Distributed Shared Objects as a Communication Paradigm

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

Communication can be viewed at different levels of abstraction. At a high level, it appears as *an exchange of information* between processes. These processes are either contained in a single parallel or distributed application, or may otherwise belong to different applications that need to communicate. At a low level, communication appears as the mere *transfer of bits* from one address space to another.

Parallel and distributed applications require support for expressing communication at a high level of abstraction. Different kinds of support are currently provided.

- There are numerous communication libraries such as PVM [16], that offer straightforward message-passing facilities across networks. Although such communication facilites are generally efficient, they provide a relatively low level of abstraction making application development unnecessarily hard.
- At a more abstract level, several systems have been developed that support shared memory communication in multicomputer environments (see e.g. [8, 14]) hiding the intricacies related to network computing. The problem with these systems is that they only provide *read* and *write* operations for transferring bytes.
- More recently, much effort is being put into the development of object-based systems by which communication is expressed in terms of method invocations at (possibly) remote objects [3]. The price for raising the level of abstraction in this case, is a loss of communication performance. For many applications, this is not acceptable.

To alleviate these problems, we propose the use of **distributed shared objects** for structuring distributed systems and expressing communication. Our model uses the concept of a shared object as the means for encapsulating communication by providing user-defined operations on shared state. In contrast to most existing object-based models, we combine the concept of shared objects with the means for realizing efficient implementations through distribution and replication of state, but without violating the principles of encapsulation. Our approach offers (1) a single, high-level communication paradigm that can be used by many applications, (2) simplifies interactions between these applications, and (3) at the same time allows for efficient, and possibly application-oriented implementations of that communication.

2 Some basic requirements

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. Of these three, latency and processing are more important than bandwidth for two reasons:

- Even though bandwidth can be increased effectively (as illustrated by gigabit networks), decreasing latency is more difficult, either because of software overhead, or due to the speed of light.
- In cases where the demands of individual requests can be met easily, we often still have to deal with a large number of such requests, as is seen in object-based systems.

The importance of focusing on latency and processing is exemplified by proposals such as those for late-binding RPC [13], or those for shipping server code to clients as is the case with Java [15].

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 general, we can state that an application developer should be allowed to formulate just these requirements, and that he should be shielded from issues dealing with:

- The underlying communication technology and topology.
- Placement of data.
- Replication of data.
- Management of the consistency of the data.
- Management of the persistence of the data.

There are two main reasons for trying to isolate application programmers from having to deal with data placement, replication, and consistency. The first reason is a software engineering one: replicating data and maintaining the consistency of data is complex and the algorithms used are not application-specific. This suggests that support for replication should be provided by system software.

The second reason is that good choices for data placement, replication strategies, and consistency guarantees depend on the environment in which the application is deployed. For example, if an application is used to share information on a single LAN, it is possible to provide very strict consistency guarantees and data placement is often not an issue. However, if the application is also used on a wide area network, the situation is quite different. In that case, data placement is an issue, replication has to be used to increase availability and fault-tolerance, and due to the higher latency, it might not be practical to provide strict consistency. Consequently, the application programmer is confronted with a much harder implementation effort compared to the local area case. It may even be so that the application's semantics have to be weakened in order to be able to come to a satisfactory implementation.

In this line of thought, we can say that to benefit from specific circumstances as offered by the underlying communication network, it is important to support runtime selection of implementations for communication protocols, replication strategies, and consistency guarantees.

Finally, we need support for security, interoperability, naming, and system management. We require location-independent, worldwide, user-chosen names. This means that the control of naming information is left to the users of the system, that it is independent of the information's placement, and that it can be accessed from all over the world under the same name, provided that security rules are not violated.

3 Current paradigms for communication

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 [16] and MPI [10], group communication systems such as ISIS [1], 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. [18]). 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 hardly support 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 as needed to communicate continuous data such as voice and video is 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 [7], and in models such as Network Objects [2].

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 ad-hoc 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 fault-tolerance 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.

In the following section we introduce distributed shared objects as a way to combine these advantages.

4 Distributed shared objects as a unifying communication paradigm

4.1 The concept of a distributed shared object.

A distributed shared object [4, 17] 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.

The main advantages of distributed shared objects are:

- A distributed object provides well-defined interfaces to its users (applications). The user is isolated from the way that communication, replication, and consistency, are implemented.
- Persistence and communication are completely encapsulated in a distributed object. This means that an implementation of a distributed is not limited to a small set of communication protocols or consistency algorithms built into a runtime system.
- We allow object implementations to be loaded at runtime.
- Invocations on distributed objects are just ordinary object invocations: a process has a local implementation of the object's interface in its own address space. In other words, to a process, a distributed shared object appears the same as a local 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 to execute operations of the distributed object, and to keep the object consistent.

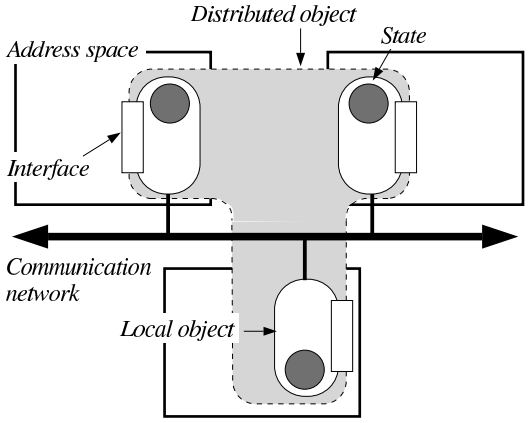

Figure 1: A distributed shared object

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 3, and at the same time avoids their disadvantages.

4.2 Making communication transparent

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 the implementation of 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.

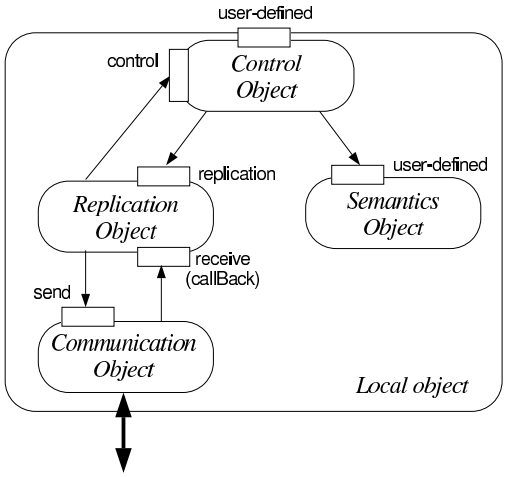

Figure 2: The organization of a local object.

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.

As we have already mentioned, the model of shared state with operations on that state leads to passive objects: activity is provided by threads running in processes. To integrate communication cleanly in this model we associate *pop-up threads* with incoming messages: when a message arrives, the communication object will create a new thread to handle the incoming message. In this new thread the communication object invokes a method on the callback interface of the replication object. The replication object calls the callback interface of the control object with the marshaled arguments of a request.

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

[2] 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.

[3] R.S. Chin and S.T. Chanson. "Distributed Object-Based Programming Systems." *ACM Computing Surveys*, 23(1):91–124, March 1991.

[4] P. Homburg, L. van Doorn, M. van Steen, A. Tanenbaum, and W. de Jonge. "An Object Model for Flexible Distributed Systems." In *Proc. 1st Annual ASCI Conference*, pp. 69–78, Heijen, The Netherlands, May 1995.

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

[6] K. Li and P. Hudak. "Memory Cache Coherence in Shared Virtual Memory Systems." *ACM Transactions on Computer Systems*, 7(3):321–359, November 1989.

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

[8] A. Mohindra and U. Ramachandran. "A Comparative Study of Distributed Shared Memory Systems." Technical Report GIT-CC-94/35, Georgia Institute of Technology, Atlanta, August 1994.

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

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

[11] Object Management Group. "The Common Object Request Broker: Architecture and Specification, version 1.2." Technical Report 93.12.43, OMG, December 1993.

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

[13] C. Partridge. *Late-Binding RPC: A Paradigm for Distributed Computation in a Gigabit Environment.* Ph.D. thesis, Harvard University, 1992.

[14] M. Stumm and S. Zhou. "Algorithms Implementing Distributed Shared Memory." *Computer*, 23(5):54–64, 1990.

[15] Sun Microsystems, Mountain View, Calif. *The Java Language Specification*, May 1995.

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

[17] 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. 4th International Workshop on Object Orientation in Operating Systems*, pp. 224–227, Lund, Sweden, August 1995. IEEE.

[18] S. Zhou, M. Stumm, K. Li, and D. Wortman. "Heterogeneous Distributed Shared Memory." *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–545, September 1992.