Communication in GLOBE: An Object-Based Worldwide Operating System

Philip Homburg, Maarten van Steen, Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam

Abstract

Current paradigms for interprocess communication are not sufficient to describe the exchange of information at an adequate level of abstraction. They are either too low-level, or their implementations cannot meet performance requirements. As an alternative, we propose distributed shared objects as a unifying concept. These objects offer user-defined operations on shared state, but allow for efficient implementations through replication and distribution of state. In contrast to other object-based models, these implementation aspects are completely hidden from applications.

1 Introduction

In the 1960s and 1970s, the computing universe was dominated by mainframes and minicomputers that ran batch and timesharing operating systems. Typical examples of these systems were OS/360 and UNIX. These system were primarily concerned with the efficient and secure sharing of the resources of a single machine among many competing users.

In the 1980s, personal computers became popular. These machines had different kinds of operating system, such as MS-DOS and Windows, and were primarily concerned with providing a good interactive environment for a single user. Resource allocation was not a major issue.

As we move closer to the year 2000, it is becoming clearer that a new environment is appearing, one dominated by millions of machines interacting over wide area networks such as the current Internet. While all these machines have individual local operating systems, the worldwide system as a whole raises many of the same issues that are found in local operating systems, including interprocess communication, naming, storage, replication, and management of files and other kinds of data, resource management, deadlock prevention, fault tolerance, and so on. Only instead of being performed on a single machine, these issues arise in the large.

In effect, we need an "operating system" for this worldwide system. This operating system will of necessity be different from existing operating systems in that it will run in user mode on top of existing operating systems (as do the servers in many modern microkernel-based operating systems). Nevertheless, it will have to perform traditional operating system functions on a huge scale.

At the Vrije Universiteit, we are developing such an operating system, GLOBE, for the worldwide computing environment of the future. In this paper we will describe some novel parts of this system, especially concerning communication.

GLOBE is based on objects in the sense that all the important items in the system are modeled as objects. Typical objects include machines, Web pages, mailboxes, News articles, etc. Associated with each object is a set of methods that authorized users can invoke without regard to the relative location of the user and the object.

GLOBE objects are unusual in that they are distributed shared objects[1, 13]. This means that processes on different machines can bind to an object even though these machines may be spread all over the world. Any process bound to the object can invoke its methods. How the objects are internally organized is up to each object's designer.

Communication in GLOBE is quite different from other systems. Most systems, especially wide-area ones, are based on messaging. Typically two processes that want to communicate first establish a (TCP or ATM) connection, and then pump bits through the connection. In other systems, a datagram-style of communication is used (e.g., UDP packets), but here, too, the basis is still message passing.

GLOBE works differently. Here processes do not send messages to communicate. Instead they communicate using the GLOBE distributed shared object mechanism. Both (or all) interested parties first bind to some common distributed shared object, and then perform operations on it. For example, to send email, a process might bind to

the recipients mailbox and then invoke an INSERT ITEM method. To read mail, the receiver would invoke a GET ITEM method. While the object would have to use physical communication internally to move the bits, the users would just deal with objects, and not with sockets, TCP connections, and other message-oriented primitives. We believe this model allows us to unify many currently distinct services. For example, GET ITEM could be used to read mail, read a News article, read a Web page, or read a remote FTP-able file in exactly the same way: in all cases, the user just invokes a method on a local object.

In addition, this model isolates programmers and users from issues such as the location and replication of data. In a communication-oriented system, users have to know how the data are replicated and where the copies are. In our model, the user of an object does not have to know how many copies there are or where they are. The object itself manages the replication transparently. To see the implications of this design, think about accessing a copy of a replicated Web page or FTP file. In both cases the user must make a conscious choice about which copy to access, since different copies have different URLs or DNS names, respectively. In GLOBE, each object has a single unique name that users deal with. Managing the replicated data is done inside the object, not outside, so users are presented with a higher level of abstraction than is currently the case. This higher level of abstraction makes it much easier to design new worldwide applications.

2 Current Paradigms for Communication

Simply encapsulating communication in objects in not enough: dealing with the wide spectrum of communication demands in complex, wide-area systems requires high-level primitives with emphasis on optimizing the ease of use of communication facilities, along with efficient use of those facilities. Realizing efficient communication requires that we look at three aspects: maximizing the bandwidth offered to an application, minimizing latencies observed by the application, and balancing the processing (CPU time) at various machines.

To effectively deal with latency, processing, and bandwidth requirements, we need a high-level description of the application's *intrinsic* communication requirements independent of network protocols, topology, etc. For example, for mailing systems we have the requirement that when a message is sent, the receiver is notified when it is delivered so that it can subsequently be read. These requirements state only that when the message is to be read, it should actually be available at the receiver's side. This means that message transfer can take place *before* notification, but possibly also later. It is not an intrinsic requirement that message transfer has taken place before notification.

In order to see how distributed shared objects can considerably alleviate current communication problems, we make the following distinction between different communication paradigms.

Synchronous data exchange. First, we distinguish paradigms centered around synchronous exchange of data. With synchronous we mean that data can only be exchanged if both the sending and receiving processes are executing at the same time. Examples include low-level data exchange based directly on TCP and UDP implementations, distributed computing based on communication libraries such as PVM [14] and MPI [10], group communication systems such as ISIS [2], and RPC-based systems like DCE [12].

Synchronous data exchange is primarily concerned with moving data from one process to one or more other processes. Naming is provided at the granularity of hosts or processes, but not individual data items or objects. The main limitation is that the data placement, replication, consistency, and persistency management are left to the application. This paradigm hides the topology of the underlying network and provides a virtual network in which every host (or process) is connected to every other host. Unfortunately, in synchronous data exchange it is hard to hide latency, and the application developer has to take explicit measures to handle it.

Predefined operations on shared state. The second class we distinguish contains paradigms centered around a fixed set of operations on shared state (often just read and write operations). Typical examples of systems that fall into this class are network file systems [5], and distributed shared memory (DSM) implementations, originating with the work on Ivy [6].

Solutions in this class generally offer a small set of low-level primitives for reading and writing *bytes*. These primitives generally do not match an application's needs. For example, in file systems data must often be explicitly marshaled, while in heterogeneous DSM systems, special measures have to be taken by the application developer (see e.g. [16]). In addition, attaining data consistency is often not that easy. For example, file systems generally offer only course-grained locks or otherwise expensive transaction mechanisms. In DSM systems, the situation can be even worse as memory consistency is often relaxed for the

sake of performance [9]. Although this does allow a reasonable transparency of replication and location of data, the application developer is confronted with a model that is much harder to understand and to deal with.

Current distributed file systems have almost no support for replication transparency, although the placement of files is generally hidden for users. However, it is mainly the limited functionality provided by file systems that poses severe problems. For example, streams for communicating continuous data such as voice and video are hardly supported.

Operations on remote shared objects. Finally, we distinguish paradigms centered around user-defined operations on remote state, such as offered by objects in Corba [11] and Spring [8], and in models such as Network Objects [3].

Solutions that fall within this paradigm implement *remote objects*, where a distinction is made between clients and servers. Clients issue requests (invoke methods), and servers implement methods and send back replies. This limits communication patterns to the asymmetrical client–server model, for example prohibiting clients to communicate directly among themselves. A disadvantage of remote objects is that every method invocation on a remote object results in the exchange of a request and a reply message between the client and the server. This problem is typically tackled by adhoc caching strategies at the client side.

Of the cited systems, Network Objects offers a pure remote object system. Corba uses request brokers to handle requests. In theory, these request brokers can hide replication and faulttolerance from the application, but general efficient solutions that do so have not yet been proposed. Spring offers subcontracts, which do provide support for transparent caching and replication, but which seem to be very limited when it comes to adaptability. For example, replication is handled by mapping an object reference to several object instances, and maintaining the mapping at the client side. This approach will never scale.

Our goal is to combine the advantages of each paradigm:

- The efficiency of implementations for synchronous data exchange.
- The transparency of actual communication as it appears through read and write operations on shared state.
- The possibility for user-defined operations on shared state as allowed in object-based systems.

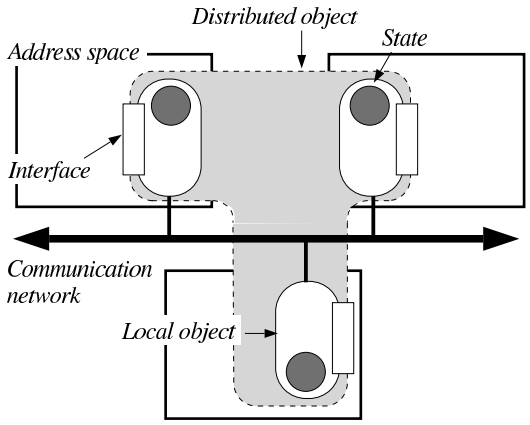

Figure 1: A distributed shared object

3 Distributed Shared Objects

A distributed shared object [4, 15] offers one or more interfaces, each consisting of a set of methods. Objects in our model are passive; client threads use objects by executing the code for their methods. Multiple processes may access the same object simultaneously. Changes to an object's state made by one process are visible to the others. An important distinction with other models is that, in our case, objects are physically distributed, meaning that active copies of an object's state can, and do, reside on multiple machines at the same time. However, 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.

Our approach makes distributed shared objects quite different from remote objects in another important way: there is no a priori distinction between clients and servers. We take the approach that processes that communicate through method invocation on the same object, are treated as equals. In particular, they are said to jointly *participate* in the implementation of that object.

In a sense, distributed objects are a collection of local objects that communicate and provide the user of the object with the illusion of shared state. This is an improvement over the remote object model because it is not restricted to a small set of predefined communication patterns.

Figure 1 shows a distributed object and its implementation. In this example, the distributed object is used in three address spaces. Each of those address spaces has a local object that participates in the distributed object. These local objects use the communication facilities of a network

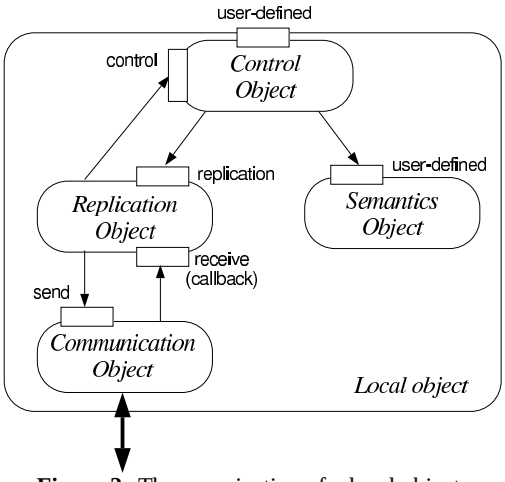

Figure 2: The organization of a local object.

to execute operations of the distributed object, and to keep the object consistent.

The implementation of local objects is separated from an application through an explicit interface table consisting of method pointers that is instantiated when the process binds to the object, but whose content may change over time. This is an important aspect of our model, as it allows us to dynamically adapt the local implementation of a distributed object, without affecting its interface to the applications that invoke its methods.

Using this approach, the implementation of a distributed object, in terms of communicating local objects, can use arbitrary communication patterns, but can also encapsulate data placement, replication, etc. In other words, the approach allows for efficient implementations of different communication paradigms. Also, because interfaces are entirely user-defined, we are not confined to a limited set of predefined operations. Our framework will thus allow us to combine the advantages of the three communication paradigms discussed in Section 2, and at the same time avoids their disadvantages.

The model described so far does not isolate the application developer from communication technology, data placement, replication, etc. The reason is that the local objects which actually implement a distributed object still have to be developed. To solve this problem we propose a standard organization for the implementation of a distributed object. This organization is shown in Figure 2.

In this architecture, the developer of a distributed object is isolated from communication, replication and consistency management by what we have called a **communication object** and a **replication object**. The developer is responsible for programming the **semantics object**, which captures the actual functionality of the distributed object. The replication and communication objects are simply selected from a library. The **control object** is responsible for handling the interaction between the semantics object and the replication object as the result of method invocation by an application. It is expected that the control object can be generated automatically, similar to the generation of RPC stubs.

This organization results in a local object that exports methods that operate on internal state. Based on the interface to the semantics object a control object is generated. The control object synchronizes access to the distributed object by serializing accesses to the semantics object to prevent race conditions and by invoking the replication object to keep the state of the distributed object consistent. The control object exports the same interface as the semantics object.

The control object implements a method invocation as three successive steps. The first step consists of invoking a *start* method at the replication object, effectively giving it control over the execution of the second step, which deals with global state operations. There are three alternatives for the second step.

- The first alternative handles remote execution. The control object passes the marshaled arguments of the method invocation to the replication object. The replication object proceeds execution according to its specific replication protocol (such as, for example, simple RPC, master/slave replication, two-phase commit, voting, etc.), effectively doing a remote method invocation. It returns the marshaled results to the control object.
- The second alternative is local execution. The control object simply invokes the corresponding method on the semantics object.
- The third alternative is active replication with a local copy. The control object provides the replication object with the marshaled arguments of the method invocation. The replication object executes the protocol to send the arguments to all replicas and to achieve synchronization with the other replicas. Next, the control object invokes the appropriate method on the semantics object.

Finally, as a third step, the control object invokes the *finish* method on the replication object. This method invocation gives the replication object the opportunity to update remote replicas.

To be practically useful, the algorithm described above has to be extended in two ways: firstly, the control object and replication object have to recognize different kinds

of operations, for example, whether operations modify the state of the object or not. Furthermore, it is necessary to distinguish operations that modify only part of the global state, which may happen in the case of partitioned or nested objects.

Secondly, some extensions are needed to deal with synchronization on conditions. Since operations on the semantics object are serialized (through locking), they are not allowed to block for a long time. Our approach is to support guarded operations: the semantics object can provide for blocking on a condition by returning status information to the control object after possibly undoing any changes made so far. The control object will suspend the execution of the operation until the next modification of the state, after which another attempt to execute the operation can be made.

A system that seems at first glance similar to ours are Fragmented Objects[7]. These objects provide a similar model to the user of the object as our model. The internals of fragmented objects are however quite different: they consist of fragments that communicate through connective objects. 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.

4 Conclusions

In this paper we have shown how distributed objects can provide a high-level interface for information sharing and exchange between processes. Separating the application from the implementation of a distributed object allows efficient implementations and dynamic adaptations to different situations. A standard architecture for implementing distributed objects isolates the object developer from data placement and replication.

References

[1] H.E. Bal and A.S. Tanenbaum. "Orca: A Language for Distributed Object-Based Programming." Technical Report IR-140, Vrije Universiteit, Amsterdam, 1987.

[2] K.P. Birman and R. van Renesse, (eds.). *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA., 1994.

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

[4] P. Homburg, L. van Doorn, M. van Steen, A. Tanenbaum, and W. de Jonge. "An Object Model for Flexible Distributed Systems." In *Proc. First ASCI Annual Conf.* , pp. 69–78, Heijen, The Netherlands, May 1995.

[5] E. Levy and A. Silberschatz. "Distributed File Systems: Concepts and Examples." *ACM Comput. Surv.*, 22(4):321–375, Dec. 1990.

[6] K. Li and P. Hudak. "Memory Cache Coherence in Shared Virtual Memory Systems." *ACM Trans. Comp. Syst.*, 7(3):321–359, Nov. 1989.

[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, Los Alamitos, CA., 1994.

[8] J. Mitchell et al. "An Overview of the Spring System." In *Proc. Compcon Spring 1994.* IEEE, Feb. 1994.

[9] D. Mosberger. "Memory Consistency Models." *Operating Systems Reviews*, 27(1):18–26, Jan. 1993.

[10] MPI Forum. "Document for a Standard Message-Passing Interface." Draft report Technical Report, University of Tennessee, Knoxville, Tennessee, Dec. 1993.

[11] OMG. "The Common Object Request Broker: Architecture and Specification, revision 2.0." OMG Document Technical Report 96.03.04, Object Management Group, Mar. 1996.

[12] OSF. "Distributed Computing Environment." OSF Document Technical Report OSF-DCE-PD-1090-4, Open Software Foundation, Cambridge, Mass., Jan. 1992.

[13] M. Shapiro. "Structure and Encapsulation in Distributed Systems: The Proxy Principle." In *Proc. Sixth Int'l Conf. on Distributed Computing Systems*, Boston, MA, May 1986. IEEE.

[14] V.S. Sunderam. "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience*, 24(4):315–339, Dec. 1990.

[15] M. van Steen, P. Homburg, L. van Doorn, A.S. Tanenbaum, and W. de Jonge. "Towards Object-based Wide Area Distributed Systems." In L.-F. Cabrera and M. Theimer, (eds.), *Proc. Fourth Int'l Workshop on Object Orientation in Operating Systems*, pp. 224–227, Lund, Sweden, Aug. 1995. IEEE.

[16] S. Zhou, M. Stumm, K. Li, and D. Wortman. "Heterogeneous Distributed Shared Memory." *IEEE Trans. Par. Distr. Syst.*, 3(5):540–545, Sept. 1992.