Consistency Maintenance in Large-Scale Systems

Pete Keleher
keleher@cs.umd.edu
Department of Computer Science
University of Maryland

August 6, 1997

1 Problems

The new-found connectivity spawned by the emergence of the WWW is clearly the most challenging, and potentially rewarding, issue facing the systems community today. This explosion of growth provides a number of opportunities, together with a like number of challenges. Both the opportunities and the challenges center around the new-found ability to tie together widely separated pieces of information into at least a semi-coherent whole. The problem of maintaining coherency in large-scale systems without sacrificing performance is the focus of this position statement. We are interested in system sizes from several dozen entities, up to at least several thousand. The application domain that we assume includes (but is not limited to) the ubiquitous world wide web (WWW), distributed databases and client-server applications, distributed collaboration systems, and traditional parallel HPC codes communicating either via message-passing or software distributed shared memory (DSM).

One of the central challenges raised by these new systems is that of efficiently maintaining consistent shared state in wide-area systems. Although the reasons for the high cost of such systems might seem obvious, we review them below.

- Scale -

  The large scale of wide-area systems poses problems both in the amount of meta-information needed to track copies or replicas of shared state, but also in update propagation when shared state is changed. Distributed applications in wide-area networks could be very large, having hundreds or thousands of disparate components. The WWW is a good example. Each page or object published on the WWW and cached elsewhere is an object that is effectively shared across all of the objects that currently cache it. Rather than requiring a WWW server to track (potentially thousands of) cachers for each and every object that it exports, the prevalent practice currently is to require cachers to periodically query the object's server to see if the object is "stale". This technique distributes the state across the cachers, rather than creating a resource consumption bottleneck at the server. Secondly, and more importantly, this approach allows cachers to individual determine their own toleration for shared data. This point will be discussed more below.

- Dynamicism -

  Wide-area networks are inherently dynamic. The set of browsers viewing a given WWW page varies enormously from one minute to the next. Network partitions, congestion, and failures at many different levels of the system can increase the amount of variability still further. Clearly, any system of organizing meta-information must be able to cope with quickly changing information.

- Heterogeneity -

Wide-area systems are inherently heterogeneous, whether at machine and processor level, or just in the capacity of the connections between individual components and the rest of the network. These differences can be categorized either as either performance or platform heterogeneity. The latter is more serious because it usually can not be ignored. For most purposes, however, it suffices to provide a method of transforming a common network format into the local format of each system.

2 Approaches to Solutions

The following sections enumerate potentially promising areas of systems research, all with the overriding goal of allowing consistency to be efficiently maintained in large-scale systems.

2.1 Global Consensus Avoidance

The amount of communication needed to reach global consensus scales at least linearly with the size of the system, and can be much worse. While techniques such as as broadcast busses and secondary synchronization busses can improve performance of tightly-coupled systems, these techniques can not generally be used in larger systems. Hence, global consensus should be avoided when possible. Unfortunately, global consensus is needed for some tasks, and very convenient for others. For example, parallel applications often alternate phases where information is collected from the entire set of shared data, and phases where each entity modifies the portion of data "owned" by that entity. Global "barrier" synchronization is needed in order to delineate the two.

The same domain contains situations where the global consensus is merely convenient, not necessary. Jacobi-type applications often shared data only among neighbors. Even though synchronization to mark the end of a phase is is necessary only between neighbors, global barrier synchronization is often used because of convenience, and because there is a large gap in power between lock synchronization and barrier synchronization.

The most effective way to avoid this type of "synchronization fragmentation" is to create flexible synchronization and communication primitives. Such primitives will allow users to better match synchronization calls with application needs. In turn, better matches should allow performance to scale better.

2.2 Exploiting the Absence of Peer-to-Peer Relationships

Both client- and server-initiated approaches to maintaining consistency with WWW browsers work because modifications to web objects are only made visible at the web server's port. The system effectively behaves as a single-writer system in which the owner of each object is static. Moreover, objects have a property analogous to spatial locality, in that most objects that are common accessed together are hosted by the same server. These characteristics enormously simplify the problem of ensuring that a given replica is current.

More generally, the fact that each web object has a single node can be thought of as application-specific semantic information about sharing characteristics. As such information can be crucial in scaling applications to large systems, we advocate research into finding useful sharing relationships, and methods for deriving such relationships from systems and applications.

2.3 Tolerating and Exploiting Transient Inconsistency

One implication of avoiding global consensus is that temporary inconsistencies may develop between replicas of a single shared object. For example, distributed collaborative environments often allow different "views" of a system to independently specify whether they should be updated optimistically or pessimistically. An optimistic interactive view might display the results of local transactions to a user before they have been committed. The view "rolls back" if and when the view is found to have displayed changes from transactions that eventually aborted.

The rollback works because of two properties of the application. First, the user of the interactive display is presumed to be able to tolerate the inconsistency. There exist situations where this is inappropriate because of irrevocable actions triggered by specific views. For instance, firing a missile is not usually undo-able.

Second, the technology underlying the view is versioning, and retains enough information to roll back the view until it is consistent with respect to the rest of the system. Versioning is easy to implement if the shared data is either small or discrete. However, versioning of structured data can be complicated by links to objects at other sites.

Traditional approaches to these problems rely on global information and consensus, often implemented by a "sweep" though all objects in the system. Current research is focusing on improving interactive response through selectively allowing replicas to diverge. We advocate research into the more general problem of predicting the performance impact of allowing inconsistency to exist within large-scale systems.

2.4 Exploring the Tradeoff Between Laziness and Eagerness

This tradeoff refers to how updates are propagated through distributed systems. Updates can be "eagerly" sent at the first opportunity, or "lazily" delayed. There are many variations of each approach, but eager systems are generally less complex, and therefore more common. Lazy systems are often made more complex by the need for making meta (or consistency) information persistent.

In DSM systems, for example, an eager system is usually one that "performs" writes to shared data at the first opportunity. Single-writer protocols allow only the owner of a page to modify it, and usually require all other copies of the page to be invalidated. In other words, consistency information (notification of the page modification) has to be sent even before the actual modification occurs. This requirement that remote communication occurs before the modification slows the process down considerably, but there are advantages. For one, no state describing the modification needs to be retained once the other copies of the page have been modified.

On the other side of the spectrum are lazy systems (systems that use lazy release consistency). Lazy systems allow each processor to locally decide to modify any valid page. The only requirement imposed by the consistency algorithm is that a notice describing the fact of the modification must be appended to future synchronization messages. This approach has the obvious advantages of allowing page modifications to proceed immediately and of allowing consistency information to be piggy-backed on already existing synchronization mechanisms. Delaying consistency propagation also has other advantages, especially in the presence of false sharing.

The central disadvantage of this approach is that it requires significantly more state to be retained. Since the set of processes that synchronize directly with the modifier does not necessarily include all the locations that cache the modified page, this information must be forwarded by those processes to others. This implies that synchronization messages must carry both direct and indirect information. It also implies that the original modifier does not know when, or even if, the notice eventually reaches all cachers. This last possibility requires the modifier to retain consistency information until it directly contacts all other processes in the system.

Hence, consistency information in lazy systems is often persistent, while consistency information in eager systems can be discarded immediately after use. Another way to look at this is that the amount of state that lazy systems must retain scales with the size of the system, whereas the state of eager systems scales only with the degree of sharing. These two can coincide, but usually will not.

We advocate investigation of protocols that are lazy enough to avoid global operations, but eager enough to limit persistent state.

2.5 Tolerating Latency

Communication cost actually refers to two separate characteristics of the communication medium: bandwidth and latency. While bandwidth appears to be scaling with machine speed, the latency of remote requests often does not, and indeed, can not. The reason is that latency is a function of operating system characteristics, switch delays, and distance. While research has produced ways of coping with operating system overheads, neither switch delay not distance is likely to go away in the near future. As bandwidth increases, therefore, latency is more and more becoming the most important characteristic of any network connection.

This increasing cost of latency needs to be addressed by any wide-area system. There are at least three promising approaches to hiding this latency. The canonical approach is prefetching, which can be accomplished either statically (through explicit fetches inserted by the compiler or the application programmer) or dynamically through runtime observation. The dynamic approach is likely to be more useful in wide-scale systems than with cache-based systems the greater cost of remote requests makes computationally expensive predictive algorithms more cost-effective.

Multi-threading is another approach to hiding latency. The idea is to divide local work into multiple independent threads; switching from one thread to the next as they become blocked. The drawback of threads is that considerable programmer or compiler effort needs to be expended in order to separate out disjoint sets of work.

Finally, several optimistic techniques that have been discussed in the literature in other contexts could be applicable. For instance, the use of optimistic Time Warp-like mechanisms could be predicated on specific responses to remote requests. If and when a "wrong" response arrives, the optimistic computation can be rolled back to the point before the request was sent.

Widespread use of this type of mechanism would require detailed application information in order to make the predictions. However, this information does not necessarily have to come from the programmer. Instead, it could come from the programming system on which the application is based. For instance, consider a non-local lock acquisition in a DSM. The only response to a lock request is that the lock is granted. However, the lock grant implies that any necessary consistency action, such as invalidation of data that is guarded by the lock, has already taken place. One approach to hiding the remote lock latency would be to assume that all necessary invalidations have been applied and to continue executing *before* the lock grant is returned. If the guess is wrong, i.e. more invalidations arrive before the lock grant, the process can be rolled back.