: implement your proposal).

30. Some authors consider RPC to be an extremely successful protocol, because it is highly transparent, reasonably robust, and can be optimized to run at very high speed — so high that if an application wants stronger guarantees, it makes more sense to layer a protocol over a lower-level RPC facility than to build it into the operating system at potentially high cost. Discuss the pros and cons of this point of view. In the best possible world, how would you design a communication subsystem?

31. Research the *end to end argument*. Does the goal of building reliable distributed systems bring aspects of this argument into question? Explain.

32. Review flow control options for multicast environments in which a small number of data sources send steady streams of data to large numbers of data sinks over hardware that supports a highly (but not perfectly) reliable multicast mechanism. How does the requirement that data be reliably delivered to all data sinks change the problem?

33. A protocol is said to be "acky" if most packets area acknowledged immediately upon reception. Discuss some of the pros and cons of this property. Suppose that a streams protocol could be switched in and out of an acky mode. Under what conditions would it be advisable to operate that protocol with frequent acks?

34. Suppose that a streaming style of multi-destination information service, such as the one in Problem 32, is to be used in a setting where a small subset of the application programs can be unresponsive for periods of time. A good example of such a setting would be a network in which the client systems run on PC's, because the most popular PC operating systems allow applications to preempt the CPU and inhibit interrupts, a behavior that can delay the system from responding to incoming messages in a timely manner. What options can you propose for ensuring that data delivery will be reliable and ordered *in all cases* but that small numbers of briefly unresponsive machines will not impact performance for the much larger number of highly responsive machines?

35. Several of the operating system technologies we reviewed gained performance by eliminating copying on the communication path between the communications device and the application that generates or consumes data. Suppose that you were building a large-scale distributed system for video-playback of short video files on demand. For example, such a system might be used in a large bank to provide brokers and traders with current projections for the markets and trading instruments tracked by the bank. What practical limits can you identify that might make it hard to use "zero copy" playback mechanisms between the file servers on which these video snippets are stored and the end-user who will see the result? Assume that the system is intended to work in a very general heterogeneous environment shared with many other applications.

36. Consider the Group Membership Protocol of Section 13.9. Suppose that this protocol was implemented in the address space of an application program, and that the application program contained a bug causing it to infrequently but randomly corrupt a few cells of memory. To what degree would this render the assumptions underlying the GMS protocol incorrect? What behaviors might result? Can you suggest practical countermeasures that would overcome such a problem if it was indeed very infrequent?

37. (Difficult) Again, consider the Group Membership Protocol of Section 13.9. This protocol has the property that all participating processes observe *exactly the same sequence* of membership views. The coordinator can add unlimited numbers of processes in each round, and can drop any minority of the members each time it updates the system membership view; in both cases, the system is provably immune from partitioning. Would this protocol be simplified by eliminating the property that processes must observe the same view sequence? (Hint: try to design a protocol that offers this "weaker" behavior). What about the partition freedom property: would the protocol be simpler if this was not required?

38. Suppose that the processes in a process group are managing replicated data. Due to a lingering bug, it is known that although the group seems to work well for periods of hours or even days, over very long periods of time the replicated data can become slightly corrupted so that different group members have different values. Discuss the pros and cons of introducing a "stabilization" mechanism whereby the members would periodically exchange values and, if an inconsistency is developed, arbitrarily switch to the most common value or to the value of an agreed upon "leader." What issues might this raise in the application program, and how might they be addressed?

39. Implement a very simple banking application supporting accounts into which money can be deposited and permitting withdrawals. Have your application support a form of *disconnected operation* based on the two-tiered architecture, in which each branch system uses its own set of process groups and maintains information for local accounts. Your application should simulate partitioning failures through a command interface. If branches cache information about remote accounts, what options are there for permitting a client to withdraw funds while the local branch at which the account really resides is unavailable? Consider both the need for safety by the bank and the need for availability, if possible, for the user. For example, it would be silly to refuse a user $250 from an account that has thousands of dollars in it moments earlier when connections were still working! Can you propose a policy that is always safe for the bank, and yet also allows remote withdrawals during partition failures?

40. Design a protocol by which a process group implemented using Horus can solve the asynchronous consensus problem. Assume that the environment is one in which Horus can be used, that processes only fail by crashing, and the network only fails by losing messages with some low frequency. Your processes should be assumed to start with a variable $input_i$ that, for each process p_i is initially 0 or 1. After deciding, each process should set a variable $output_i$ to its decision value. The solution should be such that the processes all reach the same decision value v, and this value is the same as at least one of the inputs.

41. In regard to your solution to Problem 40, discuss the sense in which your solution "solves the asynchronous consensus problem". Would Horus be guaranteed to make progress under the stated conditions? Do these conditions correspond to the conditions of the asynchronous model used in the FLP and Chandra/Toueg results?

42. Can the virtual synchrony protocols of a system like Horus be said to guarantee safety and liveness in the general asynchronous model of FLP or the Chandra/Toueg results?

43. Suppose that you were responsible for porting the Horus system to a cluster-style processor known to consist of between 16 and 32 identical high speed computing nodes interconnected by a high speed ATM-style communications bus, and with a reliable mechanism for detecting hardware failures of nodes within a few microseconds after such events occur. Your goal in undertaking this port is to implement a "cluster API" providing standard cluster-oriented operating system services to applications developers. How would you consider changing Horus itself to adapt it better to this environment? Would the Horus Common Protocol Interface (HCPI) be a suitable cluster API, or would you implement some other layer over Horus; if the latter, what would your API include? Assume that an important goal is that the cluster be highly available, easily serviced and upgraded, and that it be possible to support highly available application programs with relative ease.

44. Can the virtual synchrony protocols of a system like Horus be said to guarantee safety and liveness in a cluster-style computer architecture such as the one described in Problem 43?

45. The Horus "stability" layer operates as follows. Each message is given a unique id, and is transmitted and delivered using the stack selected by the user. The stability layer expects the processes that receive the message to issue a downcall when they consider the message "locally stable." This information is relayed within the group, and each group member can obtain a matrix giving the stabilization status of pending messages originated within the group as needed. Could the stability layer be used in a way that would add the dynamic uniformity guarantee to messages sent in a group?

46. Suppose that a process group is created in which three member processes each implement different algorithms for performing the same computation (so-called "implementation redundancy"). You may assume that these processes interact with the external environment *only using message send and receive primitives*. Design a wrapper that compares the actions of the processes, producing a single output if two out of the three or all three processes agree on the action to take for a given input, and signaling an exception if all three processes produce different outputs for a given input. Implement

your solution using Horus and demonstrate it for a set of fake processes that usually copy their input to their output, but with small probability make a random change to their output before sending it.

47. A set of processes in a group monitor devices in the external environment, detecting *device service requests* to which they respond in a load-balanced manner. The best way to handle such requests depends upon the frequency with which they occur. Consider the following two extremes: requests that require long computations to handle but that occur relatively infrequently, and requests that require very short computations to handle but that occur frequently on the time scale with which communication is done in the system. Assuming that the processes in a process group have identical capabilities (any can respond to any request), how would you solve this problem in the two cases?

48. Design a locking protocol for a virtually synchronous process group. Your protocol should allow a group member to *request* a lock, specifying the "name" of the object to be locked (the name can be an integer to simplify the problem), and to *release* a lock that it holds. What issues arise if a process holding a lock fails? Recommend a good, general way of dealing with this case, and then give a distributed algorithm by which the group members can implement the *request* and *release* interfaces as well as your solution to the broken lock case.

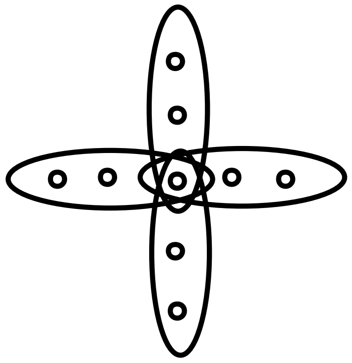


Figure 26-1: Overlapping process groups for the case of Problem 49. In this example there is only a single process in the overlap region; the problem concerns state transfer if we wanted to add another process to this region. Assume that the state of the processes in the overlap region reflects messages sent to it by the outer processes that belong to the "petals" but not the overlap area. Additionally, assume that this state is not cleanly decomposed group by group and hence that is necessary to implement a single state transfer for the entire structure.

49. (Suggested by Jim Pierce) Suppose that we want to implement a system in which *n* process groups will be superimposed much like the petals of a flower. Some small set of *k* processes will belong to all *n* groups, and each group will have additional members that belong only to it. The problem now arises of how to handle *join* operations for the processes that belong to the overlapping region, and in particular how to deal with state transfers to such a process. Assume that the group states are only updated by "petal" processes that do not belong to the overlap region. Now, the virtually synchronous state transfer mechanisms we discussed in Section 15.3.2 would operate on a group by group basis, but it may be that the states of the processes in the overlap region are a mixture of information arriving from all of the petal processes. For such cases one would want to do a *single* state transfer to the joining process reflecting the *joint state* of the overlapped groups. Propose a fault-tolerant protocol for joining the overlap region and transferring state to a joining process that will satisfy this objective.

50. Discuss the pros and cons of using an *inhibitory* protocol to test for a condition along a consistent cut in a process group. Describe a problem or scenario where such a solution might be appropriate, and one where it would not be.

51. Suppose that the processes in a distributed system share a set of resources, which they lock prior to using and then unlock when finished. If these processes belong to a process group, how could deadlock detection be done within that group? Design your deadlock detection algorithm to be completely idle (with no background communication costs) when no deadlocks are suspected; the algorithm should be one that can be launched when a time-out in a waiting process suggests that a deadlock may have occurred. For bookkeeping purposes, you may assume that a process that is waiting for a resource calls the local procedure *waiting_for(resource),* that a process that holds exclusive access to a resource calls the procedure *holding(resource),* and that a process that releases a resource calls *release(resource),* where the resources are identified by integers. Each process thus maintains a local database of its "resource status". Notice that you are not being asked to implement the actual mutual exclusion algorithm here: your goal is to devise a protocol that can interact with the processes in the system as needed, to accurately detect deadlocks. Prove that your protocol detects deadlocks if and only if they are present.

52. Suppose that you wish to monitor a distributed system for an overload condition, defined as follows. The system state is considered normal if no more than 1/3 of the processes signal that they are overloaded, heavily loaded if more than 1/3 but less than 2/3 of the processes signal that they are overloaded, and seriously overloaded if 2/3 or more processes are overloaded. Assume further that the loading condition does not impact communication performance. If the processes belong to a process group, would it be sufficient to simply send a multicast to all members asking their states, and then to compute the state of the system from the vector of replies so obtained? What issues would such an approach raise, and under what conditions would the result be correct?

53. (Joseph and Schmuck). What would be the best way to implement a *predicate addressing* communication primitive for use within virtually synchronous process groups (assume that the group primitives are already implemented and available for you). Such a primitive sends a message to *all the processes in the group for which some acceptance criteria holds* and does so *along a consistent cut*. You may assume that each process contains a predicate *accept()* that, at the time it is invoked, returns *true* if the process wishes to accept a copy of the message and *false* if not. (Hint: it is useful to consider two separate cases here: one in which the criteria that determine acceptance change "slowly" and one in which they change "rapidly", relative to the speed of multicasting in the system).

54. In discussing the notion of wrappers, we developed the example of a *world wide memory* system, in which shared memory primitives are redefined to permit programs to share access to very large scale distributed memories maintained over an ATM-style network. Suppose that you were implementing such a system using Horus over the Unet system on a wide-area ATM, and that you knew the expected application to be as an in-memory server for web pages. These pages will in some cases be updated rapidly (at video speeds) and for that purpose your browser will have the ability to memory map *video image* objects directly to the display of the viewing computer. What special design considerations are implied by this intended application? Recall that the memory architecture we developed had a notion of *prefetching* built into it, much like a traditional virtual memory subsystem would have. How should prefetching be implemented in your mapped memory system.

55. (Difficult; team programming project). Implement the architecture you proposed in Problem 54, focusing however on the case of side-by-side computers with a high speed link between them.

56. (Difficult, research topic). Implement a world-wide memory system such as the one discussed in Problem 54 and develop a detailed justification and evaluation of the architecture you used.

57. (Schneider). We discussed two notions of clock synchronization: *accuracy* and *precision*. Consider the case of aircraft that operate under *free flight* rules, where each pilot makes routing decisions on behalf of his (her) plane, but using a shared trajectory "mapping" system. Suppose that you faced a fundamental tradeoff between using clocks with high accuracy for such a mapping system, or clocks with high precision. Which would you favor, and why? Would it make sense to implement two such solutions, "side by side"?

58. Suppose that a-posteriori clock synchronization using GPS receivers becomes a world-wide standard in the coming decade. The use of temporal information now represents a form of communication channel that can be used in indirect ways. For example, process *p,* executing in Lisbon, can wait until process *q* performs a desired operation in New York (or fails) using timer events. Interestingly, such an approach communicates "information" faster than messages could possibly have done so. What issues do these sorts of hidden information channels raise in regard to the protocols we explored in the textbook? Could temporal information create hidden causality relationships?

59. Show how tightly synchronized real-time clocks can be made to reflect causality in the manner of Lamport's logical clocks. Would such a clock be preferable in some ways to a purely logical clock? Explain, giving concrete examples to illustrate your points.

60. (Difficult) In discussion of the CASD protocols, we saw that if such protocols are used to replicate the state of a distributed system, a mechanism would be needed to overcome inconsistencies that can arise when a process is technically considered "incorrect" according to the definitions of the protocols, and hence does not benefit from the normal guarantees of atomicity and ordering seen by "correct" processes. In an IBM technical report, Skeen and Cristian once suggested that the CASD protocols could be used in support of an abstraction called Δ-*common storage;* the basic idea being to implement a distributed shared memory which can be read by any process and is updated using the CASD style of broadcast protocol. Such a distributed shared memory would reflect an update within Δ time units after it is initiated, plus or minus a clock skew factor of ε. How might the inconsistency issue of the CASD protocol be visible in a Δ-common storage system? Propose a method for detecting and eliminating such inconsistencies. (Note: this issue was not considered in the technical report).

61. (Marzullo and Sabel) Suppose that you wish to monitor a distributed system to detect situations in which a logical predicate defined over the states of the member processes holds. For example, the predicate may state that process p_i holds a token and that process p_j is waiting to obtain the token. Under the assumption that the states in question change very slowly in comparison to the communication speeds of the system, design a solution to this problem. You may assume that there is a function, *sample_local_state(),* that can be executed in each process to sample those aspects of its local state referenced in the query, and that when the local states have been assembled in one place, a function *evaluate* can determine if the predicate holds or not. Now, discuss the modifications needed if the rate of state changes is increased enough so that the state can change in the same order of time as your protocol needs to run. How is your solution affected if you are required to detect *every state in which the predicate holds*, as opposed to just detecting *states in which the predicate happens to hold when the protocol is executed*. Demonstrate that your protocol cannot falsely detect satisfying states.

62. There is increasing interest in building small multiprocessor systems for use in inexpensive communications satellites. Such systems might look similar to a rack containing a small number of conventional workstations or PC's, running software that handles such tasks as maintaining the proper orientation of the satellite by adjusting its position periodically, turning on and off the control circuits that relay incoming messages to outgoing channels, and handle other aspects of satellite function. Now, suppose that it is possible to put highly redundant memory modules on the satellite to protect extremely critical regions of memory, but costly to do so. However, unprotected memory is likely to experience a low level of corruption arising from the harsh conditions in space, such as cosmic rays and temperature extremes. What sorts of programming considerations would such a model raise? Propose a software architecture that minimizes the need for redundant memory, but also minimizes the risk that a satellite will be completely lost (for example, a satellite might be lost if it erroneously fires its positioning rockets and thereby exhausts its supply of fuel). You may assume that the actual rate of corruption of memory is low, but not completely insignificant, and that program instructions are as likely as data to be corrupted. Assume that the extremely reliable memories, however, never experience corruption.

63. Continuing on the topic of Problem 62, there is debate concerning the best message routing architecture for these sorts of satellite systems. In one approach, the satellites maintain a routing network among themselves; a relatively small number of ground stations interact with whatever satellite happens to be over them at a given time, and control and data messages are then forwarded satellite to satellite until they reach the destination. In a second approach, satellites communicate only with ground stations and mobile transmitter/receiver units: such satellites require a larger number of ground systems but do not depend upon a routing transport protocol that could be a source of unreliability. Considering the conditions cited in Problem 62 and your responses, what would be best design for a satellite-to-satellite routing network? Can you suggest a scientifically sound way to make the design tradeoff between this approach and the one that uses a larger number of potentially costly ground-stations?

64. We noted that the theoretical community considers a problem to be "impossible" in a given environment if, for all proposed solutions to the problem, there exists at least one behavior consistent with the environment that would prevent the proposed solution from terminating, or would lead to an incorrect outcome. Later we considered probabilistic protocols, which may be able to guarantee behaviors to very high levels of reliability — higher, in practice, than the reliability of the computers on which the solutions run. Suggest a definition of *impossible* that might reconcile these two perspectives on computing systems.

65. If a message must be take *d* hops to reach its destination and the worst-case delay for a single link is δ, it is common to assume that the worst-case transit time for the network will be $d*\delta$. However, a real link will typically exhibit a distribution of latencies, with the vast majority clustered near some minimum latency δ_{min} and only a very small percentage taking as long as δ_{max} to traverse the link. Under the assumption that the links of a routed network provide statistically independent and identical behavior, derive the distribution of expected latencies for a message that must traverse *d* links of a network. You may assume that the distribution of delays has a "convenient" form for your analysis.

66. Suppose that a security architecture supports *revocation* of permissions. Thus: XYZ was permitted to access resource ABC, but now has finished the task for which permission was granted and we want to disable future accesses. Would it be safe to use a remote procedure call from the authentication server to the resource manager for resource ABC to accomplish this revocation? Explain.

67. (Ethical problem). Suppose that a medical system does something that a human would not be able to do, such as continuously monitoring the vital signs of a patient and continuously adjusting some form of medication or treatment in response to the measured values. Now, imagine that we want to attach this device to a distributed system so that physicians and nurses elsewhere in the hospital can remotely monitor the behavior of the medical system, and so that they can change the rules that control its actions if necessary (for example by changing the dosage of a drug). In this text we have encountered many practical limits to security and reliability. Identify some of the likely limits on the reliability of a technology such as this. What are the ethical issues that need to be balanced in deciding whether or not to build such a system?

68. (Ethical problem). An *ethical theory* is a set of governing principles or rules for resolving ethical conflicts such as the one in the previous problem. For example, an ethical theory might stipulate that decisions should be made to favor the "maximum benefit for the greatest number of individuals." A theory governing the deployment of technology could stipulate that "machines must not replace humans if the resulting system is at risk of making erroneous decisions that a human would have avoided." Notice that these particular theories could be in conflict, for example if a technology that would normally be beneficial sometimes has life-threatening complications. Discuss the issues that arise in developing an ethical theory for the introduction of technologies in life- or safety-critical settings, and, if possible, propose such a theory. What tradeoffs are required, and how would you justify them?

Bibliography

[AAD93] O. Amir, Yair Amir and Danny Dolev. A Highly Available Application in the Transis Environment. In *Proceedings of the Workshop on Hardware and Software Architectures for Fault-Tolerance*. Springer-Verlag Lecture Notes in Computer Science 774 (June 1993), 125—139.

[ABHN91] Mustaque Ahamad, James Burns, Phillip Hutto and Gil Neiger. Causal Memory. Technical Report, College of Computing, Georgia Institute of Technology. July 1991

[ABLL91] Tom Anderson, Brian Bershad, Ed Lazowska and Hank Levy. Scheduler Activations: Effective Kernel Support for the User-Leve; Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Oct. 1991), 95—109.

[ABM87] Noga Alon, Amnon Barak and Udi Manber. On Disseminating Information Reliably Without Broadcasting. *Proceedings of the 7th International Conference on Distributed Computing Systems* (Berlin, Sept. 1987), 74—81. IEEE Computer Society Press.

[ACP95] Tom E. Anderson, David E. Culler and David A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro,* Feb. 1995.

[ACBM95] Emmanuelle Anceaume, Bernadette Charron-Bost, Pascale Minet and Sam Toueg. On the Formal Specification of Group Membership Services. Technical Report 95-1534, Dept. of Computer Science, Cornell University. Aug. 1995.

[ADKM92a] Yair Amir, Danny Dolev, Shlomo Kramer, Dalia Malki. Transis: A Communication Subsystem for High Availability. In *Proceedings of the 22nd Symposium on Fault-Tolerant Computing Systems;* (Boston, MA; July 1992). IEEE. 76—84

[ADKM92b] Yair Amir, Danny Dolev, Shlomo Kramer, Dalia Malki. Membership Algorithms in Broadcast Domains. In *Proceedings of the 6th WDAG;* (Isreal, 1992). Springer Verlag Lecture Notes in Computer Science 647, 292—312.

[ADNP95] Tom Anderson, Michael Dahlin, *et al*. Serverless Network File Systems. In *Proceedings of the 15th Symposium on Operating Systems Principles;* (Copper Mountain Resort, CO; Dec. 1995). ACM. (109—126). Also appearing in the special issue of *ACM Transactions on Computing Systems,* 13:1 (Feb. 1996).

[AE84] Baruch Awerbuch and Shimon Even. Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network. *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing* (Vancouver, CA; 1984), 278—281.

[AGHR89] Francois Armand, Michel Gien, Frederic Herrmann and Marc Rozier. Revolution 89, o Distributing UNIX Brings it Back to Its Original Virtues. Technical Report CS/TR-89-36-1, Chorus Systemes, Paris, France. Aug. 1989.

[AJ95] Jo Asplin and Dag Johansen. Performance Experiments with the StormView Distributed Parallel Volume Renderer. Computer Science Technical Report 95-22, June 1995, University of Tromso.

[AK93] R. Alonso and F. Korth. Database Issues in Nomadic Computing. *Proceedings ACM SIGMOD International Conference on Management of Data*. (Washington D.C; May 1993), 388—392.

[Ami95] Yair Amir. Replication Using Group Communication Over a Partitioned Network. PhD thesis, Hebrew University of Jerusalem, 1995.

[AM95] Lorenzo Alvisi and Keith Marzullo. Message Logging: Pressimistic, Causal and Optimistic. *Proceedings 15th IEEE Conference on Distributed Computing Systems* (Vancouver, CA; 1995). 229-236.

[AMMA93] Yair Amir, Louise Moser, P.M. Melliar-Smith, *et. al*. The Totem Single-Ring Ordering and Membership Protocol. In *ACM Transactions on Computer Systems,* to appear.

[And91] Andrews, Gregory R. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, CA, 1991.

[ANSA89] The Advanced Networked Systems Architecture: An Engineer's Introduction to the Architecture. Architecture Projects Management Limited TR-03-02, November 89.

[ANSA91a] The Advanced Networked Systems Architecture: A System Designer's Introduction to the Architecture. Architecture Projects Management Limited RC-253-00, April 1991.

[ANSA91b] The Advanced Networked Systems Architecture: An Application Programmer's Introduction to the Architecture. Architecture Projects Management Limited TR-017-00, November 1991.

[AP93] Mark Abbott and Larry Peterson. Increasing Network Throughput by Integrating Protocol Layers. *IEEE/ACM Transactions on Networking* 1:5 (Oct. 1993), 600—610.

[Aga94] D. A. Agarwal. Totem: A Reliable Ordered Delivery Protocol for Interconnected Local Area Networks. PhD Thesis, U.C. Santa Barbara Dept. of Electrical and Computer Engineering, 1994.

[Bac90] Thomas C. Bache *et. al*. The Intelligent Monitoring System. *Bulletin of the Seismological Society of America,* 80:6 (Dec. 1990), 59—77.

[Bai75] Normal Bailey. *The Mathematical Theory of Epidemic Diseases*. Charles Griffen and Company, London. Second edition, 1975.

[Bar81] Joel F. Bartlett. A NonStop Kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles;* (Pacific Grove, CA; Dec. 1981). ACM. 22—29.

[BALL89] Brian Bershad, Tom Anderson, Ed Lazowska and Hank Levy. Lightweight Remote Procedure Call. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Litchfield Springs, AX; Dec. 1989). 102—113. Also *ACM Transactions on Computer Systems* 8:1 (Feb. 1990), 37—55.

[BAN89] Michael Burrows, Martin Abadi, Roger Needham. A Logic of Authentication. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Litchfield Springs, AX; Dec. 1989). ACM. 1—13.

[BBG83] Anita Borg, J. Baumbach and S. Glazer. A Message System for Supporting Fault-Tolerance. In *Proceedings 9th Symposium on Operating Systems Principles* (Bretton Woods, NH; Oct. 1993). 90—99.

[BBG96] Ozalp Babaoglu, Alberto Bartoli, Gianluco Dini. Enriched View Synchrony: A Paradigm for Programming Dependable Applications in Partitionable Asynchronous Distributed Systems. Technical Report, Dept. of Computer Science, University of Bologna. May 1996.

[BBGH85] Anita Borg, *et. al*. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*. 3:1 (Feb. 1985). 1—23.

[BBMS93] N. Budhiraja, *et. al*. The Primary-Backup Approach. In S.J. Mullender, editor. *Distributed Systems (second edition).* ACM-Press (Addison-Wesley), 1993.

[BCLF94] T. Berners-Lee, *et. al*. The World-Wide Web. *Communications of the ACM* 37:8 (August 1994), 76—82.

[BCLF95] T. Berners-Lee, *et. al*. *Hypertext Transfer Protocol —HTTP 1.0*. IETF HTTP Working Group Draft 02 (Best Current Practice), Aug. 1994.

[BCGP92] T. Berners-Lee, . Calliau, J-F. Groff and B. Pollermann. World-Wide Web: The Information Universe. *Electronic Networking Research, Applications and Policy* 2:1 (1992), 52—58.

[BD85] Ozalp Babaoglu and Rogerio Drummond. The Streets of Byzantium: Network Architectures for Fast, Reliable Broadcasts. *IEEE Transactions on Software Engineering*. 11:6 (June 1985), 546—554.

[BD87] Ozalp Babaoglu and Rogerio Drummond. (Almost) No Cost Clock Synchronization. In *Proceedings 17th International Symposium on Fault-Tolerant Computing*. (Pittsburgh, PA; July 1987).

[BD95] T. Braun and C. Diot. Protocol Implementation Using Intergrated Layer Processing. In *Proceedings of SIGCOMM-95* (Sept. 1995).

[BDGB94] Ozalp Babaoglu, Renzo Davoli, Luigi-Alberto Giachini and Mary Gray Baker. *RELACS: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems*. BROADCAST Project Deliverable Report 1994. Department of Computing Science, University of Newcastle upon Tyne, UK.

[BDM95] Ozalp Babaoglu, R. Davoli, A. Montresor. Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems, Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, Italy, November 1995.

[Be83] Michael Ben-Or. Fast Asynchronous Byzantine Agreement. *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing* (Minaki, CA; Aug. 1985), 149—151.

[BEM91] Anupam Bhide, Elmootazbellah N. Elnozahy and Stephen P. Morgan. A Highly Available Network File Server. In *Proceedings of the USENIX Winter Conference*. USENIX, Dec. 1991. 199—205.

[BG95] Kenneth P. Birman and Bradford B. Glade. Consistent Failure Reporting in Reliable Communications Systems. *IEEE Software,* Special Issue on Reliability, April 1995.

[BGH87] Joel Bartlett, Jim Gray and B. Horst. Fault Tolerance in Tandem Computing Systems. In *Evolution of Fault-Tolerant Computing*. Springer-Verlag, 1987. 55—76.

[BHG87] Philip E. Bernstein, Vassos Hadzilacos and Nat Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BHKSO91] Mary Baker *et. al*. Measurements of a Distributed File System. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Orcas Island, WA; Nov. 1991), 198-212.

[BHL93] Edoardo Biagioni, Robert Harper and Peter Lee. Standard ML Signatures for a Protocol Stack. Department of Computer Science Technical Report CS-93-170, Carnegie Mellon University, Oct. 1993.

[Bia94] Edoardo Biagioni. A Structured TCP in Standard ML. In *Proceedings of the 1994 Symposium on Communications Architectures and Protocols;* (London, Aug. 1994). ACM.

[Bir85] Andrew Birrell. Secure Communication Using Remote Procedure Calls. *ACM Transactions on Computer Systems;* 3:1 (Feb. 1985), 1—14.

[Bir93] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM;* 36:12 (Dec. 1993).

[Bir94] Kenneth P. Birman. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. *Operating Systems Review 28:1* (Jan. 1994), 11-21.

[BJ87a] Kenneth P. Birman and Thomas A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the 11th Symposium on Operating Systems Principles* (Austin, TX, Nov. 1987). ACM. 123—138.

[BJ87b] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presense of Failures. *ACM Transactions on Computer Systems* 5:1 (February 1987), 47—76.

[BKT90] Henri E. Bal, Robbert van Renesse and Andrew S. Tanenbaum. Implementing Distributed Algorithms Using Remote Procedure Call. In *Proceedings of the 1987 National Computer Conference* (Chicago, IL; June 1987). ACM. 499—506.

[BKT92] Henri E. Bal, M. Frans Kaashoek and Andrew S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Trans. on Software Engineering* (Mar. 1992), 190—205.

[BM90] S. M. Bellovin and Michael Merritt. Limitations of the Kerberos Authentication System. *Computer Communication Review,* 20:5 (Oct. 1990), 119—132.

[BM93] Ozalp Babaoglu and Keith Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In *Distributed Systems (2nd Edition),* S.J. Mullender, editor. ACM Press (Addison-Wesley), 1993.

[BMP94] L. Brakmo, Sean O'Malley, Larry Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *Proceedings ACM SIGCOMM '94* (London, England; 1994).

[BMRS94]. Kenneth P. Birman, Dalia Malki, Aleta Ricciardi, Andre Schiper. Uniform Action in Asynchronous Dist Sys. Cornell University Dept. of Computer Science Technical Report TR 94-1447, 1994.

[BN84] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Call. *ACM Transactions on Programming Languages and Systems* 2:1 (February 1984), 39—59.

[BNJL86] Andrew Black, Norm Hutchinson, Eric Jul and Hank Levy. Object Structure in the Emerald System. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (Portland, OR; Oct. 1986).

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki and T. Wobber. Network Objects. In *Proceedings of the 14th Symposium on Operating Systems Principles* (1993), 217—230.

[BR94] Kenneth P. Birman, Robbert van Renesse, eds. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[BR96] Kenneth P. Birman and Robbert van Renesse. Software for Reliable Networks. *Scientific American* 274:5 (May 1996), 64-69.

[Bro94] K. Brockschmidt. *Inside OLE-2*. Microsoft Press, 1994.

[BS95] Thomas C. Bressoud, Fred B. Schneider. Hypervisor-based Fault-tolerance. In *Proceedings of the 15th Symposium on Operating Systems Principles;* (Copper Mountain Resort, CO; Dec. 1995). ACM. (1—11). Also appearing in the special issue of *ACM Transactions on Computing Systems,* 13:1 (Feb. 1996).

[BSPS95] Brian Bershad *et. al*. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating Systems Principles* (Copper Mountain Resort, CO; Dec. 1995), 267—284.

[BSS91] Kenneth P. Birman, Andre Schiper and Patrick Stephenson. Lightweight Causal and Atomic Group Communication. *ACM Transactions on Computing Systems,* 9:3 (August 1991), 272—314.

[BW92] Anita Borr and Carol Wilhelmy. Highly Available Data Services for UNIX Client-Server Networks: Why Fault-Tolerant Hardware Isn't the Answer. . In *Hardware and Software Architectures for Fault-Tolerance,* Michel Banatre and Peter Lee, *eds*. Springer Verlag Lecture Notes in Computer Science vol. 774. 385-304.

[Car93] John Carter. Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency. PhD thesis, Rice University, August 1993.

[CASD85] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing,* IEEE, 1985. 200—206. Revised as IBM Technical Report RJ5244.

[CB94] Kenjiro Cho and Kenneth P. Birman. A Group Communication Approach for Mobile Computing. Computer Science Department Technical Report TR94-1424, Cornell University, May 1994.

[CD90] Flaviu Cristian and Robert Delancy. Fault-Tolerance in the Advanced Automation System. IBM Technical Report RJ7424; IBM Research Laboratories, San Jose, Calfornia, April 1990.

[CD95] David R. Cheriton and K. J. Duda. Logged Virtual Memory. In *Proceedings of the 15th Symposium on Operating Systems Principles;* (Copper Mountain Resort, CO; Dec. 1995). 26—39.

[CDK94] George Coulouris, Jean Dollimore and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 1994.

[CDSA90] Flaviu Cristian, Danny Dolev, Ray Strong and Houtan Aghili. Atomic Broadcast in a Real-Time Environment. In *Fault Tolerant Distributed Computing*. Springer-Verlag Lecture Notes in Computer Science 448, 1990. 51—71.

[Chau81] David Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2):84-88, February 1981.

[Chill92] Ram Chillaragee. Top Five Challenges Facing the Practice of Fault-Tolerance. In *Hardware and Software Architectures for Fault-Tolerance,* Michel Banatre and Peter Lee, *eds*. Springer Verlag Lecture Notes in Computer Science vol. 774. 3-12.

[CHT92] Tushar D. Chandra, Vassos Hadzilacos and Sam Toueg. The Weakest Failure Detector for Solving Consensus. In *ACM Symposium on Principles of Distributed Computing* (Aug. 1992). 147—158.

[CHTC96] Tushar Chandra, Vassos Hadzilacos, Sam Toueg and Bernadette Charron-Bost. On the Impossibility of Group Membership. *Proceedings ACM Symposium on Principles of Distributed Computing* (May 1996).

[CJRS89] David Clark, Van Jacobson, J. Romkey, H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications* 27:6 (June 1989), 23—29.

[CG90] Doug Comer and J. Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *Proceedings of the 1990 Summer USENIX Conference* (June 1990), 127—135.

[CF94] Flaviu Cristian and C. Fetzer. Fault-Tolerant Internal Clock Synchronization. *Proceedings of the 13th Symposium on Reliable Distributed Systems*. Oct, 1994.

[Cha91] B. Charron-Bost. Concerning the Size of Logical Clocks in Distributed Systems. *Information Processing Letters* 39:1 (Jul. 1991), 11—16.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems;* 3:1 (Feb. 1985), 63—75.

[CLFL94] Jeff Chase, Hank Levy, M. Feeley and Ed Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems* 12:4 (Nov. 1994), 271—307.

[CLJK94] Peter Chen, *et. al*. RAID: High Performance, Reliable, Secondary Storage. *ACM Computing Surveys* 26:2 (June 1994), 45—85.

[CM84] Jo-Mei Chang and Nick Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems;* 2:3 (August 1984), 251—273.

[CM96a] Callahan, J. and T. Montgomery, Approaches to verification and validation of a reliable multicast protocol, in Proceedings of the 1996 ACM International Symposium on Software Testing and Analysis, January 1996, San Diego, CA, pp. 187-194. Also appears as ACM Software Engineering Notes, May 1996, Volume 21, Number 3, pp. 187-194.

[CM96b] Matthew Clegg and Keith Marzullo. Clock Synchronization in Hard Real-Time Distributed Systems. University of California, San Diego, Dept. of Computer Science Technical Report, March 1996.

[COK86] Brian Coan, B.M. Oki, and E.K. Kolodner. Limitations on Database Availability When Networks Partition. *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*. (Calgary, CA; August 1986) 187—194.

[Com91] Doulas E. Comer. *Internetworking With TCP/IP.* Volume I: *Principles, Protocols and Architecture*. Prentice-Hall, 1991.

[Coo94] Robert Cooper. Experience with Causally and Totally Ordered Group Communication Support -- A Cautionary Tale. *Operating Systems Review 28:1* (Jan. 1994), 28-32.

[Coo85] Eric Cooper. Replicated Distributed Programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA; Dec. 1985). ACM. 63—78.

[Coo95] David A. Cooper and Kenneth P. Birman. The design and implementation of a private message service for mobile computers. *Wireless Networks*, 1(3):297-309, October 1995.

[CP88] John Crowcroft and K. Paliwoda. A Multicast Transport Protocol. *Computer Communication Review* 18:4 (Aug. 1988), 247—256.

[Cri89] Flaviu Cristian. Probabilistic Clock Synchronization. *Distributed Computing 3:3, 1989* (146—158).

[Cri91a] Flaviu Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM* 34:2 (Feb. 1991), 57—78.

[Cri91b] Flaviu Cristian. Reaching Agreement on Processor Group Membership In Synchronous Distributed Systems. *Distributed Computing* 4:4 (April 1991), 175—187.

[Cri96] Flaviu Cristian. Synchronous and Asynchronous Group Communication. *Communications of the ACM* 39:4 (April 1996), 88-97.

[CS91] Doulas E. Comer and David L. Stevens. . *Internetworking With TCP/IP*. Volume II: *Design, Implementation and Internals*. Prentice-Hall, 1991.

[CS93] Doulas E. Comer and David L. Stevens. . *Internetworking With TCP/IP*. Volume III: *Client-Server Programming and Applications*. Prentice-Hall, 1991.

[CS93] David Cheriton and Dale Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Dec. 1993). ACM. 44—57.

[CS95] Flaviu Cristian and Frank Schmuck. Agreeing on Process Group Membership in Asynchronous Distributed Systems. Technical Report CSE95-428, Department of Computer Science and Engineering, U.C. San Diego. 1995.

[CT87] D. Clark and M. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of SIGCOMM-87* (Aug. 1987), 353—359.

[CT90a] Brian A. Coan and G. Thomas. Agreeing on a Leader in Real-Time. In *Proceedings of the 11th Real-Time Systems Symposium* (Dec. 1990), 166—172.

[CT90b] Tushar Chandra and Sam Toueg. Time and Message Efficient Reliable Broadcasts. Cornell University Dept. of Computer Science, TR 90-1094, February 1990.

[CT91] Tushar Chandra and Sam Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Journal of the ACM*. To appear, previous version in PODC 1991, 325—340.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA. 1993.

[CZ85] David R. Cheriton and Willy Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems,* 3:2 (May 1985), 77—107.

[CT90] David D Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In. *Proceedings of the 1990 Symposium on Communication Architectures and Protocols* (Philadelpha, PA; Sept. 1990). ACM. 200—208.

[DB93] Rogerio Drummond and Ozalp Babaoglu. Low-Cost Clock Synchronization. *Distributed Computing* 6:, 1993. 193-203.

[DB96] Dorothy Denning and Dennis Branstad. A Taxonomy for Key Escrow Encryption Systems. *Communications of the ACM 39:3* (March 1996), 34-40.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems* 8:2 (May 1990), 85—110.

[DCE94] Open Software Foundation. *Introduction to OSF DCE*, Prentice Hall, Englewood Cliff, NJ, 1994.

[DEC95] Digital Equipment Corporation. A Technical Description of the DECsafe Available Server Environment (ASE). *Digital Equipment Corporation Technical Journal 7:4,* (Sept. 1995). 89-100.

[Dee88] Steve E. Deering. Multicast Routing in Internetworks and Extended LANs. *Computer Communications Review* 18:4 (Aug. 1988), 55—64.

[Dee89] Steve E. Deering. Host Extensions for IP Multicasting. RFC 1112, SRI Network Information Center, August 1989.

[Den84] Dorothy Denning. Digital Signatures With RSA and Other Public-Key Cryptosystems. *Communications of the ACM,* 27:4 (April 1984), 388—392.

[DEFJ88] C. Anthony DellaFera *et. al*. The Zephyr Notification Service. In *Proceedings of the Winter USENIX Conference*. USENIX. Dec. 1988.

[DES77] *Data Encryption Standard*. National Bureau of Standards, Federal Information Processing Standards Publication 46, Government Printing Office, Washington DC 1977.

[Des88] Y. Desmedt. Society and Group-Oriented Cryptography: A New Concept. In *Advances in Cryptology — CRYPTO '87 Proceedings*. Springer-Verlag Lecture Notes in Computer Science 293 (1988). 120—127.

[DFW90] Bert Dempsey, John C. Fenton and Alfred C. Weaver. The MultiDriver: A Reliable Multicast Service Using the Xpress Transfer Protocol. In *Proceedings 15th Conference on Local Computer Networks* (1990), IEEE Computer Society, 351—358.

[DFY92] Y. Desmedt, Y. Frankel and M. Yung. Multi-Receiver / Multi-Sender Network Security: Efficient Authenticated Multicast / Feedback. In *Proceedings of IEEE INFOCOM,* May 1992.

[DGHI87] A. Demers *et. al*. Epidemic Algorithms for Replicated Data Management. *Proceedings of the 6th Symposium on Principles of Distributed Computing*. (Vancouver, CA; Aug. 1987) 1—12. Also *Operating Systems Review* 22:1 (Jan. 1988), 8—32.

[DGS85] S. Davidson, H. Garcia-Molina and D. Skeen. Consistency in a Partitioned Network: A Survey. *ACM Computing Surveys* 17:3 (Sept. 1985), 341—370.

[DH79] W. Diffie and M. E. Hellman. Privacy and Authentication: An Introduction to Cryptography. *Proceedings of the IEEE,* 67:3 (March 1979), 397—427.

[Dif88] W. Diffie. The First Ten Years of Public-Key Cryptography. *Proceedings of the IEEE*. 76:5 (May 1988), 56o—577.

[DM96] Danny Dolev and Dalia Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM* 39:4 (April 1996), 64-70.

[DMS95] Danny Dolev, Dalia Malki, and Ray Strong. A Framework for Partitionable Membership Service. TR 95-4, The Hebrew University of Jerusalem, Institute of Computer Science. March 1995.

[DP93] Peter Drushel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Dec. 1993). ACM. 189—202.

[DRSK89] A. Damm, J. Reisinger, W. Schwabl and H. Kopetz. The Real-Time Operating System of Mars. *ACM Operating Systems Review* 22:3 (July 1989), 141—157.

[EBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th Symposium on Operating Systems Principles;* (Copper Mountain Resort, CO; Dec. 1995). ACM. 40—53.

[ECGS92] Thorsten von Eicken, David E. Culler, S. C. Goldstein and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture* (May 1992), 256—266.

[EKO95] Dawson R. Engler, M. Frans Kaashoek and James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th Symposium on Operating Systems Principles;* (Copper Mountain Resort, CO; Dec. 1995). ACM. 251—266

[EMS95] Ezhilhelvan, P., Macedo, R. and Shrivastava, S. Newtop: A Fault-Tolerant Group Communication Protocol. In *Proceedings of the 15th International Conference on Distributed Systems* (Vancover, CA; May 1995).

[EZ92] E. N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent Rollback-Recovery With Low Overhead, Limited Rollback and Fast Output Control. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*. May 1992.

[FB96] Roy Friedman and Kenneth P. Birman. Using Group Communication Technology to Implement a Reliable and Scalable Distributed IN Coprocessor. To appear: *TINA '96: The Convergence of Telecommunications and Distributed Computing Technologies* (Heidelberg, Germany; Sept. 1996). Also available as a Cornell University Dept. of Computer Science Technical Report; March 1996.

[FD92] Y. Frankel and Y. Desmedt. Distributed Reliable Threshold Multisignature. Technical Report TR-92-0402, Dept. of EECS, University of Wisconsin at Milwaukee.

[Fid88] C. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference,* 1988.

[FJML95] Sally Floyd, Van Jacobson, Steven McCanne, Ching-Gung Liu, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proceedings of the '95 Symposium on Communication Architectures and Protocols*. ACM. August 1995, Cambridge MA.

[FKMBD95] Roy Friedman, Idit Keider, Dalia Malki, Kenneth P. Birman and Danny Dolev. Deciding in Partitionable Networks. Cornell University Computer Science Technical Report, 95-1554, Oct. 1995.

[FLP85] Michael J. Fischer, Nancy A. Lynch, Michael S. Patterson. Impossibility of Distributed Computing With One Faulty Process. *Journal of the ACM,* 32:2 (April 1985), 374—382.

[Fort95] Fortezza Application Developers Guide. Version 3.0 July 1995. US Government. Available at web site www.armadillo.de.us

[FMPK95] Michael Feeley, *et. al*. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles* (Copper Mountain Resort, CO; Dec. 1995), 201-212.

[FR95a] Roy Friedman and Robbert van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. Cornell University Dept. of Computer Science Technical Report No. 95-1527, July 1995. *Submitted to IEEE Transactions on Networking*.

[FR95b] Roy Friedman and Robbert van Renesse. Strong and Weak Virtual Synchrony in Horus. Cornell University Dept. of Computer Science Technical Report 95-1537 (Aug. 1995).

[Fra89] Y. Frankel. A Practical Protocol for Large Group-Oriented Networks. In *Advances in Cryptology — EUROCRYPT '89*. Springer-Verlag Lecture Notes in Computer Science 434, 1989. 56—61.

[FV95] Roy Friedman and Robbert van Renesse. Strong and Weak Virtual Synchrony in Horus. Cornell University Dept. of Computer Science Technical Report TR95-1537, 1995.

[FWB85] Ariel Frank, Larry Wittie and Arthur Bernstein. Multicast Communication on Network Computers. *IEEE Software* (May 1985).

[FZ91] Ed Felton and John Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, Univ. of Washington, Dept. of Computer Science and Engineering, Mar. 1991.

[GA91] Ramesh Govindran and David P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Asilomar, CA; Oct. 1991). ACM. 68—80.

[GBCR92] Bradford B. Glade, Kenneth P. Birman, Robert C. Cooper and Robbert van Renesse. Light-weight Process Groups in the Isis System. *Distributed Systems Engineering Journal,* July 1993.

[GBH87] Jim Gray, Joel Bartlett and Robert Horst. Fault-Tolerance in Tandem Computer Systems. In *The Evolution of Fault-Tolerant Computing*. Edited by A. Avizienis, H. Kopetz and J.C. Laprie, Springer-Verlag 1987.

[GDBJ94] Geist, G.A. *et. al. PVM*: *A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA 1994.

[GDS86] Robert F. Gurwitz, Michael Dean and Richard E. Schantz. Programming Support in the Chronus Distributed Operating System. *Proceedings of the Sixth International Conference on Distributed Computing Systems* (IEEE, 1986), 486-493.

[Gib94] B. Wayt Gibbs. Software's Chronic Crisis. *Scientific American*. Sept. 1994.

[Gif79] David Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles* (Pacific Grove, CA; Dec. 1979). ACM. 150—162.

[GK92] Richard Golding and Kim Taylor. Group Membership in the Epidemic Style. U.C. Santa Cruz, Dept. of Computer and Information Sciences, TR CRL-92-13 (May 1992).

[Gla96] Bradford B. Glade. *A Scalable Architecture for Reliable Publish/Subscribe Communication in Distributed Systems*. Cornell University Ph.D. dissertation, Department of Computer Science, May 1996.

[Gle94] Barry Gleeson. Fault Tolerant Computer System With Provision for Handling External Events. U.S. Patent 5,363,503 (Nov. 1994).

[GLLG90] K. Gharachorloo, *et. al*. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Anual International Symposium on Computer Architecture* (Seattle, WA; May 1990). 15—26.

[GMS91] Hector Garcia-Molina and Annemarie Spauster. Ordered and Reliable Multicast Communication. *ACM Transactions on Computer Systems* 9:3 (August 1991). 242—271.

[GM95a] James Gosling and H. McGilton. The Java Language Environment: A White Paper. Sun Microsystems Inc. October 1995. Available as http://java.sun.com/langEnv/index.html

[GM95b] James Gosling and H. McGilton. The Java Programmer's Guide: A White Paper. Sun Microsystems Inc. October 1995. Available as http://java.sun.com/progGuide/index.html

[Gol91] Richard A. Golding. Distributed Epidemic Algorithms for Replicated Tuple Spaces. Technical report HPL-CSP-91-15, June 1991. Concurrent systems project, Hewlett-Packard Laboratories.

[Gol92] Richard A. Golding. *Weak Consistency Group Communication and Membership*. PhD thesis, Computer and Information Sciences Department, U.C. Santa Cruz, 1992.

[Gon89] Li Gong. Securely Replicating Authentication Services. In *Proceedings of the 9th International Conference on Distributed Computing Systems,* 85—91.

[GR93] Jim Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo CA, 1993.

[Gra79] Jim Gray. Notes on Database Operating Systems. *Operating Systems: An Advanced Course*. Springer-Verlag Lecture Notes in Computer Science 60, 1978. 393—481.

[Gra90] Jim Gray. A Census of Tandem System Availability Between 1985 and 1990. Technical Report 90.1, Tandem Computer Corporation, Sept. 1990.

[Gra91] Jim Gray. High Availability Computer Systems. *IEEE Computer*, Sept. 1991.

[GSTC90] Ajei Gopal, Ray Strong. Sam Toueg and Flaviu Cristian. Early-Delivery Atomic Broadcast. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing,* ACM, New York, 297—309.

[GT92] Richard Golding and Kim Taylor. Group Membership in the Epidemic Style. Technical report UCSC-CRL-92-13, University of California at Santa Cruz, May 1992.

[GS96] Rashid Guerraoui, Andre Schiper. Gamma-Accurate Failure Detectors. Technical Report, EPFL, Dept d'Informatique, 1996.

[Gue95] Rashid Guerraoui. Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. In *International Workshop on Distributed Algorithms (WDAG)* (Sept. 1995), 87—100.

[Hag87] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX; Nov. 1987). ACM. 155—171.

[Ham84] K.G. Hamilton. *A Remote Procedure Call System*. PhD Thesis, Computing Laboratory, University of Cambridge, Cambridge England. December 1984. Available as Technical Report 84-70.

[HB93] Mark Hayden and Kenneth P. Birman. Achieving Critical Reliability with Unreliable Components and Unreliable Glue. Cornell University Dept. of Computer Science, TR95-1493, March 1995. (This paper was subsequently substantially revised; a new version will be released in 1996).

[HBJM90] Andrew Hisgen, *et. al*. Granularity and Semantic Level of Replication in the Echo File System. In *Proceedings of the Workshop on Management of Replicated Data* (Houston, TX; Nov. 1990). IEEE CS Press. 5—10.

[Her84] Maurice Herlihy. *Replication Methods for Abstract Data Types*. Ph.D. Thesis, MIT. May 1984. Available as Technical Report LCS-84-319.

[HGDG94] J. Heinlein, K. Garachorloo, S. Dresser and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *6th International Conference om Architectural Support for Programming Langues and Operating Systems* (Oct. 1994), 38—50.

[HHS94] Rainer Handel, Manfred Huber and Stefan Schroder. *ATM Networks: Concepts, Protocols, Applications*. Addison-Wesley, 1994.

[Hil92] Dan Hildebrand. An Architectural Overview of QNX. In *1st USENIX Workshop on Microkernels and Other Kernel Architectures* (Seattle, WA; April 1992). 113—126.

[HKMN87] J. Howard *et al*. Scale and Performance in a Distributed File System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX; Nov. 1987). ACM. Also appearing in the special issue of *ACM Transactions on Computing Systems,* 5:1 (Feb. 1988).

[HL94] Robert Harper and Peter Lee. The Fox Project in 1994. Department of Computer Science Technical Report CS-94-01, Carnegie Mellon University, 1994.

[HO93] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Dec. 1993). ACM. 29—43.

[HP94] J. Heidemann and G. Popek. File System Development With Stackable Layers. *Communications of the ACM* 12:1 (Feb. 1994), 58—89.

[HP95] J. Heidemann and G. Popek. Performance of Cache Coherence in Stackable Filing. In *Proceedings of th 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO; Dec. 1995), 127—142.

[HT87] Maurice P. Herlihy and J. Doug Tygar. How to Make Replicated Data Secure. In *Advances in Cryptography, Proceedings of the 1987 CRYPTO,* Springer-Verlag Lecture Notes in Computer Science, 293. 379—391.

[Hun95] Guerney D. Hunt. Multicast Flow Control on Local Area Networks. PhD thesis, Dept. of Computer Science, Cornell University, Feb. 1995. Also available as TR-95-1479.

[HW90] Maurice Herlihy and Jeanette Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12:3 (July 1990), 463-492.

[IETF95] *Secure Sockets Layer Version 3.0* Internet Engineering Task Force, 1995.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM '88* (Palo Alto, CA; 1988).

[Jac90] Van Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 114, Network Working Group, February 1990.

[Jal94] Pankaj Jalote, *Fault-Tolerance in Distributed Systems*. Prentice Hall, 1994.

[JB86] Thomas A. Joseph and Kenneth P. Birman. Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. *ACM Transactions on Computer Systems* 4:1 (Feb. 1986), 54—70.

[JH93] Alan Jones and Andrew Hopper. Handling Audio and Video Streams in a Distributed Environment. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Dec. 1993). ACM. 231—243.

[JH94] Dag Johansen and Gunnar Hartvigsen. Architecture issues in the StormCast System. Springer Verlag Lecture Notes in Computer Science, LNCS 938, 1-16.

[JKW95] Kirk Johnson, M. Frans Kaashoek and Deborah Wallach. CRL: High-Performance All Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resport, CO; Dec. 1995), 213—228.

[Joh94] Dag Johansen. Stormcast: Yet Another Exercise in Distributed Computing. In *Distributed Open Systems in Perspective*. Johansen and Brazier, *eds*. IEEE. 1994.

[Jon93] Michael B. Jones. Interposition Agents: Transparent Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Dec. 1993). ACM. 80—93.

[Jos86] Thomas A. Joseph. *Low Cost Management of Replicated Data*. Ph.D. thesis, Cornell University, 1986. Available as a Cornell University Dept. of Computer Science Technical Report.

[JvRS95a] Dag Johansen, Robbert van Renesse and Fred Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*. (Orcas Island; May 1995). IEEE. 42-45.

[JvRS95b] Dag Johansen, Robbert van Renesse and Fred Schneider. An Introduction to the TACOMA Distributed System (Version 1.0). Computer Science Technical Report 95-23, June 1995, University of Tromso.

[JvRS96] Dag Johansen, Robbert van Renesse and Fred Schneider. Supporting Broad Internet Access to TACOMA. Technical report. Feb. 1996.

[JZ87] David B. Johnson and Willy Zwaenepoel. Sender-Based Message Logging. In *Proceedings of the 17th Annual International Symposium on Fault-Tolerant Computing,* (June 1987). IEEE. 14—19.

[Kaa92] Frans Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, 1992.

[Kal95] Michael Kalantar. *Issues in Ordered Multicast Performance: A Simulation Study*. Cornell University Department of Computer Science Ph.D. thesis. Available as TR-95-1531, Aug. 1995.

[Kay94] Jonathan S. Kay. *PathIDs: A Mechanism for Reducing Network Software Latency*. PhD thesis, University of California, San Diego. May 1994.

[KBMS95] Yousef A. Khalidi, *et. al*. Solaris MC: A Multi-Computer OS. Sun Microsystems Laboratories, Technical Report 95-48, November 1995.

[KC94] Vijay Karamcheti and Andrew A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages and Operating Systems;* (San Jose, CA; Oct. 1994). ACM.

[KCZ92] P. Keleher, A.L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture,* (May 1992), 13—21.

[KD95] Idit Keidar and Danny Dolev. Increasing the Resilience of Atomic Commit at No Additional Cost. In *Proceedings of the 1995 ACM Symposium on Principles of Database Systems* (May 1995), 245—254.

[KLS85] Nancy Kronenberg, H. Levy, W. Strecker. VAXClusters: A Closely-Coupled Distributed System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA; Dec. 1985). ACM. Appears in *ACM Transactions on Computer Systems,* 4:2 (May. 1986), 130—146.

[KO87] Hermann Kopetz and Wilhelm Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers* C36:8 (Aug. 1987), 933—940.

[Kop92] H. Kopetz. Sparce Time Versus Dense Time in Distributed Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems.* (Yokohama, Japan; June 1992). IEEE.

[KP93] Jon Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads. In *Proceedings of SIGCOMM-93* (Aug. 1993), 259—269.

[KRU91] Clifford Krumvieda. DML: Packaging High Level Distributed Abstractions in SML. In *Proceedings of the 3rd International Workshop on Standard ML.* (Pittsburgh, PA. Sept. 1991). IEEE.

[Kru92] Clifford D. Krumvieda. Expressing Fault-Tolerant and Consistency Preserving Programs in Distributed ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications.* June 1992. 157—162.

[KS91] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Asilomar, CA; Oct. 1991). ACM. 213—225. Also *ACM Transactions on Computing Systems* 10:1 (Feb. 1992), 3—25.

[KT87] Richard Koo and Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering,* SE-13:1 (Jan. 1990), 23-31.

[KT91] M. Frans Kaashoek and Andrew S. Tannenbaum. Group Communication in the Amoeba Distributed Operating System. In *Proceedings of the 11th Internation Conference on Distributed Computing Systems.* IEEE. 222—230.

[KTFH89] M. Frans Kaashoek *et. al.* An Efficient Reliable Broadcast Protocol. *Operating Systems Review* 23:4 (july 1978). 5—19.

[KV93] Hermann Kopetz and Paulo Verissimo. Real-Time Dependability Concepts. In S.J. Mullender, editor, *Distributed Systems (2nd Edition),* ACM-Press (Addison-Wesley), 1993. 411—446.

[LABW92] Butler Lampson, Martin Abadi, Michael Burrows and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems,* 10:4 (November 1992), 265—434.

[Lam78a] Leslie Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computing Networks* 2, 95—114.

[Lam78b] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM* 21:7 (July 1978), 558—565.

[Lam83] Butler Lampson. Hints for Computer System Design. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, NH; Oct. 1993), 33—48.

[Lam84] Leslie Lamport. Using Time Instead of Timeout For Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems* 6:2 (April 1984), 254—280.

[Lam86] Butler Lampson. Designing a Global Name Service. *Keynote presentation at the 1985 ACM PODC, published in Proceedings of the 6th ACM Symposium on Principles of Distributed Computing* (Calgary, CA; 1986), 1—10.

[LCJS87] Barbara Liskov, D. Curtis, P. Johnson and Richard Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX; Nov. 1987). ACM. 111—122.

[Ler93] Xavier Leroy. *The Caml Light System Release 0.7*. INRIA, France. July 1993.

[LH89] Kai Li and Paul Hudak. Memory Coherence in a Shared Virtual Memory System. *ACM Transactions on Computer Systems* 7:4 (Nov. 1989), 321—359.

[LH91] C. S. Laih and L. Harn. Generalized Threshold Cryptosystems. In *Proceedings of ASIACRYPT '91*. 1991.

[LM85] Leslie Lamport and P.M. Melliar-Smith. Synchronizing Clocks in the Presense of Faults. *Journal of the ACM* 32:1 (Jan. 1985).

[LLGW92] D. Lenoski *et. al.* The Stanford DASH Multiprocessor. *Computer* 25:3 (March 1992), 63—79.

[LLSG90] Rivka Ladin, Barbara Liskov, Liuba Shrira and S. Ghemawat. Lazy Replication: Exploiting the Semantics of Distributed Services. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing,* (Quebec, CA; Aug. 1990). ACM. 43—58.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira and S. Ghemawat. Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems* 10:4 (Nov. 1992), 360—391.

[LGGJ91] Barbara Liskov *et. al.* Replication in the Harp File System. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Asilomar, CA; Oct. 1991). ACM. 226—238.

[Lis93] Barbara Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. *Distributed Computing* 6:4 (Nov. 1993), 211-219.

[LL86] Barbara Liskov and Rivka Ladin. Highly Available Distributed Services and Fault-Tolerant Garbage Collection. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing,* (Calgary, CA. Aug. 1986), ACM, 29—39.

[LM85] Leslie Lamport and Peter Melliar-Smith. Synchronizing Clocks in the Presense of Faults. *Journal of the ACM*. 32:1, Jan. 1985. 52-78.

[LMKK89] Samuel J. Leffler, Marshall K. McKusick, *et. al. 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Reading MA, 1989.

[LS83] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguist Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*. 5:3 (July 1983), 381—404.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morton-Kaufman Publishing Company, 1996.

[Lyu95] Michael R. Lyu, *ed. Software Fault Tolerance*. John Wiley and Sons, 1995.

[Maf95] Silvano Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. *In Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*. USENIX. June 1995, Monterey CA.

[Mal94] Dalia Malkhi. *Multicast Communication for High Availability*. PhD thesis, The Hebrew University of Jerusalem, 1994.

[Mal96] C. Malloth. Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks. PhD Thesis. Swiss Federal Institute of Technology, Lausanne (EPFL), 1996.

[MAMA94] Louise E. Moser, Yair Amir, P.M. Melliar-Smith, and D.A. Agarwal. Extended Virtual Synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems* (June 1994). IEEE. 56—65. Additional details are included in the Technical Report version: TR-93-22, U.C. Santa Barbara, Dept. of ECE, Dec. 1993.

[Mar84] Keith Marzullo. *Maintaining the Time in a Distributed System*. PhD thesis, Stanford University, Dept. of Electrical Engineering. June 1984.

[Mar90] Keith Marzullo. Tolerating Failures of Continuous Valued Sensors. *ACM Transactions on Computer Systems* 8:4 (Nov. 1990), 284—304.

[Mat89] Freidemann Mattern. Time and Global States in Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. North-Holland, 1989.

[Mat91] David C. Matthews. A Distributed and Concurrent Implementation of Standard ML. Technical Report ECS-LFCS-91-174, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1991.

[Mar90] Keith Marzullo. Tolerating Failures of Continuous-Valued Sensors. *ACM Transactions on Computer Systems,* 8:4 (Nov. 1990), 284—304.

[MB90] Messac Makpangou and Kenneth P. Birman. Designing Application Software in Wide Area Network Settings. Cornell University Dept. of Computer Science Technical Report 90-1165. 1990.

[MBRS94] Dalia Malki, Kenneth P. Birman, Aleta Ricciardi and Andre Schiper. Uniform Actions in Asynchronous Distributed Systems. Cornell University Department of Computer Science Technical Report TR 94-1447, Sept. 1994.

[MCWB91] Keith Marzullo, Robert Cooper, Mark Wood and Kenneth P. Birman. Tools for Distributed Application Management. *IEEE Computer*. August 1991.

[Merxx] *Reference needed: Merritt's easy proofs of byzantine bounds*

[MES93] R.A. Macedo, P. Ezhilchlvan and Santash Shrivastava. Newtop: A Total Order Multicast Protocol Using Causal Blocks. BROADCAST Project Technical Reports, Volume I, Oct. 1993. Available from Dept. of Computer Science, University of Newcastle upon Tyne, UK.

[MES95] L.B. Mummert, M.R. Ebling and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th Symposium on Operating Systems Principles;* (Copper Mountain Resort, CO; Dec. 1995). ACM. (143—155). Also appearing in the special issue of *ACM Transactions on Computing Systems,* 13:1 (Feb. 1996).

[MM89] P.M. Melliar-Smith, L.E. Moser. Fault-Tolerant Distributed Systems Based on Broadcast Communication. In *Proceedings of the 9th International Conference on Distributed Computing Systems* (June 1989), 129—133.

[MM93] P. M. Melliar-Smith, L.E. Moser. Trans: A Reliable Broadcast Protocol. *IEEE Transactions on Communications* 140:6 (Dec. 1993), 481—493.

[MMA90] P. M. Melliar-Smith, L.E. Moser and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems* 1:1 (Jan. 1990), 17—25.

[MMA94] L. E. Moser, P.M. Melliar-Smith and V. Agarwala. Processor Membership in Asynchronous Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems* 5:5 (May 1994), 459—473.

[MMABL96] L.E. Moser, P.M. Melliar-Smith, D. A. Argarwal, R.K. Budhia, C.A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM* 39:4 (April 1996), 54-63.

[Mon94] Montgomery, T., *Design, Implementation, and Verification of the Reliable Multicast Protocol*, M.S. Thesis, Department of Electrical and Computer Engineering, West Virginia University, Morgantown, WV, December 1994.

[Mos82] J. E. Moss. Nested Transactions and Reliable Distributed Computing. In *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems*. 1982. 33-39.

[MPS91] Shivakan Mishra, Larry L. Peterson and Richard D. Schlichting. A Membership Protocol Based on Partial Order. In *Proceedings of the IEEE International Working Conference on Dependable Computing for Critical Applications* (Feb 1991). 137—145.

[MPS93] Shivakan Mishra, Larry L. Peterson and Richard D. Schlichting. Experience with Modularity in Consul. *Software—Practice and Experience,* 23:10 (Oct 1993), 1050—1075.

[MRA87] Jeff Mogul, Rick Rashid and M. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX; Nov. 1987). ACM. 39—51.

[MSMA90] P. M. Melliar-Smith, L. E. Moser and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems* 1:1 (Jan 1990).

[MSMA91] P. M. Melliar-Smith, L. E. Moser and V. Agrawala. Membership Algorithms for Asynchronous Distributed Systems. In *Proceedings of the IEEE 11th ICDCS* (May 1991). 480—488.

[MRTR90] Sape J. Mullender, *et. al*. Amoeba — A Distributed Operating System for the 1990's. *IEEE Computer* 23:5 (May 1990), 44—53.

[MSV91] S. Meldal, S. Sankar and J. Vera. Exploiting Locality in Maintaining Potential Causality. In *Proceedings of the 10th Symposium on Principles of Distributed Computing*. 1991. 231—239.

[MTH90] Robin Milner, Mads Tofte and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[MW91] Keith Marzullo and Mark Wood. Tools for Constructing Distributed Reactive Systems. Technical Report TR91-1193, Cornell University Dept. of Computer Science, Feb. 1991.

[MW94] Montgomery, T. and B. Whetten, The Reliable Multicast Protocol Application Programming Interface, NASA/WVU Software Research Laboratory Technical Report, NASA-IVV-94-007, August 1994.

[MWBC91] Keith Marzullo, Mark Wood, Kenneth P. Birman, Robert Cooper. Tools for Monitoring and Controlling Distributed Applications. In *Spring 1991 Conference Proceedings* (May 1991). EurOpen. 185—196. Revised and extended as *IEEE Computer* 24:8 (Aug. 1991), 42—51.

[Nei96] Gil Neiger. A New Look at Membership Services. *To appear, 15th ACM Symposium on Principles of Distributed Computing*, 1996.

[NS78] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM,* 21:12 (Dec. 1988), 993—999.

[NWO87] M. Nelson, Brent Welsh and John Ousterhout. Caching in the Sprite Network File System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX; Nov. 1987). ACM. Also appearing in the special issue of *ACM Transactions on Computing Systems,* 6:1 (Feb. 1988).

[OCDN88] John Ousterhout *et. al*. The Sprite Network Operating System. *Computer* 21:2 (Feb. 1988), 23—36.

[ODHK85] John Ousterhout *et. al*. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA; Dec. 1985). ACM. 15—24.

[O+I95] An Introduction to Orbix+Isis. Iona Ltd and Isis Distributed Systems Inc. 1995. (info@iona.ie)

[O+T95] Information about Object Transaction Services for Orbix. Iona Ltd, 1995. (info@iona.ie)

[OMG91] Common Object Request Broker: Architecture and Specification. Published by the Object Management Group and X/Open. Reference OMG 91.12.1 (1991).

[OPSS93] Brian Oki, Manfred Pfluegl, Alex Siegel and Dale Skeen. The Information Bus — An Architecture for Extensible Distributed Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Dec. 1993). ACM. 58—68.

[Ous90] John Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *USENIX Summer Conference Proceedings* (Anaheim, CA; 1990). 247—256.

[Ous94] John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading MA. 1994.

[PA95] Dhiraj Pradhan and Dimiter Avresky, *ed. Fault-Tolerant Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[PBS89] Larry Peterson, N.C. Buchholz and Richard D. Schlicting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computing Systems,* 7:3 (August 1989), 217—246.

[Pet87] Larry Peterson. Preserving Context Information in an IPC Abstraction. In *6th Symposium on Reliability in Distributed Software and Database Systems* (March 1987). IEEE. 22—31.

[Pet95] Ivars Peterson. *Fatal Defect: Chasing Killer Computer Bugs*. Time Books / Random House, 1995.

[PGK88] David Patterson, Garth Gibson and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD).* (Chicago, IL; June 1988), 109—116.

[PHMA89] Larry Peterson, Norm Hutchinson, Sean O'Malley, and Mark Abbott. RPC in the *x*-Kernel: Evaluating New Design Techniques. In *Proceedings of the 12th Symposium on Operating Systems Principles,* ACM, Nov. 1989 (Litchfield Park, AZ), 91—101.

[Pfi95] Gregory F. Pfister. *In Search of Clusters*. Prentice-Hall, 1995.

[Pit87] Boris Pittel. On Spreading of a Rumor. *SIAM Journal of Applied Mathematics*. 47:1 (1987), 213—223.

[Pow91] David Powell, *ed. Delta-4: A Generic Architectue for Dependable Distributed Computing (vol. I).* Springer-Verlag ESPRIT Research Reports, 1991. Project 818/2252.

[Pow94] David Powell. Lessons Learned from Delta-4. *IEEE Micro* 14:4 (Feb. 1994), 36-47.

[Pow96] David Powell. Introduction to Special Section on Group Communication. *Communications of the ACM* 39:4 (April 1996), 50-53.

[PP83] Michael L. Powell and David L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, NH; Oct. 1993), 100—109.

[PP93] Craig Partridge and Steve Pink. A Faster UDP. *IEEE/ACM Transactions on Networking* 1:4 (Aug. 1993), 429—440.

[Pra96] Dhiraj Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, 1996.

[Pu93] Calton Pu. Relaxing the Limitations of Serializable Transactions in Distributed Systems. *Operating Systems Review* 27:2 (*special issue on the Workshop on Operating Systems Principles at Le. Mont St. Michel),* 66-71. April 1993.

[RAAB88a] Marc Rozier, *et. al*. Chorus Distributed Operating System. *Computing Systems Journal* 1:4 (Dec. 1988), 305—370.

[RAAH88b] Marc Rozier *et al.* The Chorus Distributed System. *Computer Systems,* Fall 1988. 299—328.

[Rab83] Michael Rabin. Randomized Byzantine Generals. *24th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1983. 403—409.

[Ras86] Rick F. Rashid. Threads of a New System. *UNIX Review* 4 (August 1986), 37—49.

[RB91] Aleta Ricciardi and Kenneth P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing,* (Quebec, CA; Aug. 1991). ACM. 341—351.

[RB94] Michael K. Reiter and Kenneth P. Birman. How to Securely Replicate Services. *ACM Transactions on Programming Languages and Systems,* 16:3 (May 1994), 986—1009.

[RBCG92] Robbert van Renesse, Kenneth P. Birman, Robert Cooper, Brad Glade and Patrick Stephenson. Reliable Multicast Between Microkernels. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA; April 1992). USENIX.

[RBFH95] Robbert Van Renesse, Kenneth P. Birman, Roy. Friedman, Mark. Hayden, David Karr. A Framework for Protocol Composition in Horus. In *Proceedings of the 14th Symposium on the Principles of Distributed Computing,* (Ottawa, Ontario; Aug. 1995). ACM. 80—89.

[RBG92] Michael Reiter, Kenneth P. Birman, and Li Gong. Integrating Security in a Group-Oriented Distributed System. In *Proceedings of the IEEE Symposium on Research in Security and Privacy,* IEEE, May 1992, Oakland CA. 18—32.

[RBM96] Robbert van Renesse, Kenneth P. Birman, Silvano Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM* 39:4 (April 1996), 76-83.

[RBR95] Michael K. Reiter, Kenneth P. Birman and Robbert van Renesse. A Security Architecture for Fault-Tolerant Systems. *ACM Transactions on Computing Systems,* May 1995.

[Rei93] Michael K. Reiter. *A Security Architecture for Fault-Tolerant Systems*. PhD thesis, Cornell University, August 1993. Available as a Department of Computer Science Technical Report.

[Rei94a] Michael K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. Nov. 1994, Oakland CA. 68—80.

[Rei94b] Michael K. Reiter. A Secure Group Membership Protocol. In *Proceedings of the 1994 Symposium on Research in Security and Privacy*. IEEE.

[Rei96] Michael K. Reiter. Distributing Trust with the Rampart Toolkit. *Communications of the ACM* 39:4 (April 1996), 71-75.

[Ren93] Robbert van Renesse. Causal Controversy at Le Mont St.-Michel. *Operating Systems Review,* 27:2 (April 1993), 44—53.

[Ren94] Robbert van Renesse. Why Bother With CATOCS? *Operating Systems Review 28:1* (Jan. 1994), 22-27.

[RHRS94] Peter Reiher, *et. al*. Resolving File Conflicts in the Ficus File System. In *Proceedings of the Summer USENIX Conference* (June 1994), 183—195.

[Ric92] Aleta M. Ricciardi. The Group Membership Problem in Asynchronous Systems. PhD Thesis, Cornell University, 1992.

[Ric93] Aleta M. Ricciardi. *The Asynchronous Membership Problem.* PhD thesis, Cornell University, January 1993. Available as a Department of Computer Science Technical Report.

[Ric96] Aleta Ricciardi. The Impossibility of (Repeated) Reliable Broadcast. Technical report TR-PDS-1996-003, April 1996. Department of Electrical and Computer Engineering, University of Texas at Austin.

[Rie94] Riecken, D. Intelligent Agents. *Communications of the ACM, 37:7* (July 1994), 19-21.

[Rit84] Dennis M. Ritchie. A stream input-output system. *Bell Laboratories Technical Journal, AT&T.* 63:8 (1984), 1897—1910.

[RK79] D. P. Reed and R. K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM,* 22:2 (Feb 1979), 115—123.

[RO91] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Asilomar, CA; Oct. 1991). ACM. 1—15. Also *ACM Transactions on Computing Systems* 10:1 (Feb. 1992), 26—52.

[RS92] Lawrence A. Rowe and Brian C. Smith. A Continuous Media Player. In *Proceedings of the Third International Workshop on Network and Operating Systems Support for Digital Audio and Video,* Nov 1992, San Diego, CA.

[RSA78] Ron L. Rivest, Adi Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM* 22:4 (December 1978), 120—126.

[RST88] Robbert van Renesse, Hans van Staveren, and Andrew Tanenbaum. Performance of the World's Fastest Operating System. *Operating Systems Review* 22:4 (Oct. 1988), 25—34.

[RST89] Robbert van Renesse, Hans van Staveren, and Andrew Tanenbaum. The Performance of the Ameoba Distributed Operating System. *Software-Practice and Experience* 19:3 (March 1989), 223—234.

[RV89] Luis Rodrigues and Paulo Verissimo. *x*AMP: A MultiPrimitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems* (Houston, TX; October 1989). IEEE.

[RV91] P. Venkat Rangan and Harrick M. Vin. Designing File Systems for Digital Video and Audio. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Asilomar, CA; Oct. 1991). ACM. 81—94.

[RV95] Luis Rodrigues and Paulo Verissimo. Causal Separators for Large-Scale Multicast Communication. In *Proceedings 15th International Conference on Distributed Computing Systems* (May 1995), 83—91.

[RVR93] Luis Rodrigues, Paulo Verissimo and J. Rufino. A Low-Level Processor Group Membership Protocol for LANs. In *Proceedings of the 13th International Conference on Distributed Computing Systems* (May 1993), 541—550.

[Sat89] M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems* 7:3 (Aug. 1989), 247—280.

[SB89] Michael Shroeder and Michael Burrows. Performance of Firefly RPC. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Litchfield Springs, AX; Dec. 1989). 83—90. Also *ACM Transactions on Computing Systems* 8:1 (Feb. 1990), 1—17.

[SBM89] Alex Siegel, Kenneth P. Birman and Keith Marzullo. Deceit: A Flexible Distributed File System. Technical Report 89-1042, Department of Computer Science, Cornell University. 1989.

[Sch82] Fred B. Schneider. Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems*. 4:2 (April 1982), 179—195.

[Sch84] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems,* 2:2 (May 1984), 145—154.

[Sch88a] Frank Schmuck. *The Use of Efficient Broadcast Primitives in Asynchronous Distributed Systems*. PhD thesis, Cornell University, August 1988. Available as a Department of Computer Science Technical Report.

[Sch88b] Fred. B. Schneider . The State Machine Approach: A Tutorial. In *Proceedings of the Workshop on Fault-Tolerant Distributed Computing*. Springer-Verlag Lecture Notes on Computer Science, 1988.

[Sch90] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach. *ACM Computing Surveys* 22:4 (Dec. 1990), 299—319.

[Sch97] Fred B. Schneider. *On Concurrent Programming*. To appear: Springer-Verlag, New York, 1997

[Sch94] Jeffrey I. Schiller. Secure Distributed Computing. *Scientific American* (Nov. 1994), 72—76.

[SDW92] W. T. Strayer, B.J. Dempsey and A.C. Weaver. *XTP: The Xpress Transfer Protocol*. Addison Wesley, 1992.

[Sel93] Margo Seltzer. Transaction Support in a Log-Structured File System. *Proceedings 9th International Conference on Data Engineering*. 1993.

[SES89] Andre Schiper, J. Eggli and Alain Sandoz. A New Algorithm to Implement Causal Ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*. Springer-Verlag Lecture Notes in Computer Science 392, 1989. 219—232.

[SGS84] Fred B. Schneider, David Gries and Richard D. Schlicting. Fault-Tolerant Broadcasts. *Science of Computer Programming*. 3:2 (March 1984), 1—15.

[SHNS85] M. Satyanarayanan *et al*. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA; Dec. 1985). ACM. 35—50.

[Sie92] Alex Siegal. *Performance in Flexible Distributed File Systems*. PhD thesis, Cornell University, February 1992. Available as C.S. Department Technical Report TR-92-1266.

[Ske82a] Dale Skeen. A Quorum-Based Commit Protocol. In *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks* 6 (Feb. 1982), 69—80.

[Ske82b] Dale Skeen. *Crash Recovery in a Distributed Database System*. PhD thesis, University of California at Berkeley, Department of EECS. June 1982.

[Ske85] Dale Skeen. Determining the Last Process to Fail. *ACM Transactions on Computer Systems*. 3:1 (Feb. 1985), 15—30.

[SL87] S.K. Sarin and Nancy A. Lynch. Discarding Obsolete Information in a Replicated Database System. *IEEE Transactions on Software Engineering* 13:1 (1987), 39—47.

[SM89] V. Srinivasan and Jeff Mogul. Spritely NFS: Experiments with Cache Consistency Protocols. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Litchfield Springs, AX; Dec. 1989). 45—57.

[SM91] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations. Technical Report 215-91, Department of Computer Science, University of Kaiserslautern, Germany (1991).

[SM94] Laura Sabel and Keith Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings. 13th Symposium on Reliable Distributed Systems* (Dana Point, CA; Oct. 1994). IEEE. 138—147.

[SNS88] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proceedings of the 1998 USENIX Winter Conference*. USENIX. Feb. 1988, Dallas TX. 191—202.

[Spe85] Alfred Spector. Distributed Transactions for Reliable Systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA; Dec. 1985), 12—146.

[SRC84] J.H. Saltzer, D.P. Reed and D.D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems* 39:4 (April 1990).

[SR96] Andre Schiper and Michel Raynal. From Group Communication to Transactions in Distributed Systems. *Communications of the ACM* 39:4 (April 1996), 84-87.

[SS83] Richard D. Schlicting and Fred. B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems* 1:3 (August 1983), 222—238.

[SS93] Andre Schiper and Alain Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems* (May 1993), IEEE. 561—568.

[SS96] Marjana Spasojevic and M. Satyanarayanan. An Emperical Study of a Wide-Area Distributed File System. *ACM Transactions on Computer Systems* 14:2 (May 1996).

[SSC94] W. T. Strayer, G. Simon and R.E. Cline, Jr. An Object Oriented Implementation of the Xpress Transfer Protocol. XTP Forum Research Affiliate Annual Report, 1994. 53—66.

[ST87] T. K. Srikanth and Sam Toueg. Optimal Clock Synchronization. *Journal of the ACM* 34:3 (July 1987), 626—645.

[STB86] Richard E. Schantz, Robert H. Thomas and Girome Bono. The Architecture of the Chronus Distributed Operating System. *Proceedings of the Sixth International Conference on Distributed Computing Systems* (IEEE, 1986), 250-259/

[Ste91] Patrick Stephenson. *Fast Causal Multicast*. PhD thesis, Cornell University, Feb. 1991. Available as a Dept. of Computer Science Technical Report.

[Sto81] Michael Stonebreaker. Operating Systems Support for Database Management. *Communications of the ACM* 24:7 (July 1981), 412—418.

[SW91] Frank Schmuck and Jim Wyllie. Experience with Transactions in Quicksilver. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Asilomar, CA; Oct. 1991). ACM. 239—252.

[SWL90] Barbara Simons, Jennifer N. Welch, and Nancy Lynch. An Overview of Clock Synchronization. In *Fault-Tolerant Distributed Computing* (Simons and Spector, *eds*), Springer Verlag Lecture Notes in Computer Science 448, 1990, 84—96.

[Tan88] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Second Edition, 1988.

[Ten90] David Tennenhouse. Layered Multiplexing Considered Harmful. In *Protocols for High Speed Networks*. Elsevier Publishers, BV. 1990.

[TH90] Joseph Torrellas and John Hennessey. Estimating the Performance Advantages of Relaxing Consistency in a Shared-Memory Multiprocessor. Technical Report CSL-TN-90-265, Stanford University Computer Systems Laboratory, Feb. 1990.

[Tho79] Robert Thomas. A Majority Concensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems* 4:2 (June 1979), 180—209.

[TL93] C. A. Thekkanth and H.M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems* 11:2 (May 1993), 179—203.

[TNML93] C. Thekkath, T. Nguyen, E. Moy and Ed Lazowska. Implementing Network Protocols at User Level. *IEEE Transactions on Networking* 1:5 (Oct. 1993), 554—564.

[TR88] Andy Tanenbaum and Robbert van Renesse. A Critique of the Remote Procedure Call Paradigm. In *Proceedings of the EUTECO '88 Conference* (Vienna, Austria; April 1988), 775—783.

[TS92] John Turek and Dennis Shasha. The Many Faces of Consensus in Distributed Systems. *IEEE Computer* 25:6 (1992), 8—17.

[TTPD95] Doug B. Terry *et. al*. Managing Update Conflicts in a Weakly Connected Replicated Storage System. In *Proceedings of the 15th Symposium on Operating Systems Principles;* (Copper Mountain Resort, CO; Dec. 1995). ACM. 172—183.

[Ver93] Paulo Verissimo. Real-Time Communication. In S.J. Mullender, editor. *Distributed Systems (2nd Edition)*. ACM-Press (Addison Wesley), 1993. 447—490.

[Ver94] Paulo Verissimo. Ordering and Timeliness Requirements of Dependable Real-Time Programs. *Journal of Real-Time Systems* 7:2 (sept. 1994), 105—128.

[Ver96] Paulo Verissimo. Causal Delivery in Real-Time Systems: A Generic Model. *Real-Time Systems Journal* 10:1 (Jan. 1996).

[VK83] V.L. Voydock and Steve T. Kent. Security Mechanisms in High-Level Network Protocols. *ACM Computing Surveys* 15:2 (June 1983), 135—171.

[Vog96] Werner Vogels. The Private Investigator. Cornell University Department of Computer Science Technical Report, April 1996.

[VR92] Paulo Verissimo, Luis Rodrigues. A Posteriori Agreement for Fault-Tolerant Clock Synchronization on Broadcast Networks. In *Proceedings 22nd International Symposium on Fault-Tolerant Computing* (Boston, MA; July 1992).

[WLAG93] Robert Wahbe, Steven Lucco, Tom Anderson and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Dec. 1993). ACM. 203—216.

[Whe95] Brian Whetten. A Reliable Multicast Protocol. In *Theory and Practice in Distributed Systems,* Birman, Mattern and Schiper, *eds.,* Springer-Verlag Lecture Notes on Computer Science, 938 (July 1995).

[WGSS95] John Wilkes, *et. al.* The HP AutoRAID Hierarchical Storage System. In *Proceedings of the 15th Symposium on Operating Systems Principles;* (Copper Mountain Resort, CO; Dec. 1995). ACM. (96—108). Also appearing in the special issue of *ACM Transactions on Computing Systems,* 13:1 (Feb. 1996).

[WOE92] John Warne, E. Oskiewicz and N. Edwards. A Lightweight Group Execution Protocol. Technical Report RC.439, APM Ltd., Cambridge UK, October 1992.

[Won95] Ted Wong. *Private communication.* May, 1995.

[Woo91] Mark D. Wood. *Fault-Tolerant Management of Distributed Applications Using a Reactive System Architecture.* PhD thesis, Cornell University (Dec. 1991). Available as Department of Computer Science Technical Report TR 91-1252.

[Woo93] Mark D. Wood. Replicated RPC Using Amoeba Closed-Group Communication. In *Proceedings of the 12th International Conference on Distributed Computing Systems* (Pittsburgh, PA; 1993).

[WPEK93] B. Walter, G. Popek, *et. al.* The Locus Distributed Operating System. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, NH; Oct. 1993); 49—70.

[Wu95] Wu, Y., Verification-Based Analysis of RMP, NASA/WVU Software Research Laboratory Technical Report NASA-IVV-95-003, December 1995.

[XTP95] XTP Forum. *Xpress Transfer Protocol Specification.* XTP Rev. 4.0, 95-20, March 1995.

Index