

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

Maarten van Steen (contact)
Philip Homburg
Andrew S. Tanenbaum

Internal report IR-422
March 1997

Abstract. Developing large-scale wide-area applications requires an infrastructure that is presently lacking entirely. Currently, 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.

We 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 an 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. We describe how distributed objects can be used to implement worldwide scalable Web objects.

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



vrije Universiteit

1 Introduction

The introduction of the World-Wide Web has radically changed our view on networking. As we move closer to the year 2000, it is becoming clear that an entirely new environment is appearing, one dominated by hundreds of millions of computers allowing users all over the world to interact and collaborate in ways as yet unforeseen. For organizations, we see the construction of large-scale, enterprise-wide intranets as virtual private networks supporting a completely new way of doing business. It is clear that these developments will lead to a new generation of distributed applications that support large numbers of users in wide-area networks.

Unfortunately, it is also clear that we have a major problem at hand: developing large-scale wide-area applications requires an infrastructure that is presently lacking entirely. What we see today is that applications have to be built 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, persistence, fault tolerance, and security.

Consider the World-Wide Web, which implements its own communication protocol, HTTP, on top of TCP. The Web uses a tailor-made naming system based on URLs. Replication is supported slightly 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. Persistence is implemented by making use of local file systems. Hardly any measures have been taken to handle broken links and server crashes. Finally, security has only recently been proposed in the form of an extension to HTTP, but there are also proprietary solutions such as SSL from Netscape.

There are two major drawbacks to this approach. 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. Obviously, we should follow an entirely different approach. Rather than developing applications directly on top of the transport layer, we should be taking advantage of a technical infrastructure that provides us with a set of common services. The main requirement is that this infrastructure can scale to support millions of users all over the world.

2 Supporting Large-Scale Wide-Area Applications

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.

2.1 A Uniform Model for Distributed Computing

An important design issue for any distributed system is that it provides a consistent and uniform view of how to organize applications built on top of it. DCE[1], for example, is designed to support traditional client-server computing using RPCs as the foundation for all communication. Likewise, CORBA[2]

Table 1: Different kinds of distribution transparency relevant for distributed systems[4]

Transparency	Description
Access transparency	Hides differences in data representation and invocation mechanisms
Failure transparency	Hides failure and possible recovery of objects
Location transparency	Hides where an object resides
Migration transparency	Hides from an object the ability of a system to change that object's location
Relocation transparency	Hides from a client the ability of a system to change the location of an object to which the client is bound
Replication transparency	Hides the fact that an object or its state may be replicated and that replicas reside at different locations
Persistence transparency	Hides the fact that an object may be (partly) passivated by the system
Transaction transparency	Hides the coordination of activities between objects to achieve consistency at a higher level

supports object-based applications in which all distributed computing aspects are expressed in terms of invocations on possibly remote objects. As another example, AFS[3] offers its users a relatively simple way for distributing information by means of a wide-area file system that is based on a location-transparent file name space. Finally, consider the Web. Although restricted in its functionality, the Web offers its users the relatively simple model of worldwide distributed documents tied together through hyperlinks.

These, and a few other distributed systems have in common that they offer a single, uniform model for distributed computing. A uniform model contributes to a single-system view. However, a model should additionally 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, of which different kinds are shown in Table 1.

At best, current distributed systems offer only a uniform model. Systems that also integrate common services and support all the types of distribution transparency shown in Table 1 do not exist at present. For example, the Web misses a number of important services such as replication and fault tolerance. Its support for distribution transparency is poor. AFS does a better job at distribution transparency, but it is not universal enough to act as a common platform for all applications: the concept of a file is simply too restrictive. CORBA aims at providing a uniform integrated model by following an object-based approach. However, like nearly all distributed-object models, an object is taken as the unit of distribution. This means that an object is assumed to be located at one machine, although it may possibly be mobile, or have replicas on other machines. The drawback of this approach is that we can devise only general-purpose distribution policies that should, in principle, be applicable to *all* objects. We argue, however, that we need mechanisms for implementing *object-specific* policies. Such policies should be entirely encapsulated by an object. A similar reasoning holds for DCE which has not been designed to support application-defined distribution policies, but instead provides an integrated set of common, general-purpose services.

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. Moreover, all implementation aspects are hidden behind its interfaces to achieve distribution transparency. The Globe object model is explained in detail below.

2.2 A Flexible Implementation Framework

Having a uniform model for distributed computing in wide-area systems implies that we have to deal with different computers, operating systems, and networks. These differences should, in general, be transparent to applications to achieve portability. For example, it should be easy to develop an application that can be distributed across a set of platforms consisting of Unix-based high-end workstations as well as PCs running Windows. However, making the underlying platforms entirely transparent is not always a good idea. For example, a computational intensive part of a wide-area application may benefit from exploiting parallelism by making use of a special-purpose parallel computer. Likewise, other parts may be tailored to use the facilities of a multimedia workstation. A wide-area distributed system should thus make local, possibly non-standard facilities available to applications when needed.

Likewise, there are occasions in which the underlying network should not be entirely hidden. For example, many sites today are still connected to the Internet through a low-bandwidth link. Consequently, when dealing with a client-server application, clients may perceive poor performance. In these cases, a possible solution is to move data and computations from the server to the client, thus reducing the number of calls to the server. This is effectively the approach supported by Java. What we see here, is that the quality of a communication service can influence how the client side of an application is implemented. In general, quality of service parameters should be made available to applications so that they can choose an appropriate implementation policy.

We can even ask ourselves how, and to what extent distribution transparency should be provided. For example, consider multicasting. A shared whiteboard requires multicasting to be reliable. In contrast, an audio or video broadcast will put high demands on real-time qualities, but it is acceptable that packets are occasionally lost. In addition, where the shared whiteboard probably deals with a relatively small group of users consisting of multiple writers, the audio/video broadcast may be intended for millions of receivers but having only one sender. No single multicasting facility can support both types of applications efficiently. Instead, the distributed system should provide only a simple, unreliable form of multicasting, preferably with a set of easily configurable extensions that can be adopted on a per-application basis.

These examples show that we need 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 towards a specific application by choosing the appropriate object implementations, and, where necessary, extend the design with other objects[6]. This is the approach followed in Globe.

2.3 Worldwide Scalability

The most important design issue is scalability. It is conceivable that we may eventually have to support one billion users, each having thousands of objects. Objects include Web pages and other electronic

documents, dynamically downloaded programs such as Java applets, public domain software, high-performance computing code that is invoked remotely, etc.

To achieve scalability, we must exclude nonscalable components (including protocols, algorithms, and storage strategies). Examples include a single point of failure for the whole system, or a global synchronization by means of a systemwide barrier. Many systems fail to meet this requirement. For example, although DCE has clearly been designed with scalability in mind, its RPC-based foundation makes it hard to deal with latencies that are inherent in wide-area networks. A similar reasoning applies to CORBA which implicitly assumes that every object's state is located in a single address space. This means that the model supports only remote-object invocations, which have the same drawback as RPCs. In such cases, the application developer is forced to take special measures, for example by using threads to implement a form of asynchronous method invocation. Such solutions can easily lead to obscure, and difficult to maintain code.

Excluding nonscalable components is not always enough. For example, the Web uses only scalable components, but still has scalability problems caused by the lack of support for replication. We should therefore not look only at whether components scale, but consider scalability with respect to the system as a whole. Designing for wide-area scalability has to be done from the start. This has been the case, with varying success, in only a few distributed systems such as AFS and DCE.

3 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 existing networks and operating systems, in particular the Internet and systems such as UNIX and Windows NT, respectively. 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. In contrast to most approaches, Globe is capable of supporting millions of users, each having thousands of shared objects that are distributed worldwide.

3.1 The Globe Object Model

In Globe, processes interact and communicate through **distributed shared objects**. Each object offers one or more **interfaces**, each consisting of a set of methods. Objects are passive, but multiple processes may simultaneously access the same object. Changes to the object's state made by one process are visible to the others. 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 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.

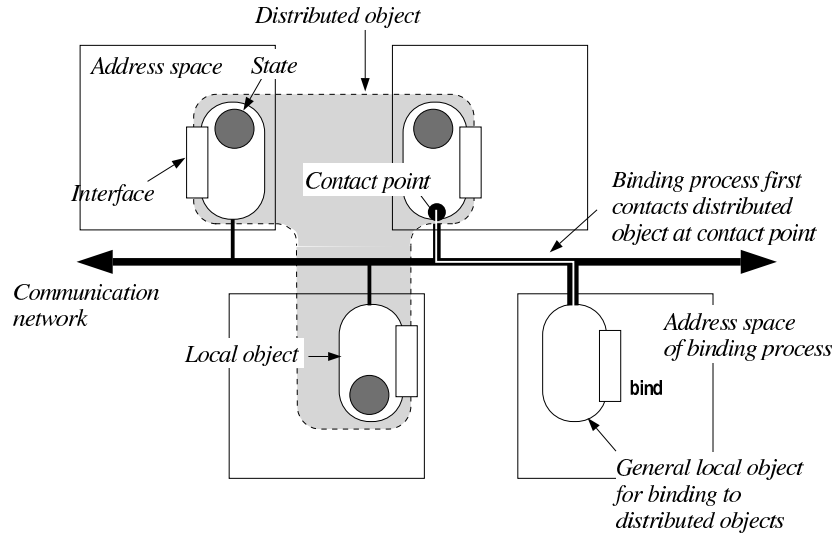


Figure 1: Example of an object distributed across three address spaces.

3.1.1 Local Objects

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. However, such implementation details are transparent to the client processes: they see only the interface to the distributed object as offered by the local object.

Most local objects are generally organized as **composite objects**: objects that aggregate one or more objects, some of which may themselves be composite objects. A composite object also resides in a single address space, that is, it is a local object. A **primitive object** is a nonaggregate local object. Clients cannot tell whether an object is composite or primitive: all implementation aspects are again hidden behind its interfaces. An interface of a composition refers to methods available at its constituents. More specifically, we implement an interface as an array of $(state, method)$ -pairs of pointers, similar to the approach followed in COM[7]. Using such binary (i.e. runtime) interfaces has the advantage that Globe objects are language independent, making runtime adaptations more easy.

We normally store code separately in a **class object**. This is a local object that contains the method implementations for objects belonging to the same class. Every local object has an associated class object. By separating code from state we can easily instantiate several instances of the same class without having to duplicate the code for each instance. More importantly, is that class objects in combination with our interface-based approach provide us the flexibility for dealing with platform heterogeneity. By

using runtime interfaces, object invocations are entirely language and platform independent. How an object's interface is actually implemented is determined by a class object, but this is completely hidden to client processes. Obviously, there may be several class objects that implement the same interface, for example, when different machine architectures need to be supported.

3.1.2 Distributed Shared Objects

A distributed object is composed of local objects located in different address spaces (Figure 1). The local objects are responsible for establishing communication between the different address spaces in such a way that a client of the distributed object is provided with a consistent view on the object's state. Local objects are thus responsible for hiding distribution, replication, and migration of the object's state from the user.

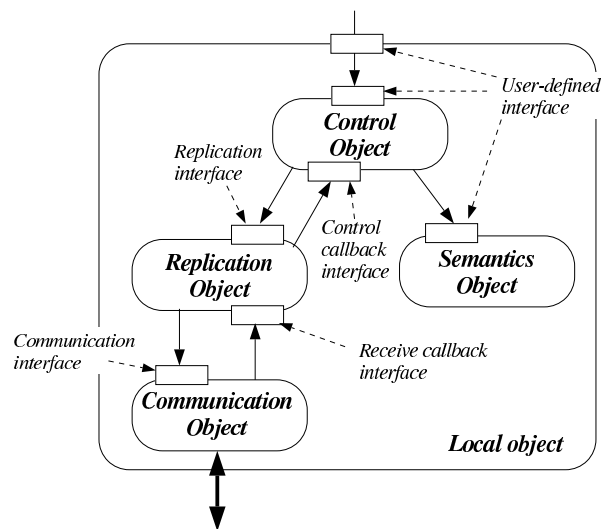


Figure 2: The general implementation of a distributed object.

Each local object forming part of a distributed object is a self-contained composite object as shown in Figure 2. A minimal composition consists of the following four components.

Semantics object. This is a local object that implements (part of) the actual semantics of the distributed object. A developer is responsible for constructing a class object for each different kind of semantics object that is part of the distributed object (e.g., by writing the code in C, C++, or some other language). A semantics object can also be constructed as a composition. In principle, the semantics objects are the only objects a developer needs to construct personally. All other parts can either be obtained from libraries, or are generated from interface specifications.

Communication object. This is generally a system-provided local object. It is responsible for handling communication between parts of the distributed object that reside in different address spaces.

Depending on what is needed from the other components, a communication object may offer primitives for point-to-point communication, multicast facilities, or both. We allow several communication objects to co-exist in the same local implementation of a distributed object. We have designed a stable communication interface that allows for a variety of different implementations. For example, we currently have implementations for network protocols such as UDP, IP multicast, and TCP-based implementations, that all provide the same interface.

Replication object. The global state of the distributed object is made up of the state of its various semantics objects. Semantics objects may be replicated for reasons of fault tolerance or performance. The replication object is responsible for keeping these replicas consistent according to some replication strategy. The replication object operates only on marshalled data. Different distributed objects may have different replication objects, using different replication algorithms. A replication object is controlled by the control object.

Control object. The control object takes care of invocations from client processes, and controls the interaction between the semantics object and the replication object. It is responsible for marshalling parameters and state, and passing these to the replication object when requested. Incoming invocation requests are also handled by the control object.

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 object. In our view, the only way to achieve wide-area scalability of objects is to concentrate on the distribution of its state, i.e. the partitioning, replication, and migration of state. Moreover, with the enormous variety of objects, it is also clear that a general-purpose distribution policy will never suffice. Therefore, what we need is a facility to implement distribution policies on a per-object basis. This is what our local replication object provides. It is responsible for keeping replicas of (parts of) the object's state consistent, and, where necessary, assists in moving that state between different address spaces.

The main role of our communication objects is that they provide a uniform interface to underlying networks and operating systems concerning their communication 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 objects are also independent of the semantics object in the sense that they are completely unaware of its methods and state. Instead, both the communication object and the replication object operate only on invocation messages in which method identifiers and parameters have been encoded. This independence allows us to define standard interfaces for all replication objects and communication objects. Consequently, we can implement different policies in separate replication and communication objects, but keep the interfaces the same. Combining this with our runtime interfaces, this also means that we can now easily adopt a policy by choosing an appropriate implementation from a library of class objects, and dynamically download that implementation into our local object framework.

Finally, state is implemented through semantics objects, and partitioned state is represented by having different semantics objects in different address spaces. In contrast to the the local objects for replication and communication, interfaces for the semantics objects are defined by the developer. This explains the need for a control object, which we generate from a semantics object very similar to the generation

of stubs in the case of client–server computing.

3.2 Process–to–Object Binding

In our model, processes communicate by sharing a distributed object. For example, a USENET news-group could be implemented as a distributed object, and be shared between processes that post and read articles belonging to that group. Likewise, communication by electronic mail could be realized by implementing a mailbox as a distributed object. The owner of the mailbox would have read and write permissions, whereas other processes would only be allowed to deposit new messages. The mailbox would be dynamically shared between senders and its owner.

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 object, and (2) installing the interface. This is illustrated in Figure 3. Finding an object is separated into a name lookup and a location-lookup step; installing the interface requires that we select a suitable contact address, as well as an implementation for that interface.

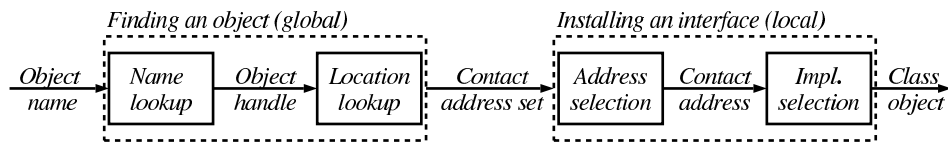


Figure 3: Binding a process to a distributed shared object.

3.2.1 Finding an Object

To find an object, a process must pass a name of that object to a naming service that can resolve that name. 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. The object handle is given to a location service, which returns one or several contact addresses. Globe thus uses a two-level naming hierarchy as shown in Figure 4.

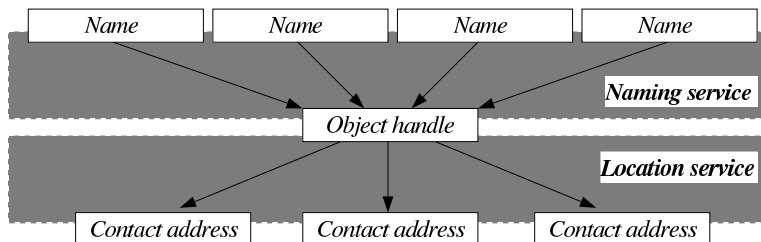


Figure 4: Globe’s two-level naming hierarchy.

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 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. Our location service provides four operations:

- Register an object that has just been created
- Unregister an object that is about to be deleted
- Change the set of contact addresses for an object
- Lookup the set of contact addresses for an object

The goal of the location service is to implement all four of these operations efficiently. This is done by means of a worldwide, distributed search tree, as shown in Figure 5. We divide the world into a hierarchical set of domains. At the lowest level there is one domain per site; a collection of sites form a region, etc. An object is recorded at each site where it has a contact address, and recursively in each enclosing region, up to the root of the tree.

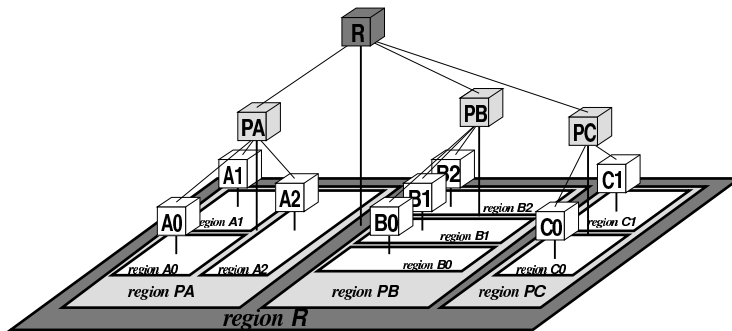


Figure 5: Globe’s worldwide search tree used for locating objects.

Initially, a record at the site level contains the actual contact addresses and records at higher levels pointers to the next lower level. Recording an object at multiple levels allows searches with expanding rings: a search starts at the local site, followed by the local region, then the next higher level region, etc., and eventually followed by the root. Searching with expanding rings provides the desired locality. If the object has a contact address in the site or region of the requesting process, then contact addresses will be found with only local communication.

To be able to actually implement regional and world nodes, we apply a partitioning scheme by which each site stores a part of the root node and parts of the regional nodes. The partitioning can be done by computing a hash value from the object handle.

3.2.2 Installing an Interface

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. We contain descriptions of protocol stacks in special distributed shared objects, called **protocol objects**. A protocol object is somewhat comparable to a published Internet RFC in the sense that it contains a protocol specification. Such a specification can be given in natural language, or perhaps using some formal notation. In contrast to a published RFC, a protocol object can additionally contain one or more reference implementations. For example, a protocol object may contain class objects that can be downloaded by a process and instantiated locally. In other words, a protocol object also acts as an implementation repository for its associated protocol. We expect that protocol objects will, in general, hardly ever be updated, so that they can be easily distributed and replicated worldwide.

As an alternative to downloading an implementation from a protocol object, we anticipate that most clients prefer a trusted local implementation instead. In this case, the protocol stack is constructed from components stored in a local implementation repository. Each component specifies exactly which (part of a) protocol it implements by means of the corresponding protocol identifier. Using that identifier, a process searches the local implementation repository for the appropriate components. Components are again stored as class objects. They are subsequently loaded into the process' address space and instantiated.

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.

4 An Example: Scalable World-Wide Web Objects

To illustrate the benefits of our approach, we consider how Globe offers the facilities for support of scalable World-Wide Web objects. Let us first consider a major problem with the current Web: its lack of performance.

4.1 Current Performance Problems with WWW

The exponential growth of the World-Wide Web community is leading to a serious performance problem. Where we experienced data transmission rates on the order of 10 Kbyte/sec a few years ago, rates now often drop to only a few hundred bytes per second. This is unacceptable.

To alleviate these problems, most sites install caches, which are integrated with Web proxies. Although studies have shown that Web caches can be effective, there is still the drawback that they are generally filled on demand. To avoid passing a stale page to its users, a proxy has to check whether the requested page is cached, and if so, whether it is still up-to-date. Consequently, contact must always be made with the page's server, and the page will have to be transferred when the cache entry turns out to be stale.

Performance can be improved if a weaker cache coherency protocol is applied. For example, in the Alex global file system[9], a staleness factor is introduced, defined as:

$$\text{staleness} = \frac{T_{\text{now}} - T_{\text{last update}}}{T_{\text{last update}} - T_{\text{last modification}}}$$

In other words, a page that is 10 days old will be 10% stale after being cached for 1 day. The idea is to refresh only those pages for which the staleness has exceeded some threshold. The advantage of this approach is that older objects may be cached longer than ones that have just been modified. The underlying assumption is that older objects are less likely to change than newer ones. Note that in this scheme it is not necessary to contact the server, provided that the time of the last modification is stored when the object is cached. On the other hand, the client runs the risk of looking at a stale page without knowing it.

The problem remains that the decision to refresh a cache entry is made at the client side, whereas up-to-date information on the status of the cached object is always at the server side. This suggests adopting server-initiated invalidation protocols, but these do not scale. An intermediate scheme is to have a server record where its objects are cached so that it can invalidate those caches when necessary[10]. As invalidation messages may be lost, the proxy cache uses its own coherence protocol as a fall-back mechanism. However, in general, as the Web grows and the fraction of pages frequently referenced falls, no caching scheme will work well.

One step further is to adopt true replication schemes. Gwertzman and Seltzer propose using *geographical push-caching*[11]. In this scheme, a server replicates a popular object to a location where it is frequently requested, preferably by many clients in possibly different domains. The scheme requires that such clients can share a single cache at a nearby site. A similar approach has been proposed by Baentsch et al.[12] which makes use of application-level multicasting. The distribution point model[13] is also based on multicasting, but is tailored towards active replication of relatively static sets of bulk, non real-time data. It is mainly applicable to magazine-like Web documents such as those that appear as electronic periodical publications.

We note that, in order to be effective, most replication schemes require that a client needs to find the *nearest* replica. This introduces an additional naming problem, as URLs contain hard-coded information on where the main object resides.

4.2 Scalable Web Objects in Globe

All proposals for caching or replication tend to treat objects equally in the sense that the semantics of an object are not taken into account. Objects 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 objects and are not universally applicable.

As Web objects 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, all caching and replication schemes assume that objects 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 object, 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 collection of local replication objects (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 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 *home object* consists of the rooted directed graph of individual pages that make up the hypertext document.

Home objects can be very different. A personal home object would generally hardly require any replication, and possibly only short-lived caching. Therefore, its owner would register the object at the location service and provide only a single contact address. When a user binds to that home object, the object's state is transferred to the user's address space, possibly in parts as in current practice, and is subsequently written to the user's private cache. Note that there is generally no need to write personal home objects to a sitewide cache as is done by Web proxies.

On the other hand, an organization's home object may be of an entirely different nature. First, we may assume that its popularity is much higher than that of personal home objects. Also, in the case of multinational organizations, readers will come from all over the world. In these cases, a primary-backup approach where the object's state is replicated to a number of mirror sites is useful. The home object 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 the home object 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 home objects, sitewide caching may now be useful.

There are also home objects that change rapidly and which may require active replication schemes. For example, home objects of online news providers may want to use a publish/subscribe type of replication by which subscribers to the provider's home object are notified when news updates occur. This also holds for home objects 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 home object, using a multicasting scheme appropriate for that object. 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 objects, 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 home object can use a replication strategy tailored to its own characteristics.

As an aside, note that finding replicas is not a problem in our approach. Using our two-level naming hierarchy, a process wanting to bind to a distributed shared object need only pass a location-independent object handle to the location service. Assuming that the replicas register their contact address at the location service, finding the nearest contact address is easy, as the location service searches in incrementally expanding regions. In this sense, it inherently supports anycasting[14].

5 Related Work on Distributed-Object Models

There is much academic and industrial activity on the design and implementation of object-based distributed systems. However, the notion of an object varies considerably. To describe the most important features of existing models and systems, we make the following distinction:

- *Traditional interprocess communication models:* These models represent the important class of distributed systems in which processes communicate by message passing or through operations on shared data.
- *Proxy-based object models:* In these models, an object is located on a single machine, but method invocation is transparent across the network by installing a surrogate or proxy at the client side. Most object models fall into this category.
- *Partitioned-object models:* Here, an object is assumed to be physically distributed across several machines. Very few systems exist or are being developed that support this model. Globe falls into this category.

5.1 Traditional IPC Models

Most distributed systems support a communication model in which processes communicate through message passing, or through operations on shared data.

Despite its low level of abstraction, the message-passing model is popular because it scales relatively well (although few people have thought of dealing with billion-user systems). Examples include applications based directly on TCP and UDP implementations, distributed computations based on communication libraries such as PVM[15] and MPI[16], group communication systems such as ISIS[17], and RPC-based systems like DCE[18]. However, developing distributed applications using message passing is hard. The main limitation is that data placement, replication, consistency, and persistence management are left to the application, and each application must invent and implement these mechanisms all over again. Having each application create its own infrastructure from scratch makes developing new applications unnecessarily difficult.

More recent systems base interprocess communication on operations on shared data. Typical examples are network file systems[19] and distributed shared memory (DSM) implementations[20]. The shared data model offers a small set of primitives for reading and writing bytes. The main problem here is achieving performance and scalability while keeping data consistent. For example, shared files may need to be protected by special locking schemes to achieve performance, and likewise, DSM systems

may need to relax memory consistency. The result is a communication model which can be harder to understand. Furthermore, page-based DSM systems do not provide information hiding, have serious performance problems, and can never scale worldwide to millions of users.

5.2 Proxy-based Object Models

As an alternative, much attention is being paid to object-based approaches. Objects not only make it easier to construct large-scale applications, they also come with an architectural model that lends itself well for distributed systems. In particular, objects can be seen as fine-grained service providers. This means that an object can be naturally implemented through its own server process which handles requests from clients. Remote-method invocation can be made transparent using the same techniques as with RPCs. However, at the same time, this approach is the major obstacle for proxy-based models 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 its simplest form, remote-method invocation is actually implemented the same as a traditional RPC. This is the approach followed in DCOM[21, 22], which is constructed as an enhancement to DCE. Clients in DCOM are offered only interfaces that are implemented by (remote) servers rather than by objects.

More interesting are approaches that explicitly support objects. In the Legion system[23], objects are located in different address spaces, and method invocation is implemented directly through message passing. Although this violates distribution transparency, the Legion approach is one of the few which explicitly addresses wide-area scalability. For example, their model does not prescribe how objects should be implemented, but only how they should interact, thus leaving room for different implementations. The Globus project has developed global pointers to support flexible implementations[24]. 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. Although distribution transparency is still not supported, global pointers offer a higher degree of flexibility than the Legion approach.

Distribution transparency is explicitly addressed by object request brokers (ORBs). An ORB is a mediator between objects and their clients. An object is contained in the address space of a server, and a client invokes methods by means of a surrogate or proxy object in its own address space. Basic ORBs provide only support for language-independent and location-transparent method invocation as in the case of ILU[25]. ORBs that comply to CORBA[26] 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. Whether objects in CORBA can set their own replication policy remains to be seen. In addition, it is unclear to what extent the current services can be implemented in a scalable way.

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[27]. 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. However, as a general mechanism, subcontracts are too limited. For example, it is hard to develop subcontracts that keep a group of objects consistent which are being shared by several clients. Furthermore, Spring was never designed as a wide-area system.

5.3 Partitioned-Object Models

An alternative approach to developing general distribution services, is that we encapsulate the service in the object that uses it. In other words, all distribution aspects of an object, including its location, migration, relocation, and replication, are part of its implementation. This leads to a model of *partitioned objects*. Partitioned objects appeared in SOS in the form of fragmented objects[28]. Globe's distributed shared objects form another implementation of partitioned objects, and have been derived from the Orca object-based shared data model[29]. To our knowledge, there are no other comparable approaches to partitioned objects.

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 object reference can be refined, meaning that a local fragment of the object is instantiated in the client's address space. 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.

The most important difference between the two models is that 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 (called connective objects in SOS) which are used as the foundation for fragmented 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.

6 Evolution Path

Our approach is feasible only if we can find a way to integrate Globe with existing solutions, and allowing a gradual migration towards the use of distributed shared objects. Fortunately, most interaction in current Internet services follows a client-server approach. Therefore, we have to deal with two situations:

- An existing client should be able to interact with a distributed object. For example, it should be possible to use current Web browsers to access hypertext documents implemented as distributed shared objects.
- A Globe distributed shared object should be able to make use of existing servers. For example, it should be possible for a Globe object to interact with an HTTP server to retrieve existing Web documents.

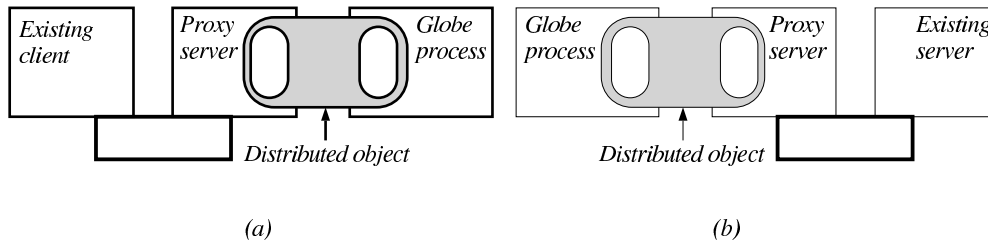


Figure 6: Combining existing clients and servers with Globe's distributed shared objects.

An evolutionary way towards integration is to make use of proxies. Figure 6a shows how an existing client is connected to a proxy server that is bound to a distributed object. The proxy server may accept HTTP requests from the client, and transform these into method invocations on the object to which it is bound. Replies from the object are returned as HTTP server responses to the client. We use a similar configuration for the situation that a Globe object has to interact with an existing server, as shown in Figure 6b. In that case, a proxy server offers an interface through which the existing server appears to be implemented as a distributed shared object. In other words, the proxy implements an object wrapper around the original server.

7 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. In particular, we need an infrastructure that

- provides a uniform model for distributed computing
- provides a flexible implementation framework
- supports development of worldwide scalable applications

Globe provides such an infrastructure. Our notion of a distributed shared object provides a uniform view on how applications are organized, and how issues such communication, replication, distribution, and migration are handled on a per-application basis. An important aspect of our model is that we provide a common framework for each distributed object. Part of this framework consists of a local composite object consisting of predefined sub-objects for handling communication and distribution. However, our replication and communication objects have only predefined *interfaces*; their *implementations* will generally be different, depending on the distribution policy needed for the distributed shared object as a whole. An implementation is completely hidden behind an object's interface. By making use of binary interfaces, Globe objects are language independent.

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[30]. 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, as well mechanisms that support large-scale applications composed of many distributed objects. At the same time, we are implementing parts of our system, primarily to validate our approach as soon as possible. We have designed and implemented communication objects that connect local objects in a single distributed object. Different implementations exist for UDP, TCP, and IP multicasting. Also, a prototype version of our location service has been implemented and is now being used for experimentation.

Finally, whether class objects are downloaded from a remote or local repository raises an interesting question on how the Globe architecture can be combined with interpretable and portable languages such as Java. In particular, we envisage that our architecture can be used as the basis for constructing worldwide scalable applications written entirely in Java. This question is subject to further research.

References

- [1] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Sebastopol, CA., 1992.
- [2] OMG. "The Common Object Request Broker: Architecture and Specification, revision 2.0." OMG Document 96.03.04, Object Management Group, Mar. 1996.
- [3] M. Satyanarayanan. "Scalable, Secure, and Highly Available Distributed File Access." *Computer*, 23(5):9–21, May 1990.
- [4] ISO. "Open Distributed Processing Reference Model - Part 3: Architecture." International Standard ISO/IEC IS 10746-3, 1995.
- [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] Microsoft Corporation. *The Component Object Model Specification, Version 0.9*, Oct. 1995.
- [8] M. van Steen, F. Hauck, and A. Tanenbaum. "A Model for Worldwide Tracking of Distributed Objects." In *Proc. TINA '96*, pp. 203–212, Heidelberg, Germany, Sept. 1996. Eurescom.
- [9] V. Cate. "Alex – A Global Filesystem." In *Proc. USENIX File Systems Workshop*, pp. 1–11. USENIX, May 1992.
- [10] D. Wessels. "Intelligent Caching for World-Wide Web Objects." In *Proc. INET '95*, Honolulu, Hawaii, June 1995. Internet Society.
- [11] J. Gwertzman and M. Seltzer. "The Case for Geographical Push-Caching." In *Proc. 5th Hot Topics in Operating Systems*, Orcas Island, WA, May 1996. IEEE.
- [12] M. Baentsch, G. Molter, and P. Sturm. "Introducing Application-level Replication and Naming into today's Web." *Computer Networks and ISDN Systems*, 28(7–11):920–930, 1996.
- [13] J. Donnelley. "WWW Media Distribution via Hopwise Reliable Multicast." *Computer Networks and ISDN Systems*, 27(6):781–788, 1995.

- [14] C. Partridge, T. Mendez, and W. Milliken. “Host Anycasting Service.” RFC 1546, Nov. 1993.
- [15] V. Sunderam. “PVM: A Framework for Parallel Distributed Computing.” *Concurrency: Practice and Experience*, 24(4):315–339, Dec. 1990.
- [16] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA., 1996.
- [17] K. Birman and R. van Renesse, (eds.). *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA., 1994.
- [18] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O’Reilly & Associates, Sebastopol, CA., 1992.
- [19] E. Levy and A. Silberschatz. “Distributed File Systems: Concepts and Examples.” *ACM Computing Surveys*, 22(4):321–375, Dec. 1990.
- [20] J. Protić, M. Tomašević, and V. Milutinović. “Distributed Shared Memory: Concepts and Systems.” *IEEE Parallel and Distributed Technology*, 4(2):63–79, Summer 1996.
- [21] N. Brown and C. Kindel. “Distributed Component Object Model Protocol – DCOM/1.0.” Internet Draft, Nov. 1996.
- [22] Microsoft Corporation. *The Component Object Model Specification, Version 0.9*, Oct. 1995.
- [23] A. Grimshaw and W. Wulf. “The Legion Vision of a Worldwide Virtual Computer.” *Communications of the ACM*, 40(1):39–45, Jan. 1997.
- [24] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. “Multimethod Communication for High-Performance Networked Computing Systems.” *Journal of Parallel and Distributed Computing*. To appear.
- [25] B. Janssen and M. Spreitzer. *ILU Reference Manual*. Xerox Corporation, May 1996.
- [26] OMG. “The Common Object Request Broker: Architecture and Specification, revision 2.0.” OMG Document 96.03.04, Object Management Group, Mar. 1996.
- [27] G. Hamilton, M. Powell, and J. Mitchell. “Subcontract: A Flexible Base for Distributed Programming.” In *Proc. 14th Symposium on Operating System Principles*, Asheville, N.C., Dec. 1993. ACM.
- [28] 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.
- [29] 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.
- [30] 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.