An Architecture for A Wide Area Distributed System

Philip Homburg,*
Maarten van Steen,
Andrew S. Tanenbaum
{philip,steen,ast}@cs.vu.nl

1 Introduction

Distributed systems provide sharing of resources and information over a computer network. A key design issue that makes these systems attractive is that all aspects related to distribution are transparent to users. Unfortunately, general-purpose wide area distributed systems that allow users to share and manage arbitrary resources in a transparent way hardly exist. In particular, they generally do not take into account the most important properties that characterize wide area systems: 1) A very large number of users and resources, 2) an inherent latency problem caused by the distance between nodes, 3) heterogeneity due to a variety of underlying operating systems and networks, and 4) involvement of multiple administrative organizations.

The research described in this paper is part of the Globe Project (Globe stands for GLobal Object Based Environment). The goal of this project is the design and implementation of a wide area distributed system that provides a convenient programming abstraction and full transparency. The main contribution of this paper is the description of a new system for distributed shared objects. In contrast to other systems, the implementation of distribution, consistency, and replication of state is completely encapsulated in a distributed shared object. This allows for object-specific solutions, and provides the right mechanism for building efficient and truly scalable systems.

2 Problems to be Solved

In this section we concentrate on four key problems that need to be solved by wide area distributed systems: a uniform model for information exchange, location and replication transparency, flexibility with respect to implementations, and scalability.

A Uniform Model Wide area applications provide users facilities for sharing and exchanging information. To that end, numerous abstract communication models have been adopted. These models generally provide low-level communication primitives. A large variety of such primitives exist: RPC, point-to-point message-passing, stream-oriented communication, distributed shared memory, etc. However, these communication models do not provide solutions to problems that are common to all wide area systems, such as replication, data migration, and concurrency control. The effect is that each application has to invent an ad hoc solution.

What is needed is a uniform model to express sharing and exchange of information. The model should be more abstract than those offered by the simple communication primitives but still be application-independent. It should incorporate aspects common to all wide area applications, and straightforward implementations.

*Postal address: Philip Homburg, Vrije Universiteit, Faculteit Wiskunde en Informatica, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

Location and Replication Transparency Another problem with current wide area applications is the lack of location transparency. In practice, names are tightly coupled to the locations of the objects they refer to. Location-dependent names make it difficult to deal with migration. If a name specifies where an object is located, migrating that object cannot be done without either changing the name or interpreting it as a forwarding pointer. Location-dependent names also cannot be used straightforwardly when referring to replicated data. Effectively, applications are forced to keep track of which objects are replicated, where the replicas are located, and how consistency between these replicas is maintained. The problem that needs to be solved is that a wide area systems should offer a naming facility that hides all aspects concerning the location of data. Users should not be concerned where an object is located, whether it can move, whether it is replicated, and if it is replicated, how consistency between replicas is maintained.

Flexibility Wide area systems require support for flexible implementations. In particular, they need to provide services with well-specified functional semantics, but with flexible consistency semantics. We need flexibility with respect to how we implement the functionality *and* the consistency of data. In particular, different implementations for the same interface may need to co-exist. These implementations all realize the same functionality, but may differ with respect to state consistency.

Scalability The most important problem which needs to be solved is that of scalability. The rapid growth of the user population on the Internet suggests that a general-purpose wide area distributed systems should eventually be capable of supporting a billion users. In order to manage the complexity of the scalability problem, we believe it is essential to first devise an architecture in which different solutions can be embedded. Such an architecture is the subject of this paper.

3 Proposed Architecture

In this section we discuss the main components of an architecture for a scalable wide area distributed system. We have adopted an object-based model. The encapsulation of state and its operations into a single object allows a clear separation between services and implementations. Aspects such as communication protocols, replication strategies, and distribution and migration of state can be completely hidden behind an object's interface. The concept of an object offers us the transparency and flexibility needed for wide area systems.

3.1 Distributed Shared Objects

Processes in our model cooperate and interact through **distributed shared objects**, objects of which the state is physically distributed and shared between processes. Changes to the state made by one process are observed by other processes. There is no a priori distinction between clients and servers.

Each object has methods for operating on its internal state. These methods are exported via interfaces. Interfaces support the separation of a (compiled) application program from object implementations. There may be several implementations for the same interface. Before a process can invoke an operation of a distributed shared object, it has to *bind* to that object (at runtime). This means that an interface and its implementation are loaded into the process' address space. Multiple processes share an object by being bound to it simultaneously.

Our model provides four features to support the separation between a program and an implementation of a distributed shared object.

1. Access to a distributed shared object is provided by a local object which represents the distributed shared object. This isolates the program that uses a distributed shared object from the implementation of the distributed shared object.

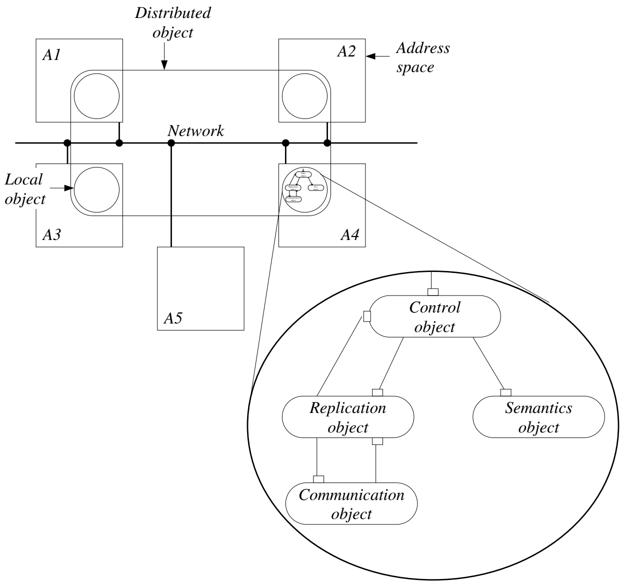

Figure 1: The structure of a Distributed Shared Object.

2. The runtime system supports downloading of new local objects into the address space of a running process. Both state and code are downloaded on demand, during execution.

3. Method invocation is done through *interface tables*, each listing one or more methods. Interface tables decouple the application from the implementation of the local object.

4. The local object which represents the distributed shared object is conceptually self-contained. This means that the implementation of a distributed shared object is completely separated from the application. The only part that the application and the object share is the layout of the interface table and the syntax and semantics of the methods in that interface table.

To simplify the implementation of new distributed shared objects we propose an implementation of local objects that consists of four local subobjects. Figure 1 shows these four subobjects: a communication object handling low-level communication with other local objects; a replication object dealing with state distribution; a semantics object providing the actual semantics of the distributed shared object; and a control object handling local concurrency control.

Communication objects provide a simple interface to send data to and receive data from other address spaces. A communication object can support connection oriented or connectionless communication and point-to-point or multicast data transfers. A replication object is responsible for keeping the state of the distributed shared object consistent. It sends marshaled copies of (parts of) the object's state, and marshaled arguments and return values of operations to other address spaces.

Communication and replication objects are generally written by system programmers and are available from libraries or class repositories. Application programmers implement the semantics objects needed for an application. A control object is generated based on the interface of the corresponding semantics object. We expect that the system comes with standard semantics objects like files and directories. Furthermore, some distributed shared objects may require replication or communication objects that are optimized to work with a specific semantics object.

A distributed shared object may replicate its state for reasons of fault-tolerance, availability, or performance. It may also partition its state, but that will not be discussed further. For each address space bound to a distributed shared object, the object decides whether to store a (full) copy of its state in that address space or not. When a process binds to an object, the object may decide to create a replica in that address space and control the consistency through update or invalidate messages. This choice optimizes the performance of read operations, which can then be executed locally. Special (server) processes can be run at strategic sites. These processes can bind to an object to create a replica. These replicas are needed to optimize read access to an object (caches) and to provide persistence and fault tolerance.

Because the semantics object is implemented by the application programmer, we would like to place as few restrictions on the implementation of the semantics object as possible. Nevertheless, we must impose some restrictions to allow a semantics object to be used in combination with a large collection of different replication objects:

- A semantics object has to be a state machine (i.e. the current state plus the arguments of a method invocation completely determine the next state of the object, and the state does not change between method invocations. A state machine is required in order to support replication objects that use active replication.
- Long-term blocking (i.e. a wait on a condition variable) is not allowed. Instead, the invocation returns an indication to the caller (the control object) that it could not continue and the control object should retry later.
- The semantics object should be able to marshal and unmarshal its state.

The control object effectively encapsulates the semantics object in a monitor. This relieves the application programmer from managing concurrency control.

3.2 Naming

Our architecture includes a naming system that maps a human readable pathname to a set of contact addresses for a distributed shared object. The contact addresses provide independent ways to bind to a distributed shared object. Typically, a replicated object provides multiple contact addresses for availability and improved performance. The set of contact addresses can change over time, and is maintained by the object itself. A replicated object typically adds contact addresses when new replicas are installed, and a mobile object creates a new contact address when it moves to a different location. At the same time, users can create and delete pathnames that refer to the object.

Figure 2a shows the most popular way of naming objects. A directory or naming service maintains a mapping from human readable object names to the address at which the object is located. This approach breaks down when an object gets multiple names and multiple addresses. This is shown in Figure 2b. In an implementation of the naming system that directly maps a pathname to a set of addresses, it is impossible to update the set of addresses efficiently.

In our architecture, object naming is split into two services. The first part, the name service, maps a pathname to an *object handle*. This object handle is passed to the second part, the location service, which maps it to the set of addresses, this is shown in the Figures 2c and 2d. This scheme allows a user to create multiple names that resolve to the same object handle, and allows the object to update the set of contact addresses that the object handle refers to. Logically, the mapping from object handle to contact addresses is stored in one place in the location service. The fact that the location service can replicate location information is hidden from the object.

An object handle consists of an object identifier and some additional information for the location service. An object identifier is worldwide unique and is never reused after the object is destroyed. Object identifiers are used during binding as an end-to-end check for the location service. The additional location information which is part of an object handle is allocated by the location service when the object registers its initial set of contact addresses.

The name service can be implemented as a worldwide hierarchical collection of directories. The feasibility of this approach has been shown by DNS. With worldwide location services there is less experience. A straightforward approach is to divide objects into different categories and use different algorithms for each category. For example, many objects will be located on a single site. These objects may have a few replicas, but the replicas are not far apart. We expect that the majority of the objects fall in this category. Traditional solutions will suffice here. A second category are highly replicated objects. For example objects that encapsulate Usenet news articles or news groups. Since replicas of such an object are stored at many sites, we expect those sites to cooperate in maintaining location

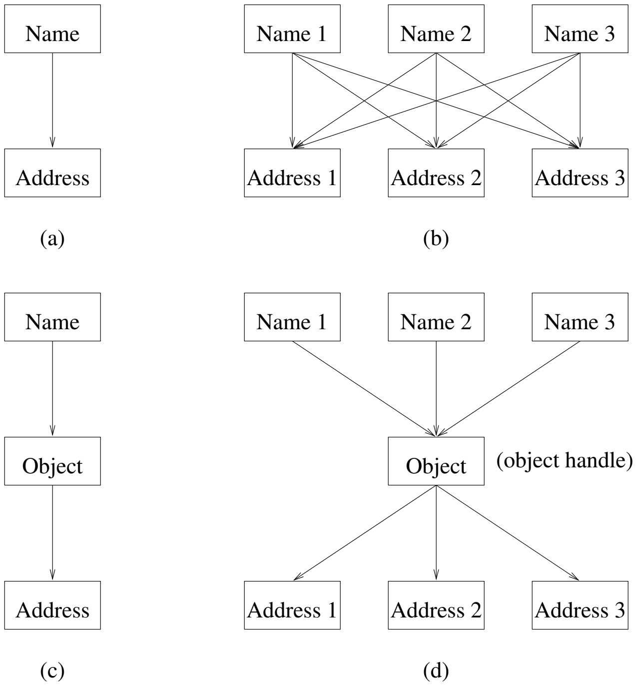

Figure 2: Two Approaches to Naming

information for the object. We also have to support mobile objects, that move from one place to another and require frequent updates of their location information.

The location information of these three categories of objects can be stored in three independent, worldwide location servers. A few bits in the object handle can be used to specify the location service used. A disadvantage of this approach is that an object can not move to a different category without changing its object handle.

We have designed a location service that is able to dynamically adapt to the migration patterns of an object. A location service provides four operations:

- Register a new object
- Unregister a deleted object
- 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 means optimizing the placement of the information needed to map an object handle to a set contact addresses such that updates to the set of contact addresses are cheap (i.e. require a small amount of (local) communication) and that lookups are also cheap. Since updates to the set of contact addresses are initiated by the object itself, it makes sense to store the mapping in a part of the location service that is near the object. Note that a replicated object is not located at a single site. Furthermore, objects that migrate require that the location of the mapping follows the object.

Our design of the location service uses a worldwide, distributed search tree. To build this search tree, we divide the world into a hierarchical set of domains. At the bottom we have one domain per site. A collection of sites form a region. This is shown in Figure 3. An object is registered at each site where it has a contact address. Furthermore,

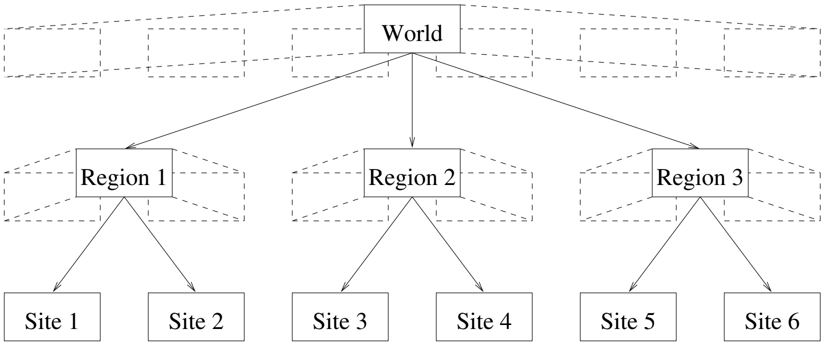

Figure 3: World Wide Search Tree

each object is registered in all regions that contain a site where the object is registered, recursively up to the top of the tree.

Initially, registrations at the site level contain the actual contact addresses and registrations at region or world level contain pointers to the next lower level. Registering at multiple levels allow 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 have to partition them. Each site stores a part of the root node and parts of the regional nodes. The partioning can be done by computing a hash value from the object handle. A more detailed description of our location service can be found in [10]

3.3 Scalability

Scalability is one of the most important requirements for any wide area system. In Section 2, we defined scalability informally as the ability to support a billion users. We can interpret that requirement in two different ways: one way is to avoid using any components (protocols or algorithms) that do not to scale. Examples include a single point of failure for the whole system, or a system wide barrier (global synchronization). Another approach is to use only algorithms of which is proven that they do scale. Unfortunately, avoiding non-scalable components does not seem to be enough: the World Wide Web is an example of a system that uses scalable components (clients, servers, TCP/IP connections, DNS) but the system as a whole has serious scalability problems. The main reason is the lack of support for replication.

To prove that a system does scale without actually building it, we can look at the asymptotic complexity of the system. A scalable system should have a cost (in terms of resources or money) of at most O(n) and a maximal performance degradation (increase of latency) of at most O(log n), where n is the number of users of the system. A cost complexity of O(n) means that adding more users to the system costs a constant amount. As the system gets larger we have to take some performance loss, and the logarithmic increase of latency caused by hierarchical structures is the best we can do.

A scalable architecture for distributed shared objects has to solve two problems: how to manage an extremely large number of objects, and how to deal with objects that have a huge number of users. The scalability of our design depends on the scalability of the distributed shared objects and of the support services. However, functionality that can not be provided in a scalable way is still useful (and often required) in smaller, more local situations. This suggests that system should not only support scalable algorithms and implementations but also non-scalable algorithms for more local use.

Distributed shared objects. The scalability issues related to distributed shared objects are critical to our architecture. The point to note is that our architecture provides the right place to encapsulate different implementations and hide them from other parts of a distributed shared object. In contrast, existing systems do not support improvements transparently. For example, the HTTP protocol as used by the World Wide Web, sets up one TCP connection per access request, and changing this requires replacing all existing WWW clients. Scalable solutions for objects with many users should be centered around hierarchies. General solutions, that is, independent of the semantics of a particular object, incorporate hierarchy as part of either communication objects or replication objects.

Support services. The organization of the name service and the location service into directory objects allows us to deal with a large number of objects. These directory objects will be hierarchically organized and effectively partition the total state of either service. In addition, each directory object is designed and implemented as a distributed shared object. The scalability of the support services is thus dependent on the scalability of distributed shared objects. For example, it can be expected that the state of the directory objects close to the root will be highly replicated and distributed, and possibly even hierarchically organized as described above. For the scalability of the location service see also [10].

4 Discussion and Related Work

In this paper we have presented a novel architecture for a wide area distributed system that provides a single, high-level interface with location and replication transparency. This interface allows efficient implementations, is flexible and does not have inherent scalability problems. We think that all related systems lack at least one of these important features.

Related projects can be found in two areas: projects that also focus on wide area distributed systems and projects that use objects to hide communication. Examples of projects and systems that focus on large-scale, or wide area distributed systems are DCE, CORBA, Globus and Legion. Distributed systems that are object based include Spring, SOS, and Network Objects.

The Globus project is developing basic software infrastructure for computations that integrate geographically distributed computational and information resources. Globus is based on Nexus[3]. Legion[5] has a similar goal, and is based on Mentat[4]. Both Globus and Legion are designed with high-performance computing in mind. As such they provide access to remote computing power. However, there is no, or only limited support for replication, migration and persistence of data.

Most related systems provide remote operations on objects or servers that encapsulate an RPC mechanism. The disadvantage of this approach that they enforce a certain implementation of the remote operations: RPC systems (like DCE) always ship the arguments of a function to a remote server. Spring[6] provides some flexibility with subcontracts but the interface presented to an application process is basically still a stub. Remote object systems like, for example CORBA[8], and Network Objects[1] provide the same functionality as an RPC based system: arguments are shipped to the (remote) object.

A system that seems at first glance similar to ours is the combination of Fragmented Objects with SSP chains[9]. Fragmented Objects, as described in [7] provide a similar model to the user of the object as our model. The internals of fragmented objects are however quite different: fragmented objects consist of fragments that communicate through connective objects. Connective objects can be communication channels or they can be fragmented objects themselves. The interface to a fragmented object is provided by a proxy object, which implements operations on the fragments object by invoking operations on the so-called group interface. Fragmented objects hide data replication and consistency management from the user of an object, but those details are exposed to the implementor of an object.

With respect to binding, the main difference with our approach is that their references are location dependent, and contain only one address. A standard RPC is used to contact the object. Furthermore, SSP chains use relative object identifiers. However, objects that use multiple contact points for fault-tolerance or efficiency reasons require that a single name may be bound to multiple contact addresses at the same time. These bindings should be easy to cache or replicate, to modify, or to pass between processes. This means that they should be stored at locations independent of the named object, and independent of processes that use them.

A distinction between semantics and replication objects is also made in [2]. The paper describes access objects, replicas and consistency managers. Access objects, replicas and consistency managers roughly correspond to our control, semantics, and replication objects, respectively. Our communication object is integrated with the consistency manager. The paper describes how to do fine-grained access control: operations that execute on different partitions of the state of the object may be executed in parallel. The interface between the access object and the consistency manager does not allow blocking operations (condition synchronization) and the operation is always executed locally. In contrast, our model does not currently support fine-grained access control but offers a greater flexibility in replication strategies and allows blocking operations.

References

[1] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. "Network Objects." In *Proc. 14th Symposium on Operating Systems Principles*, pp. 217–230, Asheville, North Carolina, December. 1993. ACM.

[2] G. Brun-Cottan and M. Makpangou. "Adaptable Replicated Objects in Distributed Environments." Technical Report 2593, INRIA, Rocquencourt (France), May. 1995.

[3] I. Foster, C. Kesselman, and S. Tuecke. "The Nexus Task-Parallel Runtime System." In *Proc. 1st Int'l Workshop on Parallel Processing*, pp. 457–462. Tata, 1994.

[4] A. S. Grimshaw. "Easy to Use Object-Oriented Parallel Programming with Mentat." *IEEE Computer*, pp. 39–51, May. 1993.

[5] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds Jr. "A Synopsis of the Legion Project." Technical Report CS-94-20, University of Virginia, June. 1994.

[6] G. Hamilton, M. Powell, and J. Mitchell. "Subcontract: A Flexible Base for Distributed Programming." In *Proc. 14th Symp. on Operating Systems Principles*, Asheville, NC, December. 1993. ACM.

[7] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. "Fragmented Objects for Distributed Abstractions." In T. L. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 170–186. IEEE Computer Society Press, 1994.

[8] Object Management Group. "The Common Object Request Broker: Architecture and Specification, version 2.0." OMG, July. 1995.

[9] M. Shapiro. "A Binding Protocol for Distributed Shared Objects." In *Proc. Int'l. Conf. on Distributed Computing Systems*, Poznan (Poland), June. 1994.

[10] M. van Steen, F.J. Hauck, and A.S. Tanenbaum. "A Model for Worldwide Tracking of Distributed Objects." In *Proc. TINA '96 Conference*, Heidelberg (Germany), September. 1996.