The Architectural Design of Globe: A Wide-Area Distributed System

Maarten van Steen (contact)
Philip Homburg
Andrew S. Tanenbaum
Vrije Universiteit, Department of Mathematics & Computer Science
De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands
tel: +31 (0)20 444 7784, fax: +31 (0)20 444 7653
e-mail: {steen,philip,ast}@cs.vu.nl

Article summary. Developing large-scale wide-area applications requires an infrastructure that is presently lacking. Currently, most Internet applications have to be built on top of raw communication services, such as TCP connections. All additional services, including those for naming, replication, migration, persistence, fault tolerance, and security, have to be implemented for each application anew. Not only is this a waste of effort, it also makes interoperability between different applications difficult or even impossible.

The authors present a novel, object-based framework for developing wide-area distributed applications. The framework is based on the concept of a distributed shared object, which has the characteristic feature that its state can be physically distributed across multiple machines at the same time. All implementation aspects, including communication protocols, replication strategies, and distribution and migration of state, are part of each object and are hidden behind its interface.

The current performance problems of the World-Wide Web are taken as an example to illustrate the benefit of encapsulating state, operations, and implementation strategies on a per-object basis. The authors describe how distributed objects can be used to implement worldwide scalable Web documents.

Keywords: *wide-area systems, distributed systems, distributed objects, Internet, middleware*

The spectacular growth of the Internet has the potential of connecting a billion computers together within the next decade into an integrated distributed system offering numerous applications for science, commerce, education, and entertainment. The hardware and communications infrastructure needed is rapidly being deployed. However, the software infrastructure is still lacking. We propose a novel scalability infrastructure for a massive worldwide distributed system.

At present, we are building applications on top of a limited number of communication services. In the Internet, for example, this means that applications communicate mainly through TCP connections, but otherwise have to implement all additional services themselves, including services for naming, replication, migration, fault tolerance, and security.

As an example, consider the World-Wide Web. The Web implements its own communication protocol, HTTP, on top of TCP. It uses a tailor-made naming system based on URLs. Replication is supported in the form of caches that are part of Web proxies, but cannot be used for other applications as cache coherence protocols rely on attribute fields of Web pages. Hardly any measures have been taken to handle broken links and server crashes. Finally, security has been proposed in the form of an extension to HTTP, but there are also proprietary solutions such as SSL from Netscape. Other Internet applications such as e-mail and USENET News each have their own software models and infrastructure, with no commonality among any of them.

As a consequence, building new wide-area applications is difficult. First, too much effort is repeatedly spent on implementing common or standard services that should already have been there to start with. Second, by using application-specific services, interoperability between different applications can be difficult or even impossible.

Instead, we propose a different approach. Rather than developing applications directly on top of the transport layer, we want to create a software infrastructure that provides us with a set of common distribution services. The main requirement is that this infrastructure, or *middleware*, can scale to support in the order of a billion users all over the world.

Requirements for Scalable Middleware

Our solution lies in the development of a wide-area distributed system called Globe. We aim to meet three major design objectives: (1) provide a uniform model for distributed computing, (2) support a flexible implementation framework, and (3) ensure worldwide scalability.

A Uniform Model for Distributed Computing

A distributed system should provide a consistent and uniform view of how to organize applications built on top of it. DCOM[1] and DCE,[2] for example, support client–server computing using only RPCs. CORBA[3] provides a remote object model for all its applications. Applications built on top of AFS[4] are offered a wide-area file system based on location-transparent naming. The Web, finally, offers a model of worldwide distributed documents tied together through hyperlinks.

A uniform model contributes to a single-system view. In addition, it should integrate common services such as communication, naming, replication, etc. Moreover, these services should be included in such a way that all aspects related to the distribution of data, computations, and coordination are effectively hidden from users. In other words, a model should provide *distribution transparency*. Worldwide systems that integrate common services and support all types of distribution transparency do not exist at present.

More importantly, at present distribution services generally have a single general-purpose policy wired in. For example, all proxy caches in the Web work the same way. The same holds for caching in AFS. In CORBA and DCE, client proxies are always the same: all they do is forward requests and handle replies. There is no straightforward way to build more sophisticated proxies.

We argue that we need mechanisms for implementing *object-specific* policies. Such policies should be entirely encapsulated by an object. In Globe, we tackle these problems by providing a model of **distributed shared objects**. The main distinction with existing models is that (1) our objects can be physically distributed, and (2) each object fully encapsulates its own policy for replication, migration, etc. In other words, in Globe, an object is completely self-contained, so that objects for different applications can have replication and often policies carefully tailored to their needs. Nevertheless, all implementation aspects are hidden behind its interfaces to achieve distribution transparency. The Globe object model is explained in detail below.

A Flexible Implementation Framework

The heterogeneity inherent to a wide-area system should preferably be transparent to applications. However, complete transparency is not always a good idea. For example, for some computations we may want to make use of a parallel computer, so it matters where the computation is done. A wide-area distributed system should thus make specialized facilities available to applications when needed. For similar reasons, aspects of the underlying network should be made visible. For example, when bandwidth is scarce it may be better to move data and computations from server to client, as in the case of Java applets.

What we need is a flexible implementation framework: a set of cooperating mechanisms that make up a reusable design for wide-area distributed applications. [5] It is here that an object-based approach will help. By strictly separating an object's interface from its implementation, we can construct reusable designs by considering only interfaces. A design can be tailored toward a specific application by choosing the appropriate object implementations, and, where necessary, extending the design with other objects. [6] This is the approach followed in Globe.

Worldwide Scalability

The real challenge is that we may eventually have to support one billion users, each having thousands of objects, and requiring services from all over the world. A worldwide scalable distributed system is capable of offering adequate performance in the face of high network latencies, congestion, overloaded servers, limited resource capacity, unreliable communication, etc. To achieve worldwide scalability we at least need to provide extensive support for partitioning and replicating objects. [7]

Adequate support for scaling techniques is precisely what is lacking in current middleware. DCOM, DCE, and CORBA do not provide the tools for replicating objects. In those cases where caching or replication is supported, such as in AFS and the Web, policies are fixed. However, efficient solutions that scale worldwide can be found only by taking application-level consistency into account. Again, this calls for flexibility.

The Globe System

To support the next generation of large-scale wide-area applications, we are currently developing Globe. Globe is a wide-area distributed system that is constructed as a middleware layer on top of the Internet. It is designed to run on top of various UNIX systems and Windows NT. We have recently finished our initial architectural design, which consists of an object model and a collection of basic support services. The object model allows for the construction of worldwide scalable objects that can be shared by a vast number of processes. Support services include, among others, services for naming and locating objects.

The Globe Object Model

In Globe, processes interact and communicate through **distributed shared objects**. Each object offers one or more **interfaces**, each interface consisting of a set of methods. A Globe object is physically distributed, meaning that its state may

be partitioned and replicated across multiple machines at the same time. However, processes are not aware of this: state and operations on that state are completely encapsulated by the object. All implementation aspects, including communication protocols, replication strategies, and distribution and migration of state, are part of the object and are hidden behind its interface.

In order for a thread in a process to invoke an object's method, it must first **bind** to that object by contacting it at one of the object's contact points. A **contact address** describes such a contact point, specifying a network address and a protocol through which the binding can take place. Binding results in an interface belonging to the object being placed in the client's address space, along with an implementation of that interface. Such an implementation is called a local object. This model is illustrated in Figure 1, which shows a Globe object distributed across four address spaces.

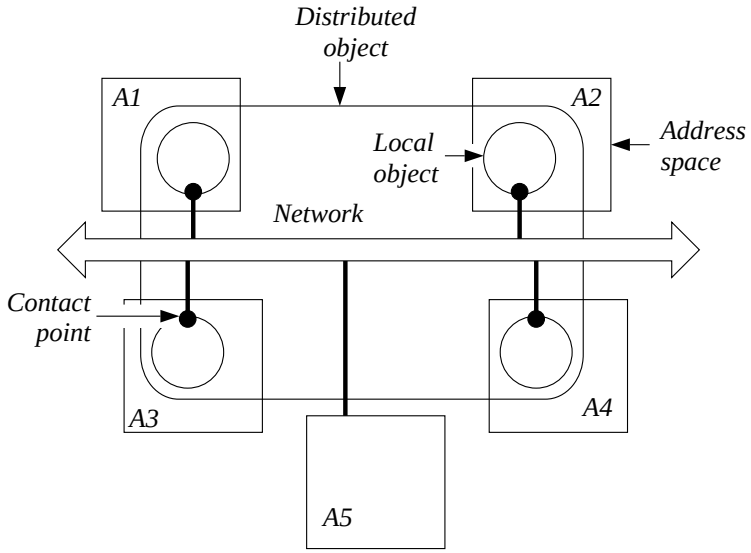

Figure 1: Example of an object distributed across three address spaces.

A distributed object is built from local objects. A **local object** resides in a single address space and communicates with local objects in other address spaces. It forms a particular implementation of an interface of the distributed object. For example, a local object may implement an interface by forwarding all method invocations, as in RPC client stubs. A local object in another address space may implement that same interface through operations on a replica of the object's state.

Our aim is to let application developers concentrate on designing and implementing functionality in terms of objects. Distribution is a different concern, and should be treated separately. For this reason, local objects are constructed in a

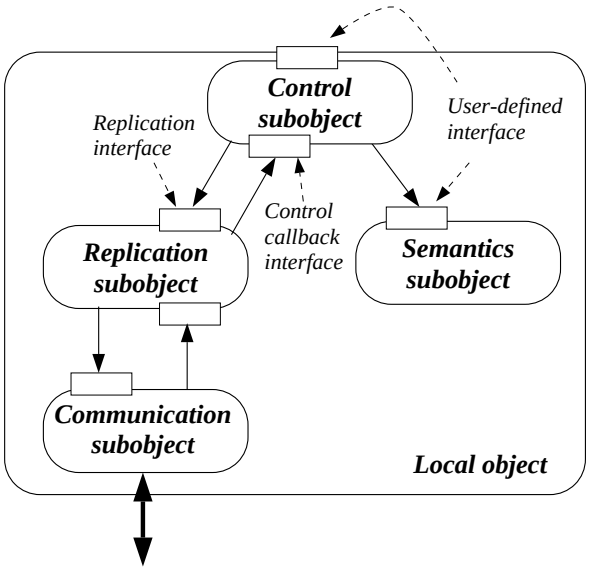

Figure 2: The general implementation of a distributed object.

modular way, to separate issues such as replication and communication from what the object actually does (i.e., its semantics). We distinguish the following four subobjects, as shown in Figure 2:

- A **semantics subobject** containing the methods that implement the functionality of the distributed shared object
- A **communication subobject** for sending and receiving messages from other local objects
- A **replication subobject** containing the implementation of a specific replication policy
- A **control subobject** handling the flow of control within the local object

These four subobjects are designed for building *scalable* distributed shared objects. Of course, we also need support for security and persistence, as well as other services, which, in our approach, are handled by separate subobjects. As scalability is the focus of this paper, we discuss only the four listed subobjects here.

Semantics subobject. The semantics subobject is comparable to objects in middleware such as DCOM and CORBA. It implements (part of) the functionality the

Table 1: Interface of the replication subobject as used by the control subobject

Replication interface	
Method	**Description**
start	Called to synchronize replicas of the semantics subobjects, obtain locks if necessary, etc.
invoked	Called after the control subobject has invoked a specific method at the semantics subobject
send	Provide marshalled arguments of a specific method, and pass invocation to local objects in other address spaces
finish	Called to synchronize the replicas again, release locks, etc.

distributed shared object has, thereby ignoring distribution issues. In Globe, a semantics subobject can be implemented in any language; its methods are made available by means of one or more interfaces. We expect that each subobject implements the *standard object interface*, which has a similar role as the IUnknown interface in COM. Like IUnknown, the standard object interface provides a method getInterface which returns a pointer to a specified interface.

In principle, the semantics subobjects are the only subobjects a developer needs to construct personally. All other parts can either be obtained from libraries, or are generated from interface and object descriptions. The only restriction we currently impose is that a thread of control is not allowed to block inside a semantics subobject. Instead, a method should return indicating a condition did not hold. In that case, the control subobject will block the invoking thread as we explain shortly.

Replication subobject. The global state of the distributed object is made up of the state of its various semantics subobjects. In our approach, replication and caching of the semantics subobjects are important techniques for scalability. However, having several copies leads to a consistency problem: changes to one copy make that copy different from the others. To what extent such inconsistencies can be tolerated, depends on the distributed object and the way it is used. Consequently, we need to support coherence protocols on a per-object basis. The replication subobject acts as a placeholder for different protocols and a variety of protocol implementations.

Our basic assumption is that coherence protocols can be expressed in terms of *when* specific methods of a local copy of a semantics subobject can be invoked. The replication subobject thus decides when local invocations can take place. Omitting specific details, it offers an interface to the control subobject shown in Table 1.

In principle, *all* invocation requests, whether they come from the local client or from the network, are first passed to the replication subobject before the method is invoked at the semantics subobject. When the control subobject receives an invocation request from the local client, it first calls start to allow the replication subobject to synchronize the copies of the semantics subobject. For example, the coherence protocol may require that a token is acquired before any method invocation at the semantics subobject takes place.

The start method returns a set of actions that the control subobject should take. The return value INVOKE tells the control subobject to invoke the method at the semantics subobject. Likewise, SEND instructs the control subobject to pass the marshalled arguments of the invocation to the replication subobject by subsequently calling send. So, for example, with a replication strategy where a method has to be invoked at all replicas, an implementation of start may return {INVOKE,SEND}, telling the control object to (1) do a local invocation, and (2) pass the marshalled invocation request so that it can be sent to the other replicas.

The final step is invoking finish, allowing the replication subobject to synchronize the replicas again (if needed). Again, *when* finish is to be invoked is determined by the replication subobject, for which it returns FINISH after the invocation of start or send. Invoking finish generally returns {RETURN} telling the control subobject that it can pass the return value of the method invocation to the local client.

A distinctive feature of our model is that we allow method invocations at the semantics subobject to block on condition failures. For example, appending data to a bounded buffer may fail when the buffer is full. Concurrent access to the semantics subobject is controlled by the replication subobject. After invoking a method at the semantics subobject, the control object always calls invoked, informing the replication subobject whether or nor a condition failure occurred, and passing control back to the replication subobject. If necessary, the current thread blocks inside the replication subobject. The replication subobject can then allow other invocations to take place, which may possibly change the state of the semantics subobject such that the blocked thread can later on continue successively.

Control subobject. The methods of the semantics subobject are always invoked by the control subobject. This subobject controls two type of invocation requests: those coming from the local client, and those coming in through the network. The control subobject is also responsible for (un)marshalling invocation requests that are passed between itself and the replication subobject. The interface of the control subobject offered to the local client is the same as the (user-defined) interface of the semantics subobject. In addition, it offers the callback interface to the replication subobject shown in Table 2.

Table 2: The callback interface of the control subobject as used by the replication subobject

Control callback interface	
Method	**Description**
handle_request	Called to invoke the specified method at the semantics subobject
getState	Returns the (marshalled) state of the semantics subobject
setState	Replace current state of semantics subobject with state passed as argument

In general, when a local client invokes a method at the control subobject, the latter will eventually invoke that method at its local copy of the semantics subobject after receiving permission from the replication subobject. Remote invocation requests, that is, requests that have been passed by replication subobjects in remote address spaces, are eventually passed to the control subobject through handle_request. The control subobject then simply does the local invocation at the semantics subobject.

Communication subobject. The communication subobject, finally, is responsible for handling communication between parts of the distributed object that reside in different address spaces. It is generally a system-provided local object. Depending on what is needed from the other components, the communication subobject offers reliable or best-effort communication, connection-oriented or connectionless communication, and point-to-point or multicast facilities. Like the replication subobject, it offers a standard interface, but allows many different implementations of that interface. The most important methods are those for sending and receiving messages, as well as methods to support request/reply semantics. Considering its similarity to interfaces for communication libraries, we omit further discussion.

Discussion

We have chosen for this organization as it provides the minimum framework for implementing scalable distributed objects in a flexible way. A key role is reserved for the replication subobject. In our view, the only way to achieve wide-area scalability of distributed objects is to concentrate on the distribution of their state. With the enormous variety of objects, it is clear that a general-purpose, "one-size-fits-all" distribution policy will never suffice, which calls for per-object solutions.

The main role of our communication subobjects is that they provide a uniform interface to underlying networks and operating systems concerning their commu-

nication facilities. By providing a standard interface, we can develop other local objects in a platform-independent way. An important observation is that communication and replication subobjects are unaware of the methods and state of the semantics subobject. This independence allows us to define standard interfaces for all replication subobjects and communication subobjects. Consequently, we can implement different policies, but keep the interfaces the same. This also means that we can now easily adopt a policy by choosing an appropriate implementation from a library of *class objects*, which contain the implementation of subobjects, and dynamically download that implementation into our local object framework.

We have omitted the description of other important subobjects, notably those that handle persistence and security. However, it is obvious that any worldwide distributed system should take both issues into account from the start.

Process–to–Object Binding

To communicate through a distributed object, it is necessary for a process to first **bind** to that object. The result of binding is that the process can directly invoke the object's methods. In other words, a local object implementing an interface of the distributed object is placed in the address space of the requesting process. Binding itself consists roughly of two distinct phases: (1) finding the distributed object, and (2) installing a local object. This is illustrated in Figure 3. Finding a distributed object is separated into a name look-up and a location look-up step; installing the local object requires that we select a suitable contact address, as well as an implementation for that interface.

Finding a Distributed Object

To find a distributed object, a process must pass a name of the object to a naming service. The naming service returns an **object handle**, which is a location-independent and universally unique object identifier, such as a 128-bit number, which is used to locate objects. It can be passed freely between processes as an object reference. It never changes over time and is guaranteed to refer to the same object, even years later (if the object still exists). The object handle is given to a location service, which returns one or more contact addresses. Globe thus uses a two-level naming hierarchy.

This organization allows us to separate issues related to naming objects from those related to contacting objects. In particular, it is now easy to support multiple and independent names for an object. Because an object handle does not change once it has been assigned to an object, a user can easily bind a private, or locally shared name to an object without ever having to worry that the name–to–object

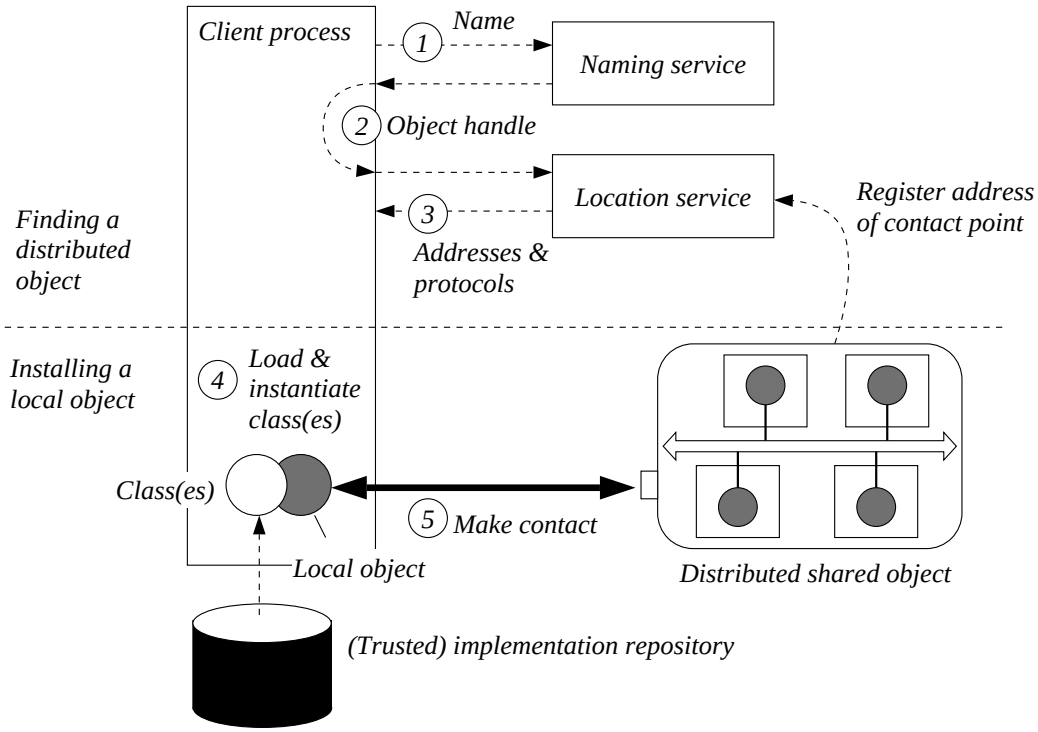

Figure 3: Binding a process to a distributed shared object.

binding changes without notice. On the other hand, an object can update its contact addresses at a location service without having to consider under which name it can be reached by its clients.

We can now remove all location information from names, thus making it easier to realize distribution transparency. However, we do require a scalable location service that can handle frequent updates of contact addresses in an efficient manner. We have designed such a service[8] and are currently implementing a prototype version that is being tested on the Internet.

Installing a Local Object

Once a process knows where it can contact the distributed object, it needs to select a suitable address from the ones returned by the location service. A contact address may be selected for its locality, but there may be other criteria as well for preferring one address over another. For example, some addresses may belong to subnets that are difficult to reach, or to which only low-bandwidth connections can be established. Other quality of service aspects may need to be considered as well. Note that an address selection service is a *local* service that builds its own administration concerning the quality of contact addresses.

A contact address describes *where* and *how* the requested object can be reached. The latter is expressed as a *protocol identifier*. It specifies a complete stack of protocols that should be implemented at the client's side in order to communicate with the distributed object.

Of course, implementation selection may fail if a (trusted) implementation cannot be found. In that case, the binding process returns to the address selection step, where the next best address is considered.

Current Status

We have built an initial prototype implementation of our system, concentrating on the support for distributed shared objects. Our initial prototype has been implemented in ANSI C. We are currently developing a Java-based implementation.

Interfaces are written in an *interface definition language* (IDL). The prototype has an interface compiler which creates a C header file for each interface definition. The interface compiler also generates skeletons for (class) object implementations. A skeleton provides the necessary glue to turn a method invocation on a (local) object into a C function call. The programmer only has to implement one C function for each method.

The interface compiler also generates composite objects. A composite object encapsulates a collection of subobjects and allows them to be treated as a single

entity. For example, our local object is a constructed as a composition consisting of the four subobjects shown in Figure 2. These four subobjects are written manually with the help of generated skeletons. Generally, replication and communication subobjects are selected from a collection of subobjects supplied with the prototype.

A class object (containing the implementation of a subobject) is stored on the local file system, and is loaded at runtime into a process' address space. A configuration file specifies for each class name the file in which the corresponding class object is stored.

Persistent distributed shared objects are supported by object repositories. An object repository provides a distributed object with support for storing its state persistently (on disk). It can activate objects that have been passivated, that is, removed from address spaces. An object repository also provides a **factory object**: a distributed object that creates new persistent objects. An object repository is a simple Unix process that stores the state of the object it manages in files in a Unix file system. In the prototype, each factory creates only one type of object. During the configuration of an object repository, it is specified what object types it can create.

An application uses a distributed object by binding to it. The prototype provides simple (Unix style) programs that create and delete distributed objects, list the contents of directories, write and read objects, etc. Our largest application so far is a Web proxy that converts HTTP requests into method invocations on distributed shared objects. To bind to an object, applications use a location service and a name service. The location service is implemented as a simple, centralized database. The name service is constructed as a collection of distributed directory objects.

Initial Prototyping Experiences

To allow concurrent access to objects, Globe supports multithreading. However, it is well known that correctly programming multithreaded applications is difficult. To minimize problems, we follow an approach in which two types of threads are strictly separated. Pop-up threads, which are used to handle requests coming in through the network, are allowed to invoke only methods from callback interfaces, except for methods of the semantics subobject. Likewise, threads originating from the local client can invoke only methods from regular interfaces. Furthermore, subobjects are programmed in such a way that critical regions need never be locked while a call is being made to another subobject.

As we explained, we have developed and implemented an interface definition language (IDL). Our IDL is similar to, for example, CORBA IDL, except that we can also support interface specifications for local objects. Interfaces in C and

Java are generated from IDL descriptions. This approach has shown to be highly effective leading to well-designed subobjects. Nevertheless, the control subobject currently has to be made by hand, which unnecessarily complicates object construction. It is better to specify the semantics subobject in an Object Definition Language (ODL), from which, together with IDL descriptions, we can generate the control subobject. We are currently developing an ODL for Globe.

Being able to implement policies on a per-object basis proved to be highly effective. For example, because we were initially not interested in persistence, we used a single database to store the state of different distributed shared objects. The problem with this approach, which is basically the same as the one followed in CORBA, was that too many policy decisions had to be implemented outside the control of the object being stored. Later, we decided to follow more closely the Globe paradigm, by which each object is in full control of handling its own state. In our current prototype, each object implements its own persistence facilities, as well as the policy that go along with it. This approach has turned out to be much more flexible and, in fact, easier to implement and maintain.

The performance of our prototype, which is currently dominated by the time it takes for a process to bind to an object, confirmed that the granularity of distributed shared objects should be relatively large. For wide-area objects, network speed and delay will additionally determine performance. Granularity is determined by the size of the semantics subobject. Unfortunately, in our model, a replication strategy operates on the entire state as contained in this subobject. This approach is not always appropriate. For example, when a semantics subobject is built from a number of Web pages, including icons, images, etc., we would like to apply different strategies for different parts of the subobject. Developing each part as a separate distributed shared object has an unacceptable performance penalty. We are currently investigating how we can support composite semantics subobjects whose elements can have separate replication strategies.

A Java-based Prototype for the Web

Based on our first prototyping experiences, we are currently developing an implementation of Globe tailored to support scalable Web documents. [9] A Globe Web document is a collection of logically related Web pages. A page may consist of text, icons, images, sounds, animations, etc., as well as applets, scripts, and other forms of executable code. Each Globe Web document is constructed as a distributed shared object.

Instead of using C, we have chosen Java as our implementation language. Construction of a Globe Web document proceeds as follows. The elements that comprise the document (i.e., text, icons, applets, etc.) are grouped together into what is called a *state archive*. As its name suggests, a state archive contains the

state of the semantics subobject.

A semantics subobject offers a standard interface. For example, it is possible to add, remove, or replace elements. At present, each element is represented as a byte image, and has an associated MIME type. Besides a standard interface for a semantics subobject, we offer a standard implementation of a control subobject, and implementations for the interfaces of the replication and communication subobjects. These implementations jointly constitute a template for a local object of a Globe Web document.

Finally, a developer has to choose Java classes that implement the interfaces of the replication and communication subobjects. This leads to one or more *class archives*. Basically, a class archive contains a Java implementation of a specific replication strategy. The state archive, local object template, and a class archive are then grouped together into a single file from which a local object can be instantiated. If no suitable class is available, the implementer is free to write a new one.

To integrate our documents into the current Web infrastructure, we use a filtering gateway that communicates with standard Web clients (e.g. browsers), as shown in Figure 4. The gateway is a proxy that runs on a local server machine and accepts regular HTTP requests for a document. In our model, Globe Web documents are distinguished from other Web resources through naming. A Globe name is written as a URL with globe as scheme identifier. So, for example, globe://cs.vu.nl/~steen/globe/ could be the name of our project's home document, constructed as a distributed shared object.

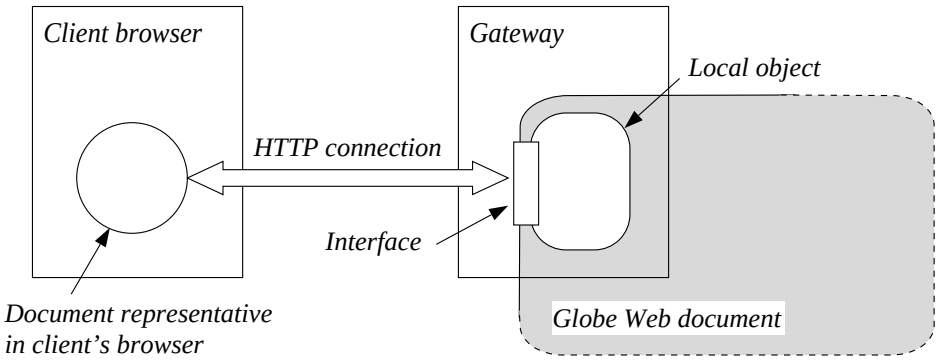

Figure 4: The general organization for integrating Globe Web services into the current Web.

The gateway accepts all URLs. Normal URLs are simply passed to existing (proxy) servers, whereas Globe URLs are used to actually bind to the named

distributed shared object. Unfortunately, existing browsers cannot handle Globe names, for which reason we embed these names in URLs with http as scheme identifiers. In addition, we use Java applets to support interactive documents. We are investigating the use of browser plug-ins to allow browser extensions for support of Globe's distributed shared objects.

Discussion

With the exponential growth of the Web, it is clear that we need a highly scalable infrastructure for implementing a wide variety of applications. Globe provides such an infrastructure.

An important aspect of our model is that partitioning, replication, and migration of an object's state is supported on a per-object basis. Different objects can use different strategies: each object fully contains an implementation of its own strategy, independent of other objects. This makes it much easier to have very different objects interoperate, for the simple reason that each hides its internals from the other behind well-defined interfaces. More importantly, is that by providing a mechanism for implementing distribution policies on a per-object basis, we can tackle worldwide scalability. In our view, the next generation of distributed systems will have to support a wide variety of objects that can be invoked from anywhere. The only way to achieve worldwide scalability is to provide extensive support for partitioning and replicating objects, and allow very different consistency strategies to co-exist.[7] Globe provides this flexibility.

We have finished the initial architectural design of our system, leaving a number of subjects open for further research. For example, we are currently working on the design of a security architecture. Furthermore, we are concentrating on specific schemes for wide-area replication and persistence, mechanisms that support large-scale applications composed of many distributed objects, and persistence. Our current efforts concentrate on developing a Java-based implementation for constructing scalable Web documents. More information on Globe can be found at our home page http://www.cs.vu.nl/~steen/globe/.

Acknowledgments

Many ideas presented in this paper have been formed during discussions with other participants in the Globe project: Arno Bakker, Gerco Ballintijn, Franz Hauck, Anne-Marie Kermarrec, Ihor Kuz, and Patrick Verkaik. We gratefully acknowledge their contribution. Leendert van Doorn is acknowledged for his contributions to the detailed design and implementation of local objects. Finally, we

thank Henri Bal, Koen Langendoen and the anonymous referees for their valuable assistance in improving this paper.

References

1. Microsoft Corporation. *DCOM Technical Overview*, 1996.
2. W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Sebastopol, CA., 1992.
3. OMG. "The Common Object Request Broker: Architecture and Specification, revision 2.0." OMG Document 96.03.04, Object Management Group, Mar. 1996.
4. M. Satyanarayanan. "Scalable, Secure, and Highly Available Distributed File Access." *Computer*, 23(5):9–21, May 1990.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA., 1994.
6. N. Islam. "Customizing System Software Using OO Frameworks." *Computer*, 30(2):69–78, Feb. 1997.
7. B. Neuman. "Scale in Distributed Systems." In T. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 463–489. IEEE Computer Society Press, Los Alamitos, CA., 1994.
8. M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. "Locating Objects in Wide-Area Systems." *IEEE Commun. Mag.*, 36(1):104–109, Jan. 1998.
9. M. van Steen, A. S. Tanenbaum, I. Kuz, and H. J. Sips. "A Scalable Middleware Solution for Advanced Wide-Area Web Services." In *Proc. Middleware '98*, The Lake District, England, Sept. 1998. IFIP.

An Example: Scalable World-Wide Web Documents

To illustrate the benefits of our approach, we consider how Globe offers the facilities for support of scalable Web documents. A Web document is taken to be a collection of logically related pages, including their icons, sounds, applets, etc.

Proposals for caching or replicating Web documents tend to treat pages alike in the sense that the semantics of a document are not taken into account. Documents and their pages are treated differently only by considering metadata such as access statistics, times of modification, etc. Alternatively, some solutions are tailored to a specific class of documents and are not universally applicable.

As Web documents are becoming more diverse, it is clear that it will be hard to find a single solution that can be used in all situations. For example, current caching and replication schemes for the Web assume that pages are modified at only one location. They are not suited to support Web pages that are actively shared by several users, such as shared whiteboards and pages manipulated through groupware editors. Likewise, it is hard to tailor a replication scheme to just a single document, as is needed with mail distribution lists.

The approach followed in Globe is radically different. Rather than searching for generally applicable replication schemes, each distributed object can adopt its own strategy. Globe offers a library of different replication subobjects (see Figure 2 in the main text) that can be adopted and subsequently fine-tuned separately for each distributed object. When required, new ones can be constructed.

For example, consider the current major application of the Web, namely providing information through a logical Web site, also called home pages. A home page is related to a person, project, consortium, organization, etc., and is generally the entry point of an entire hypertext document consisting of multiple pages. Typically, in Globe, this document would be modeled as one distributed shared object. The state of such a document consists of the rooted directed graph of individual pages that make up the hypertext document.

Web sites can be very different with respect to the kind of documents they manage. Pages of a personal site would generally hardly require any replication, and possibly only short-lived caching. In Globe, the owner of a personal site would group the pages into a single document, and provide only a single contact address. When a user binds to that document, its pages (including icons, images, etc.) are transferred to the user's browser, possibly in parts as in current practice, and are subsequently written to the user's private cache. Note that there is generally no need to write pages of such a document to a site-wide cache as is done by Web proxies.

On the other hand, an organization's Web site may be of an entirely different nature. First, we may assume that its popularity is much higher than that of personal Web sites. Also, in the case of multinational organizations, readers will

come from all over the world. In these cases, a primary–backup approach where pages are replicated to a number of mirror sites is useful. The organization's Web site could be constructed as one or more Web documents, where each document is registered at the location service with multiple contact addresses. The nearest address is always returned to a user. Note that, in Globe, the name of a Web document can be the same everywhere. Also, there is no need to tell the user that there are mirror sites, and where these sites are. In contrast to personal Web documents, site-wide caching as is done by current Web proxies, may now be useful.

There are also Web sites whose content change rapidly and which may require active replication schemes. For example, Web documents of online news providers may want to use a publish/subscribe type of replication by which subscribers to a provider's document are notified when news updates occur. This also holds for Web documents related to conferences and other types of timely events. In the current Internet infrastructure, automatic notification is often done by making use of mailing lists. Such lists are highly inefficient. In Globe, notification would be an integral part of the Web document, using a multicasting scheme appropriate for that document. Of course, notification could be combined with actively replicating the updates, but this may not be appropriate in all cases.

What we see here are similar Web documents, but that require very different replication strategies. Personal home pages need not be replicated, and should be cached on a per-user basis. Organizational home pages can apply primary–backup replication, and should be cached per site. Home pages related to timely events may benefit from a publish/subscribe type of replication where clients are notified when updates occur. Other examples where different replication policies are required can easily be thought of. Unfortunately, such distinctions are presently impossible to make. In Globe, however, each Web document can use a replication strategy tailored to its own characteristics.

Related Work

There is much academic and industrial activity on the design and implementation of shared data and objects. A shared-data model offers a small set of primitives for reading and writing bytes to shared regions of storage. Typical examples of shared-data models are network file systems and distributed shared memory (DSM) implementations. The main problem is achieving performance and scalability while keeping data consistent. Distributed shared memory and storage systems such as Munin [1] and Khazana, [2] respectively, follow an approach similar to Globe by attaching replication policies on a per-region basis. In most DSM systems, performance is improved by relaxing memory consistency. [3] The main drawback of the shared-data model is that it simply does not provide the level of abstraction needed for developing distributed applications. Therefore, much attention is being paid to object-based approaches.

Objects come with an architectural model that lends itself well for distributed systems. An object can be seen as a fine-grained service provider. To most developers, this means that an object is naturally implemented through its own server process, which handles requests from clients. This view leads to the remote-object model in which a remote-method invocation is made transparent using RPC-like techniques as is done in DCOM. [4] However, this approach is the major obstacle to scale worldwide. The problem is that remote-object invocations cannot adequately deal with network latencies. Additional mechanisms such as object replication and asynchronous method invocations are therefore necessary.

In the Legion system, [5] objects are located in different address spaces, and method invocation is implemented nontransparently through message passing. The Legion approach is one of the few which explicitly addresses wide-area scalability. The Globus project has developed global pointers to support flexible implementations. [6] A global pointer is a reference to a remote compute object. The pointer identifies a number of protocols to communicate with the object, of which one is to be selected by the client. Global pointers offer a higher degree of flexibility than the Legion approach.

When it comes to distribution transparency, Legion and Globus fall short. Transparency is explicitly addressed by object request brokers (ORBs). An ORB is a mediator between objects and their clients. Basic ORBs provide only support for language-independent and location-transparent method invocation. CORBA-compliant ORBs [7] offer additional distribution services such as naming, persistence, transactions, etc. Unfortunately, CORBA has not yet defined services for transparently replicating objects, or for keeping replicas consistent.

When an ORB is responsible for distribution services, we require additional mechanisms independent of the core object model. One such mechanism is the subcontract used in the Spring system. [8] A subcontract implements an *invocation*

protocol: it describes the effect of a method invocation at the client side in terms of the method invocation(s) at the object's side. For example, in the case of replication, method invocation by a client may result in the invocation of that method at each replica. Replicating the invocation is encapsulated in the subcontract and is hidden from the client. As a general mechanism, subcontracts are too limited. For example, it is hard to develop subcontracts that keep a group of objects consistent that are being shared by several clients.

An alternative approach is to fully encapsulate distribution in an object, leading to a model of *partitioned objects*. Partitioned objects appeared in SOS in the form of fragmented objects. [9] Globe's distributed shared objects form another implementation of partitioned objects, and have been derived from the Orca [10] programming language.

Fragmented objects in SOS are mostly language independent. Distribution is achieved manually by allowing interfaces to act as object references that can be freely copied between different address spaces. An important difference with Globe's distributed shared objects, is that fragmented objects make use of *relative* object references. In contrast, Globe's object handles are absolute and globally unique. Fragmented objects have not been designed for wide-area networks. For example, there are no facilities for incorporating object-specific replication strategies. Likewise, the communication objects have been designed and implemented for local-area networks only. Furthermore, the model does not provide facilities for implementing different coherence policies, nor does it address the problem of platform heterogeneity.

References

1. J. Carter, J. Bennett, and W. Zwaenepoel. "Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems." *ACM Trans. Comp. Syst.*, 13(3):205–244, Aug. 1995.
2. J. Carter, A. Ranganathan, and S. Susarla. "Khazana: An Infrastructure for Building Distributed Services." In *Proc. 18th Int'l Conf. on Distributed Computing Systems*, pp. 562–571, Amsterdam, May 1998. IEEE.
3. J. Protić, M. Tomašević, and V. Milutinović. "Distributed Shared Memory: Concepts and Systems." *IEEE Par. Distr. Techn.*, 4(2):63–79, Summer 1996.
4. Microsoft Corporation. *DCOM Technical Overview*, 1996.
5. A. Grimshaw and W. Wulf. "The Legion Vision of a Worldwide Virtual Computer." *Commun. ACM*, 40(1):39–45, Jan. 1997.
6. I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. "Managing Multiple Communication Methods in High-Performance Networked Computing Systems." *J. Par. Distr. Comput.*, 40:35–48, 1997.
7. OMG. "The Common Object Request Broker: Architecture and Specification, revision 2.0." OMG Document 96.03.04, Object Management Group, Mar. 1996.

8. G. Hamilton, M. Powell, and J. Mitchell. "Subcontract: A Flexible Base for Distributed Programming." In *Proc. 14th Symp. on Operating System Principles*, Asheville, N.C., Dec. 1993. ACM.
9. M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. "SOS: An Object-Oriented Operating System – Assessment and Perspectives." *Computing Systems*, 2(4):287–337, Fall 1989.
10. H. Bal and A. Tanenbaum. "Distributed Programming with Shared Data." In *Proc. Int'l. Conf. on Computer Languages*, pp. 82–91, Miami Beach, FL., Oct. 1988. IEEE.

Biography

Maarten van Steen is assistant professor at the Vrije Universiteit in Amsterdam since 1994. He received an M.Sc. in Applied Mathematics from Twente University (1983) and a Ph.D. in Computer Science from Leiden University (1988). He has worked at an industrial research laboratory for several years in the field of parallel programming environments. His research interests include operating systems, computer networks, distributed systems, and distributed-software engineering. Van Steen is a member of IEEE Computer Society and ACM.

Philip Homburg graduated in 1991 at the Vrije Universiteit, Amsterdam. Before starting with his Ph.D. study, he wrote software to transparently connect multiple Amoeba sites over the Internet as part of the Starfish project. Currently, he is a Ph.D. student working on the overall design of Globe.

Andrew S. Tanenbaum has an S.B. from M.I.T. and a Ph.D. from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam and Dean of the interuniversity computer science graduate school, ASCI. Prof. Tanenbaum is the principal designer of three operating systems: TSS-11, Amoeba, and MINIX. He was also the chief designer of the Amsterdam Compiler Kit. In addition, Tanenbaum is the author of five books and over 80 refereed papers. He is a Fellow of ACM, a Fellow of IEEE, and a member of the Royal Dutch Academy of Sciences. In 1994 he was the recipient of the ACM Karl V. Karlstrom Outstanding Educator Award and in 1997 he won the SIGCSE award for contributions to computer science.