The Architecture of a Worldwide Distributed System

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 59.

The research reported in this thesis was supported by the Netherlands Computer Science Foundation (SION, grant 612–17–117) with financial support from the Netherlands Organization for Scientific Research (NWO).

VRIJE UNIVERSITEIT

The Architecture of a Worldwide Distributed System

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen /
Wiskunde en Informatica
op donderdag 8 maart 2001 om 13.45 uur
in het hoofdgebouw van de universiteit,
De Boelelaan 1105

door

Philippus Constantijn Homburg

geboren te Oegstgeest

Promotor:	prof.dr. A.S. Tanenbaum
Copromotor:	dr.ir. M.R. van Steen

Contents

Preface

Sometime during my second year as a student at the Vrije Universiteit, Raymond Michiels, a fellow student, and I discovered that our course schedule contained a lot of holes that could be filled with courses from the third year. After passing the exams for all the courses we took, we had a good look at the schedule for the third year and decided to combine the third and fourth year. This would allow us to get a *doctorandus* degree in just three years. Even then there was plenty of time to become “student assistant,” to assist (first year) students with their practical work. Raymond actually managed to get his degree after three years. I made a small scheduling error and had to wait a couple of months before I could take an exam in an physics course.

A couple of months before graduating, Raymond asked Andy Tanenbaum whether he could become a Ph.D. student. Andy said yes. I went to Andy a couple of weeks later but Andy said no: he was booked solid with Ph.D. students. Andy offered the helpful advise to look for positions in different groups. It was soon clear to me that Andy was the only option.

I had to stay at the VU for a couple of months anyway for the physics course and decided to write a TCP/IP implementation for Minix. Minix is a Unix-like operating system, written by Andy, to teach operating-system internals to students. Writing a TCP/IP implementation was interesting because I wanted to know whether I could complete a large and complex piece of software that runs inside an operating-system kernel.

While I was working on the TCP/IP implementation, the time came for the annual European regional finals of the ACM programming contest. Raymond, Steven Reiz, and I had participated before with mixed success. That year, Victor Allis joined us and added some much needed professionalism. We practiced much more than before and we developed a good team strategy. We won the European contest as a result. The next round was the world finals in San Antonio, Texas. We practiced a lot more, under the supervision of Matty Huntjens. In the end, we were mildly disappointed with a second place; we were so close.

After the programming contest Andy had changed his mind and I could become a Ph.D. student. At least one Ph.D. student quit during that time. Raymond Michiels quit a couple of years later; he decided that research was too theoretical and wanted to work on something more practical.

Andy allows a Ph.D. student some time to find a research topic. After some time, my topic was going to be wide-area distributed systems. This research was carried out in the context of Amoeba, a distributed system developed at the VU under Andy's supervision. However, it became soon clear to me that Amoeba was not suitable as a basis for a worldwide distributed system. I wanted to design a new system.

This way I fell into the trap that is called "the second system effect[1]." This means that a system that is to replace a successful system with some minor design errors becomes much too big, much too complicated, and contains many unnecessary features. For example, my thesis contains 392 pages whereas the thesis of Sape Mullender, which describes the design of Amoeba, contains 203 pages. On the other hand, my thesis describes a worldwide distributed system and Sape's thesis describes just an ordinary distributed system; maybe the size difference is justified.

A good example of the second system effect is the object model. Leendert van Doorn started his Ph.D. studies around the same time as I did. He was going to write an operating-system kernel, called Paramecium, that could be adapted to different applications. For example, it should be possible to make the kernel more suitable for parallel programming or for embedded systems such as televisiona sets. Both Leendert and I wanted to use objects as a basis for our systems. Andy suggested the creation of a common object model that could be used for both worldwide distributed systems and specialized operating-system kernels. Needless to say, this took a long time to develop. It is not clear if we succeeded. Leendert's thesis describes an object model that is slightly different from the one described in this thesis.

From the start Andy made clear that Leendert and I were not supposed to program until the design was finished. The way I interpreted this is that I would work on the design for forty hours a week. The rest of the time could be spent on hobbies. Some of those were unrelated to computers, such as photography, high-end stereo equipment, and making television. The computer related ones are extending Minix and breaking the security of computer systems.

Leendert, which whom I shared an office, believes that during the time as a Ph.D. student one should look at as many different subjects as possible. One of those subjects includes computer security. We spent many evenings breaking NFS, the FlexLM license manager, hack NIS+, etc. We never managed to duplicate the cracking of secure NFS: understanding the math took too much time.

Extending Minix is a separate story. When programming is almost a passion and you are told not to write software as part of the research, you find some other project. Minix was written for students and is both easy to understand and, in some areas, hard to change. For example, adding support for virtual memory was supposed to

[1]This term was coined by Fred Brooks in his book "The Mythical Man-Month."

be impossible. Porting the X11 window system was also not realistic. Finally, most people believe that asynchronous I/O is very hard to use. Needless to say, I just had to experiment with all three options (and I already had the TCP/IP implementation).

In retrospect, it is more likely that Andy meant Leendert and me to write software for our research at night and write papers about that software during the day. A similar approach was used by Robbert van Renesse, one of Andy's former Ph.D. students. This allowed him to convince Andy that his designs could be implemented because the software was already written.

During my time as a Ph.D. student, I was assisted by many people. First, I would like to thank Andy Tanenbaum and Maarten van Steen. They provided me with feedback on ideas that were often incomplete, incomprehensible, or worse. Maarten was the first person to read draft material of this thesis. He once described my writing style as a "brain dump."

Dick Grune helped me on two different occasions. During the early phase of my research he helped me with structuring my thoughts about flexible naming schemes for worldwide distributed systems. More recently, Dick was willing to read many revisions of Section 4.5.7, the section with the highest density of mathematical symbols in this thesis.

Richard Golding and Wiebren de Jonge helped with the object model in completely different ways. Richard created enough confusion about object models to get us to work on a description of the model. Later, Wiebren kept asking questions about parts of the model that were unclear. Currently, I enjoy working with Wiebren and Michel Oey on the LD project.

A disadvantage of spending a long time on writing a Ph.D. thesis, besides leading to a long preface, is that many people come and go. I would like to thank two of them. After I graduated, I shared an office with Frans Kaashoek. He showed me how to conduct research efficiently. Unfortunately, I did not manage to copy his skills. His research in communication primitives for Amoeba and my attempts to extend those to wide-area networks taught me a lot about implementations of network protocols.

Hans van Staveren was the local operating-system guru. He knew the answer to almost every question about the Unix and Amoeba operating-system kernels or behavior of the network. From Hans, it is a small step to the group of system administrators as a whole. They worked hard to maintain, as well as can possibly be done under Unix, the much sought after "single system" image. And yet, I got access to whatever servers or other system resources I wanted when I asked for it. With Kees Bot, I spent many hours and lunch breaks discussing Minix. He also cleaned up and released my code as part of Minix-vmd. The discussions with him about Minix probably kept me sane.

The work of Raoul Bhoedjang and Tim Rühl on Henri Bal's Orca system provided inspiration for the implementation of replication. I would also like to thank the people who read (parts of) this thesis: Chris Hänle, Jussi Leiwo, Guido van't Noordende, Patrick Verkaik, and the members of the reading committee: Andrew Grimshaw, Franz Hauck, Thilo Kielmann, and Marten van Sinderen.

The quality of the food in the *mensa* inspired many postdocs to organize dinner parties at their homes. I would like to thank Aline Baggio, Guillaume Pierre, Thilo Kielmann, Chris Hänle, and Jussi Leiwo for inviting me.

I do not think that my family suffered more from me during the writing of this thesis than they would have otherwise. However, I would like to thank them for their feedback on the design of the cover and for getting this thesis printed.

Chapter 1

Introduction

1.1 Goal

Worldwide computer networks have become increasingly popular over the last few decades. For example, many research and educational institutions were limited to UUCP (Unix to Unix Copy) over dialup phone connections in the early 1980s, but have been connected through high-speed Internet connections in the late 1990s. Much research is still being done to determine how worldwide computer networks should be built, and how computers should communicate using these networks. The current Internet is the result of such research. Unfortunately, the technology to actually use a wide-area network to create worldwide applications is much less mature than the (Internet) technology to build the worldwide network itself. This thesis describes an architecture that simplifies the development of applications that use a wide-area network.

For a long time, the ultimate goal of operating system software for a (wide-area) distributed system was to provide a so-called "single system image." The idea is that the users and programmers can treat the collection of computers as a single, very large, virtual computer. This requires the operating system to hide all aspects of data distribution from the users of the system. An example of such a system for a local-area network is Amoeba [Tanenbaum et al., 1990]. Amoeba uses a combination of Remote Procedure Calls (RPCs [Birrell and Nelson, 1984]) and capabilities to provide transparent access to a particular data item on a possibly remote server. Programming under Amoeba means that I/O and interprocess communication are performed using RPCs and capabilities. A distributed directory service is used to store capabilities under a user-chosen name. These capabilities are used by the Amoeba RPC system to locate the appropriate servers on the network, to identify the data item on a server that should be the target of an RPC, and to secure (control access to) those data items. The net result is that the user of the system can refer to files, processes, and peripherals (printers, tape streamers) by name independent of their actual location.

There are two problems that prevent the use of Amoeba (and similar systems such as Chorus [Rozier et al., 1988] and Mach [Accetta et al., 1986]) on a worldwide scale:

1. Hiding the location of a data item, and more generally hiding any use of communication facilities or I/O primitives, from the programmer or the user works well only in a homogeneous system with a fast network or I/O system.

   The combination of homogeneity and fast networking keeps the system more or less predictable. In a homogeneous system, it does not matter which computer is used to perform a computation, and in general the network is fast enough to ignore data placement issues.[1] However, on a worldwide scale we can expect enormous differences in computing power, ranging from small pocket computers to large supercomputers, and network bandwidth, ranging from slow dial-up connections to high-speed backbone links.

2. The second issue is one of system administration. Amoeba, Chorus, and Mach are complete operating systems. On a worldwide scale it is not practical to replace every operating system with a new one, as is done by installing Amoeba, Chorus, or Mach. Instead, the distributed services should augment existing operating systems (the various versions of UNIX, products from Microsoft, Apple, etc.).

This thesis describes the architecture of Globe, a **GL**obal **O**bject **B**ased **E**nvironment. The goal of Globe is the design and implementation of a wide-area distributed system that provides a convenient programming abstraction and sufficient transparency. The main contribution of this thesis is the description of a new distributed-object architecture. In contrast to other systems, a Globe distributed object is completely autonomous with respect to the distribution, consistency, and replication of its state. This allows for object-specific solutions, and provides the right mechanism for building efficient and truly scalable systems.

1.2 Example

To illustrate the problems with sharing data in wide-area networks that are caused by the lack of proper facilities, we look at four information exchange protocols for the Internet: electronic mail (the Simple Mail Transfer Protocol, SMTP [Postel, 1982]), electronic news (the Network News Transfer Protocol, NNTP [Kantor and Lapsley,

[1]This is the case in a general purpose distributed system. In systems used for parallel programming or distributed databases this is false.

1986]), file transfer (the File Transfer Protocol, FTP [Postel and Reynolds, 1985]), and the World Wide Web (Hypertext Transfer Protocol, HTTP [Berners-Lee et al., 1996]).

From a programmer's perspective, the Internet is based on three major communication protocols: the Internet Protocol (IP), the Transmission Control Protocol (TCP), and the Domain Name System (DNS). The Internet Protocol (IP) makes it possible to send data from one computer to any other computer on the Internet. However, this protocol does not guarantee that the data will actually arrive at the other end. In particular, IP provides a best-effort service. To get reliable communication we need TCP. TCP is layered on top of IP and provides a reliable byte stream between two processes.[2] Internally, IP uses 32-bit numbers to address computers on the Internet. DNS, the domain name system, is a distributed name service that maps host names to IP addresses and vice versa. These three protocols together provide a practical way to transfer data from one computer to another.

The four information exchange protocols (SMTP, NNTP, FTP and HTTP) have the following in common:

- They are all designed to support interoperability between different implementations.
- They handle only the transfer of data between sites, ignoring aspects related to, for example, data management, replication, or security.
- They are all designed as a relatively simple layer directly on top of the TCP/IP protocol stack.

The drawback of this approach is that enhancements to a protocol are hard to realize, as it requires worldwide consensus as well as changing many independently written implementations. In addition, although very different implementations of the same protocol can easily interoperate, this is not the case for different protocols. Integrating the various protocols is hard.

A solution toward integration adopted in the Web community, is to provide a single *user* interface that supports each protocol. However, having such a powerful Web browser solves only a very small part of the problem. In particular, the distinction between how data is accessed and manipulated remains: there is no integration of underlying *communication models*. In addition, no solution is provided for integrated data management.

Table 1.1 shows how the four protocols differ, where we distinguish naming, data distribution, data encoding, grouping of documents, authentication and security, replication of data, and self-containedness. These differences are discussed next.

[2]Actually, TCP communication is between **ports** on two computers. However, the common case is that a TCP port is used by at most one process.

	Mail	**News**	**FTP**	**WWW**
Naming	mailbox (user@domain)	newsgroup (comp.os.misc)	host + path (cs.vu.nl, /pub)	URL (http://www. cs.vu.nl/)
Distribution	Push	Push	Pull	Pull
Encoding	7 bit + Mime 8 bit extensions	7 bit + Mime	7 bit text or 8 bit binary (user has to guess)	8 bit + content type
Containers	mailbox	newsgroup	directories	none
Authentication	PEM or PGP	None (PGP)	Username + Password	Username + Password, SSL
Secrecy	PEM or PGP	None	None	SSL
Replication	None	Flooding	Caching + DNS tricks	Caching + DNS tricks
Complete	No	in theory: Yes, in practice: No	in theory: Yes, in practice: No	No

Table 1.1. Features of the four Internet protocols

Naming The mail system names *users* at *domains*. Although a mailbox is globally unique, the interpretation of the user name is left to individual mail transfer agents (MTA). SMTP defines only a mapping from the domain part of a mailbox to the machine or machines on which the MTA runs that accepts mail for that domain. The news system names *newsgroups*. Newsgroup names are valid in a particular domain (“distribution”). Names for local newsgroups are typically not unique, which causes confusion if an article is posted in several domains.[3] Articles in a single newsgroup are uniquely distinguished by their message identifier. For FTP, there is no standard at all for naming documents. The only naming convention adopted is that of the FTP server, whose name is coupled to the DNS host name on which the server is running. Finally, the World Wide Web uses Uniform Resource Locators (URLs) for naming Web pages and other documents. Its distinguishing feature is that a protocol identifier is part of a URL, allowing documents available under HTTP, FTP, mail, and news, to be named within a single naming scheme.

Data Distribution With respect to distribution of data, both mail and news actively *push* data toward the destination, which is a mailbox, or a peer news server, respectively. On the other hand, for FTP and Web documents, the user is expected to explicitly *pull* the document from a server or a cache to its own location.

[3]For example, an article posted in the newsgroup vu.general at the Vrije Universiteit ended up in the newsgroup vu.general at the Vancouver University.

Data Encoding Data encoding also differs among the various protocols. Mail and news allow 7-bit ASCII data transfers. MIME (Multipurpose Internet Mail Extensions [Freed and Borenstein, 1996]) extends this by defining standard encodings for binary and 8-bit text. Extensions to SMTP allow passing 8-bit data directly from an SMTP client to an SMTP server if they both implement the extension. In practice, almost all mail and news transport agents allow 8-bit text encodings such as Latin-1. FTP supports both text files and data files. Unfortunately, the user has to guess the correct type, and explicitly configure either a binary or ASCII transfer. HTTP supports 8-bit binary data, and provides the recipient with a content type that describes the data.

Document Containers Again, there is much difference in the way that documents are collected and organized. mail, news, and FTP support different concepts of a directory in which documents are collected. Mail messages are collected in mailboxes; news articles are collected into newsgroups; and FTP supports the traditional concept of hierarchically organized directories, and, in contrast to mail and news, provides the user with full support for manipulating directories. The World Wide Web does not support document containers at all. It is only possible to have references to other documents contained in a Web page, but there is no support for collecting pages into something analogous to a directory.

Authentication and Secrecy Security is primarily supported only in news and mail. PEM (Privacy Enhancement for Internet Electronic Mail [Linn, 1993]) is a standard protocol for authentication and secrecy of mail messages. In practice, PGP (Pretty Good Privacy) is commonly used to authenticate both mail messages and news articles, and to encrypt mail messages. Authentication for FTP and HTTP is limited to simple user name/password combinations. Non-standard security extensions exist for Web documents, which are specific to Web browsers, such as SSL for Netscape™.

Data Replication The only system for which replication is defined is news. News articles are replicated using a flooding protocol; mail messages are not replicated. If a message has multiple recipients (e.g. a message sent a mailing list), copies are sent to each recipient separately. FTP and HTTP do not support replication, but some replication of data is done manually. Entire FTP and HTTP servers can be replicated, and registered under a single DNS name using round-robin DNS. In the case of Web documents, caches are typically installed by users to avoid fetching a document multiple times. This caching scheme works relatively well in practice, despite the lack of cache invalidation when documents are changed.

Completeness None of the four services is really self-contained, although the news system comes close. NNTP supports both posting and reading news articles. Furthermore, news articles may contain control fields to create and delete newsgroups.

Unfortunately, the usefulness of these control messages is limited by the lack of authentication of news articles. There is, however, no standard protocol for introducing a new news server to neighboring servers.

Initially, the mail system defined only a protocol for *sending* mail to a mailbox, but lacked a protocol for *reading* mail. POP (Post Office Protocol [Myers and Rose, 1996]) was introduced to solve this problem. However, there are no standard protocols for creating or deleting mailboxes.

FTP provides a fairly complete set of commands for maintaining a file system. For various reasons these commands are typically not used, and instead the underlying local file system is modified directly.

Finally, HTTP is also not self-contained. For example, it relies on an existing file system, and has no commands for creating directories. Also, it has only limited commands for adding new documents to an existing directory.

Conclusion Many differences between the four services are historical. This is the case for naming, the 7-bit data limitation for mail and news, and for the lack of authentication. Other differences are part of the underlying model. All four services are described in terms of *transfer* protocols. And indeed, this makes the service incomplete and often highly inflexible. For example, data management is lacking and where applicable, replication strategies are hard-wired into a protocol.

Rather than integrating the services directly, or choosing ad-hoc solutions such as adopted in Web browsers, we advocate the creation of a software layer on which data sharing services in large-scale internets are to be based. This layer is to be created on top of existing operating system kernels (which we cannot modify) and below applications. Because of its place (between the kernel and the application), such a layer of software is often called middleware. A single paradigm should provide us with the means to properly integrate the four services described above.

1.3 Requirements

An important goal of a distributed system is to support sharing, exchange, and management of resources and information such that implementation aspects are transparent to its users.

A worldwide network basically provides unreliable message-passing between any two machines on the network. The Internet shows that such a worldwide network can be built, and that unreliable message passing mechanism can be used for reliable end-to-end communication between processes (TCP).

For a distributed system to be useful on a worldwide scale, it has to be (1) scalable, (2) flexible, (3) and secure. Scalability is necessary to accommodate the large number of users, machines and data items that can be expected in a worldwide system. In addition to the scalability problem due to the number of users, a worldwide system also has to support geographical scalability. Compared to local-area systems, a wide-area system has to handle an increase in latency because signals have to travel over larger distances. Furthermore, network bandwidth over a long distance is in general more expensive and less reliable compared to bandwidth in a local-area network.

A worldwide system should last a long time. During this time the system has to be flexible enough to adapt to new technologies, new demands, etc. Furthermore, information distributed through, and stored in a worldwide system is useful only when it can be trusted. Secrecy and access control is necessary to ensure that the information can be stored safely in the system in the first place.

Beyond the three constraints describe above, we would like a system that has the following properties:

- A uniform (programming) model with support for application programmers
- Location and replication transparency
- Autonomous administration
- Interoperability between independently developed components
- Backward compatibility with existing (legacy) systems

Scalability The most important problem that needs to be solved is that of scalability [Neuman, 1994]. The rapid growth of the user population on the Internet suggests that a general-purpose wide-area distributed system should eventually be capable of supporting a billion users. 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.

We define the scalability of a design or architecture as the ability to support smaller and larger implementations of the design with adequate performance. Smaller and larger implementations can be distinguished based on the number of objects or users they support or based on the kind of network (local area or wide area).

This definition makes it possible to verify whether a design is scalable. Create both a small system and a large system that implement the design and compare the performance of the small version with other small systems that provide the same functionality and do the same for the large version. The design is scalable if the performance of both the small and the large implementations of the design is reasonable (i.e., comparable to the systems). For example, a scalable design of a network file system can be compared to NFS [Nowicki, 1989] or products from Microsoft and Novell for the small-scale version and can be compared to AFS [Howard et al., 1988] and possibly FTP for the large-scale version.

A scalable system should not depend on centralized resources or on algorithms that need global information. For some techniques, the resource usage grows too fast to be used in scalable designs. For example, the use of broadcasting and multicasting to locate resources (hosts, objects, services, etc.) on a local Ethernet can be quite effective but should probably be avoided on a worldwide scale.

However, functionality that can not be provided in a scalable way is still useful (and often required) in smaller, more local situations. For example, strong consistency is possible if the network latency is low enough and the number of users of an object is small. However, in a large system with a high network latency and many users sharing a single object, strong consistency leads in many cases to unacceptable performance. Less consistent implementations often work better in large systems.

This suggests that a system should support not only scalable algorithms and implementations but also nonscalable algorithms for local use. We argue that a scalability requirement is also a flexibility requirement. There is also an economic argument for functionality that can be provided only by nonscalable implementations: if providing such functionality at a larger scale is more costly and those costs can be paid by the users, then the law of supply-and-demand suggests that a break-even point exists where the number of customers willing to pay the required amount is equal to the number of users that can be supported. For example, the phone companies solve a geographical scalability problem by making long-distance and international phone calls more expensive than local calls.

Flexibility We can distinguish two kinds of flexibility that are needed in a large system. The first one is the ability to introduce new services, and to extend existing services without disrupting the system. For example, most Internet protocols use version numbers, protocol numbers, and port numbers to support new protocols that co-exist with old protocols.

The second kind of flexibility is support for completely different implementations of certain functionality. This is required for scalability, but also to allow the user of the system to make certain trade-offs. For example, a user may wish to reduce the time to perform certain operations at the expense of the accuracy of the results.

The second kind of flexibility is hardly supported on the Internet. For example, there are a number of Internet routing protocols. All those protocols perform basically the same function: the dissemination of routing information. There is however no overall architecture that describes how these routing protocols should be used together. This means that multiprotocol routers designed by different people may differ in their ability to support new routing protocols. Widespread use of a router with an inflexible architecture may hamper the introduction of new routing protocols.

Security The basic security functions are secrecy and integrity. A secure system should prevent unauthorized parties from reading data (secrecy) and from modifying data (integrity). The distinction between authorized and unauthorized parties is based on authentication and access control information. An important issue with respect to security in a worldwide system is to standardize authentication. Systems where many applications have there own independent way of authentication often cause the users of the system to circumvent those security mechanisms. For example, it is possible that applications for remote login, remote file access, and remote mail access all require the user to enter his password. It is tempting to add functionality to those applications to allow the password to be stored in a file on the user's computer. This relieves the user from typing his password multiple times but provides other people (co-workers or family members) with easy access to the user's files.

Uniform model and support for application programmers Wide-area systems provide users with 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.

Dealing with the wide spectrum of communication demands in complex, wide-area systems requires high-level operations with an 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 processing demands of individual requests can be met easily, we often still have to deal with a large number of such requests.

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.

In effect, 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 that 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.

Wide-area distributed systems have to deal with a variety in services, functionality, and interfaces of underlying operating systems and networks. At the same time they should hide these differences for their clients (i.e., users or applications), leading to different implementations for similar components.

Hiding the heterogeneity of the environment is a major requirement for any distributed application, but is a particularly hard one to meet in the wide-area case. The problem can be alleviated by adopting a common middleware layer [Bernstein, 1996]. Such a layer presents a single interface to the underlying system, hiding differences with respect to operating systems and networks.

Although this approach is attractive, there is a danger that the interface to the middleware and its implementation is simply too general to be useful for applications. In particular, it should be avoided that policies are hard-wired into the middleware rather than that they can be adapted per application. A middleware layer should therefore offer a *framework* for implementing applications.

A question that needs to be answered then is what this framework should look like. To this end, several object-based models have been proposed. Implementations of these models allow application developers to describe interfaces and objects. They provide support for placing objects in a heterogeneous environment and offer means for transparently invoking methods in that environment. Additional support may be provided for storing and retrieving objects, object creation and destruction, etc.

Location and replication transparency A problem with many wide-area systems 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. A wide-area system should offer a naming facility that hides all aspects concerning the location of data. Users should not be concerned where a data item is located, whether it can move, whether it is replicated, and if it is replicated, how consistency between replicas is maintained.

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 implementations of replication algorithms are often not application-specific. This suggests that a collection of replication protocols 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 in 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, may even be so that the application's semantics have to be weakened to come to a satisfactory implementation.

Going one step further, we can envision a situation in which a new replication protocol has to be used in an application that is already widely deployed. For example, a new replication protocol may have been discovered or the environment in which the application runs has changed. 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 run-time selection of implementations for communication protocols, replication strategies, and consistency guarantees.

Backward compatibility and interoperability In some cases, a new system can be introduced without the need to be backward compatible with an existing system. For example, many sites that are connected to the Internet did not previously have any wide-area network connection, so backward compatibility at the network level was not an issue. On the other hand, the Internet mail and news system had to connect to the existing UUCP network as well as various other mail systems. This required the mail transport agents on the Internet to understand UUCP style addresses. With a sufficiently flexible system, backward compatibility is usually no problem.

The disadvantage of a flexible system is that ensuring interoperability is much harder. A flexible system allows independently developed components to be tailored to specific needs. This freedom works against interoperability between those components.

Autonomous administration A worldwide system requires the cooperation of many organizations. A system that allows autonomous administration allows those organizations to operate more or less independently. The alternative, where large parts of the network are centrally controlled, or where different parties need to trust each other tends to limit the growth of the system.

1.4 Limitations of Current Systems

The four example protocols in Section 1.2 (SMTP, NNTP, FTP and HTTP) are built using a single paradigm: explicit message passing. Message passing is just one of the paradigms used to build distributed systems. Other paradigms are based on the shared memory abstraction or on remote objects. Each of these paradigms has advantages, but they also have serious disadvantages. Note that this section does not provide an overview of the current state of the art. See Chapter 6 for an overview of related work.

The paradigms can be described as follows:

- **Synchronous data exchange.**

  With synchronous we mean that data can be exchanged only 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 [Sunderam, 1990] and MPI [MPI Forum, 1995], group communication systems such as ISIS [Birman, 1994], and RPC-based systems like DCE [OSF, 1992].

  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 persistence 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.

  Advantages of low-level message passing systems, such as UDP and to some extent TCP, are that the programmer can choose between many different communication protocols. This allows a programmer to select the appropriate replication protocol, group communication protocol, distributed checkpointing algorithm, etc., for his application. Furthermore, some message-passing protocols can be implemented almost directly on top of the hardware, which leads to primitives with low overhead.

  Unfortunately, with synchronous data exchange primitives, it is hard to hide latency and the application developer has to take explicit measures to handle it. Another disadvantage is that the use of these communication primitives leads to applications with hard-wired communication patterns. If the original programmer selects the wrong replication protocol or if the environment changes, it is very hard to adapt the application.

- **Predefined operations on shared state.**

  Typical examples of systems that fall into this category are network file systems [Levy and Silberschatz, 1990] (NFS [Nowicki, 1989], AFS [Howard et al., 1988] and Alex [Cate, 1992]), and distributed shared-memory (DSM) implementations, originating with the work on Ivy [Li and Hudak, 1989].

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. [Zhou et al., 1992]). 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 [Mosberger, 1993]. 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 from 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.**

  These paradigms are centered around user-defined operations on remote state, such as offered by objects in CORBA [Object Management Group, 1999], Spring [Mitchell et al., 1994], and Network Objects [Birrell et al., 1993b].

  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 systems cited, Network Objects simply offers a remote-object system. Clients in one address space can invoke methods on an object in another address space. CORBA does not specify how exactly a request is forwarded over the network. Instead, a part of the run-time system of a CORBA implementation, which is called the object request broker (ORB), handles method invocations. 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 provides another mechanism to extend the remote-object model. This mechanism, called subcontracts, provides support for transparent caching and replication, but which seems 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 does not scale.

Conclusion Current paradigms for interprocess communication are not sufficient to describe the exchange of information at an adequate level of abstraction. The main drawbacks can be summarized as follows:

- Inflexible once deployed (message passing)
- Cannot capture application semantics (file systems)
- Rigid communication patterns (simple object models)

Our goal is to combine the advantages of each paradigm:

- The efficiency (low overhead) of low-level message passing primitives.
- 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.

1.5 Approach Taken in this Thesis

We propose an object-based approach, in which important items in the system are modeled as objects. Typical objects include Web pages, mailboxes, news articles, files, machines, etc. Associated with each object is a set of methods that authorized users can invoke regardless the relative location of the user and the object. In particular, a special kind of object, which we call a **distributed shared object**, is be used to achieve transparency in wide-area distributed systems.

In this architecture, applications communicate by invoking methods on distributed shared objects they share. Changes to a distributed shared 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 copies of an object's state can, and do, reside on multiple machines at the same time.

Distributed shared objects are constructed using a second kind of object, which we call a local object. A **local object** is an object that exists in only one process. Local objects are not only the building blocks for distributed shared objects, they also provide the application interface to the operating system kernel.

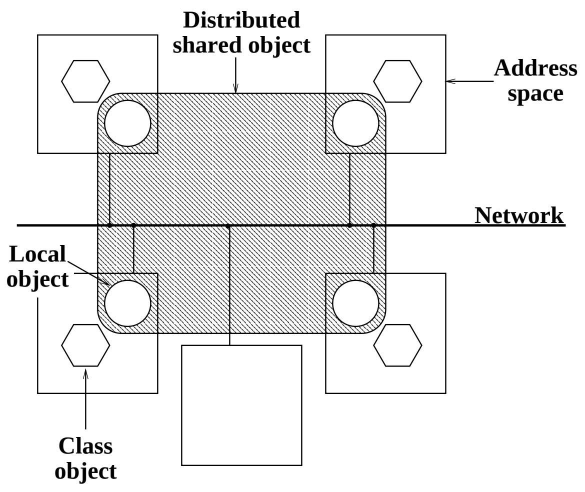


Figure 1.1. Topology of a distributed shared object

We describe many aspects of our architecture as operations on address spaces. An **address space** is a segment of (virtual) memory. Address spaces are part of processes and of operating system kernels. It is possible that processes contain multiple address spaces, for example parallel or distributed processes. We assume that processes in different address spaces interact only through explicit communication primitives.

Distributed shared objects are implemented as a collection of local objects that communicate and provide the user of the object with the illusion of shared state. This is shown in Figure 1.1. Four processes share a distributed shared object. Each address space contains two local objects: a class object and an instance of the class object. These instances communicate over the network to keep the state of the distributed shared object consistent. A local object that is part of a distributed shared object is called a **representative** of the distributed shared object.

The distributed shared object can use any communication protocol, replication protocol, etc. as long as the right local objects are available that implement these protocols. This is an improvement over the remote object model because it is not restricted to a small set of predefined communication patterns. Furthermore, there is no *a priori* distinction between client and server roles within a distributed shared object. We take the approach that processes that communicate through method invocation on the same distributed shared object are treated as equals. In particular, they are said to jointly *participate* in the implementation of that object.

Both local objects and distributed shared objects offer one or more interfaces, each consisting of a set of methods. At run time, the interfaces are implemented in the form of interface tables. An interface table is a table with one entry per method. Objects in our model are passive; client threads use objects by executing the code for their methods.

The main advantages of distributed shared objects are:

- A distributed shared 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 shared object. This means that an implementation of a distributed shared object is not limited to a small set of communication protocols or consistency algorithms built into a run-time system.
- Object implementations can be loaded at run time.
- Invocations on distributed shared objects are just ordinary method 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.

Before a process can use a distributed shared object it has to *bind* to the distributed shared object. Objects have names that are used to bind it to a running process (address space). During the bind process, code and data needed to support the object is loaded into the process' address space.

The code that is loaded during the bind process is a new class object that is initialized with marshaled state (the code) stored on a class repository. **Class objects** are local objects that contain the implementations of the methods of other local objects. This class object is used to instantiate a new local object. The bind process maps a name to a class identifier and a set of network addresses that can be used by the local object to connect to the distributed shared object. The class identifier is used to load a class object which is then used to create a new instance: the local object. Next, the local object is initialized with the set of network addresses.

Example Distributed shared objects can be used to unify the four Internet services from Section 1.2. We start by making a distinction between document objects that have the actual data, and container objects which are analogous to directories. A **document object** is suitable for containing information that is now encoded into separate documents for mail, news, FTP, and WWW, respectively. Document objects are collected into container objects. A **container object** contains references to document objects, but also to other container objects allowing for a graph-based organization.

We use three interfaces to access document and container objects. The main interface of a document object is like a simple file interface with read and write methods. Container objects provide a standard naming interface with list, lookup, link and unlink methods. Finally, we have a metadata interface which provides access to (attribute,value)-pairs. Table 1.2 lists the methods of the naming, file, and metadata interfaces.

Naming Interface	
list	return a list of directory entries
lookup	look up a directory entry, return the object handle
add	add a directory entry
delete	delete a directory entry

File Interface	
read	read the contents of a document
write	change the contents of a document

Metadata Interface	
setattr	set a (name, value) pair
getattr	look up a name
listattrs	return a list of attributed

Table 1.2. Main interfaces to document and container objects

Document and container objects can be used to implement the four Internet services as follows. Mail messages and News articles are implemented as document objects, whereas the metadata interface can be used to store header information (such as date, subject, sender, MIME-version, content-type, etc.). Mapping mailboxes and newsgroups to container objects is somewhat harder, as messages and articles are typically not named. There are two solutions to this problem: messages and articles are entered under an arbitrary name, for example their ID, or the implementation of a container object can ignore the name argument altogether and simply number messages in a mailbox and articles in a newsgroup.

Files and directories on an FTP server map quite easily to document objects with a file interface, and container objects with a directory interface, respectively. The World Wide Web consists of pages that are linked together through hyperlinks, and there is no concept of a directory. There are two ways to map WWW pages. The first one is to store a WWW page in a document object, use a container object to structure the name space, and treat hyperlinks as symbolic links. This means that the name of the file containing the WWW page is used as a link. This corresponds with current practice. A disadvantage of symbolic links is that the target of the link can be deleted or renamed without updating the link. The alternative is to implement a WWW page as an object with both a file and a directory interface. The directory can contain “hard” links to related documents, and the document itself contains references to its directory entries. In our model, hard links are object handles which do not change when an object is renamed.

Our model supports the separation of application programs from object implementations. A program and an object implementation have only an interface in common. This allows objects with new implementations to be used by old applications without recompilation provided that the new implementation supports the interface that the application expects.

To implement container and document objects for the four services we need support for different replication algorithms. In Chapter 3, we show how implementations of replication algorithms can be stored in what we call replication objects. Examples of replication objects are the following:

- The client/server replication object is the simplest replication object. Actually, such a replication object provides no replication at all. The distributed shared object’s state is stored in the address space of a server and replication objects from other address spaces merely forward requests to the server’s address space.

- A simple extension to the client/server replication object is the primary-backup approach. The state of the distributed object is replicated and one copy is designated (or elected) as the primary copy. All operations on the distributed object are forwarded to the primary. The primary executes the method and updates all other backup copies before sending a reply to the requesting address space. This scheme provides increased fault tolerance and availability over client/server replication.

- Unique to Mail and News systems is that messages and articles are immutable: they do not change after they are sent or posted. Replication of immutable objects is trivial: each address space that needs to access the object gets a copy of its state. Since the object cannot be changed, there is no need for a consistency protocol.

- To be able to support mutable objects with a large number of replicas, we can use active replication where changes to the object are flooded. In this scheme, each replication object maintains a network connection to a few neighboring replication objects. Local changes to the object’s state are then also forwarded to its neighbors.

- To improve the performance of accessing WWW and FTP documents, we introduce push/pull replication. Push/pull replication combines a replication scheme for a small number of copies (such as primary/backup replication) with caches.

Changes to the state of the object are actively pushed to a relatively small number of replicas. Processes that use the distributed object fetch a copy of the state of the object through cache servers. Caches maintain their copy of the state consistent by discarding copies after some time, and by polling authoritative replicas of the object.

The appropriate replication object depends on the purpose of a distributed shared object and on the environment in which the object is used. For example, in general, a container object for a newsgroup is expected to use flooding. However, a very small newsgroup for discussing a highly specialized subject may use a primary/backup protocol or even a client/server approach. For WWW documents that are accessed seldomly, a client/server approach may be appropriate. For popular WWW documents, the use of caches is needed to distribute the load.

1.6 Outline of this Thesis

The next three chapters contain a detailed description of our architecture. The description is split into three parts: local objects, the construction of distributed shared objects, and naming. Chapter 2 describes the local object model and the basics of distributed shared objects. Distributed shared objects can be constructed in different ways. Chapter 3 describes a structured way of constructing distributed shared objects. This chapter describes replication, persistence, and security.

To be able to access distributed shared objects, it is necessary to have a naming and binding mechanism. Naming and binding are described in Chapter 4. Related to naming is garbage collection. Objects are garbage collected when they are no longer referenced by other parts of the system, in particular the naming system. For this reason, Chapter 4 also presents a distributed garbage-collection algorithm.

A prototype implementation is described in Chapter 5 to show that distributed shared objects can be implemented. An overview of related work is presented in Chapter 6. Finally, Chapter 7 presents the conclusions.

Chapter 2

The Object Model

2.1 Introduction

In this chapter, we will discuss our object model. An object combines some amount of data, the state of the object, with operations on the state. Objects with the same operations but different state are said to belong to the same **class**. The collection of operations (and the semantics of those operations) that are supported by an object is called the object's **interface**. The main advantage of an object-based system is that it is possible to hide the details of data structures and algorithms. This means that is does not matter what data structures an object uses internally, as long as the object provides the right interface.

For a worldwide distributed system, it is necessary to be able to use different implementations for the same interface. With an object-based approach it is possible to provide encapsulation of state and its operations to allow a clear separation between services and implementations of those services. Implementation 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.

The purpose of a wide-area network is to exchange information between different computers on the network. In our model, information is exchanged using distributed shared objects. Changes to the state made by one process can be observed by other processes. Figure 2.1 shows two address spaces that share an object. Communication between processes (in different address spaces) takes place by invoking methods on shared objects.

Objects can be divided into two categories: local objects and distributed shared objects. **Local objects** are objects that are restricted to one address space. This in contrast to distributed shared objects that can span multiple address spaces. Distributed

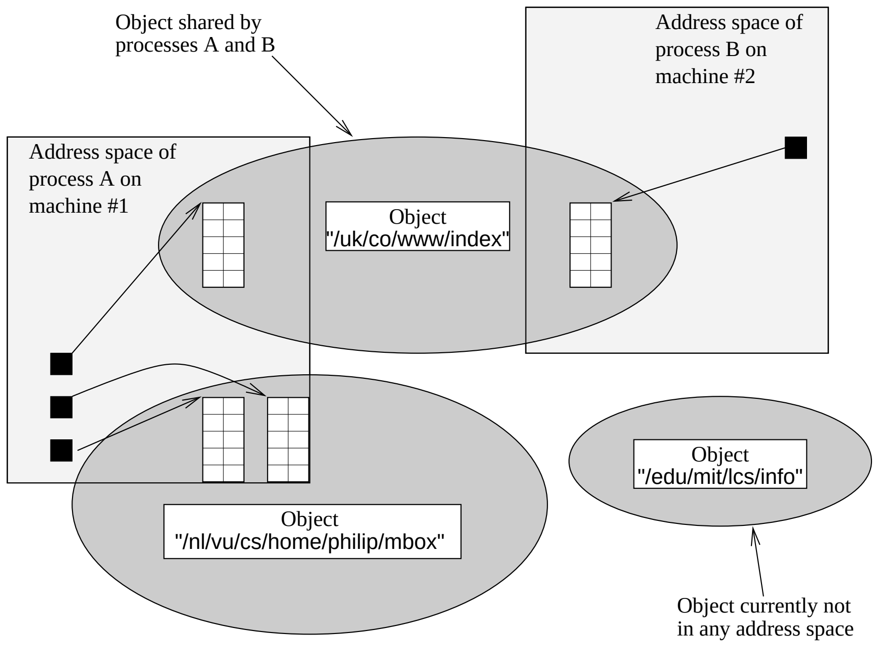

Object shared by processes A and B
Address space of process B on machine #2
Address space of process A on machine #1
Object "/uk/co/www/index"
Object "/edu/mit/lcs/info"
Object "/nl/vu/cs/home/philip/mbox"
Object currently not in any address space

Figure 2.1. Sharing objects

shared objects are built out of local objects, as is shown in Figure 2.2. Local objects are layered on top of the actual hardware and the local operating system.

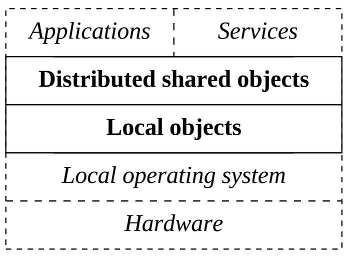


Figure 2.2. Software layers

Before a process can invoke an object's method, it has to **bind** to the object. Binding involves installing and initializing one or more local objects in the process' address space. These local objects offer an implementation of the interface used by the process. In contrast to many other object models which essentially follow the client/server approach, we support a *symmetric* model of computation. Once a process is bound to an object, it forms part of the object's overall implementation. We therefore say that a process *participates* in the implementation of a distributed shared object.

Processes can communicate by binding to a common distributed shared object. Sharing a distributed shared object can be done concurrently or sequentially. For example, a video stream in a multimedia conference can be implemented as an object that is concurrently shared between a sender (camera) and one or more receivers. Alternatively, we can store a video stream on disk (a movie) to read and display it later. In this case, the object is shared sequentially.

In our model, an object logically consists of three parts:

- The object's current "value" or **state**.
- The object's **methods**. These methods are the only way to inspect or modify an object's state.
- A collection of **interfaces**. An interface is a collection of pointers to methods implementations. Methods can only be invoked via interfaces.

Interfaces are represented at runtime as **interface tables**. An **interface table** is an array of (method pointer, state pointer) pairs. Multiple copies of the state pointer are stored in an interface table to support composite objects (objects that are an aggregation of one or more subobjects). The method pointer points to the start of the executable code that implements that particular method, and the state pointer points to the state of the object on which the operation should be invoked. This scheme allows multiple objects to share the same method implementation, and also gives some freedom as to where a particular method implementation is stored.

Objects in our model are passive. A **passive object** relies on an external source of activity (such as a thread) to execute its methods. In contrast, an **active object** typically contains a thread that executes code to accept method invocations and to execute them. In our model, activity is provided by processes that invoke an object's method, and through pop-up threads that handle, for example, incoming messages that arrive over a network.

Classes have runtime existence in the form of **class objects**. The object model supports dynamic configuration of applications by loading class objects at run time from an object repository. An application loads implementations for any distributed shared objects it tries to access. This dynamic loading is supported by a flexible object naming system.

2.2 Local Objects

Local objects are the basic building blocks of the object model. They are used for three purposes: (1) as a basis for the construction of distributed shared objects, (2) as a platform-independent operating system interface, and (3) by applications to store their (private) data structures.[1]

A local object may belong to zero or more of the following overlapping object categories:

1. Composite objects: objects that consist of subobjects
2. Class objects: objects that create other objects and usually contain the method implementations of their instances.
3. Built-in objects: objects that are part of the run-time system (in contrast to objects that are created or loaded at run time)
4. System objects: objects that provide an interface to operating system functionality

An object that does not fall into any of those categories is just an ordinary instance of a class. An object which belongs to four classes, a composite, built-in, system, class object is theoretically possible, but not used in practice. Composite, class and system objects will be discussed in the Sections 2.2.4, 2.2.3 and 2.2.5 respectively.

The main requirements for the design of local objects are:

- Relatively light weight

  The overhead of a local object should be low enough that many user-defined data structures can be implemented as objects.

- Language and operating system independent, binary interfaces

  A strict binary interface for an object allows object implementations to be used on all platforms that use the same processor, independent of the programming language used to generate them, or the operating system that is used. In contrast, source code compatibility, such as used by CORBA and POSIX, requires recompilation on different platforms.

- Run-time existence for classes and interfaces.

Run-time existence of classes and interfaces is required for flexibility reasons. The flexibility of local objects is very important to allow the system to evolve and to adapt to different environments. Our model for local objects makes it possible for programmers to deal with flexibility in a controlled way.

To support adding new object implementations (classes) to a running process, we need to be able to identify those implementations and have to ensure type compatibility between the type of the loaded implementation and what is expected by the running

[1]The local object model has been developed together with L. van Doorn and with input from R. Golding and W. de Jonge. Van Doorn uses this object model to implement an extensible operating system [van Doorn et al., 1999].

process. Identifying implementations can be simplified by storing executable code in (class) objects, as will be discussed in Section 2.2.3.

As a comparison, in C++, classes do not have explicit run-time existence. Instead, the compiler translates operations to create new objects, as well as method invocations on an object, into ordinary function calls. This approach makes it impossible to add new object implementations at run-time. To avoid calling specific functions directly, the C++ programmer can specify methods to be *virtual*. A pointer to the implementation of a virtual method is stored in the object. This allows different objects to have different implementations of the same method. Although this provides some support for loading new object implementations at run-time, it requires the programmer to declare all methods as virtual methods. No support is provided for identifying implementations, and, worst of all, no support is available for ensuring type compatibility.

Some programming languages are more flexible than others. In Pascal, all data structures used by a program have to be known (defined) when the program is compiled. Furthermore, the definition of a new procedure requires the programmer to specify the number and type of the arguments. In contrast, variables in a Python program are not typed and can refer to any kind of object. Methods of an object are stored in an associative array which can be changed at run-time.[2] Type compatibility is checked at run-time when a method is invoked. An exception is raised if the method is not in the object's dictionary, or if the method is called with the wrong number of arguments. This approach allows very flexible systems, but makes static analysis of the system almost impossible.

2.2.1 Interfaces

An **interface** is simply an ordered collection (list) of methods. An example of an interface for a lock object is shown in Figure 2.3. This mutex interface contains four methods: lock, unlock, read, and await. These methods respectively lock, or unlock the object, read the state of the lock, and wait until a particular state (locked or unlocked) is reached.

```
interface mutex
{
        void lock(void);
        void unlock(void);
        int read(void);
        void await(int);
} =     "396900003ba7424735adfdbd000e3ba8"
        "0000000000000000000000000c01fe7ae";
```

Figure 2.3. Mutex interface

[2]except for instances of built-in classes

Every interface is identified by a 256-bit worldwide unique identifier. This identifier is allocated by the person who creates the interface. The identifier is large enough to allow people to create new interfaces independently.

The **interface identifier** uniquely identifies both the syntax and the semantics of an interface. This means that an interface should get a new interface identifier when the *syntax* is changed, for example by adding a new method, but also when the *semantics* of a method is changed. This approach contrasts others (such as Modula-3) that compute identifiers (called signatures) based only on the syntax of an interface.

Interface identifiers can be allocated in a distributed fashion by delegating parts of the 256-bit number space hierarchically. This is similar to the delegation of host names in DNS and the delegation of filenames under UNIX. An efficient way to delegate parts of the number space is to incorporate other number spaces that are already delegated. Examples include 32-bit Internet addresses,[3] 48-bit Ethernet addresses, 160-bit OSI addresses, and, in the future, 128-bit IPv6 addresses. The interface identifier in Figure 2.3 is generated based on the 32-bit IP address of a UNIX workstation, the current time, and process identifier.

The concept of an interface can be viewed in two different ways. In one way, a collection of interfaces defines the service provided by an object. An application can use interfaces to access or update the state of an object.

From another perspective, we view interfaces as a design tool. By designing interfaces that support a wide variety of implementations (in different objects), a system gains flexibility. Interfaces can be defined at different levels of abstraction. An interface at a low level of abstraction is a file interface which simply defines read, write and locking operations on a linear sequence of bytes. For example, such a low-level interface can be used by applications to create a mailbox. Alternatively, special objects can be created that offer mailbox interfaces directly. The difference between the two approaches is that in the first case, applications do not have to rely on special objects, standard (file) objects are sufficient. Furthermore, standard utilities, for example, the equivalent of the UNIX grep command, can be used. The main disadvantage of the first approach is that it is very hard to change the format used to store the mailbox in the file because all mail applications have to be changed to use the new format. With the second approach, the internal structure of the mailbox is hidden from the applications, and can be changed easily.

Some features present in other systems, such as interface inheritance and versioning, are left out of our object model. For the most part they can, however, be emulated using the support for multiple interfaces per object, and the worldwide unique interface identifiers. Inheritance comes in two flavors: interface inheritance and property inheritance [Taivalsaari, 1996]. **Interface inheritance** allows an interface to be defined as an extension to one or more other interfaces. **Implementation inheritance**,

[3]With the exception of private use addresses[Rekhter et al., 1996].

the most common form of property inheritance, allows an object to be implemented as an extension (or, more general, modification) of some other object implementation. Implementation inheritance is supported by composition (see Section 2.2.4). An interface that is defined using interface inheritance can be replaced with one interface for each of the original interfaces, and a new interface for the methods that were added. The main drawback of this approach is that the user of the object may have to deal with multiple interface references.

Versioning of interfaces means that (version) identifiers are associated with interfaces to identify different versions of an interface. Over time, an interface can be extended or changed and version identifiers can be used to make sure that applications and objects use the same interface. This problem can also be solved by assigning different interface identifiers to different versions of a single (abstract) interface. To interoperate, applications and objects have to support multiple interfaces. This allows a "new" application to use "old" objects, and the other way around.

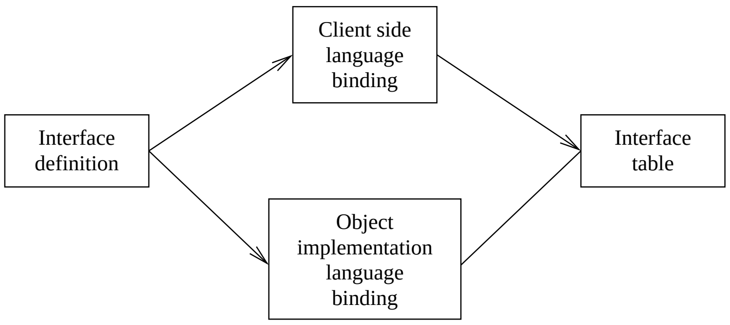

Client side language binding
Interface definition
Interface table
Object implementation language binding

Figure 2.4. Interface compilation

Interfaces are defined in an interface definition language. An interface compiler compiles an interface definition into different programming languages. This is shown in Figure 2.4. The user of an object (client side) needs language-specific routines called client stubs that allow method invocation using the interface. Similarly, the object implementor need server stubs that accept a method invocation and implement a routine that actually implements the method. The complexity of the stubs depends on the mismatch between the calling conventions used in the application/implementation language and the interface definition.

The client and the server stubs have to agree about a common layout of the interface in memory. Figure 2.5 shows the interface table that corresponds to the mutex interface. An interface table contains two pointers for each method: a method pointer and a state pointer. The method pointers are the fields with suffix _m after the method name, and they point to the method implementations. The state pointers (suffix _o[4])

[4]points to the **o**bject

1	soi	unused
2	lock_m	lock_o
3	unlock_m	unlock_o
4	read_m	read_o
5	await_m	await_o

Figure 2.5. Interface table

point to the state of the object(s) on which the method implementation should operate. The reason for storing a copy of the state pointer with each method is related to composite objects, as we describe in Section 2.2.4.

An interface table contains two additional fields labeled soi and unused. The first field is a pointer to the "standard object interface" and the unused field is there to provide alignment and to allow future extensions to the interface table.

```
interface stdObj
{
        int init(ctx_t, error_p);
        void cleanup(int how);
        struct noInf *getinterface(uniqid_p, error_p);
        interface class release(void *infp);
};
```

Figure 2.6. Standard object interface

The **standard object interface** is a standard interface provided by all objects. Figure 2.6 shows the definition of the standard object interface. The methods init and cleanup are used to initialize and to cleanup an object's state. The method init is invoked just after an object is created and cleanup prior to the destruction of an object. The getinterface method provides access to other interfaces. The worldwide unique identifier of an interface is passed to getinterface to indicate which interface is desired. The object either returns a pointer to the right interface table or an error indicating that the object does not support that particular interface. This way the pointer to the standard object interface of an object can also serve as a generic object pointer.

Reclaiming unused, local objects can be implemented in three ways:

1. Explicitly by the "owner" of the object
2. Using reference counts on the objects or on interfaces
3. Using a garbage collector

A garbage collector is the most convenient solution for an application programmer but is hard to implement (a conservative garbage collector is needed to support traditional languages such as C and C++, and the garbage collector needs to promptly collect system resources such as file-descriptors). Explicitly freeing objects is complicated in larger systems where objects may have multiple owners. Reference counting is the solution that remains.

Counting references for objects is implemented by keeping track of the number of references to each interface of the object. The getinterface method increments the reference count and the release method decrements the reference count. The getinterface and release methods are the reason that every interface table contains a pointer to the standard object interface for an object. Access to getinterface is needed to obtain access to other interfaces, release allows the user of an interface to indicate that the interface is no longer needed.

The main problem with reference counts (apart from the accuracy required to maintain the correct count) is to reclaim garbage with cyclic references. Cyclic references appear when, for example, two objects hold a reference to each other. In Section 2.2.5 we will see that to avoid deadlocks, we need to make a distinction between "normal" interfaces and call-back interfaces. This distinction makes it possible to create two acyclic graphs, one containing the normal interfaces and the other with the call-back interfaces.

In practice this means that after the total number of references on all its normal interfaces drops to zero, an object can release all references it holds to normal interfaces of other objects. However, the object should continue to process method invocations that are invoked on call-back interfaces. After all references to normal interfaces are released, we have a collection of objects that have only references to call-back interfaces of other objects. By construction, these objects and references form a directed acyclic graph (DAG). In this DAG, there is at least one object with a reference count of zero. This object can be destroyed, and it releases its references during the cleanup.

The standard object interface is one of a collection of "standard" interfaces. Other standard interfaces are the class interface to create new instances, various interfaces to system objects (memory, thread), and interfaces for distributed shared objects. Other interfaces are application specific, such as the mutex interface that was shown above.

2.2.2 Primitive Objects

A **primitive object** is an object that does not contain other objects. This in contrast to a composite object which *is* composed out of multiple subobjects. The implementation of a primitive object uses three different kinds of data structures: the object's state, its interface tables, and the implementation of object's methods. Figure 2.7 shows a primitive object and its class object. In this example, the object's state is stored (contiguously) in one place and the methods are stored in the object's class object.

The example shows five interfaces: three ("Standard Object Interface," "Interface 1," and "Interface 2") belong to the object,. The other two (a second "Standard Object Interface," and a "Standard Class Interface") belong to the class object. As described

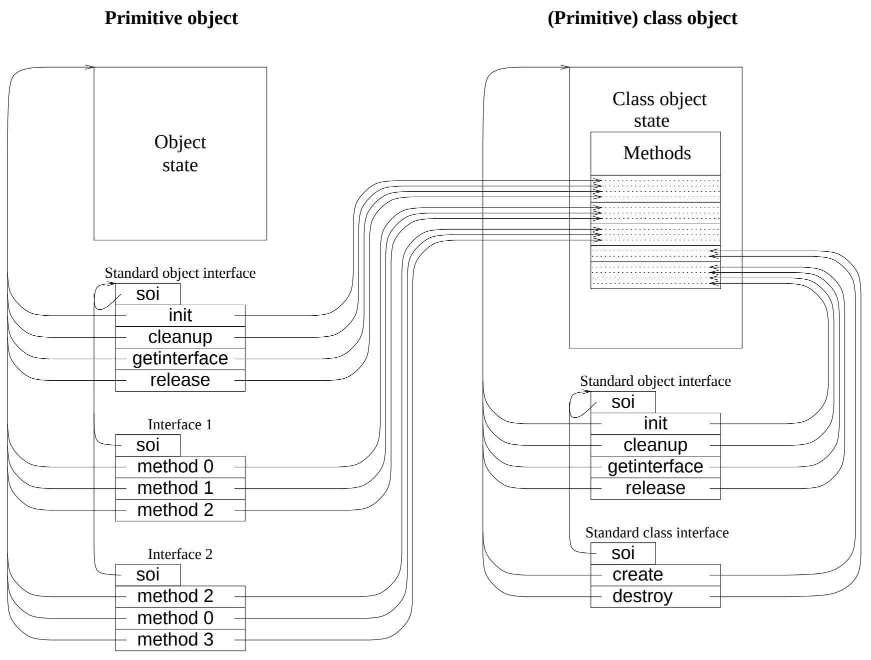


Figure 2.7. A primitive object and its class object

before, each interface table contains two pointers per method (a state pointer and a method pointer) as well as a pointer to a standard object interface.

The object model does not impose restrictions on where an object stores its state, or where its methods are kept. In general, an object's state consists of a fixed sized part with references to dynamically allocated data structures, or references to (interfaces of) other objects. The methods of an object can be stored in three different places. They can be (1) linked with an application, (2) stored in the object itself, or (3) stored in the object's class object. Methods of built-in objects are linked with an application. Figure 2.7 shows that the methods for both the object and its class object are stored in the class object. This is the case when the class object is dynamically loaded.

2.2.3 Class Objects

Class objects are used in three different roles. The first role, and which is the one how a **class object** is defined in our object model, is that of an object that implements the class interface. This requires the class object to create and destroy other objects. The second role of a class object is to store the method implementations for its instances. The third role for a class object is to explicitly identify a particular object implementation. For example, a system can contain multiple memory allocators, or support different communication protocols. To refer to a particular implementation, it is sufficient to pass a reference to the relevant class object.

```
interface class
{
        interface stdObj create(error_p);
        void destroy(interface stdObj);
};
```

Figure 2.8. Class interface

Figure 2.8 shows the class interface. This interface has two methods: create and destroy. The create method creates a new object and returns a pointer to the standard object interface of the newly created object. Before the object can be used, the user of the object has to invoke the init method of the object to allow the object to initialize itself.

The destroy method requests a class object to destroy a particular instance identified by the standard object interface pointer passed as an argument. Before destroy is invoked, the object should be given the opportunity to release any dynamically allocated resources. This is done by invoking cleanup, using the standard object interface pointer, which is part of every interface table.

As described in Section 2.2.1, garbage collection of (local) objects is done by reference counting. The first reference to an object is returned by the create method, and additional references are returned by the getinterface method. References can be released by invoking the release method. When the reference count drops to zero, the release method returns a reference to the class interface of the class object that was used to create the object.

Some class objects are built-in, but most class objects are loaded on demand from some class repository. The implementation of the class object and the method implementations of the instances of the class are stored in marshaled form in the class repository. Dynamically loaded class objects are instances of a super class object. A super class object creates instances without a standard class interface, but with a pickle interface. The pickle interface is a standard interface to save and restore the state of an object. The marshaled state of a class object is generated by a compiler, and can be stored into the newly created class object by invoking the unmarshal method.

2.2.4 Composition

Composite objects are objects that are built out of other objects. The primary purpose of a composite object is to construct, at run-time, an object from a number of smaller objects. For example, in the next chapter we will see that a distributed shared object needs (among others) a replication object and a communication object. The use of a composite object allows the decision about the specific replication and communication objects to be made at run-time.

The main requirements for a composite object are:

1. A composite object should behave like a normal object with respect to method invocation, interfaces, and instantiation.

   This requirement ensures that a primitive object can be replaced with a composite object and vice versa.

2. The methods provided by a composite object should be implemented by the subobjects of the composite object. Exceptions are the methods that are part of the standard object interface, or of interfaces specifically designed to control composite objects.

   The idea here is that code for a composite object can be generated by a compiler only based on a listing of the subobjects, their interfaces and the interfaces exported by the composite object.

3. An object should not be aware that it is part of a composition.

   This requirement allows an object (implementation) to be used both as part of a composition and as an independent object.

4. A composite object should not add much overhead to method invocations.

   Object-based systems often lead to implementations that contain many layers of abstraction when compared to systems designed using more traditional languages. These extra layers of abstraction can be used to create a more flexible system. However, it is important to keep the extra overhead of these layers as low as possible. Some systems go quite far in reducing this overhead. For example, the SELF compiler dynamically optimizes (through inlining) method invocations based on type feedback [Hölzle and Ungar, 1996]. The performance goal for composite objects is to reduce (or eliminate) the extra overhead created by using compositions.

The fourth requirement is met by storing state pointers along with the method pointers in interface tables. This approach introduces extra overhead for every method invocation (an extra memory reference to fetch the state pointer). The advantage is that methods are directly invoked on primitive objects without going through the composite object. This is also the case if one of the subobjects is a composite object itself.

Figure 2.9 shows the components of a simple composite object. This composite object consists of two subobjects: object A and object B. The figure shows the three parts of all objects: their state, methods and interfaces. The state of the composite object consists of the state of the two constituent objects and a small amount of extra state. The methods of A and B are placed in their class objects, which is the normal

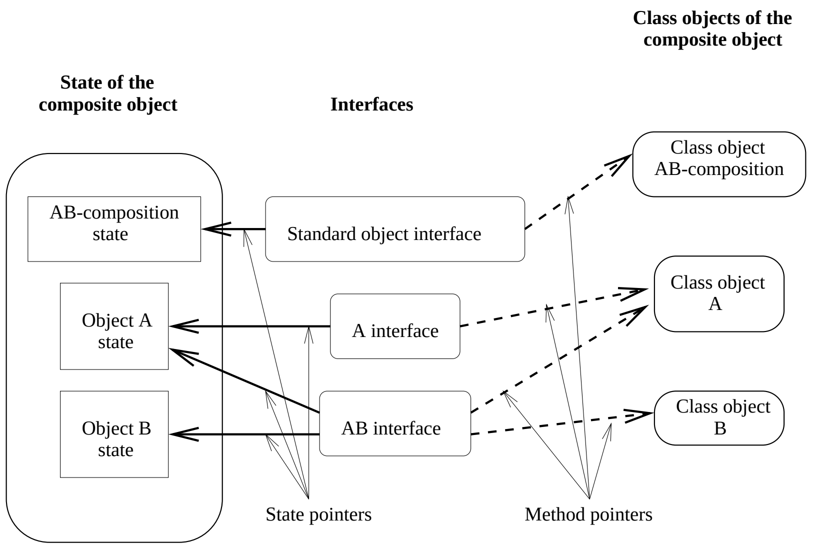


Figure 2.9. A composite object, it's interfaces and class objects

case. An extra class object is present to implement the composition. This object provides the class interface for this composition. Methods that are specific to the composition such as the init and cleanup methods, are also stored in the class object of the composition.

Finally, there are the object-specific interfaces. The "A Interface" is just an interface provided by object A. But the "AB Interface" is more interesting. Some of the methods in this interface are invoked on object A and other methods are invoked on object B. Note that the example does not include a "B interface." In general, compositions can provide arbitrary interfaces based on collections of methods from their constituent objects. Because interfaces tables store both method pointers and state pointers, it is no problem to provide interfaces that are a mixture of methods from different objects.

A composition has a small managing part, called a **compositor**, which has three functions:

1. To create and destroy subobjects
2. To create interface tables based on the interfaces of the subobjects
3. To support the creation of references from one subobject to another

An object that is part of a composition can be private to the composition or it can be shared with other users. Only objects that are private to a composition are created by the compositor.

The requirement that objects should not be aware that they are part of a composition is met using a local name space. In general, local objects are expected to use names in a local name space to obtain references to other objects. This local name space is manipulated by compositors to make sure that subobjects of a composition obtain references to other subobjects in the same compostion. In this scheme names are associated with objects with they are instantiated. The name space provides a set of global bindings with the option of overriding those bindings within a composition. This will be described further in Section 4.3.

2.2.5 System Objects

Figure 2.10 shows the relation between the hardware, operating system, application and various objects. The operating system is layered on top of the hardware as usual. The application is in turn layered on top of the operating system. Normally, an application uses system calls, or shared libraries to access the operating system. In Globe, access to the operating system is also provided by system objects. Examples of system objects are communication objects (for reliable/unreliable point-to-point/multicast communication), memory allocation, threads, locks, file (I/O) objects, and class loaders.

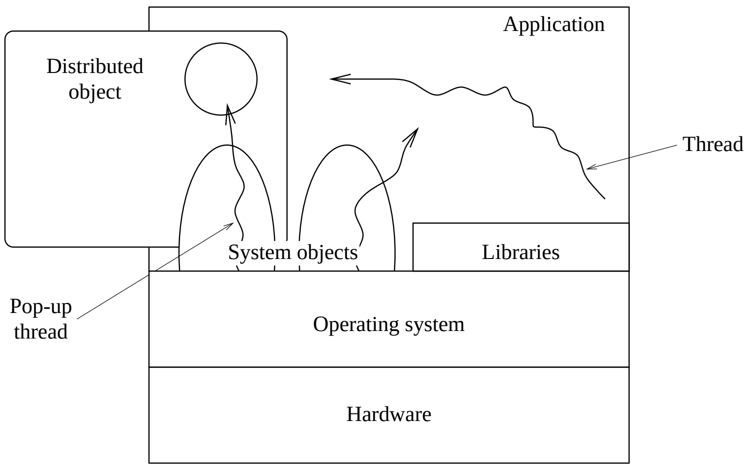


Figure 2.10. Process model

For applications (as well as for distributed shared objects), the advantage of using system objects is that they provide an abstraction of the underlying operating system, and thus increase the portability of the application to different operating systems. However, the main reason for system objects is to provide dynamically-loaded class objects with access to operating system services. Avoiding the use of system calls or explicit library calls simplifies the implementation of the dynamic loader.

Thread Objects Figure 2.10 shows two kinds of threads: a "normal" thread and two pop-up threads. 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. A normal thread is implicitly created when the application is started and is controlled by the application. An application can create additional threads when necessary. These threads execute functions that belong to the application. These functions can invoke methods on (distributed) objects.

A pop-up thread is created by a system object in response to some asynchronous event. To integrate communication cleanly with the model of passive objects, 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 call-back interface of the distributed shared object. Further details of thread objects are described in Section 5.10.3.

Communication Objects The communication object is the main system object used by a distributed shared object. In the next chapter we will see that a single set of interfaces should be offered by all communication objects to allow a distributed shared object to use different communication objects.

A communication object, as used by a distributed shared object, typically consists of two parts: a system object that provides an interface to low-level communication facilities with on top of that object other objects (layers) that implement higher level communication primitives. For example, the system object offers access to UDP/IP and another object implements an RPC protocol on top of a datagram service, in this case UDP/IP. In addition to protocol layers, communication objects can also provide certain security services such as link-level encryption.

Existing approaches to composite communication stacks are the x-Kernel [Hutchinson and Peterson, 1991], and Horus [van Renesse et al., 1996]. The standard UNIX socket interface attempts to provide an interface independent of the underlying communication protocol.

To implement communication interfaces that are more or less independent of the communication protocol they provide access to, it is necessary to create a protocol-independent address data structure. For example, the UNIX socket system calls use the `sockaddr` data structure. The first field of this data structure describes the "address family" to which to address longs. All other fields are specific to the particular "address family."

The two main problems with this approach are that (1) in general, the size of an address is not bounded, and (2) there is no easy way to convert the data in an address structure to and from a human readable ASCII representation. The first problem requires a programmer to assume an upper bound with the risk of failure if an address exceeds that bound, or dynamically allocate address data structures, which is both inefficient and often error prone. Another problem is that relatively large data structures are passed between different protocol layers.

The solution adopted in the interface to a communication object is to install the ASCII representation of an address in the communication object. The object returns an integer that represents the address. This integer is used to specify source and destination addresses of datagrams, data streams, etc.

Communication objects can be classified as follows:

- Unicast or multicast

  A unicast communication object delivers data to one recipient. A multicast or group communication object delivers data to multiple recipients, in some cases even including the sender.

- Reliable or unreliable

  A communication object that provides reliable data delivery attempts to ensure that the data are actually delivered to the intended recipient(s).

- Connection oriented, connectionless, request/reply

  A connection-oriented communication object provides a connection setup procedure that is executed before any data is sent. Connectionless communication objects send the data without any prior arrangements. Request/reply style communication is special in that a single segment of data (the request) is sent to a remote party which in turn replies with a single segment of data.

- Addressing: explicit or implicit

  For communication objects, explicit addressing means that each send operation requires a destination address, and received data is delivered along with the source address. In contrast, with implicit addressing the remote address is stored in the communication object and is not supplied with the send operation.

These four aspects of communication objects allow for a wide range of possible implementations, ranging from a relatively simple unreliable, connectionless, unicast service (for example UDP) to reliable, request/reply multicast objects. The communication objects provide a common set of interfaces, which will be described in Section 5.10.5.

The group communication provided by multicast communication objects is different from group communication in ISIS. In ISIS, group communication is between a group of *processes*. All communication, both multicast and unicast is causally ordered. This means that if process A multicasts a message, which is received by process

B, and subsequently B sends a unicast message to C, then C will receive the multicast message from A before the unicast message from B. The problem with this approach in a wide-area network is that it requires global coordination to be able to order messages (for causal order, all processes in the world would have to participate in the virtual clock algorithm). In contrast, in our model group communication objects join a multicast group and order messages only within a that group.

There is a distinction between open and closed groups. Systems which provide open groups allow any process to send a message to a group. In a system with closed groups, a process has to join a group before it can send messages to the group. In both cases, receiving messages requires joining the group. Open groups can be simulated in a system with closed groups by joining a group, followed by sending a message and leaving the group. For this reason, our groups are closed. Note that explicit join requests in a closed group provide a communication object with more opportunities to optimize the traffic between the various group members, because it allows the communication object to maintain a list of group members.

Group communication objects that support groups with a large number of group members can not be expected to maintain a consistent group membership list. Without group membership lists, it is relatively easy to construct hierarchical groups to provide scalability. Within a group, a group communication object may support point-to-point communication with a group member.

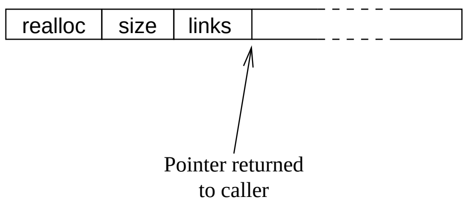


Figure 2.11. Memory segment datastructure

Memory A simple memory allocation system object is provided to allocate segments of memory. Figure 2.11 shows the layout of the memory segments returned by the memory allocator. These segments start with a small header. The size and links fields contain respectively the size and the number of references to the segment. The realloc field is a function pointer which is used to grow, shrink and delete the segment. Storing a realloc function pointer in the segment header allows the use of multiple memory allocators in a single system.

A library that provides packets for communication objects is layered on top of the memory allocator. These packets are designed to minimize copying of data. All data is stored in buffers, which are simply segments allocated by a memory allocator object. Access to a buffer goes through a data structure that is called pkt, which contains a

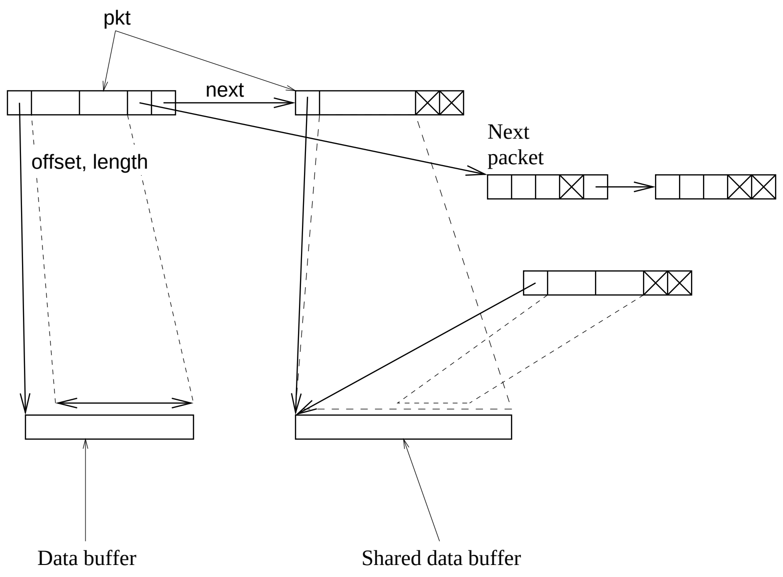


Figure 2.12. Communication packet datastructure

pointer to a buffer, the offset and length of the data in the buffer, and two pointers to other pkt data structures. This data structure is shown in Figure 2.12. The first pointer is used to create a linked list of buffers. A linked list of packet "parts" makes it easy to append, or delete data without copying the data to a new buffer. The second pointer is available to the user of the packet to create a linked list of "complete" packets, for example for a transmit or receive queue. Further details of memory allocation are described in Section 5.10.2.

Locks Synchronization primitives are necessary in a multithreaded environment to protect data structures that are accessed by multiple threads, and to allow threads to wait for other threads to complete an operation. Many kinds of synchronization primitives have been defined: semaphores, mutexes, monitors with condition variables, mailboxes with messages, wait-free (or lock-free) data structures.

A standard approach to synchronization within a single address space is necessary for interoperability between separately developed components. For example, the caller of a method should know whether the invoked object performs concurrency control (thus allowing the caller to invoke the method from multiple threads). Alternatively, the caller has to perform concurrency control if the object does not allow concurrent access.

The approach taken in Globe is that by default objects perform concurrency control for their own data structures. The semantics of an interface are such that methods can be invoked simultaneously from different threads. The advantage of concurrency control on a per object basis is that it makes objects relatively independent: each object is responsible for its own resources. An alternative approach is that a "container" object performs concurrency control for the objects it encapsulates. However, this requires the container object to mediate all accesses to encapsulated objects.

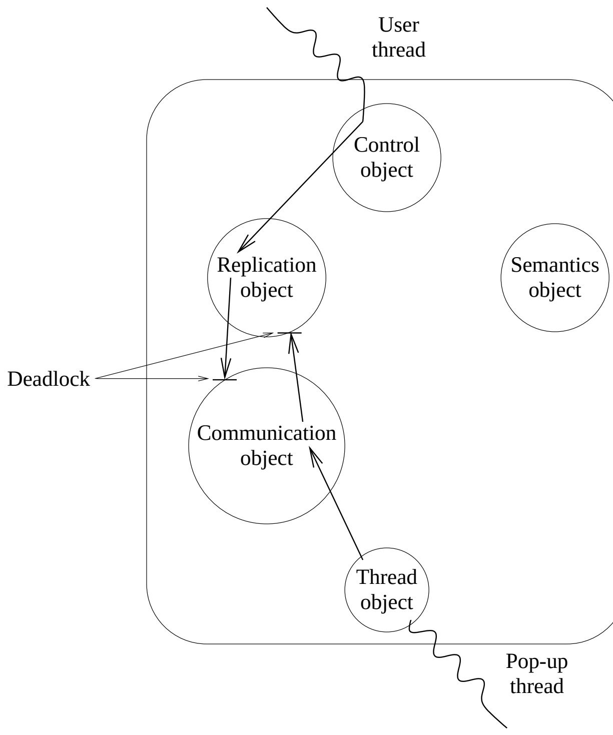


Figure 2.13. Deadlock

The main disadvantage of concurrency control performed by individual objects is risk of deadlocks. Figure 2.13 shows two threads and a composite object. The first thread is a user-level thread which tries to invoke a method on a distributed shared object. At the same time, a pop-up thread is created to handle an incoming message over the network. It is possible that the user thread locks the control object and the replication object, while at the same time, the pop-up locks the communication object. The two threads deadlock when the user thread tries to lock the communication object and the pop-up thread tries to lock the replication object.

We can solve this problem by associating three locks with each object. The idea is that each object may provide two kinds of interfaces: "normal" interfaces that allow other objects (or the application) to use an object, and call-back interfaces that are used to report certain events such as the arrival of a packet over the network.

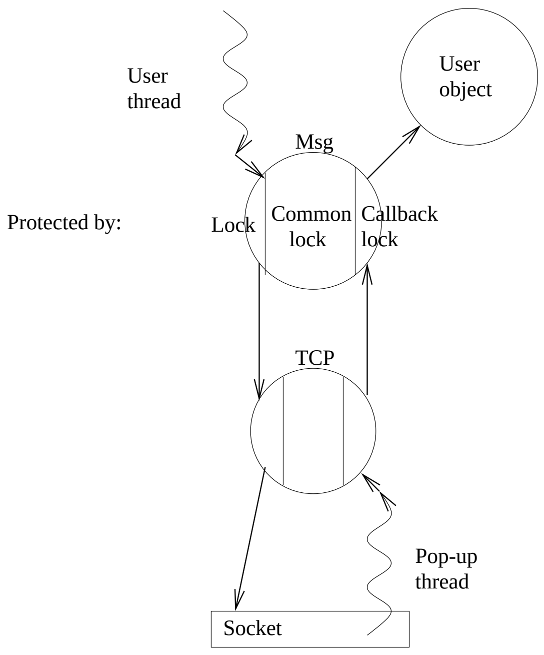


Figure 2.14. Three locks per object

The object uses one lock to perform concurrency control for method invocations on normal interfaces, a second lock to perform concurrency control on the call-back interface, and a third lock to perform concurrency control on the state shared between normal interfaces and call-back interfaces. This is shown in Figure 2.14.

This approach solves the problem shown in Figure 2.13: the user thread uses the “normal” lock, the pop-up thread uses the call-back lock, and shared state within an object is updated by briefly locking (and unlocking) the common lock.

To guarantee that a collection objects cannot deadlock, it is important that three constraints are met:

1. The graph that consists of objects and method invocations by one object of methods in normal interfaces of other objects should form a directed acyclic graph (DAG). This prevent deadlocks on normal locks. Invocations on call-back interfaces form a separate DAG.

2. For normal method invocations, an object may wait for an event to occur before completing the method invocation. On the other hand, methods in call-back interfaces are not allowed wait for other events, and should complete promptly.

3. A thread holding a call-back lock is not allowed to lock a normal lock. This means that a pop-up thread can not reverse direction. Figure 2.15 shows that

the replication object in the middle tries to use the same communication object that delivered the request to send a message to another address space. A deadlocks results if the communication object blocks during the send operation, for example, to wait for an acknowledgment of previously sent data. The reason for the deadlock is that the same thread that waits for the acknowledgment also holds the call-back lock, which means that no new messages can be delivered to the communication object.

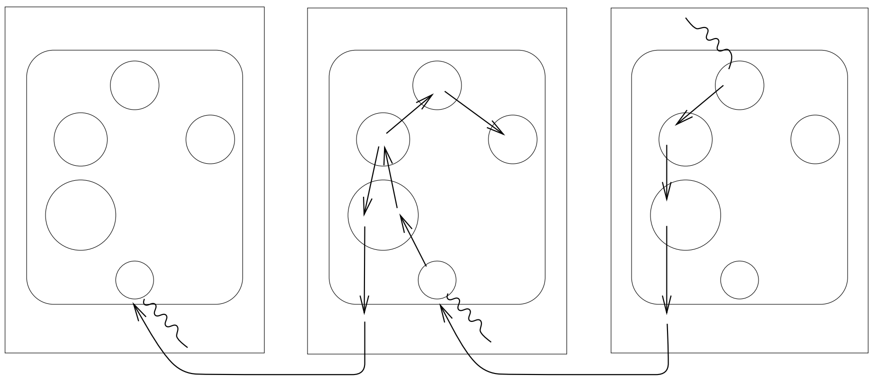


Figure 2.15. Reversing the direction of a thread

The most flexible solution is to implement synchronization primitives as objects. For example, this allows a semaphore to be implemented either directly as a system object, but it is also possible to implement semaphores on top of a message passing system. The main disadvantage of this approach is the (memory) overhead: three lock objects per "regular" object would result in a lot of memory wasted on interfaces tables, administration, etc. In other words, lock *objects* are too heavy weight. A better approach is to have a single locking module (implemented as an object) which provides a collection of locks.

The locks provided by a lock module implement a combination of a mutex with an associated condition variable. This allows the lock to be used for mutual exclusion and for condition synchronization. Further details of lock implementations are described in Section 5.10.6.

2.3 Distributed Shared Objects

The local objects described so far are limited to a single address space. This implies that all state and the interface tables are in one address space. To be able to create objects that span multiple address spaces we introduce distributed shared objects. A **distributed shared object** is an object that can have interfaces in multiple address spaces, or can have its state spread out over multiple address spaces, or both. Distributed shared objects are an extension to local objects, and from a client's point of view, do not differ from local objects.

Distributed shared objects are structured as a collection of local objects that communicate with each other. These local objects are responsible for establishing communication channels between the different address spaces in such a way that a user of the distributed shared 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.

Using this approach, the implementation of a distributed shared object, in terms of communicating local objects, can use arbitrary communication patterns, as well 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 model provides four features to support the separation between a program and the 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.

2. The run-time 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 selfcontained. 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.

Figure 2.16 shows the basic architecture of a distributed shared object. We see a distributed shared object which spans four address spaces: A1 through A4. Each of these address spaces contains a local object that *represents* the distributed shared object, by providing the interfaces of the distributed shared object. Together, these local objects implement the distributed shared object. To distinguish local objects that represent a distributed shared object from other local objects, we call those objects **representatives**.

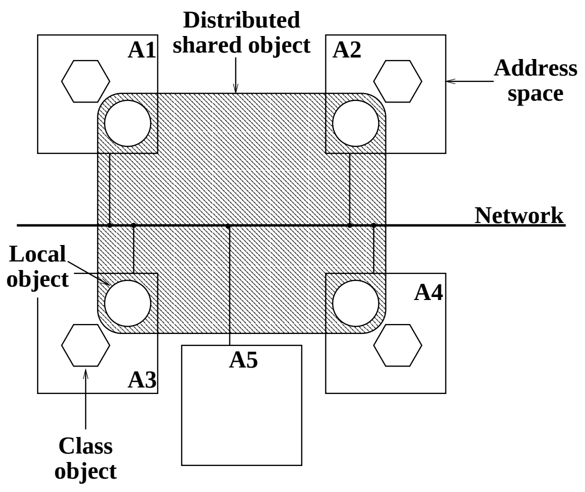


Figure 2.16. Topology of a distributed shared object

Method invocation on a distributed shared object is possible in a given address space only if that address space contains a representative of the distributed shared object. Each representative is typically an instance of a class object which is present in each address space. However, these class objects are not considered to be part of the distributed shared object.

It should be noted that the current architecture does not support automatic worldwide distribution of class objects and interfaces. Even though a distributed shared object is available worldwide, it is not specified how class objects are made available at remote sites or how new interfaces are published. More research is needed to determine how class objects can be downloaded automatically from remote implementation repositories.

Binding Address space A5 does not contain a local object connected to the distributed shared object. Before the program that runs in A5 can access the distributed shared object, it first has to *bind* to the object. Binding to a distributed shared object is a three-step process. First, name resolution is performed to translate a high-level name into the object's low-level name (an address). This step will be discussed in Chapter 4, where it will be split-up into four substeps (a name lookup, a location lookup, destination selection, and implementation selection). The second step is the creation of a new local object. This involves locating and loading the appropriate class object, which is subsequently used to create the local object. The information needed to locate the appropriate class object is provided by the naming system. Finally, the new local object uses network addresses, which are also provided by the naming system, to setup a network connection to a representative in one of the address spaces that are already bound to the distributed shared object.

The representatives that are part of a distributed shared object collectively perform the following three tasks: (1) they store and maintain the state of the distributed shared object, (2) they provide processes with access to the state of the distributed shared object, and (3) they allow processes to bind to (i.e., setup network connections with) the distributed shared object. Not all local objects have to perform all three tasks. For example, a distributed shared object that uses a client/server architecture internally, would have one "server" representative that stores the state of the object and accepts network connections, and zero or more "client" representatives that merely forward method invocations to the server.

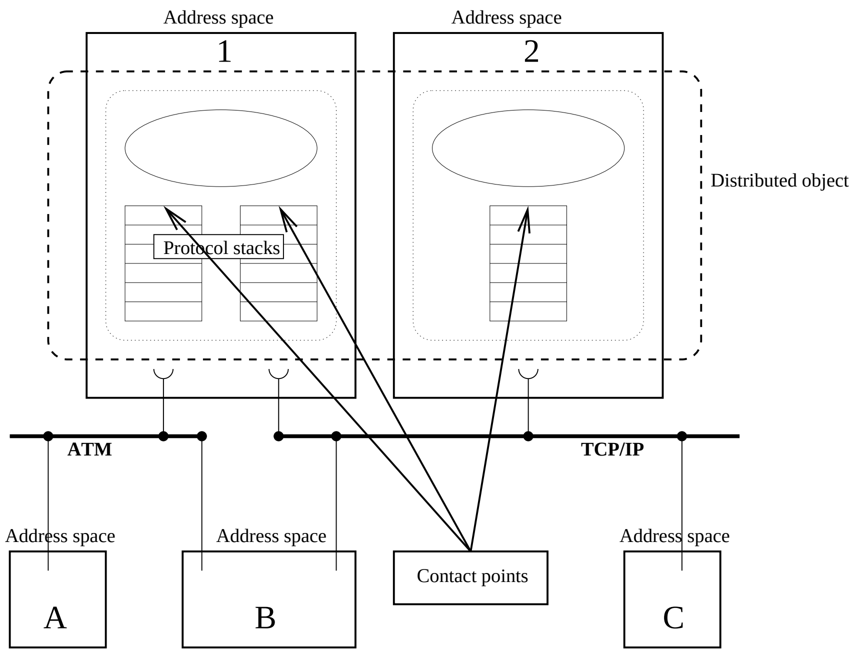


Figure 2.17. Contact points

To bind to a distributed shared object, a local object connects to a contact point. A **contact point** is a communication endpoint provided by a distributed shared object to allow binding. Figure 2.17 shows a distributed shared object with representatives in two address spaces (labeled 1 and 2 in the figure). Three other address spaces (A, B, and C) are connected to two different networks, an ATM network and a TCP/IP network. The representative in address space 1 provides two contact points, one on each network, and the representative in address space 2 provides a second contact point on the TCP/IP network.

Associated with each contact point is a contact address. A **contact address** consists of two parts: a protocol identifier and a network address. The **protocol identifier** not only specifies the network protocol and the transport protocol used by the contact points, but identifies the complete protocol stack that is needed to bind to the distributed shared object. For example, the current Internet e-mail protocol would be identified as SMTP/TCP/IP (i.e., the Simple Mail Transfer Protocol on top of the Transmission Control Protocol on top of the Internet Protocol). The address part of the contact address consist of a network address and any additional demultiplexing information that is needed to locate the right object on a particular machine. To identify an e-mail address on an SMTP service this would be an IP address, TCP port 25, and a login name.

A distributed shared object may provide multiple contact points. These contact points can use different network protocols (ATM versus TCP/IP) or support different application level protocols (FTP versus HTTP). Another possibility is that a distributed shared object provides multiple contact points that use the same protocol on a single network. Multiple, equivalent contact points for a single network improve reliability, allow better locality[5] and better load distribution. Contact addresses are stored in a location service that will be described in Section 4.4.

Note that if we bind to the distributed shared object from address space A, we need a local object that can communicate using ATM. If we bind from address space C we need a local object that speaks TCP/IP. Depending on the communication protocols we may need different implementations of the local objects, but the interface which is presented to the user of a distributed shared object is always the same.

A distributed shared object can choose between different styles of communication to connect a representative to the distributed shared object. Alternatives include the use of connection oriented protocols (e.g., TCP), RPCs, and group communication. For simplicity, we assume the use of a connection oriented protocol when we describe some aspects of the system. For example, connection oriented protocols are assumed in the description of binding to a persistent object in Section 3.3.3 and in the section on security. The distributed shared object is, however, free to use one of the other styles of communication.

Distributed shared objects are generally registered with the name service. This name service will be described in Section 4.2.1. The name service maps one or more user-chosen names to an object handle. An **object handle** consists of two parts: an object identifier and location information. An **object identifier** is a unique identifier assigned to a particular (distributed) object. An object identifier is a worldwide unique 256-bit number, similar to the interface identifier described earlier. The most

[5]Using a nearby site often results in higher throughput, lower delay, and better use of available network resources.

Method name	**Description**
init	Initialize a distributed shared object with its name and contact addresses
gethandle	Return the object's object handle

Table 2.1. Distributed object interface

important properties of an object identifier are that they uniquely identify an object, and that the object identifier is never reused after the object is destroyed. The object identifier allows different address spaces to check whether two object references refer to the same distributed shared object or not. This can be verified by exchanging the object identifiers that correspond to the objects that the references refer to. Furthermore, the object identifier is used to avoid a problem with stale caches, which will be described in Section 4.4.5. To avoid this problem, the local object that tries to connect to a distributed shared object sends the object identifier of the desired object over the communication channel. The distributed shared object verifies that the object identifier corresponds to its own object identifier.

An object identifier is completely location independent. For some implementations of the location service, it is necessary to associate some extra information with the object identifier to be able to find the contact addresses for the object. The extra information in an object handle is called the **location information**.

Table 2.1 lists the methods of the distributed object interface. This interface provides two methods, init and gethandle. The init method is used during binding to inform the new local object about the distributed shared object's name, object handle, and set of contact addresses. The gethandle returns the distributed shared object's object handle.

Factory Objects To create new distributed shared objects, we need something analogous to a class object. We call an object that creates distributed shared objects a **factory object**. The two main differences between a class object and a factory object are:

1. A class object is always in the same address space as the caller, and creates its instance also in that address space.

   In contrast, a factory object is typically a distributed shared object which (also) creates part of the distributed shared object in a remote address space.

2. The creation of a local object consists for most part of allocation and initialization of a memory segment.

   A factory has to allocate a least a contact point for the distributed shared object to allow binding. The contact address that belongs to the contact point has to

Method name	Description
start	Start a factory for a specific distributed shared object
create	Create a new distributed shared object and register the object with the name service
createRep	Create a replica of a distributed shared object on this object repository
deleteRep	Delete a replica of a distributed shared object
startObj	Activate a persistent distributed shared object
gc	Run a garbage collection cycle

Table 2.2. Factory interface

be registered with the location service. Many distributed shared objects require additional resources such as disk space, and registration with security services.

An application that wishes to create a new object has to select an appropriate class or factory object. In Section 4.3 we will see how a local, per-process name space is used to provide access to class objects. That local name space also provides access to a worldwide name space with references to distributed shared objects, including factory objects.

However, there is a difference between class objects and factory objects in that a class object represents nothing more than a particular implementation for a collection of interfaces. In contrast, a factory object not only creates objects of a particular kind, but also allocates resources for the object such as contact points, disk space, etc.

For example, it is quite possible to have two factory objects that both create file objects with the same implementation, but create those objects on different machines. The user (or application) has to select the right factory object depending on security issues or disk quota. This issue does not exists for class objects because they only consume (memory) resources in the application's address space.

This raises the issue of whether a factory object should combine access to certain resources with a particular type of object or keep those issues separate. The main disadvantage of the first approach is that it may lead to an explosion of factory objects: for every collection of resources (machine, cluster) we need one factory object for each type of object.

An alternative approach is that a factory object represents only the resources and that the type of object to create is explicitly passed by the caller. Disadvantages are the need for a standard way of referring to distributed shared object types, and the need to pass two names (the factory name, and the implementation identifier).

Further research is needed to determine which approach is best. At the moment, the lack of worldwide identifiers for distributed shared object implementations prevents using the second approach. To support the first approach, it is necessary to create multiple factories on a single machine. This can be done by introducing meta-factory objects, which create factory objects and are factory objects themselves.

Factory objects implement the factory interface.Table 2.2 lists the methods of the factory interface. Only the first two methods (start and create) are directly related to factories. The other four methods support a service related to factories: the object repository. An **object repository** provides a distributed shared object with persistence by storing replicas of the object persistently. Object repositories will be described in detail in Section 3.3.2.

The start method tells a factory object what kind of distributed shared object it should create. The create method is the method that actually results in the creation of a new distributed shared object, which is registered under the name passed as an argument to create.

Chapter 3

The Architecture of Distributed Objects

3.1 Introduction

The architecture described in Section 2.3 provides a standard way to access a distributed shared object, that is to bind to the object and to invoke methods on the object. Although this architecture shields the user of the object from the underlying communication technology, data placement, replication of data, etc., it still requires the object implementor to deal with those issues.

In this chapter, we will focus on a standard architecture for implementing distributed shared objects. The main reason for a standard architecture for implementing distributed shared objects is to simplify the construction of new (application specific) kinds of distributed shared objects. We concentrate on supporting replication, persistence, and security. These three issues form the basics of a distributed shared object.

We focus on objects that provide operations on shared state. Many persistent objects fall into this category: WWW documents, e-mail messages and mailboxes, news articles, news groups, and simple databases. Nonpersistent examples include multimedia broad- and unicasts, and multiperson game servers.

Objects that fall outside this category are: front-ends to legacy systems, such as existing database systems, specialized objects such as printers, compute servers, mobile objects, and intelligent agents. These objects will be ignored in this chapter, although some of the security techniques that will be described also apply to these objects.

3.2 Replication

The goal of the replication part of this architecture is to separate the semantics of a distributed shared object from the replication protocols used to distribute the state of the object.

Before we discuss the architecture in Section 3.2.4, we look at various replication protocols. We recognize three different classes of models that underlie such protocols:

1. Memory coherence models and algorithms, or more generally: models that are independent of the semantics of an object and that are also independent of the passing of time.

2. Models independent of the semantics of an object that include a notion of time.

3. Models that depend on the semantics of an object.

Instead of maintaining multiple copies of the state of a distributed shared object, it is also possible to partition the state. We say that the state of distributed shared object is **partitioned** if it is split up in a number of disjunct parts that are stored in different address spaces. The state is **replicated** by storing entire copies of the state of the object in different address spaces. These two techniques can be combined by replicating partitions.

Objects with partitioned state are not supported by the architecture to be described in this section for two reasons. First, in simple cases, partitioning is a straightforward extension to the architecture. All that is needed is a mechanism to identify individual partitions. Second, operations that involve multiple partitions generally need support for transactions, lock ordering, etc. These issues are beyond the scope of this thesis.

3.2.1 Memory Coherence Models

Memory coherence models are used to specify the semantics of shared memory multiprocessors and distributed shared memory systems. These models assume a collection of memory cells with read and write operations that access individual memory locations. Some examples of memory coherence models, in decreasing order of provided memory consistency, are: strict consistency, linearizability [Herlihy and Wing, 1990], sequential consistency, causal consistency, and PRAM consistency [Tanenbaum, 1995]. In general, the models that provide stronger consistency are more convenient for the (application) programmer, but are much harder to implement than weaker models.

In an object-based system, we can extend memory coherence models beyond read and write operations on individual memory locations. One approach is to describe consistency with respect to synchronization primitives (locks). Examples are weak consistency [Dubois et al., 1986], entry consistency [Bershad et al., 1993], release consistency [Gharachorloo et al., 1990]. and lazy release consistence [Keleher et al., 1992]. In the **release consistency model**, propagation of changes (write operations) can be delayed while a process or thread is in a critical section. The synchronization primitives used to enter and leave critical sections ensure that pending write operations of a thread that leaves a critical section are propagated to other processors when threads try enter a critical section. **Weak consistency** and **entry consistency** have similar but slightly different semantics.

In an object-based system, we can replace read and write operations on memory locations with a set of object specific methods. We can divide those methods into four categories:

- Read operations

  These operations do not modify the (abstract) state of the object.

- Pure-write operations

  The state of the object is completely overwritten by a write operation, and does not depend on previous states.

- Modify-only operations

  The state of the object is only modified and depends on the previous state of the object.

- Modify-and-read operations

  Modify-only operations that also return a value.

Some categories provide additional opportunities for optimizations. For example, a pure-write or modify-only operation followed by a pure-write operation means that the first operation can be dropped. Modify-only operations and a pure-write operations can be executed in the background (asynchronously) because the operations do not return a value.

Note that there is a difference between a sequentially consistent memory system, and an individual object that provides sequential consistency. In a sequentially consistent memory system, **all** accesses to the memory system are globally ordered, whereas in most object systems (including Globe) accesses to different objects are independent and unordered.

A **replication protocol** is a protocol that runs between a collection of address spaces to keep multiple copies of the state of an object consistent.[1] We can classify various replication protocols using the structure shown in Figure 3.1. In the classification of replication protocols that will be presented below, an address space contains a representative that belongs to one of three possible classes:

[1]This is the case in a distributed system, in a shared memory multiprocessor the replication protocol runs between a collection of caches and memory modules.

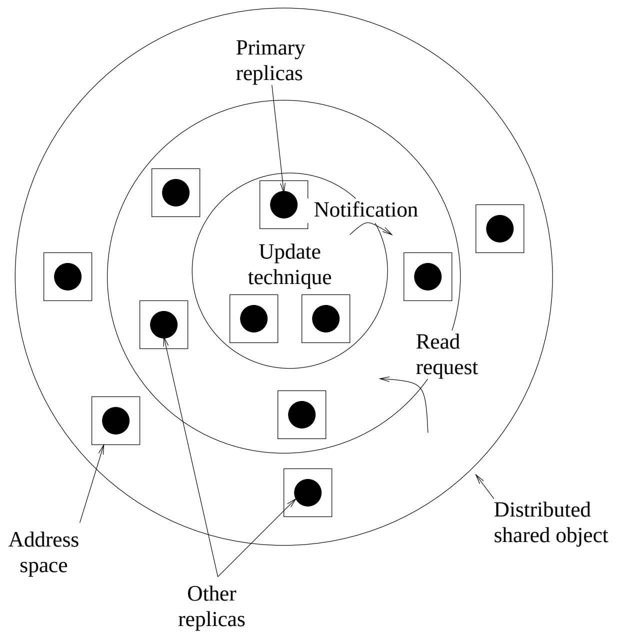


Figure 3.1. Different categories of replicas in a distributed shared object

1. The inner ring contains the collection of representatives that actually modify the state of the distributed shared object. These representatives are called primary replicas, and use an update technique to maintain the state consistent.

2. The second ring contains representatives that receive notifications from the primary replicas that the state has changed.

3. The representatives in the third do not receive notifications, instead they ask representatives in the first or second ring whether the state has changed. In a voting algorithm, a client would ask multiple representatives to obtain a consistent copy of the state.

These issues apply only to algorithms that actually try to replicate some state. Without replication, the state is stored in a single location (we ignore partitioning), which can be either local (a "server") or remote (a "client").

Table 3.1 shows the main issues in replication algorithms that are independent of the semantics of the object and are not time-based. This classification is based on the integration of four kinds of algorithms: voting algorithms [Gifford, 1979], master/slave replication [Budhiraja et al., 1993],[2] active replication [Schneider, 1990] and copy/invalidate (called "invalidation approach" in [Li and Hudak, 1989]).

[2]Also called primary-backup approach in the context of fault-tolerant systems.

	Issue	**Alternatives**	**Secondary issue**	**Alternatives**
1	Size of read set	1		
		> 1		
2	Number of replicas	Constant		
		Variable		
3	Notification algorithm	Update	Change distribution	Method invocations
				State differences
				Complete state
		Invalidate		
4	Number of primaries	1		
		> 1	Update technique	Group communication
				Two phase commit
5	Location of primaries	Fixed		
		Variable		

Table 3.1. Replication issues

The issues listed in Table 3.1 are the following:

1. Do we need to read more than one replica to obtain an up-to-date value or not? Most algorithms keep all replicas consistent. This means that accessing a single replica is sufficient to read the state of the object. Voting algorithms are an exception: the most up-to-date value is obtained by querying multiple replicas.

2. Is the total number of replicas constant or variable? The number of replicas is variable in a system that uses a copy/invalidate approach. A read access causes a local copy to be created, and a write operation destroys all copies except the modified one. An example of a system with a constant number of replicas is a system that uses a single, migrating copy of the state. On both read and write accesses, the state is migrated to the process that wants to access the object.

3. Does the notification algorithm update or invalidate other replicas? When updating, there are several alternatives. It is possible to update the state of replicas in other address spaces by executing the method in those address spaces, by sending the differences between the old and the new state, or by sending the complete state. For example, in a master/slave replication system, the master updates the slaves, whereas in a copy/invalidate system, the current primary copy invalidates all other copies.

4. Some algorithms modify one replica first, and then propagate the changes (using a notification algorithm) to other replicas. Others update multiple replicas together before notifying other replicas. For example, master/slave and copy/invalidate approaches have one primary. In contrast, systems that use voting update a collection of replicas called the "write set." In systems based on active replication, all replicas are primaries.

   When updating more than one replica there is the secondary issue of how to order different operations that try to modify the state. This is not an issue for a single primary replica because concurrency control in a single address space is well understood.

   There are two ways to do multiple operations. The first approach is to use ordered group communication to deliver the modification requests to the replicas. Total ordered group communication is used to implement sequential consistency, causal ordered group communication is used to provide causal consistency in some stock exchange applications. The Usenet News system uses a group communication mechanism which provides no order guarantees.

   The second approach is two phase commit. To modify the state in a collection of replicas, it necessary to first lock the relevant parts of the state in the primary replicas and distribute the modification request. The second step is to tell the primary replicas to apply the operation and unlock the state. Two phase commit is used to update replicas in voting algorithms.

5. Some algorithms use a fixed set of address spaces that contain the primary replicas. In other algorithms this set is variable. For example, master/slave replication typically uses a primary at a fixed location (the master). On the other hand, using copy/invalidate, the location of the primary moves to the site that wishes to update the state of the object.

3.2.2 Time-based Consistency Models

The algorithms that were described and classified in the previous section impose a certain order on operations that change the state of an object and on the distribution of those state changes. However, these algorithms do not limit the amount of time it takes to complete an update operation, or how long it takes to distribute a state change. In many cases, users expect "soft realtime" behavior from a system, that is, the effects of an operation should propagate quickly enough. Time-based consistency protocols are also used when the desired consistency is too expensive. The solution is to allow the state to be inconsistent, but only for a small amount of time.

Time-based consistency protocols use timers and time-out values to determine when a certain action should be taken. The semantics of time-out values used for read operations are different from those used for write operations. For read operations, a time-out indicates how long a local copy of the state of the object is considered valid. The assumption here is that changes to state are made remotely, and that the local copy has to be updated (or invalidated). For write operations, a time-out delays the propagation of modifications to other replicas. In this case we know that we have to update other replicas, but we delay the actual propagation to be able to cluster multiple write operations.

The main issue is where to store time-out values:

- Fixed in the protocol
- Part of the object's state
- Based on the object's last modification time. Or, more general, based on the object's modification history.

For read operations we have several choices of what to do when the time-out value expires: delete the copy of the state, immediately verify the validity of the copy, verify just before the next use, or verify after next use. This last choice (verify after next use) provides a very weak consistency guarantee, but with a low latency on read operations and lower overhead than immediately verifying the validity of the copy. This approach is feasible if the user of the cache can independently verify the correctness of the data. This is the case in the location service, which will be described in the next chapter.

Two examples of time-based consistency algorithms can be found in the NFS remote file system and in WWW caches. UNIX file systems delay writing modifications to the actual file system on disk. For a local file system, the inconsistency between the modifications stored in memory and the contents of the disk can be observed only after a crash. However, in the case of NFS, if a write operation is executed on one host then processes on that host observe the effect of the write operation immediately but processes on another host see the effect after a delay.

Similarly, NFS caches data read from the remote file system for a while. After the time-out expires, the NFS client asks the server for the modification time of the file for which data is cached. If the modification time is not changed, the cached data is kept.

In contrast to NFS, which uses fixed timers, the time-out of cached data is computed by many WWW caches based on the last modification data of a WWW document. The time-out is set to a fraction of the amount of time that has passed since the last modification to the document.

Both NFS and WWW caches provide the user with a limited control over their caches. In UNIX (and therefore NFS) the user may call sync or fsync to force writing data to the underlying file system immediately. This allows a user to bypass the caching of write operations. NFS does not support bypassing the caching of read operations.

In the case of WWW caches, the user can explicitly request a WWW cache to ignore the cached and fetch a new copy of the document. Furthermore, certain documents are not cached at all (all documents with names starting with /cgi-bin/) and documents may contain an upper limit on the amount of time they are cached.

3.2.3 Models Based on the Semantics of an Object

In a wide-area system, different objects often have different consistency requirements. We can say that from the point of view of the entire system, the appropriate type of replication always depends on the use and semantics of an object. For some objects, it is sufficient to select a standard replication algorithm that matches to object's semantics; for others, a custom algorithm is required.

The use of different, but standard replication algorithms for different classes of objects can be illustrated by the following three examples: information publishing, groupware (shared whiteboards and software projects), and working documents that are also accessible to other users.

Objects used for information publishing use a small number of writers that can change the state of an object and a large number of readers. The read/write ratio is typically quite high. A suitable replication strategy for those objects is a consistent core with rings of caches. This architecture will be discussed in Section 3.2.5.

On the other hand, in groupware objects, all participants are more or less equal. In this case, all processes observe the state of the shared object and modify the state occasionally. These objects benefit from active replication in combination with group communication protocols that implement total order, causal order or unordered message delivery.

In the third example, the objects are mostly accessed by processes belonging to the owner of the object, which read and modify the state of the objects. Occasionally, the state may also be read (and sometimes modified) by processes that belong to other users. The owner of the document requires strong consistency for his processes but other users will have to do with less consistency. A good replication strategy is to use a strong consistency replication algorithm, such as master/slave replication or copy/invalidate. Processes that belong to other users and only observe the state of an object use polling. This approach allows a user to share its objects with colleagues or friends without a noticeable performance impact or lower consistency guarantees.

For other objects we need object-specific consistency algorithms. An example of an object-specific consistency algorithms is described in [Golding, 1992b]. This consistency algorithm is used to implement REFDBMS, a distributed database for bibliographic references. The replication protocol uses an anti-entropy algorithm to distribute changes to the state of the database. In this algorithm, neighbors exchange updates to the database. These updates contain a timestamp, which is used to order those updates. Due to the nature of the replication algorithm, it is possible to get conflicting updates.

These conflicts are resolved by taking the semantics of the update into account. For example, two insert operations may specify the same citation key. The solution is to generate a new citation key for the second record.[3] The key Smith90 would be replaced by Smith90a, or Smith90b, or in general the first one that does not exist.

Another example is the Usenet News system. The News distribution protocol NNTP uses a flooding algorithm to distribute News articles over the Internet and over Usenet. This protocol uses an article identifier and date stored in the header of the article to prevent the flooding algorithm from looping. Finally, some News readers (application programs) use the References field in the header to present the News articles in causal order.

Note that object-specific replication algorithms are usually optimizations that provide higher performance or better scalability than standard algorithms. Cheriton argues that message ordering should be implemented at the application level, and that general purpose group communication systems do not work [Cheriton and Skeen, 1993]. Another reason why *existing* systems use application specific protocols is the lack of standard replication protocols.

The main disadvantage of using object-specific replication algorithms is that the object writer, usually an application programmer, then has to deal with replication. This is also argued by Birman in response to Cheriton [Birman, 1994].

3.2.4 Architecture

The main goal of a standard architecture for implementing distributed shared objects is to separate the implementation of the semantics of a distributed shared object from the replication protocols used to keep the state of the object consistent. The idea is that an application programmer implements only the semantics of a distributed shared object, and that any replication protocols needed are part of the "system" or written by systems programmers.

Two ways to meet this goal are: (1) using a special, distributed-object compiler, or (2) by linking application-specific code with standard implementations taken from a library.

The first approach is used in the parallel programming language Orca [Bal et al., 1992a]. This language allows a programmer to assume that his code runs on a virtual multiprocessor. The compiler analyzes the program for suitable replication policies and generates code that makes calls to the appropriate communication primitives that are part of the runtime system.

[3]The algorithm assumes that the message propagation delay is bounded but high. All operations are ordered based on their timestamps and the originating host.

The main disadvantage of this approach is that recompilation is needed to adapt to changes. For example, when new replication algorithms are introduced, objects have to be recompiled to take advantage of those new algorithms.

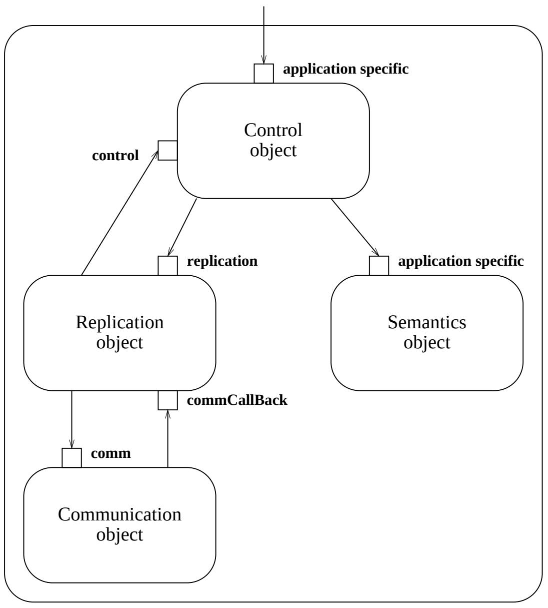


Figure 3.2. Distributed object architecture

The alternative approach, which is used in Globe, is to combine an object that implements the semantics of the distributed shared object, with another object that implements a replication protocol. This approach is shown in Figure 3.2. This architecture isolates the developer of a distributed shared object from communication, replication and consistency management. The developer is responsible for the implementation of the "semantics object" which captures the actual functionality of the distributed shared object. The replication and communication objects are selected from a library of standard communication and replication objects.

A **communication object** provides a simple interface to send data to other address spaces and to receive data from them. A communication object supports 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 the object's state, and marshaled invocations (and their return responses) to other address spaces. Communication and replication objects are generally written by system programmers.

Note that replication objects and communication objects have to agree about the desired respectively provided functionality. The two main issues are (1) group communication versus point-to-point communication and (2) reliable versus unreliable delivery data delivery.

The **semantics object** is implemented by the application programmer as a single copy, nondistributed local object. To simplify the implementation of the semantics object, the architecture hides concurrency control from the semantics object (i.e., the semantics object does not have to support method invocations from multiple threads at the same time). The semantics object provides one or more application specific interfaces to change the state of the object and a standard interface to transfer the state from one semantics object to another.

To connect the semantics object to a replication object, we need a fourth object, called the control object. 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. Additionally, it marshals arguments to method invocations and unmarshals the results. The semantics object is effectively encapsulated in a monitor.[4] This relieves the application programmer from managing concurrency control.

The reason that the control object is separated from the semantics object is that the implementation of the control object can be generated by a compiler, similar to the generation of RPC stubs. The input to the compiler that generates the control object is a list of interfaces exported by the semantics object and the particular marshaling technique to be used.

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 finite 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).

  This requirement is needed for any replication object that "replicates" method invocations. For example, the active replication protocol multicasts a method to all replicas, which execute these methods all in the same order. To keep the state of the distributed shared object consistent, it is important that all semantics objects have the same state before and after the method invocation.

[4]Both the replication object and the control object can implement local concurrency control. The replication object performs concurrency control for accesses that cross address space boundaries. An advantage of making the replication object responsible for local concurrency control is that a single object is responsible for both kinds of concurrency control. However, associating local concurrency control with the control object allows some extra (local) optimizations and leads to a slightly better local locking model. The prototype implementation for Globe implements local concurrency control in the control object.

Without a state machine, it is not possible to use active replication, but many other replication objects do not replicate method invocations. These replication objects invoke a method on a single semantics object and distribute the new state to other semantics objects.

- 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. In this case, the state of the semantics object should be left unchanged.

  To implement this behavior, we split the implementation of a method into two parts. The first part computes, without changing the state, whether the method can be executed without blocking or not. The first part returns directly to the caller when the method blocks. The second part actually implements the method. Essentially, the first part is a **guard** for the second part. The second part is only executed if the guard does not fail (that is, does not return to the caller).

- The semantics object should be able to marshal and unmarshal its state.

  There are three approaches to implement marshaling of the state: (1) the (application) programmer who writes the semantics object implements marshaling, (2) marshaling code is generated based on a description of the object's state (the various fields and their types), (3) a standard interface is defined between the control object and semantics object. The control object gets a list of field names, their types and their contents and converts this information to a format suitable for transferring the data over a network.

  Currently, the programmer who writes the semantics object has to implement marshaling. However, this should be changed to one of the two other approaches.

To be able to use different replication and communication objects with a single semantics object and derived control object, we need a standard interface between the control object and replication object and another interface between the replication object and the communication object. The interface to the communication object is straightforward. There are methods to setup a connection, to join and leave multicast groups, and to send data. Incoming data and connect indications are handled by pop-up threads which invoke methods on callback interfaces exported by the replication object.

We can structure a method invocation on a distributed shared object as a three-step process. This is shown in Figure 3.3. The first step invokes the local replication object, giving it control over the execution of the second step, which deals with global state operations. In this first step, the replication object may acquire any necessary

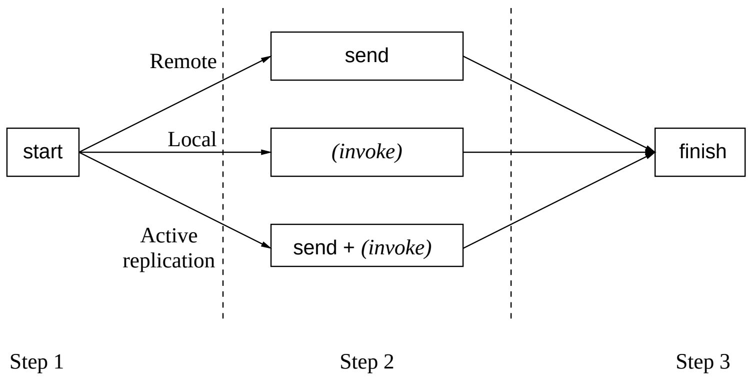


Figure 3.3. Three alternative ways of performing a method invocation

locks, fetch a copy of the state, invalidate other copies, etc. The replication object also decides which of the three alternatives should be taken in the second step. The three alternative in the second step (local execution, remote execution, or active replication) will be explained below.

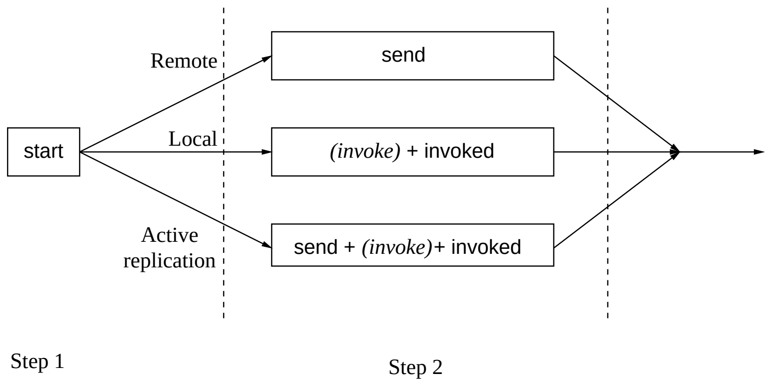


Figure 3.4. Performing method invocations without the third step

Finally, as a third step, the replication object is given the opportunity to release locks, update remote replicas, etc. In the actual implementation, the third step is integrated with each of the three alternatives. The result is the picture in Figure 3.4. Two alternatives for step 2 get an extra action (invoked) to replace the third step.

The three alternatives for the second step are:

- Remote execution

  This alternative is used, for example, by the client-part of a client/server replication object to forward the method invocation to the server. In the remote address space (or address spaces), the replication object proceeds to execute according to its specific replication protocol, effectively doing a remote method invocation.

In the send step, the control object passes the marshaled arguments of the method invocation to the replication object. Later, the replication object returns the marshaled results to the control object.

- Local execution

  This alternative is used for a method invocation on the server-part of a distributed shared object that uses client/server replication. Another example, is a read operation on an object that uses active replication. The control object simply invokes the appropriate method on the semantics object. Afterward, the replication object is informed that the method has been invoked.

- Active replication

  This alternative is a combination of local and remote execution and is used for write operations on an object that uses active replication. 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.

The replication object deals with two kinds of user data. The first kind are marshaled copies of the state of the semantics object. The second kind are the marshaled arguments and return values of operations. Marshaled state and marshaled arguments are provided by the control object. The control object marshals the arguments itself and obtains the marshaled state from the semantics object. For example, if a replication object acts as an RPC stub, it does nothing more than sending marshaled arguments to the remote server and waiting for the marshaled results. On the server side, the marshaled requests are passed to the control object and marshaled requests are sent back.

A master/slave replication object may pass around new copies of the state of the object from the master to the slaves after a write operation has been performed. A replication object that supports active replication only has to send marshaled arguments around. Note that the replication object decides whether and when any state is stored in the semantics object. The replication object can, for example, make this decision based on observed read/write ratios.

To illustrate the interaction between communication, replication, control and semantics objects in different address spaces, consider the following example in which a distributed shared object uses master/slave replication. Figure 3.5 shows the flow of control when invoking one of the object's methods. The method execution consists

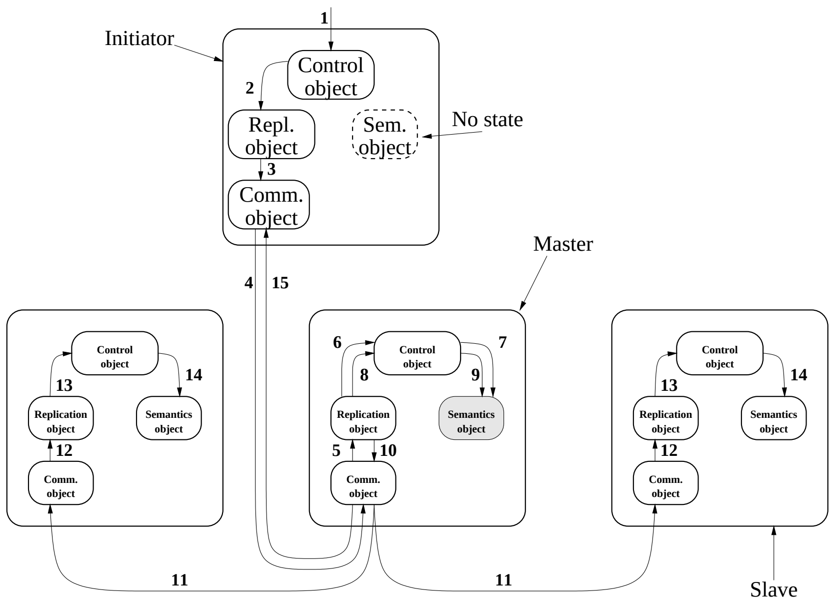


Figure 3.5. A method invocation on a distributed object using master/slave replication

of four phases. In the first phase, the initiator, the replication object in the address space where the method is invoked, forwards the request to the master. In the second phase, the master executes the method locally. The master updates the state of slaves in the third phase, and in fourth phase the initiator receives the results of the method invocation.

Figure 3.5 shows fifteen steps, some of those steps invoke methods , other steps send messages over the network. Note that for method invocations, the arrows only show the invocation and not the return of the invocation.

1. The method invocation starts at the control object in an address space without a local copy of the state of the distributed shared object.

2. The control object passes the marshaled arguments to the replication object.

3,4. The replication object calls the communication object to send a request to the master. After the request is sent, the replication waits for a reply.

5. When the request arrives at the communication object of the master, the communication object creates a pop-up thread and passes the contents of the request to the replication object.

6,7. The replication object passes the marshaled arguments to the control object, which invokes the method on the semantics object. The semantics object returns the results of the method invocation to the control object, which marshals the results and returns them to the replication object.

8,9. There are three ways to propagate the changes to the slaves: the master can either send a new copy of the state, send the changes in the marshaled version of the state, or send the marshaled arguments and tell the other replicas to execute the method themselves. The last alternative is close to active replication.

Figure 3.5 shows the first alternative, where the replication object calls the control object to get a marshaled copy of the new state of the semantics object, the control object in turn calls the semantics object, and returns the result to the replication object.

10,11. Next, the replication object calls the communication object to send the new state to the slaves.

12,13. The communication objects of the slaves create pop-up threads and invoke their replication objects, which pass the marshaled state to the control objects.

14. The control objects install the state in the semantics objects.

15. After the new state is successfully shipped, the replication object of the master returns the marshaled results to the communication object, which sends them back to the initiator's address space.

Finally, the communication object in the originating address spaces returns the reply packet, which is decoded by the control object, and the results of the method invocation are returned to the caller.

The scheme described above can be enhanced by allowing requests to be sent to a slave instead of just to the master. If the request modifies the state of the object, the slave has to forward the request to the master. However, if the request is a read operation, it can be executed locally. This increases the throughput and decreases the latency for read operations.

In Globe, the steps shown in Figure 3.4 are implemented as a finite state machine executed by the control object with input from the replication object. Figure 3.6 shows the transitions between the various states of the control object. For each method invocation, the control object starts in the start state. After the start state, the control object moves to either the invoke or the send state. The control object returns control to the invoker of the method when it reaches the return state.

Each state has an associated action which is performed by the control object when it enters the state. Table 3.2 lists the states and the associated actions. The start, invoke, and send states have corresponding methods in the replication interface. These methods return the input to the state machine. For example, the start method returns

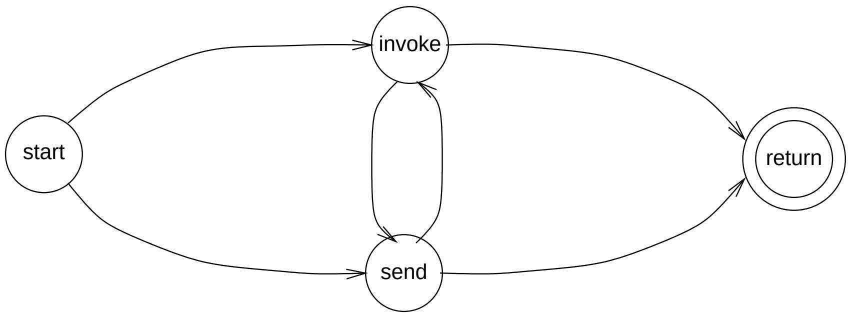


Figure 3.6. Method invocation state machine

State	Action
start	Invoke the start method on the replication object
invoke	Invoke the operation on the semantics object and invoke the invoked method on the replication object
send	Create a packet with the marshaled arguments of the method invocation. Invoke the send method on the replication object and pass the packet as an argument. The send method may return a packet that contains the result of the operation.
return	Return the results of the operation to the caller

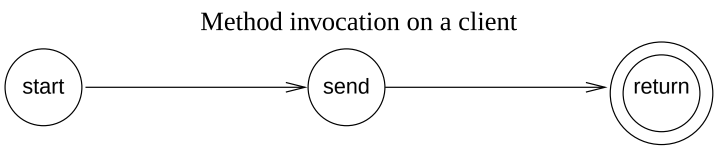
Table 3.2. States of the control object

whether the next state is the invoke state, or the send state. The replication object thus decides what the next state should be.

The implementations of the start, invoked and send methods in the replication object depend on the replication protocol. The various possibilities can be illustrated with four example replication objects: a client/server approach, active replication, a master/slave replication object, and a replication object that implements a copy/invalidate scheme.

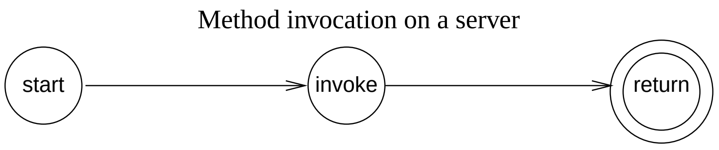
To be practically useful, the control and replication objects have to recognize different kinds of operations. Currently, the control object tells the replication object whether the operation is a read operation, which does not modify the state of the object, or a write operation, which potentially modifies the state of the object. Further research is needed to determine whether additional categories are needed. Some of the examples show different implementations for read-only operations and modify operations, for other examples this distinction does not matter.

The examples are illustrated with figures that show the sequence of states of the state machine for a particular replication protocol. Additionally, a table lists the implementations of the start, send and invoked methods in the replication object. Some implementations of the start only return the next state. In this case the actions are labeled "(nothing)."



Method	Method implementation
start	(nothing)
send	Marshaled arguments are sent to the "server" replication object. The marshaled results of the remote method invocation are returned.

(a)



Method	Method implementation
start	Grab a local, exclusive lock to prevent concurrent method invocations on the semantics object. As described earlier, in the prototype implementation, this locking is done by the control object just before a method is invoked on the semantics object. For completeness, the lock and unlock calls needed when the replication provides the locking are included in these examples.
invoked	Release the lock.

(b)

Figure 3.7. Method invocations on a client/server replication object

Client/Server Replication Object The simplest replication object occurs in client/server computing where no replication is provided. The distributed shared object's state is stored in the address space of a server and replication objects from other address spaces merely forward requests to the server's address space.

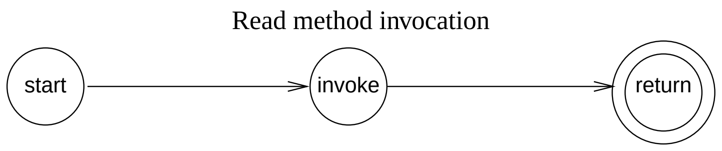
Method invocations on the server can be executed locally. Method invocations on the client are always forwarded (send) to the server. Figure 3.7 shows the state transitions for the client/server replication object. Method invocations always start in the start state. The control object invokes the start method of the replication object. If the replication object is a client, the start method does nothing. On the server, the replication object may grab a lock (this depends on the implementation of the control and replication objects). The replication object returns the next state.

For the client, the next state is the send state. The control object marshals the arguments and invokes the send method. This method returns the marshaled results. On the server, the next state is the invoke state. The control object first invokes the appropriate method on the semantics object, and afterward invokes the invoked method on the replication object. The invoked method releases the lock. Both the start and the invoked method return a transition to the return state. Instead of entering the return state, the control object returns to the user.

Active Replication A replication object that implements active replication keeps an up-to-date copy of the state of the object in every address space. The replication objects keep the state consistent by executing all requests in the same order in all address spaces. This order is typically provided by a totally ordered group communication object. Figure 3.8 shows the schedules for active replication. A read-only operation can be executed locally because every address space has an up-to-date copy of the state of the object. For a modify operation, the marshaled arguments are passed around using total order group communication. The calling thread is suspended until its request is received from the above mentioned group communication object. At that point, the send method returns and the control object invokes the corresponding method on the semantics object. The invoked method is used to signal the replication object that the method invocation on the semantics object is complete and that the replication object can continue processing messages that are delivered by the group communication object.

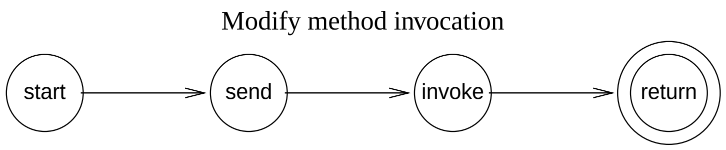
A Master/Slave Replication Object A simple extension to the client/server replication object is master/slave replication. The state of the distributed object is replicated and one copy is designated (or elected) as the master copy. All operations that modify the state of the distributed object are forwarded to the master. The master executes the method and updates the slaves before sending a reply to the requesting address space. This scheme provides increased fault tolerance and availability over client/server replication. Note that it is also possible to invalidate the state of a slave instead of sending updates. The use of invalidations generally provides increased performance. However, fault tolerance and availability are reduced.

Master/slave replication needs three different schedules depending on the role (master or slave) and the kind of operation (read or modify). These schedules are shown in Figures 3.9 and 3.10. The master executes the method invocation locally and informs the slaves if the method invocation was a modify operation. A read method



Method	Method implementation
start	Grab exclusive lock.
invoked	Release lock.

(a)



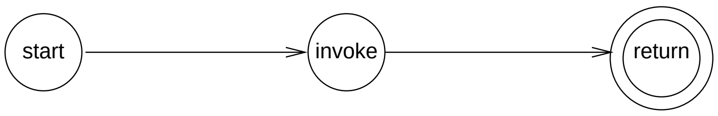
Method	Method implementation
start	(nothing)
send	Marshaled arguments are sent to all replication objects using total ordered group communication. The send operation blocks until the current operation is at the head of the queue of requests to be processed. At this point, the replication grabs the exclusive lock and returns control to the control object. The processing of incoming requests is suspended until the current request has completed. No results are returned to the control object.
invoked	Release the lock and continue processing other (incoming) requests.

(b)

Figure 3.8. Method invocations on an "active replication" replication object

invocation on a slave is also a local method invocation on the semantics object, except that sometimes the replication object has to fetch the state from the master. Modification operations are forwarded from the slaves to the master. The master returns the marshaled results of the method invocation.

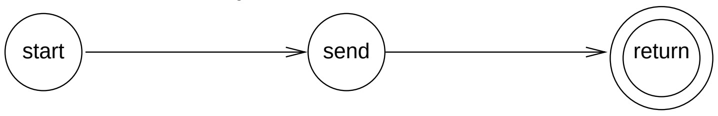
A Copy/Invalidate Replication Object The schedule for a copy/invalidate replication object is shown in Figure 3.11. The copy/invalidate algorithm starts by obtaining a copy (read operation), or ensuring an exclusive copy (modify operation). The method



Method	Method implementation
start	Grab exclusive lock.
invoked	Release lock. After a modify operation, inform the slaves about the new state, either by sending them the new state or by sending an invalidate message.

Figure 3.9. Method invocations on the master

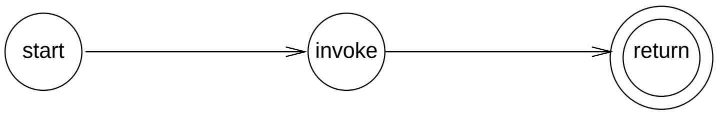
Modify method invocation on slave



Method	Method implementation
start	(nothing)
send	Forward the request to the master. The master returns the result of the operation.

(a)

Read method invocation on slave

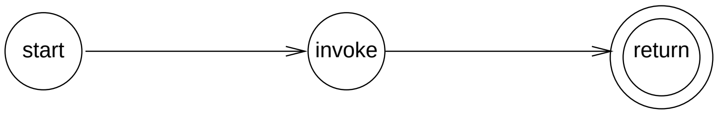


Method	Method implementation
start	Obtain an up-to-date copy of the state, if necessary. Grab exclusive lock.
invoked	Release lock.

(b)

Figure 3.10. Method invocations on a slave

is executed locally, and after the method invocation on the semantics object, the replication object needs to be told that the state of the semantics object can be passed to other address spaces.



Method	Method implementation
start	Obtain an up-to-date copy of the state. Invalidate all other copies if the operation modifies the state of the object. Block incoming invalidate requests or request for a copy of the state. Grab exclusive lock.
invoked	Release lock and continue processing incoming requests.

Figure 3.11. Method invocations on a copy/invalidate replication object

Blocking Operations The implementation of blocking operations in a distributed shared object is different from the one used in a local object. A monitor like construction is used for local objects. The monitor is implemented using a mutex lock to protect access to shared state and condition variables to block a thread in a method invocation until a particular condition becomes true. The thread that caused the change that makes the condition true is responsible for "signaling" the condition variable.

In a distributed shared object, this approach is not always feasible. A large number of blocked method invocations may result in an equally large number of blocked threads in a single address space. Furthermore, in a distributed shared object that distributes method invocations (for example one using active replication), a single method invocation would result in blocked operations in a lot of address spaces. A large number of blocked threads leads to scheduling and ordering problems [Langendoen et al., 1997], and consumes a lot of memory, both virtual and physical.

The solution is to explicitly queue continuations for blocked method invocations [Draves et al., 1991]. A **continuation** is a data structure that contains the necessary information to allow another thread to continue the execution of the blocked method invocation. The routine that cannot continue simply returns to its caller after the continuation is created. This allows the pop-up thread to be deleted.

Creating continuations for method invocations can be quite complicated if the method invocation has already changed part of the state of the object. For this reason, we require a method invocation that blocks not to change the state of the object before it blocks. This allows the method to be aborted and retried multiple times before it succeeds. The information that needs to be stored for the continuation is the method to call and the arguments to the method.

A method invocation blocks when a semantics object indicates that it cannot complete the operation. Figures 3.12, 3.13, and 3.14 show three different situations in which a semantics object may report a guard failure. In Figure 3.12 the method is executed by the semantics object in the same address space as the caller. In Figure 3.13

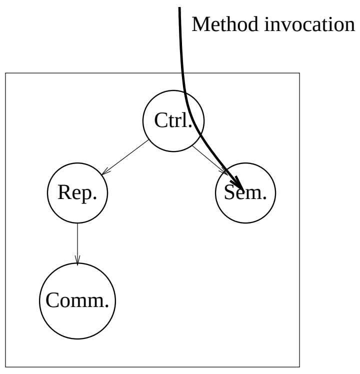


Figure 3.12. Method invocation — local semantics object first

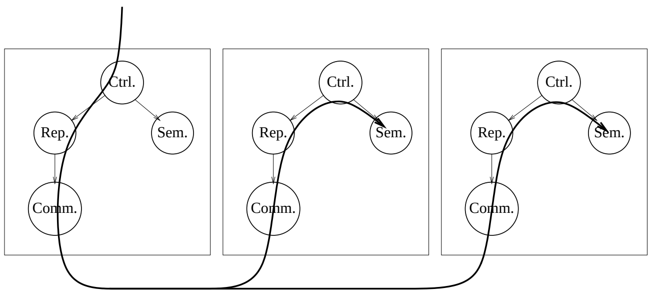


Figure 3.13. Method invocation — remote semantics objects first

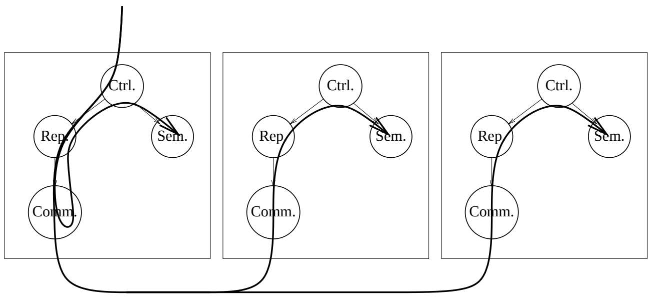


Figure 3.14. Method invocation — local and remote semantics

the method invocation is executed in two remote semantics objects. The third alternative is that both local and remote semantics objects are involved.

Note that these figures show the method invocation only until the execution blocks in the semantics object. In general, it is quite possible that the entire state of a semantics object is copied from one address space to another, before or after the method is invoked on the semantics object. For example, it is possible that control is returned to the caller immediately after the method invocation on the semantics object in Figure 3.12. Alternatively, control can be passed to the replication object which distributes the new state to other address spaces. When multiple semantics objects are involved, we assume that they are consistent: either they report a guard failure or none of them do (i.e., we assume a “state machine” approach).

We could either store the continuation in the control object or in the replication object. To keep the semantics object as simple as possible, we do not store the continuation in the semantics object. Furthermore, we use the communication object only for communication. The replication object is the first choice for storing a continuation: the right place to store a continuation (which address space) depends on the particular replication protocol used. Therefore the continuation should be stored in the replication object(s). A disadvantage of storing the continuation in the replication object is that the control object has to decode the marshaled arguments each time the method invocation is retried.

Note that the thread that originally started the method invocation has to block until the method invocation is complete. This means that in some cases, we can avoid creating a continuation by blocking the caller’s thread. For example, a blocking operation in the address space that holds the master replica in a distributed shared object that uses master/slave replication can block in the invoked method.

3.2.5 Highly-Replicated Objects

A distributed shared object with a high read/write ratio and a large number of readers can benefit from the introduction of read-only caches. A read-only cache executes read operations locally and forwards modify operations toward a central part of the object, called the **core**.

To keep the caches up-to-date we can let a cache poll its parent before each request, or every once in a while (after some time or after some number of requests). Alternatively, the parent of the cache may inform the cache about a new state by sending an invalidate message, by sending the state, or by forwarding the modify operations that are executed on the state.

An important parameter of the update policy is the fault tolerance of an update operation. A parent may store contact addresses for child caches in its persistent state and continue sending update notifications until the child reacts, or the parent may send updates as an optimization and delete information about a child upon failure. We can distinguish persistent callbacks, where the parent knows about its children and can wake them up if necessary, and nonpersistent callbacks which are active only when the cache is active. Figure 3.15 shows a distributed shared object that consists of a consist core with layers of caches around that core. Note that different objects may use different layers of caches.

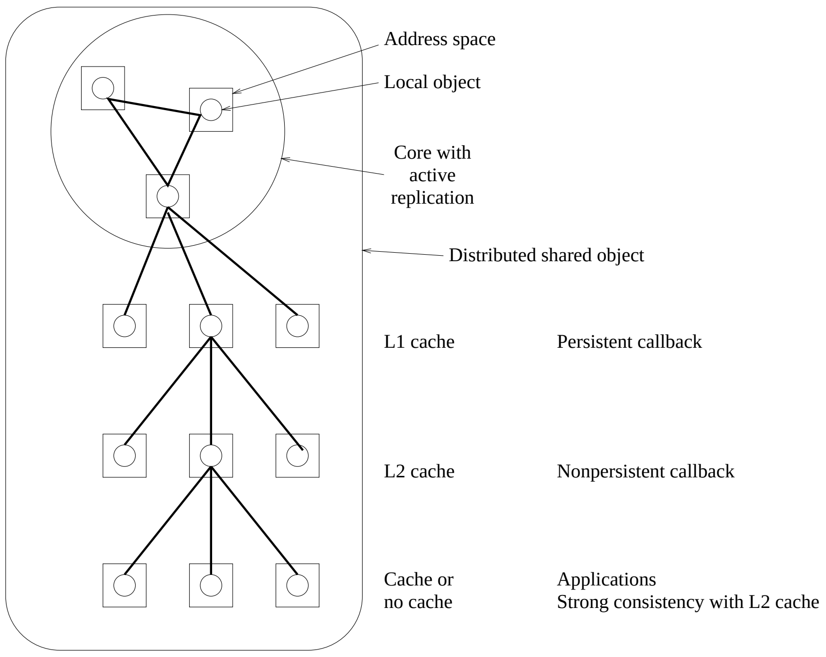


Figure 3.15. Core with caches

This model can be generalized into a model that groups the local objects of a single distributed shared object into three classes. These three classes are (see Figure 3.16):

1. Consistent read/write access
2. Consistent read access
3. Inconsistent (read) access

These three classes have been chosen to match three corresponding classes that deal with the security within an object (see Section 3.4.1).

Representatives in ring 1, the "core," can arbitrarily modify the state of the object. These representatives interact using the strongest consistency protocol available in the distributed shared object. A representative in ring 2 contains a copy of the state of the object. Update, or invalidate messages are sent to representatives in ring 2 by representatives in ring 1. A representative in ring 2 cannot change the state of the distributed shared object directly. Instead, a request to change the state of the object has to be sent to an representative that belongs to the core. Finally, representatives in ring 3, the "leaf nodes," do not contain a copy of the state of the object. Instead, all requests are forwarded to representatives in ring 1 or ring 2.

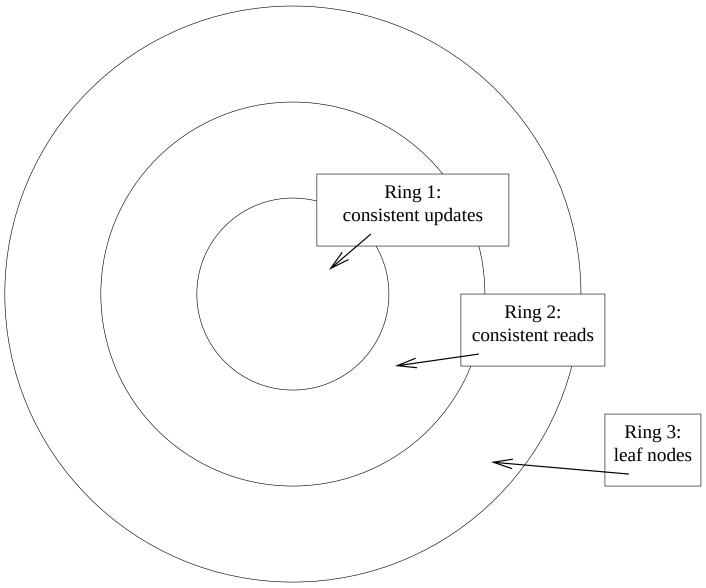


Figure 3.16. Replication rings within a single object

The relative size of the rings depends on the object. For example, a WWW page is typically modified on one WWW server and then copied (or cached) by other servers. Thus, it has a very small core. In contrast, a Usenet newsgroup is modified world wide by people posting new articles (with the exception of moderated newsgroups).

3.3 Persistence

This section describes the support for persistent distributed shared objects. A **persistent object** is an object that (after being created) continues to exist until it is explicitly destroyed.[5] Nonpersistent objects do not provide this guarantee: they are typically destroyed when not in use by any process.

Data can be made persistent at different levels of abstraction. Figure 3.17 shows the different levels of abstraction that exist in a Globe process. Each of those layers may provide a "persistence service." There is a trade-off between the different levels that can provide persistence. In general, lower levels are more convenient, are more transparent, and have better access to hardware and other resources. On the other hand, higher levels provide more control, and can better use the semantics of the data.

[5]Explicit is taken from the point of view of the object, that is, a global garbage collector explicitly destroys an object when it is no longer reachable.

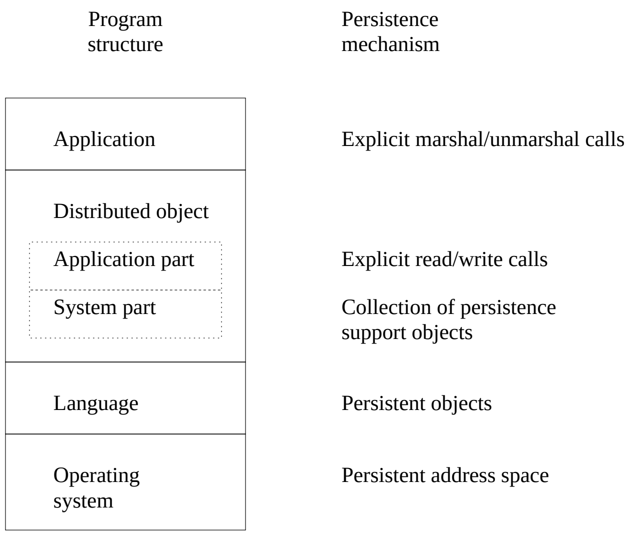


Figure 3.17. Persistence in different layers

In this thesis, we ignore persistence implemented by the application, the language or the underlying operating system, instead we focus on persistent distributed shared objects.

3.3.1 Persistent State

In general, objects are made persistent by storing the state of the object on some kind of nonvolatile medium such as a hard disk. Alternatively, persistence can also be implemented by maintaining enough replicas of the state of an object that at least one of them always survives a sequence of failures. We will ignore this last (theoretical) option; hard disk storage is cheap enough to assume that it is always used for persistence. The main issue is therefore how to store the state of an object onto a hard disk. For convenience, we assume that hard disk storage is represented by persistent state objects. A **persistent state object** is a local, system object that can be used to store some number of bytes persistently. Note that these objects typically use the file system provided by the underlying operating system. It is quite possible that such a file system is in fact a network file system mounted from a remote machine.

Figure 3.18 shows a classification of different approaches to make a distributed shared object persistent. There are two main issues (the dashed arrows in the figure): (1) what kind of mechanism is used to store the state of the distributed shared object persistently and (2) where the implementation of persistence is placed with respect to the local part of a distributed shared object.

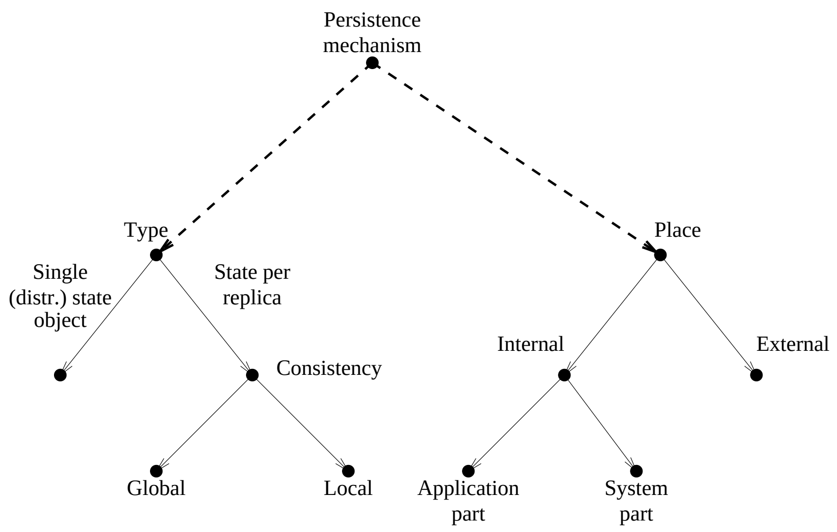


Figure 3.18. Classification of persistent distributed shared objects

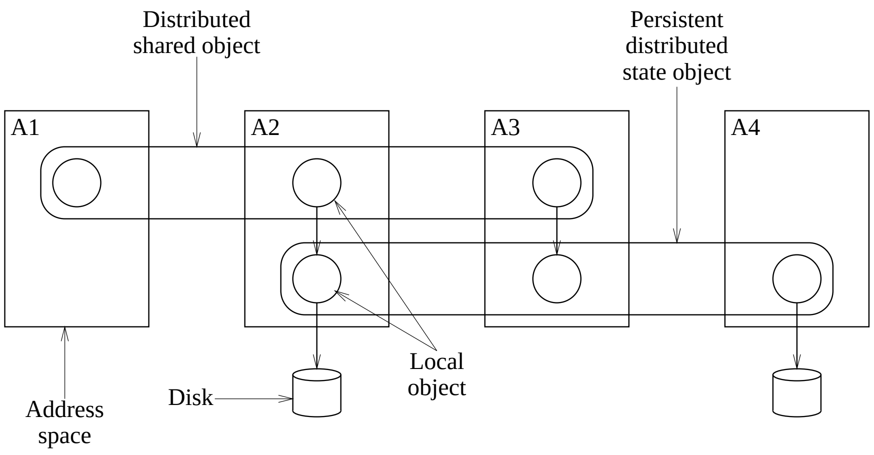

Distributed shared object
Persistent distributed state object
A1
A2
A3
A4
Local object
Disk
Address space

Figure 3.19. Persistence using a persistent distributed state object

The two basic mechanisms to store the persistent state of a distributed shared object are:

1. The system provides persistent *distributed* state objects. A **persistent distributed state object** is similar to persistent state objects in that it can be used to store a linear sequence of bytes persistently. However, these objects are also distributed shared objects and can be accessed from multiple address spaces.

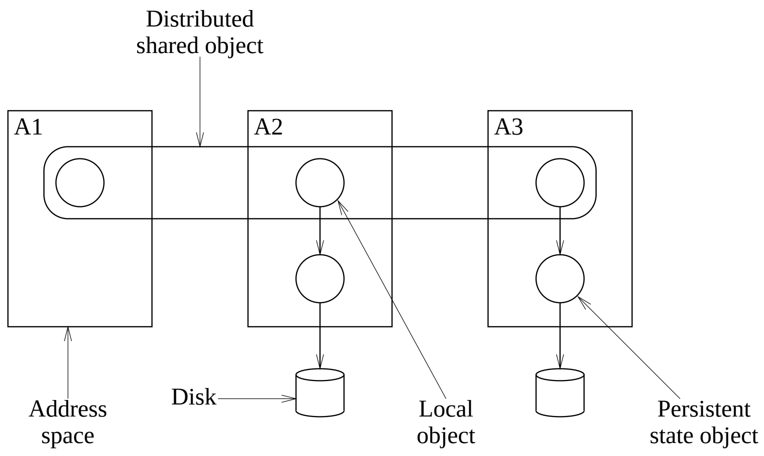


Figure 3.20. Persistence using persistent replicas

A persistent distributed shared object can store its state in a persistent distributed state object. This approach is illustrated in Figure 3.19. Address spaces A1, A2, and A3 contain local objects that belong to a single distributed shared object. This distributed shared object uses a persistent distributed state object, with local objects in address spaces A2, A3, and A4. Address spaces A2 and A4 have an attached local disk. Note that the local object in address space A1 does not have direct access to the persistent state. When necessary, it gets a copy of the state from its peers in A2 or A3. Likewise, the persistent distributed state object's local object in address space A3 does not have a local disk.

2. One or more representatives independently maintain a persistent copy of the distributed shared object's state. Those representatives provide *persistent replicas* and store the object's state in a (nondistributed) persistent state object. This approach is shown in Figure 3.20. The representatives in the address spaces A2 and A3 use a persistent state object to store their states. In turn, each persistent state object stores it state on a local hard disk.

The main differences between these two mechanisms are:

- When the persistent state is stored in a single distributed state object, it is necessary to use a (distributed) locking mechanism to allow multiple local parts of the distributed shared object to access the persistent state.

  In contrast, in the second mechanism, individual representatives are made persistent independently. Furthermore, the persistent state objects belongs to exactly one representative. Therefore no concurrency control is needed by a representative to access its persistent state.

- The first mechanism is closer to the traditional shared file system approach. In this approach, data is stored in files and interpreted by applications. All distribution aspects are hidden by the file system implementation. This mechanism provides some backward compatibility with existing files by allowing them to be encapsulated in a distributed shared object (i.e., the existing file is treated as a persistent distributed state object).

- A persistent distributed state object may be replicated internally, but this fact is hidden from the distributed shared object that uses the state object. The choice of replication algorithms for the persistent state object is limited because the semantics of the distributed shared object are not available to the state object.

  In contrast, with the second mechanism, it is the distributed shared object that is responsible for keeping multiple copies of the persistent state consistent. The distributed shared object can use whatever consistency algorithm is most suitable.

A persistent distributed shared object based on persistent replicas (the second mechanism), can choose between two broad categories of consistency guarantees for the persistent state of the entire object. Differences between those consistency guarantees are observed only in the presence of failures.

1. Global consistency.

   All persistent state objects are kept consistent with respect to each other.

2. Local consistency.

   The persistent state of a persistent replica is kept consistent only with respect to the volatile state of the replica and not with respect to the persistent state of other replicas.

For example, an airline reservation system needs to keep its persistent state objects globally consistent, even in the event of crashes. On the other hand, the persistent state of a Usenet newsgroup does not need to be globally consistent.

The implementation of the persistence support can be placed in different parts of a representative. The three alternatives are: (1) external to the existing representative, (2) internal to the existing representative, interfacing with system-provided parts of the object (replication or control object), or (3) internal, interfacing with the application specific parts of the object (semantics object).

External to the existing representative

Figure 3.21 shows the representative of a nonpersistent distributed shared object which has been extended externally with support for persistence. Added to a standard local object for a nonpersistent distributed shared object, are a persistence object and a

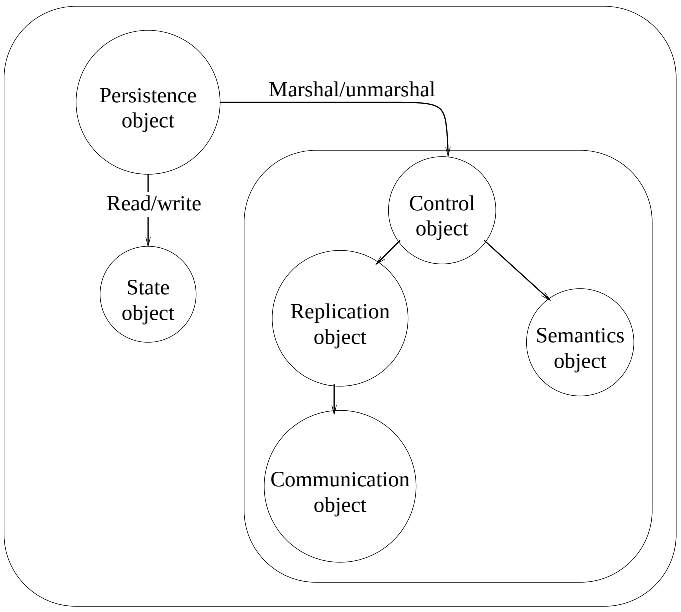


Figure 3.21. Persistence, external to a distributed shared objects

persistent state object. The **persistence object** uses the persistent state object to store the state of the replica it encapsulates. This state is downloaded into the representative by invoking an unmarshal method.

The first disadvantage of this approach is that the persistence object knows nothing about the replication strategy used by the distributed shared object. A second disadvantage is that the persistence object can save and restore only the complete state of the local object. This situation can be improved by extending the interface beyond just marshal and unmarshal. However, this approach remains inflexible.

Internal, interfacing with system-provided parts of the representative

In the second approach, support for persistence can be tightly integrated with the replication protocol used. A drawback is that the persistence part still reads and writes the entire state of the semantics object.

Internal, interfacing with application specific parts of the representative

The third approach is to let the application programmer explicitly deal with saving and restoring the state of the semantics object. This allows for object specific solutions (i.e., no need to read the entire state of the semantics object to execute an operation that needs only part of the state). A disadvantage is the lack of integration with the replication object.

The last approach is useful when the semantics objects only communicate through a shared, persistent state object (i.e., without the use of a replication and communication object). The semantics objects operate by manipulating the shared state provided by the persistent state object. This approach is practical when the semantics object is only a thin layer of interpretation on top of some shared state.

3.3.2 Object Architecture

The persistence architecture in Globe provides persistent replicas and the implementation of persistence is integrated with the replication object ("state per replica" and "internal/system part" in the classification of Figure 3.18). Two advantages of this approach are that it isolates the application programmer from persistence management and that it supports tight integration of persistence management with replication protocols.

A persistent replica can be in one of two states: it can be active or passive. **Active replicas** are similar to (nonpersistent) representatives: they have a local object in some address space that can communicate with other parts of the distributed shared object. In contrast, A **passive replica** consists of only the persistent state.

Before a passive replica can be used, it has to be activated. To support activating passive replicas and passivating active replicas, we need a persistence manager object. A **persistence manager object** activates a passive replica in response to some external events, such as a bind operation from another address space. An active replica can be passivated when it is idle. A persistence manager object is typically associated with an address space. Multiple persistent replicas, belonging to different distributed shared objects can share a single address space when they are active. These replicas are managed by a single manager.

Figure 3.22 shows the representatives of a persistent distributed shared object. There are three kinds of representatives: nonpersistent representatives, persistent replicas that are active and persistent replicas that are passive. In most cases, the persistent replicas form the core of a distributed shared object with nonpersistent representatives for client address spaces and (volatile) caches. In Figure 3.15, the replicas in the core and the L1 cache would be persistent.

In general, a persistence manager supports multiple persistent replicas that belong to different distributed shared objects. Figure 3.23 shows a single address space with a manager object. For passive replicas, the manager simply holds a reference[6] to a persistent state object that contains the state of the replica.

[6] In this context a reference is typically the name of a local state object. The manager binds to the state object when it activates a persistent replica.

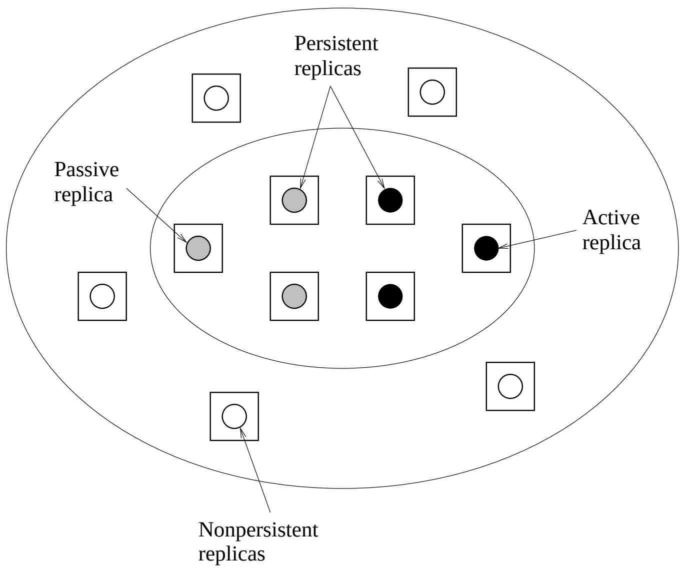


Figure 3.22. Persistence architecture of a distributed shared object

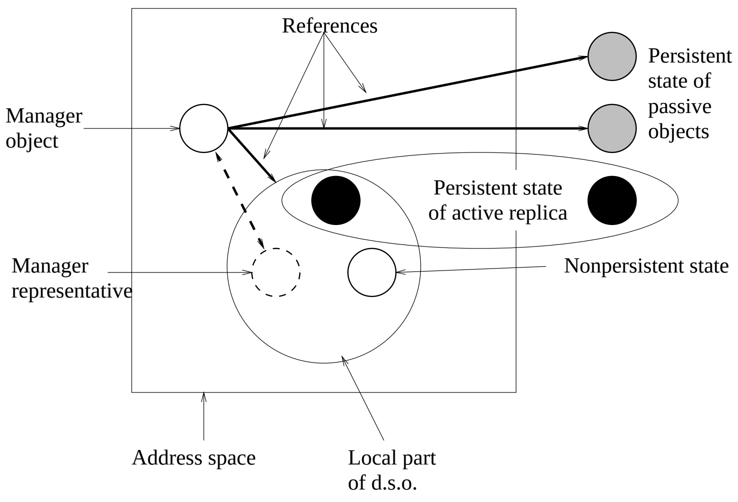


Figure 3.23. A persistence manager process

Figure 3.23 also shows one active replica. An active replica consists of three parts: (1) the persistent state object that contains the persistent state of the replica, (2) the nonpersistent representative (typically, the four basic objects) and, (3) an object that represents the manager object. The persistence implementation in a representative uses the state object to initialize the semantics object. The object that represents the manager is informed when the replica becomes idle.

3.3.3 Binding to a Persistent Distributed Shared Object

Binding to a persistent replica is a two step process. The basic idea is that the persistence manager provides contact points for the persistent replicas. A new (nonpersistent) local object in a client address space sets up a connection to the contact point, thus establishing a connection with the manager object. The manager object then activates the replica and redirects the binding party to a newly created contact point in the replica. Redirection means that the manager provides the local object in the client address space with the contact address associated with the new contact point.

The connection to the manager object is terminated and a new connection to the replica is established. Various optimizations on this scheme are possible (and necessary to reduce latency). For example, the manager object can pass the connection to the replica instead of redirecting the client to a new contact point. Note that the persistent replica can publish this new contact address (in the location service part of the naming system), to allow further bind requests to contact the replica directly. Before the replica is passivated, it should delete the contact address.

This redirection is shown in Figure 3.24. A manager object M holds references (local object names) to the persistent state of four replicas of four different distributed shared objects (S1 . . . S4). The manager object also provides four contact points for binding to those replicas. In Figure 3.24(a), the new local object L1 in address space A1 sets up a connection to the contact point in the manager object. In the second step, shown in Figure 3.24(b), the persistence manager has created object L2. Object L2 creates a new contact point and uses the persistent state object S2 to initialize its semantics object. L1 sets up a connection to L2 just like it would do when binding to a nonpersistent representative.

After a while, L1 may drop the connection to L2. If no connections to L2 remain, L2 will invoke a method on the representative of the manager object to inform the manager that it has become “idle.” The manager may decide to passivate L2.

So far, the description implicitly assumes that the manager object shares its address space with all active objects. The manager may activate passive replicas in separate address spaces for two reasons. The first reason is efficiency. Too many objects in one address space may lead to huge address spaces, too many threads to be scheduled efficiently, lock contention, etc. The second reason is fault isolation. A single manager may support different types of objects and objects belonging to different users. Due to the lack of protection *within* a single address space, a malfunctioning object may harm objects belonging to different users.

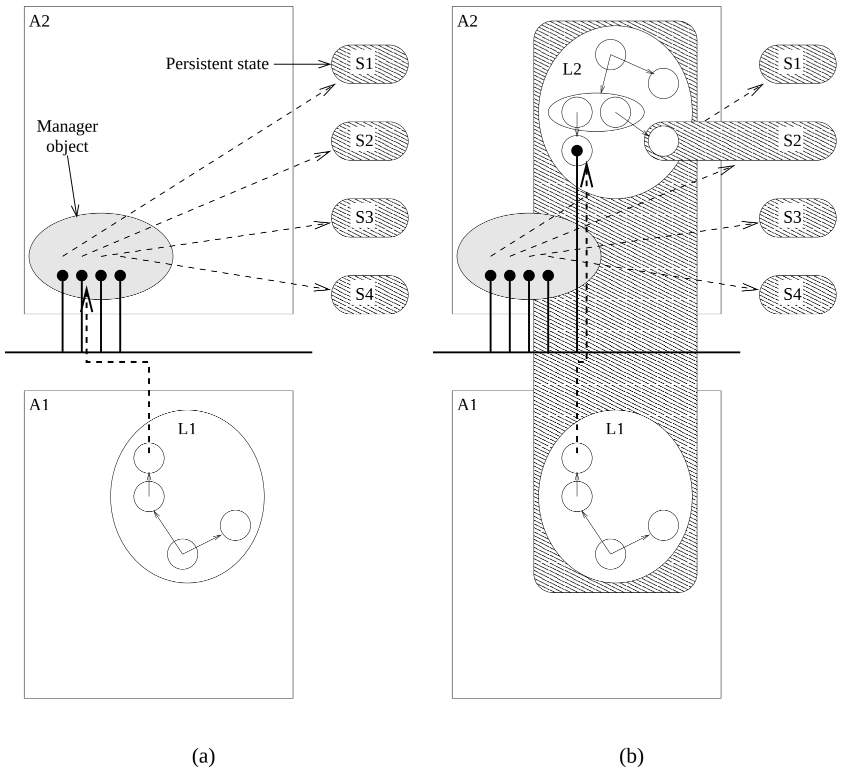


Figure 3.24. Binding to a persistent distributed shared object

Alternatively, it is possible to create multiple managers, each running in separate address spaces. Note that the new contact address returned by the manager can refer to a contact point in a different address space.

Figure 3.25 shows the architecture of the representative of a persistent distributed shared object. The replication object is replaced by the combination of a replication and a persistence object. The replication object informs the persistence object about suitable moments to update the persistent copy of the state of the object.

The persistence object uses a persistent state object to store the state of the semantics object. The persistent state object can be a local state object, or it can be a distributed shared object itself. The persistence object also communicates with the manager object during initialization and to coordinate shutdown.

This architecture is a relatively straightforward extension of the four-object architecture described in the previous section. This architecture becomes recursive when the state object is a distributed state object: the persistent distributed object uses persistent replicas which are implemented using persistent distributed shared objects. The

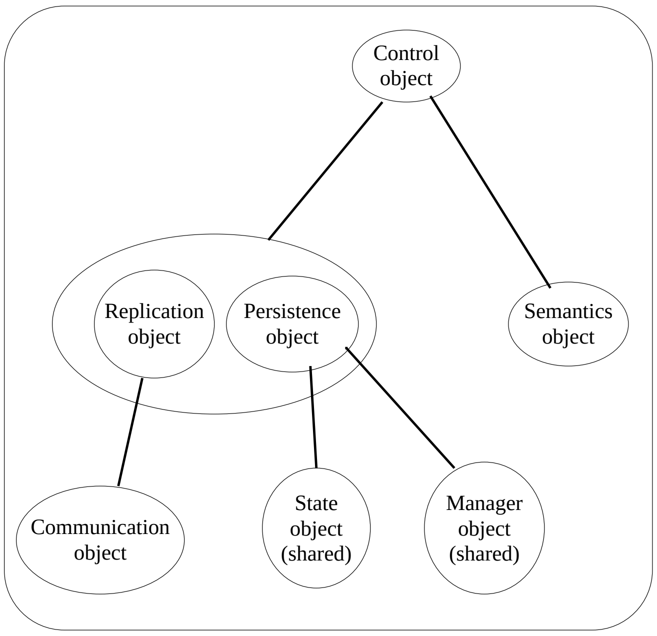


Figure 3.25. Persistent representative of a distributed shared object

solution is to bootstrap persistence by implementing persistent state objects directly using the underlying operating system's file system or using direct access to a hard disk.

Contact Addresses of Persistent Distributed Shared Objects The structure of a contact address provided by a persistent distributed shared object depends on the mechanism used to store the persistent state. An object that uses a distributed state object to store its state has to provide contact addresses that refer to the distributed state object. Similarly, a contact address for a persistent replica refers to the object manager that can activate the replica.

The contact addresses of a distributed shared object that stores its persistent state in a single distributed state object are derived from the contact addresses provided by the distributed state object. The contact addresses for the persistent state object contain a protocol identifier that refers to a (standard) protocol to access state objects. Those derived contact addresses contain a different protocol identifier that refers to a class of local objects that can operate on the contents of the state object. Note that the distributed shared object may offer additional contact points when it is active, to allow binding without using the persistent state object.

A problem with this approach is that a persistent distributed state object is an independent distributed shared object, which may create additional replicas or migrate replicas without informing the distributed shared object that uses the persistent state object. If the distributed shared object simply copies the contact addresses of the state object, it has to keep this set of contact addresses consistent.

This problem can be solved by using an object handle as a contact address. Using an object handle as a contact address introduces an extra level of indirection that isolates the distributed shared object from contact addresses of the state object. This kind of contact address is called a **persistent contact address**.

The contact address for a persistent replica refers to a contact point provided by the manager object and some additional demultiplexing information to identify the proper replica. In this case the contact address for the persistent replica is derived from a contact address provided by the manager object.

The problem with the use of a contact address of an object manager is similar to the problem caused by using a contact address of a persistent distributed state object: a manager can change its contact addresses without informing the persistent replica that uses the manager. This problem can also be solved using persistent contact addresses.

For the purposes of binding, a manager object can be treated as an independent distributed shared object. By binding to the manager object (instead of using an actual contact address and setting up a connection directly) the manager object gets additional flexibility to distribute the replicas it manages over different address spaces.

The main disadvantage of the use of persistent contact addresses is the extra overhead that is introduced. Some of the extra overhead can be avoided if the naming system returns a contact address for the embedded object handle along with the contact address that contains the object handle. This technique is used by the Domain Name System and is called “additional information.”

3.4 Object Security

The last part of the distributed object architecture is the security architecture. The first part of this section describes how to secure communication within an a single distributed shared object. The second part of this section provides a model for authentication and access control.

A distributed object is called **secure** if it can defend itself against security attacks.[7] A secure object can resist (1) unauthorized changes of its state, invalid replies to remote operations, (2) updates to the state of the object without an audit trail[8], or (3)

[7] Only distributed shared objects can be made secure. We assume that a *local* object cannot defend itself against attacks from within the process that the object is part of.

[8] An audit trail is a set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions [Abrams et al., 1995].

passing the state of the object, or a request or reply message to unauthorized parties. Additionally, we would like to prevent denial of service attacks and traffic analysis. Depending on the object, it may not implement one or more of those features.

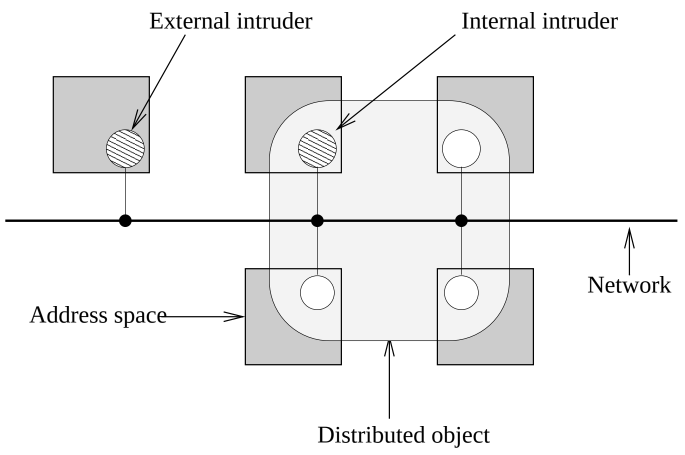


Figure 3.26. Attacks on a distributed shared object

From the point of view of a distributed shared object, we recognize two kinds of intruders. **External intruders** are completely outside and are not part of the distributed shared object. On the other hand, **internal intruders** try to compromise the security of the object from the inside. Figure 3.26 shows the positions of both an external and an internal intruder.

This distinction between internal and external intruders is different from more traditional security architectures that distinguish clients and servers. In general, a client has to verify that it is communicating with an authentic server and not with an imposter (someone who assumes a false identity for the purpose of deception). A server verifies whether a client is authorized to make certain requests. In contrast, an external intruder is not part of the distributed shared object at all (it is neither a client nor a server). An internal intruder does not behave according to its role within a distributed shared object. For example, a client may pretend to be a server. The distinction between internal and external intruders is necessary because some distributed shared objects use a symmetric replication protocol, for example, a replication protocol based on active replication.

In the case of an external intruder, the distributed object as a whole tries to protect itself from the intruder. In contrast, the presence of an internal intruder requires the individual representatives in different address spaces to protect themselves from the intruder. An example of an internal intruder is an address space with read-only access to a distributed shared object that tries to modify the state of the object by issuing malicious state-update methods.

The security architecture for distributed shared objects is based on the assumption that an address space is a single protection domain. In theory, all executable code running in a single address space can access the data structures of all (local) objects in that address space. With this assumption, we can reformulate the attacks in terms of communication between address spaces. Firstly, an attack from a external intruder can be prevented only by discriminating between communication between address spaces that are part of the distributed shared object and those that are not. Secondly, an attack from an internal intruder can be prevented only by labeling the rights of the individual representatives within a distributed shared object.

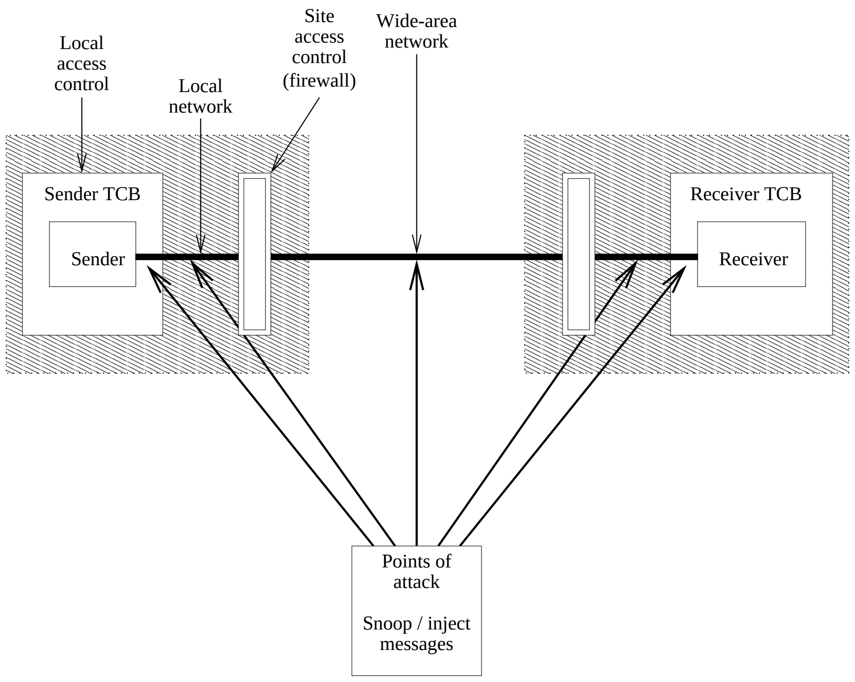


Figure 3.27. Low-level security

Figure 3.27 presents a low-level view on sending data from one address space to another. We look at a message being sent from a sender to a receiver. In an RPC, the request message and the reply message are different messages with a different sender and receiver. Note that data in this context can be information, but it can also be a request to perform a certain operation.

Both the sender and the receive have a trusted computing base. The **trusted computing base** (TCB) of a process (or a user) is that part of a computer system that has to be trusted. The minimal TCB for a process running on a UNIX workstation is:

- Everything in the same process.

- Every process that runs under the same user ID.
- Every superuser process on the same machine.
- The operating system kernel.
- The immediate physical environment of the workstation.

Of course, the TCB may be much larger, including, for example, other (local) computers and system administrators. Everything in the TCB should be cooperating to protect (or at least, not to violate) the sender's or the receiver's security.

The sender's and receiver's TCBs may perform access control. For the sender, the TCB decides whether it is safe/allowed to send this data to a particular destination. The receiver's TCB determines whether or not the received data can be trusted.

Note that in many cases the TCB is controlled by the user. For example, most users of a personal computer can install a new operating system on their computer. This allows them to determine what the TCB should look like. However, the sender and the receiver may be part of larger organizations that enforce their own security policy, independent of the sender's or the receiver's wishes. This is shown as Figure 3.27 as "Site access control." In practice this kind of functionality is implemented using firewalls. A **firewall** examines all packets and/or network connections and decides whether they are allowed according to the site's security policy. Firewalls can operate at different levels of a protocol stack. At the network layer, a firewall may pass or reject packets for specific hosts or networks or packets for certain transport protocols. At the application level, a firewall may deny, for example, request to modify a database and allow requests to read the database.

From a security point of view, a packet may travel over three different kinds of networks: the sender's local network, a wide-area network, and the receiver's local network. Note that in this model, we assume that a wide-area network is controlled by an untrusted third party. Any networks that are wide-area networks from a network point of view and that are trusted by either the sender or the receiver are assumed to be part of respectively the sender's or receiver's local network. This model gives an intruder five different places to attack: the sender's TCB, the sender's local network, the wide-area network, the receiver's local network and the receiver's TCB.

In addition to the five different places to attack, an intruder has the choice between active or passive attacks. At the network level, an intruder can look at data transmitted over the network, this is called a **passive attack**, or the intruder can change packets that are in transit, inject old packets (replay) or generate completely new packets. This is called an **active attack**.

From the point of view of an intruder, the most powerful network attack is a combination of passive attacks on the sender's local network with active attacks of the receiver's local network. Remember that in this model, communication is unidirectional: an RPC uses two different senders (client sends the request and the server sends the reply) and two receivers. A passive attack on the sender's local network allows the intruder to capture requests, replies, or state transfers depending on the role of the sender. An active attack on the receiver's local network provides the opportunity to change the state of the object (spoofed request message), or to provide false information to a client.

An attack from the wide-area network means that the intruder is limited by the security policy enforced by the sender's and receiver's firewalls. Sometimes an intruder can attack a TCB. This can be done, for example, by changing the access control information stored in the TCB, by attacks on address spaces that are part of the TCB, but also by direct physical attacks. Another kind of attack is distrust among the communicating parties themselves (repudiation of an action). For example, a client may deny buying some stock after a collapse of the stock price.

The basic mechanisms to protect against attacks are: encryption, signed and authenticated messages, access control lists, and digital signatures for nonrepudiation.

A problem with combining firewalls that operate at the application level and distributed shared objects is that the firewall and the distributed shared objects have to agree on a common communication protocol that allows the firewall to filter traffic and to provide access control. This reduces the freedom of the object designers.

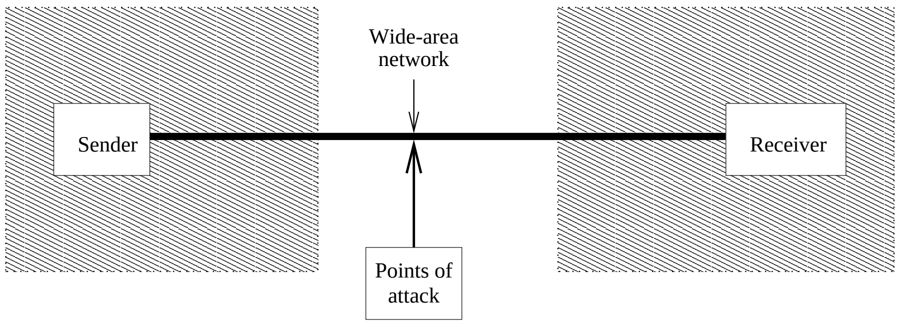


Figure 3.28. Simplified low-level security

In many cases, site access control and TCBs are not explicitly recognized. In this case the low-level security model reduces to a sender who sends a message to a receiver over a communication channel. This is shown Figure 3.28.

When multiple objects are part of a single TCB, then each object can violate the security of the other objects. Each of the two distributed shared objects in Figure 3.29 can compromise the security of the other distributed shared object.

It is important to note that in our model, low-level security is hidden inside of an object. The sender and receiver of Figure 3.27 are part of the same object.

The protection of the distributed shared object as a whole is provided by the individual representatives that are part of the distributed shared object. This means the

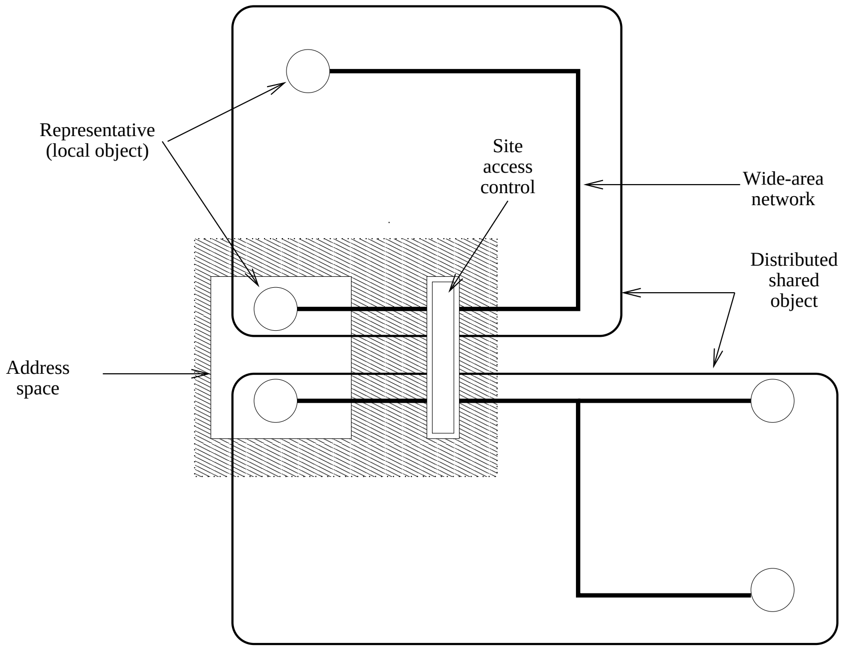


Figure 3.29. A shared address space

security implementation for a distributed shared object is translated into checks and encryption performed by individual representatives. The reason for this model is that ultimately, the security of the entire distributed shared object depends on signed, encrypted messages sent between the local parts of the distributed shared object.

If we extend the basic architecture for implementing a local part of a distributed shared object, we get Figure 3.30. Two new objects are added: a security object and a principal object. In addition, the communication object is extended to provide secure communication channels. These secure channels provide secrecy and basic integrity mechanisms. Sometimes (for nonrepudiation, or in multicast communication channels) additional signatures are needed. The main function of the **security object** is access control. The communication, replication, and control objects ask the security object whether a particular operation (allocation of resources, accepting a message, invoking a method on the semantics object) should be carried out or not. The **principal object**[9] assists the security object with key management, and site-wide security policies.

[9]The term “principal” will be explained in the next section.

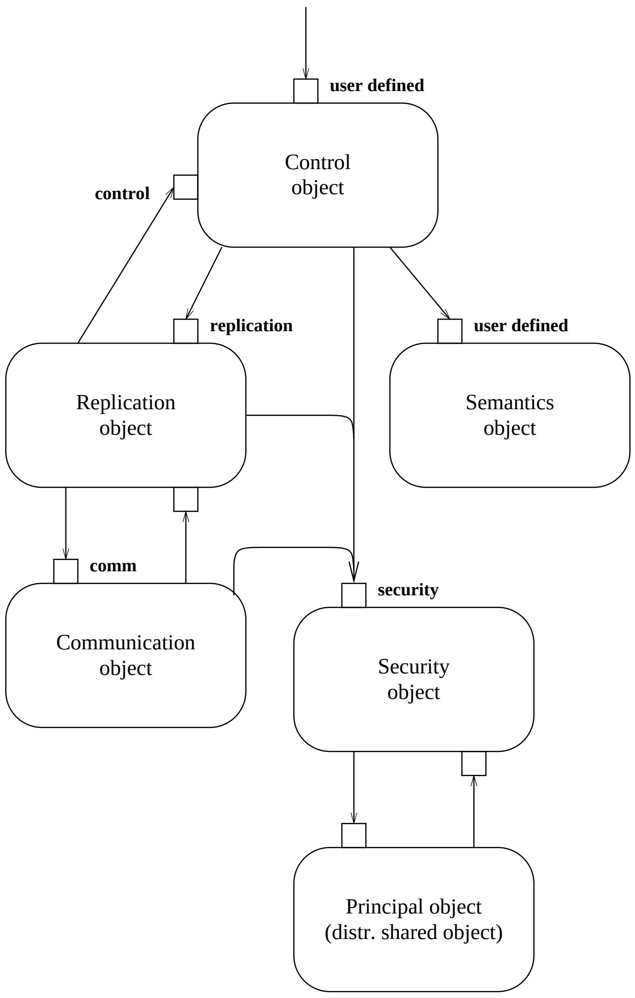


Figure 3.30. Local part of a secure distributed object

Authentication and access control is performed when a communication channel is established or when a process joins a multicast communication group. The communication object passes the identity of the sender's principal along with a message to the replication object.

Requests can be classified as follows: (1) operations on the semantics object, (2) control messages of the replication protocol, (3) operations on the state of the security object (for example an access control list, or a list of valid capabilities), and (4) other operations for managing the object (for example persistence).

Within the object architecture shown in Figure 3.30, the replication object is the most suitable place to perform access control. Performing access control means taking an access control decision and (optionally) returning an error to the sender if the decision was negative. The actual access control decision is typically delegated to the security object. The replication object has detailed knowledge about the source of the message, the replication protocol and the kind of operation requested (read-only, read/write, etc). The replication object is the only object that has access to this information. The communication object is too low-level, it does not recognize different message types relating to replication, method invocations, etc. The control object and the semantics object cannot deal with messages that are part of the replication protocol.

In theory, it is more efficient to download access control information once into the replication object, than to have the replication object call the security object for each message that arrives. However, this distributes security functionality over multiple objects, which complicates the analysis of the security of the system.

A problem with providing access control in the replication object (or in the security object for that matter), is that the actual decoding of the arguments of an incoming method invocation is done by the control object. This means, for example, that the replication object *can* control who can withdraw money from a bank account object, but *cannot* make a distinction between withdrawing small or large amounts of money. Moving the application specific part of the access control to the semantics object might be a solution to this problem. However, the network identity of the caller is not passed by the replication to the control object. It means that the control object interface and the interface provided by the semantics object has to be extended with extra arguments for the identity of the caller.

The security object uses the principal object to deal with (site wide) security policies. Typically, the actual access control for an object is based on access control information stored in the object itself, combined with global security policies installed by the user or the system administration of a particular site. One such policy might be that all information leaving the site should go through a firewall.

In general, we expect that a process binds to its owner's principal object soon after the process is created. The necessary authentication information that is needed to bind to the object is provided by the creator of the process. Alternatively, the process can be a login process and executes a login protocol. These protocols will be described in the next section. Note that after the process has bound to its first principal object, it can use that principal object to bind to other principal objects. A process may use multiple principal objects to be able to act on behalf of multiple users.

The control object verifies (by calling the security object) that operations invoked by the user of the object (as opposed to requests arriving from the replication object) can be executed. If the control object would omit this check, then the operation might fail because messages sent to other parts of the object may be rejected by remote replication objects.

3.4.1 Authentication and Access Control

The basis for authentication and access control is a security policy. A **security policy** describes what resources should be protected, who should be allowed to access those resources, etc. A security policy is based on a real world security problem (keeping information secret, privacy, preventing thefts, etc.) that interacts with computer systems. The of goal of a security model for a worldwide system such as Globe, is to be able to support as many different security policies as possible.

In the first part of this section, we provided the low level security mechanisms in a bottom up approach. This section will start at a high level and work toward the low level security mechanisms.

Principals

We use the following terminology. **Principals** are allowed to act; computers act on *behalf* of principals. A principal is a person, a collection of persons (a group), or a specific role within an organization. In the system we can create complex derived principals such as organizations, companies, universities, departments, etc. Principals with certain roles, for example a head of a department, get additional rights. A principal is given certain rights but also bears responsibilities.

We need a way to represent principals in the system. In most systems, principal identifiers are separate from object identifiers. For example, in UNIX, user identifiers and group identifiers are small integers. Real file object identifiers are not used very much under UNIX, but consist of a device identifier plus an inode number.

The advantages of associating a distributed shared object with every principal are the following:

- Object identifiers can be used to identify principals.
- The object that represents the principal can store information about the principal. For example his public key certificates.
- An object that represents a group principal can store the membership list of the group.
- A principal can allow other principals to act on behalf of itself. The object can store who may act on behalf of the principal. For example, a specific role such as head of a department may be delegated to different principals at different times.

A security system based on principals associates a person with every action. In many real-world situations, this is not the case. For example, the keys that are used to unlock a door do not always identify the person that entered through the door. Money allows a person to enter a theater without providing his identity. Both money and anonymous capabilities (the computer equivalent of keys) can be introduced as special principals.

A principal may wish to create subprincipals for different roles (at home, at work, playing a game, hobbies). These subprincipals should have a subset of the rights of the principal. For example, while playing a game, you do not want the game to be able to sign a check to order more games automatically.

Access Control

The two basic models for access control are mandatory access control and discretionary access control. In a system that uses **discretionary access control**, the owner of an object specifies which principals have access to the object. The system grants or denies access based on access control information provided by the object's owner.

Mandatory access control means that the system itself can also impose restrictions on the use of an object. For example, mandatory access control is used by the military to implement the policy that secret documents can only be read by people with a high enough security clearance. Furthermore, if somebody with a top-secret security level modifies a secret document, it becomes top-secret. The best known military security model is the Bell-LaPadula model [Bell and LaPadula, 1973].

Mandatory access control is typically implemented by associating a security label with every data item in the system. When secret information is stored in an object with a low security classification, the system increases the security classification of the object to match the security label of the data to be stored in the object. With discretionary access control the system protects access to an object independent of the contents of the object.

One of the drawbacks of the Bell-LaPadula model is need for declassification. Normally, the security label of a data item can only go up. However, this does not allow a general to address his troops. The approach taken in [Myers and Liskov, 1997] is to allow the owner of a piece of information to declassify it. All others are still prevented from violating the mandatory access control policy.

The main drawback of mandatory access control in a worldwide distributed system is the need for a trusted computer system to enforce this access control. In a system that consists of independent users and organizations, it is impossible to guarantee that everybody obeys all access control rules. For this reason, we will focus on discretionary access control.

An **access control matrix** is a simple way to model discretionary access control in a system. The principals are listed along one axis of the matrix, and the objects are listed along the other axis. The contents of the matrix are the operations a principal is allowed to perform on an object. A principal may have no access, read-only access, write-only access, read/write access, etc.

To model an access control matrix in an object based system, it is necessary to create a mapping between the operations that can be listed in the matrix and the methods implemented by the objects. This can be done in two different ways. One approach is to have a one-to-one correspondence between operations and methods. Effectively, the set of all possible operations in the matrix is the union of all methods provided by different objects. The second approach is to have a standard set of operations, for example read, write, and append, and map the methods of an object onto those operations.

The advantage of the first approach is that it makes fine-grained access control possible. The main disadvantage of the first approach is that it makes analyzing the security of a system harder. For example, it is in general not possible to compute which principals can modify the state of an object if it is not known which methods are read-only and which ones modify the state. The opposite is true for the second approach. Analyzing the security of the system becomes easier, but access control will be more coarse-grained. The choice between the two approaches requires further research.

Two obvious ways to distribute the access control matrix are: (1) store the list of principals that have access to an object along with the object itself. This is called an **access control list**. (2) Provide a principal with a set of capabilities that provide access to the various objects the principal is allowed to access. A **capability** is a protected identifier that both identifies a object and specifies access rights to be allowed to the accessor who possesses the capability [Abrams et al., 1995].

Advantages of access control lists over capabilities are:

- In a distributed system, capabilities suffer from the so-called confinement problem: it is hard to prevent a capability from being passed from one principal to another without the object's approval.

- In a capability-based system, a principal has to manage his capabilities. This presents two problems:

    1. An additional security risks because it provides additional ways to attack an object.
    2. When a principal has access to a large number of objects, it becomes necessary to structure the name space in which the capabilities are stored.

The advantages of capabilities are:

- Access control lists may become large if a large number of principals has access to an object.

  This problem can be reduced by the introduction of groups. In UNIX file systems, the access control "list" contains three entries: one for a principal (called the owner of the object), one entry for a group, and one entry for all other principals in the system.

- An operation on an object with an access control list cannot be anonymous. In capability-based system, authentication of the acting principal is independent of access control.

Finally, off-line security analysis for a system based on access control lists is different from a system based on capabilities. In a wide-area system that uses access control lists, it is relatively easy to compute which principals have access to an object. It is much harder to compute the set of objects that a specific principal has access to. In a capability-based system, the reverse is true.

The confinement problem for capabilities can be solved in centralized systems but is very hard to solve in a wide-area system with independent administrative organizations. For example, ICAP solves the confinement problem by adding identities to capabilities [Gong, 1989]. Due the lack of confinement in pure capability based systems, we mainly focus on access control lists. However, we can always augment access control lists with capabilities, for example, for performance reasons.

Access Control Predicates

As described above, an access control list stores the access rights of the principals that are allowed to access an object. This approach has two limitations: the number of principals that is allowed to access the object can be very large, and sometimes the rights of a principal depend on other factors such as the time of the day (access only during working hours). The first problem can be solved with the introduction of a group principal that represents a collection of other principals. The second problem can be solved by generalizing access control lists into access control predicates. An **access control predicate** (ACP) uses logical propositions to combine authentication information with other sources such as the current time, the machine issuing the request, etc. Note that in theory, an ACP can also include restrictions on parameters that can be passed in a method invocation, or even depend on the current value of the object's state. However, in the current object security architecture, access control checks are executed by the replication object before the arguments are unmarshaled. The replication object also sees the state of the object only in marshaled form. This means that restrictions on arguments, or on the state of the object cannot be enforced. To support restrictions on parameters, the architecture has to be extended to execute application specific access checks in the semantics object.

We can implement an access control predicate as an ordered set of tuples, where each tuple adds or deletes access rights for a set of principals if the predicate specified

Op	Principals	Predicate	Rights
+	Ed, Kees, Ruud	true	Read, write
+	Helpdesk	time $>$ 8am and time $<$ 6pm	Read, write

(a)

Op	Principals	Predicate	Rights
include " . . . /acp/sysadmin"			
+	*	true	Send
+	owner	true	Read, write
–	spammer	true	Send

(b)

Table 3.3. Example access control predicates: (a) system administrator ACP, (b) mailbox ACP

in the tuple is true. Elements in the predicate are statements about the environment of the object, such as the current time, or the location of the process that tries to gain access. Using a list of tuples also allows an object to use common access control information with additional object-specific refinements. This can be implement by first processing a common list of tuples, followed by the tuples stored in the object itself.

Table 3.3 shows two examples of access control predicates. The first column contains a plus sign (+) when rights are added, and a minus sign (–) when rights are removed. The second column lists the set of principals that a granted or denied any rights. The third column specifies when the entry is valid, and the fourth column lists the rights that should are or removed.

The first ACP defines a group of system administrators. Three people (Ed, Kees, and Ruud) are listed explicitly, and have full access. In this example the access right "read" should be interpreted as the set of all methods (implemented by the distributed shared object) that do not modify the state of the object; all other methods require the "write" access right. Read and write access should allow any method to be invoked. The second entry allows members of the group "helpdesk" access during office hours. An ACP does not distinguish between principals and groups of principals. The ACP just contains the object handles of the relevant principal objects (in this case, for Ed, Kees, Ruud, and Helpdesk).

The second ACP first includes the system administration ACP. The object specific entries grant everybody the right to send mail to the mailbox ("*"). The owner of the mailbox is granted full access. Lastly, a group of principals that are known to send spam are denied access.

Authentication

We need authentication mechanisms to verify that a particular process acts on behalf of a specific principal. When a process tries to bind to a distributed shared object, it provides authentication information to the object. The object uses this information to compute the access rights for the binding process. The reverse is also possible: during binding the distributed shared object provides its identity to the binding process. The process verifies that it is binding to the real object and not to an imposter.

Popular authentication mechanisms are:

- Mechanisms based on secret key encryption.

  The basic idea is to use a trusted key server that is shared between communicating parties. Each party shares a key with the key server. Both parties indicate to the key server that they would like to set up an encrypted communication channel. The key server generates a new session key and returns this key to them, along with the identity of each party's peer. The best-known example of this approach is Kerberos.

- Mechanisms based on public key encryption

  Public key encryption is based on the idea that every principal has a private key that is kept secret and a public key that is made public. A certification authority publishes signed statements called certificates that link a particular public key to a principal. Using these certificates it is sufficient for the party that needs to provide its identity to encrypt a message with his private key. The recipient decrypts the message with the public key listed in the certificate.

  The best known public key algorithm is RSA [Rivest et al., 1978]. X.509 [ITU, 1997] is a standard for certificates. X.509 allows only relatively simple certificates. More general certificates that contain logical expressions and multiple principals are described in [Rivest and Lampson, 1996]

  A serious problem with certificates is how to revoke them when a private key has been stolen. Two common approaches are to use a certificates with a limited life time or the use of revocation lists. A disadvantage of the first approach is that the certification authority has to constantly generate large numbers of new certificates to replace expired ones. A certification authority that generates new certificates on-line is relatively easy to compromise. Generating certificates off-line is more secure but is complicated if new certificates have to be generated to replace expired ones. The main problem with the second approach is the timely distribution of updates to the revocation list to all parties that use certificates, in the case of Globe this includes at least all contact points provided by distributed shared objects.

Note that in both cases the user remains vulnerable for some period of time. In the first case, the stolen key can be used until the certificate has expired and in the second case, the key can be used until the new revocation list is installed everywhere. Both approaches can be combined to reduce the size of revocation lists: a key can be deleted from the revocation list after the associated certificate has expired.

One approach to prevent private keys from being stolen is to store private keys that are valid for a long time in a vault. These long term private keys are used to sign certificates for online keys with a much shorter expiration date. These online keys are in turn used to generate private keys for login sessions. Individual processes may get their own short term keys.

Some mechanisms, such as capabilities and digital money, can be used for authorization without authentication. In other words, access to a distributed shared object is granted without establishing the identity of the user who issues the request.

- Capabilities

  A capability is a special ticket provided by an object that allows a process that holds the ticket to access the object. In contrast to the two previous mechanisms, this approach is not based on principals. Instead, every process that holds the right "key" is allowed to access the object.

  Capabilities can be implemented in two different ways. In centralized systems they can be implemented in the kernel or even in dedicated hardware. In a (wide-area) distributed system, capabilities have to be protected cryptographically. Amoeba uses a secure hash-function to compute a checksum value which is stored in the capability [Tanenbaum et al., 1986]. Any change to a capability will invalidate the checksum. The secure hash-function, which is keyed with a per object random number ensures that it is infeasible to compute the required checksum for the modified capability.

  Kernel-managed capabilities are not very useful in a wide-area system. The "kernel" has to be trusted by the object, and has to be easily accessible by a process that tries to invoke a method on the object. This situation is rare in a wide-area distributed system.

  The main problem with cryptographically protected capabilities is the confinement problem: it is too easy to pass a capability from one process to another. The object can no longer control who is allowed to access the object, other than by invalidating the capability. Capabilities with a limited lifetime can be useful to speed up access to an object. The idea is to authenticate using some other mechanism. After the authentication phase, the object returns a capability with a limited lifetime that also provides access to the object [van Doorn et al., 1999].

Note that in this case, confinement is less problematic because objects are free to expire capabilities at any time (forcing the processes that are using them to authenticate using other mechanisms).

- Digital money

  Digital money can be seen as an anonymous authentication mechanism that allows access in exchange for some amount of money. Many forms of digital money exist ranging from true digital "coins" to secure credit card transactions. Most governments try to prevent the use of truly anonymous digital money because it allows people to freely move large amounts of money around the world. In some schemes, banks know all details of a transaction but the identity of the client is hidden from the merchant.

The mechanisms that are most useful in a wide-area distributed system are the ones based on public key encryption and digital money. Systems based on secret key encryption are hard to scale mainly due to the large number of keys that have be kept both secret and on-line by key servers. Capabilities are useful as an optimization technique that is mostly hidden from the users of the system and can be introduced when needed.

Secure Binding to a Distributed Shared Object

As described above, an object stores access control information to be able to verify that a bind request is allowed and that a method invocation can be executed. In general, the binding process also needs access control information to protect itself from intruders who try to impersonate an object. The process that binds to an object needs to verify that the bind function returns the right object.

Often, this access control information is the identity of the owner of the object. Alternatively, it is possible to use host-based access control. In general, we can say that the user needs an special reverse-ACP for the owner/provider of the object. This reverse-ACP requires the object to authenticate itself to the user of the object. How does the user get the right ACP? Different alternatives are:

1. The name service stores the ACP together with the object handle. Note that if the name service cannot be trusted then we cannot trust the object handle returned by the name service either.

2. The location service stores the ACP along with the contact addresses.

3. The ACP is computed based on the contact addresses returned by location service.

4. The ACP is stored locally (with respect to the application).

5. A separate service maps an object handle to an ACP.
6. A public-key certification authority signs tuples consisting of a pathname, object ID, a public-key and a textual description of the organization, its (postal) address, etc. These tuples link a particular object (pathname and/or object ID) to a real-world entity (organization and postal address). The certification authority defines also the lifetime of a certificate: a limited time, or forever and whether a blacklist is used or not.

Storing the ACP locally (alternative 4) is very secure because only the local environment needs to be trusted. For example, a user may store his bank's public key locally to ensure that the objects used by his banking applications are run by the right bank.

The first alternative (storing an ACP along with the object handle) is the most convenient for worldwide communication. In general, the name service is trusted to return the object handle of the right object. Storing access control information in the name service eliminates the need for trusting the location service (alternative 2). Note that the use of a trusted location service also requires a trusted name service because the location service cannot tell whether the name service returned the correct object handle or not. Alternative 5 is similar to alternative 2, but uses a separate service that provides security information about an object handle.

Alternative 3 is similar to the host-based security used in the current Internet. Access control is not based on authentication information, but instead access control lists contain lists of Internet addresses that can be trusted. This approach assumes that the network is secure, an assumption that is invalid for most networks.

The last alternative provides additional security because it removes the need to trust the name service. The application has to demand a certificate which can be returned by the name service or by the object itself (after binding). The application compares the pathname stored in the certificate with the pathname used to bind to the object. Additionally, the application may also check the textual description of the organization. A disadvantage of this approach is that an object cannot be registered in the name service without contacting the certification authority to get a certificate. This approach is useful for objects with a well-known name.

Login Protocol

The security model requires a process to authenticate itself. Most authentication mechanisms require the use of secret information (shared keys, private keys, digital coins). For the purpose of this discussion we can assume that (at least part of) those secrets are stored in the process' address space. In practice it is possible to store keys in a separate process on the same machine. However, this approach only moves the problem to the process that *does* store the keys, and introduces a need for secure communication between two processes on the same machine.

A process can obtain the necessary keys in different ways depending on whether the process is a foreground process or a background process. A process is called a **foreground process** when it is (indirectly) attached to a terminal and controlled interactively by a user. Other processes are called **background processes** and provide services, for example, they manage persistent objects.

Three ways in which a process can obtain its keys are: (1) secrets can be passed by the parent process when the process is created, (2) secrets can be read from a local file, or (3) secrets can be derived from interaction with the user (password). The last option is available only to an interactive process. On the other hand, the second option is usually available to background processes, and is available only to foreground processes after the user has provided his password.

Interactive users expect to be able to use terminals (here defined as the combination of an input device, an output device, and a computing device connected to an untrusted network over which the user wishes to communicate) anywhere in the world. The assumption is that users may carry smartcards and/or memorize small passwords. A login protocol is needed to provide the user's first process in a session with the necessary keys.

Standard (UNIX) login protocols are not suitable for use over a wide-area network: many provide no protection against attacks such as password guessing, relaying information, or impersonation of the server. Furthermore, these protocols provide a login service; this login process starts a new process that is connected to the user's terminal. In Globe, we need a service that provides a process with the necessary secrets that can be used for authentication.

As described above, we can rely on users to carry a smartcard and/or to memorize a small password. In theory, biometric authentication is also possible, but it is not clear whether it can be used as a general purpose authentication mechanism in a worldwide system. Some issues are:

1. Are the input devices required to be tamper-proof? In many places, such as automatic teller machines, it is possible use tamper-proof devices that, for example, scan a person's iris. For communication over a wide-area network, it is possible that an intruder tries to use his computer to fake an iris scan.

2. How to avoid privacy violations? Most existing systems allow a person to have multiple identities, to share a access with other people, or to remain completely anonymous. Money, login/password combinations, ATM cards, and, in some countries, bank accounts are not directly tied to a single person.

3. Support for nonrepudiation? Public-key encryption can be used to actually sign a request. With biometric authentication it is not possible to directly tie a particular request to an authentication procedure.

A smartcard can be used to store a strong key. In contrast to a password that can be guessed, a strong key is "unbreakable." The smartcard can use this key to setup a secure, authenticated channel over the untrusted network. The necessary keys are passed over this channel to the smartcard, and from the smartcard to the user's process. Alternatively, the user's process can use the information stored on the smartcard directly. This is slightly less secure because it allows the smartcard's keys to be leaked.

A disadvantage of smartcards is that they can be lost or stolen. Note that it is hard to make smartcards that are tamper-proof. For example, since around 1994, almost every type of smartcard processor used in pay-TV systems has been successfully reverse engineered [Kömmerling and Kuhn, 1999].

Without a smartcard we have to use a password. We can use a secure, shared password protocol such as described in [Gong, 1995]. These protocols use random bits for padding and masking in such a way that a legitimate user can authenticate using a weak password, but the message exchange does not leak any information that allows an adversary to guess the user's password. Such a secure, shared password protocol can be used to setup a secure (encrypted) communication channel to the login server. The login server sends the user's private key over this channel to the login process. A disadvantage of these protocols is that the user's password has to be available in cleartext to the login server.

This problem can be solved by replacing encrypted nonces or timestamps with a list of challenge-response pairs stored at the login server. This approach involves the following steps:

1. The user's process looks up the public key of the login server.
2. A secure communication channel is established between the user's process and the login server using the login server's public key. Note that only the login server is authenticated at this point.
3. The login server sends the user's process a challenge.
4. The login process returns the challenge encrypted with user's password.
5. The login server verifies the encrypted challenge and sends the user's private key encrypted with users password.

This algorithm assumes that the public-key of the login server can be looked up reliably. The user can trust his terminal to store or obtain this key reliably or the user can carry for example a small piece of paper with a secure checksum (message digest) of the login server's public key.

The user should refresh his password after too many failed login attempts. Note that increased security comes at the cost of reduced availability. In this protocol, the login server has to stop when it runs out of challenges. It is thus quite easy for an intruder to mount a denial of service attack. The shared password protocols provide better resistance against denial of service attacks.

After a break-in, an intruder can directly steal the list of passwords when a shared password protocol is used. With a challenge-response scheme, the intruder gets a list of private keys encrypted with their owner's passwords. The intruder can try a password guessing attack to recover one or more of the private keys. A user can protect himself from this attack by using a good (hard to guess) password.

Using a smartcard is in most cases the best solution. Without smartcards, one has to choose between the possibility of denial of service attacks and the increased risk of a stolen password file.

Large, Secure Distributed Shared Objects

Figure 3.16 shows how large distributed shared objects can be structured using three rings with different consistency guarantees. A similar structure can be designed for security. Figure 3.31 shows three different levels of trust within a large distributed shared object. The core, level 1, is completely trusted and can arbitrarily modify the state of the object. Level 2 is trusted to keep the state of the object secret but is not trusted to modify the state of the object. Address spaces in level 2 have to send requests to modify the state of the object to level 1. Level 3 is, in general, not trusted. Operations have to be forwarded to a replica in ring 1 or ring 2.[10] This rings structure provides some containment of security breeches: malicious representatives in ring 2 cannot directly modify the state of the entire distributed shared object. They can, however, return incorrect replies to requests from representatives in ring 3.

In general, a representative may belong to different rings for security and consistency. The security ring required by a representative that provides inconsistent read access (replication ring 3) depends on whether the object needs a complete copy of the state of the object or not. To get a copy of the state of the distributed shared object requires the representative to be in security ring 2, otherwise it can be in ring 3. In more or less the same way, consistent read access can be provided in both security ring 2 and 3. However, representatives in the first ring have to be able to perform consistent updates which requires them to be in the first security ring where they are trusted with respect to integrity of the object.

[10]It is possible that some replicas in ring 2 are trusted to keep the state secret, but cannot be trusted to serve or forward request from ring 3. This problem can be solved using digital signatures.

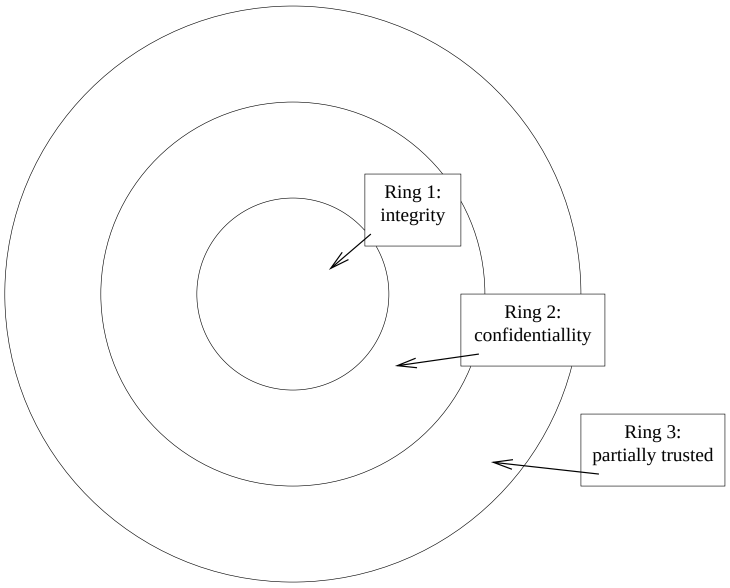


Figure 3.31. Security rings within a single object

3.4.2 Examples of Authentication

This section provides some examples of the interaction between different objects to execute an authentication protocol. Figure 3.32 shows the parties involved in the authentication of a (remote) method invocation on a distributed shared object. Local object L2, in address space A2 is part of a distributed shared object. Local object L1, in address space A1 tries to perform an operation on the distributed shared object. This operation is typically a bind request, but can also be a request to transfer or update the state or a method invocation.

Objects P1 and P2 are respectively the user's and the distributed object's principal objects. A principal object stores the public keys and certificates of the principal they represent. The object ID of the principal object is used to represent the principal worldwide. Figure 3.32 also shows three trusted third parties that may participate in the authentication protocol. A trusted third party is (by definition) trusted by both L1 and L2 (and their principal objects).

RSA The data flow for an authentication protocol based on RSA public key cryptography is shown in Figure 3.33. The protocol consists of the following steps:

1. There is a single trusted third party, the certification authority, that signs certificates binding the name and unique identifier of a principal to the principals

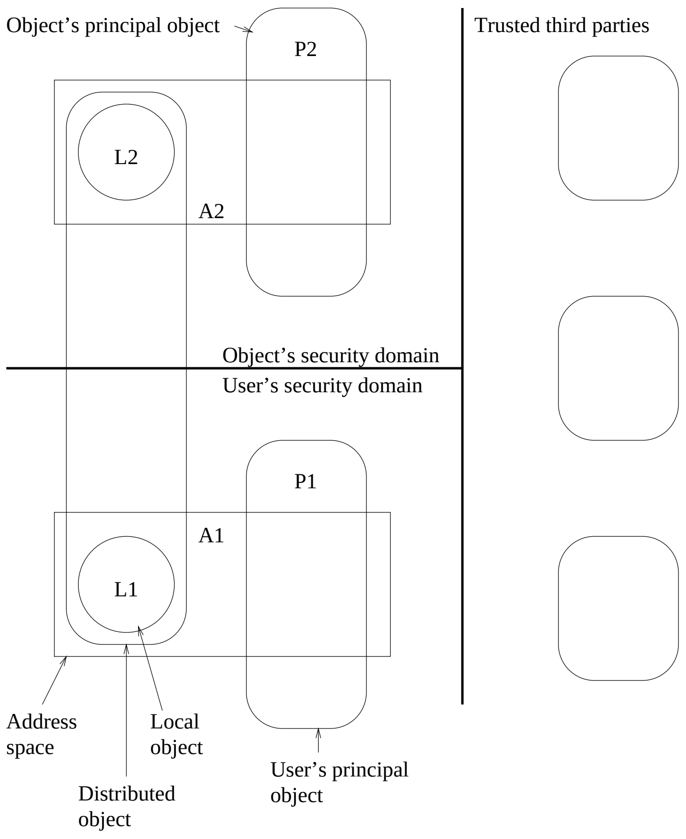


Figure 3.32. Three parties

public key. These certificates are created, and transfered to P1 and P2 long before L1 binds the distributed shared object. Both P1 and P2 know the public key of the certification authority.

2. The local object L1 sends a hash of the current session key (or a Diffie-Hellman public key [Diffie and Hellman, 1976]) to the user's principal object P1.

3. The principal object signs the hash with the user's private key, and returns both the signature and the certificates it got from the certification authority to L1.

4. This information is forwarded to the local object L2 in address space A2.

5. L2 asks P2 for the certification authority's public key which is used to verify the validity of the user's certificate. Finally, local object L2 verifies the certificate and the signature on the session key.

The unique identifier listed in the certificate is used to extract the access rights from the access control list. Note that no authentication information is encrypted. The authentication protocol involves computing (and verifying) only digital signatures.

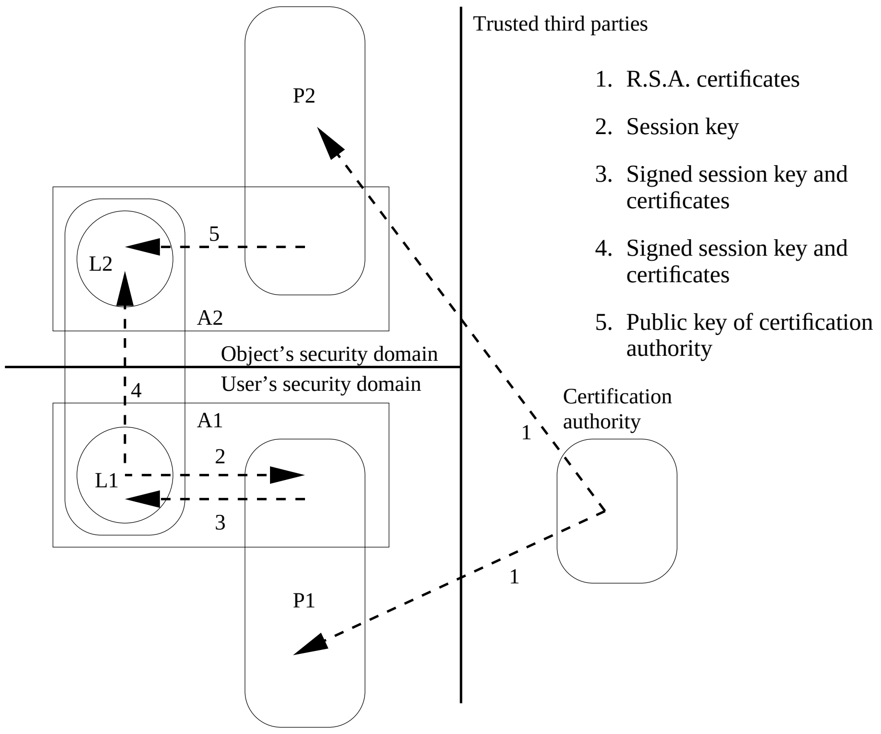


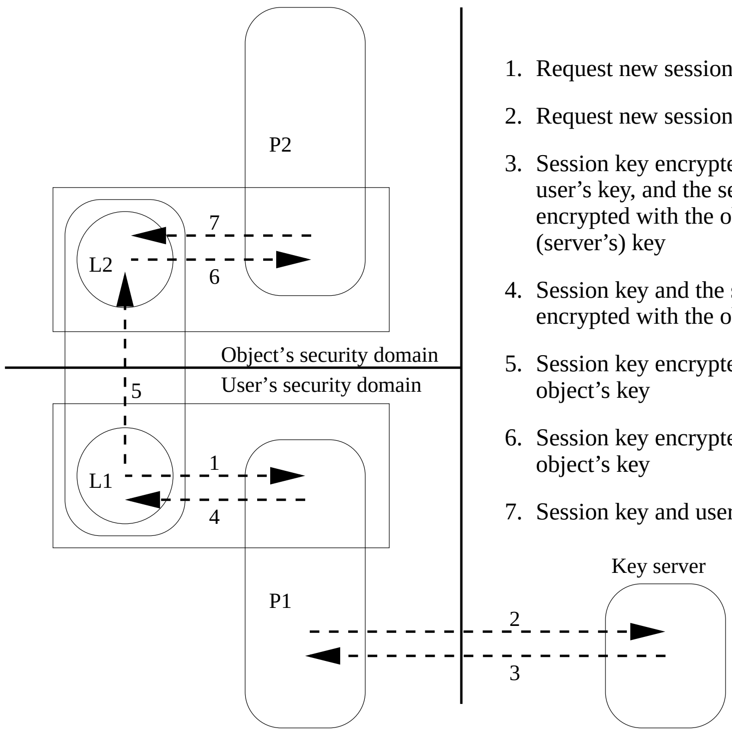
Figure 3.33. RSA authentication

Kerberos In contrast to RSA, the Kerberos authentication protocol depends on passing encrypted information between a key server and the communicating parties, as shown in figure 3.34. The Kerberos protocol uses the following seven steps:

1. In the first step, local object L1 passes a request for a session key to its principal object P1.
2. P1 passes the request to the key server along with a so-called "ticket granting ticket".
3. The key server shares secret keys with P1 and P2. These keys are called K_{P_1} and K_{P_2}. This allows the key server to generate a new session key K_s and to create two messages M1 and M2 that contain the session key and the identities of the communicating parties (P1 and P2) and are encrypted with K_{P_1} and K_{P_2}, respectively. These two encrypted messages are returned to P1.

4. P1 decrypts M1 and returns K_s and M2 to L1.
5. L1 sends M2 to L2.
6. L2 forwards M2 to its principal object P2.
7. P2 decrypts M2 and returns the session key K_s and the identity of the user (P1) to L2.

L2 uses K_s to communicate with L1 and computes the access rights based on P1. The authentication protocol described here is similar in structure to the one used by Kerberos, but leaves out many details that prevent the replay of messages.



1. Request new session key
2. Request new session key
3. Session key encrypted with the user's key, and the session key encrypted with the object's (server's) key
4. Session key and the session key encrypted with the object's key
5. Session key encrypted with the object's key
6. Session key encrypted with the object's key
7. Session key and user's identity

Figure 3.34. Kerberos authentication

Digital money The protocol described here creates an electronic check that is passed from a customer to a merchant, as is shown in Figure 3.35. In step 1, the user's bank creates a new check. This is simply a signature on a message. In step 2, local object L1 asks user's principal object P1 to fill-in the check and sign it. The signed check is passed to L2 and L2 passes the check to P2. The check is endorsed by P2 and forwarded to bank B2. Finally, bank B2 forwards the check to the user's bank B1.

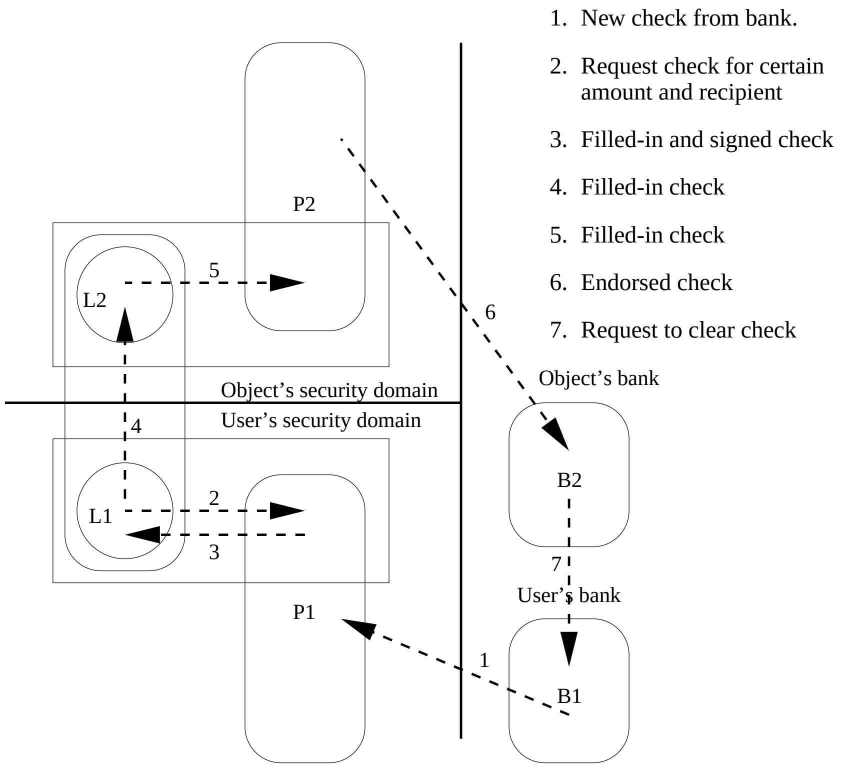


Figure 3.35. Digital check

Note that L2 either has to wait for confirmation from bank B1 that the check is valid, or it has to be able to verify the validity of the check in some other way. In general, L1 has to link a promise of L2 to deliver certain goods to its signature on the check. After all, L2 receives the check before the purchased goods are shipped. L2 may crash and lose all knowledge of the transaction after the check is cached but before any goods are shipped. The use of digital money for authentication means that L2 does not have to perform any additional access control based on the identity of P1: the transfer of money is sufficient.

3.4.3 Examples of Secure Communication

In the previous section, we described three authentication mechanisms. These authentication mechanisms combined with secure communication channels provide secure method invocations. Now, we present three examples: a simple client/server approach to explain the basics, an object that uses active replication, and an object that uses active replication for the primaries combined with other replicas (caches) that have read-only access to the state.

The examples assume that RSA public key authentication is used. The use of Kerberos shared key authentication results in a different protocol to setup a secure communication channel. The difference is that the Kerberos protocol provides both parties with a shared secret key. Using RSA, it is possible to create a shared secret key. However, a more interesting approach is to setup a secure channel first, and authenticate later. Two advantages of the latter approach are that the RSA private key is used only to sign information and not for encryption. The second advantage is that a passive intruder cannot observe the authentication protocol.

Client/Server In an object structured using the client/server approach, we have one representative that contains the state of the object and accepts network requests to read or modify this state. Other representatives are clients stubs: they only translate operations on the distributed shared object into network requests sent to the representative at the server.

Security can be provided as follows (see Figure 3.36):

(a) The communication objects of the client and the server setup a secure channel. This secure channel uses a fresh key to encrypt all traffic going over the channel. The encryption algorithm is a shared key cipher, such as triple DES or IDEA.

The key is generated by the communication objects themselves, using the Diffie-Hellman public key algorithm. Diffie-Hellman works as follows [Diffie and Hellman, 1976]:

1. Part of the protocol is a large prime number p that is used as a modulus to create an algebraic group, and a generator for that group g. It is possible that the protocol allows multiple pairs p and g.
2. Both sides pick a secret key, respectively x and y.
3. They send "$g^x \mod p$" and "$g^y \mod p$" to the other side, respectively.
4. Now both sides can compute $g^{xy} \mod p$.
5. This last value is used as the basis for their shared key.

Diffie-Hellman provides a secure communication channel. An extension to Diffie-Hellman, called ElGamal, can be used when nonrepudiation is needed.

(b) The next step is authentication and access control. The client signs its Diffie-Hellman public key ($g^x \mod p$) with his RSA private keys and sends this along with any certificates to the server. The server signs its Diffie-Hellman public key with the RSA private key of the object.

C	Control object	R	Replication object
Co	Communication object	S	Semantics object
P	Principal object	Sec	Security object

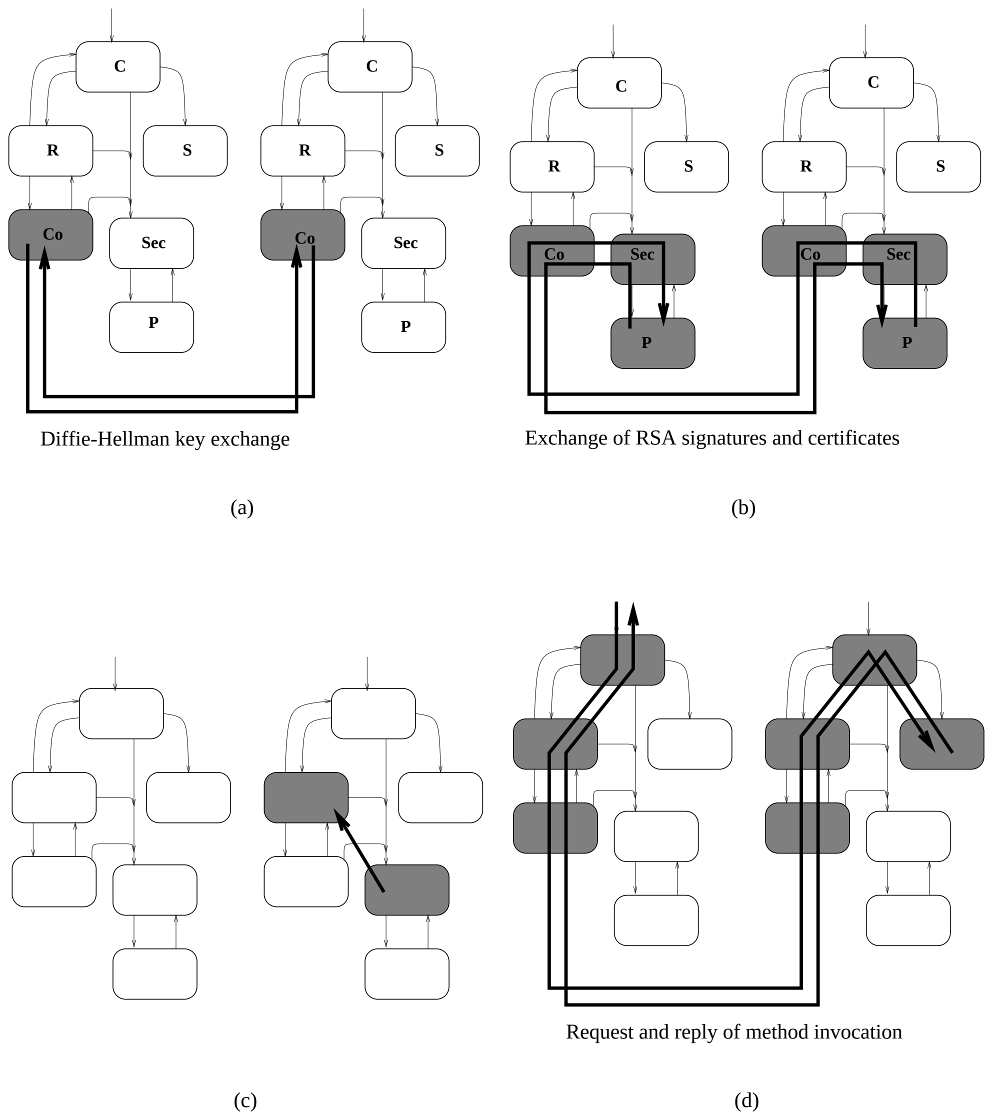

C
R
S
Co
Sec
P
Diffie-Hellman key exchange
Exchange of RSA signatures and certificates
Request and reply of method invocation
(a)
(b)
(c)
(d)

Figure 3.36. (a) Establishing a secure channel. (b) Authentication. (c) Replication object gets access control information. (d) Method invocation.

As described in the previous section, the RSA signature is computed by the principal object. Both parties need access to certificates to verify that the right RSA key pair is used for the signature.

(c) The security object in the server part of the distributed shared object, informs the replication object about the access rights of the client at the other end of the communication channel.

(d) A method invocation flows normally through the client and the server parts of the distributed shared object. The only difference is that the replication object also performs access control (i.e., does not pass illegal requests to the control object).

Active Replication With respect to security we can consider a distributed shared object that uses active replication to be equivalent to a collection of (equal) peers that communicate using a group communication protocol. The details of the implementation of the protocol (especially with respect to message ordering) can be ignored.

Secure group communication can be implemented as a straightforward extension to the client/server approach. For the group as a whole there is one secret key that is shared between all parties. This secret key is generated when the group is created, and should be changed when access control information is changed.

Figure 3.37 shows a join of a new group member to an existing group:

(a), (b) During binding, a new process sets up a secure, point-to-point channel to one of the group members using the client/server approach.

The first two steps, setting up a secure channel and authentication are the same as in the client/server approach.

(c) The group member informs the new member of the group's secret key. Usually, the group member also sends (part of) the state of the object.

(d) The new member sends a join message to the group encrypted with the key obtained in step (c).

There are two ways to join the main communication group:

1. A single communication object executes the complete protocol.

   After access control has been performed, the existing group member directs its communication object to allow the new member to join. This involves sending the secret key for the group over the established point-to-point channel.

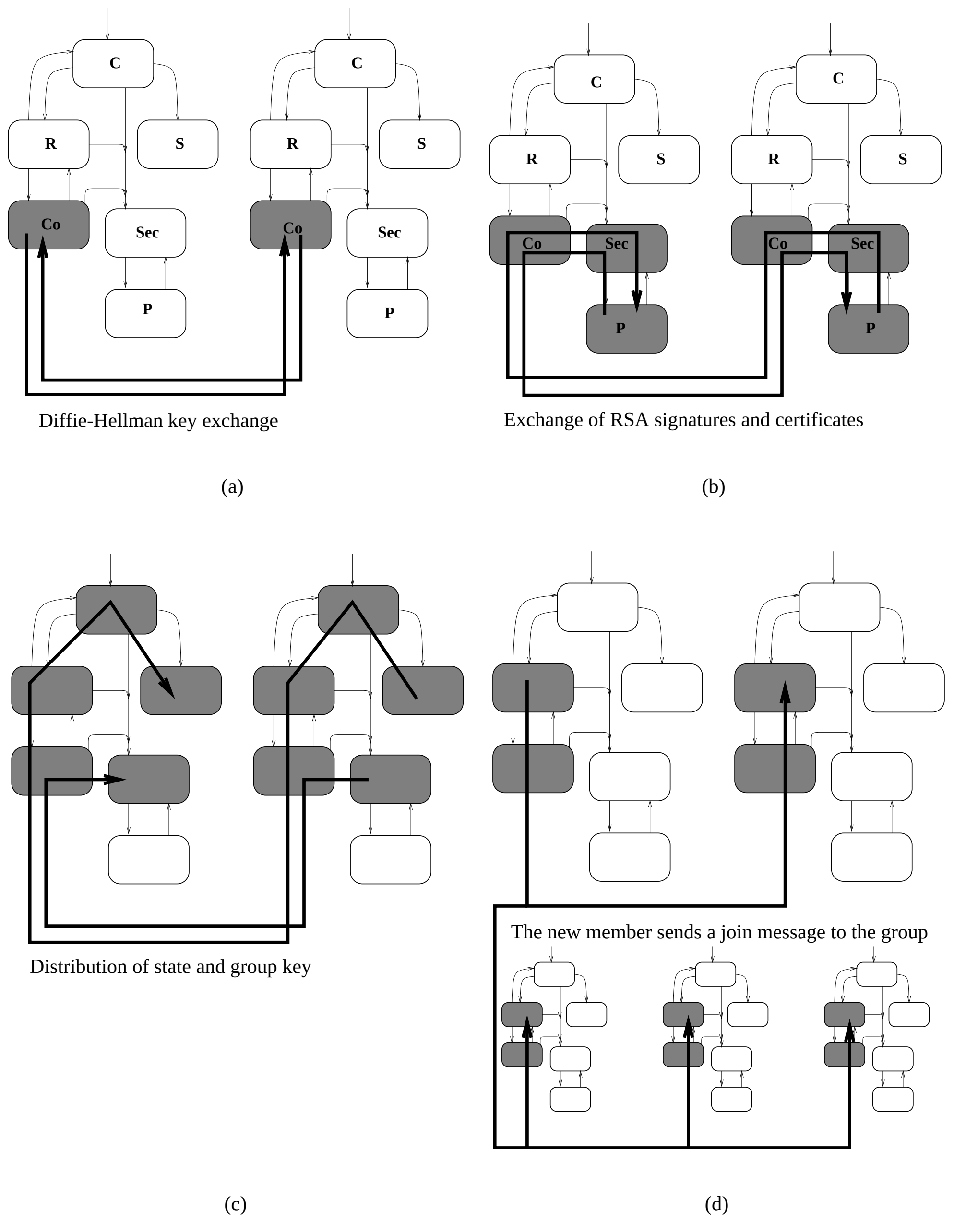

C
R
S
Co
Sec
P
Diffie-Hellman key exchange
Exchange of RSA signatures and certificates
(a)
(b)
Distribution of state and group key
The new member sends a join message to the group
(c)
(d)

Figure 3.37. (a) Establishing a secure channel. (b) Authentication. (c) Distribution of the state and the group key. (d) New member joins the group.

2. Two different communication objects are used.

   The first communication object is used to setup a point-to-point channel to a group member. The second one is the multicast group communication object that is used to actually join the group. In this case, the secret key is sent over the point-to-point channel to the security object in the local object trying to join. The security object passes the key to the group communication object.

Read-only Access combined with Active Replication In the previous example, all peers were equal. In this example, all peers have read access to the state of the object, but some peers are not allowed to execute modification operations.

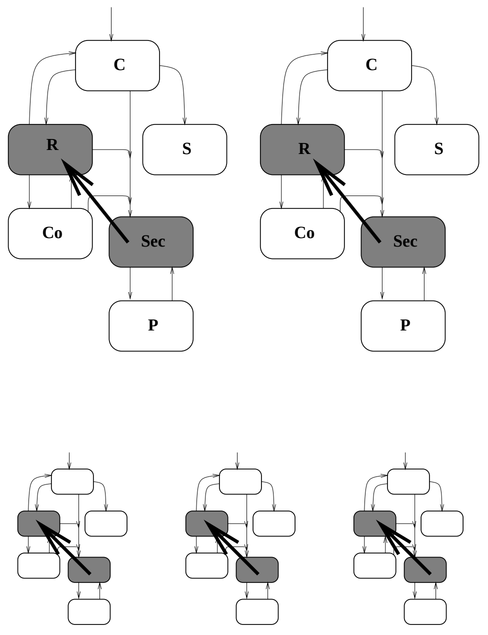


Figure 3.38. (e) Download access control information into the replication object

We setup a group similarly to the active replication example. The main difference is that a fifth step is added (Figure 3.38) in which the security objects of all members update the access rights stored in the replication objects. All existing group members update their information about the new group member. The security object of the new member computes the access rights of all other members.

However, to discriminate read-only members from read/write members, all messages need to be signed. Note that read-only members should not get access to the object's private key to prevent them from impersonating the object.

Chapter 4

Naming and Binding

4.1 Introduction

A system that contains a large collection of objects needs a structured way to refer to an individual object. We can distinguish two different approaches. The first approach is to use specialized access methods for specific application domains. For example, e-mail addresses refer to mailboxes in e-mail systems, and tables and records refer to specific data items in relational databases. The second approach is to introduce a generic mechanism to refer to an object independent of the application using the object.

A system which is used to refer to objects independent of their type is called a **naming system**. Standard naming systems are especially important to systems that provide a unifying paradigm for storing and manipulating data. A unifying paradigm for accessing data promotes interoperability between applications because different applications can share a single object. For the same reason, we also need a standard way to refer to those objects.

For example, in UNIX, files are used as the main paradigm for I/O. Ordinary files are used to store information which has to be kept on disk, and so-called special files provide access to terminals, network devices, printers, etc. Both ordinary files, special files, and, in modern versions, even processes are made accessible through a single naming system in which all files are essentially treated the same. This approach allows a UNIX shell to redirect the output of UNIX programs to ordinary files, special devices, etc. In most cases, programs are not aware that their output has been redirected and can use a single set of system calls to read and write data.

The object model described in Chapter 2 uses distributed shared objects as the unifying paradigm. In general, a distributed object is created in one address space, and registered under a certain name with a name service. After that, other address spaces can bind to that distributed object. The bind function, which implements binding, accepts the name of a distributed shared object, and returns a pointer to a standard object interface that can be used to access the object.

In general, naming services can be provided with different levels of abstraction. At a very low level of abstraction, close to the hardware, a naming system refers to individual bits and bytes. At higher levels of abstraction, we find objects and data structures in processes, processes and files on a single machine, machines in a network and worldwide distributed structures such as distributed shared objects.

At a very high level of abstraction, a naming system might be oriented toward a human user of the system. An example of a very high level name is the query 'Give me the address of a nearby bookstore that sells 'The Hitchhiker's Guide to the Galaxy." This high-level query may ultimately refer to a collection of bytes with the very low-level reference "bytes 163 – 205 of object 10 in process 23 on machine 10.3.1.5." which contains the address of a nearby bookshop. Going from this high-level name to a low-level name usually involves different naming systems at different levels of abstraction. For example, the first query may return the name of the bookstore object which contains not only the name of the bookstore but also the catalogue. A lower level name service may return the network location of a distributed object that represents the bookstore. Further down, the network routing tables (another naming system) return a route to the machine, and on the machine itself, the right process is located, and so on.

Below we present a collection of issues that influence the structure and semantics of a name *space*. Later, we will focus on issues that influence the implementation of a name *service*.

- Location transparency versus location dependence

  Names that have to be valid for a long time should not embed information about the location of the object or data item they refer to. Location-independent names are sometimes called pure names [Needham, 1993]. A pure name is just a label and does not contain any information about the item or items it refers to. An impure name, on the other hand, does contain location information, or semantical information about the item it refers to. An impure name prevents an object from being moved, or requires an object to get a new name when it is moved.

- Human-readable versus machine readable

  Humans are quite good at dealing with context sensitive or ambiguous names. On the other hand, computers can easily deal with arbitrary, fixed-length bit strings, which are hard to remember for humans. Programmers, and other computer professionals are somewhere in between: they typically use names based on words and abbreviations (instead of arbitrary bit strings) that contain a certain amount of syntactic sugar. For example, http://www.cs.vu.nl/globe/ is an acceptable way for a computer programmer to refer to "The Globe home page at the Vrije Universiteit."

- Unique object identifiers versus variable object references

  Unique object identifiers satisfy three requirements: (1) there is a one-to-one correspondence between an object and its object identifier (i.e., an object has exactly one object identifier and a valid object identifier is associated with exactly one object). (2) Unique object identifiers are never reused. After an object has been destroyed, its object identifier becomes invalid forever. (3) An object is never assigned a new object identifier [Wieringa and de Jonge, 1995].

  In a distributed system, unique object identifiers provide an easy way to verify whether two object references refer to the same object or not. Two object references refer to a single object if and only if the object identifiers associated with the objects, to which the two references refer, are equal.

  Without unique object identifiers, it is quite hard to distinguish between two references to a single object and to two objects of the same class with exactly the same state. Another problem is that an object reference may possibly be reassigned to a different object without notice of the holder of the reference. Without unique object identifiers, it is hard to make sure that an object reference continues to refer to the same object.

  A naming system based on unique object identifiers provides an easy way to refer to a particular object over a period of time. For example, NFS uses unique object identifiers in the form of **file handles**.[1] An NFS client can continue to use a specific file even if the file is renamed or if the server reboots.

  On the other hand, the UNIX file system does not provide unique file identifiers. It is in general not possible to reopen a particular file because it may have been moved to an unknown directory. Furthermore, the same inode number may be reused for a different file after the first one is destroyed.

  Note that in a program, the memory address of a global variable would be a unique identifier. In contrast, a pointer variable can refer to at most one object at a certain time but can refer to different objects at different times.

- Precise versus imprecise

  A name may refer to a single, well-defined object, or it may refer to (one of) a collection of objects. We call a reference to a single object "precise," and a reference to a (possibly empty) collection of objects "imprecise."

[1]Well almost. Sun uses a generation number to identity files that sequentially share the same inode. However, the generation number is 32 bits and can overflow. Furthermore, it is possible to destroy an old file system and create a new one on the same disk device.

	X.500	DNS	URLs	URNs	UNIX files	SSP chains
Location transparent	Yes	Yes	No	Yes	Yes/No	Yes
Human-readable	Yes	Yes	Yes	Yes	Yes	No
Unique OID	No	No	No	Yes	No	No
Precise	No	Yes	Yes	Yes	Yes	Yes
High-level	Yes	Yes/No	No/Yes	Yes	Yes	No
Relative	No	No	No/Yes	No	Yes/No	Yes

Table 4.1. Features of various naming systems

- The level of abstraction

  We have already discussed name-lookup operations at different levels of abstraction. In general, we can assume that a name service accepts lookup operations at one level of abstraction and the result it returns is at a lower level of abstraction. This allows multiple name services to be stacked hierarchically to translate one request at a high-level of abstraction into a reference to a collection of bytes in a data structure.

- Relative versus absolute names

  Relative names are names that are interpreted relative to a specific starting point, a naming context. A name in a naming scheme that uses relative names may refer to different objects depending on the naming context used when the name is looked up. An absolute name always refers to the same object until the name-to-object mapping is changed.

  UNIX combines these two schemes by using absolute and relative pathnames.[2] If the first character of a pathname is a slash character ('/'), the pathname is absolute, otherwise it is relative. Relative pathname are resolved relative to a context called "current working directory" which is per process and is set with the chdir system call.

Table 4.1 shows the features of some example naming schemes. X.500 [Heker et al., 1992] is an ISO standard naming service. The main characteristic of an X.500 name is that it is a collection of attribute/value pairs. For example: "c=us, o=Merit, cn=Chris Weider." This name means: country equals U.S., organization equals Merit and name equals Chris Weider. X.500 names are location-transparent, human-readable, imprecise, high-level, and absolute.

[2]Ignoring the chroot system call which changes the root directory for a process.

The Domain Name System (DNS) [Mockapetris, 1987] is an Internet standard for naming hosts and mail exchange servers. DNS names are location transparent, human-readable, precise.[3] DNS is designed to map host names to Internet address and is not designed to find the machines, for example a WWW server, to belong to a particular company or organization. DNS is only somewhat high-level.

Uniform Resource Locators (URLs) [Berners-Lee et al., 1994] are the basic naming mechanism for the World Wide Web. A URL consists of a naming scheme followed by a name in the naming scheme. In this comparison we ignore name schemes other than HTTP and FTP. In the HTTP and FTP naming schemes, a name consists of two parts: a host name and a filename. A name is therefore not location-transparent. This is done on purpose: a URL is a resource *locator*. URLs are human-readable and precise. URLs exist in both absolute and relative forms. Absolute URLs are typically used to access an initial WWW page and relative URLs can be used in a WWW page to refer to other WWW pages on the same site. A URL provides the current location of a resource. In the design of the World Wide Web, a URL is not indented to be high-level, instead a URL is designed to be returned as the result of a lookup operation on a higher level name. However, these higher level services are not commonly used, and URLs are used as a high-level name.

A URN is a Uniform Resource Name [Sollins and Masinter, 1994]. URNs are resolved to URLs. URNs are not widely used but the design criteria include location-transparency, easy to transcribe by humans, and uniqueness. A URN refers to a single (logical) document and is therefore precise.

Files in a UNIX file system are somewhat location-transparent: (distributed) file systems can be mounted and integrated in the name space. A UNIX filename does not imply a location. However, files that belong to a single file system are stored on a single disk device. UNIX filenames can be absolute or relative (to the current working directory), but even absolute names are relative to the root on the local machine. Filenames provide the highest level of abstraction for object names in a UNIX system.

A Stub-Scion Pair chain (SSP chain) [Shapiro et al., 1992] is a naming scheme for distributed objects. In this scheme, a (distributed) object reference is always a local object pointer, either to the object directly (if it is local) or to a local stub (if the object is remote). The stub communicates with a scion in a remote address space. Object pointers are location-transparent because they either point to a local object or to a local stub. Pointers are not human-readable, not unique, and not high-level. Furthermore, those object pointers are always relative (to the current address space) and refer to one object.

[3]A single DNS name can refer to multiple hosts (round-robin DNS), but this is not how the system was intended to be used.

Object-Based Naming Figure 4.1 shows a hierarchy of name spaces. High-level naming schemes are at the top, low-level ones at the bottom. High-level naming services are, for example, white and yellow page systems for human users and factory finders for computer systems. Low-level naming systems that refer to smaller, per process data structures are layered below the local object that implements part of a distributed shared object. For example, within a process objects are referred to by interface pointers. Within composite objects, subobjects can be referred to. Between the high-level and the low-level naming schemes is the naming system for distributed shared objects.

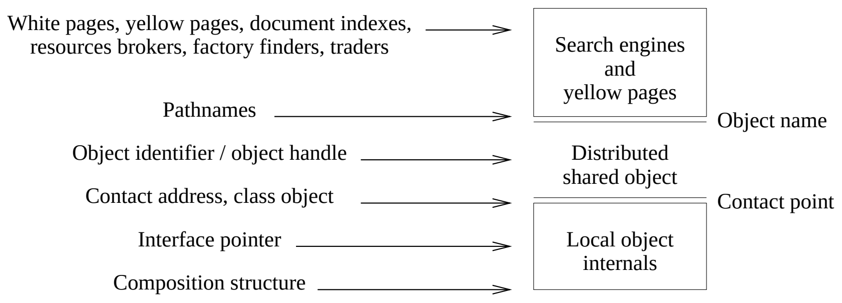

White pages, yellow pages, document indexes, resources brokers, factory finders, traders
Pathnames
Object identifier / object handle
Contact address, class object
Interface pointer
Composition structure
Search engines and yellow pages
Object name
Distributed shared object
Contact point
Local object internals

Figure 4.1. Naming a distributed shared object

To structure naming in Globe, we will concentrate on two different name spaces that refer to distributed shared objects: a pathname that refers to a distributed shared object as a whole, and a contact address to actually bind to the object.[4] All high-level naming schemes, such as white pages, yellow pages and resource brokers should be designed to return a set of pathnames as the result of a query. These high-level name services will not be discussed further in this chapter. The Globe naming services will translate such a pathname into the low-level set of contact addresses and class objects. Below this low-level name, we no longer deal with distributed shared objects but with objects and data structures in a single process. Low-level name services will be discussed in Section 4.3.

The use of a contact address as the low-level name for a distributed shared object is part of our distributed object model. For a process to bind to a distributed shared object, it has to install a suitable, local object which can then connect to the distributed shared object using the contact addresses published by the distributed shared object. The naming scheme for distributed shared objects in Globe should therefore provide three functions:

- Provide a way to refer to (to name) distributed shared objects.

[4]Together with a suitable implementation that implements the right communication protocol for the contact address.

- Determine the location (in terms of communication endpoints) of the object.
- Select, based on the communication protocol used by the distributed shared object, a suitable implementation (class object) for a local object to connect to the distributed shared object.

The name space for naming distributed shared objects is not constrained by the object model. However, we have the following requirements:

- We want location-transparency, as this is needed for replication and for object migration.
- We would like a naming scheme for *objects*. Therefore, the naming system should be precise. References to collections of objects can be implemented in higher level naming systems.
- We need absolute names, as they allow the users of the system to exchange names without explicitly or implicitly specifying the naming context.

What remains is to select whether the name should be human-readable or not, and whether unique object identifiers should be used.

Using a unique object identifier can be an advantage or a disadvantage. The advantage is that higher level naming systems can refer to one particular object; there is no risk that the object identifier is reused and refers to a different object. The disadvantage is that it is impossible to replace an object with a different object, only the contents of the object can be updated. Note that human-readable, unique object identifiers are almost impossible. A unique object identifier has to remain unique forever, where as humans prefer to use names with semantic content which are likely to be reused for different objects at different times (imagine a programming language, or a file system where you can use a particular identifier or pathname only once).

Using human-readable names also has advantages and disadvantages. Human-readable names for objects simplify referring to those objects in day-to-day conversations. The main disadvantage of human-readable names is that it is harder for a program to use, store and compare them. Furthermore, internationalized character sets (e.g., Unicode) are needed to satisfy many users.

Human-readable names, combined with a sufficiently powerful naming service can be used to provide a way to integrate unique object identifiers. Part of the name space can be reserved to provide access to objects referenced by unique object identifiers.

Hierarchical name spaces have nice scaling properties compared to alternatives such as flat name spaces and name spaces with attributes. These scaling properties include a virtually unlimited collection of names, support for distributing and partitioning the name space over multiple servers, machines, organizations, etc. Furthermore, hierarchical name spaces support delegating responsibility for parts of the name space to different users or organizations.

For example, the UNIX file system name space can easily support millions of objects. A typical Usenet-news file system alone stores a million news articles. The mount system call allows a large name space to be constructed from smaller name spaces in file systems. Individual users or projects are given their own private portion of the name space by creating home directories, for example /home/globe.

On a worldwide scale, DNS (Domain Name System) is used to name a large number of computers on the Internet. In DNS, distribution over multiple machines and delegation of authority are combined in a single mechanism. A DNS NS record delegates a part of the name space to a new "zone," on a different name server; a SOA record at the root of a "zone" describes the responsible person or organization.

4.2 Binding to Distributed Shared Objects

Binding to a distributed shared object consists of translating a path name into a contact address and a class object, followed by the creation and initialization of an instance of the class object. The newly created object is the local part of the distributed shared object (as described in Section 2.3) and uses the contact address to connect to the rest of the distributed shared object.

It is important to realize that input to the binding process is only a (possibly pure) name for the object, no other information about the object is available. In contrast, in the case of a URL, the protocol is explicitly encoded in the first part of a URL by means of the protocol identifier (HTTP, FTP, MAILTO, etc.).

In most distributed systems, the naming system maintains a mapping from names to network addresses. This approach has two disadvantages: the naming system does not store information about the communication protocol that is to be used, and a name resolves to only one contact address. For example, in the URL http://centaur.cs.vu.nl:80/, the name of the computer to be contacted is centaur.cs.vu.nl. This hostname resolves to the Internet address 192.31.231.174. The parts that identify the protocol (HTTP) and the TCP port to be used (80) are part of the name and are not stored in the name service. This means that changing the TCP port number of the WWW server requires changing the name of the object.

The same hostname to address mapping is also used by other programs such as telnet for remote login, NNTP for news, finger for login information, etc. This means that, adding an extra address to host centaur, for example the address 168.100.189.97, to replicate the WWW server means that the other protocols (Telnet, NNTP, etc.) also start using the extra address. The solution that is currently used on the Internet is to use special hostnames such as www, mail, and news that resolve to the addresses for the real WWW, mail and news servers.

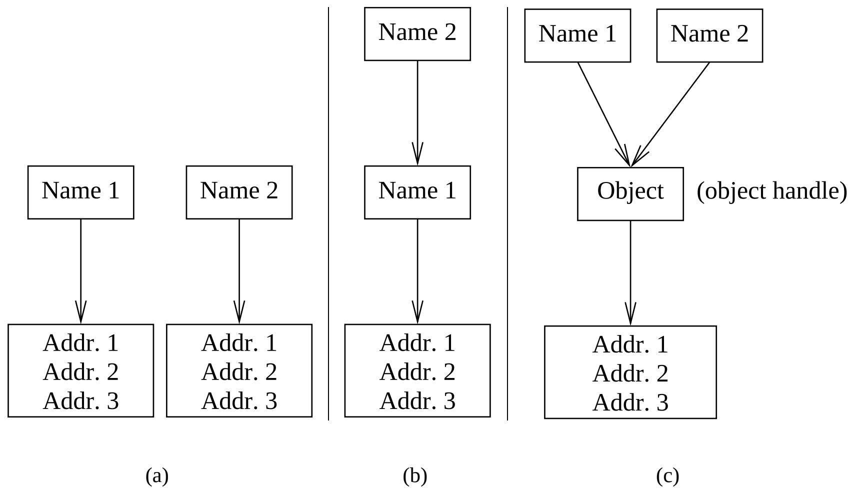


Figure 4.2. Mapping object names to contact addresses

Another issue is how to register an object under multiple names. Figure 4.2 shows three approaches to map multiple names for a single object to a set of contact addresses. The first approach shown in Figure 4.2(a) simply duplicates the set of contact addresses for each name of the object. The second approach recognizes one primary name (Name 1) which directly maps to a set of contact addresses, and zero or more secondary names (Name 2) which point to the primary name. The third approach introduces an extra level of indirection: the object's names are first mapped to a (unique) object identifier, called **object handle**, which is subsequently mapped to the set of contact addresses.

We compare the three approaches in three areas:

1. Changing the set of contact addresses.

   It is quite hard to change the set of contact addresses when the first approach is used. The reason is that in a worldwide system, it is not possible to find all the names in the name service that refer to a single object, unless the system also contains a complete reverse index that returns all the names for a single contact address. In this respect, the second and third approach are better because the contact addresses are stored in one place.

2. Creating and deleting names for an object.

   In the first and the third approach, different names for an object are quite independent. New names can be added by either copying the set of contact addresses (first approach) or by simply referencing the object handle (third approach). In the second approach, deleting the primary name when there are still secondary names around is a problem. It is necessary to select a new primary name, and to update the remaining secondary names. Otherwise the secondary names are left dangling. Adding a new name using the first approach is somewhat less efficient than the third approach for objects with a large set of contact addresses. The reason is the each new name requires copying the set of contact addresses. In short, the third approach is the preferred alternative.

3. The cost of a complete name lookup.

   Of the three alternatives, the third approach always has an extra level of indirection, which tends to increase the cost of a name lookup. The first alternative is the most efficient.

For Globe, we chose the third approach, for two reasons. The first reason is that compared to the other two alternatives, this approach provides the most flexible way to deal with (possibly large) sets of contact address and objects with multiple names. Furthermore, handling path names separate from sets of contact addresses provides an opportunity to use different technologies for the two services. A separate name service, called the location service, is introduced that maps object handles to sets of contact addresses. The regular name service maps object names to object handles. The separation of the name service from the location service allows the name service to focus on adding and deleting names for an object. At the same time, the location service deals with mobile objects, which frequently replace one address with another, and highly replicated objects (with possibly large sets of contact addresses).

The first step of the binding process, that is, mapping a name to a contact address, can be subdivided into four smaller steps: a name-service lookup, a location-service lookup, destination selection and implementation selection (see Figure 4.3). We will describe each step briefly before going over the details.

Step 1: name lookup

In the first step, the path name passed as an argument to the `bind` call is resolved to an object handle. This effectively decouples the name of an object from the actual location of that object.

Step 2: location lookup

In the second step, the object handle is resolved to a set of contact addresses. Each element of this set describes the network address, port number, etc., where the object can be reached, and the protocol stack that has to be used to communicate with that particular destination.

A protocol identifier combined with actual address information is similar to a URL: in the URL `http://www.cs.vu.nl/~philip`, `http` is the protocol and the remainder is the address. In other name services, the protocol is implicit. An example

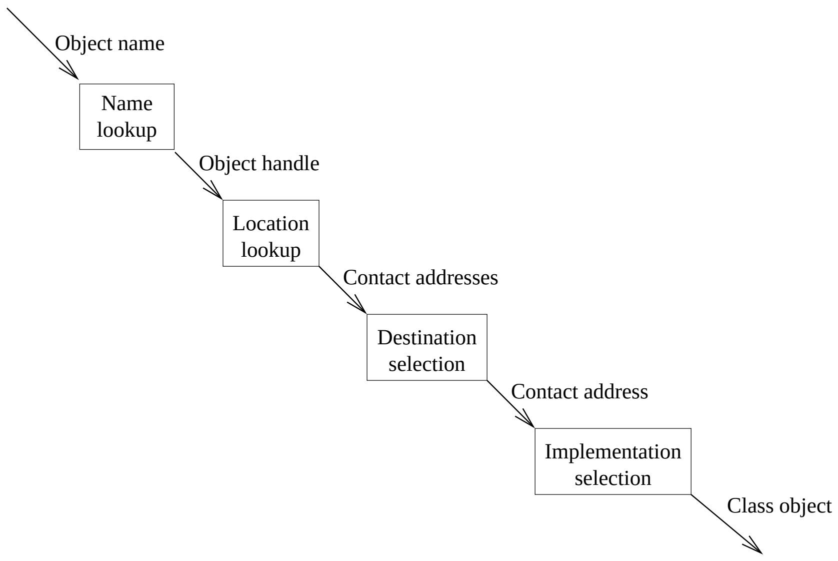


Figure 4.3. Four-step naming of a distributed shared object

of the latter is a DNS MX record. The contents of the record is an IP address and a priority. The protocol and TCP port are implicitly specified as SMTP and port 25.

Step 3: destination selection

In the third step, one contact address is chosen. In general we expect that there is more than one contact address for two reasons: a single object may use different protocol identifiers to support different protocols for binding to the object, and objects with replicated or partitioned state offer multiple contact addresses for the same protocol to increase availability and locality. This selection should be made, based on locality, security constraints, and the set of protocol implementations that are available in the address space from which binding takes place.

Step 4: implementation selection

In the fourth step, a suitable implementation for the selected protocol is chosen. Implementations are stored as class objects; new ones can be added dynamically. Once a class object is loaded, it can be used to access multiple distributed shared objects of the same class. To access a single object from all over the world, we use worldwide unique protocol identifiers that are resolved locally to a class object that implements that protocol.

After the class object, selected in step 4, has been loaded, it is used to create a new local object. This local object is then initialized with the distributed shared object's object identifier and the contact address of the chosen communication endpoint. To complete the binding process, the new local object sends the object identifier over the communication channel to the distributed shared object. The distributed shared object verifies that the object identifier corresponds to its own object identifier. The object identifier is sent by the new local object to avoid a problem with stale caches. This problem will be described in Section 4.4.5. The local object becomes a representative of the distributed shared object.

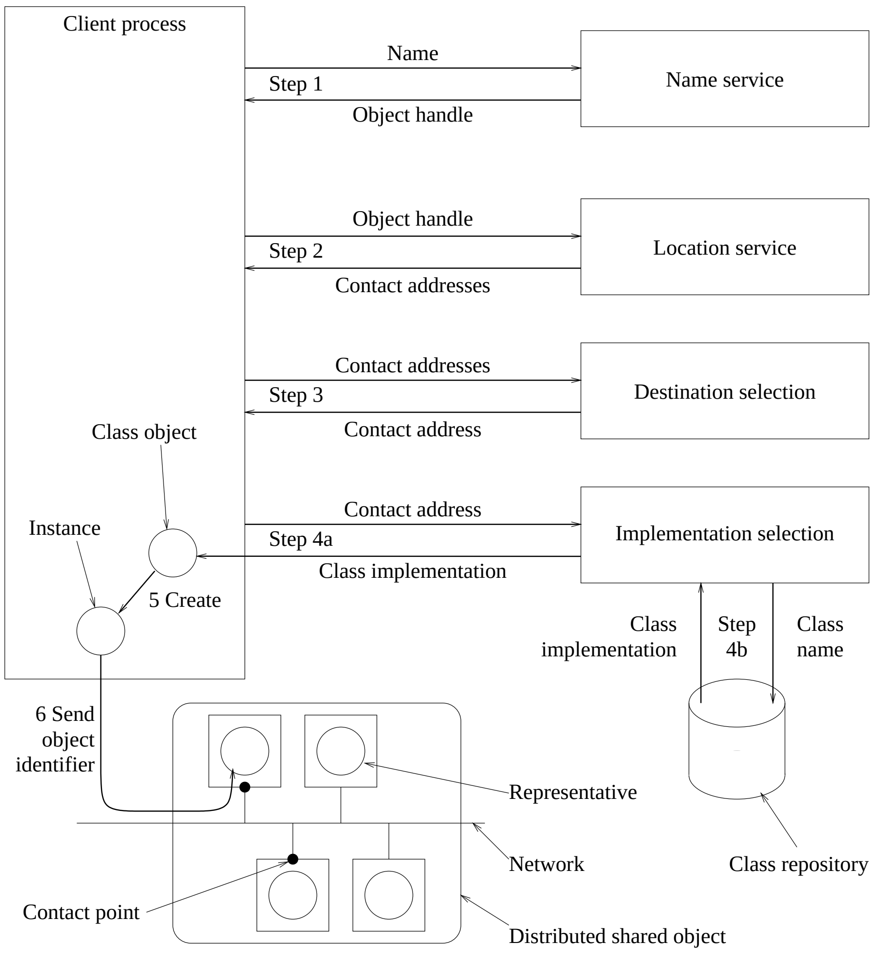


Figure 4.4. Binding to a distributed shared object

The binding process is illustrated in Figure 4.4. Steps 1, 2, and 3 perform, respectively, the name lookup, the location lookup, and destination selection. Step 4, implementation selection, uses a class repository. In step 4b, the name of the class

Existing name spaces	**Prefix**
Telephone numbers	/globe/E164/
D.N.S.	/globe/dns/
X.400	/globe/X400/
New name spaces	**Prefix**
Geographical	/globe/geo/
Companies	/globe/biz/

Table 4.2. Globe name spaces

that implements the protocol in the selected contact address is passed to the class repository. The class repository returns a class implementation, which is returned to the client process. In step 5, the class object creates a new instance. This new instance connects to the distributed shared object in step 6 and sends the object identifier to the distributed shared object for verification.

4.2.1 Name-Space Objects

In the previous section, we saw that in the Globe system a complete name lookup uses two services: a name service and a location service. The name service maps a name to an object handle, and the location service maps the object to a set of contact addresses. In this section we concentrate on the name service. The two main issues we will discuss are the structure of the name space and the implementation of the name service as a collection of distributed shared objects.

Table 4.2 shows two approaches to the design of the name space for distributed shared objects. One approach is to incorporate existing name spaces into a single Globe name space. Another approach is to create completely new name spaces.

To integrate multiple name spaces into a single name space, we allocate unique prefixes for each name space. Table 4.2 shows the prefixes allocated for three existing name spaces. For example, a DNS name cs.vu.nl would become /globe/dns/cs.vu.nl/.

All names for distributed shared objects start with "/globe" for two reasons: it makes names of distributed shared objects easy to recognize, and it makes them easy to embed in other name spaces. The first reason is similar to the use of the '@' symbol for Internet e-mail, http:// and ftp:// for URLs and even www.*domain-name* for home pages. The second reason is that this approach allows the Globe name space to be incorporated into the name space for local objects (Section 4.3) or the UNIX local file-system name space. A similar approach is used by AFS, which uses the prefix /afs/*DNS-name*/

A good reason for re-using existing name spaces is that the creation of a global name space is complicated by nontechnical issues. An example of a global name space is the global telephone numbering plan (ITU recommendation E.164). Two other examples of worldwide name spaces are X.500 addresses and DNS.

An example of a nontechnical issue is trademark conflicts. A flat name space, combined with user-chosen names may limit the name space to the point where there are too few names to name a large collection of objects. This problem occured on the Internet in the .com DNS domain. Internet users are used to finding companies, trademarks and sometimes even products in the .com domain. For example, the WWW-server of a company called Apple is expected to be at www.apple.com. Unfortunately, trademark laws allow multiple companies to use the same trademark as long as they sell different products or operate in different areas. In the case of Apple, www.apple.com refers to the WWW-server of the company that makes Apple computers. However, there are other companies that are also called Apple, for example, the Beatles' record company. In general, only one company can register a particular trademark as *trademark*.com. Even worse, it is not clear whether a trademark is required at all to register a name in the .com domain [Oppedahl, 1997b].

The main problem here is not one of trademarks, but that users use a naming scheme that was designed for programmers and system administrators to name hosts on the Internet, as a white pages service to locate companies. Users prefer to enter just a single word or abbreviation to refer to a particular company. This worked well when few companies were present on the Internet. However, the name space of easy-to-guess and easy-to-remember words is quite small compared to the number of companies in the world and collisions were bound to occur.

The solution to this problem is to use longer names, which are still relatively easy to remember but are harder to guess. To assist users in finding the Internet name for a particular company, product or individual, we need an interactive white pages service. Users provide this service with some initial information, such as company name, or an individual's last name. The white pages service then provides list of matching entries, or if there are too many matches, asks the user to refine the query by adding for example, geographical information. Without such an interactive white pages service, managing a worldwide name space can be quite complicated, especially if the name dispute policy is flawed [Oppedahl, 1997a].

There are two different ways to integrate an existing name space into the Globe name space:

1. Use the "values" of the name space (for example the telephone numbers of E.164).

   This approach treats the existing name space as nothing more than a collection of values. This is necessary for telephone numbers, which are just that: a collection of values without support from any accessible, distributed name service.

2. Actually incorporate the existing distributed name service for a name space in the Globe name service.

Both approaches to the integration of existing name space have drawbacks. The main drawback of the first approach is that the name server used by Globe has to be kept synchronized with the official allocation of the name space. A problem with the second approach is that existing name servers (such as DNS and X.400) use a completely different technology to distribute, replicate, and update their information.

Both approaches can be integrated into a single system. If we take, for example, the DNS name space, we can use the first approach to create a hierarchical collection of directories that follows the structure of DNS. Each DNS domain becomes a separate directory. An example of a name would be /globe/dns/nl/vu/cs/.... The nl, vu.nl, and cs.vu.nl DNS names map onto the directories /globe/dns/nl, /globe/dns/nl/vu, and /globe/dns/nl/vu/cs. With the second approach the lookup is redirected to a DNS resolver. Note that the result of this lookup operation has to be an *object handle*. An object handle can be stored in DNS using the TXT fields. In this case, the name looks like /globe/dns/cs.vu.nl/.... The cs.vu.nl part is passed to the DNS resolver.

In addition to using existing name spaces, we can also use new ones such as the two name spaces shown in Table 4.2. /globe/geo is the start of a geographically allocated name space, starting with the two letter country codes, and followed by entries for states, provinces and cities. /globe/biz is for the promotion of products and services.

Implementation We can use two different approaches to build a name service in a system based on distributed objects: treat the name service as a separate entity which is completely independent of the object technology used, or implement the name service as a collection of directory objects that are linked together. An example of the first approach is the World Wide Web. The name server used for WWW documents is a combination of existing name services such as DNS and local (e.g., UNIX) path names.

An example of the second approach is NFS [Nowicki, 1989]. In NFS, both files and directories are identified by 32-byte identifiers called **file handles**. NFS directories implement a name-lookup operation which returns the file handle for a name in a directory. A complete name lookup for a filename is split up in a sequence of directory-entry lookups which are performed on different NFS directories.

In their purest forms, both approaches have drawbacks. A completely separate name service does not benefit from the technology used to implement the objects (support for persistence, security, replication, etc.). The use of directory objects restricts either the name space, or the object implementations or both. For example, in some cases, part of a name space is derived from an external source, such as a (product) database. Enforcing the rule that each directory is a separate object leads to a large collection of objects that have to reflect the contents of the database.

For Globe, a combination of the two approaches is the most promising solution. The name space is implemented as a collection of distributed shared objects. However, each object can implement an arbitrary number of directories. Note that these "name-space objects" are not *required* to store their contents explicitly. Instead, they are allowed to compute or to look up the necessary information on the fly.

A name lookup on a directory object is functionally:

$$o.lookup(pathname) \rightarrow objectHandle,\ remainingPath$$

The lookup method is invoked on a name-space object o to resolve a path name. The result of this method is an object handle and the remainder of the path name. The object handle returned by the lookup method can either refer to the requested object (the object denoted by the name) or to a directory object. The requested object handle was returned when the remainder of the path name returned by the lookup method is empty. Otherwise, the object handle should refer to a directory object which is used for the next lookup. In this case it is necessary to bind to the directory object (i.e., we get a location lookup, destination selection and an implementation lookup).

This scheme is quite flexible:

- Multiple (independent) name spaces are supported by allowing the name lookup to start with different root objects. Note that this has to be supported by the syntax that is used for naming.
- A single lookup-method invocation can process multiple components. This is an important optimization if the state of multiple directory objects that are used during a name lookup, is stored in a single address space (or on a single machine). In that case, the directory object should try to look up the next component itself, and return the result of that second lookup. Looking up one component at a time, requires forwarding multiple lookup-method invocations to the same address space. The additional network latency can significantly increase the total time needed to complete the lookup operation.
- The lookup method returns a new path name. This can be used to implement something similar to symbolic links. Normally, the lookup-method invocation replaces a single component with an object reference. However, the lookup method can also return a reference to a root directory object and prepend some number of new components to the list of components passed as its argument. A problem with this feature is that a loop can be constructed that is almost impossible to detect.
- Storing directories in distributed shared objects provides support for different replication strategies, and different consistency guarantees for different objects. But directory objects can also use alternative data structures for storing the directory. For example, a directory object can support "union mounts" or "filters." A union mount is a merge of a number of other directories, a filter provides access to part of an underlying directory.

An alternative lookup method is used by DNS. In DNS, the original, absolute path name is passed as argument to the lookup request. Each DNS server consumes as much of the absolute path as it can and either returns the requested information or a redirect to a different DNS server. This approach is less suited for a name-space *object* as it requires the object to know its own name to be able to skip the initial part of the path name.

The lookup method presented above allows a name-space object to chose between iterative and recursive name lookups. In an **iterative name-lookup** algorithm, a name lookup is performed one step at a time (i.e., a client invokes a lookup operation on one object, gets a reference to a second object on which the next lookup operation is invoked, and so on). In a system with **recursive name lookups**, the client invokes one operation on the root directory object which invokes a lookup operation on another object. This object, in turn, invokes a lookup operation, and so on, until the name lookup is complete. Both algorithms have one serious disadvantage over the other algorithm. The net effect is that both are needed in different situations.

The major disadvantage of using an iterative lookup algorithm in a wide-area distributed system is the inherent latency associated with this algorithm. An iterative lookup which requires n method invocations on remote objects costs at least n round trip times. The recursive lookup algorithm usually incurs less latency due to locality in the object placement. The major disadvantage of the recursive lookup algorithm is that the cost of actually performing a lookup operation is much higher. Instead of returning information from a table, which is the case for an iterative lookup, the directory object has to invoke a lookup on another object. Popular directory objects, usually the ones close the root, may get overloaded as a result. The lookup interface we presented allows an individual object to decide whether it has sufficient resources to perform a recursive lookup or not. If it does not have the necessary resources it can simply return the result of a simple table lookup.

4.2.2 Location Service

The location service provides a mapping from an object handle to a set of contact addresses. We will give a detailed description of the location service in Section 4.4. In this section we will present the requirements for the location service.

The main functional requirement for the location service is that it is to provide a mapping from an object handle to a set of contact addresses. An **object handle** consists of two parts. The first part is a 32-byte unique object identifier. The second part is used by the location service implementation to quickly locate the set of contact addresses for a particular object. An object handle is created when an object is first registered with the location service, and remains unchanged for the duration of the registration. The interface to the location service has to support five operations: a

Operation	Description
$lookup(h) \rightarrow \{a\}$	Look up a set of contact addresses $\{a\}$ for an object handle h
$register(o) \rightarrow h$	Register a new object o with the location service. This method returns the object handle h
$unregister(h)$	Unregister a destroyed object
$insert(h, a)$	Insert an address a into the set of contact addresses for object handle h
$delete(h, a)$	Delete address a from the set of contact addresses for object handle h

Table 4.3. Location Service operations

lookup operation, registration and unregistration of objects, and changing (insert and delete) the set of contact addresses. The operations are listed in Table 4.3.

The two main reasons for changing a set of contact address are:

- Migration

  An object can migrate on-line or off-line. A mobile agent implemented as an object typically migrates on-line. During migration, such an object first inserts a new contact address into its set of contact addresses and later deletes the old contact address. The original host is expected to forward or redirect any bind requests to the new host. An object located on a notebook is usually disconnected from the network when the notebook is moved to a different location and then reconnected the network. These objects get a new contact address the moment the notebook is turned on again. In this case the set of contact addresses is simply replaced.

- More or fewer contact points

  The set of contact addresses is simply updated to reflect the addresses added or deleted.

In general, we can expect that the complete set of contact addresses of a highly replicated object is too big to be dealt with in one lookup operation. Returning the complete set is not required because we are looking only for a nearby contact point. The addresses in the set of contact addresses vary in two dimensions: the set may include addresses for different protocols, or the distance of the various address with respect to the caller may vary. When the set summarized, it is important to include maximal variation in the first dimension, and also some variation in the second dimension. This ensures that at least one address for each supported protocol is returned. In addition, more than one address for a single protocol can be returned for increased fault tolerance. In practice, we expect that the location service returns a somewhat fixed subset of the actual set of contact addresses.

A major constraint on the design of the location service are the scalability requirements:

- The main scalability requirement for network traffic is locality. Informally, we require that the location service uses only local communication if a process tries to bind to a nearby object.
- Scalability of storage can be achieved if the storage requirements of the location service grows linearly with the number of objects registered.
- Scalability of CPU usage is not expected to be a problem. The location service performs relatively simple operations. We can assume that the location service will be implemented as a distributed service. In such a distributed service, the majority of operations are executed in response to requests that arrive over a network. As a result, the CPU time needed of the location service as a whole is roughly proportional to the total network traffic.
- Scalable (system) administration.

  Ideally, the location service should be designed such that every organization and every individual is able to provide location services for its own objects. This allows an object owner to decide what performance and reliability guarantees are appropriate to locate his objects. It also allows an object owner to buy location services for his objects from a company specialized in location services. The alternative, one or only a few organizations providing location services for the whole world, cannot be considered scalable from a system administration point of view.

  Unfortunately, finding the location service run by of an object's owner is not at all trivial. A separate location service would be needed to locate to object's location service. This brings back the other solution: only one, or a few organizations provide worldwide location services. The result of this is that completely scalable system administration can not be obtained for the location service.

  In general, long term stability can be required only from (semi) government bodies because other organizations can go bankrupt, or they can simply decide not to provide a certain service anymore. On the other hand, small organizations that compete with each other often provide better service than governmental organizations or contractors.

4.2.3 Destination Selection

The destination selection module has to sort the set of contact addresses returned by the location service. Selection criteria include predicted network performance, for both latency and bandwidth, expected reliability of the contact point (a high bandwidth path to a crashed server process is not very useful), and administrative policies.

Decisions can be based on static information about the system, for example the network topology, or on the results of performance measurements. Low-level communication latency can be measured on the Internet using ICMP echo requests and replies, see for example [Golding, 1992a].

A simple destination selection scheme is present in the BIND (Berkeley Internet Name Domain) DNS implementation. The bind resolver that is part of a DNS server keeps track of the latency of DNS requests sent to other DNS servers. When multiple servers for a particular domain exist, the server with the lowest latency has a *high probability* of being used first. Furthermore, if multiple address records are returned as the result of a DNS query, the local resolver in a process sorts those addresses such that an address on an attached subnetwork will be tried first.

A highly replicated object, with a large number of contact addresses, may have a very large set of contact addresses. This set may be too large to be passed efficiently from the location service to the destination selection module. There are two techniques to deal with this problem. One technique is to integrate part of the destination selection module with the location service. Especially, destination selection based on network topology is easy to integrate with certain kinds of location servers. Using this approach, the location service returns only a subset of the set contact addresses. The other technique is the use of a “stream” to move contact addresses from the location service to the destination selection module and further on to the implementation selection module. In this approach, the destination selection module acts more like a filter than a sort algorithm. The destination selection module scans the stream for an address with suitable performance and passes it on to the implementation selection module.

Further research is needed to determine how important destination selection is, and whether performance measurements should be preferred over topological or geographical information.

4.2.4 Implementation Selection

As described in Section 2.3, a protocol identifier specifies the complete protocol that is used to connected to a distributed shared object. This includes not only communication protocols, but also data encoding, security, application-level protocols, etc. Implementation selection involves locating a class object that implements the protocol that is identified by the protocol identifier. In general, we expect the class object to be locally available, either on the same machine or on a nearby file server.

The designer of a distributed shared object can use the protocol identifier for two different purposes. One approach is to use the protocol identifier to identify the complete protocol used by the distributed shared object: the low-level communication protocol, the replication protocol, the encoding of method invocations, how to marshal the state, and the semantics of the object. Another approach is to use a protocol that implements only the initial handshake with the distributed object. After initial option negotiations , the real protocol is selected, and an implementation for this protocol has to be loaded.

There is a trade-off between the two approaches. The first approach leads to a large collection of protocol identifiers but binding does not incur additional overhead and the “type” of an object can be inferred from its set of contact addresses. The second approach results in a much smaller set of protocol identifiers, but an initial handshake is unavoidable, and binding may fail later on if no implementation for the distributed object’s protocol is available. Note that a single distributed shared object is allowed to use multiple protocol identifiers in its set of contact addresses, so, in theory, both approaches can be used by a single object at the same time.

We would like protocol identifiers to satisfy the following, conflicting, set of requirements:

1. The protocol identifier should uniquely identify the protocol to be used.
2. The protocol identifier should be self describing, that is, in the ideal case the protocol identifier should point to a precise description of the semantics of the particular protocol.
3. The protocol identifier should allow run-time protocol composition.
4. The protocol identifier should be small (in number of bits).

The fourth requirement is in conflict with the second and the third. A protocol identifier cannot be both small and descriptive. The choice between satisfying either the second and third requirements or the fourth requirement roughly corresponds to the two different uses for the protocol identifier. It is possible to use small protocol identifiers and sacrifice descriptiveness, if there are only a few protocols to connect to the object. On the other hand, if protocol identifiers describe the complete protocol used by a distributed object then descriptiveness becomes more important.

A protocol identifier that satisfies the first three requirements can be created as follows. First create a distributed shared object that describes the protocol. The description can be an informal, human-readable text, or it can be a formal protocol description. It is also possible that the document refers to existing implementations of the protocol, or describes how to compose an object that implements the protocol from different pieces. This distributed shared object has an object handle, which not only uniquely identifies the document object but also allows binding to the object to read the document. This object handle is an ideal candidate for a protocol identifier.

In addition, to satisfy the fourth requirement, relatively small protocol identifiers for well-known protocols can be allocated by numbering authorities such as the Internet Assigned Numbers Authority (IANA) or the ITU.

Further research is needed to determine how a protocol identifier can be used to obtain a suitable class object. The case were the class object is locally available is relatively easy: a simple table that maps protocol identifiers to class names is sufficient. The case where implementations have to be downloaded from a remote repository is much more complex.

4.3 Local Name Spaces

To invoke a method on an object, the caller needs a pointer to the proper interface of the object. An interface pointer can be obtained through a call to getinterface using a pointer to the standard object interface. In some cases, pointers to the standard object interface are returned by method invocations. For example, invoking the create method on a class object returns a pointer to the standard object interface of the newly created object. However, in most cases, the bind library routine is used to look up a name and return a pointer to (the standard object interface of) the desired object.

The local naming system is designed to provide a flexible way of referring to local objects in the same address space, and, through delegation, to distributed shared objects. The use of a proper naming system simplifies combining separately developed components [Saltzer, 1978].

The bind library routine provides access to:

1. Distributed shared objects.

   These objects have names that can be used in a location-transparent way all over the world.

2. Local class objects.

   These objects support the creation of instances of a particular class.

3. Local instances.

   Objects that are dynamically created by class objects.

4. Subobjects that are part of a composition.

A single, per-process name space is used to access these objects. This name space and the name-space implementation have three features to accommodate the object categories mentioned above: (1) part of the name space can be delegated to an object, (2) the name space provides both absolute and relative names, and (3) the name space can be manipulated through overrides.

Basically, the local, per-process name service provides a hierarchical name space that associates a name with a pointer to a local object's standard object interface. This allows a program to refer to local objects by their names instead of passing around pointers to those objects. Part of this name space can be delegated to a local object.

Name space	Description
/globe/	Worldwide name space for distributed shared objects
/classes/	Class object repository
globe.csRep	A client/server replication class object (example)
/builtin/	(Class) objects linked with an application
memory	memory allocator (example)

Table 4.4. Standard name spaces

For example, the /globe name space of distributed shared objects is delegated to an object which knows how to bind to a distributed shared object.

Names are resolved with respect to a context. A **context** is a set of bindings of names to objects [Saltzer, 1978]. In the local name space, contexts are explicitly associated with local objects, the main program, and with libraries. This is different from most other naming system, which operated more or less independently. For example, the naming system in a UNIX file system is independent of the processes that use the file system.

Absolute names are resolved with respect to a global, per-process context; relative names are resolved with respect to the object's (or library's, or main program's) own context. Every context has an absolute name. For an object, the name of its context is equal to the name under which the object has been registered. Special contexts are created (with their own absolute names) for the main program and for the libraries. From a name-space perspective, binding to an object using a relative name a from a context with name /p/q is equivalent to binding using the absolute name /p/q/a.

An **override** is a feature of the local name space that allows the object associated with a particular *absolute* name to be changed for one (or a few) naming contexts. This means that binding to an object with the absolute name /p/q in one context may return a different result than binding to an object with the same name in a different context.

These overrides allow different parts of a process to use a different implementation for certain functionality. For example, a different object implementing /builtin/memory may be associated with some local object to provide a special high performance memory allocator for a particular object (or to keep track of the memory that object allocates such that it can be deleted when the object is destroyed) An override is similar to a skeleton directory in Jade [Rao and Peterson, 1993].

Class Repositories and Compositions The local name space provides seamless integration for class repositories and compositions with the local object model. A class repository provides access to a collection of object classes which are loaded dynamically into a process' addresses when needed. The class repository is accessed through part of the local name space: /classes. This part of the name space is delegated to an object which has access to a list of class objects on a local or remote file system. When a class object is needed, the application (or the requesting object) passes the name of the class to the bind function (for example /classes/globe.csRep). The class repository object dynamically loads the class object into the address space (if it is not already there) and returns a pointer to the object's standard object interface. The caller does not care whether the class object is already loaded, or whether a class is built-in or should be loaded from a file system, a simple bind using a name in /classes is sufficient.

In some object-oriented programming languages, such as C++, classes and objects are accessible by name only in the program *source*. In the executable process, only pointers to objects are available and classes are completely inaccessible. This makes such a language less suitable for dynamic addition of new classes to a running process due to the lack of a standard mechanism to refer to those classes. The objects that are created dynamically and which are to be shared between independent modules have to be registered with nonstandard, ad-hoc local naming systems.

In contrast, Globe explicitly names local (class) objects. Creating a new instance of a class involves binding to the class object followed by invoking a create method on the object returned by bind. In this way, it does not matter whether the caller tries to create an instance of a built-in class object (for example, /builtin/classes/thread or a dynamically loaded class object (/classes/globe.csRep, a Globe client/server replication object). The same thing holds for nonclass objects such as the default memory allocator (/builtin/memory).

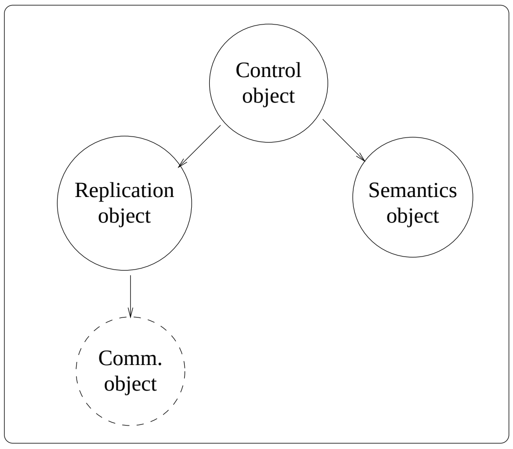


Figure 4.5. A composite object

Compositions use the local name space for two purposes: to keep track of their subobjects, and to allow subobjects to obtain references to each other. The composition in Figure 4.5 contains four subobjects: a control, semantics, replication, and

Name	Value
/.../obj	Pointer to composite object
/.../obj/fCtrl	Pointer to control object
/.../obj/fCtrl/sem	../fSem
/.../obj/fCtrl/repl	../csRepl
/.../obj/fSem	Pointer to semantics object
/.../obj/csRepl	Pointer to replication object
/.../obj/csRepl/comm	../tcp
/.../obj/tcp	/classes/tcp

Table 4.5. Name space of a composite object

communication object. In this example, three of those subobjects (the control, semantics and replication object) are private to this composition and the communication object is shared by other compositions. The arrows in the figure show which objects need references to which other objects. Table 4.5 shows the additions to the name space after an instance of this composition is created. Name-space entries for subobjects are created in the context of the composite object (i.e., if the composite object is called /.../obj and an subobject is called fCtrl then the absolute name of the subobject is /.../obj/fCtrl). In this case, the four subobjects are registered under the names fCtrl, fSem, csRepl, and tcp. The registration of the communication object (tcp) is special: instead of actually creating this object (as is done for the other three subobjects) we install an override to the actual location of the object. This is indicated with the string "/classes/tcp."

In a composition, overrides are used for two purposes. The first purpose is to add a reference to an existing object outside the composition. The reference to the /classes/tcp object in the previous paragraph is an example of this usage. Another use for overrides is to allow subobjects to obtain references to each other. The replication object needs a reference to a communication object, which it tries to obtain by binding to the relative name "comm." The name-space implementation tries to resolve this name in the context /.../obj/csRepl, which results in a reference to the absolute name /.../obj/csRepl/comm. To allow these bind calls to succeed, the composite object has to install overrides in the contexts of its subobjects. The name "../tcp" simply refers to a name-space entry (object or override) called tcp in the parent's context. Note that the overrides that are used in this example are all relative (to the composition's naming context).

The main reason for a fairly complicated, local name space is to provide an application with enough flexibility to be used in a heterogeneous environment. In an heterogeneous environment, we expect an application to run on different operating systems, and to deal with different implementations of distributed shared objects. The local name space in a Globe process combines a single mechanism to refer to other objects with a mechanism to manipulate this name space (overrides).

4.4 Location Service

The functions of the location service that were described in Section 4.2.2 can be divided into three categories: (1) looking up a set of contact addresses for an object, (2) registration and unregistration of an object, and (3) adding/deleting contact addresses for an object. The design of a location service involves a trade-off between the efficiency of the three categories of operations. For example, a faster lookup implementation typically means that adding a new contact address becomes more expensive (for instance, due to replication). More efficient ways of adding new addresses may come at the expense of increased costs of registering new objects.

In general, this trade-off requires optimizing the placement of the information needed to map an object handle to a set of contact addresses such that updates to the set of contact addresses are cheap (i.e., require only a small amount of preferably local communication) and that lookup operations are also cheap. Since updates to the set of contact addresses are initiated by the object itself, it makes sense to store the contact addresses initially in a part of the location service that is near the object. Note that a replicated object with contact addresses at different sites requires storing contact addresses at multiple locations. Furthermore, the location of a mapping should follow a mobile object after migration.

The right trade-off between the cost of lookup operations and the registration costs depends on the use of the object. Some objects are registered once, do not change their contact addresses, and are looked up relatively often (a high read/write ratio for the contact address). These objects benefit from a location server implementation that replicates the contact addresses. In contrast, a mobile agent typically moves very often which suggests moving the contact address with the object at the expense of lookup operations. Highly replicated objects require a location service that can distribute a large set of contact addresses.

A straightforward approach to these trade-offs is to divide objects into different categories and use different algorithms for each category. A hierarchical, DNS like, structure is sufficient for objects that are located on a single site. These objects may have a few replicas, but the replicas should not be far apart. We expect that the majority of the objects will fall in this category. Highly replicated objects require a location service where different sites cooperate in maintaining location information for a single object. Mobile objects require a more centralized approach that allows objects to be found in the network.

The location information of these three categories of objects can be stored in three independent, worldwide location services. A few bits in the object handle can be used to specify which location service is used. A disadvantage of this approach is that an object cannot move to a different category without changing its object handle. Another approach is to design a location service that is able to dynamically adapt to the migration patterns of an object.

Scalability The scalability of the location service requires minimizing the overall cost of using the location service. This cost has various components, such as the communication bandwidth used, the amount of storage used, the CPU cycles needed and the latency observed by a process. Some costs such as communication bandwidth and latency depend partly on how frequently the various methods are invoked. For example, in a system with objects that live relatively long and do not change their set of contact addresses, the cost depends mainly on the execution of the lookup method. In other systems with objects that live for a relatively short time, or that move frequently, the total costs associated with creating and deleting objects and with inserting and removing contact addresses may dominate the total costs of the lookup operations.

In general, we cannot compute reasonable limits on the cost of operations. However, it is possible to relate the cost of a lookup operation to the overall cost of binding. One thing we would like to prevent is that when a process tries to bind to an object, it has to communicate with a location server that is much farther away than the nearest contact point of the object. In that case, the total cost of the bind operation is dominated by the cost of the location lookup. On the other hand, making lookup operations much cheaper (at the expense of modification operations) than the initial interaction with the distributed object is not very useful either. The reason is that location lookup operations are almost always part of a bind operation, and binding typically requires connecting to a contact point.[5] The result of this is that the communication costs of a lookup operation should be comparable to the costs of connecting to the nearest contact point of the distributed shared object. The cost of a lookup operation is allowed to be lower but not at the expense of modification operations.

Comparing Location Servers To compare different architectures for implementing the location service, we select a number of example objects, based on the following criteria:

1. Lifetime

   For a short-lived object, the cost of registering a new object may be a significant fraction of the total location costs for that particular object.

[5]With the exception of certain kinds of caches. But in that case we can cache the location information as well.

2. Migration speed

   An object may not move at all, move occasionally, or move constantly. For an object that moves occasionally we expect that many lookup operations will be performed between two moves. For an object that moves constantly the opposite is true: there are many more updates on the set of contact addresses than that there are lookups. We may also have to consider speed at which the object moves with respect to the network topology. An object like a cellular phone takes many small steps, whereas a mobile agent might move crisscross all over the network.

3. Number of addresses in the set of contact addresses

   A highly replicated object is expected to have a large set of contact addresses whereas a nonreplicated object has a small set.

4. Network area covered by the object.

   An object with contact points that are far apart covers a large part of a network, whereas an object with a few closely spaced contact points does not.

We selected the following five example objects:

1. A temporary object

   This object lives for a short period of time, does not migrate, and has a single contact address.

2. A stationary object

   This object gets one initial contact address which does not change.

3. A migrating object

   This object moves once in a while.

4. A mobile object

   This object moves constantly

5. A highly replicated object.

   This object has a large set of contact addresses and covers a large part of the network.

The following four sections contain four approaches to building location servers: the central server approach, the home-base approach, the distributed search tree, and an integration of the home-base approach with the distributed search tree. Table 4.6 shows the strengths and weaknesses of different approaches.

Architecture	**Object types**				
	Temporary	Stationary	Migrating	Mobile	Highly replicated
Central server	– –	– –	– –	– –	– –
Home-based approach	++	++	– –	– –	– –
Distributed search tree	–	+	++	++	++
Integrated approach	+	+	+	+	+

Symbols: ++ very good, + good, – poor, – – very poor.

Table 4.6. Compatibility of different object types with various location servers

4.4.1 Central Server

The central server concept is the most straightforward way of building a location service. In this approach, all contact information is stored in a single database, on a single, central server. Although the central server is theoretically quite simple, it does not scale well. The lack of scalability may overload the CPU, overload the connecting network, or require too much storage to be connected to a single computer. Furthermore, it introduces a single point of failure.

In general, we can solve these problems by replicating and/or partitioning the database. Replication is an option if read (lookup) operations occur more frequently than write operations. In the case of mobile telephones or mobile agents this is not the case. Partitioning the database is an effective solution for the scalability of the CPU and storage usage. However, partitioning does not solve scalability problems for highly replicated objects because the set of contact addresses is stored in one place.

The main scalability problem is the access to the network. From a network point of view we can either co-locate all partitions (i.e., create a specific place in the network where location information is stored), or we can spread out the partitions over the complete network. In the first case, the location service creates a single hot spot which overloads the network. In the second case, the entire network will be overloaded due to lack of locality.

The lack of locality is inherent in this model. Even if a process tries to bind to an object that resides on the same machine, there will be communication over the wide-area network to (part of) the central server. Optimizing the data placement such that the set of contact addresses for an object is initially close to the object they refer to leads to an alternative to the central server which is the home-based approach.

Another problem with such an architecture is not technical but organizational. A single, worldwide location service based on the central server concept has to be run by a small number of organizations that can provide a uniform service all over the world. Autonomous administration of (part of) the location service is not possible in this model.

In Table 4.6, we see that the central server approach is expected to perform poorly in all cases. The reason is that this approach lacks both locality and distribution.

4.4.2 Home-based Approach

The **home-based approach** is used in a scheme for dealing with mobile computers on the Internet. In this scheme, a mobile computer (notebook) spends most its time at one location, and connected to one particular network. This network is called the notebook's **home network**. When the notebook is moved and connected to a different network, it will continue to use its old Internet address for communication. When packets destined for the notebook arrive on the home network, a special agent captures those packets and tunnels them to the notebook.

The GSM standard for mobile phones uses a Home Location Register (HLR), to locate mobiles phones. The current location of a phone is maintained in the HLR. Usually, the first part of the mobile phone's phone number determines which HLR should be used (i.e., which mobile phone network in what country).

The home-based approach to the location service uses a similar technique. Records in the location service are allocated near the place where an object is created. The location information for the object remains forever at the original location, the home base, even when the object moves.

The home-based approach can be implemented in a completely distributed fashion. The complete state of the location service is partitioned in a large number of **segments**. Every segment stores the mapping for a (relatively small) collection of objects. The segments themselves are connected by a hierarchical structure. This hierarchical structure consists of a root node and a number of **interior nodes**. This hierarchical structure can be specific to the location service or it can be an existing hierarchy, for example DNS.

Figure 4.6 shows a hierarchical location service. Segments and interior nodes are labeled by a sequence of integers. For object 0.1.3.2.4, its set of contact addresses is stored in segment 0.1.3.2.

A design issue for a home-based location service is whether segments and interior nodes are distributed shared objects or not. A location service implementation that implements segments as distributed shared objects stores the set of contact addresses for segment 0.1.3.2 in interior node 0.1.3. Similarly, the contact addresses for interior node 0.1.3 are stored in node 0.1.

This approach has two advantages. Firstly, the implementation can fully use the features of distributed shared objects, especially the flexibility in implementing them.

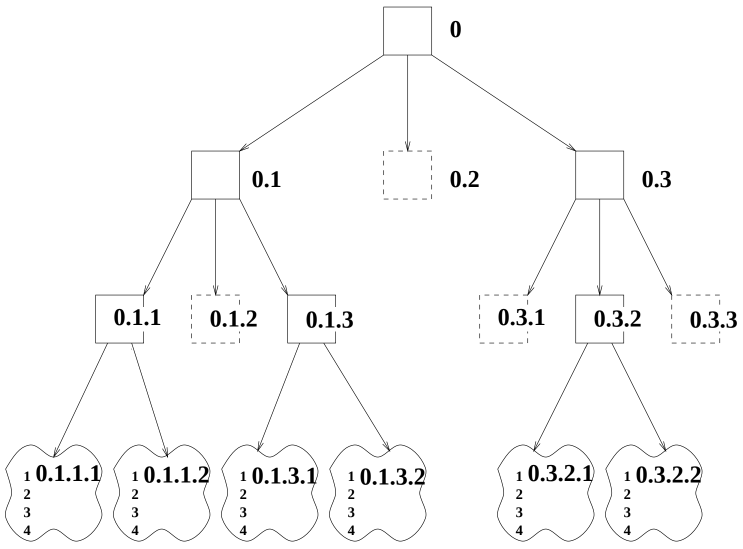


Figure 4.6. A hierarchical location service

The root and other top level interior nodes can use implementations that support replication and caching, whereas leaf nodes use implementation that allow fast and cheap updates. Secondly, it is easy to have a tree with variable height, or to extend the height of the tree because all nodes simply store references to distributed shared objects.

An important aspect of a location service is interoperability. The location service has to be available worldwide, which means that implementations for the communication protocols that are used to access the location service, and the protocols used within the location service have to be sufficiently wide spread such that location-service lookups and registrations do not fail due to protocol incompatibilities. This limits the freedom for implementing distributed shared objects that are part of the location service.

The use of distributed shared objects may also suffer from too much data hiding. For example, a reply to a DNS query (often) contains additional information that was not explicitly requested by the client. The extra information often saves future queries. In an object-based system, additional information would be the result of a method invocation on a different object, which is not available to the object that is currently serving the request.

An alternative approach is to use a single dedicated protocol within the location service. The main advantage is tight integration. However, a single protocol provides only limited flexibility compared to an implementation based on distributed shared objects. For example, it very hard to add support for replication to an existing protocol.

Because the home-based location service is a single, hierarchical structure, the top nodes of the tree have to be provided by an independent organization. However, the actual leaf nodes (segments) and maybe some interior nodes can be provided by individual organizations. The advantage of segments that are provided by individual organizations is that every organization has complete freedom to decide which objects it should make available worldwide by storing the contact addresses of only those objects in the local segment.

A problem with this approach is reliability. The organization where the object was created has to provide a reliable location service for the object, even after an object has moved. In some cases it is possible to anticipate such a situation. For example, the location information for all objects that belong to a single person can be stored in a single segment. When that person moves, the distributed shared object that implements the segment can migrate with the user. New contact addresses have to registered higher up in the location service for the segment's distributed shared object.

The home-based approach performs well when objects do not move and do not have widely distributed replicas. Temporary and stationary objects are given high marks in Table 4.6. Migrating and mobile objects suffer from a lack of locality. All lookup operations and address registrations have to go to the home-base. Highly replicated objects suffer from both a lack of locality but also from a lack of distribution.

4.4.3 Distributed Search Tree

The previous two approaches have two major problems:

1. Information is stored only at the extremes of a tree. The home-based approach stores all information in the leaves of the tree. In contrast, the central server stores all information in one place, which would correspond to storing everything in the root of a tree.

2. The complete set of contact records for an object is stored as a whole in one location.

We can solve both problems by storing information about an object at all levels of a distributed tree. At the leaves of the tree, we store contact addresses in leaf nodes close to the corresponding contact points. In interior nodes, we store pointers to the contact addresses in leaf nodes or to other interior nodes.

Actual contact addresses are found using a hierarchical lookup algorithm. The lookup operation tries to look up an address for an object handle in the closest leaf node. If that lookup fails, the operation is forwarded to the parent node, and so on until a lookup succeeds or the root is reached. If a lookup in the root node fails (authoritatively), the object handle is invalid. 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. The result of a successful lookup in an interior node is a collection of pointers to other nodes, which can be other interior nodes, or leaf nodes. This approach solves three problems: (1) the lack of locality of the lookup operations, (2) the distribution of the contact addresses of highly replicated objects, and (3) the migration of contact addresses of mobile objects.

To actually implement this distributed 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. An object is registered at each site where it has a contact address. Furthermore, each object is registered in all regions that contain a site where the object is registered, recursively up to the top of the tree. The root node stores information about all objects in the entire system.

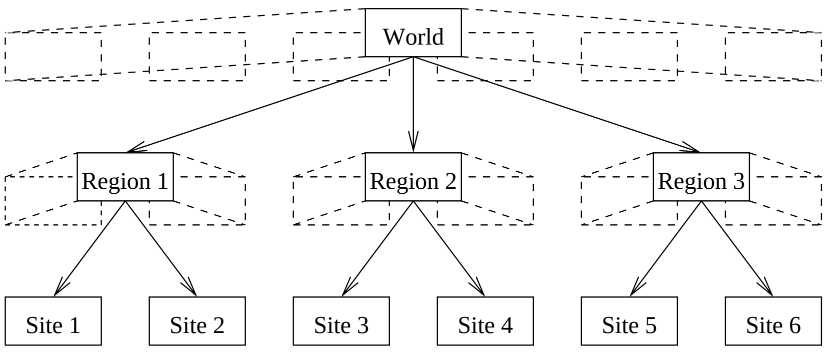


Figure 4.7. Distributed search tree

In this respect, the root node is similar to a central server. The main difference is that the root node stores pointers to other interior nodes (regions) instead of storing the actual contact addresses. A scalable implementation requires the root node, and most other interior nodes to be partitioned. An interior node can be partitioned by distributing its state over its children. This process is repeated until all data is divided over the participating sites. This is shown in Figure 4.7.

We saw in Section 4.4.1 that main problems not solved by partitioning are the large sets of contact addresses to belong to highly replicated objects and the lack of locality that is inherent in the central server model. Both limitations are not an issue for the root node of the distributed search tree. The root node stores only pointers to other interior nodes, so the maximum number of pointers stored for a highly replicated object is bound by the fanout at the root node. Furthermore, locality is already provided for lookup operations by the search mechanism, and update operations mainly take place in leaf nodes.

Unfortunately, the distributed search tree approach has two drawbacks: there is no locality in initial object registration and lookup operations are always hierarchical.

The first problem is that this approach does not provide locality for object creation. The registration costs are mainly an issue if the object is used only locally and is destroyed soon after it creation, as is the case for a temporary object.

Object creation involves creating an entry in the root node. The cost of creating an entry in the root node depends on the organization of the root node. A standard approach to partitioning is to compute a hash function over the index to a table, in this case the object handle. The hash is then used to select a partition. An advantage of using a hash function is that objects are distributed over all partitions and that, with large numbers of objects, a good load balance will be achieved.

If the goal is to achieve locality for initial object registrations, then a home-based approach is much better. The reason is that a home-based approach registers a new object in a part of the location service that is close to the place where the object is created. We can try to emulate the advantages of a home-based with respect to registration using an explicit partition identifier. This partition identifier is added to the object handle and is used to select a partition of the root node. When an object is to be registered, the object registration algorithm selects a suitable (nearby, and not overloaded) partition of the root node, and adds the partition identifier of that partition to the object handle.

A problem with this approach is that the procedure has to be repeated for each level in the hierarchy. Registering in an arbitrary partition of a "regional node" right below the root node may cause the same problems with locality. A clever encoding of partition numbers for all interior nodes might solve this problem. One approach to such an encoding is to use the geographical location of the object and to map geographical locations to partitions in the location service [Ballintijn et al., 1999].

The second problem is related to the search strategy for lookup operations. The search strategy exploits maximum locality by starting at a nearby leaf node working toward the root node. This works well for the system as a whole, but an application which exhibits poor locality is seriously slowed down compared to a home-based approach. The reason is that with a home-based approach, looking up an address requires only one lookup operation on a remote partition of the location service (assuming that contact addresses for that partition are cached). With the distributed search tree approach, for an n level tree, $n-1$ lookup operations are executed before the root node is reached, and then $n-1$ additional lookup operations are required to reach the right leaf node.

Looking up Multiple Protocols The lookup algorithm described earlier in this section returns the first contact address found. This algorithm is too simplistic for two reasons: firstly, contact addresses for different contact points are needed to deal with a network partition or the failure of a contact point. Secondly, multiple contact addresses with different protocol identifiers are also needed in case a particular protocol is not supported. Table 4.7 shows how multiple protocols may introduce locality problems.

		Distance				
		Local	Site	Region	...	World
Protocol	P_1	L_1	S_1	R_1		W_1
	P_2		S_2			
	...					
	P_n					W_2

Table 4.7. Multiple protocols with addresses at various distances

This table classifies the contact addresses for a particular object by protocol identifier ($P_1 \ldots P_n$) and by distance from the perspective of the binding process (local, site, ... world). For P_1, contact points exist everywhere (both locally and far away), for P_2 a contact address exists somewhere in this site but nowhere else. For P_n a contact address exists somewhere in the world. Because contact address L_1 is the only contact address that is locally available, the naive lookup algorithm returns only L_1, and the binding process is not informed about the existence of contact addresses for the protocols P_2 and P_n. Another problem is that the binding process does not know what to do if connecting to L_1 fails: there is no indication that other contact addresses exist.

We can solve these problems in two ways:

1. The location service replicates contact addresses.

   For example, if two contact addresses for each protocol are preferred then a lookup operation returns the set $\{L_1, S_1, S_2, W_2\}$. Note that the contact addresses S_2 and W_2 are the only contact addresses for, respectively, protocols P_2 and P_n. For protocol P_1, the contact address L_1 and S_1 are returned because those two contact addresses are the first ones that are found by the hierarchical search algorithm. This approach has two disadvantages: first, it takes far more storage. Previously, interior nodes would store only pointers to child nodes with contact addresses, now interior nodes also store sets of contact addresses. The second disadvantage is that a low read/write ratio results in poor efficiency. This means that the location service can waste a lot of resources to replicate contact addresses for objects that often update their set of contact addresses, and where lookups are relatively rare (a mobile object for example).

2. The alternative is to replicate metadata.

   The metadata that is replicated includes the set of protocols for which contact addresses exist, and whether remote contact addresses do or do not exist for a particular protocol. This information does not change as often as the actual contact addresses themselves, and inconsistencies are less important.

   With this approach, a lookup would proceed as follows: the first lookup returns the contact addresses L_1, the set of protocols P_1, P_2 and P_n, and the existence of remote contact addresses for all protocols. If the binding process is not satisfied with L_1, a lookup at the site level would return the contact addresses S_1 and S_2, the set of protocols P_1, P_2 and P_n, and the existence of remote contact addresses for P_1 and P_n.

The lookup method may be extended to accept a set of protocols as an argument and return only contact addresses in this set.

The second alternative combined with caching is likely to perform better than the first alternative. The drawback is however that a binding process has to issue multiple location-lookup requests to get a complete set of contact addresses. However, in general, one lookup request is likely to be sufficient for two reasons. The first reason is that processes are typically satisfied with the nearest contact address which is returned in the first lookup. The second reason is that, due to caching, the set may contain popular alternative contact addresses.

A generalization of the second approach is to associate arbitrary metadata with a contact address. Metadata for an object o can take the form of attribute=value pairs and is distributed to nodes that contain information about o. Whether such a generalization would be useful is not clear at the moment and requires further research.

An important addition to a contact address returned by a lookup operation is a distance indication, similar to the distances in Table 4.7, that is whether the contact point can be anywhere in the world, or in the same site. This simplifies the destination selection step.

A location service based on a distributed search tree, is capable of dealing with migration and replication, but due to the relatively high initial registration costs, the performance for temporary objects is rather poor.

4.4.4 Integrated Approach

The main problem with the distributed search tree is object creation. Creating an object requires an entry in the root node in the tree, which is expensive due to both communication and storage requirements. By combining the distributed search tree with the home-based approach, we can lower the object creating cost, at the price of reduced flexibility.

Table 4.6 shows that some object categories are better served by a home-based location service and others are better served by a distributed search tree. It is possible to construct a location service which combines a home-based location server with one based on a distributed search tree. For example, it is possible to register an object with a home-based location service if it is known that the object is a temporary object.

We can combine a distributed search with the home-based approach in two different ways. One approach is a home-based location server attached to the leaves of a

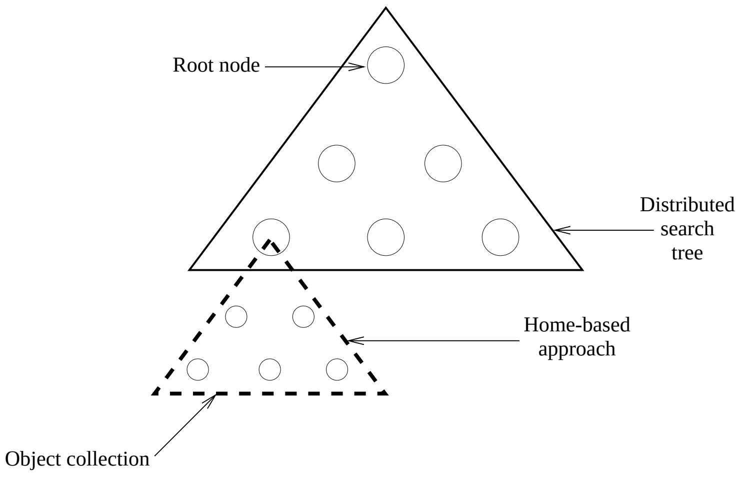


Figure 4.8. A collection of objects

distributed search tree (Figure 4.8). The alternative is a collection of distributed search trees attached to a home-based location server (Figure 4.9).

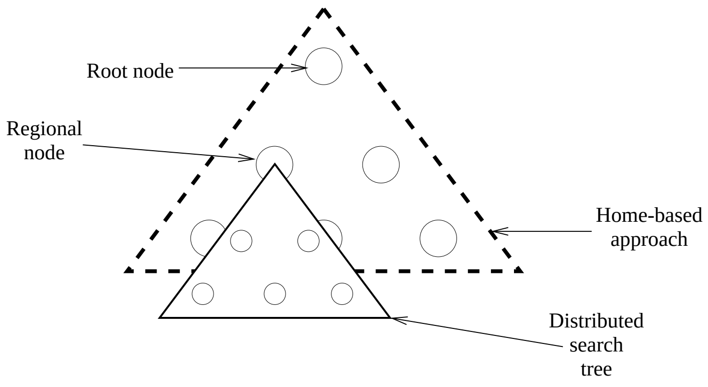


Figure 4.9. Multiple distributed search trees

Attaching a home-based location server to a distributed search tree can be effective when a collection of objects is homogeneous with respect to distribution properties (replication, migration, etc.). Object handles for those objects are located using a home-based approach relative to a single entry in a distributed search tree. An example is a hypertext document which consists of a collection of objects but is treated as a single container object with subobjects. The trade-off here is that independent registration in the distributed search tree is more costly, but allows the object to have different distribution properties.

The other combination (a distributed search tree attached to a home-based server) reduces the cost of registering a new object because only a regional "root" node is updated instead of a single worldwide root node. The object handle is first passed to the home-based service to locate the regional "root" of the distributed search tree. This allows objects to be replicated and to roam within a single region. The drawbacks of the home-based server become a limiting factor, when an object migrates (or gets replicas) outside the region covered by the distributed search tree. In this case, the trade-off is that cheaper registration limits the migration and replication of an object.

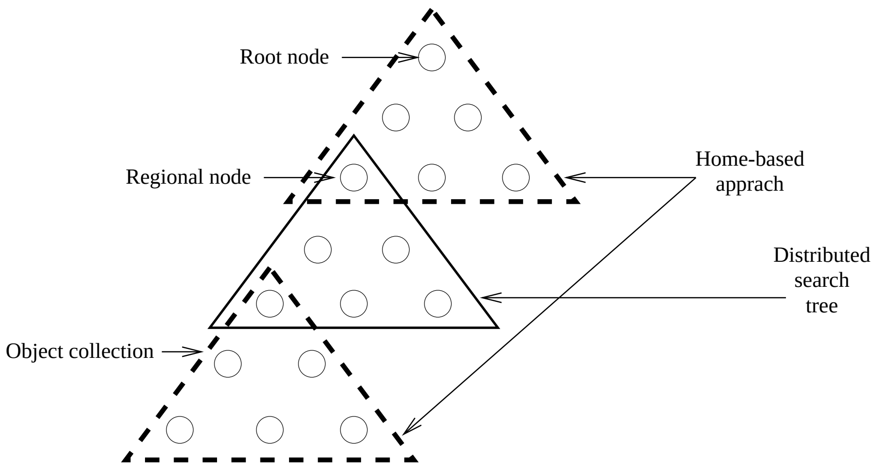


Figure 4.10. A combined location server

One step further is to combine Figure 4.8 and Figure 4.9 into a three-layer hierarchy. This is shown in Figure 4.10. The root of the location service is based on the home-based approach, with distributed search trees attached to nodes of the home-based service. Attached to the search tree is in turn a home-based location server.

The combined location server moves some trade-offs from the design phase of the location service to the deployment phase. For the design it is no longer necessary to choose between a home-based approach or a distributed search tree. Furthermore, deploying an initial, worldwide location service based on a home-base approach is relatively easy because existing services, such as DNS, can be reused for the root of the tree.

Multiple Object Handles The first three location servers (central server, home-based approach, and distributed search tree) treat all objects as equal and use a single object handle for a particular object. To be able to fully use the integrated location server, we have to relax those assumptions.

First, during the initial registration, an object has to decide where to register. For example, a temporary object should register in a part of the location service that is mostly home-based. An object which is part of a collection of objects should register with the (home-based) location service associated with that collection. If we make the wrong decision, two things can happen: the initial registration cost is too high, or the registration is too specific and limits the migration or replication of an object.

Further research is needed to determine whether the garbage collection techniques that will be described in Section 4.5, can provide an object with the ability to start using additional object handles. Additional object handles can be registered in different parts of the location service which better accommodate the object's needs. The garbage collector is useful in this respect because it provides the ability to track some references from the name service to an object. With some extensions to the name service, this allows an object to update its object handle.

If an object has the ability to update its object handle, it is possible for the object to expand its scope by registering in a larger area. This allows an object to select a cheap registration at first and re-register when necessary. Unfortunately, this means that an object can have multiple object handles during its lifetime. As a result, object names have to be used instead of object handles to store references to objects persistently.

4.4.5 Caching

In the previous sections, we described four architectures for the location service but ignored one important aspect: caching. In general we can distinguish between two kinds of caching: caching the results of lookup operations, and delaying write or modify operations (write-back caching). For the location service, there is no point in delaying write operations. Adding new objects or contact addresses should not be delayed unnecessarily. Delayed delete operations may result in attempts to communicate with stale contact points, which should be avoided. As a result, we will consider only read caches for the location service.

Normal caches cache *values*, in the case of the location service a value would be a set of contact addresses for an object. The distributed search tree approach introduces an interesting alternative: finding the right leaf node which contains a contact address for an object is a large part of the work of a lookup operation. A so-called "pointer cache" can be used to cache a pointer to the right leaf node instead of (or in addition to) the actual set of contact addresses. To use the pointer cache, the user of the cache sends a lookup request to the leaf node. The leaf node returns either a valid contact address or an indication that it no longer stores a contact address for the object.

There are two advantages. The first advantage is that the client does not see any stale contact addresses. Either the leaf node returns a valid contact address or it returns an error. The second advantage is that a nonpersistent contact address that is inserted during the activation of a persistent distributed shared object is not ignored, which would be the case if the persistent contact address is cached. A disadvantage is extra communication costs to query the leaf node.

In general, caches have to use a cache consistency protocol to prevent cached data from becoming stale. A cache consistency protocol can be a simple time-out or a more complicated scheme in which a cached value is updated. For the location service the two main inconsistency problems are stale contact addresses and an incomplete set of contact addresses.

- Stale contact addresses may cause binding to the wrong distributed object, or it may be impossible to connect to the contact point designated by the contact address. To detect a connection to the wrong distributed shared object, we require the binding party to send the expected object ID over the newly established communication channel. Upon receipt of a wrong object ID, the distributed shared object returns a failure indication over the communication channel and the binding process continues with the next contact address.

  A failure to connect to a contact point may cause a much longer delay because the time-out needs to be set high enough to be able to reach the extremes of the network before declaring a contact address unusable. Communication failures due to stale contact addresses may occur at different levels. At the network level, the network address may be invalid or the host designated by the network address may be down. At the transport level (for connectionless transport protocols), it is possible that no process listens to a particular port. At the higher levels (session and presentation layers) it is possible that the process at the other end speaks a different protocol, or that the remote process simply does not respond.

  One way to deal with this problem is to overlap multiple connect attempts in time (i.e., a connect attempt to the next contact address is started after a short period of time).

- An incomplete set of contact addresses may cause two problems. The first problem is that a contact address for a nearby contact point is not returned to the binding process. This causes a lack of locality during binding.

  Whether this lack of locality can be corrected after binding depends on the structure of the particular distributed shared object. A simplistic object continues to use the initial connection between the local object in the binding process and the contact point specified by the contact address. Lack of locality during binding results in performance loss for all communication with the distributed shared object. A more complex distributed shared object can tell the local object to connect to a different contact point. Alternatively, the distributed shared object can use a balanced group communication structure based on multicast support at the network level. In that case, the multicast routing protocols optimize the communication paths between different parts of the distributed shared object.

The second problem is that a contact address for a new protocol is made available by the distributed shared object but is not seen by a binding process. This is unlikely to be a serious problem because objects are not expected to introduce new protocols regularly. Instead, multiple protocols are offered from the creation of the distributed shared object, or there may be a long transition period from one protocol to another.

To implement a cache consistency protocol we can use two different paradigms: based on callbacks or based on time-outs. In a scheme based on callbacks, the cache is updated (or invalidated) when a cached value becomes stale. Such a callback requires a list of caches that store a copy of a particular variable or multicast mechanism that reaches all relevant caches. Schemes based on time-outs cache a value only for a certain amount of time. After that time, the cache needs to verify whether the value is stale or not.

For a location service, implementing a consistency scheme based on callbacks is not practical. First, multicasts cannot be used because the number of places where a set of contact addresses may be cached (possibly in every process) is simply too high. The alternative, storing a list of caches that keep a copy of the set of contact addresses is not very practical either. This list can potentially become quite large, and causes extra traffic when a cache has to remove itself from a list after purging a cached value. For schemes based on time-outs we refer to the discussion of time-based consistency models in Section 3.2.2.

4.5 Garbage Collection

Systems that support dynamic allocation of objects (or in general, resources) have to provide a way to deallocate those objects. Objects can be deallocated in different ways [Plainfossé and Shapiro, 1995]:

- Explicitly, under the user's control

  In this case, the user of an object is typically an application program. A system may provide primitives to allocate and free objects (such as malloc and free in ANSI-C). The advantage of this approach is that only a small amount of support from the underlying system is required, and that objects can be freed promptly.

  Disadvantages include the burden on the programmer to free objects at the right moment. Dealing with object references that form cycles and objects shared by different modules is complicated. There is a reasonable chance that the programmer does not free all objects, which results in "memory leaks": garbage accumulating and usable memory being wasted.

- Reference counting

  The system associates with each object a counter that is used to keep track of the number of “users” of a particular object. The counter is incremented when a new reference is handed out, and is decremented when a reference is no longer needed. An object is freed when the reference counter drops to zero.

  There are two main problems with reference counting. Firstly, it cannot be used in a system with cyclic references, and secondly, achieving fault tolerance is hard. The problem with cyclic references is that the reference count does not become zero when the last reference to the cycle is deleted.

  For example, UNIX file systems use reference counting for both files and directories. The first problem is solved by simply disallowing cyclic references. The second problem requires the use of tools such as `fsck` to recover a file system after a system crash. The `fsck` program runs a “mark and sweep” algorithm to recompute reference counts and to break any cycles that may have been created accidentally during the crash.

- Reference listing

  This approach is an extension to reference counting. Instead of the single counter that is maintained with the reference counting approach, an object maintains a list of all its users. Reference listing is more robust than reference counting because listed references can be verified explicitly, that is, the system can recover from a lost message and from crashed users.

  In a worldwide system, reference listing provides a scalability problem: the number of address spaces (users) that refer to an object can potentially be very large.

  Note that reference listing needs to detect unreachable cycles. Two techniques for doing so are:

  1. Trial deletion

     Try to delete an object and see if it becomes unreachable. If so, delete the object. The main problems are the selection of suitable candidates for trial deletion, and, in a distributed system, preventing concurrent attempts in a single connected collection of resources.

  2. Object migration

     This technique tries to migrate objects that are part of a cycle to a single address space. A local garbage collector can be used to reclaim the cycle. The main problem with this approach is again identifying suitable candidates to migrate.

- Tracing garbage collection

  A tracing garbage collector consists of two phases: the first phase marks (or copies) all reachable objects. The second phase deletes all nonmarked, or uncopied objects. Note that the second phase cannot be started before the first phase is finished.

  The advantage of this approach is that unreachable cycles can be collected. However, the main disadvantage in a distributed system is that all processes in the system have to be synchronized and that they have to be alive.

  Tracing garbage collectors can be used to augment reference counting or reference listing. For example, the UNIX fsck program provides the necessary fault tolerance that a reference-counting system lacks.

  In a distributed system, a tracing garbage collector can be run across a small collection of address spaces. This allows unreachable cycles to be collected that fall entirely within the collection of address spaces. By creating hierarchical groups, a scalable garbage collector can be created that collects cycles in larger collections of address spaces [Lang et al., 1992]. However, this works only if the number of objects that have to be traced in larger groups is relatively small. Grouping address spaces this way may be hard to achieve in a wide-area system.

  One problem with tracing garbage collectors in a large distributed system is the relatively long time it takes to run a complete garbage-collection cycle. Although various local optimizations are possible, a complete cycle still requires global coordination. Tracing in groups is effective only if most cycles are created in relatively small groups. A disconnected cycle that is created in a collection of processes that have only a large group in common may continue to exist as garbage for a long time.

  Consider, for example, a distributed shared object that acts like a UNIX pipe. Two processes share such an object to connect the output of one process to the input of another. If those processes are created by a shell script, and have the short life time that is common for some UNIX processes (such as grep, wc, etc.), it is possible that hundreds of those pipe objects are created per minute. These two processes may be executing on computers that are on opposite side of the globe. It is almost impossible to use a tracing garbage collector to verify that no reference to the pipe object exists anywhere in the world after the two processes are terminated.

  This example indicates a mismatch between the rate at which garbage is created (many objects per minute), and the rate of garbage collection (a complete run takes a long time). The problem is that we cannot make assumptions about the locality of unreachable collections of objects. If we assume no locality at all, we are left with a garbage collector which requires all processes in the system to be synchronized and alive.

The purpose of a garbage collector is to reclaim objects that are not reachable from a set of object references. In garbage-collection literature this set is called the **root set** [Wilson, 1992]. Objects that are reachable from the root set are said to be **live**. The other objects, which are not reachable from the root set, are called garbage. A garbage collection algorithm is called **safe** if it does not reclaim objects that are still reachable from the root set. A garbage collector that eventually reclaims all garbage is called **complete**.

4.5.1 Wide-Area Aspects

To be practically useful, a garbage collection has to be both safe and complete. Additionally, a garbage collector for a worldwide system also has to be scalable. In Chapter 6 on related work, we will see that existing garbage collectors for distributed systems are safe but fail to combine completeness with scalability.

In the Globe system, the natural unit of allocation is a distributed shared object. How a (large) distributed shared object performs garbage collection internally will be ignored; instead we will focus on collecting distributed shared objects as a whole.

An important difference between a worldwide distributed system and most other distributed systems is that resources, such as disks, memory, communication bandwidth, etc., are owned by different organizations. These organizations mainly care about their own resources and try to be as autonomous as possible. The result is that a distributed shared object with replicated state may have to deal with different resource owners.

The world of autonomous resource owners conflicts with the safeness aspects of garbage collectors. The garbage collector avoids destroying objects that are still referenced. However, it is quite possible that the owner of the resources wants the object to be destroyed because the resources are needed for something else, even though references to it still exist.

A similar problem exists in the UNIX file system. A user can make a (hard) link to another user's file on the same device. This link prevents the system from deleting the file when the original owner removes its link to the file. In systems with disk quota this can be a problem because the original owner is still charged with the blocks in the file.

To avoid this problem, we can structure the relationship between a resource owner and an object as follows. A **resource owner** is a user that sponsors a distributed shared object by allowing the object to use some of his resources. A resource owner can sponsor an object for some time and decide not to sponsor the object anymore if the object is not needed or when the resources are needed for other objects. The resource

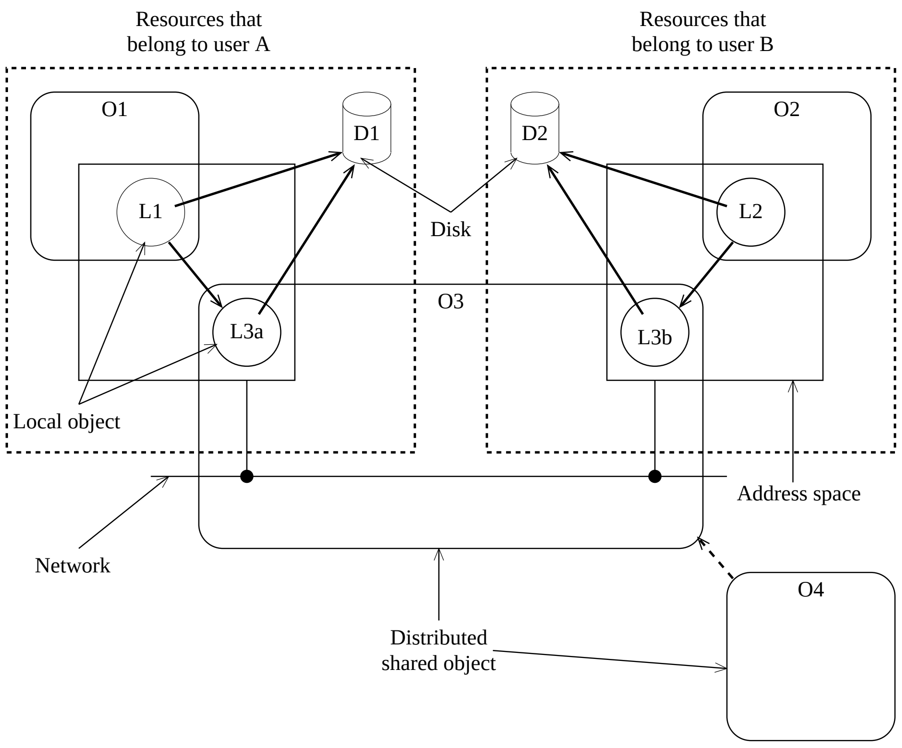

Resources that belong to user A
Resources that belong to user B
O1
D1
D2
O2
L1
L2
Disk
O3
L3a
L3b
Local object
Address space
Network
O4
Distributed shared object

Figure 4.11. Resources and distributed shared objects

owner specifies which objects are worth keeping by adding and deleting references to those objects.

Figure 4.11 illustrates the relation between resource owners and distributed shared objects. This figure shows four distributed shared objects (O1 ... O4). The distributed shared object O1, the local objects L1 and L3a, and disk D1 are owned by user A. Likewise, user B owns O2, L2, L3b, and D2. In this example, the resources provided by users A and B are the disks, the address space, and part of the network.

The distributed shared object O3 uses those resources through the local objects L3a and L3b, which are part of O3. In addition, O3 uses the references from O1 and O2 to determine whether it is still reachable from the root or not. The distributed shared object O4 also contains a reference to O3, but this reference is not used by O3 to determine whether it is still alive or not. After some time, user A may delete the reference from O1 to O3. This causes local object L3a to be deleted because it is completely unreachable (and is, therefore, unreachable from the root set, which is not shown in Figure 4.11); object L3b continues to exists.

For example, a collection of distributed shared objects that contain WWW pages for a single WWW site typically belong to one resource owner. This means that the garbage collector can be used to delete old WWW pages that are no longer referenced within the WWW site. By default, references from other WWW sites are ignored. However, objects can be designed to accept a small number of references from other resource owners as long as sufficient resources (disk space) are available. This allows the garbage collector to be used for references between objects of different resource owners as long as there are enough resources available.

Another possibility is for one resource owner to pay another resource owner for storing reverse references. For example, the author of an on-line paper may wish to keep other papers to which his paper refers, on-line.

4.5.2 Solution

Tracing garbage collection algorithms compute which objects are (not) reachable from the root set. We propose to do exactly the opposite: start with an individual object and try to see whether a member of the root set can be reached.

In a garbage collector for a programming language, the root set is (usually) defined to be all object references that can be found in activation records (the stack) and global variables (the data segment). In Globe, we split distributed shared objects into two groups: objects that are to be garbage collected and objects that have to be destroyed explicitly. Objects know which category they are in, either by their design or as part of their state. We define the root set to consist of all objects that have to be destroyed explicitly. A distributed shared object is not garbage collected as long as it is (indirectly) reachable from an object in the root set.

The garbage collection algorithm will be described as if objects store the references to other objects they need for garbage collection separate from references for other purposes, for example naming. The interaction between references for garbage collection and naming will be discussed at the end of this section.

The garbage collection algorithm for Globe works by trying to find a path from an object to an object in the root set. To make this possible, each object stores a set of reverse references. A **reverse reference** refers to an object thats stores a normal, forward reference to the object.

Figure 4.12 shows a collection of objects and object references. Object 0 is a member of the root set and contains an object reference to object 2. We will call object 0 a **parent** of object 2. Object 2 is called a **child** of object 0. The parents of an object plus the parents of the parents, etc. are called the **ancestors** of the object. In some cases, one parent is special. That parent will be called the **primary parent**. The role of the primary parent will be discussed later. Both object 1 and object 2 contain a reverse reference to object 0. The reverse reference from object 2 complements the

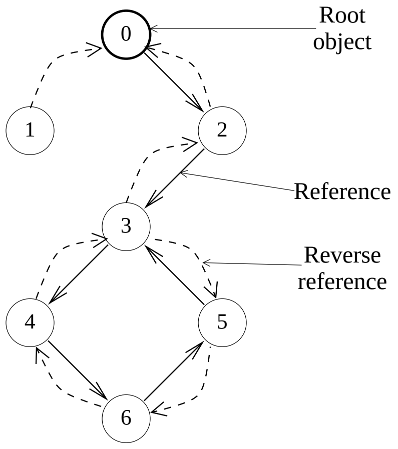


Figure 4.12. Collection of objects

forward reference from object 0. In contrast, the reverse reference from object 1 is stale, and will be deleted by the garbage collection algorithm.

Objects 3, 4, 5, and 6 form a cycle. This cycle is connected to the root node through object 2. Note that in this example, object 3 is the only object with two parents (object 2 and object 5).

A path to a root object is a sequence of object identifiers. This sequence contains the object identifier of a parent object, of the parent's parent, and so on. For example, the path to the root for object 4 is the sequence of object identifiers for the objects 3, 2, and 0.

Storing these reverse references can be a problem if there are a huge number of forward references to an object. We think this is not a problem for two reasons:

1. Only references maintained by resource owners that sponsor an object need reverse references. A study of our local UNIX and Amoeba file-systems shows that less than 1% of the files (and under Amoeba, also directories) has more than 1 link. The number of objects with more than 10 links is negligible. Even though the cost of a link is small in these two systems (just a directory entry), most files would need only one reverse reference to support the Globe garbage collection algorithm.

   These results are relevant to Globe if we can assume that most distributed shared objects do not contain references to other distributed shared objects (and are connected to the root set by directory objects). On the other hand, many WWW documents contain references to the "home page," which may lead to a different distribution of reference counts.

2. Objects that have a huge number of references worldwide (for example saved news articles) also have a large number of replicas. So the number of reverse references per replica is still small. Each replica stores only the few reverse references for its users. This implies that for those objects, instead of running the garbage collection algorithm once per object, each replica has to run the garbage collection algorithm itself.

4.5.3 Algorithm

The garbage collection algorithm is based on the principle that each distributed shared object independently computes whether it is reachable from the root set or not. An object verifies its reachability on a regular basis or when triggered from the outside, for example by a resource manager. To verify its reachability, an object asks its parents for reachability information and makes a decision based on the information received. Possible outcomes are that the object is reachable, that the object is unreachable, or that the information is incomplete or inconclusive. In the latter case, the object waits for a while and asks the parents again. Effectively this boils down to polling the parents until a decision can be made. In general, the use of polling is avoided because it wastes resources. However, in this case the use of polling simplifies the algorithm and provides some degree of fault tolerance. In an actual implementation it should be easy to replace polling with callbacks or operations that block until new results become available.

The presentation of the algorithm is split into three parts. The first part describes the **path algorithm**, an algorithm that tries to find a path to the root set. The object is reachable if a path to the root can be found. However, the path algorithm does not terminate if the object is part of an unreachable cycle. The second part describes a **graph algorithm** that allows an object to find out if it is unreachable from the root set. The algorithm that will be described in the second part works only if no reference to objects are inserted or deleted during the execution of the algorithm. The third part describes an extended version of the graph algorithm that can handle this kind of concurrency.

Initially, the algorithm will be described without regard to safety or completeness. The safety and completeness of this algorithm will be described in Sections 4.5.6, 4.5.7, and 4.5.8.

The first part, the algorithm that tries to find a path to the root, is relatively simple. Each object that knows a path to an object in the root set stores that path as a sequence of object identifiers, which consists of the object's own identifier and the identifiers of the object's ancestors on the path to the root including the member of the root set. To find a path to the root, an object asks each of its parents for their paths to the root. An object has found a path to the root if it gets a sequence of object identifiers that does not already include its own object identifier. The object appends its own identifier to the sequence and stores it as part of its (persistent) state. Upon request, the object returns this sequence to its children.

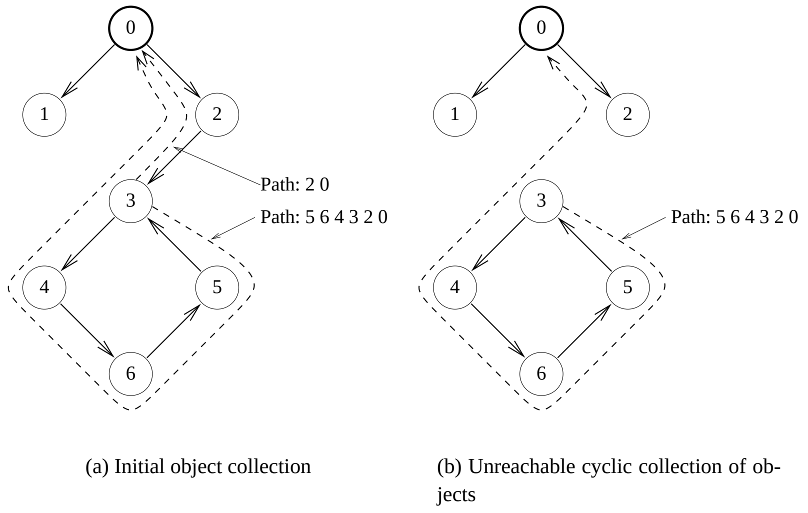

0
1
2
3
4
5
6
Path: 2 0
Path: 5 6 4 3 2 0
0
1
2
3
4
5
6
Path: 5 6 4 3 2 0
(a) Initial object collection
(b) Unreachable cyclic collection of objects

Figure 4.13. Avoiding cyclic paths

Figure 4.13 illustrates the need to avoid paths that already contain the recipient's object identifier. Figure 4.13(a) shows a root object (object 0) with references to object 1 and object 2. Object 2 contains a reference to object 3. Object 5 also contains a reference to object 3. Object 3 is offered two paths: one that lists object 2 and object 0 and another path that lists objects 5, 6, 4, 3, 2, and 0.

In Figure 4.13(b), the reference from object 2 to object 3 is deleted. However, object 5 still claims to know a path to the root. Object 3 can reject this path because it already contains the object's object identifier.

The algorithm consists of three steps:

1. The first step is to verify the set of reverse references. All reverse references that either refer to a parent that no longer exists, or to a parent that deleted the forward reference, are removed from the set of reverse references. The object is unreachable when its set of reverse references becomes empty.

   The set of reverse references becomes empty if the object has no parents. For example, this is the case for object 1 in Figure 4.12 after the reverse reference to object 0 has been deleted.

2. In the second step, the object invokes a method on each of its parent objects to get their paths to the root. Each method either returns a possible path to the root or an indication that the parent does not know a path to the root. It is also

possible that the method invocation fails, for example due to a network partition. In that case, the algorithm proceeds as if the parent does not know a path to the root. Note that these failures cannot cause an object to be deleted prematurely because only the graph algorithm can declare objects to be unreachable.

3. The third step is to discard any paths that contain the object's own object identifier. A path that already contains the object's own object identifier is not a true path to the root but a path that the object reported earlier to one of its children and has returned to the object through a cycle.

   Any path that has not been discarded can be used as a valid path to the root and is stored, after adding the object's own identifier, in the object's (persistent) state. Otherwise, the algorithm has failed to find a path to the root.

These three steps are executed repeatedly until a path to the root is found.

To avoid asking all parents for their paths to the root each time the algorithm runs, the algorithm is extended with a cache. The object handle of the parent that returns the path to the root is cached in the object's state. This parent is called the **primary parent**. The next time the algorithm runs, only the primary parent is asked for a path to the root, and only the existence of a forward reference from the primary parent is verified. Note that the path contains *object identifiers* and we need an *object handle* to actually contact the primary parent. The cache will be cleared and the full path algorithm will be executed if either the primary parent does not return a useful path or if the primary parent has deleted its forward reference.

The graph algorithm is the most complicated part of the complete garbage collection algorithm. Based on the reverse references we can define a **reverse-reference graph**. Nodes in this graph are object and (directional) edges are formed by reverse references from one object to another. Each node that participates in this algorithm tries to build a complete view of the reverse-reference graph. Normally, this graph is far too large to be stored in a single address space. However, if the object is part of a disconnected subgraph, then the graph will be much smaller. There are two mechanisms to limit the amount of data that an object will receive if it is in fact reachable from the root. First, objects that know a path to the root do not participate in the graph algorithm and second, information flows only from a parent to its children. This means that an object can receive information only from ancestors that do not know a path to the root.

The data structure that is used to represent the reverse-reference graph is a set of reverse-reference records. Each **reverse-reference record** contains information about one object: the object's object identifier and a set of object identifiers of the object's reverse references.

The graph algorithm consists of a graph distribution algorithm and a termination detection algorithm. The graph distribution works as follows. Initially, each object that participates creates a record for itself. This record contains the object's object identifier and the set of object identifiers of the object's parents. Repeatedly, the object asks all its parents for their sets of records. The records returned by the parents are added to the object's current view of the graph. Upon request, the object provides its children with a copy of its set of records.

The object runs the termination detection algorithm each time it changes its set of records. The algorithm can terminate when the object has a record for each of its ancestors. Initially, the object has a record only for itself. It cannot terminate until it has received a record for each of its parents. After receiving records from its parents, the object now has to wait for records for its parent's parents, and so on. As said before, an object that knows a path to the root (including root objects), does not participate in the graph algorithm. This means that the graph algorithm does not terminate if one of the object's ancestors is a member of the root set since the object has to object a reverse-reference record from a member of the root set but members of the root set do not provide one.

The path algorithm and the graph algorithm nicely complement each other. The path algorithm is mainly useful if a path to the root set exists. The main drawback of the path algorithm is that it does not terminate if the object is part of a cycle that is unreachable from the root. The graph algorithm is the opposite: the graph algorithm terminates if the object is not reachable from the root set. However, if the object is in fact reachable from the root set, then some object will not cooperate (objects in the root set and any other object that knows that it is reachable) and the graph algorithm will try forever to obtain a complete view of the reverse-reference graph. Running the two algorithms in parallel, in a single integrated algorithm, results in an algorithm that terminates when either a path to the root set has been found or when the object is known to be unreachable.

The graph algorithm that was just described, works only if the graph remains constant during the execution of the algorithm. In a worldwide system, we cannot make this assumption. It should be possible to add and delete object references during the execution of the garbage collection algorithm. The last part of the description of the garbage collection algorithm describes how the graph algorithm can be extended to handle this situation.

To obtain a consistent view of a reverse-reference graph that changes during the execution of the graph algorithm, it is necessary to add two fields to each reverse-reference record. The two fields that are added are a version number and a flag. Each object associates a version number with its set of reverse references. This version number is incremented each time the object changes its set of reverse references. The object keeps the record for itself up-to-date with respect to the set of reverse references and the version number. The use of the flag will be described later on. Note that the object changes only the record for itself in its own set of records. The object's new record is passed to the object's children whenever the children ask for it.

The version number is added to make a distinction between older and newer versions of the record for an object. When an object receives a set of records from its parent, it adds new records and it updates records that have a newer version number than the existing record. Records with older versions are ignored. The version number ensures that, eventually, all nodes get an up-to-date view of the graph.

Unfortunately, even with version numbers, a single object may use inconsistent data to decide that it is unreachable. For this reason, we need to extend the termination detection algorithm to become a distributed termination algorithm. This can be achieved by using the flag that was added to each record. This flag is called the **complete flag**.

Initially, each object creates a record for itself with the complete flag set to false. These records are distributed as described before, until an object discovers that it has records for all of its ancestors. At that point it sets the complete flag in its own record to true and continues propagating records to its children.

At some point it starts receiving records from its parents that also have the complete flag set to true. The object sets the complete flags for the corresponding records in its set of records to true. However, the object has to check whether the records it received from a parent have the same version numbers as the corresponding records in its own set of records. If some of the records have different version numbers, the object has to set all complete flags in both the received set and the object's own set of records to false. This ensures that complete flags are propagated only from a parent to a child if the parent and the child have the same view of the world. Setting the complete flags to false requires the object and the object's children to wait until new records are received from all ancestors.

Clearing all complete flags may seem a bit much; maybe we should clear only the flags in a few relevant records. The problem is, however, that an object does not know which flags to clear. Consider the following example. An object detects that one of its parents deleted a forward reference to the object. As a result, the object deletes the parent from its set of reverse references and increments its version number. This new record is distributed to other objects. It is possible that the first object's parent deleted the last remaining reference to the collection of objects, which turns the objects into garbage. It also possible that a new reference to the collection was added *before* the other reference was deleted. To distinguish between those two cases, each object has to ask all other objects for their sets of reverse references. This is implemented by clearing all complete flags.

The graph algorithm terminates when all records for the object's ancestors have their complete flag set to true. At this point, all of the object's ancestors also have a complete set of records for all of their ancestors. Note that in case of a cycle, all objects that are part of the cycle have the same ancestors (all objects in the cycle), and therefore the same set of records.

Variable	**Description**
object-id	The object identifier of the object.
forw-ref-set	A set of forward references (object handles) to objects.
rev-ref-set	A set of reverse references (object handles) to parent objects.
path-known	A boolean variable that indicates whether a path to a root object is known or not.
primary	One of the reverse references is designated to be the primary reverse reference.
path	A path to the root object. The path is a list of object identifiers.
version-nr	A version number used to propagate reachability information.
graph	The current set of records for the disconnect subgraph algorithm.
root	A boolean variable that indicates if the object is a member of the root set.

Table 4.8. Instance variables used by the garbage collection algorithm

The path that is selected in the path algorithm is added to the (persistent) state of the object to speed up future runs of the garbage collection algorithm. In addition to the path, the object also stores the object handle of the parent that returned the path. This parent was called the primary parent. As described above, the garbage collection algorithm starts by verifying the cached path. If no path is cached, this step can be skipped and the garbage collection algorithm continues with the path and graph algorithms. To verify the cached path, the object binds to its primary parent and asks the parent to return its path. The cached path is invalid if the object fails to return a path or if the path is not identical to the path stored in the object. If the path is valid, the object has to verify that the parent object is still a parent. The object invokes a method on the parent object to ask whether the parent stores a forward reference to the object. After a positive reply from the parent, the garbage collection algorithm returns that the object is reachable from the root set. Otherwise, the cache algorithm failed, the cached path is deleted, and the garbage collection algorithm continues with the path and graph algorithms.

4.5.4 Implementation

The implementation of the garbage collection algorithm in a distributed shared object uses a number of methods and instance variables. The instance variables used by the garbage collection algorithm are listed in Table 4.8. The variable forw-ref-set contains the set of object handles for other objects. Normally, this variable is shared with other parts of the object that actually use those references. The variable rev-ref-set stores

Method	**Description**
add-rev-ref(object-handle)	Add a new reverse reference.
check-rev-ref(object-handle)	Tell an object to check if a parent object is still storing a forward reference to the object.
check-forw-ref(object-id)	Return whether any of the object handles in forw-ref-set contains the object identifier object-id.
get-path	Return the current path or graph
reachable(interval)	Asks an object whether it is still reachable. This is the main garbage collection method.

Table 4.9. Methods that are part of the garbage collection system

a collection of object handles for parent objects that may contain references to the distributed object.

The next three variables are used to store the current (cached) path to the root. The variable path-known is a boolean variable that specifies whether the object knows a path to the root set or not. The variable primary contains the object handle of the parent object that returned the path (the primary parent). The variable path stores the path itself as a list of object identifiers.

The next two variables are used for the graph distribution. The variable version-nr contains the current version number of the object's record in the graph. The variable graph contains the current set of records used in the graph algorithm. This variable is cleared when a path to the root is found.

The last variable, root, is a boolean variable that specifies if the object is a member of the root set. An object that is a member of the root set, has to set the variable path-known to true, and the variable path should be set to an empty list of object identifiers.

Table 4.9 lists the methods that are used for garbage collection. The method reachable implements the garbage collection algorithm. This method invokes the methods check-forw-ref and get-path on parent objects to verify reverse references and to obtain reachability information, respectively. The methods add-rev-ref and check-rev-ref are used by users (owners) of the object to maintain the set of reverse references.

The add-rev-ref method adds a reverse reference to the variable rev-ref-set. The security (access control list) of the object should be such that only resource owners that sponsor the object can invoke this method. After adding the reverse reference to rev-ref-set, the method increments version-nr and, if graph is not empty, updates the object's record in graph and sets all complete flags to false.

The check-rev-ref method should be invoked after the object's object handle is removed from the parent object's forw-ref-set variable. This tells the object to verify that it is still reachable and allows unreachable objects to be deleted promptly. The object can return immediately if the object handle that is passed as a parameter, is not listed in the object's rev-ref-set. The object verifies the reverse reference by binding to the parent object (using the object handle) and invoking the check-forw-ref method on the parent object. An object is allowed to ignore invocations of this method. Obsolete reverse references will be deleted by the garbage collection algorithm.

The check-forw-ref method is invoked on a parent object to verify that a reverse reference is still valid. The parent returns whether it stores a reference to the object or not.

Depending of the value of the path-known variable, the get-path method returns either the current path with the object's identifier prepended (if path-known is true) to the root or the current set of records for the graph (otherwise).

The method reachable implements the garbage collection algorithm. This method computes whether the object is connected to the root set or not. The implementation of the algorithm consists of 12 steps. The first step limits the amount a time a single distributed shared object spends on garbage collection. Within a single distributed shared object, only one thread at a time can execute the garbage collection algorithm. Furthermore, the interval parameter specifies how long the result of a previous run is considered valid. Steps 3 verifies a cached path. The remaining steps execute the path and graph algorithms in parallel. Step 7 executes the path algorithm. Steps 8 and 9 update the graph variable. Steps 10 and 11 compute whether the set of records in graph is complete and step 12 suspends the execution for a while and jumps back to step 5 to retry the path algorithm and continue the graph algorithm.

1. Return an error[6] to the caller if another thread is currently executing this method. The parameter interval indicates the caller's idea of how frequently the object should garbage collect. Return that the object is reachable if the variable path-known is true and the path has been checked less than interval seconds ago.

2. The verification of the cached path can be skipped if the object does not know a path to the root. If path-known is false, go to step 4.

3. Verify that the primary parent still knows a valid path to the root and that the parent also contains a valid forward reference. Use the object handle stored in primary to bind to the primary parent. Invoke get-path and check-forw-rev(object-id) on the parent. The object is reachable if the result of get-path is a path, does not include our object identifier, and check-forw-rev returns true. Update path if the path returned by the parent is different from the current value of path. Return to the caller if the object is reachable.

4. Set path-known to false and clear path. Initialize graph with a single record, containing our object-id, the object identifiers in rev-ref-set, the version number version-nr, and the complete flag set to false.

[6]An error is returned because information about the reachability of the object incomplete. Alternatively, it is possible to return that the object is reachable until it is certain that the object is unreachable.

5. Invoke check-forw-ref(object-id) on all parent objects listed in rev-ref-set. Delete all entries from rev-ref-set for parent objects that return false. It is possible that rev-ref-set becomes empty. In that case, the object is unreachable and can be garbage collected. Indeed, if none of an object's parents still points to it, then it is garbage.

6. Verify that the set of object identifiers in the object handles in rev-ref-set matches the set object identifiers in the object's record in graph. In case of a mismatch, increment version-nr, update to object's record in graph, and set all complete flags to false.

7. Invoke get-path on all parent objects in the rev-ref-set.

   The result of a method invocation can be a path, a set of records or an error. In case of a path, we can select the object returning the path as the new primary parent (provided that the path does not contain our own object identifier). The object stores the received path in path and the object handle of the parent object that returned the path in primary. Furthermore, path-known is set to true, graph is cleared, and the caller is informed that the object is still reachable .

   Any paths that include the object's own identifier are ignored.

8. Every set of records that is returned by get-path is first checked for completely new records and for records that have a corresponding record (for the same object) in graph but with different version numbers. All complete flags (both in the received records and in the object's current set of records) are cleared if one or more of such records are found.[7]

9. Add new records to graph and update records with newer versions, or with complete flags that are set to true. Records for objects which are not ancestors according to the current information about the graph, are deleted.

10. Compute whether our graph is complete or not. Set the compete flag to true in our record, if the graph is complete.

11. Check the complete flags in all records. We can conclude that the object is unreachable if the complete flags are set to true in every record in the graph.

12. At this point we have an incomplete graph. Wait for a while, for information to propagate and for transient failures to be resolved, and go back to step 5.

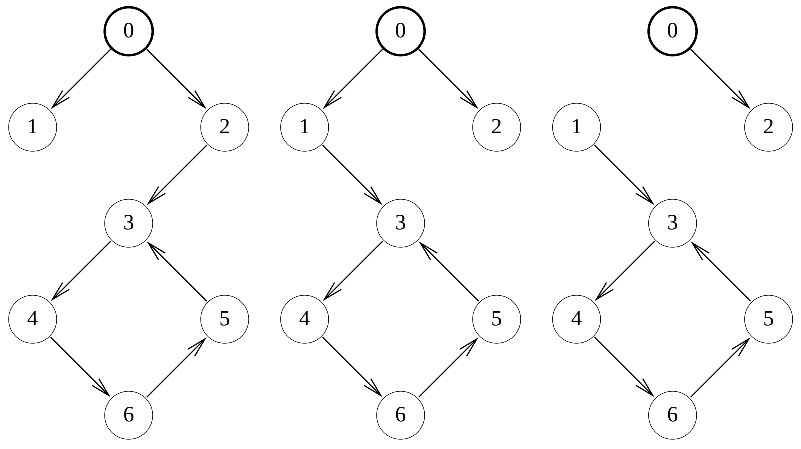
[7]The reason for clearing all complete flags is that the complete flags are used to verify that all objects in the collection have the same view of the graph. If the parent has records with different version numbers, then we have to update the relevant graphs first before we continue with the propagation of complete flags. In theory, we have to clear the complete flags in the received set of records only if that set contains records with lower version numbers. This optimization is probably not worth the extra effort.

An object may optionally invoke reachable(interval) on any parent that returns a path that contains the object's own object identifier, to trigger a garbage collection cycle on that parent. The interval parameter should be equal to the interval parameter that was passed to the object itself.

4.5.5 Example

To illustrate the garbage collection algorithm we analyze two examples. The first example shows the basic operations of the garbage collection algorithm, finding a path to the root and detecting a disconnected subgraph. The second example demonstrates the need for the complete flags in the graph algorithm.

The first example is shown in Figure 4.14. The first figure, Figure 4.14(a), shows the initial configuration. Objects, numbered 0 through 6, contain references to each other. Object 0 is the only element of the root set. Note that reverse reference are not shown in this figure, but they exist. Every reference becomes a reverse reference when the direction of the arrow is reversed.



(a) Initial object collection

(b) Moved object 3

(c) Unreachable, cyclic collection of objects

Figure 4.14. Example

At some point in time, a reference to object 3 is added to object 1 and the reference to object 3 is deleted from object 2. The result of these two actions is shown in Figure 4.14(b). Figure 4.14(c) shows the graph after deleting the reference to object 1 from object 0. This creates an unreachable collection of objects that contains a cycle.

Time	t_0						
OID	0	1	2	3	4	5	6
rev-ref-set	{}	{0}	{0}	{2,5}	{3}	{6}	{4}
path-known	T	T	T	T	T	T	T
primary		0	0	2	3	6	4
path		0	0	0,2	0,2,3	0,2,3,4,6	0,2,3,4
version-nr	1	1	1	1	1	1	1
graph							
root	T	F	F	F	F	F	F

Table 4.10. Initial configuration

Time	t_1	t_2	t_3	t_4	t_5
OID	3	3	4	6	5
rev-ref-set	{1,2,5}	{1,5}	{3}	{4}	{6}
path-known	T	T	T	T	T
primary	2	1	3	4	6
path	0,2	0,1	0,1,3	0,1,3,4	0,1,3,4,6
version-nr	2	3	1	1	1
graph					
root	F	F	F	F	F

Table 4.11. Moving an object reference from one object to another

Table 4.10 shows the garbage collection state of the objects in Figure 4.14(a). Table 4.11 shows what happens after the reference to object 3 has been moved to object 1. At time t_1, the new reference from object 1 to object 3 is created. Object 3 is informed about the new reference (a call to add-rev-ref), and adds the reference to rev-ref-set. At sometime between t_1 and t_2 the reference from object 2 to object 3 is deleted. At time t_2, object 3 starts a garbage collection cycle and discovers that the reference from object 2 has been deleted. Fortunately, the reference from object 1 can be used as a primary reference. The fields primary, path, and version-nr are updated. Note that version-nr is updated each time the set of reverse references changes. At some point in time (t_3), object 4 discovers that its primary, object 3, has a new path. Object 4 can simply update its path field. Object 6 and 5, in that order, also update their path fields.

Tables 4.12 and 4.13 show what happens after the reference to object 1 is deleted. First (time t_6), object 1 discovers that it is no longer referenced by any object. After object 1 has been deleted, object 3 tries to find a new path to a root object. Unfortunately, the only parent is object 5, which can not be selected as primary (the path provided by object 5 already contains object 3, see time t_5). Object 3 sets path-known

Time	t_6	t_7	t_8	t_9
OID	1	3	4	6
rev-ref-set	{}	{5}	{3}	{4}
path-known	F	F	F	F
version-nr	2	4	1	1
graph		{3, {5}, 4, F}	{3, {5}, 4, F}, {4, {3}, 1, F}	{3, {5}, 4, F}, {4, {3}, 1, F}, {6, {4}, 1, F}
root	F	F	F	F

Table 4.12. Graph distribution (1)

Time	t_{10}	t_{11}	t_{12}	
OID	5	3	4	
rev-ref-set	{6}	{5}	{3}	
path-known	F	F	F	
version-nr	1	4	1	
graph	{3, {5}, 4, F}, {4, {3}, 1, F}, {5, {6}, 1, T}, {6, {4}, 1, F}	{3, {5}, 4, T}, {4, {3}, 1, F}, {5, {6}, 1, T}, {6, {4}, 1, F}	{3, {5}, 4, T}, {4, {3}, 1, T}, {5, {6}, 1, T}, {6, {4}, 1, F}	
root	F	F	F	
Time	t_{13}	t_{14}	t_{15}	t_{16}
OID	6	5	3	4
rev-ref-set	{4}	{}	{}	{}
path-known	F	F	F	F
version-nr	1	2	5	2
graph	{3, {5}, 4, T}, {4, {3}, 1, T}, {5, {6}, 1, T}, {6, {4}, 1, T}			
root	F	F	F	F

Table 4.13. Graph Distribution (2)

to false, and initializes graph. The record in the graph ({3, {5}, 4, F}) consists of the object ID (3), the set of parents ({5}), the current version number (4), and the value of the complete flag (**F**alse). This record is returned to object 4 as the result of its invocation of method get-path on object 3. Object 4 creates its own record, and returns the two records to object 6. At time t_{10}, object 5 is the first object to have a complete graph. It sets the complete flag in its entry to true. The graph is passed to objects

3, 4 and 6, which also set the complete flags in their records. Object 6 is the first object to have a complete graph with all complete flags set. At this point object 6 is deleted because the entire collection of objects is no longer reachable. After object 6 is deleted, object 5 discovers that it is no longer referenced. In a similar way, object 3 and 4 discover that they are no longer referenced after, respectively, objects 5 and 3 are deleted.

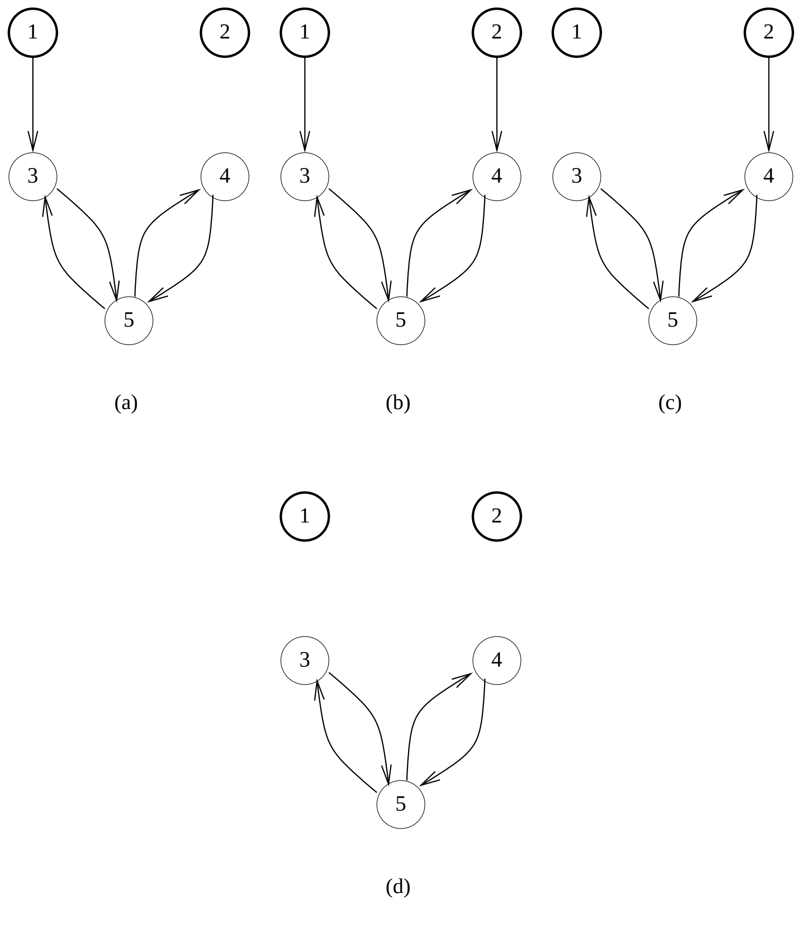


Figure 4.15. Example with complete flag

To illustrate the need for the complete flag in the graph distribution algorithm, we give a second example using the object configurations shown in Figure 4.15. Figure 4.15(a) shows two root objects (object 1 and object 2), two objects that form a cycle with object 5 (object 3 and object 4) and a reference from root object 1 to object

3. Figure 4.15(b) shows an additional reference from root object 2 to object 4, and Figure 4.15(c) shows the result of deleting the reference from object 1 to object 3.

The goal of the complete flags is to avoid the situation that one of the objects believes that the current configuration is the one shown in Figure 4.15(d) even though the actual configuration is one of the other three configurations. To show the need for the complete flag, we assume certain kinds of network failures. In particular we assume that object 5 cannot communicate with object 3 whenever object 1 has a reference to object 3. This means that object 5 never learns the path to the root through object 3. We make the same assumption for object 4 and the reference from object 2 to object 4.

Time	t_0				
OID	1	2	3	4	5
rev-ref-set	{}	{}	{1,5}	{5}	{3,4}
path-known	T	T	T	F	F
primary	—	—	1	—	—
path			1		
version-nr	1	1	1	1	1
graph	—	—	—	{4,{5},1,F}	{5,{3,4},1,F}
root	T	T	F	F	F

Table 4.14. Complete flag example — initial configuration

Table 4.14 shows the initial state of the objects for the configuration as shown in Figure 4.15. Objects 1 and 2 are root objects, object 3 knows a path to the root (object 1), and objects 4 and 5 have initialized their graphs. At time t_1 in Table 4.15, objects 4 and 5 exchange graph records.

Between time t_1 and time t_2, a reference is added from object 2 to object 4 and the reference from object 1 to object 3 is deleted. At time t_2, object 4 knows a path to the

Time	t_1		t_2	
OID	4	5	4	3
rev-ref-set	{5}	{3,4}	{2,5}	{5}
path-known	F	F	T	F
primary			2	
path			2	
version-nr	1	1	3	2
graph	{4,{5},1,F} {5,{3,4},1,F}	{4,{5},1,F} {5,{3,4},1,F}		{3,{5},2,F} {4,{5},1,F} {5,{3,4},1,F}

Table 4.15. Complete flag example — continued

Time	t_3	t_4	t_5	t_6
OID	3	5	5	3
rev-ref-set	{5}	{3,4}	{3,4}	{1,5}
path-known	F	F	F	T
primary				1
path				1
version-nr	2	1	1	4
graph	{3,{5},2,T} {4,{5},1,F} {5,{3,4},1,F}	{3,{5},2,T} {4,{5},1,F} {5,{3,4},1,F}	{3,{5},2,T} {4,{5},1,F} {5,{3,4},1,T}	

Table 4.16. Complete flag example — continued

Time	t_6	t_7	t_8
OID	4	4	5
rev-ref-set	{5}	{5}	{3,4}
path-known	F	F	F
primary			
path			
version-nr	4	4	1
graph	{3,{5},2,**F**} {4,{5},4,F} {5,{3,4},1,**F**}	{3,{5},2,F} {4,{5},4,T} {5,{3,4},1,F}	{3,{5},2,**F**} {4,{5},4,**F**} {5,{3,4},1,**F**}

Table 4.17. Complete flag example — continued

root and object 3 has created a record for itself and retrieved records for object 4 and 5 from object 5. Without complete flags, object 3 would declare itself disconnected from the root set because its graph is complete. With the complete flags, object 3 simply sets the complete flag in its record. This is shown in Table 4.16 at time t_3.

At times t_4 and t_5, object 5 retrieves the current set of records from object 3 (including the record with the complete flag) and sets its own complete flag. After t_5, the configuration is changed back to the original configuration. At time t_6, object 3 records a new path to the root.

In Table 4.17 and also at time t_6, object 4 create a record for itself and fetches a set of records from object 5. Note that the complete flags are set to true in the records for objects 3 and 5 that are received from object 5. However, object 4 has to set all complete flags to false due to the version mismatch in object 4's own record. Without this rule, object 4 would have only have to set its own complete flag to true to set a complete set of records with all complete flags set to true. In other words, without this rule, object 4 would decide that it is unreachable.

Time	t_9	t_{10}	
OID	5	4	5
rev-ref-set	{3,4}	{5}	{3,4}
path-known	F	F	F
primary path			
version-nr	1	2	1
graph	{3,{5},2,F} {4,{5},4,F} {5,{3,4},1,T}	{3,{5},2,F} {4,{5},4,T} {5,{3,4},1,T}	{3,{5},2,F} {4,{5},4,T} {5,{3,4},1,T}

Table 4.18. Complete flag example — continued

At time t_7, object 4 sets its complete flag to true. Next, at time t_8, object 5 receives the set of records from object 4. Object 5 also sets all complete flags to false due to the version mismatch in the record for object 4.

At time t_9 in Table 4.18, object 5 sets its complete flag to true. Both object 4 and 5 now have the same versions of all records. At time t_{10}, object 5 and 4 exchange theirs sets of records. Note that the set of records in object 5 is similar to the set records it had at time t_5, except that the roles of objects 3 and 4 are reversed.

4.5.6 Safety

A garbage collection algorithm is safe if it does not delete objects that are still reachable from the root set. In this section, we give an informal argument why the algorithm is safe; in the next section we will look at safety more formally. In general, we can analyze this problem by looking at an object that is referenced by one other object. At some point in time a new reference to the object is added from a different object and the old reference is deleted. In a distributed system we have to look at a collection of objects (in different address spaces) with a single reference from the root set. We look at the scenario where, in a short period of time, a new reference from the root set to one of the objects in the collection is added and the old reference is deleted. We have to deal with two kinds of distributions: different objects run independently at different machines and a single object may have replicated state. We will first look at a collection of nonreplicated objects that run on different machines.

We can analyze this problem by looking at an object O, with two ancestors, A and B, and a root set with two objects R_1 and R_2. Figure 4.16 shows one possible configuration. Small circles represent objects, arrows represent object references and ellipses represent collections of objects. Initially there is an (indirect) reference from an object in the root set (object R_1) to object A. At some point in time, a new reference

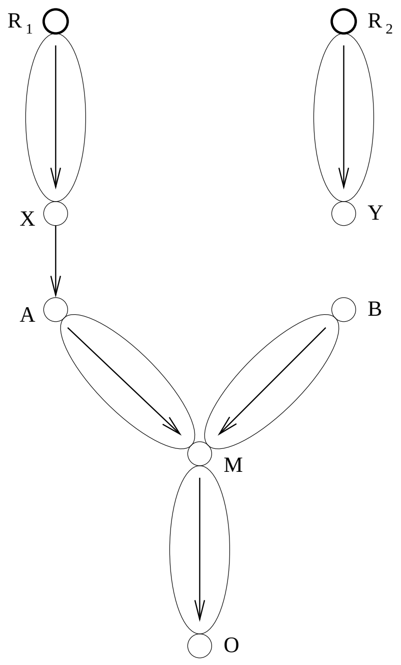


Figure 4.16. No relationship between A and B

is added to object B from Y and the reference from object X to object A is deleted. The whole process is observed from the point of view of object O.

We can classify different situations depending on the relationship between object A and object B. In Figure 4.16, A and B are independent. Object M is the first object that has both A and B as ancestors. In Figure 4.17, A is an ancestor of B. In Figure 4.18 the reverse is true, B is an ancestor of A. Finally, in Figure 4.19, A and B are each other's ancestors. They form a cycle.

Note that in the situations in Figure 4.16 and Figure 4.18, there are no references to object B and therefore, object B is deleted the next time it runs a garbage collection cycle. As such, adding a new reference to object B can be considered to be a programming error, and we can safely ignore those two scenarios.

Figure 4.17 is easy to analyze. Object A receives a path from X, and forwards this path to both B and O. A is deleted when it detects that the reference from object X has been deleted. B continues to advertise the path through object A until its next run of the garbage collector. It detects that the path through A is invalid, but instead it has a new path through Y and starts advertising this path. Object O eventually learns about the path through Y.

The analysis of Figure 4.19 requires the propagation of the complete flags. Object A announces a path through X as long as the reference from X to A is present.

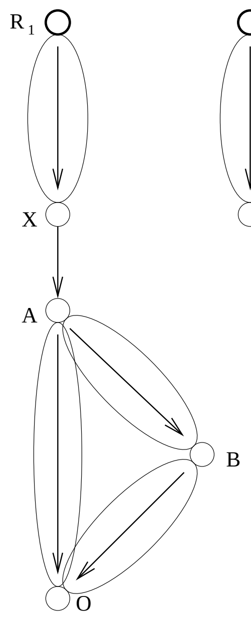


Figure 4.17. A is an ancestor of B

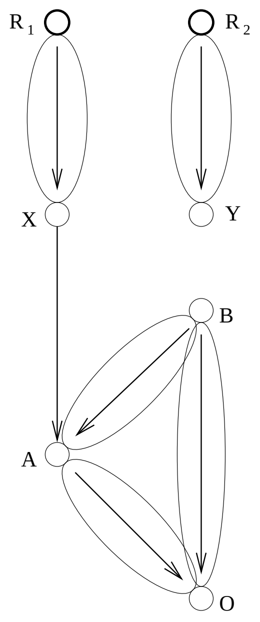


Figure 4.18. B is an ancestor of A

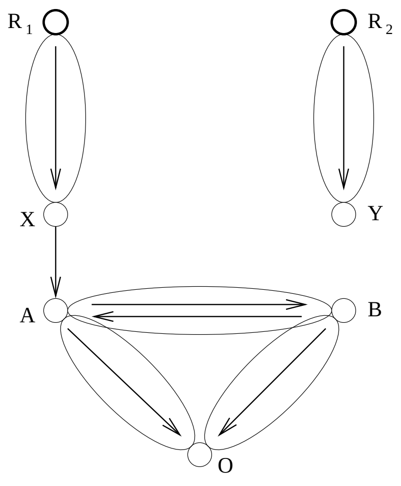


Figure 4.19. A and B form a cycle

However, it is possible that object B does not know its path through A, and as result B advertises a graph (at this point without the reference from object Y). This graph may reach object O. At some point in time, the reference from Y to B is added and the reference from X to A is deleted. A starts advertising a graph because it no longer knows a path to the root. The graph reaches object O, and object O concludes that neither A nor B nor any other object near O knows a path to a root object and as a result the collection of objects should be deleted. However, object O has to wait for records with the complete flag set to true from all objects. In the mean time, object B discovered the reference from object Y, and starts announcing a path. Eventually, this path will reach O, A and every other object in the collection.

4.5.7 Formal Approach to Safety

A garbage collector is safe if it does not delete objects that are reachable from the root set. In this garbage collector, an object is defined to be reachable from the root set if a path from a member of the root set through zero or more other objects to the object can be complemented with a path from the object to the member of the root set using reverse references. This means that an object can be unreachable from the root, even though there are still references to the object.

This definition of reachable from the root set leads to another problem: it is possible that an object is temporarily unreachable from the root set. It is possible to cause an object to become unreachable by deleting all references to the object's parents, and to create a new reference (with the appropriate reverse reference) just before the object start a garbage collection cycle.[8] We consider this to be an error on the part of the user of the object. For the safety analysis, objects are required to be reachable from the root set throughout their lifetime. In fact, we go one step further: to determine whether, at a certain time, an object is reachable from the root set or not, we ignore any ancestor of the object that has not been continuously reachable from the root set.

With respect to the root set, we assume that an object cannot become a member of the root set after it has participated in the graph algorithm. In particular, an object that has sent graph records to other objects cannot become a member of the root set. This is a limitation of the garbage collection algorithm; it is not clear whether this limitation should be addressed or not.

The garbage collection algorithm considers an object to be unreachable from the root set if one of two conditions is met: either the object's set of reverse references has become empty or the graph algorithm has completed. We will start with the first condition, which is easy to analyze. The second condition is much harder to analyze.

The set of reverse references is updated by both the user (owner) of an object and by the garbage collector. The user of the object is responsible for adding new reverse references (by invoking the add-rev-ref method), and the garbage collector deletes obsolete entries.

[8] In a traditional single address space garbage collector the same effect can be achieved by writing a pointer to an object to disk and reading that pointer back into memory at a later time.

The garbage collector deletes a reverse reference when either the reverse reference refers to an object that has been destroyed or when it refers to an object that no longer stores a corresponding forward reference. Below, we will see how the location service can be augmented to reliably report whether an object exists or not. The existence of an appropriate forward reference is checked by invoking the method check-forw-ref on the object that the reverse reference refers to. When the set of reverse reference becomes empty, after all parents are either destroyed or have deleted their forward references, the object is unreachable from the root set and can be deleted.

To prove the safety of the graph algorithm we view the algorithm as a distributed snapshot algorithm [Babaoğlu and Marzullo, 1993]. Each object tries to get a consistent snapshot of the sets of reverse references of its ancestors. We prove that when an object is unreachable according to its snapshot, the object is in fact unreachable. To prove this, we first construct a model of the system and prove that the existence of a snapshot in which an object is unreachable combined with the assumption that the object is in fact reachable leads to a contradiction.

Model

The set of all object identifiers is called OID. The variables a, b, o, and p refer to elements of this set. Every object maintains a set of reverse references (also called parents) and assigns a local version number to different versions of this set. Version numbers are taken from the set of natural numbers $\mathbb{N}_0$. For each object, its version number is set to 0 when an object is created and is incremented each time the set of reverse references changes. The variables i, j, n, k, and l are used to denote version numbers.

A combination of an object identifier and a version number uniquely identifies a set of reverse references. We introduce we function $R : \text{OID} \times \mathbb{N}_0 \rightarrow 2^{\text{OID}}$ which returns a particular version of the set of object identifiers of the reverse references of an object.

Each object knows the set of reverse references of some of its ancestors. This knowledge is called a view. Each object o has a sequence of views $V_{o,i}$, where i is the version number of the view. The views of an object are numbered independently of the set of reverse references of the object. A view is represented by a function $V_{o,i} : \text{OID} \rightarrow \mathbb{N}_0$. This function returns for an ancestor object a of o the version number of the set of reverse references of a in the i-th view of object o.

We define the operator Dom as follows:

$$\text{Dom}(f) = \{x \in \text{domain}(f) \mid f(x) \text{ is defined}\} \tag{4.1}$$

For example, $\text{Dom}(V_{o,i})$ is the set of object identifiers in view number i of object o.

We define the subset relationship between two views as follows:

$$V_1 \subseteq V_2 :: \text{Dom}(V_1) \subseteq \text{Dom}(V_2) \wedge \forall o \in \text{Dom}(V_1) : V_1(o) = V_2(o) \tag{4.2}$$

Ancestor	**Version**	**Reverse references**	**Complete flag**
OID_1	V_1	$\text{OID}(\text{rr}_1), \ldots, \text{OID}(\text{rr}_m)$	C_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
OID_n	V_n	$\vdots$	C_n

Table 4.19. Collection of reverse-reference records

Object	**Version**	**Reverse references**
OID_1	1	$\text{OID}(\text{rr}_1), \ldots, \text{OID}(\text{rr}_m)$
OID_1	2	$\vdots$
$\vdots$	$\vdots$	$\vdots$
OID_n	1	$\vdots$
$\vdots$	$\vdots$	$\vdots$
OID_n	k	$\vdots$

Table 4.20. Reverse-reference function $R(o, i)$

In other words, V_1 is a subset of V_2 if we can derive V_1 from V_2 by deleting zero or more values from V_2.

In addition to a view, each object also maintains a collection of complete flags. The collection of complete flags for an object o is represented by a total function $C_{o,i} : \text{OID} \rightarrow \{\text{T}, \text{F}\}$. A total function is defined for its entire domain. This means that $C_{o,i}(a)$ is defined for every $a \in \text{OID}$. A function $C_{o,i}$ contains the complete flags that are associated with view $V_{o,i}$. Initially, an object starts with all complete flags set to false (i.e, $\forall a \in \text{OID} : C_{o,0}(a) = \text{F}$). We define the logical-or operator on two functions C_1 and C_2 as follows:

$$(C_1 \vee C_2)(o) = C_1(o) \vee C_2(o) \tag{4.3}$$

Tables 4.19, 4.20, 4.21(a), and 4.21(b) illustrate the difference between the data structures used by the actual algorithm and the functions that are used to model the algorithm. Table 4.19 shows a reverse-reference graph for one object at one point in time. Different objects are likely to have different graphs and their graphs change over time. The reverse-reference function R, shown in Table 4.20, models the relation between a set of reverse references and the combination of an object identifier and a set of reverse references. The view in Table 4.21(a) can be combined with function R to obtain the first three fields (object identifier, version number, and set of reverse-references) of a particular version of an objects reverse-references graph.

Object	**Version**
OID_1	V_1
$\vdots$	$\vdots$
OID_n	V_n

(a) $V_{o,i}$

Object	**Complete flag**
OID_1	T or F
$\vdots$	$\vdots$
OID_n	T or F
$\vdots$	$\vdots$

(b) $C_{o,i}$

Table 4.21. View and corresponding complete flags

The corresponding complete flags function (Table 4.21(b)) provides the complete-flag field.

We call the object that has just decided that it is unreachable Ω. The version number of Ω's most recent view is M. We will use $\bar{V}$ to refer to Ω's most recent view ($V_{\Omega,M}$) and $\bar{C}$ to refer to the corresponding collection of complete flags ($C_{\Omega,M}$).

An object o with view $V_{o,i}$ is unreachable according to the graph algorithm when the complete flags in all records are set to true (step 11 of the reachable method). This is shown in the following equation:

$$\forall o \in \text{Dom}(V_{o,i}) : C_{o,i}(o) = \text{T} \tag{4.4}$$

For an arbitrary view V, we can define a function $P^*_V : \text{OID} \rightarrow 2^{\text{OID}}$ which returns the ancestors of an object $o \in \text{Dom}(V)$ according to view V as follows:

$$P^0_V(o) = \{o\} \tag{4.5}$$

$$P^i_V(o) = \{p_1 \in \text{OID} \mid \exists p_2 \in P^{i-1}_V(o) : p_1 \in R(p_2, V(p_2))\} \qquad (i > 0) \tag{4.6}$$

$$P^*_V(o) = \bigcup_{i \in \mathbb{N}_0} P^i_V(o) \tag{4.7}$$

This model assumes that version numbers associated with sets of reverse references form a strictly monotonic sequence starting with zero. This can be verified by looking at the algorithm. The version number is incremented each time the set of reverse references is changed. This is done in the add-rev-ref method (see page 168) and in step 6 of the reachable method (see page 170).

In this model, an object identifier and a version number uniquely identify a set of reverse references. In the actual algorithm, each record contains, apart from the object identifier and the complete flag, both a version number and a set of reverse references (see Section 4.5.3, pages 164–165). This means that in theory records may exist (in different objects) that have the same object identifier and version number but different sets of reverse references. However, only the complete flag in a record can be changed during the execution of the algorithm; the object identifier, the version number, and the corresponding set of reverse references are always copied together. For this reason, we can assume that the combination of an object identifier and a version number is associated with at most one set of reverse references.

An object sets the complete flag in its record to true when it has records for all of its ancestors. This is expressed in the following equation:

$$C_{o,i}(o) = \mathrm{T} \iff \mathrm{Dom}(V_{o,i}) = \{o\} \cup \bigcup_{a \in \mathrm{Dom}(V_{o,i})} R(a, V_{o,i}(a)) \tag{4.8}$$

To model the propagation of complete flags, we introduce the total function Rcvd : $\mathrm{OID} \times \mathbb{N}_0 \times \mathrm{OID} \times \mathbb{N}_0 \rightarrow \{\mathrm{T}, \mathrm{F}\}$. $\mathrm{Rcvd}(a, i, b, j) = \mathrm{T}$ if, and only if, the following two conditions are met: (1) object a received a set of records from object b that correspond to view $V_{b,j}$ and the collection of complete flags $C_{b,j}$; (2) the view that results from processing the received records is $V_{a,i}$ (with complete flags $C_{a,i}$). Note that $V_{a,i}$ is not necessarily a new view. It is possible $V_{b,j}$ does not contain any new information and, therefore, object a does not change its view.

An important property of the algorithm is that when an object a receives a set of records from a parent object b, all complete flags are cleared, unless the view that is passed from the parent is subset of the view of the child (step 8). Note that the test in step 8 whether or not to clear the complete flags is the same as is used in the definition of the subset relation in equation 4.2. If any complete flags other then object's own complete flag are true after receiving a view from a parent, then the parent's view must have been a subset of the child view. This is expressed in the following implication:

$$\begin{aligned} &\forall a, b \in \mathrm{OID}, i, j \in \mathbb{N}_0, o \in \mathrm{OID} \neq a : \\ &\qquad \mathrm{Rcvd}(a, i, b, j) = \mathrm{T} \wedge C_{a,i}(o) = \mathrm{T} \Longrightarrow V_{b,j} \subseteq V_{a,i} \end{aligned} \tag{4.9}$$

When the set of records received by object a from an object b is a subset of object a's current set of records, object a sets its complete flags to true using the equivalent of an or operator: $C_{a,i} = C_{a,i-1} \vee C_{b,j}$ (step 9).

The algorithm clears the complete flags in all records in an object's graph whenever that object modifies its set of reverse references (the add-rev-ref method and steps 4 and 6 of the reachable method) or when it receives a set of records that is not a subset of the current set of records (step 8). This is expressed in the following implication:

$$\begin{aligned} &\forall a \in \mathrm{OID}, i \in \mathbb{N}_0, o \in \mathrm{OID} \neq a : \\ &\qquad C_{a,i}(o) = \mathrm{T} \wedge C_{a,i+1}(o) = \mathrm{T} \Longrightarrow V_{a,i} = V_{a,i+1} \end{aligned} \tag{4.10}$$

An object directly sets only its own complete flag; all other complete flags are received from parents. This means that if an object a has the complete flag for object b ($\neq a$) set to true in version i of its collection of complete flags (i.e. $C_{a,i}(b) = \mathrm{T}$), there has to be a cycle-free path that specifies the propagation of the complete flags from object b to object a. This path can be described by a sequence of tuples $(o_1, k_1), (o_2, k_2), \dots, (o_{n-1}, k_{n-1}), (o_n, k_n)$ with $o_1 = b$, $o_n = a$, $k_n = i$, and $o_j \neq b$ $(2 \leq j \leq n)$.

In this path, either an object receives the complete flag from its parent (i.e., $o_{j+1} \neq o_j$) or its view is changed but the complete flags remains set to true ($o_{j+1} = o_j$ and $k_{j+1} = k_j + 1$). This is expressed in the following equation:

$$\begin{aligned} \forall 1 \leq j < n : (&o_{j+1} \neq o_j \wedge \\ &\text{Rcvd}(o_{j+1}, k_{j+1}, o_j, k_j) = \text{T} \wedge C_{o_{j+1},k_{j+1}}(b) = \text{T}) \\ &\vee \\ &(o_{j+1} = o_j \wedge k_{j+1} = k_j + 1 \wedge \\ &C_{o_j,k_j}(b) = \text{T} \wedge C_{o_{j+1},k_{j+1}}(b) = \text{T}) \end{aligned} \tag{4.11}$$

Finally, object b has its own complete flag set to true (i.e, $C_{o_1,k_1}(o_1) = C_{b,k_1}(b) = \text{T}$).

Proof

For the proof of the safety of the garbage collector we have to assume that the views of all ancestors of Ω are contained in $\bar{V}$. We will prove two lemmas that support this assumption. The first lemma generalizes implication 4.9 for all ancestors of Ω. The second lemma shows that the ancestors of an ancestor of Ω can be computed using $\bar{V}$.

Lemma 1

$$\forall a \in P^*_{\bar{V}}(\Omega) : \bar{C}(a) = \text{T} \Longrightarrow \exists i \in \mathbb{N}_0 : V_{a,i} \subseteq \bar{V} \wedge C_{a,i}(a) = \text{T} \tag{4.12}$$

Proof The proof is trivial for $a = \Omega$: $i = M$ gives the desired result. For $a \neq \Omega$, there exists a path $(o_1, k_1), (o_2, k_2), \dots, (o_{n-1}, k_{n-1}), (o_n, k_n)$ with $o_1 = a$, $o_n = \Omega$, $k_n = M$, and $i = k_1$. A property of the path is $C_{o_1,k_1}(o_1) = C_{a,i}(a) = \text{T}$.

Using implication 4.9, implication 4.10, and equation 4.11 we can show that $V_{o_j,k_j} \subseteq V_{o_{j+1},k_{j+1}}$ $(1 \leq j < n)$, as follows. According to equation 4.11, either

$$o_{j+1} \neq o_j \wedge \text{Rcvd}(o_{j+1}, k_{j+1}, o_j, k_j) = \text{T} \wedge C_{o_{j+1},k_{j+1}}(b) = \text{T} \tag{4.13}$$

or

$$o_{j+1} = o_j \wedge k_{j+1} = k_j + 1 \wedge C_{o_j,k_j}(b) = \text{T} \wedge C_{o_{j+1},k_{j+1}}(b) = \text{T} \tag{4.14}$$

From implication 4.9 with $a = o_{j+1}$ $b = o_j$, $i = k_{j+1}$, and $j = k_j$ we obtain

$$\text{Rcvd}(o_{j+1}, k_{j+1}, o_j, k_j) = \text{T} \wedge C_{o_{j+1},k_{j+1}}(b) = \text{T} \Longrightarrow V_{o_j,k_j} \subseteq V_{o_{j+1},k_{j+1}} \tag{4.15}$$

Combining implication 4.15 with equation 4.13 leads to

$$o_{j+1} \neq o_j \Longrightarrow V_{o_j,k_j} \subseteq V_{o_{j+1},k_{j+1}} \tag{4.16}$$

Similarly, from implication 4.10 with $a = o_j$, $i = k_j$, and $o = b$ we obtain

$$C_{o_j,k_j}(b) = \mathrm{T} \wedge C_{o_j,k_j+1}(b) = \mathrm{T} \Longrightarrow V_{o_j,k_j} = V_{o_j,k_j+1} \tag{4.17}$$

Combining implication 4.17 with equation 4.14 and with $o_{j+1} = o_j$ and $k_{j+1} = k_j+1$ from equation 4.14 leads to

$$o_{j+1} = o_j \Longrightarrow V_{o_j,k_j} = V_{o_{j+1},k_{j+1}}$$

and therefore to

$$o_{j+1} = o_j \Longrightarrow V_{o_j,k_j} \subseteq V_{o_{j+1},k_{j+1}} \tag{4.18}$$

Combining equating 4.16 and equation 4.18 results in

$$V_{o_j,k_j} \subseteq V_{o_{j+1},k_{j+1}}$$

Using the transitivity of $\subseteq$, it follows that $V_{o_1,k_1} \subseteq V_{o_n,k_n}$ and, therefore, $V_{a,i} \subseteq V_{\Omega,M} = \bar{V}$. □

Lemma 2

$$\forall a \in \text{Dom}(\bar{V}) : \exists i \in \mathbb{N}_0 : P^*_{\bar{V}}(a) \subseteq \text{Dom}(V_{a,i}) \wedge V_{a,i} \subseteq \bar{V} \tag{4.19}$$

Proof From equation 4.4 we know that $\bar{C}(a) = \mathrm{T}$ (because $a \in \text{Dom}(\bar{V})$). Combined with lemma 1 this ensures the existence of $V_{a,i} \subseteq \bar{V}$ and $C_{a,i}(a) = \mathrm{T}$. Using equation 4.8, we find that $\text{Dom}(V_{a,i})$ contains the ancestors of every object in $\text{Dom}(V_{a,i})$. Furthermore, $V_{a,i} \subseteq \bar{V}$ means $\forall o \in \text{Dom}(V_{a,i}) : V_{a,i}(o) = \bar{V}(o)$ (definition 4.2). This means that the same sets of reverse references are used to compute $P^*_{V_{a,i}}(a)$ as are used to compute $P^*_{\bar{V}}(a)$. The set $P^*_{\bar{V}}(a) = P^*_{V_{a,i}}(a)$ contains the ancestors of just one object (a) in $\text{Dom}(V_{a,i})$ and, therefore, $P^*_{\bar{V}}(a) \subseteq \text{Dom}(V_{a,i})$. □

We show that the existence of view $\bar{V}$ implies that the object is unreachable by looking at changes to the sets of reverse references of various objects. Each change to a set of reverse references (either an invocation of the add-ref-ref method to add a new reverse reference or step 5 of the reachable method) is called an event. Each time the set of reverse references of an object is changed, the object increments its version number. Events are therefore identified by a tuple that consists of an object identifier and the version number of the object's set of reverse references that reflects the event. The set of all events is $\{(o, n) \mid (o, n) \in \text{Dom}(R)\}$.

We define $\bar{E} \subseteq \text{OID} \times \mathbb{N}_0$ as the set of all events that ever occurred to objects in $\text{Dom}(\bar{V})$:

$$\bar{E} = \{(o, n) \mid o \in \text{Dom}(\bar{V}) \wedge (o, n) \in \text{Dom}(R)\} \tag{4.20}$$

$E \subseteq \bar{E}$ is defined as the set of all events $(o, n) \in \bar{E}$ for which the object o was reachable when the event occured. We can split E into three subsets depending on whether an event occured logically before view $\bar{V}$ (E^-), after $\bar{V}$ (E^+), or is part of $\bar{V}$ (E^0):

$$E^- = \{(o, n) \in E \mid n < \bar{V}(o)\} \tag{4.21}$$

$$E^+ = \{(o, n) \in E \mid n > \bar{V}(o)\} \tag{4.22}$$

$$E^0 = \{(o, n) \in E \mid n = \bar{V}(o)\} \tag{4.23}$$

The assumption that the algorithm is incorrect and that object Ω is reachable from the root implies that at least one of its ancestors has to be an object not in $\text{Dom}(\bar{V})$. The reason is that an object in the root set does not participate in the graph algorithm and, therefore, cannot be a member of $\text{Dom}(\bar{V})$. Either Ω has a reverse reference to an object not in $\text{Dom}(\bar{V})$ or one of Ω's ancestors in $\text{Dom}(\bar{V})$ has a reverse reference to an object not in $\text{Dom}(\bar{V})$ (or both). Since $\Omega \in \text{Dom}(\bar{V})$, we can say that, under the assumption that Ω is reachable from the root, at least one object in $\text{Dom}(\bar{V})$ must have a reverse reference to an object not in $\text{Dom}(\bar{V})$. In other words, the existence of a path from the root to object Ω implies at least one event $(o, n) \in E$ with $R(o, n) \not\subseteq \text{Dom}(\bar{V})$.

We can define subsets A, A^-, A^+, and A^0 of E that contain those events:

$$A = \{(o, n) \in E \mid R(o, n) \not\subseteq \text{Dom}(\bar{V})\} \tag{4.24}$$

$$A^- = A \cap E^- \tag{4.25}$$

$$A^+ = A \cap E^+ \tag{4.26}$$

$$A^0 = A \cap E^0 \tag{4.27}$$

Note that $A^0 = \emptyset$ due to equation 4.8 and the fact that $\bar{C}(\Omega) = \text{T}$.

To show that the algorithm is incorrect, we have to assume that $E^+ \neq \emptyset$. However, we show that any $E^+ \neq \emptyset$ leads to a contradiction. This implies that E^+ is empty and that the object is unreachable.

Since events in E are related to physical events in a distributed computer system, we can assume that all events can be ordered according to some global time. Note that this global time is a hypothetical clock and is not actually maintained by the distributed system.

If $E^+ \neq \emptyset$ then there must be an $(a, n) \in E^+$ which happens before all other events in E^+. $(a, n) \in E^+$ implies that event (a, n) happened after event $(a, \bar{V}(a)) \in E^0$. According to the definition of reachability, object a is reachable at the moment that event (a, n) occurs only if it has been continuously reachable before that time. Object a is reachable only if there is at least one path to the root. This implies that there is at least one object b and an event (b, k) such that $b \in \text{Dom}(\bar{V})$, $R(b, k) \nsubseteq \text{Dom}(\bar{V})$, event (b, k) occurs (in real time) before event (a, n), and the next event $(b, k + 1)$ occurs after (a, n). In other words, (b, k) is the last event of b that occurs before (a, n) and therefore $R(b, k)$ is the most recent set of reverse references of b when (a, n) occurs. The fact that $b \in \text{Dom}(\bar{V}) \wedge R(b, k) \nsubseteq \text{Dom}(\bar{V})$ implies that $(b, k) \in A$. We know that $(b, k) \notin A^+$ because $A^+ \subseteq E^+$ and (b, k) has to precede (a, n) which is the earliest event in E^+. Therefore, $(b, k) \in A^-$.

We can distinguish two cases depending on whether $b \in P^*_{\bar{V}}(a)$ or not. In both cases we will show that no suitable event (b, k) exists.

We will first look at the case $b \in P^*_{\bar{V}}(a)$. We know that the event $(b, \bar{V}(b)) \in E^0$ happens after (b, k) because $(b, k) \in A^- \subseteq E^-$. From lemma 2 we know that (for some i)

$$P^*_{\bar{V}}(a) \subseteq \text{Dom}(V_{a,i}) \wedge V_{a,i} \subseteq \bar{V}$$

With both a and $b \in P^*_{\bar{V}}(a)$ we know that view $V_{a,i}$ contains both $(a, \bar{V}(a))$ and $(b, \bar{V}(b))$ (all events in view $\bar{V}$ have the form $(o, \bar{V}(o))$). The existence of a view $V_{a,i}$ that contains both $(a, \bar{V}(a))$ and $(b, \bar{V}(b))$ implies that $(b, \bar{V}(b))$ happened (in real time) before event (a, n) (because $n > \bar{V}(a)$). This contradicts the assumption that event (b, k) was the last event of object b before event (a, n) because $(b, \bar{V}(b))$ intervenes. In other words, it is impossible to select an object $b \in P^*_{\bar{V}}(a)$ and an event (b, k) such that $R(b, k)$ is both the current set of reverse references of b at the time event (a, n) occurs and $R(b, k) \nsubseteq \text{Dom}(\bar{V})$.

If object b is an ancestor of a but $b \notin P^*_{\bar{V}}(a)$ then there must exist a $(p, l) \in E$ such that $p \in P^*_{\bar{V}}(a)$ and $R(p, l) \nsubseteq P^*_{\bar{V}}(a)$ (i.e, p is an object in $P^*_{\bar{V}}(a)$ with at least one parent not in $P^*_{\bar{V}}(a)$).[9] $(p, l) \notin E^+$ because (p, l) happens before (a, n) and (a, n) is the earliest element in E^+. We argue that no suitable (p, l) exists in a way similar to the reasoning that no suitable (b, k) for $b \in P^*_{\bar{V}}(a)$ exits. We require (p, l) to be the last event of object p that precedes event (a, n) in real time. $(p, l) \notin E^0$ because $R(p, \bar{V}(p)) \subseteq P^*_{\bar{V}}(a)$. However, if $(p, l) \in E^-$ then (p, l) happens before $(p, \bar{V}(p))$. From lemma 2 and $p \in P^*_{\bar{V}}(a)$ we know that some view of a contains both $(a, \bar{V}(a))$ and $(p, \bar{V}(p))$. This means that $(p, \bar{V}(p))$ happened before (a, n). This means that event (p, l) was not the last event of object p before event (a, n) and, therefore, no suitable event (p, l) exists.

We can conclude that no suitable object b with associated event (b, k) exists to provide a with a path to the root at the time event (a, n) occurs. This means that $(a, n) \notin E^+$ and therefore $E^+ = \emptyset$. This completes the proof of the safety of the garbage-collection algorithm.

[9]This argument is similar to the one used to argue the existence of (b, k). Object p is a member of a smaller set (in this case $P^*_{\bar{V}}(a)$) that has at least one ancestor in a proper super set.

4.5.8 Completeness

A garbage collection algorithm is complete if all unreachable objects will be reclaimed eventually. We can analyze completeness by looking at a collection of objects that is unreachable from the root set.

In the absence of cycles, which means that the references between the objects in the collection form a directed acyclic graph, there has to be at least one object that has no parent objects. This object will be deleted, which results in a smaller directed acyclic graph. After a number of iterations, the complete collection of objects is destroyed. The speed at which single object is garbage collected depends on how often its ancestors perform a garbage collection cycle, and on transient failures that prevent the object from communication with its parents.

A collection of objects that does contain a cycle requires that all objects in the collection that are part of the cycle build a complete view of the graph that connects them and their ancestors. Transient failures may delay the creation of that view. However, a more serious problem is that the graph has to be stable to be propagated, that is, insertion and deletion of object references causes the garbage collection algorithm to restart. To solve this problem, an object should stop accepting new reverse references (delay calls to add-rev-ref) when it has been unable to find a path to the root for some period of time. Further research is needed to determine how long this period should be.

4.5.9 Replication

The description and analysis of the garbage collection algorithm ignored distributed shared objects with replication state. To handle replication, we have to look at three different issues. The first issue is the consistency requirements of the garbage collection algorithm. The second issue is running multiple copies of the algorithm at the same time in different replicas of a single distributed shared object. The third issue is dealing with replicas with different sets of reverse references.

The consistency requirements can be described in two different ways. One approach is to look at a suitable memory coherence model. Alternatively, it is possible to look at models that take the semantics of the garbage collector into account. The consistency requirements are best illustrated with an example. Consider a distributed shared object with one parent. A user adds a reference from a second parent, invokes the add-rev-ref method on the distributed shared object, and deletes the reference from the first parent. The next time the object runs the garbage collection algorithm, it discovers that the reference from the original parent has been deleted and deletes the parent from the set of reverse references. It is important that the object's set of reverse references contains the second parent otherwise the object will be deleted. The garbage collection algorithm invokes the check-forw-ref method on the second parent. It is important the second parent reports that it has a reference to the object.

This example shows that the method add-rev-ref should be implemented such that the next time the garbage collection algorithm runs, it should know about the additional parent. Similarly, a reference to an object that has been added to one replica should cause the check-forw-ref method to return that the distributed shared object as a whole contains a reference to the object.

Memory coherence models that are linearizable provide the desired consistency. A concurrent computation is called **linearizable** if it is “equivalent” to a legal sequential computation [Herlihy and Wing, 1990]. This means that all lookup operations that are started after an insert operation has completed, have to take the inserted reference into account. A straightforward way of implementing a linearizable coherence model is to update all copies of the state during a write operation or to invalidate all other copies. More advanced algorithms are described in [Herlihy and Wing, 1990].

An alternative approach is to use the so called **union rule**: for the purpose of garbage collection, an object should use (report) the union of all object references in its replicas. This means that the garbage collection algorithm should assume the union of all copies of the set of reverse references to determine whether the object is reachable or not. Similarly, the method check-forw-ref has to scan all forward references in all replicas before returning false.

In distributed shared objects with multiple replicas and especially if those replicas belong to different resource owners, it is desirable to be able to execute the garbage collection algorithm concurrently in different replicas. The path algorithm is easy to parallelize. Different replicas can execute the path algorithm independently, even if they have different sets of reverse references. With different sets of reverse references it is possible that one replica’s set is empty whereas other replicas have nonempty sets. This is not a problem. The replica with the empty set can be deleted and the other replicas can continue to exist.

Unfortunately, it is much harder to parallelize the graph algorithm. The graph algorithm depends on the fact the a single distributed shared object has a single version number, a single set of reverse references, and a single complete flag. More research is needed to determine whether parallel versions of the graph algorithm are possible. It is however possible to execute the complete garbage collection algorithm in one replica and execute the path algorithm in other replicas. The replica that executes the complete algorithm has to be careful with its conclusions if it detects that it is unreachable. If the graph algorithm has determined that it is unreachable, it should inform all other replicas that the entire object is unreachable. On the other hand, if its set of reverse references is empty, if should tell the other replicas to select a new replica that executes the complete algorithm.

Replicas that belong to different resource owners may have different sets of reverse references. A parallel version of the path algorithm can easily handle different sets of reverse references. Unfortunately, it is necessary to choose which sets of reverse references are used in the graph algorithm. One approach is to use all reverse references (the union rule). Another approach is to use only the reverse references of the replica that executes the graph algorithm.

For example, distributed shared objects that consist of a core with one replica and a ring of cache replicas around the core ignore the reverse references of the cache replicas. The reverse references that are stored in cache replicas can be used to assist the resource owners of those replicas but are not to be used to determine the lifetime of the distributed shared object as a whole. Another example is a distributed shared object that provides a collection of users with a shared whiteboard. In this case, the distributed shared object should continue to exists as long as at least one of the users is interested in the object. This type of object is expected to use all reverse references to determine whether it is reachable or not.

4.5.10 Exceptional Conditions

Occasionally, a method invocation on a parent object may fail. These failures fall into two categories: transient failures and permanent failures. A **transient failure** is a temporary failure. For example, a method invocation cannot be executed due to a communication failure, a power failure, a crashed server, etc. In general, we can expect that at a later time the operation will succeed.

On the other hand, if the method fails permanently, we know that retrying the method is pointless. A **permanent failure** is an authoritative statement that an operation cannot succeed. The garbage collection algorithm has to consider only one situation that causes an operation to fail permanently: an attempt to invoke an operation on an object that has been destroyed. A child object that tries to invoke a method on a parent object can never succeed if the parent object has been destroyed.

Unfortunately, it is quite hard to determine whether a method invocation failed because the object has been destroyed or whether there is a communication failure or a transient failure in the location service that makes it impossible to bind to the object. One way to resolve the issue is to try to communicate with the object for some time. The object can be declared dead if communication fails long enough. This is not an acceptable solution: if the time-out is set too large, it will take a long time before garbage is removed. A short time-out means that a network problem that lasts a couple of days may lead to the loss of a large number of objects.

The alternative is to introduce a service, indexed on the object handle, which states whether an object exists or not. A suitable candidate for providing this service is the location service: this service is already indexed on the object handle, and internal garbage collection of the location service requires that entries for dead objects are deleted.

The location service needs to be extended in the following way:

1. The location service should report an object as destroyed if the location service does not contain an entry for the object.

2. The location service deletes entries for objects without any contact addresses.

3. For each contact address, the location service needs to be able to verify whether the address is still valid or not. Invalid contact addresses are deleted.

We can define a service that reports whether a contact address is valid using a bottom-up approach. Contact addresses of persistent objects are already managed by an object manager. The location service can use the object manager to verify whether a particular contact address is still valid or not. This can be implemented in the location service by storing the object handle of the object manager for the contact address along with the contact address itself. For nonpersistent objects, it is possible to create a distributed shared object in each address space that provides a similar service.

The next problem is that the object manager itself may fail. The solution is to use a higher-level manager that can verify whether a particular object manager still exists. The object manager is a distributed object with contact addresses which means that the object handle of the higher-level manager can be associated with the contact addresses of the object manager.

This approach creates a complete hierarchy of managers that verify the existence of other managers or, at the lowest level, contact addresses. The root of that hierarchy has to be handled manually by system administrators.

4.5.11 Scalability

The three main scalability problems for this garbage collection algorithm are: (1) the number of references to an object, (2) the loss of efficiency due to garbage collection cycles, and (3) the resources (memory, bandwidth) needed to run the disconnected graph algorithm.

The first problem has been described already: an object can store only a small number of reverse references. This is a problem in pure reference listing systems. However, this is not a problem in Globe for two reasons: only the owner (sponsor) of an object is allowed to store reverse references in the object and, for highly replicated objects, the reverse references are spread out over the replicas. Future work is needed to verify that garbage collection of highly replicated objects is feasible.

The second problem is the cost of garbage collection cycles in an otherwise idle system. The question is how often an object should run a garbage collection cycle. For example, suppose that a server allocates on average 1% of its resources per hour. This means that all resources are allocated after 100 hours. The server needs to free 1% of its resources each hour to reach equilibrium.

A simple-minded server simply starts a garbage collection cycle for all of its objects at regular intervals. A better solution is to start a new garbage cycle when a certain amount of resources have been allocated since the last cycle. Another technique is the use of generational garbage collection. With **generational garbage collection**, objects that are relatively old are checked less frequently than younger objects [Lieberman and Hewitt, 1983]. In most systems, many objects live only a short time and objects that already exist for a longer period are likely to stay alive for a while. A typical exception is a news spool directory, where most news articles are deleted when they reach a certain age.

An improvement that avoids superfluous garbage collection cycles is the check-rev-ref method. This method triggers a garbage collection cycle when an object reference is deleted instead of at random moments. It is possible that enough objects are reclaimed due to check-rev-ref method invocations so that the server does not have to start a garbage collection cycle.

The techniques that avoid garbage collection until the server runs out of storage are not suitable for objects that contain references to other objects. Those objects should be destroyed promptly, otherwise they may prevent other objects from being reclaimed. Fortunately, objects can give a hint to their parent objects through the timeout parameter of the reachable method, that is, an object can invoke reachable on its (primary) parent to trigger a garbage collection cycle.

The last scalability problem is the size of the reverse-reference graphs in a (disconnected) collection of objects. Graphs are computed in three different situations: (1) the object is in fact reachable but the path to the root did not yet reach the object, (2) the object is unreachable and part of a cycle, and (3) the object is unreachable but not part of cycle. In the second case, computation of the graph is necessary to decide that the object is unreachable. In the two other cases, computing the graph is a waste of time and resources.

In the case of a cycle, the objects probably belong to a single user or organization. The reason they probably belong to a single user is that each of the objects in the cycle can keep the other objects reachable from the root (and thus prevent them from being collected). For this reason, collecting cycles is likely to be a local issue, where the users can limit size of the cycles to the garbage collection resources available.

In the other two cases the computation of the graph is not critical. Either the object learns a new path to the root, or when the object is unreachable but not part of a cycle, the object can simply wait for its ancestors will be destroyed. For this reason, the garbage collection can start out by limiting the size of the graph, and compute only the full graph if no progress has been made for some time.

4.5.12 Name Service

At the end of this section about garbage collection we look at the interaction between the garbage collector and the naming system. The garbage collection algorithm requires that references to distributed shared objects be stored in other distributed shared objects (with the exception of references to objects in the root set). With relatively little effort we can turn those objects into directory objects for the naming system. The object just has to associate an ASCII name (directory entry) with each object reference. The result is that if a particular object is reachable from an object in the root set, then that particular object is also reachable from the root object via a pathname.

The requirement that an object associates directory entries with its object references is a modest requirement. Distributed shared objects that store references to other objects can easily implement the required methods to be considered a directory object. The advantage of directory objects is that the users of the system have easy access to object references stored in different address spaces.

A pathname for an object can be created by concatenating a pathname for the object's primary parent with a directory entry that refers to the object. A pathname for a nonroot parent object can be created by applying the same procedure recursively. For a root object it is necessary to configure its pathname explicitly. An object that does not have a primary parent, or that has an ancestor without a primary parent cannot compute a pathname. Note that a nonroot object does not have to store its pathname persistently. Upon request, the object can ask its parent for the necessary information (the parent's pathname and a directory entry). If the parent also does not know its pathname, it has to ask its parent, and so on. Absence of cycles in every ancestor's path to the root guarantees that this recursive procedure terminates.

The availability of pathnames provides an opportunity for an interesting optimization. Instead of caching a list of object identifiers (the path) to verify that an object is still reachable through the same path to the root, it is also possible to use a pathname for the same purpose. An object is reachable through the same path as long as its parent has the same name. Storing a pathname instead of a list of object identifiers is more efficient for two reasons. The first reason is that the size of a pathname is usually shorter than the size of a list of object identifiers, which use 32 bytes per entry. The second reason is that storing a pathname persistently allows an object to return a pathname immediately without going to its parent. A disadvantage of persistently storing a pathname instead of a path, is that the garbage collection algorithm requires the use of paths to avoid cycles. The path to the root has to computed whenever it is needed by the garbage collection algorithm. A newly created (grand)child has to ask its parent for a path before it can compute a pathname.

When the name service and the garbage collector share a single table to store (forward) references to other objects, it becomes necessary to distinguish ordinary name service references from references that are maintained by sponsors of an object.

The problem is that ordinary name service references should be ignored by the garbage collector. Name service references are ignored automatically if they are stored in a directory that is not a parent of the object. Normally, a user who does not sponsor a particular object, can insert a reference to the object in a directory object but cannot insert a new reference in object's set of reverse references.

A problem may arise if a single directory contains two references to an object, one reference from a sponsor and one reference from a user. The sponsor can add the directory object's object handle to the object's set of reverse references. The problem is that the garbage collection algorithm considers the object reachable through the user's directory after the sponsor deletes its reference to object. Further research is needed to determine whether this problem should be addressed or not.

Chapter 5

Prototype implementation

5.1 Introduction

This chapter describes a prototype implementation of the Globe architecture. The prototype demonstrates that the architecture described in the previous chapters can be implemented. The goal of the implementation is to verify that the use of interfaces and late binding provides the flexibility to use multiple implementations of an interface or object type in a single system. This prototype is too small to demonstrate the scalability of our architecture. Furthermore, the prototype lacks supports for security and fault tolerance. Still, it demonstrates that the underlying ideas work.

The description of the prototype will use a simple WWW server[1] as a running example. The WWW server described in this chapter consists of two major components: a collection of objects that contain WWW documents, images, etc. and a program that implements a WWW service and serves standard HTTP requests sent by WWW browsers. This approach provides existing WWW browsers with access to Globe objects.

To demonstrate the flexibility of the system, the prototype allows the user to choose between different object implementations. For example, the name space consists of directory objects. The user may select dedicated directory objects or use WWW document objects that also the provide directory functionality. Similarly, the user may choose between a client/server replication object, a replication object that uses active replication, or one that uses master/slave replication. Finally, small WWW documents (such as inline images) can be clustered with the main document, or stored as independent distributed shared objects.

[1]Actually, the "server" is a WWW proxy. It is designed to examine all HTTP requests from a browser and to implement the Globe related ones locally.

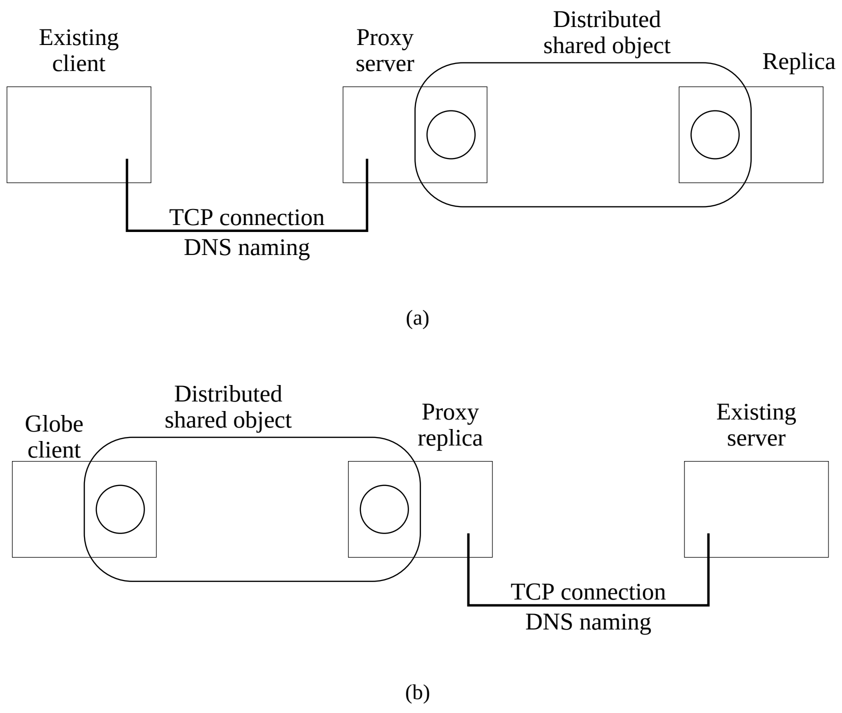


Figure 5.1. Existing clients/servers and distributed objects

This WWW server is an example of interoperability between Globe and existing (Internet) protocols. Most of the existing protocols are client/server-oriented, therefore we have to deal with two situations per protocol: an existing client using a distributed shared object, and a Globe program using an existing server. Figure 5.1 shows a standard way to solve this problem using "proxies." The Figure 5.1(a) shows an existing client connected to a proxy server that is bound to a distributed shared object. The Figure 5.1(b) shows a client using a distributed shared object implemented in such a way that operations on the object are forwarded by the proxy to an existing server.

A proxy has to implement two functions. Firstly, it has to convert requests messages of the existing protocol into method invocations on a distributed shared object and vice versa. A proxy also provides a mapping between the different naming systems. Note that in the second configuration (a Globe client using an existing server) it is possible to move the proxy replica into the client process and have the client process (without the application being aware of this) communicate with the server directly. This is possible because object implementations are loaded dynamically, and because objects completely encapsulate all aspects of communication.

An interesting example of a proxy is an e-mail proxy. A proxy server for e-mail has to implement the Simple Mail Transfer Protocol (SMTP [Postel, 1982]) to accept new mail and the Post Office Protocol (POP [Myers and Rose, 1996]) to allow reading of mail. The proxy server creates a mail message object from an incoming mail message and stores the object in a mailbox object. The fields in the header of a mail message are stored as metadata in the mail object. Furthermore, the proxy returns listings of the messages in a mailbox object and the contents of mail objects in response to POP requests.

For the reverse situation (a distributed shared object that provides access to an existing mailbox) the proxy has to fake one mailbox object and multiple mail objects per mailbox. Requests to add new documents to the mailbox are forwarded over SMTP to the real server. Lookup requests for the contents of the mailbox and the contents of mail objects are implemented using POP.

A proxy has to implement two functions. The first function is to convert request messages of the existing protocols into method invocations on distributed shared objects. The second function is to provide a mapping between the naming system used by the existing protocol and the name space for distributed shared objects. This mapping between name spaces is the most interesting aspect of the e-mail proxies.

The proxy server has to create old-style (user@domain) e-mail addresses that map to object names. The domain part is typically fixed for a particular proxy. There are, however, several alternatives for the user part. One approach is to use the object name. For example mailbox /org/foo/bar/mailbox might be mapped to /org/foo/bar/mailbox@proxy.com. An other alternative is to export all mailboxes that are listed in one directory. In this case /org/foo/mailboxes/bar might be mapped to bar@proxy.com. Yet another alternative is to store an explicit mapping in the proxy server.

Providing object names for existing mailboxes can be done automatically. All that is needed are some proxy directory objects that fake other directory objects based on DNS domains and on information returned by SMTP servers. For example, an e-mail address such as philip@cs.vu.nl might get the object name /dns/nl/vu/cs/mx/philip. The proxy directory object would create nl/vu/cs based on information stored in DNS, a directory mx for the mail service, and the proxy container object philip by querying the appropriate SMTP server for cs.vu.nl.

5.2 WWW server

A disadvantage of the e-mail proxy described above, is that when an existing client connects to a mailbox object, the connection to the proxy will be over a wide-area network. The connection used by the distributed object to supply the e-mail proxy with the necessary information will be local. This means that the distributed shared object cannot hide the latency of the wide-area network.

This is different with WWW browsers. Most WWW browsers support proxy HTTP daemons. Although they were originally developed to provide support for firewalls, currently, these daemons are often used to provide site-wide caching. A HTTP proxy can be used as a local service, which provides existing browsers with access to all distributed shared objects. In contrast, the e-mail proxy provides access only to e-mail objects in a single domain.

Architecture of the WWW server Figure 5.2 shows the various parts of the WWW server. In this figure, rectangles represent processes. Labels in the corners are the names of the programs that are running. For example, proxy for the actual proxy, locServ for the location service, and ps1 for an object repository.

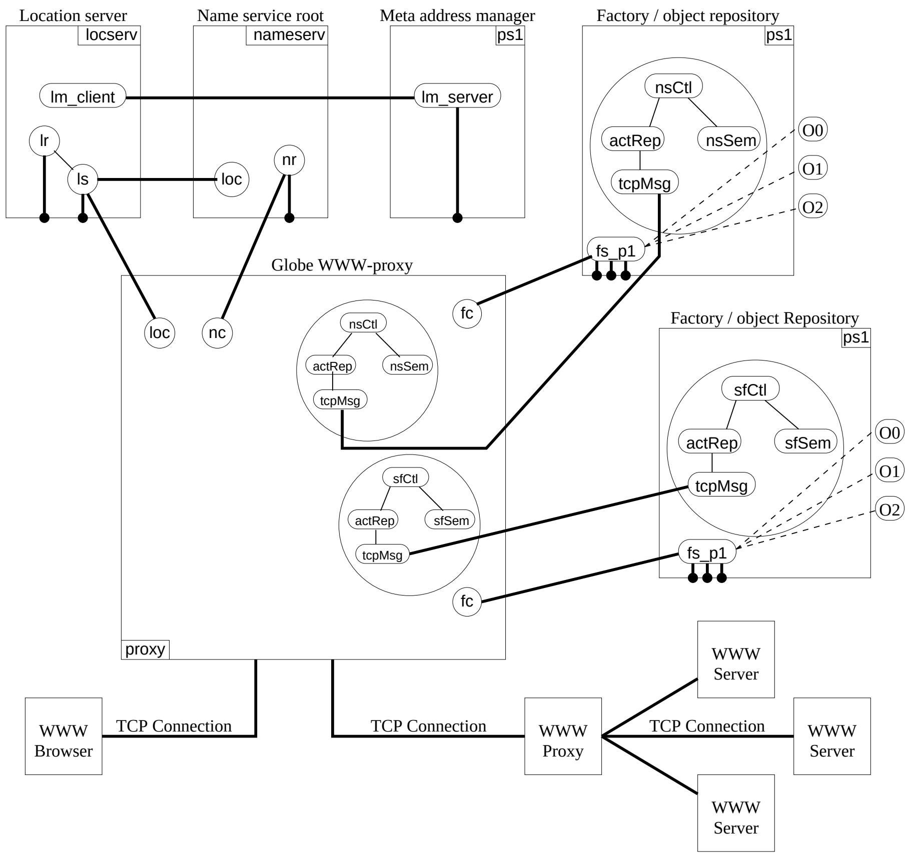


Figure 5.2. Globe WWW-proxy

Circles represent representatives and ovals represent local objects. The boundaries of distributed shared objects are left out, only representatives and communication channels are present. Thin lines between objects are local object references (pointers to interfaces). Thick lines between objects are communication channels, and thick lines that end in a heavy dot are communication endpoints that are ready to accept new connections.[2] The various parts of Figure 5.2 are discussed in the following sections. The following components will be discussed:

- Local objects in general in Section 5.3 and distributed shared objects in Section 5.4.
- Control, replication, and semantics objects in Section 5.5, followed by persistence and garbage collection.
- The location service, name service, address managers, and object repositories are described in Section 5.8.
- WWW documents are described in Section 5.9.
- System objects (such as communication objects), libraries, and binding are described in Section 5.10.

A summary of all the interfaces that are used in this prototype can be found in Appendix B.

5.3 Interfaces, Classes, and Compositions

As described in Section 2.2.1, an interface is a data structure that contains a list of pointer pairs. The first pointer points to the method implementation and the second pointer points to the state. This list is preceded by a pointer to a standard object interface and an unused field which is reserved for future use.

All objects provide the standard object interface stdObj. The methods of this interface are init, cleanup, getinf and relinf. Class objects implement the class interface. The class interface contains two methods: create and destroy.

The interfaces class and stdObj are used to create and destroy objects. Additionally, the standard object interface provides access to other interfaces, and serves as a generic object reference.

The create method in the class interface creates a new object. A pointer to the new object's standard object interface is returned. The first method that has to be invoked on a new object is the init method in the stdObj interface. This allows the object to initialize its state, and informs the object about its name-space context (see Section 5.10.1).

[2]Passive communication endpoints (i.e., communication endpoints that wait for incoming connections) exist when connection oriented communication protocols are used. For connectionless communication, they typically do not exist as separate entities.

```
interface stdObj p_create(c, ctx)
{
        o = c−>create();
        ctx_register(ctx, o);
        o−>init(ctx);
        return o;
}
```

Figure 5.3. Implementation of p_create

These two method invocations are combined in the p_create library function (shown in Figure 5.3). In this chapter, implementation details are presented in a modified version of the C programming languages. The two main extensions are interface types and syntax for method invocations. In Figure 5.3, the return type of the function p_create is interface stdObj. This type represents a pointer to an interface table for the standard object interface.

Figure 5.3 contains two examples of method invocations: c->create() and o->init(ctx). The variables c and o are interface pointers. In the actual implementation, macros are used for method invocations. For example, the method invocation o->init(ctx) is rewritten as STDOBJ_init(o, ctx). Some examples invoke a method from a specific interface. For example, self->stdObj.cleanup(1) invokes the cleanup method in the stdObj interface. In the actual implementation, the getinterface method is used to obtain a pointer to the standard object interface. The variable self is often used in method implementations to denote the current object. Finally, many examples omit variable declarations and types of formal parameters.

The function p_create gets two parameters: a pointer to a class interface (c) and a name-space context. This name-space context is passed to the function ctx_register to register the new instance in the local name space, which will be described in Section 5.10.1.

The new instance is created by invoking the create method on a class object. The create method returns a pointer to the standard object interface of the new object. The init method is invoked to allow the new object to initialize itself and a pointer to the standard object interface is returned to the caller.

Deleting an object is more complicated. Basically, the procedure to delete an object is the reverse of the procedure to create an object: first the cleanup method is invoked on the object, followed by an invocation of the destroy method on the class object. This simple procedure is complicated by the use of reference counting to keep track of interface references.

Each time an object returns an interface pointer (usually as the result of an invocation of the getinf method), it increments an internal counter that keeps track of the number of references to its interfaces. These interfaces are released through an invocation of the relinf method. Note that every interface has a pointer to the standard object interface, therefore it is possible to call relinf (and getinf) from all interfaces.

The relinf method decrements the reference count and returns to the caller whether the reference count dropped to zero or not. To destroy an object, the caller needs a pointer to the object's class object. This pointer is returned by the relinf method when the reference count is zero. Otherwise, the method returns a NULL pointer.

```
#define INF_FREE(i) ((i) ? p_inffree((i)->soi, (i)) : (void)0)

void p_inffree(o, i)
{
    c= o->relinf(i);
    if (c)
    {
        o->cleanup(0);
        c->destroy(o);
        INF_FREE(c);
    }
}
```

Figure 5.4. Release of an interface reference

Figure 5.4 shows how releasing an interface is implemented. The macro INF_FREE is used to release interfaces in C because only macros can provide the necessary polymorphism needed to extract the standard object interface pointer from an arbitrary interface in a type-safe way. The INF_FREE macro expands to a call to the p_inffree library function which does the actual method invocations. Note that the pointer to the class interface has to be released after the object is destroyed. This causes a recursive call to p_inffree.

The function p_inffree invokes the cleanup method with the parameter zero. The value zero indicates a full cleanup. Below, the method cleanup will also be invoked with the parameter one, which specifies a partial cleanup. A partial cleanup of an object's state is needed to release callback references.

A further complication of freeing interfaces is the use of callback interfaces. Not only the references to interfaces handed out to the user of the object are counted, references to callback interfaces are counted too. An object has to maintain separate counters for normal references and for references to callback interfaces. Without separate reference counters, many objects will not be deleted due to cyclic data structures. For example, object A holds a normal reference to object B, and object B hold a callback reference to object A. Object A releases its reference to object B after the last normal reference is released. With a separate counter for references to callback interfaces, the reference from object B does not count as a real reference and both objects are deleted after all normal references to object A have been released.

An object with a callback interface is destroyed in two steps. After the reference count for the normal interface drops to zero, the object performs a partial cleanup which releases all normal references to other objects. However, the object should continue to accept method invocations that are invoked on callback interfaces.

Objects recursively release references to normal interfaces but keep references to callback interfaces. After a while, there is a collection of objects with only references to callback interfaces. By construction, this collection of object references forms a directed acyclic graph. In this graph, there is at least one object with a reference count of zero. This object is destroyed, and releases its references to callback interfaces during the cleanup. After this object is destroyed, there is another that can be destroyed, and so on.

```
interface class relinf(infp)
{
    if (self->ref_count > 1)
    {
        self->ref_count--;
        return NULL;                        /* Object is still in use */
    }
    if (self->cb_ref_count != 0)
    {
        self->stdObj.cleanup(1);            /* Partial cleanup */
        if (self->ref_count > 1 || self->cb_ref_count != 0)
        {
            self->ref_count--;
            return NULL;
        }
    }

    /* Reference count is 1, return class interface */
    return &classObj.class;
}
```

Figure 5.5. Implementation of relinf

When an object detects that all callback references are released, it performs the final cleanup. The kind of cleanup is a parameter to the cleanup method: a value of zero requests a full cleanup; other values request a partial cleanup. The implementation of the relinf method for an object with a callback interface is shown in Figure 5.5. The method relinf gets the original interface pointer as parameter. This parameter is currently not used but can be used for debugging purposes (to implement a separate reference count for each interface, instead of one reference count for all interfaces).

The pointer to the class interface that is returned by the relinf method is created on the fly. In this prototype, references to class objects are not counted (class objects are never destroyed). To create a pointer to the class object of an object, it is sufficient to return a pointer to the structure that contains the interface table of the class interface. Note that method implementations have direct access to the internals of the class object that they belong to.

Callback interfaces use a different relinf implementation that decrements the cb_ref_count variable instead of ref_count. Most of the code for the init, cleanup, getinf, relinf, create and destroy methods is generated by the interface generator ifgen, which will be described in Section 5.11.1.

The interface generator optionally provides the objMgt interface, which has one method: getInfList. This method returns a list of interface names and identifiers supported by the object. This interface can be used (for example) to determine the list of interfaces implemented by a distributed shared object.

Some class objects are linked with applications, others are loaded at runtime. Classes that are loaded at runtime are instances of a meta-class object. This prototype contains the meta-class object /builtin/classes/class.elf, which uses the dlopen and dlsym library functions of the underlying operating system to access an ELF shared library. When a class object is loaded, a pointer to the bind library function is passed to the initialization function of the class object. This allows the class object to bind to object objects, including objects in the run-time system. In fact, this is the only way a dynamically-loaded class object can access the run-time system. This way, the design of the run-time loader is simplified and portability across platforms is enhanced.

A composition is a collection of objects grouped together by a compositor. The compositor creates the various subobjects, registers them in its name space, installs overrides, and initializes the subobjects. The compositor also creates the interface tables it exports based on its own methods (for object management) and based on methods or interfaces provided by subobjects. Class objects for compositors can be generated completely from a simple composite object specification.

5.4 Distributed Shared Objects

Distributed shared objects consist of representatives in different address spaces. Binding to a distributed shared object involves a local object that connects to the representatives of the distributed shared object that exist in other address spaces. This local object has to implement the distrObj interface to be compatible with the binding mechanism (see Section 5.10.1). This interface provides two methods: init and gethandle. A local object that is bound to a distributed shared object is called a representative.

The init method directs a local object to connect to a distributed shared object. The name that was used to look up the distributed shared object in the name service, the object handle, and a set of contact addresses are passed as parameters. The local object may setup a connection immediately, or it may defer this until the first method invocation. Note that when the local object connects to the distributed shared object, it should verify whether the object handle passed as a parameter matches the object handle stored in the distributed shared object. This is an end-to-end check to ensure the validity of the contact addresses returned by the location service. If an attempt to use one contact address fails, the local object should try the next address. This kind of error recovery is not implemented in the prototype. The second method, gethandle, returns the object handle that belongs to the distributed shared object.

The distrObj interface is usually implemented by both the replication object and the control object. The replication object contains the real implementation of the distrObj interface, which uses a communication object to connect to the distributed shared object. The control object implements the init method in the distrObj to verify the semantics parts of the protocol identifiers in the contact addresses. If the contact addresses contain the correct protocol identifiers, the control object invokes the init method on the replication object. The verification of protocol identifiers is described in Section 5.5.1.

Most distributed shared objects also provide the distrCtrl interface. This interface contains four methods to create new instances of a particular class of distributed shared objects (by a factory, see Section 5.8.4) and to create new persistent representatives.

The create method directs a local object which is not already connected to a distributed shared object to become a new distributed shared object and to create a contact point that will allow other local objects to connect. This method returns a set of contact addresses for the newly created contact point. This method is typically invoked by a factory. The factory will allocate an object identifier for the new distributed shared object and register the object identifier and the contact address with the location service. The location service returns an object handle based on the object identifier. The factory tells the object about its object handle by invoking the setHandle method.

The last method[3] in the distrCtrl interface is replicate. This method asks the distributed shared object to create a persistent representative in the address space in which the method is invoked. The representative on which the method is invoked allocates a contact point, registers the corresponding contact address with the location service, and, depending on the distributed shared object's replication strategy, obtains a copy of the object's state. The method is usually invoked by an object manager.

5.5 Control, Replication and Semantics Objects

In Section 3.2.4, we described replication of the state of a distributed shared object. Replication is implemented in two objects, the control object and the replication object, and uses two interfaces, the control interface and the replication interface. The

[3]For historical reasons, the distrCtrl method also contains the newLink method, which is equivalent to the addRef method in the garbage collection interface gc (see Section 5.7).

actual replication protocol is implemented in the replication object. The control object provides the necessary glue between the replication object and the semantics object.

The replication[4] interface provides four methods: init, start, invoked and send. The control interface is a callback interface for the replication object. The control object passes a reference to this interface as an argument to the init method. The control interface provides three methods: handle_request, getState and setState. The handle_request method is used by the replication object to invoke a method on the semantics object. The replication object passes a packet with the marshaled request and arguments. The control object unmarshals the packet and performs the method invocation. The results of the method invocation are marshaled into a reply packet and returned to the replication object. The methods getState and setState, marshal and unmarshal the state of the semantics object, respectively.

A method invocation on a semantics object that may block requires an extra boolean output parameter (called blocked) which informs the control object whether the operation blocked or not. The semantics object reports that a method invocation blocked by setting the parameter blocked to true and has to return to the control object with its state unmodified. The control object forwards the fact that the semantics object blocked to the replication object. This is done in two different ways depending on what kind of method was invoked on the control object. When the control object was invoked by the user of the distributed shared object, it invokes the invoked method on the replication object. The control object passes a flag as argument to the invoked method to indicate whether the semantics object blocked or not. If the replication object invoked the handle_request method on the control object to process a request that arrives over the network, then the control object returns a NULL pointer instead of a reply packet as the result of the handle_request method to indicate that the operation blocked.

An operation that blocks may have been multicast to multiple representatives. In that case, we have two alternatives for continuing. One approach is to enqueue the operation at each replication object that received the multicast request and have them retry the operation individually. The second alternative is to abort the operation and let the initiator retry. The trade-off is that the first alternative is more efficient, but is also more complex to implement.

Enqueuing an operation at multiple replication objects requires that every new representative has to get a copy of the current list of blocked operations along with a copy of the state of the object. Furthermore, when a process unbinds, pending blocked operations that were initiated by that process have to be removed from the queue. The other alternative requires the initiating thread to retry the operation regularly (each time the state is modified). A problem with this last approach is limited fairness and

[4]Actually, the replication interface is called replication0. An older interface is called replication. Likewise, the control interface is actually called control0 instead of control.

loss of efficiency if many processes try the same operations and only one of them can succeed at a time.

5.5.1 Control Object Implementation

The implementation of a control object consists of three parts:

- General initialization and cleanup routines
- Methods that implement the object-specific interfaces
- Callback methods that are called by the replication object to handle incoming requests, and to transfer the state of the semantics object

We will focus on the methods that are part of object-specific interfaces and the callback method that executes incoming requests.

The methods that handle incoming requests are relatively simple RPC server stubs. The pseudo-C code for a distributed lock object is shown in Figure 5.6. This object provides the mutex interface, which contains four methods: lock, unlock, read, and await.

First, the marshaled request is extracted from the marshaled data. A method identifier (cmd) is used to select what method should be invoked on the semantics object. For each method, the control object contains code to decode arguments, to invoke the method, and to encode the results. Most of the methods (lock, unlock, and read) do not have any arguments and, therefore, do not need code to decode them. Before a method is invoke on the semantics object, a local mutex is locked to serialize accesses to the semantics object.

The code for the unlock method simply locks the local mutex, invokes the unlock method on the semantics object, unlocks the mutex, and sets the result to OK. The code for the read method is a bit different because the read method returns a value. The result (the variable v) is encoded in the reply packet, which is stored in the variable reply_pkt.

The methods lock and await may block. For this reason these methods have an extra parameter, a pointer to the variable blocked. The semantics object writes a nonzero value in the variable when it cannot complete the operation without blocking. The control object returns a NULL pointer to tell the replication object that the operation blocked. The replication object is responsible for retrying the operation at a later time.

At the end of the handle_request function, a result code is added to the reply packet that specifies whether the stubs encountered any errors (an illegal request, arguments that could not be decoded, or results that could not be encoded) or not. The reply packet is returned to the replication object.

```
pkt_t handle_request(pkt_t pkt)
{
    reply_pkt= NULL;
    cmd= decode_cmd(pkt);                  /* extract method from request packet */
    switch(cmd)
    {
    case UNLOCK:
        /* no arguments to decode */
        lock_enter(self->lockCommon);          /* lock semantics object */
        self->sem->m_mutex.unlock();           /* invoke unlock method */
        lock_leave(self->lockCommon);          /* unlock semantics object */
        res= OK;
        break;
    case READ:
        /* no arguments to decode */
        lock_enter(self->lockCommon);          /* lock semantics object */
        v= self->sem->m_mutex.read();          /* invoke read method */
        lock_leave(self->lockCommon);          /* unlock semantics object */
        res=encode_read(&reply_pkt, v);        /* encode value */
        break;
    case LOCK:
        /* no arguments to decode */
        lock_enter(self->lockCommon);
        blocked= 0;                            /* success is default */
        self->sem->m_mutex.lock2(&blocked);
        lock_leave(self->lockCommon);
        if (blocked)                           /* tell the replication object */
            return NULL;                       /* that the request blocked */
        res= OK;
        break;
    case AWAIT:
        res= decode_await(pkt, &v);            /* extra argument */
        if (res != OK)
            break;
        lock_enter(self->lockCommon);
        blocked= 0;
        self->sem->m_mutex.await(v, &blocked);
        lock_leave(self->lockCommon);
        if (blocked)
            return NULL;
        res= OK;
        break;
    default:
        res= UNKNOWN_CMD;
        break;
    }
    encode_res(&reply_pkt, res);               /* encode result */
    return reply_pkt;
}
```

Figure 5.6. Implementation of handle_request

The implementation of the object-specific methods is more complicated. Each method contains an implementation of the state machine that was described in Section 3.2.4. The algorithm shown in Figure 5.7 starts in the start state of the state machine. The main body of the method implementation is a for loop which executes until the return state is reached. The result of the method invocation is either provided by a method invocation on the semantics object, or it is stored in the reply_pkt variable.

```
int read()
{
    op= RO_READ;
    s = RS_START;
    for(;;)
    {
        if (s == RS_START)
        {
            s= self->repl->replication.start(op);
            continue;
        }
        if (s == RS_INVOKE)
        {
            reply_pkt= NULL;
            lock_enter(self->lockCommon);
            result= self->sem->m_mutex.read();
            lock_leave(self->lockCommon);
            s= self->repl->replication.invoked(op, 0);
            continue;
        }
        if (s == RS_SEND)
        {
            /* Create a packet with the command (no arguments). */
            pkt= encode_read(READ);
            reply_pkt= self->repl->replication.send(op, pkt, &s);
            continue;
        }
        if (s == RS_RETURN)
            break;
    }

    if (reply_pkt)
        result= decode_read(replt_pkt);            /* Decode reply packet */

    return result;
}
```

Figure 5.7. Implementation of an object-specific method in the control object

The control object invokes three methods on the replication object: start, send, and invoked. The start method allows the replication object to acquire any locks that are required, get a consistent copy of the state, etc. Similarly, the invoked method allows the replication object to release locks, distribute the new state, etc., after a method invocation on the semantics object. In this example, the read method cannot block. A value of zero is always passed to the replication object.

The send method provides the replication object with a packet that contains the marshaled arguments of the operation. The replication object distributes this packet to other replication objects. The return value of this method is either the result of the method invocation or no result (a NULL pointer).

Some replication protocols differentiate between read operations, which leave the state of the semantics object unchanged, and write operations that (potentially) modify the state of the object. For this reason, the control object passes the type of operation in method invocations on to the replication object. Currently, two types are recognized: RO_READ for read only operations and RO_MODIFY for other operations.

Semantics objects use interfaces to marshal and unmarshal state that are different from the interfaces used by replication objects. The interfaces used by semantics objects are stream interfaces. The state of a semantics object is marshaled by writing marshaled versions of the state of instance variables to an output stream. To unmarshal state, a semantics object reads marshaled state from an input stream. In contrast, a replication object expects the control object to return the marshaled state of a semantics object in a network packet.

The control object bridges the gap between the semantics object and the replication object. To marshal the state of the semantics object, the control object implements the outputStream interface. The semantics object uses the write method in the outputStream stream interface to marshal its state. The control object collects the output of the semantics object into a network packet and returns it to the replication object. A similar procedure that uses the inputStream interface is used to unmarshal state.

The control object partly implements two methods of the distrObj and distrCtrl interfaces. These are, respectively, the init and the create method. The reason for implementing those methods by the control object is that the control object participates in the creation and verification of protocol identifiers. In this prototype, a contact address (which includes a protocol identifier) is simply an ASCII string. When a contact address is created, each layer prepends its own identifier to the address. When the contact address is used, each layer verifies that its identifier is present and subsequently strips it off.

For example, a TCP communication object may generate the string TCP(192.31.231.174/33745). The protocol identifier is TCP, and the Internet address and TCP port number are, respectively, 192.31.231.174 and 33745. All replication objects use a small message passing layer on top of TCP which adds the string MSG:. In a distributed lock object that uses active replication, the replication object adds the string ACT0.0: and the control object adds MUTEX0.0:. The result is MUTEX0.0:ACT0.0:MSG:TCP(192.31.231.174/33745).

5.5.2 Replication Object Implementation

The prototype provides several replication objects: csRep uses a simple client/server protocol, actRep provides active replication, and msRep uses master/slave replication.

In addition to the implementation of a replication protocol, the replication object also implements persistence and garbage collection. In this section, only the implementation of the replication protocol will be described. The example that will be used is the master/slave replication object. Instead of an update protocol, the master/slave replication object described below uses an invalidate mechanism to inform slaves about a new state. The slaves fetch a new copy of the state whenever they have to execute a read operation. The use of invalidate messages instead of update messages reduces the fault tolerance of the original master/slave replication algorithm. The main benefit, however, is increased performance of read operations in a distributed shared object with a high read/write ratio.

The master/slave replication object operates in one of two modes: it is either the master or it is a slave. The replication object determines the correct mode when it is initialized. Persistent representatives store the mode in their persistent state. A replication object that connects to another replication object (for example during binding), is a slave. The replication object that “creates” a new distributed shared object is the master.

Operations in the master’s address space can be invoked directly on the local semantics object. After a modify operation, the replication object sends invalidate messages to slaves that hold a copy of the state. Modify operations that are invoked on a slave have to be forwarded to the master. Read operations can be executed locally by a slave after a copy of the state has been fetched.

```
state_t start(operation)
{
    if (self->mode == MASTER)
        return INVOKE;

    if (operation == MODIFY)
        return SEND;

    if (!self->semValid)
    {
        /* Send a request to the master for the state */
        pkt= request_state();
        self->ctrl->control.setState(pkt);
        self->semValid= TRUE;
    }
    return INVOKE;
}
```

Figure 5.8. Implementation of the start method

Figure 5.8 shows the implementation of the start method. On the master, the method is always executed locally. A state transition to the INVOKE state is returned to the control object. On a slave, a transition to the SEND is returned to signal that a modify operation should be forwarded (to the master). A read operation can be executed locally on the slave, but the replication object has to ensure that an up-to-date copy of the state is present in the semantics object. If necessary, a request for a copy of the state is sent to the master. The implementation of the function request_state is not shown here. This function sends a request for the state to the master and returns a packet with the marshaled state of the master's semantics object. This packet is passed to the control object which downloads the state in the local semantics object.

```
pkt_t send(operation, req_pkt, state_ptr)
{
    req_pkt= prepend_op(operation, req_pkt);
    reply_pkt= self−>comm−>comm.sendrec(req_pkt);
    ∗state_ptr= RETURN;
    return reply_pkt;
}
```

Figure 5.9. Implementation of the send method

The send method (shown in Figure 5.9) is straightforward. The replication object prepends the kind of operation (read or modify) to the packet with the marshaled method name and arguments, which are provided by the control object. The sendrec method of the communication object is used to send the packet to the master. The replication object directs the control object to the RETURN state and returns the marshaled results.

```
state_t invoked(operation, blocked)
{
    if (!blocked)
    {
        if (operation == MODIFY)
        {
            retry_reqs();
            broadcast(self−>cv);
            invalidate_slaves();
        }
        return RETURN;
    }

    wait(self−>cv);
    if (self−>state == MASTER)
        return INVOKE;
    return SEND;
}
```

Figure 5.10. Implementation of the invoked method

Figure 5.10 shows the implementation of the invoked method. The replication object has to test for two conditions: whether the operation is a modify operation or not and whether the semantics object blocked or not. A method invocation that did not block and potentially changed the state of the semantics object requires an invalidate message to be sent to those slave replication objects that have a copy of the state. Furthermore, all threads that are blocked on guard failures are woken up.

To handle a blocking operation, the replication object first blocks the current thread until another thread modifies the state of the semantics object. The replication object simply retries the operation if it is the master. On a slave, the replication object can simply retry the read operation (modify operations are already forwarded to the master). Instead of retrying the read operation locally, the operation is forwarded to the master to avoid fetching new copies of the state (polling) until the guard no longer fails.

The replication object has to implement two methods in the callback communication interface commRdCB. This interface will be described further in Section 5.10.5. The first method, msg_arrived, is invoked by the communication object when a point-to-point message arrives. The replication object uses point-to-point messages to invalidate the state in slave replication objects. The method req_arrived is used for request/reply style communication (i.e., the method req_arrived has to return a reply message for each request message it gets as a parameter). Request/reply style communication is used by the replication object to forward method invocations (the reply message contains the results of the method invocation) and to request a copy of the state (the marshaled state of the semantics object is sent in the reply message).

```
void  msg_arrived(pkt)
{
    if  (is_invalidate(pkt))
    {
        self−>semValid=  FALSE;

        /* State has changed, retry read operations */
        broadcast(self−>cv);
    }
}
```

Figure 5.11. Implementation of the msg_arrived method

The msg_arrived method (shown in Figure 5.11) checks if the message is actually an invalidate message. If so, the current state is marked invalid and a wakeup signal is sent to any blocked threads. The wakeup signal is sent using the broadcast function, which is part of the collection of local thread synchronization primitives discussed in Section 5.10.6.

The req_arrived method (shown in Figure 5.12) recognizes three different classes of requests: a request for the state of the semantics object, a read operation, and a

```
pkt_p  req_arrived(ref,  pkt)
{
      pkt= del_header(pkt, &hdr);           /* delete and decode repl. header */
      if (is_getstate(hdr))                 /* request to send copy of state */
      {
            remember_slave(ref);            /* remember to send invalidate */
            return self->ctrl->control.getState();
      }
      if (is_read_req(hdr)                  /* read-only operation */
      {
            reply_pkt= self->ctrl->control.handle_req(pkt, ref);
            if (reply_pkt == NULL)
            {                                         /* op. blocked */
                  enqueue_pkt(ref, hdr, pkt);         /* retry later */
                  return NULL;
            }
            return reply_pkt;                         /* send results */
      }
      if (is_modify_req(hdr)                          /* modify operation */
      {
            reply_pkt= self->ctrl->control.handle_req(pkt, ref);
            if (reply_pkt == NULL)
            {
                  enqueue_pkt(ref, hdr, pkt);
                  return NULL;
            }
            retry_reqs();                   /* retry queued requests */
            broadcast(self->cv);            /* wake-up local blocked threads */
            invalidate_slaves();            /* invalidate copies at slaves */
            return reply_pkt;               /* send results */
      }
}
```

Figure 5.12. Implementation of the req_arrived method

modify operation. First, the implementation strips off the header and examines the header to determine the kind of request that was received.

A request for a copy of the state simply returns the packet that is returned by the control object. In addition, the slave (actually, an identifier for the connection to the slave) is added to a list of slaves that should get an invalidate message when the state is changed.

A read request is passed to the control object. The control object returns the marshaled results after a successful method invocation on the semantics object. These results are returned to the communication object. The control object returns a NULL pointer when the method invocation on the semantics object blocks. A blocking request is added to a queue with blocked requests. A NULL pointer is returned to the communication object to indicate that the reply message will be sent at a later time.

A modify request is similar to a read request except that this request also has to retry other blocked requests, send invalidate messages, and wakeup blocked threads.

```
void retry_reqs()
{
    list= self−>req_queue;
    self−>req_queue= NULL;
    tail= &self−>req_queue;
    while (list)
    {
        e= list;
        list= list−>next;
        reply_pkt= self−>ctrl−>control.handle_req(e.pkt, e.ref);
        if (reply_pkt != NULL)
            self−>comm−>comm.reply(ref, reply_pkt);
        else
        {
            e−>next= NULL;
            *tail= e;
            tail= &e−>next;
        }

    }
}
```

Figure 5.13. Implementation of the retry_reqs function

Figure 5.13 shows the implementation of the retry_reqs. This function is called from the methods invoked and req_arrived to retry blocked requests. Each request in the queue is handed in turn to the control object. The reply method is invoked to send a reply packet if the method is executed successfully. Otherwise, the request is re-enqueued.

5.5.3 Semantics Object Implementation

An interesting example is the semantics object of a simple binary semaphore dl (distributed lock). This lock object implements the mutex (MUTual EXclusion) interface. The mutex interface contains four methods: lock, unlock, read, and await. The lock and unlock methods, respectively, decrement and increment the value of the binary semaphore. The read method returns the current value of the object: 0 if the object is locked and 1 if the object is unlocked. The await method blocks until the value of the object is equal to the parameter passed in the method invocation.

The mutex interface contains two methods that may block: lock and await. In the interface between the control object and the semantics object, these two methods get an extra parameter. A new interface is needed to accommodate those parameters. This interface is called m_mutex and is shared only between the semantics object and the control object.

Figure 5.14 shows the implementation of the methods in the m_mutex interface in pseudo-C. The state of the object is stored in the instance variable v. The lock method

```
void lock(int *retryp)
{
      if (!self->v)
      {
            *retryp= 1;
            return;
      }
      self->v= 0;
}

void unlock()
{
      self->v= 1;
}

int read()
{
      return self->v;
}

void await(int value, int *retryp)
{
      if (value != self->v)
            *retryp= 1;
}
```

Figure 5.14. Implementation of mutex methods

indicates a guard failure when the object is locked. The unlock and read methods respectively write and read the state of the object. The await method consists of nothing more than a guard.

```
marshal(outputStreamInf out)
{
      if (self->v)
            str= "1";
      else
            str= "0";
      out->write(str, strlen(str));
}

unmarshal(inputStreamInf in)
{
      buf= in->readbuf(&size);
      vc= buf[0];
      self->v= (vc != '0');
}
```

Figure 5.15. Implementation of marshaling methods

Figure 5.15 shows the implementation of the marshal and unmarshal methods. The state is marshaled as an ASCII “0” or “1.” This byte is written to the output stream provided by the control object (which converts the byte to a packet and returns the packet to the replication object). The unmarshal method reads a single byte from an input stream and sets the instance variable v.

Optimizations The interaction between the control object and replication object can be optimized. The invoked method of the replication object is necessary only when the method invocation actually blocks and the start method is in many cases an empty operation that returns only the next state (SEND or INVOKE).

The optimizations mainly reduce the overhead for operations which can be executed locally (i.e., without communicating). A simple, local method invocation involves four method invocations: (1) the initial method invocation on the control object by the user of the distributed shared object, (2) a start method invocation on the replication object, (3) the method invocation on the semantics object, and (4) the invoked method on the replication object. The number of method invocations can be reduced to two if the two method invocations on the replication object can be avoided.

The invoked method is easy to avoid: the only reason for the invoked method is to return the next state to the control object. For example, without optimizations, the start method returns a transition to the INVOKE state, and the invoked method returns a transition to the RETURN state. The start method can return a transition to an "INVOKE-followed-by-RETURN" state.

For many replication objects, the start method just returns the first real state. Calling the start method can be avoided if the real initial state is stored in the control object. Note that the initial state depends on a number of conditions: (1) the kind of operation (read or modify), (2) the replication algorithm implemented by the replication object, and (3) the role of this address space (client or server for a client/server replication object). For this reason, the control object has to ask the replication object during initialization for the initial states for the different categories of operations.

A problem caused by these optimizations is that the replication object does not always know when the state of the semantics object is modified: the control object invokes the method directly on the semantics object without informing the replication object whether the state of the semantics object was changed or not. This is problem because the replication object is responsible for retrying blocked operations. A solution to this problem is to store information about blocked operations in the control object. However, this solution complicates the interface between the replication object and the control object.

The prototype contains an additional set of replication and control interfaces that implement these optimizations, but the gains are probably not worth the increased complexity.

5.6 Persistence

As discussed in Section 3.3.1, persistence can be implemented external to a representative (shown in Figure 3.21 on page 79) or it can be integrated with the replication object (shown in Figure 3.25 on page 84). The prototype provides both mechanisms to implement a persistent distributed shared object. Both mechanisms allow a *representative* of the distributed shared object to become persistent. This is one of the two types of persistence mechanisms described in Section 3.3. The other approach is to store the state of the entire distributed shared object in a persistent distributed *state* object.

The two mechanisms that are implemented differ in the placement of the persistence support with respect to the replication object. In one approach, associated with the persist interface, persistence is placed outside the representative. The representative receives persistent state from the object repository and provides the object repository with new persistent state when the state of the representative is modified. The approach taken with the other interface is that the representative is provided with a persistent state object, which implements the state interface. The replication object invokes methods on the state object to read old persistent state or to update the persistent state. As a result, the persistence interface is used only to activate and passivate the representative and does not explicitly deal with the persistent state. This approach is supported by the persist2 interface.

At least one of the two persistence interfaces has to be provided by a persistent distributed shared object (it is possible to provide both). Methods in these interfaces are invoked by the object repository that contains the persistent representative. Associated with these two interfaces are two callback interfaces persistCB and persist2a. These callback interfaces are implemented by the object repository.

Both interfaces provide a create and a restart method. The create method is invoked by the object repository when the persistent representative is created. The object handle of the persistence manager is passed as an argument and a contact address is returned.

Binding to a distributed shared object using a contact address provided by a persistent representative is different from binding to a nonpersistent distributed shared object. The reason is that the contact address for a persistent representative does not contain an actual network address, transport protocol identifier, etc. Instead the contact address contains the persistence manager's object handle. A local object that tries to use this contact address (during binding) has to bind to the object manager first. This is illustrated in Figure 5.16.

1. The application calls the bind function to bind to the distributed shared object.
2. The bind function calls the name service, location service, etc. to create a new local object. This is not shown. Finally, the bind function invokes the init method

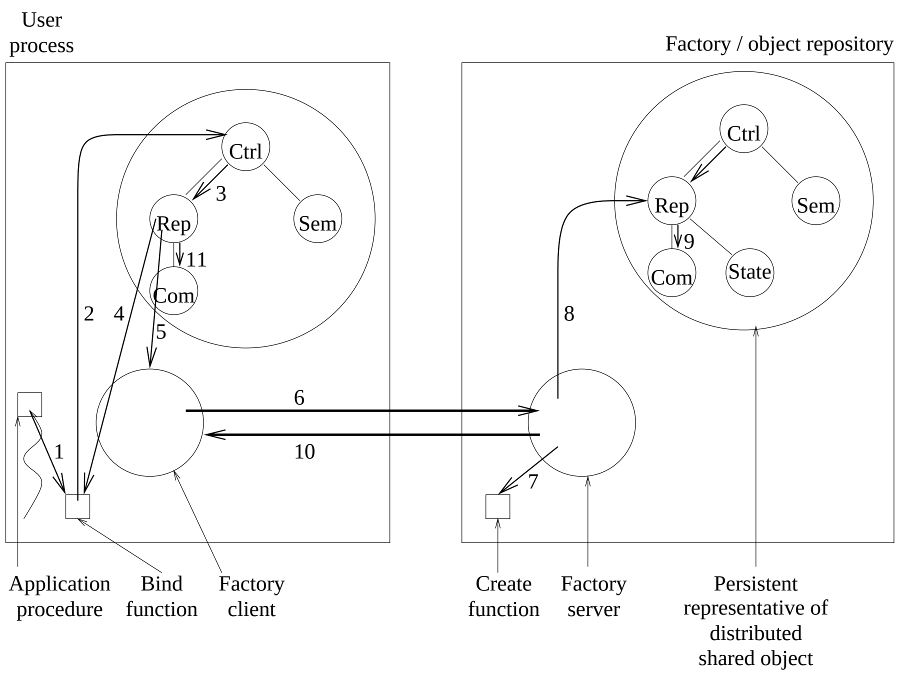


Figure 5.16. Binding to a persistent distributed shared object

in the distrObj interface. This method is implemented by the control object to verify its part of the protocol identifiers in the contact addresses that are passed as parameters.

3. The control object strips off its part of the protocol identifiers in the contact addresses and invokes the init method on the replication object.

4. Instead of using the communication object to set up a connection to a remote representative, as is the case for nonpersistent objects, the replication object binds to the factory object. As described in Section 5.3, a pointer to the bind function is passed to the initialization routine of a dynamically-loaded class object. This means that every object calls the same bind function (the one in the run-time system that is linked with the application).

5. It then invokes the startObj method in the factory interface to activate the persistent representative (on the repository). The distributed shared object's object handle is passed as a parameter.

6. The factory client sends a message to the factory server.

7. In this example, we assume that the representative is not already active. The factory server creates a new local object.

8. The server invokes the restart method. The restart method directs the local object to load its persistent state from the state object and become an active persistent representative.

9. The replication object uses the communication object to create a new contact point. The communication object returns a contact address for this contact point. The replication object prepends its protocol identifier to the contact address and returns the contact address to the server object. Depending on the replication protocol, the replication object may activate representatives at other object repositories.

10. The contact address is sent back to the factory client and the client returns the contact address to the replication object in the user process.

11. The replication object uses the communication object to connect to the persistent representative.

Specific to the persist interface are the methods setCB and unmarshal. The unmarshal method provides the object with previously saved state. An invocation of the setCB method passes a reference to the persistCB interface. The persistCB interface contains only one method: newstate. This method returns an outputStream interface which can be used by the representative to save its state. Each time the representative needs to save its state it obtains a new output stream, writes the state, and closes the stream. This means that the representative can save only its entire state.

In addition to the create and restart methods, the persist2 interface contains an explicit shutdown method. This method is used by the object repository to ask a representative to shut down.

5.7 Garbage Collection

The garbage collection interface gc is provided by directory objects and by (persistent) distributed shared objects that are allocated dynamically and need to be garbage collected at some time in the future. The gc interface provides eight methods: addRef, checkRef, reachable, getName, dirEntry, getPath, delRef, and listRefs.

The addRef method adds an object handle of a directory object to the list of directory objects that hold references to this object. Note that for historical reasons, the addRef method is duplicated in the distrCtrl interface. The checkRef method informs the object that a reference to the object has been deleted from a directory object. The addRef and checkRef methods are invoked on the object by processes that, respectively, add and delete references to the object. This is an opportunity to run a garbage collection cycle. The object handle of the directory object is passed as an argument.

The reachable method implements the actual garbage collection algorithm. This method directs the object to verify that it is still reachable. A time value (interval) is passed as a parameter to specify the level of consistency desired. For example, if the object verified 30 seconds ago that is was reachable and the reachable method is called with a parameter of 60 seconds, then the object can immediately return that it is reachable.

The getName method returns a name in the name space that leads to the object. Names are assumed to be in the worldwide name space. The object computes the name by asking one of its parent directories for the name of the directory object and the directory entry that refers to the object. The object returns the concatenation of the two names.

The getPath method is used by the garbage collection algorithm to obtain reachability information from an object's parents. The result of a getPath method is either a list of object identifiers on a path to a root object or a collection of records that describe (part of) the graph that connects the ancestors of the object. The garbage collection algorithm iterates until the object knows a path to the root or until the object and its ancestors know (by analyzing the graph) that they are disconnected from the root.

Only directory objects actually implement the dirEntry method. This method returns the directory entry that refers to a particular object handle passed as an argument. The delRef method deletes a stored object handle of a parent directory object. The listRefs method returns a list of all references to parent objects.

Most of the gc interface is implemented by the replication object. However there is one exception: the method dirEntry is provided by the semantics object. As described in Section 4.5.9, most methods in the gc interface (addRef, reachable, getName, dirEntry, getPath) have to be linearizable[5] even when the replication object implements a weaker consistency protocol. With a weaker consistency protocol, it is possible that the garbage collection protocol deletes objects that are in fact reachable. In the prototype, the garbage collection implementation always uses client/server remote procedure calls, independent of the replication protocol used by the replication object. The implementation of the garbage collection interface is a straightforward translation of the algorithm described in Section 4.5.4.

[5]Alternatively, it is possible to implement the union rule.

5.8 Services

5.8.1 Location Service

The location service provides two interfaces to look up and register contact addresses. Two additional interfaces, related to the garbage collection of contact addresses, are described in the next section. The first location service interface, locRes, provides a single method, called lookup. This method takes an object handle as parameter and returns a list of contact addresses. In the prototype, nondistributed implementation of the location service, this method returns all contact addresses that belong to an object. Section 4.4.3 describes how a small subset of contact addresses can be constructed that contains both contact addresses for all protocols used by a distributed shared object and contact addresses at different distances from the invoker.

The second interface, locServ, provides five methods: register new distributed shared objects (regObj), register new contact addresses for a distributed shared object (regAddr), replace a contact address with another contact address (update), and unregister both addresses and objects (respectively, delAddr and delObj).

The method regObj has three input parameters. The first parameter is the object identifier of the distributed shared object that is to be registered. The second parameter is a set of contact addresses for the object. The third parameter is the object handle of a location service address manager. These address managers will be described in the next section. The method returns an object handle for the distributed shared object and address identifier. The address identifier is used to update or delete a specific contact address for a distributed shared object.

The reason for distributing the methods of the location service over two interfaces is that the two interfaces are used in completely different situations. The lookup functionality is needed for binding to a distributed shared object. Special bootstrapping mechanisms are needed to get access to the location service. It is not possible to simply bind to the location service. The bind function always does a name lookup first, followed by a lookup in the location service, etc. Without access to the location service it is not possible to call the bind function. In fact, what is needed is a set of contact addresses for the location service. This way, the last part of binding (destination selection, implementation selection, loading a suitable class object, etc.) can be executed. It is possible that operating system specific mechanisms, such as local RPCs, are used during the bootstrap to obtain a set of contact addresses for the location service.

In contrast, for the registration services we can treat the location service as a distributed shared object. Furthermore, the flexibility to actually bind to the location service for registration services provides the opportunity to use standard Globe facilities such as name spaces, security mechanisms, etc. to control access. For example, different organizations may provide registration services, for different fees, with different performance guarantees, etc.

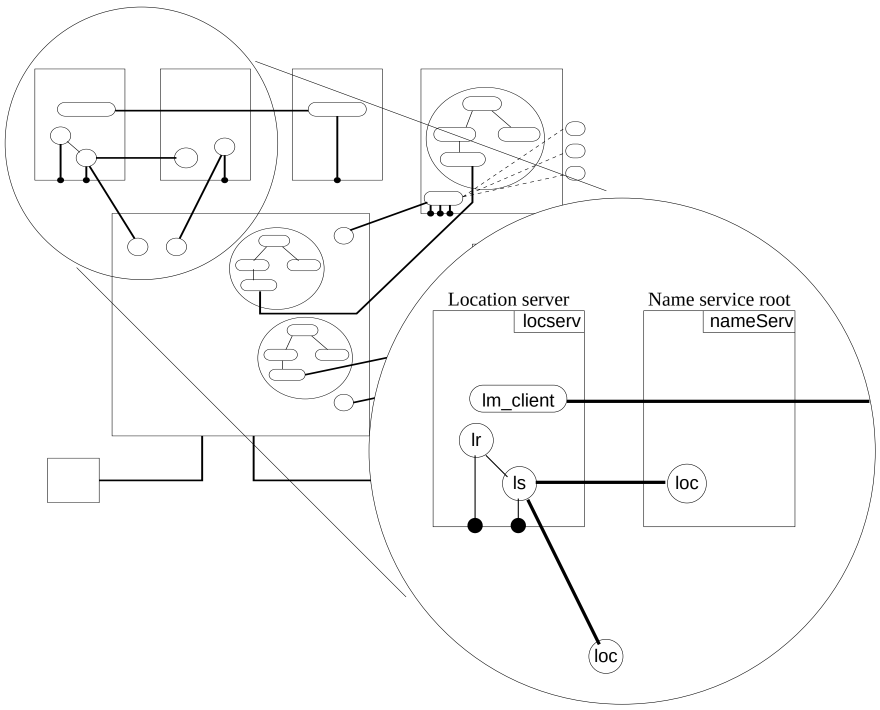


Figure 5.17. The location server

The location service in this prototype is implemented as a simple, centralized location server. This explains why *all* contact addresses for an object are returned by the lookup operation. This server can be viewed as a single leaf node in a worldwide distributed location service. The implementation consists of three local objects and one program. This is shown in Figure 5.17.

The location service is accessed as a distributed shared object. The implementation of the distributed shared object is asymmetric and does not follow the standard architecture for the implementation of distributed shared objects. This means that there are no semantics, control or replication objects. Instead, one local object called loc implements an RPC-stub on top of the communication object tcpMsg. However, the implementation *does* follow the requirements for distributed shared objects listed in Section 2.3. In particular, the location service provides contact points with contact addresses and is accessed through a local object.

This stub provides both interfaces to the location service. The server side is split into two parts, using two objects. One object, called lr (location root), maintains the location service database and implements the RPC for the lookup method. The other object is called ls (location services). This object accepts the RPCs that correspond to the methods in locServ interface. The RPCs are implemented by invoking methods on the lr object. Both the lr object and the ls object have their own tcpMsg object and both objects provide contact points to handle incoming network connections.

The location server program provides an execution context for the two objects. The program also assists the lr object in saving and restoring the persistent state of the location service. Within the program, the two objects communicate using two application-specific interfaces, called lr interface and the ls interface. The ls object provides the ls interface with methods start, restart, and gethandle. The lr interface, provided by the lr object, also contains the methods start, restart, and gethandle. Additionally, the lr interface contains methods to access the location service database: register, regAddr, update, lookup, regMan, and gc.

The start methods are invoked by the program when the location server is started for the very first time. The methods allocate object identifiers, create contact addresses, etc. The restart methods are invoked when the location service is restarted after a crash or a reboot. The gethandle methods return the object handles of the two distributed shared objects provided by the lr and the ls objects.

The methods regMan and gc will be discussed in the next section. The set of methods that provide access to the database is incomplete. Two more methods are needed to delete a contact address (delAddr) and to delete an object (delObj). These two methods are not strictly necessary because obsolete addresses and unreachable objects will be deleted by the garbage collector.

5.8.2 Location Service Address Managers

Section 4.5 described how the location service is used by the garbage collector to distinguish transient failures from permanent failures. The technique that is described associates an address manager with each contact address stored in the location service. An **address manager** is a distributed shared object that returns an authoritative statement about the validity of a contact address. Address managers implement the locMgr interface. A persistent distributed shared object relies on its object managers to act as address managers for the object's contact addresses.

Unfortunately, the problem is not solved with a single level of indirection. An address manager is an independent distributed shared object, which can be destroyed without informing the location service. The solution is to build a hierarchy of meta-address managers. If the location service can not bind to an address manager it will ask the address manager's meta-manager whether the manager still exists. If binding to the meta-manager fails, the meta-manager's meta-manager is asked about the meta-manager. A meta-address manager reports whether an object handle still refers to a valid address manager or not. At the top, there are meta-address managers that are simply assumed to exist. The assumption is that the system administrators who are responsible for the location service are also responsible for these root address managers.

The locMgr interface contains two methods: validAddress and validManager. The first method is implemented by regular address managers. The method gets three arguments: an object handle, an address identifier and the actual set of contact addresses. The method returns whether the contact addresses are still valid or not. The second method, validManager, is implemented by meta-address managers. The arguments for this method are an object handle for the address manager that should be verified, the number of consecutive unsuccessful attempts to contact the address manager and the amount of time (in seconds) that the address manager is unreachable. The idea is that a meta-address manager generates log entries for address managers that are down for too long.

Before an address manager can be used, it has to be registered with the location service. The location service provides the locAdm interface for this purpose. The interface contains three methods: regMan, delMan, and gc. The method regMan gets two arguments: the object handle of the address manager to register and the object handle of its meta-address manager. A root address manager can be registered by invoking regMan with a NULL value for the meta-address manager argument. The delMan deletes an address manager. Finally, the gc method directs the location service to perform a garbage collection cycle.

Figure 5.18 shows the meta-address manager, which is implemented as a distributed shared object that uses the client/server approach directly (i.e., without using separate replication, control and semantics objects). One local object (lm_client) is an RPC client stub, and a second object (lm_server) contains the RPC server-stub and the implementation of the meta-address manager.

Note that the locMan interface that was described above provides only a lookup service. The interface provides no methods to register or unregister address managers. The meta-address manager provides these functions through the directory interface (naming). With this interface, the meta-address manager looks like a normal directory object where objects (address managers) can be entered, deleted, etc. As an additional service, the meta-address manager registers new address managers with the location service (i.e., it invokes the regMan method on the location server).

The lm_server object uses a generic persistence program (ps1). The persistence programs provide the server part of a distributed shared object with support for persistence, registration with the name service and the location service. The ps1 program will be described in Section 5.8.4.

5.8.3 Directory Objects

The prototype provides a single, worldwide name space for all distributed shared objects. In contrast, the name space for *local* objects is a per-process name space. The

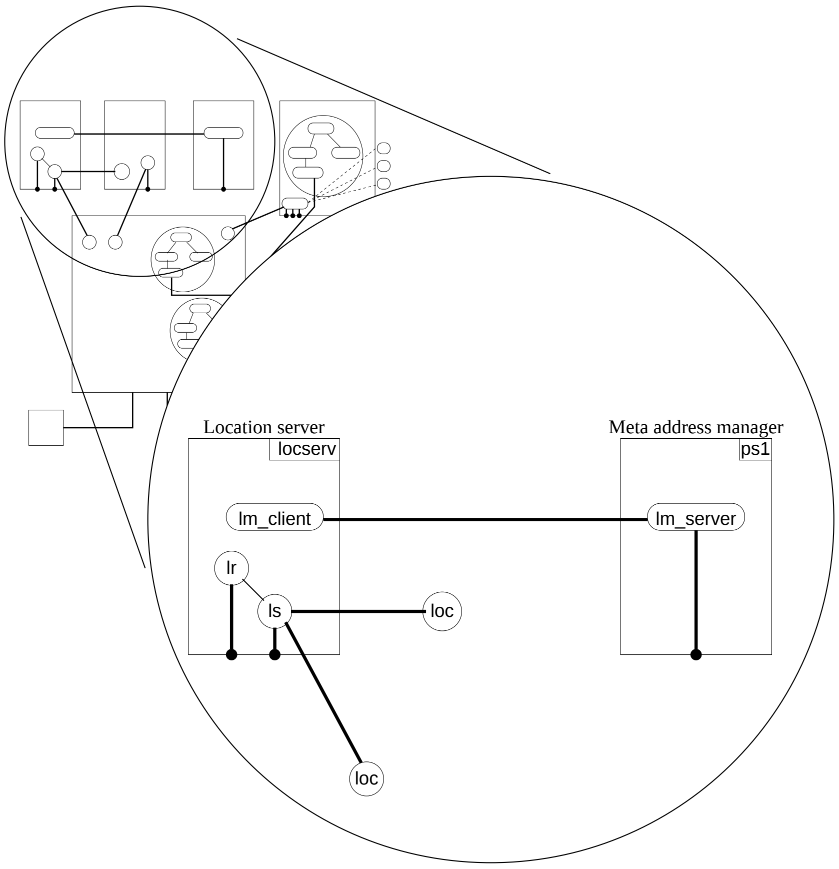


Figure 5.18. Meta address manager

worldwide name space is implemented with directory objects that contain references to other distributed shared objects. This creates a hierarchical name space based on a directed graph with a designated root object.

Directory objects provide the naming interface. This interface provides four methods: list, lookup, add, and delete. The list method returns a list of directory entries. The add method gets two arguments, a name and an object handle, and creates a directory entry. The delete method deletes a directory entry.

The lookup method is special: instead passing of a single component of a path name as a parameter, the lookup method gets the remainder of the path name. This allows the directory object to look up multiple entries in a single method invocation. The result is an object handle and the remaining part of the path name. The object handle of desired object is found when the remaining path name is the empty string.

A similar idea can be found in DNS, except that DNS always passes the entire path, instead of the remainder of the path. DNS, which has been designed for a wide-area network, provides two kinds of optimizations. Firstly, in DNS, the unit of distribution is a zone instead of a directory. A single zone can resolve multiple path name components (as opposed to only one for UNIX directories). A single lookup request on a DNS server matches all components in a single zone. The second optimization is the "recursion desired" bit. This bit, which is set by a client in a query, directs the server to recursively resolve the query instead of returning an incomplete answer when a further lookup operation on an other server is needed. A side effect of this recursive lookup operation is that the server's cache will be filled with information from other servers, which may speed up future queries. The server may choose to ignore the recursion desired bit and return a partial answer to a query (usually a set of NS delegation records). In general, DNS root servers ignore this bit because it may overload them and the cache is unlikely to be large enough to be useful.

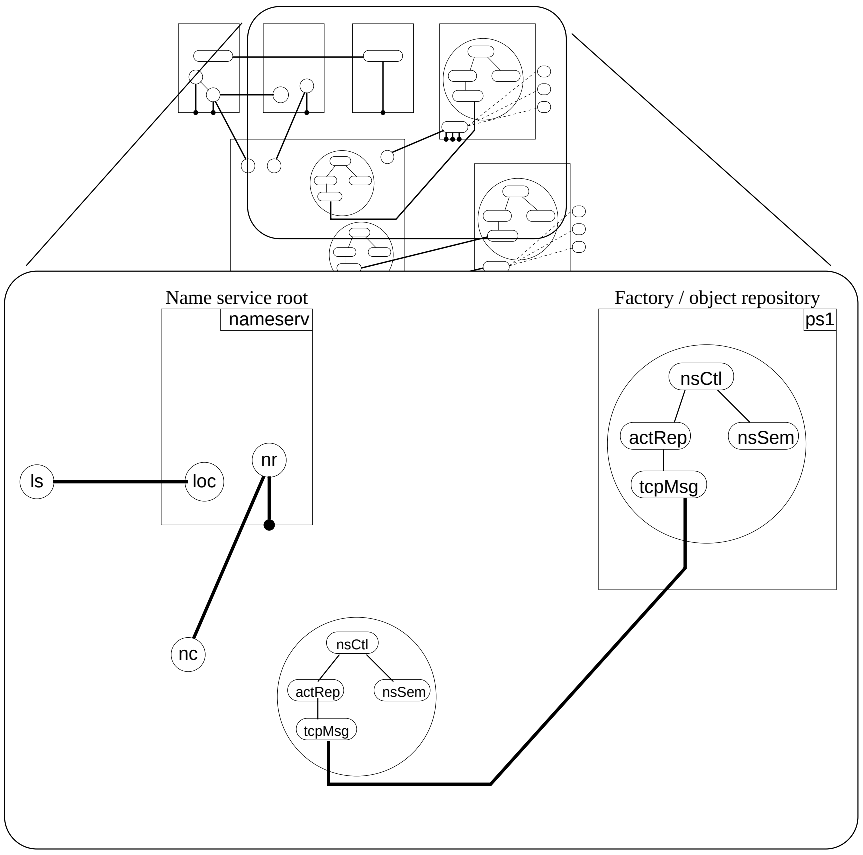


Figure 5.19. Name service

Figure 5.19 shows two implementations of directory objects. One implementation is a special root directory object and the other one is based on a semantics object that implements the naming interface.

The implementation technique used for the root directory is similar to the one used to implement the location server. The implementation consists of three parts: a local object that implements an RPC client-stub, a local object that contains the RPC server-stub and the directory implementation, and a name server program. The Globe run-time system associates this root directory object with the name /globe in the per-process local name space.

The client stub object, called nc, provides both the naming interface and the gc interface. The server object, called nr, maintains the state of the directory in a simple table. The nr object provides a special interface, called nr, for bootstrapping, which is used by the nameserv program. Persistence is implemented using the persist interface.

The second implementation of directory objects consists of three kinds of objects. There is the semantics object nsSem, the control object nsCtl, and two compositions that combine the semantics and control object with a replication and a communication object. The first composite object, nsP, uses client/server replication (csRepP) and the TCP message communication object tcpMsg. The second one, ns-act, uses active replication (actRep) and also the TCP message communication object.

The semantics object uses a simple table to implement the directory. The control object is written manually due to the lack of a control object generator.

Directory objects and garbage collection The garbage collection algorithm used for distributed shared objects requires that a distributed shared object maintains a list of directory objects that refer to the object. This means that adding an object handle to a directory also requires a method invocation on the object itself:

1. d->naming.add("name", o->distrObj.gethandle())

2. o->gc.addRef(d->distrObj.gethandle())

The first statement adds the object handle of object o to directory object d. The second statement provides object o with the object handle of the directory object. Note that the addRef operation will succeed only if the calling program sponsors the object. At the moment the prototype provides no mechanism to identify different sponsors for an object.

To rename an object o, from old-name in directory d1 to new-name in directory d2, we have to add a new link to the object and delete the old one.

1. d2->naming.add("new-name", o->distrObj.gethandle())

2. o->gc.addRef(d2->distrObj.gethandle())

3. d1->naming.delete("old-name")

Note that we perform an operation on three objects. If any of these objects is unavailable, we can not complete the rename procedure. Furthermore, the operations have to be executed in order. We do not have to inform the object about the deleted directory entry (and remove the reverse reference to the old directory object) because garbage collection will take care of that. Finally, support for multi-object transactions would be useful.

An object should be deleted as follows:

1. d->naming.delete("name")
2. o->gc.checkRef(d->distrObj.gethandle())

The second statement tells the object that it may have become unreachable. This allows the object to start a garbage collection cycle immediately. Without the second statement, an unreachable object continues to exist until its object repository asks the object to perform a garbage collection cycle.

Bootstrapping To bootstrap a Globe system, the location server is started first, followed by the server for the root directory and the meta-address manager. Bootstrapping Globe is similar to setting up a root DNS zone: in a real worldwide system it will be done only once (and only by the organization that runs the root services[6]).

During initialization, the location server creates two distributed shared objects (lr and ls). These two objects are registered internally to create object handles. The location server program writes the object handles for both objects and the contact address for the lr object to a file.

The program for the root directory server (nameserv) reads the output from the location server. The distributed shared object that implements the root directory (nr) is created and registered with the location service. Next, the object handle for the location service object ls is entered in the root directory (under the name .locServ). The last step is to create a file with the contact information for the location service (contact addresses and object handle) and the object handle of the root directory. This file is used by Globe programs to connect the location service and the root directory. This approach is similar to the named.root file used by DNS.

The next step is to start the meta-address manager. The distributed shared object is registered with the location service (using the .locServ entry created by the name service) and is entered in the root directory. Note that until now there was no address manager. All contact addresses that were registered so far are entered without the object handle of an address manager.

The next step is to start the first factories for directory objects, document objects, etc. A system that uses security services, requires additional steps to bootstrap key servers and authentication authorities.

[6]However, many sites behind a firewall also run root DNS servers.

5.8.4 Object Repositories and Factories

The prototype implements a persistent object repository that also acts as a factory. The factory creates new distributed shared objects with one persistent representative located in the factory's object repository. Additionally, the object repository can also create new persistent representatives for existing distributed shared objects.

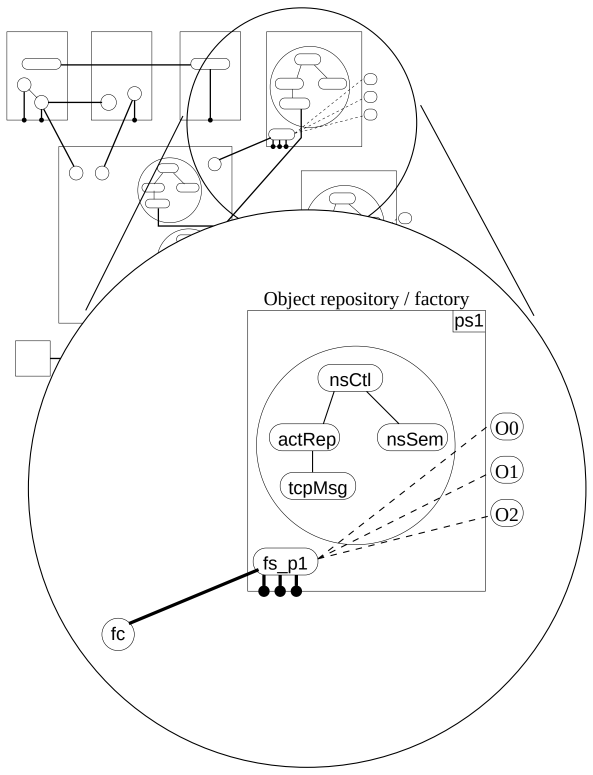


Figure 5.20. Object repository / factory

The implementation of the object repository consists of two layers. The lower layer is a program that provides persistence and registration services for a single distributed shared object. For an object repository, this distributed shared object is a factory object. The upper layer is the factory, which provides persistence and registration services for many distributed shared objects. The factory object is the server side of a distributed shared object that allows a client in a remote address space to create new distributed shared objects, to create new representatives, and to activate existing persistent representatives. This configuration is shown in Figure 5.20.

The prototype contains two implementations of the lower layer, [7] called ps1, and ps2, and two implementations of the factory object (fs_p1, and fs_p2). Of these two, only ps1 is shown in the illustrations. The other implementation differs in the persistence model it implements (see Section 5.6). The program ps1 and the object fs_p1 provide the persist interface. The persist2 interface is provided by ps2 and fs_p2.

The two factories, fs_p1 and fs_p2, share a single client object, called fc. The fc object implements the factory interface. The factory interface contains six methods: create, createRep, deleteRep, start, startObj, and gc.

The create method creates a new distributed shared object, and registers it with both the name service and the location service. The name of the new object is passed as a parameter. The createRep method creates a new representative of an existing distributed shared object. The deleteRep method deletes a representative of a distributed shared object from the object repository.

Each factory creates a single class of distributed shared objects. Before a factory can be used it needs to know what kind of objects it should create. The start method provides the factory with the class name of the (composite) local object that should be used. The class is specified by the name of local (relative to the factory) class object. Note that class names are not standardized. The caller simply has to know about the names of the classes available to a factory process.

An alternative design is to replace the start method with an extra parameter in the create method. However, this requires a standard way of referring to classes of distributed shared objects. In the current approach, a factory, when started, represents a distributed class object that creates a particular kind of distributed shared objects.

The startObj method is used to bind to a persistent distributed shared object. This method gets the object handle of the desired persistent object as an argument. The object repository activates the persistent representative (if necessary) and returns a (nonpersistent) contact address. Persistent contact addresses were described in Section 3.3.3 and Section 5.5.1. Finally, the gc method directs the object repository to perform a garbage collection cycle on its objects.

The difference between the implementation of the two persistence interfaces is that the persist interface uses a callback interface (persistCB and two methods (unmarshal in persist and newstate in persistCB) to marshal and unmarshal the state of a persistent representative. The persist2 interface uses an independent state object, which is inserted in the persistent representative's local name space.

Creating a new class of distributed shared objects

A new class of distributed shared objects is created in nine steps:

1. Create the object-specific interfaces.

[7] Actually, there is a third one called objserv. This program and corresponding factory fs do not provide persistence, and are mainly useful for debugging.

In some cases, a new distributed shared object provides new implementations of existing interfaces.

2. Implement the semantics object and generate the control object.

   In this prototype, the control object is also written by the programmer.

   For example, for a file object, two interfaces have to be implemented: the marshaling interface pickle and the file interface. The implementation of most methods in the standard object interface stdObj is generated by ifgen (See Section 5.11.1).

   In addition to the file interface, the control object provides seven other interfaces: the distrObj and distrCtrl interfaces to assist binding to a distributed shared object and the creation of new distributed shared objects and representatives; the persist and persist2 interfaces for the two persistence models; the control interface to interact with the replication object; and finally, two stream interfaces inputStream and outputStream that are used to handle the marshaled state of the semantics object.

3. Compile the semantics and control object and add the compiled objects to the class repository. Note that implementations should be made available to everybody who needs to access the new distributed shared objects.

   In this prototype, the class repository consists of a file (.globe/classes) which binds an entry in the /classes name space to the name of the class in the local file system. For example, the class objects for the file object are registered under the names do.sfSem and do.sfCtl. The class repository maps these names to the marshaled class objects on disk: /map/file/arch/sparc-sunos-5/sfSem and /map/file/arch/sparc-sunos-5/sfCtl.

4. Complete the local part of the distributed shared object by selecting replication, communication, persistence and security implementations. Create a composite object that combines those local objects. The composition has to be registered with the class repository.

   For example, /classes/globe.actRep is a replication object that implements active replication, and /classes/comm.tcpMsg is a message passing layer on top of TCP. tcpMsg itself is also a composition which consists of a message passing object /classes/comm.msg and the (built-in) TCP object /classes/comm.tcp.

   The new composition is registered under the name /classes/do.sf-act and the marshaled class object is accessed as /map/file/arch/sparc-sunos-5/sf-act.

5. Create a protocol identifier for the distributed shared object.

   In this prototype, a protocol identifier consists of a string that identifies the semantics and control objects and a string that identifies the replication object.

In some cases, there is a third string that identifies the communication object. The concatenation of those strings is the protocol identifier.

The protocol identifier for the composition, created in the previous step is: P-FILE0.0:ACT0.0. This protocol identifier is entered in the protocols file and refers to the class /classes/do.fs-act.

6. Create a factory for the new distributed shared object.

   Start either a new ps1 or ps2 process (depending on the persistence model implemented by the replication object) with the appropriate factory implementation (respectively fs_p1 and fs_p2). Provide the factory with the class name of the composition registered in step 4.

   An example is the command ps2 –m /globe/mm/sf-4 –c /classes/globe.fs_p2 /globe/sf-f4. The program ps2 and the object fs_p2 have already been described. The –c option requests the creation of a new factory (instead of restarting an existing one after a crash or reboot). The arguments after the –c option are the full name of the fs_p2 object and the name under which the factory should be registered in the name space (/globe/sf-f4). The option "–m /globe/mm/sf-4" registers the factory with the location service as a location service address manager (as described in Section 5.8.2).

   The command f_start /globe/sf-f4 /classes/do.sf-act directs factory to produce distributed shared objects whose representatives have class /classes/do.sf-act.

7. Test the distributed shared object.

8. Advertise an object-specific interfaces, a description of the protocol, and the protocol identifier.

9. Optionally, advertise the factory.

New distributed shared objects can be created by invoking g_create /globe/sf-f4 /globe/*object-name*.

Creating a new persistent distributed shared object

The creation of a new persistent distributed shared object is a complex process. Figure 5.21 shows the components that are involved in the creation of the new object and the interactions (method invocations and communication) between them. The whole process consists of 16 steps:

1. fo = p_bind("/globe/factories/www");

   Bind to the desired (distributed) factory object, in this case the factory is called /globe/factories/www and creates WWW document objects.

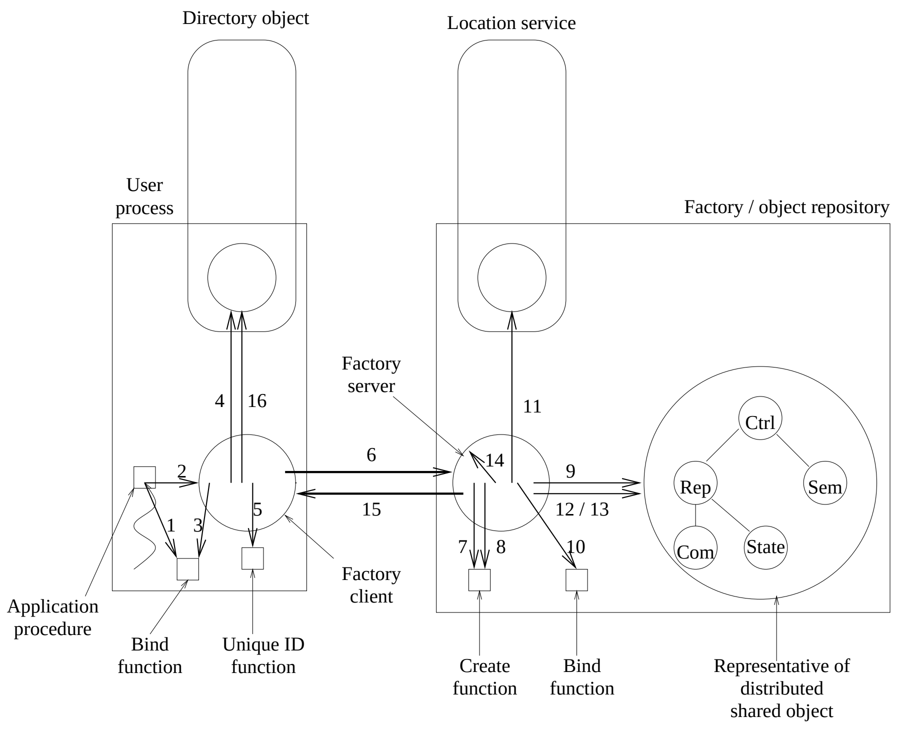


Figure 5.21. Creating a new persistent distributed shared object

2. r = fo->factory.create("/globe/cs.vu.nl/philip/www");

   Invoke the create method in the factory interface on the factory object. This is the only method that the application has to invoke to create a new distributed shared object.

The create method is executed partly in the client's address space and is partly forwarded to the address space of the factory. The execution starts in the client's address space:

3. do = p_bind("/globe/cs.vu.nl/philip");

   Bind to the directory object where the new distributed shared object should be registered.

4. h = do->distrObj.gethandle();

   Ask the directory object for its object handle.

5. oid = p_getuniq();

   Generate a new 256-bit unique object identifier.

6. Send the create request with the new object identifier oid, and the directory object's object handle h to the address space of the factory.

The execution of the create method continues in the factory's address space:

7. `do = createbyname("/classes/do.www-ms", "/main/o25");`

   Create a new local object. The name /classes/do.www-ms refers to the composition of a WWW semantics object with a master/slave replication object. The new instance is registered in the local name space under the name /main/o25

8. `so = createbyname("/builtin/classes/statefile", "/main/o25/state");`

   Create a state object in the object's composition. This step is necessary only for the second persistence model (the persist2 interface). The name of the class (/builtin/classes/statefile) refers to a class object that is linked with the program, in this case ps2. Note that the state object is directly inserted in the composite object's name space.

9. `pa = do->persist2.create(self->handle);`

   Ask the new local object to create a contact address. The contact address is based on (encapsulates) the factory's object handle, which is passed as a parameter. The call to persist2.create also tells the local object that it should become a new distributed shared object, instead of binding to an existing one. Additionally, the create method verifies the presence of a state object in its name space, and initializes the state object.

10. `lo = p_bind("/globe/locServ);`

    Bind to the location service to register the new distributed shared object.

11.
    ```
    manager = self->handle;
    handle, addr-id = lo->locserv.regObj(oid, pa, manager);
    ```

    Register the distributed shared object with the location service. The method regObj gets three parameters: the new object's object identifier oid, its first contact address pa, and the address manager for the contact address. The address manager is the same distributed shared object as the factory.

    The location service returns two values: the object handle for the new distributed shared object, and the address identifier that should be used to delete or update this contact address.

12. `do->distrCtrl.sethandle(handle);`

    Pass the object handle to the local object.

13. do->distrCtrl.newLink(dir_handle);

    Install the object handle of the directory object, that was passed in the create message, as the first reverse reference in the new distributed shared object.

14. Add the object handle and other information to the list of persistent representatives served by the object repository.

15. Send a reply message that contains the new object handle.

Back in the client's address space, the create method continues:

16. do->naming.add("www", handle);

    Finally, the new distributed shared object is registered with the name service.

The create method returns an exit status to the caller. If the caller wants to use the new distributed shared object, it should bind to the object name (in this example /globe/cs.vu.nl/philip/www).

The new distributed shared object is created in step 7 and is registered with the location service in step 11. What happens if the client process fails to execute step 16 (registration with the name service)? This leaves an unreachable distributed shared object. Fortunately, the garbage collector takes care of the object. The first time the object runs a garbage collection cycle, it will discover that it is not registered and it will be deleted.

This garbage collection cycle creates a new problem: what happens if the client process is relatively slow, or when the reply message takes a long time to reach the client, or when the client cannot communicate with the directory object? It is possible that the object is already deleted by the garbage collector before the client gets a change to register the object.

The prototype implementation does not contain a solution to this problem. Several solutions are possible: (1) the server can perform the registration with the directory object. This complicates the security architecture because the object repository has to be able to act on behalf of the client. (2) The client can create a tentative registration with the directory object based on the generated object identifier. (3) The server can tell the new object not to perform a garbage collection cycle and the server verifies that the client is still alive.

Creating a new persistent representative

In addition to the creation of completely new distributed shared objects, the factory interface also provides a createRep method to create a new persistent representative for an existing distributed shared object. The execution starts in the address space of the process that wants to create a new representative:

1. `fo = p_bind("/globe/factories/www-elsewhere");`

   Bind to the desired object repository (distributed factory object).

2. `r = fo->factory.createRep("/globe/cs.vu.nl/philip/www");`

   Create a new representative for the object created in the previous example on a different object repository.

3. The createRep method simply sends the name of the distributed shared object to the object repository.

Execution continues in the object repository's address space:

4.
   ```
   handle = name_lookup(name);
   addrs = location_lookup(handle);
   ```

   The calls to name_lookup and to location_lookup are normally performed by the p_bind library routine. However, in this case it is necessary to insert the state object in the local object's name space, which means that p_bind cannot be used.

5.
   ```
   do = createbyname("/classes/do.www-ms", "/main/o43");
   so = createbyname("/builtin/classes/statefile", "/main/o43/state");
   ```

   Create a new local object, and a new state object.

6. `do->distrObj.init(addrs);`

   Tell the new local object to bind to the existing distributed shared object.

7. `do->distrCtrl.replicate();`

   Ask the distributed shared object if the representative can become a representative that accepts bind requests from other address spaces. It is possible that the distributed shared object refuses. For example, a client/server replication object does not support contact points in more than one address space.

8. `pa = do->persist2.create(self->handle);`

   Create a new contact address based on the object handle of the object repository.

9.
   ```
   lo = p_bind("/globe/locServ-elsewhere");
   manager = self->handle;
   addr-id = lo->locserv.regAddr(handle, pa, manager);
   ```

   Bind to the location service and register the new contact address. Note that different object repositories may use different parts of the location service to register addresses.

10. Add information about the new representative to the list of objects served by this object repository.

11. Send a reply message to the client's address space.

12. The createRep method returns an exit status to the caller.

5.9 WWW Document Objects

The prototype provides three interfaces that are used by the semantics object of a WWW document object. These three interfaces are file, wwwAttr and wwwSO.

The file interface is a simple interface for storing raw data in an object. The two methods are read and write. The read method reads a block of data at a given offset in the file and the write method (over)writes part of a file. The file interface can be used for WWW objects by storing raw HTML, images, etc. directly in the object.

The simple file object sf implements this interface. It is possible to use these objects together with the ns directory objects described in Section 5.8.3 to create a simple WWW service. The directory objects provide the name space and the file objects store the contents. The suffix of the name of a file object is used by the browser to determine how to interpret the contents of the object. For example .html or .htm suggests that the object contains HTML structured text, .gif implies a GIF image, .jpeg or .jpg suggests a JPEG image, etc. This approach is similar to the way WWW documents are stored in a UNIX file system.

The use of file objects has several disadvantages. The first disadvantage is the lack of attributes. For example, it can be useful to pass caching hints to WWW proxies, to provide additional information about the encoding of the content or character set that is used. The wwwAttr interface stores the attributes of a WWW document. For example the attribute Content-type with value text/html. The methods in this interface are add to add an attribute, update to update the value of an attribute, del to delete an attribute, lookup to look up the value of an attribute and list to list all attributes with or without their values.

Sometimes, a WWW document inlines other documents (in most cases images). To improve performance, the other documents should be associated with the main document. The wwwSO interface provides storage for named subdocuments. The methods are similar to the attribute interface (add, update, del, lookup, and list). The main difference is that the values used are binary data in the wwwSO interface instead of the strings used in the wwwAttr interface.

The proper way to support these kinds of inline documents is an architecture for composite distributed shared objects. Two mechanisms are needed: support for grouping multiple semantics objects (one semantics object for each subdocument) and support for binding to those subobjects. Composite distributed shared objects are not implemented in this prototype.

The www object provides four interfaces: the three interfaces just described and the naming interface. The naming interface allows a WWW document to refer to other WWW documents using object handles. By storing object handles, the WWW document contains “hard links” which are taken into account by the garbage collector. Alternatively, a WWW document can refer to other WWW documents by storing the names of these documents (“soft links”). These soft links are ignored by the garbage collector. A single object that provides both a directory interface and a file interface removes the need for a separate index.html file which is returned by the WWW server when a WWW browser tries to access the directory. Instead, a single object can be both document and directory at the same time.

5.10 Libraries and Local Objects

In Globe, executable code can be either linked with an application or it can be dynamically loaded at runtime. Executable code can be part of a (class) object or it can be taken from a library. Combining these two gives four possibilities:

1. Built-in objects.

   These objects provide access to the underlying operating system. For example, memory allocators, a thread system, locks, I/O, low-level communication. These objects are also called system objects (see Section 2.2.5).

2. The runtime system for the application.

   This library provides the same convenience functions as the previous library but sometimes uses a different implementation. The library also provides primitives for initialization.

3. Runtime loaded classes.

   Most of the classes used in an application are expected to be loaded at runtime. Examples are high-level communication primitives, replication objects, application specific objects (semantics and control objects), and objects that provide access to standard distributed services such as the location service.

4. The runtime system for runtime loaded (class) objects.

   This library provides various convenience functions.

Additional components that are visible at compile time are interface definitions, object skeletons and compositions.

Some objects need to access to the runtime library that is linked with the application. For example, to allocate locks or to manipulate the local name space. Unfortunately, a class object that is loaded at runtime cannot access these routines directly. Instead, the runtime system contains a few built-in local objects that provide methods that correspond to functions in the runtime library. The library for dynamically loaded

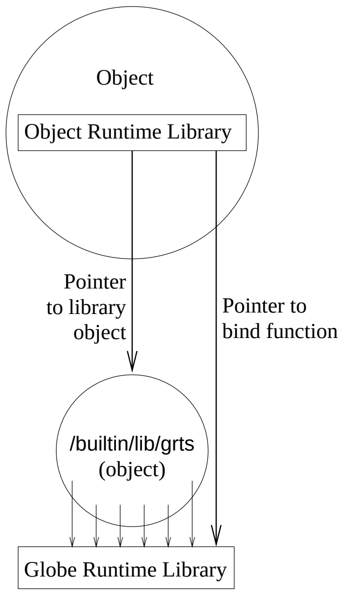


Figure 5.22. Object run-time system

objects contains "wrapper" functions that bind to these objects. This is shown in Figure 5.22. Note that these "wrapper" functions define in effect the Globe operating system interface. A pointer to the p_bind library function is passed to the initialization function of a dynamically loaded module. This pointer allows dynamically loaded objects to bind to other objects.

For historical reasons, the prototype consists of two parts, called the globe part and the paramecium[8] part. The paramecium part provides local objects, local name spaces, etc. and the globe part provides distributed shared objects, distributed services, etc.

5.10.1 Local Name Space

The runtime system maintains a local name space. Part of this name space is a tree that consists of naming contexts that refer to other contexts. At most one object can be associated with a context and every object is associated with exactly one context. Attached to this tree are objects that implement parts of the name space. This is called delegation.

[8]Paramecium uses the same local object model as is used in Globe to structure operating-system kernels and run-time systems [van Doorn et al., 1995]. The idea behind Paramecium is to explore the tradeoffs between user processes and kernel boundaries. Through the use of code signing a user process may put objects into the kernel address space. Loading modules in kernel space is efficient. On the other hand,

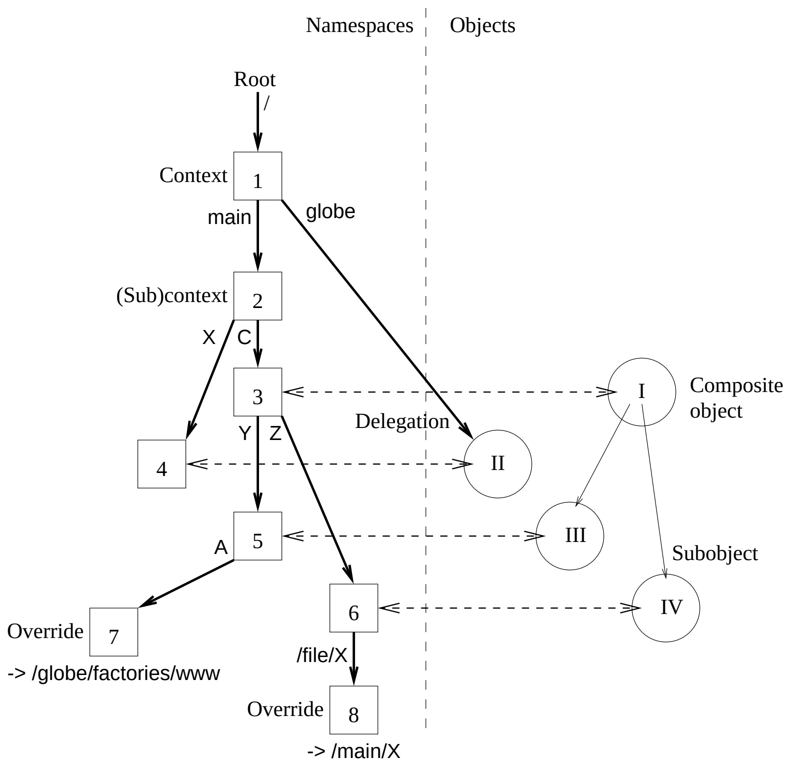


Figure 5.23. Local name space

Figure 5.23 shows a local name space. In this figure, squares represent name-space contexts and circles represent objects. Each context has a context identifier, a simple integer (1 . . . 9 in this example). Objects are numbered with Roman numerals (I . . . IV). The edges between the contexts are labeled (main, X, etc.). The figure contains one delegation: names starting with /globe are passed to the object /main/X. Context 5 contains a name-space override. In this case the override acts like a symbolic link: the name /main/C/Y/A refers to the object repository /globe/factories/www.

This name space is used to *bind* to objects, and, through delegation, also to classes in a class repository and to distributed shared objects. The resolution of a name in the name space starts in a context. For an object, this is the context associated with the object, for example, object I uses context 3, object II uses context 4, etc. The application and libraries use the context with the name /main, in this case context 2.

loading modules in user process provides better fault isolation. The object model is used to hide these details from application programs.

Function	Description
ctx_context	Create (or look up) a context
ctx_lookup	Look up a context without creating one
ctx_register	Register an object under a certain name in a context
ctx_unregister	Unregister an object
ctx_remove	Delete a context (mark a context deleted)
ctx_override	Install an override
ctx_delegate	Install a name-space delegation
ctx_lock	Lock a context (only prevents other calls to ctx_lock)
ctx_unlock	Unlock a context

Table 5.1. Name-space context functions

Two kinds of names are used: relative names and absolute names. Relative names are always interpreted relative to the starting context. By default, absolute names are relative to the root, but this can be changed through overrides. An example of such an override can be found in context 8. Context 6 contains a reference to context 8 with the name /file/X. This means that a name lookup for /file/X/... in context 6 is redirected to /main/X/.... To be practically useful, the lookup algorithm looks for overrides in all parent contexts. For example, memory is allocated using the system object /builtin/memory. Using overrides it is possible to provide an object with a different memory allocator for debugging purposes. By inserting the override /builtin/memory -> /main/X in context 6, object IV uses object II as its memory allocator. And by inserting the override in context 3, not only object I but also objects III and IV use this override.

An object that accepts a delegation has to implement the bind interface. This interface contains one method: bind. The remainder of the name is passed as an argument to the bind method and the method should return a pointer to the standard object interface of an object.

Nine library functions provide access to the local name space. These functions are listed in Table 5.1. All functions, with the exception of ctx_unregister, get a starting context identifier as the first parameter. New contexts are created by the ctx_context function. This function returns the context's identifier. The ctx_lookup function returns a context identifier for an existing context.

The function ctx_register associates an object with a context. This function is usually called right after the object is created. The function ctx_unregister deletes the association and is called just before the object is destroyed. Contexts are deleted with the ctx_remove function. A deleted context continues to exist until the associated object is unregistered.

An override consists of a name and a replacement string. Note that the name of an override can be an absolute path name. The ctx_override function adds an override to a context. Delegations are created with ctx_delegate function. A pointer to the bind interface of the object that implements the delegation is passed as a parameter.

The functions ctx_lock and ctx_unlock can be used to lock a part of the name space, for example when an object has to be registered under an absolute name, instead of relative to the current context.

The prototype contains five delegations: /file, /map, /classes, /globe, and /cache. The /file delegation provides read-only access to a small part of a local file system. These files usually contain loadable class files and configuration data. For example, binding to /file/arch/sparc-sunos-5/actRep returns a standard object interface for a file object that contains an implementation of a replication class object for a system with a SPARC processor running Solaris.

The /map delegation creates a new object that is initialized with marshaled state stored in another object. The name space implemented by /map is identical to the name space provided by the root. For example, the name /map/file/arch/sparc-sunos-5/actRep returns a new class object that is created from the state stored in /file/arch/sparc-sunos-5/actRep. For /map to work, it is necessary that the marshaled state of an object starts with the class name of the object. A meta-class, called class_so0, is used to load classes dynamically. An instance of this metaclass implements the unmarshal method in the pickle interface. The instance of the meta-class becomes a new (class) object after a successful invocation of this method. Note that the meta-class can be used to load any kind of executable code, not just class objects.

The /classes delegation provides a uniform naming scheme for class objects. For example, /classes/comm.tcp maps to /builtin/classes/tcp and /classes/comm.tcpMsg maps to /map/file/arch/sparc-sunos-5/tcpMsg. The users of those two class object do not have to know that the first class object is linked with the application, and that the second one is loaded a runtime. Note that /classes uses the /map and /file delegations to load a class object from the local file system. In the current implementation, the /classes delegation simply uses a file that contains the mapping between class names and their locations. This file is read when an application is started.

The name space that starts with /globe is delegated to an object that binds to distributed shared objects. Figure 5.24 shows the various functions, modules, objects, and files that are used to bind to a distributed shared object. The implementation of the bind algorithm is distributed over three functions: globe_bind, globe_bind_handle, and globe_bind_addr. All three functions get a context identifier as a parameter. This is the context where a newly created local object should be registered.

The function globe_bind uses the local name resolver to look up an object handle for the name that was passed as an argument. The name is relative to the root of the worldwide name space. The resolver invokes methods on various directory objects.

The function globe_bind_handle is called by globe_bind with an object name and the object handle as parameters. The location service resolver is called to look up a

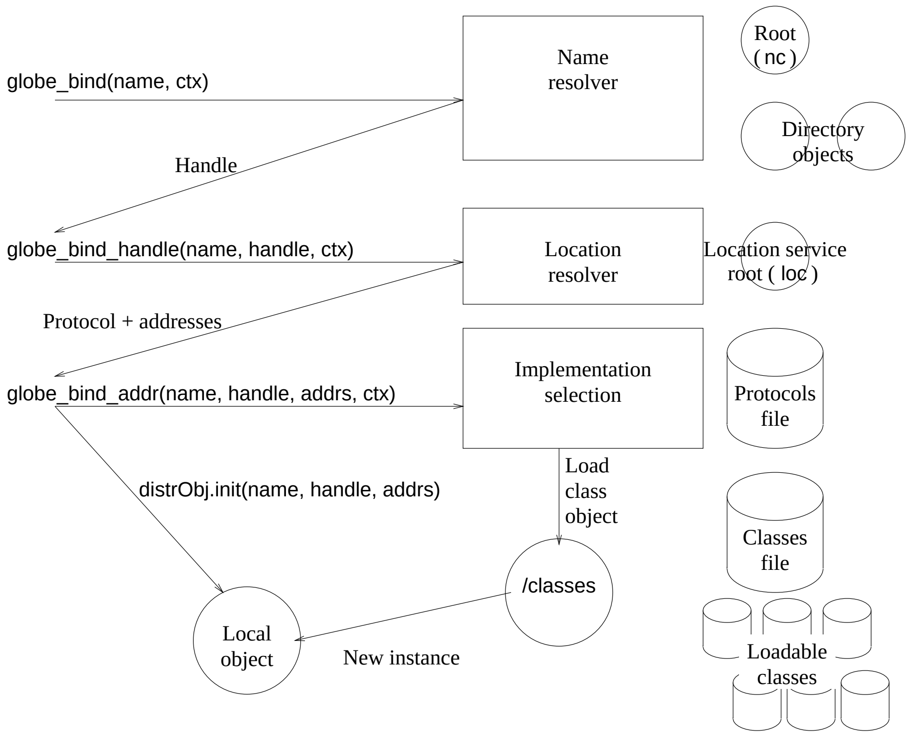


Figure 5.24. Binding

set of contact addresses for the object handle. The location service is accessed as a single distributed shared object.

The function globe_bind_addr is called with the name, object handle, and contact addresses of an object (the name is provided for debugging purposes). First, the protocol is extracted from the first contact address and is looked up in a file with protocol identifiers to find the name of a class object that implements the protocol. Note that destination selection is not implemented in this prototype. For example, the protocol MUTEX0.0:P-MS0.0:MSG:TCP is implemented by the (composite object) class /classes/do.dl-ms0. The /classes delegation uses a class object file to find the actual implementation. The class is loaded and an instance of the class is created. The last step is to invoke the init method in the distrObj interface with the name, object handle, and contact address. The new instance sets up a connection to the distributed shared object to complete the binding procedure.

The reason that these functions are separately available is that sometimes the name lookup and the location lookup can be skipped. Binding to the location service has to be done without calling globe_bind_handle, and the name resolver calls globe_bind_handle directly to bind to a directory object.

Figure 5.24 omits one aspect of binding to distributed shared objects. A process may bind multiple times to a single distributed shared object. For example, a single directory object may be used in successive name-lookup operations. This problem can be solved by caching local objects that are connected to distributed shared objects. This is implemented by the delegation of /cache to an object that caches bindings to distributed shared objects.

The function globe_bind_handle tries to bind to /cache/oid/*object-identifier* before calling the location service to look up contact addresses. Later, the function globe_bind_addr tries to use protocol-specific caches by binding to /cache/proto/*protocol-identifier/object-identifier*. Currently, no protocol-specific caching is implemented. Finally, the completely initialized object is registered in /cache/oid/. One complicated problem is how to decide when a cached object should be released. This prototype caches only a small number of objects to avoid dedicating too many resources to cached objects. A simple second-chance algorithm is used to delete objects from the cache when the cache is full.

The main bind function in the library is p_bind. This function uses contexts and delegations to bind to existing local objects, to load class objects and to bind to distributed shared objects. The p_bind function returns a pointer to the standard object interface of an object. This function gets two parameters: the name of the object and the starting context for name-lookup operations. Most of these context functions (including p_bind) are also provided by the /builtin/ctx object that exports the ctx interface. Wrapper functions in the library use this ctx object to provide the context functions to dynamically loaded class objects.

5.10.2 Memory Allocation

Table 5.2 lists the main memory allocation functions. These functions can be divided into three groups: functions that operate on simple segments of memory (p_alloc, p_free), functions that operate on collections of memory segments (used for network packets), and functions to increment and decrement reference counts on both packets and segments.

As described in Section 2.2.5, memory segments consist of a small header followed by an array of bytes. This header contains the function pointer realloc, which is used to grow, shrink, and delete memory segments, the size of the segment, and a dummy word for alignment reasons. The link field is a reference count, which is used to do garbage collection on dynamically allocated data. A network packet consists of a linked list of packet data structures that point to memory segments. Figure 5.25 shows the fields of the memory segments and the network packet headers.

The p_alloc function allocates a segment of memory and returns a pointer to the first byte past the header. The size of the segment is passed as a parameter. The p_realloc function expands or shrinks a previously allocated segment. It is possible

Function	Description
p_alloc	Allocate a segment of memory
p_realloc	Resize a previously allocated segment
p_free	Decrement a reference count and free a segment
p_strdup	Duplicate a string in a segment
pkt_alloc	Allocate (part of) a network packet
pkt_free	Decrement reference count and free a packet
pkt_totlen	Amount of user data in packet
pkt_cpout	Copy data from a packet to a buffer
pkt_merge	Concatenate two packets
pkt_delhead	Delete first part of a packet
pkt_trunc	Truncate packet
pkt_shallow_dup	Duplicate access data structures but not user data structures
a_incref	Increment a reference count
a_decref	Decrement a reference count without freeing

Table 5.2. Memory allocation routines

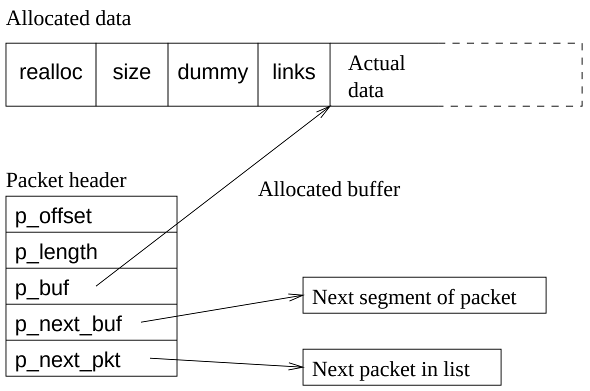


Figure 5.25. Memory segment and network packet data structures

that a new segment has to be allocated. In that case, p_realloc copies the old data to the new segment and frees the old segment. There are two reasons for p_realloc to allocate a new segment. First, the usual reason is that the segment has no room to grow at its current location. Second, a segment with a reference count larger than one is not resized, to avoid interfering with other users of the segment. Segments with a reference count larger than one are assumed to be immutable. Assuming that shared memory segments are immutable simplifies concurrency control and allows references to be handed out freely to other modules and objects.

The p_free function decrements the reference count and deallocates the segment when the reference count becomes zero. The last function, p_strdup is a convenience function to copy a string to a segment of memory.

Note that the functions p_free and p_realloc do nothing more than invoking the function in the realloc field in the header of a segment. This approach allows the use of multiple memory allocators in a single system. The segment's realloc function is called with two parameters: a pointer to the segment, and the desired size. A size of zero is used to free the segment (decrement the reference count in the links field). The function returns a new pointer to the segment. The size field in the header is used by the realloc function to determine the amount of data that has to be copied.

There are two implementations of p_alloc, one for the library that is linked with an application and the other one for the library that is linked with a loadable class object. The first p_alloc implementation uses the malloc function in the C library. The second implementation is a wrapper for a memory allocation object /builtin/memory. This memory allocation object provides the memory interface and is linked with the application. The memory interface contains only one method: alloc. The memory allocation object implements this method by calling p_alloc.

As described in Section 5.3, a dynamically-loaded class object can call functions in the run-time system directly. Only a pointer to the bind function is passed during the initialization of the class object. The advantage of accessing the malloc function through an object is the ability to assign different memory allocators to different objects, for example, to limit the total amount of memory that can be allocated by an object. This feature is not used in the prototype implementation.

Network packets are higher-level data structures based on dynamically allocated memory segments. A network packet stores its data in one or more memory segments. These segments use reference counts and can be shared between multiple network packets. A special packet header is used to refer to those memory segments. In addition to a pointer to a memory segment, a packet header also contains a p_offset and a p_length field and two pointers (p_next_buf and p_next_pkt) to other packet headers. The offset and length fields specify what part of the memory segment is addressed. Those two fields allow protocol headers to be deleted and packets to be truncated without copying any data.

The p_next_buf pointer is used to create a linked list of packet headers. The data in the linked list is treated as a single network packet. The p_next_pkt pointer can be used to create a linked list of network packets. For example, to create a send or receive queue.

The function pkt_alloc calls p_alloc to allocate a new packet header and sets the header fields to zero. The pkt_free function decrements the reference count of a packet header. When the reference becomes zero, pkt_free calls p_free to free the header after freeing p_buf, p_next_buf and p_next_pkt. The function pkt_totlen returns the amount of user data in a network packet (the list of packet headers) and pkt_cpout copies a specified amount of data, starting at a specified offset, to a buffer. Note that this may involve copying data from multiple memory segments.

The function pkt_merge concatenates two packets. This function does not modify a packet header with a reference count larger than one. Usually, this means that a collection of new packet headers has to be allocated for the merged packet. The pointer to the first packet header is returned.

The functions pkt_delhead and pkt_trunc delete some amount of data from the head and the tail of a packet, respectively. New packet headers are allocated when the packet is shared. The function pkt_shallow_dup can be used to guarantee that all packet headers have a reference count of one.

5.10.3 Threads

Threads are represented by objects. A thread object is an instance of the class object /builtin/classes/thread. Thread objects provide the thread interface. This interface contains four methods: start, join, detach and cancel.

The user of a thread object is expected to implement the callback interface threadCB. This interface contains only one method called start which is used to start the execution of user defined code in a thread.

The user of a thread object invokes the start method in the thread interface and passes two parameters: a pointer to the user's threadCB interface and callback reference (usually a small integer). The thread object creates a new thread and invokes the start method in the threadCB interface from this new thread. The callback reference is passed as a parameter.

The use of a callback interface to start a thread simplifies some reference counting issues. An object which has become unreachable but is still executing in a thread will not be destroyed until the start method in the threadCB returns. The thread object releases the threadCB interface and this destroys the object.

A disadvantage of this approach is that only objects can create new threads. An application that needs additional threads has to create a small object to provide the threadCB interface.

The join method in the thread interface blocks until the thread is completed (i.e., the start method returned). The detach method breaks the connection between the thread object and the actual thread. The thread will continue but it is no longer possible to join or cancel the thread. The same effect can be obtained by releasing all references to the thread object. Finally, the cancel method kills the thread. Additional support for safely terminating a thread is required but not currently implemented.

The implementation of thread objects in the prototype consists of a small layer on top of the Solaris thread library. The start method in the thread interface creates a new thread which starts executing a function of the thread object implementation. This function invokes the start method in the threadCB interface. After the method invocation returns, the function releases the interface, a flag is set in the state of the thread object to indicate that the thread is terminated and any threads executing a join method are signaled.

5.10.4 Persistence

A collection of objects and interfaces provides access to local persistent storage. Typically, this storage is the file system of the underlying operating system. Two models are used to access the persistent state of those objects: a stream model where data is processed as a continuous byte stream and a file model where data can be read and written at arbitrary positions.

The stream model uses three interfaces. The inputStream interface provides read access to an object, the outputStream interface provides write access, and the openStream interface directs a file object to open a file in the file system in read or write mode. Some file objects provide the openStream, but other objects use only the inputStream and outputStream to implement marshaling and persistence.

The inputStream interface provides two methods: readbuf and readall. The readbuf method allocates a buffer, reads data from a stream into the buffer and returns the buffer and the amount of data read. The caller is responsible for freeing the buffer. The readall method tries to fill a user supplied buffer. The outputStream interface provides a single write method which writes the contents of a buffer to the stream. openStream opens a (local) file for input or output. The only method is open. Parameters to open are the name of the file and a C stdio-style mode ("r", "w", etc).

The stream interface is used by several objects. The /file delegation was described in Section 5.10.1. The implementation of this delegation uses the file class object called /builtin/classes/file. These file objects encapsulate the fopen, fclose, fread and fwrite in the C library. The stream interface is also used in the pickle interface to marshal and unmarshal the state of the semantics object, to load a new class object, and in the persist interface.

The file model uses two interfaces: a state interface to read and write data, and an openState interface to open a particular file or to create a new one. The openState interface provides two methods: open and create, which respectively, open existing files and create new files. The name of the file is passed as an argument. Files are always opened in read/write mode.

The state interface contains five methods: write, read, truncate, mark, and sync. The write method write a block of data, the read method allocates a buffer, reads a block of data up to a specified maximum size, and returns the buffer and the actual size.

The truncate method truncates the file to a specified size. Two methods, mark and sync provide support for atomic operations on the file. Invoking the mark method indicates that the current state is a consistent state. The sync method forces all changes to be made persistent. The idea is that after a crash, the system rolls back to either the state of the last invocation of the sync method, or the state of a later mark method invocation. The mark and sync methods are not implemented in the prototype.

The instances of the class object /builtin/classes/statefile implement the state and openState interfaces. As described above, in Section 5.6, the persist2 interface requires the presence of a state object in the persistent object's local name space.

5.10.5 Communication

Six interfaces are used for communication: the comm_p2p interface and associated callback interface commConnCB set up point-to-point connections, the comm interface sends data that is received with the callback interface commRdCB. Support for multicast group communication is provided by the commGroup interface.

The sixth interface is the commAddrs interface, which provides a standard way to convert between address identifiers and ASCII strings. An **address identifier** is an integer that represents an address in a communication protocol, for example, in the TCP/IP protocol, an address consists of an IP address and a TCP port number. The use of address identifiers abstracts from the representations of addresses in different communication protocols. The commAddrs interface contains two methods: lookup and install. The lookup method returns the ASCII string that corresponds to an address identifier. For example, a lookup of address identifier 5 on a TCP communication object may return TCP(192.31.231.174/63468). A communication object allocates address identifiers for addresses stored in its tables. The install method is the opposite of lookup: an ASCII string is installed in the tables of a communication object. An address identifier for the address is returned.

The comm interface contains four methods: send to send a packet, sendrec to send a packet and wait for a reply, setCallBack to install a receive callback interface, and reply to reply to a previously received request. The send method gets a packet and a destination address identifier as arguments. The packet is a pointer to the packet data structure that was described in Section 5.10.2. The sendrec method is similar to the send method except that the method waits until it receives a reply. The reply packet is returned.

The setCallBack method installs a pointer a commRdCB interface. The second parameter is a callback reference value. The commRdCB interface is implemented by the user of a communication object. Received packets, received requests and receive errors reported by invoking one of three methods in this interface. The msg_arrived method delivers a packet that is not a request packet. This method is invoked with three arguments: the callback reference value, an address identifier for the source address, and a packet.

The req_arrived method delivers a request packet. The communication object expects the method to return a reply packet. It is possible that a reply packet cannot be returned immediately. In that case, the method should return a NULL pointer and invoke the reply method in the comm interface at a later time. The req_arrived method gets an additional callback reference from the communication object which should be used in the invocation of the reply method. The error method reports a failure in the communication object to receive more data.

The comm_p2p interface contains four methods: bind to bind to a particular local address (or port), listen to accept incoming connection requests, connect to setup a connection, and setCallBack to register a commConnCB interface. A communication object can setup a point-to-point connection actively connecting to a remote communication object. Alternatively, the communication can wait until a remote connection requests arrives.

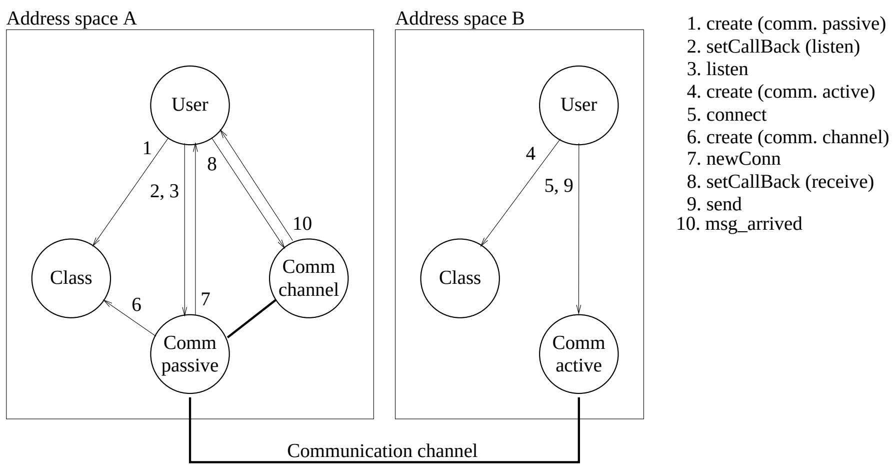


Figure 5.26. Communication setup

In order to setup a connection actively, the user creates a new instance of a communication object class and invokes the connect method. Passively waiting until a connection request arrives requires the registration of a callback interface. The communication object creates a new communication object when the connection is established. Figure 5.26 shows the objects and method invocations to setup a connection and to send some data.

First, the user of the communication object in address space A creates a new communication object (“comm passive”), registers the callback interface and invokes listen (steps 1, 2, and 3). Usually, the address that is returned by listen, is converted to an ASCII string and is advertised. This is not shown. At a later time, an object in address space B decides to connect to the object in address space A. The object creates a new communication object and invokes the connect method (steps 4 and 5).

The communication object in address space A receives the connection request and creates a new communication object, which is called a channel object (step 6). This channel object establishes the connection. The passive communication object invokes the newConn method on the callback interface and passes a pointer to the channel object as a parameter (step 7). The user object registers its receive callback interface with the channel object (step 8).

After the connect method returns, the user object in address space B invokes the send method to send some data (step 9). The data is sent over the network to the channel object. In many implementations, the passive communication object implements the demultiplexing of data for different channel objects. The channel object invokes the msg_received method on the receive callback interface to deliver the data (step 10).

The commGroup contains three methods. The create method creates a new multicast group. An address identifier for the group is returned. The join method allows a communication object join an existing group, and the leave method directs the object to leave the group. The commGroup interface is not implemented and the provided functionality is probably incomplete. For example, there is no method to obtain a list of group members, and no indication that a new group member joined or that an existing group member left the group. For scalability reasons some group communication protocols do not support this kind of functionality. It is not clear whether these methods should be added to the commGroup interface (with the option of leaving them unimplemented by some object) or whether a separate interface should be created.

The prototype provides two communication objects, the first one is built-in (/builtin /classes/tcp) and provides access to TCP sockets. The second one (/classes/tcpMsg) is loaded at run-time and provides a reliable message passing layer on top of the built-in TCP data streams.

The TCP communication object is the main source of pop-up threads. To implement the listen method, the communication object creates a new thread that repeatedly creates a new channel object, calls the accept system call, and invokes the newConn method. Similarly, the registration of a receive callback interface results in the creation of a new thread. This “receive” thread calls the read system call and delivers the result by invoking error on a read error, or req_arrived with actual data.

The TCP communication object provides a byte stream. However, the interface to the communication object suggests message passing. The TCP object treats the packet that is passed as parameter to send, sendrec, and reply as a sequence of bytes. There are two primitives that send data (send and sendrec) and also two primitives that deliver data (msg_arrived and req_arrived). The most straightforward approach is the use of send to send data and msg_arrived to deliver data. Note that the TCP protocol does not preserve packet boundaries: a sequence of byte passed to a send method may be delivered in multiple invocations of msg_arrived and multiple send invocations may result in a single invocation of msg_arrived.

Often, the user of TCP object needs to send data in response to the received data. Unfortunately, the locking model described in Section 2.2.5 does not allow the user to invoke the send method in the same (pop-up) thread that invoked the callback method msg_arrived. The user is forced to use another thread the invoke the send method. This inconvenience can be avoided by using the req_arrived method instead. If the user needs to send reply, he can simply return the reply packet as the result of the req_arrived method. The TCP object sends the reply to its peer. A NULL pointer should be returned if no reply is to be sent.

A similar argument can be made for the use of sendrec instead of send. However, in this prototype the sendrec method did not prove to be necessary and is therefore not implemented.

The tcpMsg object is a composition that contains both a TCP object and a msg object. The msg object converts a byte stream, such as provided by the TCP object, into a stream of messages. The msg object provides both the send method to send a message, and the sendrec method to send a request and wait for the reply.

To implement message passing, the msg object prepends a small header to the message that contains the length of the message and a message identifier. The message length is used to split the reliable byte stream offered by TCP into a reliable message stream. The message identifier is used for two purposes: to distinguish three different types of messages and to assign sequence numbers to messages.

The three different kinds of messages are requests messages, reply messages and normal point-to-point messages. Request messages are sent using the sendrec method, reply messages are returned by the callback method req_arrived or sent using the reply method, and normal point-to-point messages are sent using the send method. The encoding of the message identifier is as follows. The identifier is an unsigned 32-bit integer. The value zero is used for point-to-point messages. The most significant bit (MSB) is used to distinguish request and reply messages. Request messages use identifiers with values between zero and 2^{31} and reply messages used to identifier of the corresponding request message with the MSB set (in effect adding 2^{31} to the identifier in the request message). This way, multiple threads can invoke the sendrec method simultaneously on a single msg object. The request will be sent with different identifiers. Later the user of the remote msg object may send replies to the requests in a different order than they were received.

The msg object implements the methods send and sendrec, and the callback method req_arrived. The send method simply prepends a header with the length of message and identifier zero, locks a mutex to avoid race conditions, and invoke the send method on the underlying communication object (the TCP object in this prototype). The sendrec method is slightly more complex. A header with the length of the message and the next request identifier is prepended to message and sent using the underlying communication object. After sending the request, the method puts a

Function	**Description**
lock_alloc	Allocate a lock
lock_free	Deallocate a lock
lock_enter	Enter a critical region
lock_leave	Leave a critical region
lock_wait	Wait for a condition to become true
lock_signal	Wakeup at least one thread to re-evaluate its condition
lock_broadcast	Wakeup at all waiting threads
lock_once	Call an initialization function exactly once
globallock_enter	Lock a single, global lock per address space
globallock_leave	Unlock the global lock

Table 5.3. Locking functions

small data structure that contains the request identifier and a place holder for the reply message on a list and blocks on a condition variable to wait for the reply.

Most of the work done by the msg object is part of the callback method req_arrived. The implementation of this method collects bytes from the TCP object and assembles complete messages. A normal point-to-point message is passed directly to the user by invoking the msg_arrived method. For a reply message, the object locates the data structure in the list of blocked sendrec method that contains the identifier of the corresponding request message. A reference to the reply message is store in the data structure. The current implementation wakes up all threads that are blocked in a sendrec method invocation on the current object; the thread that finds the reply message continues.

A request message is passed to req_arrived and any reply is returned to the TCP object after prepending a header. The message identifier in the request message is passed as a reference in the req_arrived method invocation. The user of the msg object passes this reference to the reply method (if the reply method is invoked at all, the alternative is to return the reply message as the result of the req_arrived method). This allows the reply method to put the right reply identifier the message header.

5.10.6 Synchronization

Table 5.3 lists the library functions that control access to critical sections and provide thread synchronization. The function lock_alloc allocates a new lock and returns a lock identifier. The lock is freed by the function lock_free. The functions lock_enter and lock_leave respectively, lock and unlock the lock specified by a lock identifier passed as parameter.

Each lock also contains a condition variable. The condition variable can be used in a critical section that is protected by the lock (after calling lock_enter). The function lock_wait atomically releases the lock (equivalent to calling lock_leave) and waits for a signal or a broadcast. The functions lock_signal and lock_broadcast, respectively, wake up at least one or all threads blocked on this condition variable. A thread that gets a wakeup signal in a lock_wait function executes the equivalent of lock_enter before returning to the caller. The order in which threads are woken up after a lock_signal or lock_broadcast is undefined.

The function lock_once can be used to execute a function exactly once. This is useful when multiple threads try to initialize the same module simultaneously. The function lock_once gets a function pointer and a pointer to a (global) variable of type once_t as parameters.

The functions globallock_enter and globallock_leave lock and unlock a single lock for the whole process. The normal lock primitives require a lock identifier, which is allocated by lock_alloc. For each critical section, only one thread is allowed to call lock_alloc. In some cases, for example during the initialization of certain parts of the run-time system, it is necessary to use a single, global lock to ensure that only one thread calls lock_alloc. The functions globallock_enter and globallock_leave can also be used when more fine-grained locking is unnecessary.

A special interface called libprts, which is implemented by an object called /builtin/lib/prts, also provides access to these functions.[9] This object allows dynamically loaded (class) objects, which cannot call the functions in the run-time system directly, to use these functions. Access to this object is provided by a set of wrapper functions which have the same name and function prototypes as the real functions listed in Table 5.3.

Unfortunately, the extra overhead that results from the these extra layers may be too large for fine-grained locking. A lock_enter call from a dynamically loaded class involves a function call, a method invocation, a function call, and another function call. These lock functions should be augmented with atomic memory operations and spin-locks to obtain decent performance. However, to use of atomic memory operations and spin-locks reduces the modularity of the system: a lock is no longer an abstract data structure identified by a lock identifier but is partly implemented in-line in the callers function.

5.10.7 Unique identifiers

Section 2.2.1 introduced 256-bit unique identifiers to identify interfaces. These identifiers are also used to uniquely identify distributed shared objects. The library function p_getuniq generates such a unique identifier.

Table 5.4 shows the general format for a unique identifier. The first four bytes specify a particular numbering authority and the remaining 28 bytes are specific to

[9]As described in Section 5.10, the part of the prototype that provides local objects, local name spaces, etc. is called paramecium. The name prts stands for paramecium run-time system.

Bytes	Value
00 . . . 03	Numbering authority
04 . . . 31	Data

Table 5.4. General format for allocating unique identifiers

Bytes	Value
00 . . . 03	Numbering authority (0x39, 0x69, 0, 0 for this format)
04 . . . 07	Process ID
08 . . . 11	Seconds
12 . . . 15	Microseconds
16 . . . 31	IPv6 address, or right justified, zero-padded IPv4 address

Table 5.5. UNIX/Internet unique identifier format

that numbering authority. The prototype uses an allocation scheme based on Internet addresses and UNIX process identifiers. This is shown in Table 5.5. The identifier for this scheme is the magic number 0x39690000 (on a big-endian system). The next 12 bytes contain a four-byte process identifier (pid), the four-byte UNIX time (in seconds since 1970), and a four-byte counter for microseconds. The last 16 bytes are either a zero-padded Internet address, or a 16-byte next generation (version 6) IP address.

After the first call to p_getuniq in a process, it is no longer necessary to use the current time. Instead, it is sufficient to simply increment the microsecond field and increment the second field when the microsecond field overflows.

The current implementation of p_getuniq naively assumes that Internet addresses are unique. This is not always true for machines behind a firewall or a network address translation unit. Other numbering authorities, for example based on telephone numbers, are needed to deal with these cases.

Privacy is another issue. The current implementation identifies the creator of an object or an interface. A scheme that uses a hardware random number generator to generate 200 bits of random data can solve this problem. Unfortunately, most current computers do not have hardware random number generators. Some operating systems use unpredictable events, such as key presses on a keyboard, or hard disk seek latencies, to generate random numbers that might be suitable for this purpose. Note that with 200-bit random numbers, after allocating in the order of 2^{100} identifiers, the chance that a single number has been used twice is about 50%.[10]

The functions p_uniq2str, and p_str2uniq convert unique identifiers to and from strings. The libprts interface provided by the object /builtin/lib/prts contains the method getuniq to support dynamically loaded objects.

[10] 2^{100} is a bit more than 10^{30}. If 10^{10} processes allocate 10^9 identifiers per second, it takes 4000 years to allocate 2^{100} identifiers.

5.11 Programs

5.11.1 Interface and Object Generator

The interface and object generator, ifgen, generates C header files for interface definitions, templates for (class) objects implementations, and compositors for compositions. The syntax for interface and object definitions is an extended version of the C declaration syntax. A grammar can be found in Appendix A.

```
@#ifndef P__INF__STDOBJ_INF
@#define P__INF__STDOBJ_INF
@
@struct eor;

interface stdObj
{
        int init(ctx_t ctx, error_p err);
        void cleanup(int how);
        REF struct noInf *getinf(uniqid_p infid, error_p err);
        REF interface class relinf(REF void *infp);
} = 3;

@typedef struct infName
@{
@        char *infName;
@        uniqid_p infId;
@} infName_t, *infName_p;

interface objMgt
{
        infName_p getInfList(void);
} =      "39690000bb45006935093566000ebb46"
         "000000000000000000000000c01fe7ae";

@#endif /* P__INF__STDOBJ_INF */
```

Figure 5.27. Standard object interface

Interface Definition Figure 5.27 shows the file <p/inf/stdobj.inf> and Figure 5.28 shows the generated C definitions for the objMgt interface. The lines that start with the at symbol ("@") are called line literals and are copied by ifgen directly (without the at symbol) to the output file. The macro DEFINE_INTERFACE_ID is defined in the library to generate a collection of variables that contain interface identifiers. Each interface is a structure that consists of a pointer to the standard object interface followed by pairs of function pointers and state pointers.

The macro GETINF_objMgtInf can be used on every interface to request a pointer to the ojbMgt interface. This macro uses the pointer to the standard object interface. For each method, a macro is generated that invokes the method. In this example, there is

```
extern uniqid_t objMgtInf_id;
#ifdef DEFINE_INTERFACE_ID
uniqid_t objMgtInf_id=
{ {
      0x39, 0x69, 0x00, 0x00, 0xbb, 0x45, 0x00, 0x69,
      0x35, 0x09, 0x35, 0x66, 0x00, 0x0e, 0xbb, 0x46,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0xc0, 0x1f, 0xe7, 0xae,
} };
#endif /* DEFINE_INTERFACE_ID */

typedef struct objMgtInf
{
      struct stdObjInf *soi;

#define GETINF_objMgtInf(_i_, _err_) ((objMgtInf_p) \
      (_i_)->soi->getinf_m((_i_)->soi->getinf_o, \
            &objMgtInf_id, (_err_)))

      infName_p (*getInfList_m)(object_p);
      object_p getInfList_o;

#define OBJMNG_getInfList(_i_) \
      ((_i_)->getInfList_m((_i_)->getInfList_o))

} objMgtInf_t, *objMgtInf_p;
```

Figure 5.28. C definitions for objMgt interface

only one method (getInfList), and therefore just one macro (OBJMNG_getInfList). The name of the interface is encoded in the name of the macro to avoid duplicate macro names in C programs that use multiple interfaces. At run-time, interface identifiers are used to distinguish different interfaces.

Object Implementation Figure 5.29 shows an example of an object description for a control object. An object description consists of a list of interfaces. These interfaces can be normal interfaces (the default), or callback interfaces (specified by the callback keyword).

The stdObj interface has to be provided by all objects. The interfaces distrObj, distrCtrl, persist, persist2, control, inputStream, and outputStream are provided by most control objects. The mutex interface is specific to this control object (dl stands for distributed lock, see Section 5.5.3). The callback interface threadCB is used by control objects that have to retry blocking requests.

It is possible to generate four different skeletons for a single object description. Which version is generated depends on two flags. The –o flag selects whether the skeleton should be for a class object or a single instance of a class. Normally, a class

```
#include <p/inf/mutex.inf>
#include <p/inf/stdobj.inf>
#include <p/inf/stream.inf>
#include <p/inf/thread.inf>
#include <globe/inf/control.inf>
#include <globe/inf/distr.inf>
#include <globe/inf/distrCtrl.inf>
#include <globe/inf/persist.inf>
#include <globe/inf/persist2.inf>

object dlCtl
{
        /* list of normal interfaces provided by this class */
        interface stdObj;
        interface distrObj;
        interface distrCtrl;
        interface persist;
        interface persist2;
        interface mutex;

        /* list of callback interfaces */
        callback interface control;
        callback interface inputStream;
        callback interface outputStream;
        callback interface threadCB;
};
```

Figure 5.29. Control object description for a distributed lock

object is generated that creates instances of the type specified in the object description. With the –o flag, code is generated for just a single instance.

The second flag (–r) selects whether the code is intended to be loaded dynamically, or is part of the run-time system and has to be linked with the application. The default is dynamic linking, and the –r flag selects statically linked implementation.

Composite Object Implementation Figure 5.30 shows the definition of a composite object. The specification of a composite object consists of four parts. The first part lists the interfaces that are provided by the composite object. For example, “interface mutex;” specifies that the composite object provides the mutex interface.

The second part contains a list of subobjects, and for each subobject the way to obtain a pointer to its standard object interface. Currently only the creation of new instances is supported. The line “object tcpMsg = new "/classes/comm.tcpMsg";” defines a subobject called tcpMsg which is created by the class object /classes/comm.tcpMsg.

The third section specifies the initialization of the interfaces that are provided by the composite object. The line “interface distrObj = csRepl.distrObj;” specifies that all methods in the distrObj interface are copied from the corresponding methods provided by the csRepl subobject. It is also possible to initialize individual methods, for example "interface distrObj.init = dlCtl.distrObj.init;" specifies that the init method in the distrObj should be initialized with the corresponding init method of the control object.

The last part of a composite object definition specifies additions to the composite object's name space. The statement "addlink dlCtl/"semMutex" = dlSem;" inserts a reference (symbolic link) to the subobject dlSem in the name space of subobject dlCtl with the label semMutex. This allows dlCtl to bind to the string "semMutex" to get a reference to the standard object interface of dlSem.

```
#include <p/inf/stdobj.inf>
#include <p/inf/mutex.inf>
#include <globe/inf/distr.inf>
#include <globe/inf/distrCtrl.inf>
#include <globe/inf/gc.inf>
#include <globe/inf/persist.inf>

composition dl
{
        /* list of interfaces exported by the composition */
        interface stdObj;
        interface distrCtrl;
        interface distrObj;
        interface gc;
        interface persist;
        interface mutex;

        /* list of subobjects, each subobject is instantiated */
        object dlCtl = new "/classes/do.dlCtl";
        object dlSem = new "/classes/do.dlSem";
        object csRepl = new "/classes/globe.csRepP";
        object tcpMsg = new "/classes/comm.tcpMsg";

        /* initialization of exported interfaces */
        interface distrCtrl = csRepl.distrCtrl;
        interface distrObj = csRepl.distrObj;
        interface distrObj.init = dlCtl.distrObj.init;
        interface distrCtrl.create = dlCtl.distrCtrl.create;
        interface gc = csRepl.gc;
        interface persist = csRepl.persist;
        interface persist.create_ = dlCtl.persist.create_;
        interface persist.restart_ = dlCtl.persist.restart_;
        interface mutex = dlCtl.mutex;

        /* additions to the local name space */
        addlink dlCtl/"semMutex" = dlSem;
        addlink dlCtl/"replObj" = csRepl;
        addlink csRepl/"msgComm-passive" = tcpMsg;
        addlink csRepl/"msgComm-active" = tcpMsg;
};
```

Figure 5.30. Composite local object for a distributed lock

5.11.2 WWW-proxy

The WWW proxy is the main application program in this prototype. The implementation of the proxy uses the traditional UNIX networking system calls (socket, select, listen, accept, connect, read and write) to accept connections from WWW browsers, to setup connections to a real WWW proxy and to copy request data from the browser to the real proxy and WWW documents from the proxy to the browser.

Before the Globe proxy forwards a request to the real proxy, it reads and examines the requested URL. The proxy recognizes three different kinds of URLs that refer to a distributed shared object:

1. GLOBE://*path name*

   This URL specifies the use of Globe as a transport mechanism for accessing a WWW document. Unfortunately, many WWW browsers cannot be extended easily to accept these URLs and forward them to the Globe proxy.

2. HTTP://globe/*path name*

   This URL can be forwarded by almost any browser to a WWW proxy. The use of globe as a host name avoids confusion with normal URLs.

3. HTTP://globe.*domain-name*/globe/*path name*

   The second approach works only when the WWW browser is configured to use the Globe WWW proxy. A URL that is encoded using the third approach can be used in any situation. For example, the URL HTTP://globe.cs.vu.nl/globe/dns/cs.vu.nl/philip/ would be forwarded to a special WWW server that can access distributed shared objects.

The Globe proxy creates a new thread for every request for a distributed shared object. The total number of connections for browsers is limited to avoid an excessive number of connections and threads. Once the limit is reached, the proxy stops accepting new connections.

The method that is invoked on a distributed shared object depends on which interfaces are provided by the object. The proxy checks for the presence of the following interfaces in order:

1. file interface

   The contents of the file is copied to the browser. The contents of optional attributes, provided by the wwwAttr interface, are inserted before the contents of the file.

2. naming interface

   The proxy generates a list of directory entries formated in HTML. Attributes accessible through the wwwAttr interface are returned before the directory entries.

Program	**Description**
f_gc	Garbage collect a factory
f_start	Start a factory (tell it what class of objects to produce
g_create	Tell a factory to create a new distributed shared object, or a new representative for an existing object
g_del	Delete a directory entry
g_link	Create a hard link to a distributed object
g_list	List the contents of a directory object
g_lock	Lock or unlock an object (invoke the lock or unlock methods of the mutex interface on a distributed object)
g_read	Read some data from a file object
g_write	Write data to a file object
g_locman	Register a (meta) location service address manager with either the location service or with a meta address manager
l_gc	Garbage collect the location service
g_wwwattr	Manipulate attributes and subobjects of WWW objects

Table 5.6. Globe utilities

3. objMgt interface

   The list of interfaces supported by the object is returned for objects that implement neither the file interface nor the naming interface.

An error message is generated for objects that do not support at least one of the three interfaces. Subobjects are a special case. The name of subobject consist of a the name of a actual distributed shared object combined with the name of the subobject. The two parts are separated by a dollar character ("$") The proxy first tries to bind to a name. When binding fails, the proxy strips off the part starting with the dollar character and tries to bind to the prefix. After the second bind succeeds, the proxy uses the wwwSO interface to return the contents of the subobject.

5.11.3 Utilities

Table 5.6 lists a collection of simple utilities for manipulating distributed shared objects. Most of these utilities do nothing more than parse arguments, bind to the relevant disitributed objects, and invoke a few methods on those objects. For example, the f_gc program invokes the gc method from the factory interface, the g_create program invokes either the createRep method or the create method. The g_link program is slightly more complex: the add method of the naming interface is invoked on the destination directory object, and the addRef method of the gc interface is invoked on the actual object.

Chapter 6

Related Work

The related work that is described in the chapter is split into two parts. The first part describes other systems or approaches to worldwide communication. The second part describes other work that relates to the specific techniques that are described in this thesis.

6.1 Worldwide Systems

6.1.1 The Internet

Today, the Internet protocols form the most popular network technology for worldwide communication. The Internet protocols use a simple, layered approach. At the lowest level, standards define the transmission of Internet packets over various types of networks, such as Ethernet [Hornig, 1984], ATM [Laubach, 1994], or serial links [Simpson, 1994]. The Internet Protocol itself forms the network layer [Postel, 1981a]. The two most common transport protocols are TCP [Postel, 1981b] and UDP [Postel, 1980]. Most applications run directly over TCP (mail [Postel, 1982], news [Kantor and Lapsley, 1986], WWW [Fielding et al., 1999]) or UDP (time synchronization [Mills, 1991] and the Domain Name System [Mockapetris, 1987]).

An advantage of most Internet protocols is that the structure of these protocols is relatively simple. Usually, there are only a few states and protocol encodings are simple. The application-level protocols often use ASCII to encode requests and replies. Unfortunately, these protocols are also inflexible. For example, the news protocol NNTP is designed to push news articles from one news server to another. This means that this protocol is hard to use to pull a selection of news articles over a dial-up line.

The Point-to-Point protocol (PPP) is designed to be flexible; many parameters can be negotiated at run-time. However, this makes the implementation of this protocol harder because each implementor has to figure out how, for example, the data-compression feature [Rand, 1996] should be combined with the multi-link option [Sklower et al., 1996].

The flexibility of the Internet protocols is also limited by lack of standard models for implementation. For example, the original HTTP specification sets up a new TCP connection for each document that is transfered to the browser [Berners-Lee et al., 1996]. Using a TCP connection for only one document wastes network bandwidth (extra packets to set up and tear down connections) and increases latency. The obvious solution was to add persistent connections to the next version of the protocol [Fielding et al., 1997]. However, by that time, most WWW browsers were using multiple parallel connections to a single server to download images and frames that are embedded in WWW documents. The use of multiple persistent connections from each client is likely to overload servers. A common implementation technique for UNIX WWW servers is to fork a new child process for each HTTP request. Unfortunately, forking new child processes complicates the resource management which is needed to avoid overloading the server. Even though support for persistent connections has been added to many WWW browsers and WWW servers, a small scale survey shows that 86% of the analyzed requests use a connection of their own [Pierre, 2000]. It is not clear why the use of persistent connections is still not common.

A limitation of the current Internet name service (DNS) is that the name service names hosts, not services or objects. This means that usually, only one instance of service (mail, news, etc.) can run on a machine, and that every protocol has to provide an additional name space to select the right object within a service. Some extensions to DNS are proposed to name services (but not objects) [Gulbrandsen et al., 2000]. Another alternative is to assign more than one IP address to a single machine. Effectively, this machine becomes multiple hosts and can therefore run multiple instances of a single service.

6.1.2 Legion

The goals of the Legion project [Grimshaw and Wulf, 1997] are similar to the goals of Globe: extensibility, scalability, security, language independence, and support for heterogeneity. Legion is implemented using an object model [Lewis and Grimshaw, 1996]. Classes play a central role in the naming and binding of objects in Legion. Every legion object is named by a Legion Object Identifier (LOID). These object identifiers consist of four parts: a format identifier, a class identifier, an instance number, and a public key.

Each class is responsible for providing bindings (a mapping from a LOID to a contact address) for its instances. In contrast, Globe uses a location service, which operates independently of the classes that implement distributed shared objects. The binding algorithm described in [Lewis and Grimshaw, 1996] is not very scalable: the algorithm relies on the effectiveness of "**binding agents**" that cache bindings, and the ability to "clone" (split) overloaded classes. At first glance, this use of classes seems to be similar to a home-based approach: the place where an object is registered is independent of where the object is located. Better implementations are possible (it is possible that each class uses a distributed search tree to locate objects), but are not described. The use of binding agents that cache the location of objects can speed-up binding and reduces the load on classes. However, the effectiveness of these binding agents is unclear. Other problems with a home-based approach, such as dealing with objects that have a large number of replicas are not addressed either.

The implementation of objects is described in terms of a message-passing system between independent address spaces. An object accepts a set of methods. A method has a signature which describes the parameters and return values of the method. A complete set of method signatures for an object describes the object's interface. An interface is specified in one of two interface description languages: either CORBA IDL or the Mentat programming language [Grimshaw, 1993].

The persistence mechanism described in [Grimshaw et al., 1998] is similar to the persistence model shown in Figure 3.19 on page 76. This model is hard to use for highly-replicated objects. Furthermore, the objects that store persistent state, called **vault objects**, only operate on complete copies of the objects state (i.e., it is not possible to make small changes to the persistent state).

Replication is discussed in the context of a general purpose reflection mechanism called RGE (Reflective Graph & Event model) [Nguyen-Tuong and Grimshaw, 1997]. This mechanism can be used to cause a client to sent a message to multiple servers instead of to just one server. Compared to Globe's replication object, this approach is less flexible because it operates on messages that are sent and received. A mechanism for invoking a method in the same address space as the invoker is not described.

The security of Legion is based on MayI methods that are provided by objects [Wulf et al., 1996]. The MayI method is invoked with three parameters: the calling agent (CA), the responsible agent (RA) and the security agent (SA). The calling agent is the object that initiated the method invocation, the responsible agent is a generalization of the user who is responsible for the method invocation, and the security agent allows security policies to be forced external to the object that is invoked. The MayI method is invoked prior to the actual method. If the MayI method allows access, it returns a special license. This license specifies the set of methods that can be invoked, the time that the license is valid, the number of method invocations that are allowed, and which if the three credentials (CA, RA, and SA) should be checked upon future invocations. The system supports delegation of rights to another object through the Delegate method. Mandatory access control can be implemented by inserting a security agent in an object environment. The security agent's pass method will be invoked before a method of the target object is invoked. This provides the security agent with full control over the methods that are invoked on the object. The pass method may refuse the invocation of the real method on the target object.

Object placement is described in [Karpovich, 1996]. Special mappers decide where an object should be placed (this happens also when a persistent object is activated). An object's default mapper is stored in the object's class object.

6.1.3 Globus

The goal of the Globus project is to enable the construction of "computational grids", which provide access to high-performance computational resources, despite geographical distribution of both resources and users [Foster and Kesselman, 1998]. Globus provides a toolkit with components that implement basic services for security, resource location, resource management, communication, etc. An important design principle of Globus is that interfaces are used to manage heterogeneity, rather than hiding it. In contrast, Globe use high-level interfaces to separate applications from objects. The Globus approach is that high-performance applications adapt to the underlying facilities, or request more efficient implementations with weaker guarantees (for example unreliable communication). Globe uses different object implementations to solve this problem.

The Globus security architecture assumes that a local (site) security policy cannot be replaced [Foster et al., 1998]. Furthermore, Globus is not allowed to override local policy decisions. The central component of the Globus security architecture is a **resource proxy** which maps worldwide security operations to local security mechanisms and vice versa. In this approach, the Globus security mechanisms are built on top of local security mechanisms. The resource proxy maps the global user identifier to a local identifier. The local identifier is used in normal, local access control checks. Inter-domain access control is needed when resources (processors, network bandwidth, storage) are allocated.

In contrast, the Globe security mechanisms replace the existing local mechanisms. This is done for two reasons. The first reason is that, at the moment, few sites have installed proper distributed security systems (such as Kerberos). Many protocols that are used on the Internet use host-based authentication and/or username/password combinations. Both approaches are not suitable for use over a wide-area network (spoofing and eavesdropping are too easy). The second reason is that in Globe, the ratio between local, intra-domain operations and global, inter-domain operations is expected to be small. Global security mechanisms have to be used when either a process uses an object in a remote domain or when the object itself is replicated across multiple domains.

Globus introduces a **user proxy**,[1] which handles the allocation of resources on behalf of a user. Whenever a remote process (which is typically part of a large computation) wants to allocate more (remote) resources, it sends a request to the user proxy to allocate those resources. The user proxy interacts with one of more resource proxies to actually allocate the requested resource. The advantage of that approach is that allocation of resources for a single computation is centralized.

The communication layer in Globus is called Nexus. This layer provides one basic operation, the Remote Service Request (RSR) [Foster et al., 1996]. A **Remote Service Request** is an asynchronous request for a remote process to invoke a particular handler. The recipient of an RSR is identified by a Global Pointer (GP). A Global Pointer can be thought of as a capability and identifies a memory location in an address space. How different address spaces are addressed is left unspecified.

In contrast, Globe does not require a specific communication mechanism. Only the interfaces to communication objects are specified. The main requirement for a Globe communication object is that the object can marshal and unmarshal addresses as ASCII strings.

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. The design of Globe is more oriented to accessing and distributing data over a wide-area network.

6.1.4 CORBA

The Object Management Group (OMG) was founded to promote the use of object-oriented technology in software development. Primary goals are the re-usability, portability, and interoperability of object-based software in distributed, heterogeneous environments. The OMG defines an object system as a collection of objects that isolates the requesters of services (clients) from the providers of services by a well-defined encapsulating interface. In particular, clients are isolated from the implementations of services as data representations and executable code [Object Management Group, 1999].

This section evaluates CORBA for use in a worldwide distributed system. It should be noted that CORBA was not designed for use in a wide-area environment (in contrast to, for example, Globus, Legion, and Globe). However, it is often suggested that the Internet inter-ORB protocol can be used to connect different CORBA systems over the Internet.

An Object Request Broker (ORB) is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface that the client sees is completely independent of where the object is located, what

[1]The user proxy is called **GRAM** (Globus Resource Allocation Manager) in [Foster et al., 1999].

programming language it is implemented in, or any other aspect which is not reflected in the object's interface. The Common Object Request Broker Architecture (CORBA) specifies the interface to the ORB.

Note that in Globe an object encapsulates some data. An object exports an interface table with methods that can be invoked directly by users (clients) of the object. The user of an object is in the same address space as the object, that is, a method invocation does not require actually sending any messages. Furthermore, method invocations are direct without any intermediary. Distributed objects also expand into the client's address space; when invoking a method on a Globe object, the object is always local.

Object references in CORBA are defined to specify an object within an ORB. Both clients and object implementations have an opaque notion of object references according to the language mapping and thus are insulated from the actual representation of them. Two ORB implementations may differ in their choice of object reference representations. The representation of an object reference handed to a client is valid only for the lifetime of that client. Equivalence of object references is defined poorly. The ORB interface provides a function to compare two object references. If this function returns true, the two object references refer to the same object. Otherwise, the object references may or may not refer to the same object.

In contrast, in Globe a local object reference (which is discussed here) is simply a pointer to an interface table for the object. Other object references are first turned into a pointer to an interface table by binding to the object. Every object handle for a distributed shared object contains the object's unique identifier, which can be used to check whether two object handles refer to the same object.

CORBA includes an Interface Repository, which is a service that provides run-time access (using persistent objects) to interface definitions. Using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make an invocation on it. The Interface Repository is a common place to store additional information associated with interfaces to ORB objects, for example, debugging information, libraries of stubs or skeletons, routines that can format or browse particular kinds of objects, and so on.

Note that a CORBA Interface Repository is more like a class repository: it stores information about classes of objects. In Globe, the equivalent would be a repository with protocol descriptions. Interfaces are local in Globe and an object provides a collection of interfaces. Parsing a new interface description at run-time is complicated without a formal description of the semantics.

In addition to static typing, CORBA also provides dynamic typing in the form of the Dynamic Invocation Interface (DII). This interface allows a client to create a request message with a run-time selected method name and parameter types. Globe cannot directly provide an equivalent to DII because method invocations on Globe objects are local function calls using binary interface tables. In theory, Globe objects can provide a special interface to allow function names and parameter types to be specified at run-time. As an alternative it might be possible to extend Globe objects with support for a scripting language such as Tcl [Ousterhout, 1989].

The Dynamic Skeleton Interface (DSI) is the complement of the Dynamic Invocation Interface. This interface allows an object to accept method invocations that are not part of its interface. In Globe, the interface between the client and the distributed shared object is provided by the local object in the client's address space. Therefore, this kind of functionality would be an implementation detail private to a Globe distributed shared object. It is possible that certain object-implementation architectures provide a way to dynamically extend the network messages they support through a DSI-like interface. Again, a scripting language might be more appropriate.

The interface repository associates "unique" identifiers with language constructs such as modules, interfaces, methods, types, etc. These identifiers are supported by the IDL through pragma constructions. Four categories of unique identifiers exist: IDL, RMI, DCE and LOCAL. The IDL space uses an informal, hierarchical name space. The RMI space is based on Java Remote Method Invocation specification. The DCE space is based on DCE's UUIDs, and the LOCAL space is unstructured and for local use only. Only the DCE UUIDs space looks like our unique identifiers. It is not clear what happens if two interface repositories use the same identifiers for different interfaces.

To actually implement an object, CORBA provides a Portable Object Adapter (POA). This object adapter provides a portable environment for implementing objects. The two main functions of the POA are to pass requests from clients to an object implementation and to activate and passivate objects. The second function is similar to the manager object that is described in Section 3.3.2.

CORBA does not specify which network protocols should be used by an ORB. An ORB may, for example, use a standard RPC protocol to forward a request from a client to an object server. Alternatively, for lightweight objects, the object implementation may be linked with the client. This is called a library-based ORB. In theory, an ORB provides each object with a choice between a collection of replication protocols.

In the context of a large-scale distributed system, it is relevant to look at the protocols that are used between different ORBs. In such a system, it is possible that clients of an object use different ORB implementations. CORBA defines a collection of inter-ORB protocols to allow different ORBs to cooperate. The General Inter-ORB Protocol (GIOP) defines generic, CORBA-specific parts of an inter-ORB protocol such as a standard low-level data representation, and a set of message formats. The Internet Inter-ORB Protocol (IIOP) specifies how GIOP messages are exchanged over the Internet, using TCP/IP connections.

One issue is that object references have to be converted from the form used within the ORB that sends the reference to the form that is used in the receiving ORB. Various techniques are described (reference translation, reference encapsulation, domain reference translation, and reference canonicalization). Reference canonicalization is similar to the use of global object references in Globe. The other approaches either require extra state or lead to unbounded growth of object references. Without the requirement that object references are global, it is hard to verify whether two object references refer to the same object or not.

A serious limitation of the general inter-ORB protocol is that it does not support replication. The protocol supports requests sent by clients and replies sent by servers. This allows an object to be replicated in the server ORB, but does not support objects that contain replicas in different ORBs. For objects with few replicas, it is possible that all replicas are located in a single ORB. CORBA does not prevent an organization from creating a single ORB that contains machines in different countries. In practice, lack of locality may impact the performance of this approach because clients are unable to select addresses that refer to a nearby replica of an object.

GIOP contains a primitive interface to a location service. When a ORB receives a request for an object, it can return a redirect reply which contains the new location of the object.

The core CORBA object model has been extended with a collection of services [Object Management Group, 1998]. The services have been defined for, among others, naming, events, persistence, transactions, concurrency control, queries, licensing, time, and security. Most of these services provide interfaces, some of which are used by clients and others by objects. For example, a client can use the concurrency control service to synchronize access to shared resources with other clients. Objects can use the persistent object service to implement persistence. The difference with the approach taken in Globe is that the services are external to an object. In Globe, an object would use a standard local object which provides the service internal to the object.

The CORBA name service is a standard hierarchical name service. The naming system is based on naming contexts: objects that store references to other objects. These objects can be used to create a naming graph. The name service does not include a global naming root; name resolution is relative to a naming context. A name in a naming context is called an identifier attribute. Associated with this name is a kind attribute. A kind attribute is a string that describes the kind of object, for example "c_source," or "executable." The name service always returns local object references. In contrast, the Globe name service returns object handles.

The CORBA security service is based on an extensive security model, which includes, among others, many forms a delegation, multiple security policies, non-repudiation, and client-side access control. As with most CORBA services, security is implemented outside the object. In general, security mechanisms are implemented by the security services in the ORB.

The CORBA persistent object service separates persistent state from objects. For example, this means that one piece of persistent state can be loaded into two different objects. The CORBA object model does not include explicit support for replication. As a result, each persistent object uses only one persistent data object to store its state. In contrast, the approach taken in Section 3.3.2 uses one piece of persistent state for each replica of the state of a distributed shared object that is made persistent.

The life cycle service supports creating, copying, and deleting objects. Object are created by factories. Each factory creates one particular kind of object. A collection of factories can be registered with a factory finder, which allows clients to locate an appropriate factory to create a specific kind of object.

The concurrency control service provides a standard interface to lock objects. Clients can use those objects to coordinate access to shared resources. Locks can also be used by object implementations for internal concurrency control. The interface to the locks provides various locking modes (read, write, upgrade, intention read, and intention write), but provides no support for condition synchronization (semaphores, or condition variables).

CORBA provides an event service, which allows events to be sent over a channel. Two kinds of clients are attached to a channel: suppliers, which provide new events, and consumers, which accept events. Two different distribution strategies are recognized: using the push style, a supplier pushes an event to consumers. The pull style works the other way around: a consumer pulls data from suppliers. A separate event service is needed because implementing an event channel as a CORBA object is complicated due to the lack of standard replication strategies and the lack of condition synchronization. In Globe, blocking operations can be executed either locally, to wait for new events to be pushed, or they can block remotely to pull new events. An appropriate replication object can be chosen to push or pull data.

6.1.5 COM and DCOM

Microsoft's Component Object Model (COM) and Distributed COM (DCOM) are described in [Microsoft Corporation, 1995; Brown, 1998] and [Eddon and Eddon, 1998]. Like CORBA, DCOM is not explicitly designed for use in a wide-area environment. However, it is suggested that DCOM can be used to connect applications over the Internet [Microsoft Corporation, 1997].

The object model of COM is similar to the local object model in Globe. A COM object provides one or more interfaces, an interface is a list of methods, and interfaces are run-time, binary data structures (as opposed to language constructs). Furthermore, interfaces have worldwide unique interface identifiers. These interface identifiers are 16-byte, worldwide unique bit strings.

COM interfaces are reference counted, just like Globe interfaces. In contrast to Globe, a COM object destroys itself (the class object is not involved) when its reference count drops to zero. Furthermore, COM does not solve the problem of cyclic references due to callback interfaces.

Each COM interface inherits from the IUnknown interface which provides three methods QueryInterface, AddRef and Release. In general, COM library functions and COM methods do not return a pointer to the "standard object interface" (IUnknown), instead, the caller always asks for a particular interface. Unfortunately, the implementation is such that no type checking is done.

Method invocation involves an extra level of indirection. The pointer to an interface table is actually a pointer to a pointer to an interface table. The reason is (probably) the C++ origin in which interfaces can be implemented as virtual base classes. For historical reasons, the Pascal calling convention[2] is used to invoke methods on COM objects.

The object model in COM also deals with class objects, which are called "class factory objects." The class objects are somewhat special because they are explicitly part of a shared library, a local process or a remote process. Classes have class identifiers which are used to locate a suitable implementation. The COM implementation heavily depends on the Windows Registry to map class identifiers to implementations.

COM supports a composite object mechanism called aggregation. A problem is that COM's interface tables do not contain state pointers, which makes mixing methods from different subjects impossible. Unfortunately, mixing is required for aggregate objects because every interface inherits from IUnknown. This problem is "solved" by passing a pointer to the IUnkown interface of the outer object to a subobject. The subobject is expected to delegate its QueryInterface, AddRef and Release methods to the outer object's IUnkown interface. Disadvantages of this approach are that the subobject is aware that it is part of a composition and that this trick only works for the IUnknown interface. COM does not recognize the concept of a compositor, instead one object delegates some of its interfaces to subobjects.

The way callback interfaces are set up is quite different from Globe. In Globe, a pointer to a specific callback interface is usually passed as an argument to a method invocation on a subobject. In COM, a connection is established between two objects. The subobject that wishes to invoke methods on a callback interface provides so-called "connection points." The parent object that implements the callback interface registers its interface at the subobject's connection point. Using the registered callback interface, a subobject can ask for an arbitrary interface from the "parent" object. It is not clear how an object with multiple "parents" can distinguish between different callback interfaces from those parents.

[2]The Pascal calling convention pushes arguments on the stack in the opposite order from the one used by C and C++. Furthermore, the callee is responsible for popping the arguments from the stack, instead of the caller as is common in C and C++ implementations.

Persistence is for most part external to an object (explicit load/store). Two interfaces are used for persistence. The first interface is a file interface and the second a stream interface. The file interface simulates a hierarchical file system within a single file. This allows the subobjects of an object to store their states in different parts of the file. Limited support for transactions is available to avoid partial updates. The stream interface has more in common with a UNIX file than with a pipe or a network connection. In addition to read and write methods, the stream interface also contains seek, "copyto," and commit methods. Some stream objects implement only the read and write methods. The file interface uses the stream interface for the actual data transfers, and implements all methods in the stream interface.

DCOM The distributed object model in DCOM is a simple remote object model with some extensions. By default, the DCOM run-time system provides an RPC transport layer and each interface provides proxy objects that marshal and unmarshal method arguments and return values. These objects are typically generated by an IDL compiler.

DCOM provides two extensions to the standard proxy mechanism. The first extension, called handler marshaling, allows the client stub to be encapsulated by another object that provides, for example, caching. Note that this is an extension to the marshaling of a specific interface. Method invocations are still handled by the generated stub, and COM library. The second extension is the use of custom marshaling. In this case, the programmer writes the client and server stubs manually, but still uses the COM library for communication.

With custom marshaling, the programmer can also decide to handle all aspects of the distributed object himself. However, there are no standard communication objects, there are no replication protocols, and dealing with addresses and communication protocols is not part of the model. DCOM does not provide anything equivalent to object handles and contact addresses in Globe. Instead, marshaled interface pointers are passed from one address space to another. Using custom marshaling, it would be possible to marshal an interface pointer as a Globe contact address or object handle.

DCOM defines a standard IDL (which may be ignored by the programmer) based on DCE IDL. Extensions are added to the original DCE IDL to deal with objects and interfaces. Special tricks are needed to deal with methods that return pointers to interfaces. In Globe, only the getinterface method in the standard object interface can be used to obtain a pointer to an arbitrary interface. Other methods typically return a pointer to the standard object interface. In DCOM, many methods return pointers to arbitrary interfaces. Unfortunately, there is no type checking when pointers to interfaces are returned.

The default communication protocol used by DCOM is DCE RPC. Among the extensions to DCE RPC are Causality IDs. Causality IDs are associated with RPCs to prevent deadlocks. An outgoing RPC during the processing of an incoming RPC uses the same causality identifier as the incoming request. A server should not block requests with incoming causality identifiers equal to one of its outstanding causality identifiers. DCOM uses reference counts between machines to implement garbage collection. Clients ping servers to keep an object alive.

Monikers are the main naming mechanism of COM. A moniker is a persistent immutable pointer to a COM or DCOM object. A moniker can refer to a local object, a class object, a remote object, etc. Internally, monikers can use concepts such as “the current month.” This means that in different months, such a moniker points to different objects.

Multiple monikers can be combined into a composite moniker. A composite moniker is a list (or path) of monikers that leads to a subobject of a distributed object. (Composite) monikers are similar to name spaces, but lack the structured syntax that is normally associated with a name space. The result is that monikers are more efficient, binary data structures, but it requires a lot of special interfaces to operate on monikers.

Monikers support a reduction operator. This means that the moniker is partially evaluated and that a level of indirection is lost. For example, a moniker may refer to a file object by its filename. A reduction results in a moniker that refers to the same file based on its file id, or object id. A COM object may contain subobjects. A moniker can refer to one of those subobjects. In some cases, these subobjects are created on the fly, such as a row in a spreadsheet.

6.1.6 WebOS

The goal of the WebOS-system is to provide a common set of Operating System services to wide-area applications, including mechanisms for naming, persistent storage, remote process execution, resource management, authentication, and security [Vahdat et al., 1998; Vahdat, 1998]. For persistent storage, WebOS uses a network file system abstraction called WebFS. This file system uses the HTTP protocol for communication. The URL name space is integrated with the name space provided by the local file system. For example, it is possible to list the directory /http/www-c.mcs.anl.gov/hpdc7/. This approach is similar to the Alex file system, which used the FTP protocol for this purpose [Cate, 1992]. The naming mechanism in WebOS is designed with load balancing issues in mind. To this end, some amount of executable code is downloaded onto clients to allow them to select the best replica of a particular service. During a session, clients receive load information from servers. Clients use this information to make load balancing decisions. WebOS includes a mechanism for starting processes on remote computers. Processes on remote computers are prevented from executing arbitrary system calls by the Janus-system [Goldberg et al., 1996]. The Janus-system uses the debugging facilities of an operating system to stop a process when it is about the perform a system call. The process is aborted when the system call is not allowed according to the security policy. If the system call is allowed, the process is resumed and can continue until the next system call.

The security architecture that is used in WebOS is called CRISIS [Belani et al., 1998]. Two interesting aspects of CRISIS are the use of countersigning to implement revocation lists and the use of transfer certificates to delegate rights. A problem with public key certificates is that they are hard to revoke. It is possible to limit the life-time of a certificate, but in many cases it is more important for the security of the system to keep certification authorities off-line and, thus, use certificates with a relatively long life-time. In CRISIS, a certificate is valid only if it is counter-signed by another service. This second service consults the revocation list before counter-signing a certificate. Counter-signed certificates have a relatively short life-time. The advantage of this approach is that the certification authority can be kept off-line, and that certificates can be trusted without accessing revocation lists.

CRISIS explicitly models the transfer of rights from a user to a process that runs on a remote machine. The user signs a certificate that grants a process on a remote machine certain rights, for example, to access a specific file. This process can also sign a certificate that grants a second process, possibly running on a different machine, a subset of the first process' rights. It is also possible for a process to transfer the right to transfer rights to another process.

6.2 Local Objects

The use of objects to implement operating-system kernels or run-time systems is not new. Choices is an example of an operating-system kernel which is constructed using objects [Russo, 1991]. A technique called frameworks is used to structure the use of objects in Choices [Campbell et al., 1991]. These frameworks are used during the design of the system and are represented by abstract C++ classes at run time.

In Globe, the main structuring mechanism for local objects is the composite object. The main difference between frameworks and composite objects is that frameworks are design elements and composite objects are run-time constructs. The result is that composite objects are more flexible than frameworks. However, frameworks are designed to impose some constraints on the use of objects.

The use of multiple interfaces for a single object combines static type checking with the possibility to add more interfaces as objects and applications evolves while ensuring backward compatibility. In programming languages designed for scripting, this kind of flexibility is built-in [Ousterhout, 1998]. The disadvantage of almost no type checking at compile time is that many kinds of type errors show up at run time. The language Emerald provides stronger type checking while maintaining most of the flexibility of languages without type checking. In Emerald, the programmer declares the abstract type of the objects that an identifier may name [Black et al., 1987; Raj et al., 1991]. The legality of an assignment is based on the conformity of the assigned object and the abstract type declared for the identifier. The advantage of this approach is that (in C++ terminology) object classes can be subclasses of virtual base classes without being declared as such. This means that a new abstract type can be added to a program without changing any class declarations.

The disadvantage of both scripting languages and the approach used in Emerald is that extensive compiler and run-time support is needed to implement the necessary type checking. The approach used in Globe requires only one simple check (in the getinterface method), and is therefore also suitable for use in nonobject-oriented languages such as C, Modula-2, or even assembly language.

The language Java provides an interface concept that is at first glance similar to interfaces in Globe [Gosling et al., 2000]. The most important difference is that a Globe object can provide two interfaces that contain two different methods with the same name and signature but with different semantics. In Java, this is not possible. This means that in Java, interface designers should avoid using method names that are also used in other interfaces.

Paramecium uses the same local object model as is used in Globe to structure operating-system kernels and run-time systems [van Doorn et al., 1995]. The idea behind Paramecium is to extend the operating system by loading modules either in user space or in kernel space. Loading modules in kernel space is efficient. On the other hand, loading modules in user process provides better fault isolation. The object model is used to hide these details from application programs.

The design of the communication objects is for a large part influenced by the x-Kernel [Hutchinson and Peterson, 1991]. Features that the communication object have in common with the x-Kernel are:

- The creation of a separate object for each communication channel (session).
- The ability to connect a collection of communication objects to implement a protocol stack.
- The use of pop-up threads to propagate incoming network packets through the protocol stack.
- The use of a mapping service that maps network addresses and port numbers to fixed sized identifiers for internal use.
- A message data structure that allows message buffers to be manipulated without copying the data in those buffers.

The main difference between the Globe communication objects and the x-Kernel is the ability to load Globe objects dynamically. In contrast, the x-Kernel contains not only a fixed set of protocol objects, the graph that connects the those protocols is also fixed. In Globe, a protocol stack is created by connecting communication objects in a composite object.

The Horus system extends the x-Kernel approach to group communication (the x-Kernel focuses on point-to-point communication such as remote procedure calls and streams) [van Renesse et al., 1996]. Even though Globe includes an interface for multicast communication objects, no multicast communication objects have been implemented yet. For this reason, it is not possible to compare multicast communication objects in Globe with the approach taken in Horus.

6.3 Distributed-Object Architecture

6.3.1 Object Models

Some object-based distributed systems rely heavily on compiler support. For example, in the language Emerald the compiler is used to solve a problem that exists in older object-based languages for distributed systems such as Argus [Liskov, 1988]. The problem is that in Argus, the programmer has to decide whether an object can be accessed by multiple processes, or whether the object is just local. In Emerald, the compiler generates more efficient code if it finds that an object is accessed only locally [Jul et al., 1988].

In Orca, the compiler is used to distinguish read-only methods from methods that (may) change the state of the object. For replicated objects, read-only accesses can be implemented more efficiently than accesses that modify the state of the object [Bal et al., 1992a].

Most object-based distributed systems try to avoid the use of special compilers. Instead, support for objects is implemented in a run-time system, which is called from applications written in standard programming languages. For example, the Amber system implements the distributed model and mobility primitives from Emerald, but is implemented using C++ and a run-time system [Chase et al., 1989].

Other systems that implement support for distributed objects in a run-time system include CORBA [Object Management Group, 1999], Fragmented Objects [Makpangou et al., 1994], Spring [Hamilton et al., 1993], and Network Objects[Birrell et al., 1993b]. The main difference between these systems and our model is the way access to a distributed object is provided. Network Objects supports "remote objects": a method invocation in one address space is transparently transferred by the local run-time system to the address space where the object resides. This system lacks support for replication, other than layered on top of the remote objects. In Network Objects, an object always stays in the same address space. CORBA and Spring are more complicated but basically have the same model. CORBA supports "request brokers" to get some flexibility. Spring supports "subcontracts" that can be used to implement caches, support for replication, etc.

In the three systems mentioned above, the client stub is provided by the run-time system and is not part of the distributed object. In contrast, in our model the local object that provides the interface to the distributed object is logically part of the distributed object. This local object is (again logically) self-contained and provides its own communication mechanisms. This means that completely new ways of structuring distributed objects can be deployed without changes to the model.

The use of object-oriented techniques to provide a better World-Wide Web has been proposed in [Ingham et al., 1995]. Advantages of the use of objects are extensibility (different interfaces), support for replication, and better naming systems, including referential integrity.

Fragmented Objects

Fragmented objects are an implementation technique for distributed objects. A Fragmented Object consists of a collection object called fragments, connected through connective objects [Makpangou et al., 1994]. A fragment corresponds roughly to a representative of a Globe distributed shared object. A fragmented object uses two interfaces: a client interface and a group interface. The client interface is provided to the users of the fragmented object to invoke a method on the object. The group interface is used by the fragments to communication with each other. Connective objects are fragmented objects at a lower level of abstraction.

This approach is similar to Globe's distributed shared objects. The main difference is that a fragmented object uses connective objects for internal communication. The communication objects used in Globe are just end points of a communication channel. Distributed shared objects use replication objects to handle the replication and placement of state. The fragmented object model leaves this to the programmer.

Spring

Spring uses a mechanism called subcontracts to implement different replication policies [Hamilton et al., 1993]. Subcontracts provide a client stub with five operations: marshal, unmarshal, marshal_copy, invoke_preamble, and invoke. Client stubs implemented using these operations can be used with different subcontracts. Example subcontracts deal with simple remote objects (singleton subcontract), replication (cluster subcontract), caching and persistence.

Compared to Globe, a subcontract implements the functionality of a replication and a communication object. A stub is layered directly above the subcontract and corresponds to the control object in Globe. Methods that are invoked on the object are actually invoked on the stub. The main difference is absence of the semantics object. This means at a subcontract cannot be used to implement a method invocation completely in the clients address space. For example, the caching subcontract that is described in [Hamilton et al., 1993] uses separate cache managers to actually store a copy of the state of the object.

Network Objects

By design, Network Objects are a straightforward remote object extension to Modula-3 [Birrell et al., 1993b]. The system includes support for invoking methods on remote objects and for passing object references from one address space to another. The local Modula-3 garbage collector is complemented with reference listing to handle remote object references [Birrell et al., 1993a].

Java RMI

The Java Remote Method Invocation (RMI) specification is quite similar to that of Network Objects [Sun Microsystems, Inc, 1998]. Apart from the use of Java instead of Modula-3, the main difference is the garbage collector. In Java RMI, the client of a remote object gets a remote object reference that is valid for some period of time. The reference has to be renewed before it expires. The garbage collector can assume that an object is no longer referenced from other address spaces when all remote references are expired. When a client deletes a remote reference, it can inform the object. This allows an object to be garbage collected before the reference is expired.

Clouds

Clouds (version 2) supports objects on top of a small micro kernel called “Ra.” A Clouds object is persistent, and is not tied to any process [Wilkenloh et al., 1989]. The interface to the kernel is provided through system objects. In this sense, the Clouds local object is similar to the local object model in Globe. The main difference between Clouds and Globe with respect to the object models used, is that in Globe an object implementor is free to decide how an object should be replicated, or whether an object should be persistent or not.

Infospheres

The Infospheres project focuses on providing support for collaborative applications. The canonical example is a distributed calendar application which is used to setup a meeting with multiple people, often from different organizations. This project is implemented using active objects, called dapplets [Chandy et al., 1996]. Each object has a set of inboxes and a set of outboxes. An object receives messages from its inboxes and sends messages to its outboxes. Each outbox is connected to a set of inboxes that belong to other objects. The advantage of explicitly modeling inboxes, outboxes, and the connections between inboxes and outboxes is that the correctness of the system can be verified.

This model is quite different from Globe. Firstly, in Globe, the state of an object is usually replicated. Secondly, Globe objects are passive; threads of control are provided by applications and by the communication system as pop-up threads. Lastly, in Globe, objects are used by applications to communicate; objects themselves typically do not interact.

6.3.2 Alternative Approaches

Distributed Shared Memory

A relatively recent example of a system that provides distributed shared memory (DSM) is Khazana [Carter et al., 1998]. In addition to the normal read and write calls, Khazana provides explicit lock and unlock calls for synchronization and operations to get and set attributes of a region of shared memory. These attributes include the desired consistency protocol, access control, and number of replicas. Relatively large (128-bit) identifiers are used to address memory regions. Khazana tries to provide many common features that are needed by distributed applications such as replication, fault tolerance, access control, and location management.

One of the problems with the use of distributed shared memory is heterogeneity [Zhou et al., 1992]. Processes running on different architectures may use different byte orders, use different word sizes, etc.

The tuple space provided by Linda can be viewed as a high-level distributed shared memory system [Carriero and Gelernter, 1989]. In contrast to most other systems, which are addressed using names, object references, memory addresses, etc., tuples are addressed by specifying part of their contents. A new tuple is simply inserted in the system using the out operation. No address is passed or returned. The rd operation searches for (and returns) a tuple that matches the template passed as a parameter. The in operation is similar to the rd operation, except that the tuple is also removed from the tuple space.

This kind of associative memory is unlikely to be useful as a basis for a worldwide system. The reason is that content-based addressing is hard to scale. A tuple can be selected on any combination of its fields. This makes it very hard to partition a tuple space or to build a proper index structure to quickly find tuples in a large system. A second problem is that the Linda tuple space provides strong consistency guarantees. The in instruction atomically selects a tuple and deletes it from the tuple space. Furthermore, a common technique to update a variable in a tuple space is to remove the tuple that contains the old value and to insert a tuple with the new value. The net result is that the use of replication is unlikely to improve the performance of the system as a whole.

Distributed File Systems

Distributed file systems and distributed shared memory systems have in common that they both operate on bytes. The main difference is that files are by default persistent and shared memory is usually volatile. Furthermore, files are usually read and written using system calls, and memory is accessed by default using virtual memory.

The Andrew File System (AFS) was designed as a file system for between 5,000 and 10,000 workstations at Carnegie Mellon University [Howard et al., 1988]. Compared to NFS [Nowicki, 1989], AFS provides a much better cache consistency protocol. The drawback is that the communication between an AFS client and server is stateful instead of the stateless protocol used by NFS. Frolic improves file replication in a large-scale system using clusters [Sandhu and Zhou, 1992].

AFS has been used to provide access to WWW documents [Spasojevic et al., 1994]. The main advantages of AFS over the normal HTTP protocol are: (1) transparent file migration and replication, (2) better caching, and (3) improved security.

The main drawback of the distributed shared memory and distributed file system approaches is that all applications that share a file or a memory region have to agree on a common representation, and a common policy for synchronizing accesses. For example, under UNIX, the layout of a mailbox is shared between the various mail transport agents, mail user agents, and even news readers. Currently, the trend is to use the POP3 [Myers and Rose, 1996] and IMAP [Crispin, 1996] protocols instead of a shared file to access mailboxes over the Internet.

(Group) Communication

The Internet is currently the prime example of a worldwide system that is based on explicit communication. The Internet is mostly used for point-to-point communication. Multicast protocols are defined but are not in widespread use.

A system that makes extensive use of (software) multicast is ISIS [Birman and van Renesse, 1994]. The ISIS-system organizes processes in groups. Communication between processes in a group is ordered by ISIS based on causal relationships between messages. For example, if a process receives a multicast message A and subsequently sends a point-to-point message B, then the process that receives message B will receive message A before it receives message B. Causal multicast is used to implement atomic multicast using a token holder which multicasts the order in which messages should be delivered [Birman et al., 1991]. It is important to note that ISIS is used to order all communication between the processes in a group. This in contrast to many other systems (including Globe) that order data in a single communication channel or in a single multicast group, but not between different communication channels.

The disadvantage of systems based on explicit communication is that they do not provide replication transparency. Instead, they require the (application) programmer to build replication and consistency protocols on top of the offered message passing primitives.

Mobile Code

Most of this thesis deals with the movement of data from one address space to another, either explicitly using communication primitive or implicitly using shared memory, shared objects, or distributed file systems. An alternative approach is to move computations from one address space to another. We can distinguish two kinds of mobile code [Fuggetta et al., 1998]. The first kind moves only a program from one address space to another. The second kind moves (part of) a process (i.e., a program plus some amount of state).

The first kind of mobile code is frequently used to download user interfaces into clients or to upload complicated queries into servers. For example, Javascript is frequently used to make WWW pages more interactive. Another example is the NeWS system, which allowed programs to upload scripts into a windowing system [Gosling et al., 1989]. SQL queries are an example of simple scripts that are sent by clients to database management systems.

The second kind of mobile code is commonly referred to as mobile agents. In these systems a more or less independent process roams the network. An example of such a system is Agent Tcl (later renamed to D'Agents) [Gray, 1997]. Agent Tcl is, of course, based on Tcl [Ousterhout, 1989].

6.4 Replication

Replication of data is common in (distributed) shared memory systems. At the hardware level, the use of caches causes data to be replicated. Cache-consistency protocols are needed to ensure that DMA controllers operate on current data. In multiprocessor-systems, data can be replicated in caches that belong to different processors. At this level, different memory coherence models are used to trade consistency for performance [Tanenbaum, 1995].

Distributed shared memory implementations can use various techniques to implement memory-coherence models [Stumm and Zhou, 1990]. An interesting consistency model for distributed systems is Local Consistency (LC) [Ahamad and Kordale, 1999]. LC combines a sequential consistency model with efficient updates to the state of a collection of objects. Furthermore, a client gets some control over how quickly updates performed by other clients become visible. The model focuses on the consistency of various objects that are cached by clients. For updates the model assumes that for every object there is one server that accepts update requests.

Two implementations are described. The first implementation assumes a single server that stores the state of all objects. For each client, the server maintains a list of objects that are cached by the client and for which the client no longer has an up-to-date copy of the state of the object. Whenever a client communicates with the server, it receives an invalidation list. This approach allows objects to be updated without explicitly updating or invalidating client caches. Clients that require up-to-date copies can poll the server for invalidation information. Other clients can simply continue with old, but consistent, information.

The second implementation uses timestamps. In this implementation, objects can be distributed over multiple servers. Along with a copy of the state of an object, a server returns two timestamps. The first timestamp is the time that the object was last modified, and the second timestamp is the current time at the server. Whenever an object receives a new copy of the state of an object, it invalidates old information in its cache that it received before the current object was last modified. This approach requires synchronized clocks on the servers. A version of the algorithm that uses logical clocks is also described in [Ahamad and Kordale, 1999].

It is possible to integrate consistency models with synchronization primitives. Examples of this approach are weak consistency [Dubois et al., 1986], entry consistency [Bershad et al., 1993], release consistency [Gharachorloo et al., 1990], and lazy-release consistence [Keleher et al., 1992]). The advantage of these techniques is that memory coherence is ensured when entering or leaving a critical section and not on every read or write operation. Weak memory-coherence models can be extended by taking the semantics of operations into account. Epsilon serializability allows a limited amount of inconsistency [Ramamritham and Pu, 1995; Kamath and Ramamritham, 1993]

A second approach to data replication is the use of group communication. Using this approach, high-level operations are multicast to all processes that have a replica of the state. These processes update their state according to the messages they receive. Logical and physical clocks can be used to provide either a partial or a total order on message delivery [Lamport, 1978]. The ISIS-system provides message ordering for a collection of processes [Birman et al., 1991]. There is some controversy whether this kind of (causal) delivery should be provided at all and whether this should be done at the application level or as part of a standard library. A positive answer to the first question can be found in [van Renesse, 1993]. This kind of functionality should be provided at the application level (instead of as part of the system or a standard library) for four reasons: (1) to avoid unrecognized causality, (2) to group multiple operations, (3) to express the desired semantic ordering, and (4) to gain efficiency [Cheriton and Skeen, 1993]. A response that argues in favor of the ISIS approach can be found in [Birman, 1994].

Some systems use anti-entropy protocols to propagate updates to all replicas of an object. In anti-entropy protocols, replicas periodically send updates to other replicas. Eventually, an update reaches all replicas. These replication protocols provide eventual consistency. Two examples of systems that use this approach are REFDBMS

[Golding, 1992b] and Bayou [Petersen et al., 1997]. Both systems provide a total order for all stable updates. The two systems use different mechanisms for providing a total order on all updates. Bayou uses a centralized approach which uses a sequencer to assign sequence numbers to updates. On the other hand, REFDBMS uses a simple timestamp protocol, which assumes a synchronous network.

Both systems rely on the application to deal with conflicting updates. For example, one of the applications that runs on top of Bayou is a distributed calendar manager [Terry et al., 1998]. In this application, conflicting updates result in appointments for a single that overlap in time, or two meetings that are scheduled in the same room at the same time. In other systems, a user who tries to schedule an appointment that conflicts with another appointment gets instant feedback from the system. However, in these systems, it takes a long time for a message propagate (ranging from several hours to more than one day). The solution is to make the application responsible for canceling one of the conflicting meetings.

Object-based approaches provide an abstraction on top of the two previous approaches. For example, different approaches to implement the programming language Orca are compared in [Bal et al., 1992b]. It is also possible to use object-based approaches to extend other mechanisms. For example, VDOM (Versioned Distributed Object Memory) provides the programmer with different versions of the state of an object [Feeley and Levy, 1992]. Because each version is immutable, it is possible to have concurrent read and write accesses to an object.

The replication architecture that is offered by BOAR is similar to the replication architecture of Globe [Brun-Cottan and Makpangou, 1995]. Three objects, “access objects,” “consistency managers,” and “replicas” correspond to respectively control objects, replication objects and semantics objects in the Globe architecture.

The main difference between the two architectures is that Globe provides support for condition synchronization and replicated method invocations. These two features are not supported by the BOAR architecture. Instead, the BOAR architecture explicitly supports partitioned state and uses locking to support fine-grained concurrency control. Methods that operate on different parts of the state can be executed in parallel. Parallel execution of commutative methods is also supported. Finally, BOAR can group a collection of method invocations in a single “activity.”

6.5 Persistence

Section 3.3 showed that persistence can be provided at different layers. At the operating-system level, it is possible to provide all data storage, including address spaces, with persistence. This approach is called orthogonal persistence. Grasshopper is a system that provides orthogonal persistence [Rosenberg et al., 1996]. Grasshopper provides containers as a storage abstraction, and loci as an abstraction for the registers of a processor. Each locus is associated with a host container. A locus can ask the kernel to make a snapshot of the current state.

Some arguments against orthogonal persistence can be found in [Cooper and Wise, 1996]. Arguments include the possibility of a spaghetti of persistent object references, the problem of providing fine-grained access control, pointer size, and data compression. The authors advocate the use of type-orthogonal persistence. This means that all objects can be made persistent. However, persistent objects are grouped into larger, coarse-grained objects. Persistent objects can only refer to other persistent objects that are stored in the same coarse-grained object.

One level higher are systems that provide persistent objects at the language level. The programming language E extends C++ with (among others) persistent objects [Richardson et al., 1993]. Persistent objects are declared with the storage class persistent. In addition, E supports collections of persistent objects. E provides an iterator mechanism that can be used to examine all objects in a collection. This frees the programmer from the responsibility to maintain indexes that refer to all persistent objects. The EXODUS Storage manager is used by the run-time system to actually store persistent objects. A drawback of E, in the area of naming, is that variable names in the program are used to name objects in the database. The programmer has to avoid writing different modules that use the same variable names for persistent objects, because that will cause conflicts in the database. Compared to Globe, the programmer is not given any control over the actual storage of the persistent state (data representation, or when changes are written to disk).

Thor is an object-oriented database system designed for use in a distributed environment [Liskov et al., 1996]. It provides type-safe sharing and heterogeneity. Type safety is provided by implementing objects using a new programming language Theta. Client code that is written in unsafe languages (such as C), is run in separate address spaces. In the (second) implementation of Thor, a hybrid between page and object caching is used in clients [Castro et al., 1997]. This approach benefits from the low overhead of page caching (many objects at the same time, fixed granularity) for those pages that contain many relevant objects. Isolated objects in pages with few interesting objects are moved to an object store. Another interesting technique is cooperative caching [Adya et al., 1997]. This technique fetches data from nearby clients that contain a cached copy of the data instead of fetching the data from disk.

In Clouds, each object is in a virtual address space [Wilkenloh et al., 1989]. Furthermore, objects are persistent and not tied to processes. The implementation of virtual address spaces for objects is supported by the Clouds kernel.

The persistence model of the BOSS project, described in [Soulard and Makpangou, 1992], is somewhat similar to the persistence model that uses persistent distributed state objects, such as shown in Figure 3.19 on page 76. In the proposed model, to update the state of an object, the client grabs a lock, updates the state, and releases the lock. This approach is not scalable. The problem is that replication of the distributed object is independent of the replication of the state object. In Globe, this problem is avoided by integrating the persistence management with the replication of the state.

The PerDiS project aims to provide a Persistent Distributed Store [Shapiro et al., 1997]. This store provides the user with a shared memory abstraction, similar to the abstraction provided by a distributed shared memory system. PerDiS uses the Larchant garbage collection algorithm to delete unreachable data in the persistent store. As an architecture for a large-scale distributed system, this is a step backward compared to the BOSS project. In an object based system it is relatively easy to provide different interfaces, use different data encoding conventions, etc. within a single object. Hiding the representation of data in a shared memory system is hard.

6.6 Security

We can divide related work in the area of security into three categories. The first category contains low-level algorithms and protocols for encryption, key distribution, signatures, etc. The second category consists of systems that combine a collection of low-level algorithms into a high-level security mechanism. We put work in the area of security policies in the third category.

The first category contains basic encryption algorithms such as DES, IDEA, triple-DES, Blowfish, etc. The third category contains, for example, the Bell-LaPadula model.

The systems in the second category are comparable to the Globe security architecture. Kerberos is an authentication scheme based on shared-key encryption, timestamps and a trusted third party [Steiner et al., 1988]. Additionally, the Kerberos implementation provides primitives to communicate over an encrypted channel. Several network programs such as rsh, rlogin, telnet, and the X Window System have been modified to use Kerberos.

Kerberos has several limitations and weaknesses [Bellovin and Merritt, 1991]. Two examples of weaknesses are the need for secure time services and the opportunity of password guessing attacks. One of the limitations of Kerberos is that it combines a particular encryption algorithm (DES) with a specific authentication protocol. Large-scale systems need a more flexible approach that allows different encryption algorithms and authentication protocols. For example, current computers are so fast that an exhaustive key search has become a practical attack on DES [Wiener, 1996].

A more flexible framework is provided as an extension to the current Internet Protocol [Atkinson, 1995]. This protocol explicitly allows different encryption algorithms, etc. A generic security service application program interface (GSSAPI) is defined to provide a standard way to access the security extensions to the Internet Protocol [Linn, 1997]. An API that provides explicit support for delegation, roles, etc. is described in [Wobber et al., 1994].

A disadvantage of a separate security API is that the application programmer has to deal with another complex issue. Integrating security with a run-time system for a distributed object system reduces the burden of the application programmer. In the Secure Network Objects system, the programmer can select whether he needs an insecure object, an object that is integrity protected, or an object that also provides secrecy [van Doorn et al., 1999]. This choice is made by declaring the object to be a subclass of the generic network object class (NetObj.T), the AuthNetObj.T class, or the SecNetObj.T class. The application programmer remains responsible for access control. The run-time system provides the object with the identity of the principal who invoked the method and the method implementation is responsible for the decision whether to return an appropriate error code or to carry out the execution of the method.

Secure network objects has two disadvantages compared to the security architecture of Globe. The first disadvantage is the lack of choice in encryption algorithms and authentication protocols. The assumption is that the choice is hard-wired in the run-time system. The second problem is that whether an object is secure or not, is reflected in the type of the object. In Globe, the object writer can specify which secure mechanisms should be used by selecting an appropriate security object. The type of the object, that is the set of the interface that it provides, is independent of the choice of a security object.

6.7 Naming

Yeo, Ananda and Koh provide a taxonomy of name systems which is quite different from the classification that was presented in Section 4.1 [Yeo et al., 1993]. They introduce 10 categories with many subcategories. These 10 categories can be grouped into three supercategories:

1. Name services, name assignment, multiple mappings of names, context, name resolution,
2. Name distribution, database,
3. Fault tolerance, access control (and authentication), and communication mechanism.

The first supercategory classifies the name space, the second supercategory describes how the name service is distributed and how data is stored. The third supercategory classifies some implementation aspects of a naming system. They recognize a centralized and a distributed model for name distribution. Only the distributed model makes sense in a wide-area distributed system due to independent network failures and for scalability reasons.

Compared to Section 4.1, they ignore the issue of location transparency, whether a name has to be human readable or not, the use of unique object identifiers, and the level of abstraction of names. The need for unique object identifiers is argued in [Wieringa and de Jonge, 1995].

A general discussion of naming in distributed systems can be found in [Needham, 1993]. Issues are human-sensible names, unique identifiers, pure names, binding, consistency, and scaling.

The naming service that is described in [Cheriton and Mann, 1989] uses three levels of naming: global, administrational, and managerial directory systems. These levels have different performance, reliability, security, and administrative requirements. The global level contains names for independent organizations that are assumed to be somewhat distrustful with respect to each other. The administrational directories are organized hierarchically with respect to security issues. Furthermore, directories at this level should remain accessible even when the global directory fails. A directory at the lowest level (the managerial level) provides access to objects that are stored in a single object manager. In this system, servers for the lowest level of the naming system are integrated with the object servers for the objects they refer to. In contrast, in Globe the name service is separate from object storage.

Clients maintain a prefix cache for managerial directories. The prefix cache maps a name to a combination of a specific object managers and a directory identifier. Multicasting is used to find new mappings for local managerial directories. Special liaison servers are used for names that are outside of the local administration. Due to the use of multicasting, it is necessary for directories at the administrational level to explicitly return failure indications for directories that they do not serve. For other names, the appropriate server returns a reply. The main advantage of this approach is fault tolerance. The use of multicasting ensures that available servers can be reached even when higher-level servers are down. The global directory is implemented using a high degree of replication and gradual propagation of updates.

Note that this design assumes that every object knows its name. A simple renaming of a directory requires all objects and directories below that directory to be informed of their new names. Another issue is lack of location independence at lower levels which restrict the placements of an object's replicas.

The name service just described provides objects with only one name. Neuman argues in [Neuman, 1992a] that the naming system should be used to organize resources. To support organizing resources, Prospero provides two kinds of special directories: directories that act as filters and directories that provide the union of two underlying directories [Neuman, 1992b]. In addition, each user is provided with his own part of the name space to maintain a collection of references to interesting documents or directories.

Note that the World Wide Web and many operating systems also provide each user with his own part of the name space. However, a UNIX user can use only symbolic links to refer to files on other UNIX file systems. Furthermore, operating system name spaces are in general limited to a single machine, or sometimes, a single organization. Both the World Wide Web and Prospero provide a worldwide name space. Unfortunately, in the World Wide Web references to other documents are part of free-form web documents, instead of structured directories. This makes the use of general purpose filters and unions impossible.

Another interesting technique that allows users to structure their own name space is the use of skeleton directories [Rao and Peterson, 1993]. A user can use skeleton directories to change his view of the (global) name space by mounting a file system on top of a directory that belongs to someone else. However, the effect is only visible in the user's view of the name space. In a UNIX system, this technique would allow users to create a view with extra files in the /bin directory by mounting a union directory on top of the real /bin. Plan 9 provides a similar feature, but in this case all processes in a process group share a view [Presotto et al., 1992].

The Hobbes project advocates a binding architecture that is quite different from the one used in Globe. The key difference is that Globe uses a few types of object references, such as pathnames, object handles and contact addresses. These object references are used in sequence in a standard, 4-step binding protocol. In contrast, the binding architecture that is described in [Shapiro, 1994] supports an arbitrary number of object reference types. During binding, a remote procedure call is made by the run-time system to the target of the reference. The target verifies that it can supply the interface that is desired by the client, and it returns a reference to a class object that implements the appropriate interface and can act as a proxy.

An advantage of the Globe mechanism is that object references can always be resolved to the level of a contact address. At the level of a contact address, it is possible that protocol identifier is not understood by the client and that binding fails. Because there are only three types of object references in Globe, they can be passed easily between address spaces, for example as arguments to method invocations. When the Hobbes approach is used, it is possible that a type of reference is passed which is not understood by the recipient.

Closely related to the location service, are systems that try to provide a single 64-bit address space. An example of such a system is Opal, which implements a lightweight shared object system [Chase et al., 1992]. Objects are named by 64-bit pointers. Opal defines a single virtual address space that maps all data in a small-scale distributed system, including persistent data.

Location services are commonly found in mobile phone systems. Systems that are currently operational, for example GSM, use a home-based approach. A home-based approach is also used for mobile hosts on the Internet [Perkins, 1996]. An overview of location service approaches can be found in [Pitoura and Samatas, 1998].

For example, a collection of forwarding pointers can be maintained in a tree such that pointers close to the root of the tree are updated less frequently than pointers near the leaves of the tree [Awerbuch and Peleg, 1991].

An interesting caching technique for locating mobile phones is described in [Jain, 1996]. In Globe terminology, this technique does not cache the actual contact addresses for objects, but caches where a particular contact record is stored in the location service. The advantage of this technique is that a contact address can be updated without invalidating the cache. The obvious disadvantage is that at least one RPC is necessary to look up the current contact address. This technique has been discovered independently during the design of the Globe location service [Hauck et al., 1996]. A home-based system with a hierarchical search algorithm can be found in [Wang, 1993]. In this algorithm, the root of registration is the least-common-ancestor of the current location and the home base.

6.8 Garbage Collection

A survey of distributed garbage collection techniques can be found in [Plainfossé and Shapiro, 1995]. A similar survey that also includes a description of local garbage collection can be found in [Abdullahi and Ringwood, 1998]. The second survey recognizes three different classes of algorithms: those based on reference counting, based on reference listing, and tracing garbage collectors. A disadvantage of the reference counting and reference listing techniques is that they cannot reclaim garbage that contains cycles.

Tracing garbage collectors use three different techniques: tracing with timestamps, centralized garbage collection, and tracing within groups. Timestamps can be used to detect collections of unused objects. Each process periodically starts a marking phase and assigns a new timestamp to all objects that are reachable from its root set. These timestamps propagate to other address spaces. The marking phase is complete when the timestamp has been propagated to all reachable objects in remote address spaces. All processes cooperate to compute the timestamp of the oldest complete marking phase. All objects with older timestamps can be deleted. A problem with this technique is that global agreement on the oldest complete marking phase is hard to achieve, and is nonscalable. Furthermore, one crashed address space prevents garbage collection in the entire system. A (nonscalable) alternative is to collect information about all references between objects in one process to compute which objects are unreachable from that information.

Tracing within groups is a technique to increase the scalability of tracing with timestamps. Instead of computing the oldest complete marking phase in the whole, a group of process cooperate to trace objects that are not referenced from other address spaces outside the group. All unreachable cyclic garbage in the group of processes can be collected. Unfortunately, heuristics are needed to determine the set of processes that should form a group. Ultimately, a group that contains all processes in the system is needed to be sure that all garbage is collected. We can conclude that the garbage collection techniques that are described in this survey are either incomplete (reference counting and reference listing), or are not scalable.

The Larchant garbage collector collects replicated objects in a distributed system [Ferreira and Shapiro, 1996]. Objects are grouped into bunches for the purposes of caching and garbage collection. The algorithm is based on reference listing: each object maintains a list of all objects in other bunches that refer to it. These references are called scions. Intra-bunch garbage collection is done separately for each replica using a single address garbage collector. Intra-bunch garbage collection is implemented by creating and deleting scions. A bunch that has no scions is unreachable, and can be deleted.

Disadvantages of using this algorithm in a large distributed system are twofold. Firstly, it is in general not possible to implement reference listing in such as system. The number of references to a single object becomes a scalability problem. The second, more serious problem, is that the garbage collector does not collect all cycles. A simple heuristic is used to determine what potential cycles have to be inspected.

Nonscalability of the tracing garbage collectors based on timestamps can be solved by combining a copying garbage collector with reference listing [Hudson et al., 1997]. This garbage collector, called DMOS (Distributed Mature Object Space), can be explained using trains and cars as a metaphor. All objects are located in cars which may contain more than one object. A car is located in a single address space. In this model, objects are located in a single address space and can be accessed remotely. Cars from multiple address spaces are grouped into trains. Each car belongs to exactly one train. Trains are timestamped.

The garbage collection algorithm works as follows. In each garbage collection cycle, all objects that are referenced by other objects are copied to new cars. The objects that are not copied are destroyed. Objects may be copied to a new car in the same train or to a car in a different train. Objects that are reachable from the local root set are copied to a new train. Reference listing is used to check whether an object is reachable from an object in a train that is created later than the current train (i.e., an object in a younger train). Objects that are reachable from objects in younger trains are copied to a car in one of the younger trains. What remains are unreachable objects, objects reachable from older trains, and objects reachable from other cars in the same train.

A distributed termination detection algorithm, in this case based on token passing, is used to verify that the objects in a train are not referred to by objects in other trains. If that is the case, the entire train is deleted. Eventually, the oldest train in the system will be deleted.

Unfortunately, this is a copying garbage collector. The algorithm depends on the property of copying garbage collectors that garbage is left behind after all live data is copied. In a system where many objects are referenced by directory objects, this requires the entire name service to be copied at regular intervals to reclaim deleted files. This is not very efficient in a system with a large number of persistent objects.

A garbage collection algorithm that is similar to the garbage collection for Globe in that it also tries to find a path to the root is described in [Fuchs, 1995]. This algorithm uses a coloring algorithm, similar to those used in mark-and-sweep garbage collectors, to color live objects and to delete dead ones. This algorithm has two important drawbacks. First, the algorithm does not store a path to the root. This means that an object has to run the coloring algorithm each time it needs to verify its reachability. Second, two objects that have a common parent cannot verify their reachability in parallel.

An improvement over [Fuchs, 1995] can be found in [Maheshwari and Liskov, 1997]. For each (distributed) object, this algorithm keeps track of the distance to the root set. A member of the root set has distance zero and an unreachable object should get distance infinity. In practice, objects start counting to infinity when they become unreachable. The algorithm uses a cut-off distance as a heuristic to determine which objects are potentially unreachable. For objects that are potentially unreachable, their ancestors are examined to see if their distances are also above the cut-off distance. An object is unreachable if all ancestors are above the cut-off distance.

To maintain consistency, the garbage collector requires the application to report (to the garbage collector) any new references that may compromise the safety of the garbage collection algorithm. In other words, the activities of an application program are closely tied to the garbage collector. The advantage of this approach is that the storage requirements are much less than the garbage collector for Globe. However, it is not clear whether a strong connection between applications and the garbage collector is feasable in a world-wide distributed system with highly replicated objects.

Globe uses reference counts to collect local objects within one address space. An overview of local garbage collection techniques can be found in [Wilson, 1992, 2000].

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis describes a novel architecture for wide-area distributed systems based on distributed shared objects. This architecture contains a separate location service to support location and replication transparency. The architecture for local objects includes a local, per process name space which is integrated with a wide-area naming service. A new garbage collection algorithm provides distributed detection of unreachable objects.

The construction of a distributed shared object is simplified by separating the implementation of the replication protocols from the implementation of the object specific semantics using communication, replication, control and semantics objects. The interface between the control and the semantics object allows different replication strategies. Blocking method invocations are also supported. This implementation architecture is completed with support for persistence and security.

Chapter 1 listed a number of requirements. This section describes how they are met by the Globe architecture.

- Scalability

  The Globe architecture contains two centralized components: the worldwide root of the name space and the (root of the) location service. The scalability of a single worldwide root has been demonstrated in the Domain Name System. The contents of the root of the name space changes infrequently and can be replicated easily.

  The scalability of the location service is more complicated. The location service uses partitioning to distribute the load over multiple servers. Furthermore, the use of the hierarchical search algorithm and caching should avoid many lookup operations on the root of the location service. The use of home-based location servers in conjunction with a distributed search tree is another technique to reduce the load on the root of the location service. We expect that in practice the location service has to be monitored closely to be able to take measures to avoid overloading parts of the location service.

The scalability of the garbage collection algorithm has been argued already in Section 4.5.11. The Globe architecture does not contain any other components that are likely to be nonscalable. However, individual implementations of distributed shared objects may be nonscalable. For example, the use of a client/server replication object to achieve strong data consistency may lead to scalability problems when the object becomes popular.

- Flexibility

  A large part of the Globe architecture is designed to provide flexibility. At the lowest levels, run-time interfaces are used to decouple separately compiled components. The local name space provides a flexible mechanism to refer to local objects. The use of protocol identifiers combined with dynamic loading of class objects allows existing applications to benefit from new implementations of distributed shared objects.

  Each object encapsulates its own implementation. In particular, implementations will differ with respect to replication strategies. A distributed shared object can be implemented in various ways: with many replicas and a voting mechanism, or partitioned, or just stored in one place. Distributed shared objects that look to the user as if they belong to the same “class” can have different implementations. The implementation of a distributed object, in terms of communicating local objects, can use arbitrary communication patterns and encapsulates data placement, replication, etc.

- Security

  Globe contains a minimal security architecture. This architecture is comparable to security architectures for other distributed systems. Future research is needed to determine whether this architecture can protect highly replicated objects.

- Uniform model

  The Globe architecture uses distributed shared objects to allow processes to communicate. The user of a distributed shared object sees a model that includes naming, contact addresses, and local objects. The implementation of the distributed shared object is completely hidden. The implementor of a distributed shared object writes a semantics object and has to select an appropriate replication object to complete the distributed shared object.

  For special distributed shared objects, the implementor can write local objects that provide object handles, contact addresses, etc. directly without using a replication object. The local object model provides a uniform environment for implementing distributed shared objects on different platforms.

- Location and replication transparency

  The location service provides most of the support for location and replication transparency. Location transparency is provided through the resolution of location-independent object handles. The location service supports replication transparency by associating multiple contact address with a single object handle. Replication transparency is completed through the use of dynamically loaded object implementations. This allows each distributed shared object to choose an appropriate replication strategy.

- Autonomous administration

  Most of the Globe architecture allows autonomous administrations (i.e., organizations can independently administer their computers, networks, etc.). The three exceptions that require global cooperation are the name service, the location service, and the allocation of protocol identifiers. In the garbage collection algorithm each object independently verifies whether it is reachable from the root set or not. The use of 256-bit interface identifiers avoids the need for central coordination to allocate identifiers for new interfaces.

  The name service requires worldwide coordination for the root. Again, the Domain Name System can be used as an example of name-space delegations to individual organizations. The location service requires a large amount of central coordination. Note that home-based extensions to the location service provide some amount of delegation, which allows a trade-off between independence and location transparency.

  In the current implementation, protocol identifiers are ASCII strings. This approach requires cooperation to avoid duplicates. An alternative is the use of protocol identifiers that are based on object handles. Object handles are, by definition, worldwide unique and can be used as protocol identifiers without global coordination. The advantage of using object identifiers instead of just unique identifiers, is that special protocol objects can be introduced which contain descriptions of the protocol and, optionally, references to implementations of the protocol.

- Interoperability

  The Globe architecture does not provide explicit support for interoperability. Distributing object implementations ensures interoperability but is unattractive from a security point of view. Documenting protocols in protocol objects is one approach to solve this problem, but has not been implemented. The Internet community has a lot of experience with informal protocol descriptions in the form of RFCs (Request For Comments).

Interface definitions are another area where interoperability is required. In this case, distributed shared objects have to provide interface that are supported by applications (and vice versa). In general, a new implementation of a representative for a distributed shared object has to implement the right protocols to connect to the distributed shared object and has to implement the right interfaces to be useful in applications.

- Backward compatibility

  The general approach to backward compatibility has been described in Section 5.1. The Globe architecture does not contain any special features for backward compatibility with other systems.

The prototype implementation has provided the following insights:

- Flexibility

  In Globe, it is possible to write a semantics object once and combine it with many different replication objects. These objects can coexist in a single system. Furthermore, applications do not have to be recompiled.

- Pop-up threads

  Pop-up threads are used in Globe to deliver messages that arrive over the network to passive objects. Disadvantages of pop-up threads are the need for concurrency control and the difficulty of limiting the number of threads in the system.

  An alternative model would be to implement objects as state machines. This would give semi-active objects that react to the reception of events (messages). Without blocking primitives, the need for multiple threads would be reduced. With fewer threads, the need for concurrency control can be reduced. For example, see [van Renesse, 1998] for an approach to concurrency without threads.

- Interface design

  There is a trade off between a single (large) interface or multiple, smaller interfaces. For example, compare the gc interface with the distrObj, distrCtrl, and persist interfaces. Multiple interfaces are often harder to use. In many cases, multiple interface pointers are needed to perform a task. The advantage is that a new interface, such as the persist2 interface, can be introduced without affecting other interfaces.

- Latency

  An object-based system quickly leads to a situation where an application needs to access multiple objects to perform its task. Special care should be taken that the latency for accessing objects does not become a bottleneck.

  For example, compare the Internet news protocol NNTP with an object-based solution. Using NNTP, only one connection is established to access many news groups and to transfer many new articles. Using separate objects for different news groups, and possibly for different new articles requires more connections and introduces more overhead.

7.2 Future Work

More research is needed in many areas of the Globe architecture. Some of those areas are listed below.

Local Objects Globe shares its local object model with Paramecium [van Doorn et al., 1995]. It would be interesting to see whether a Globe program can run directly on top of a Paramecium kernel.

Replication More research is needed to support partitioned state and the composition of distributed shared objects [Bakker et al., 1999]. Support for partitioned state in a distributed shared object allows different parts of the state to be replicated in different places. It should be possible to invoke methods that operate on one or more partitions of the state. The main reason for partitioning is efficiency: less state has to be sent to new replicas, and fewer replicas have to receive updates to the state. Another reason for partitioning is security: different parts of the state may have different security requirements. The main problem is the implementation of methods that operate on more than one partition of the state.

Composition of distributed shared objects is the opposite of partitioning: independent distributed shared objects are treated as one object. Efficiency is also one of the reasons for the composition of distributed shared objects. A distributed shared object is a rather heavy-weight construction and the composition of multiple distributed shared objects allows multiple objects to share the cost. Object management may be another reason. For example, it can be desirable to have a single replication strategy for a collection of objects.

Interoperability is another area where more research is needed, both in the area of local interfaces and the area of communication between different parts of a distributed shared object. For example, standards may be required for marshaling the state of an object or the encoding of method invocations.

Techniques that hide the latency of binding and of method invocations have to be studied. For example, currently binding to a distributed shared object and invoking a method on the object requires operations on directory objects, the location service, setting up a connection to the distributed shared object and sending either a request for a method invocation or a request for the state of the object. In theory, all of this can be reduced to sending a request to the name service to invoke a particular method on a named object. More research is needed to determine whether this is feasible and whether the gains are high enough to warrant the increased complexity of the system.

Persistence Standard encodings may be necessary for the persistent state of a replica to ensure that future versions of the implementation of a distributed shared object are capable of operating on old copies of the persistent state of the object.

Fault tolerance, which is mostly ignored in this thesis, is related to both persistence and replication. A persistent replica that recovers from a crash has to verify that its persistent copy of the state of the object is not obsolete compared to the state of other replicas. Fault tolerance in replication protocols requires dealing with crashed nodes and network partitions.

Security More research is needed in the area of security, both within a single distributed shared object and for the system as a whole. Especially the location service has interesting security requirements. The security implications of the garbage collection algorithm require more research (it may be possible to use a denial of service attack on the location service to delete distributed shared objects that belong to another user).

Resource Control, Allocation, and Tracking Factories and object repositories create and manage distributed shared objects and persistent replicas. The garbage collection algorithm deletes unreachable objects. More research is needed to determine how users and applications find the appropriate factories and object repositories (brokers or factory finders). Another issue is how users can find out about objects they sponsor and how much resources are used by those objects.

One approach is to introduce a domain structure. Domains encapsulate hierarchical collections of resources. Domains contain resource managers which allocate resources to distributed shared objects.

We can introduce a yellow pages services to find resource managers. Queries are sent to the yellow pages service which consist of a resource category, some constraints and some selection criteria. Resource categories include factories for specific objects, services like the location service, or hardware like printers, or guaranteed communication bandwidth. A constraint might be that a printer has to be able to accept PostScript as input.

This yellow pages service can use a similar structure as is used for the distributed search tree in the location service: the local domain should be queried first, followed by larger domains, etc.

Appendix A

Interface grammar

The syntax of the interface and object definition language is an extension of the C declaration syntax. In general, lexical elements (tokens) are the same as in C. The main classes of lexical elements are: identifiers, numbers, strings between double quotes ("), and the ellipsis (...). The following standard C keywords are used: char, const, double, extern, int, long, short, struct, typedef, union, unsigned, and void. New keywords that were added are: callback, composition, interface, new, object, override, ref, and tmpref.

One special construction is a line literal. A line literal consists of optional white-space, the at character (@), some text, and a terminating newline character. The text after the at character is copied directly to the output file without further interpretation.

The grammar consists of C declaration syntax with some additions for interface, object, and composite-object definitions.

translation-unit:
    external-declaration
    translation-unit   external-declaration

external-declaration:
    declaration
    compobj-def
    Line-literal

declaration:
    basetype   declarator-list$_{opt}$ ';'

declarator-list:
    declarator
    declarator   ','   declarator-list

declarator:
    Identifier
    ptr-decl declarator
    ’(’ declarator ’)’
    declarator ’(’ argument-list ’)’
    declarator ’[’ expression ’]’

ptr-decl:
    ’*’ ptr-qualifier-list$_{opt}$

ptr-qualifier-list:
    ptr-qualifier
    ptr-qualifier ptr-qualifier-list

ptr-qualifier:
    typespec

argument-list:
    argument
    argument ’,’ argument-list

argument:
    basetype abstract-declarator$_{opt}$
    basetype abstract-declarator$_{opt}$ ’=’ expression
    Ellipsis

abstract-declarator:
    ptr-decl abstract-declarator$_{opt}$
    abstract-declarator ’[’ expression$_{opt}$ ’]’
    abstract-declarator ’(’ argument-list ’)’
    Identifier
    ’(’ abstract-declarator ’)’

expression:
    Number

basetype:
    typespec basetype$_{opt}$
    storage-class basetype$_{opt}$
    type basetype$_{opt}$

typespec:
    const
    ref
    tmpref

storage-class:
    extern
    typedef

type:
    void
    long$_{opt}$ double
    int-type
    typedef-type
    struct-type
    union-type
    interface-type
    object-type

int-type:
    int-sign int-size$_{opt}$
    int-size

int-sign:
    unsigned

int-size:
    char
    int
    short int$_{opt}$
    long long$_{opt}$ int$_{opt}$

struct-type:
    struct Identifier struct-def$_{opt}$
    struct-def

struct-def:
    '{' declaration-list '}'

union-type:
    union Identifier union-def$_{opt}$
    union-def

union-def:
’{’ declaration-list ’}’

typedef-type:
Identifier

interface-type:
interface Identifier interface-def$_{opt}$

interface-def:
’{’ method-list ’}’ ’=’ interface-id

interface-id:
String
Number

method-list:
basetype declarator ’;’ method-list$_{opt}$

object-type:
object Identifier object-def$_{opt}$

object-def:
’{’ interface-list ’}’

interface-list:
callback$_{opt}$ interface Identifier ’;’ interface-list$_{opt}$

compobj-def:
composition Identifier ’’ comp-el-list ’’ ’;’

comp-el-list:
comp-el comp-el-list$_{opt}$

com-el:
interface-impl
sub-object
override

interface-impl:
interface Identifier dot-method$_{opt}$ inf-init$_{opt}$ ’;’

dot-method:
    ’.’   Identifier

inf-init:
    ’=’   Identifier   ’.’   Identifier   dot-method$_{opt}$

sub-object:
    object   Identifier   ’=’   new   String   ’;’

override:
    override   Identifier   ’/’   String   ’=’   Identifier   ’;’

Appendix B

Interface Definitions

This appendix contains the list of interfaces used in the prototype of Globe. The interfaces are shown in an abstract form as is shown in Figure B.1. Each interface starts with the name of the interface and its interface identifier. The interface identifier is a 256-bit unique identifier, which is shown as a string of 64 hexadecimal digits. For layout reasons, the string is split into two strings, implicitly assuming C-style string concatenation. A sequence of method descriptions follows the interface identifiers. Each method description includes the name of the method, a description of the purpose of the method, a list of parameters, and the return value. Parameters can be input parameters (passed by value) or output parameters (passed by reference). In the actual interface definitions, output parameters are declared as a pointer to a location.

interface inf		
id ”00000000000000000000000000000000”		
”00000000000000000000000000000000”		
method	m1	Method 1
in	p1	Parameter 1
returns		v1
method	m2	Method 2
in	p2	Parameter 2
out	p3	Parameter 3
returns		v2

Figure B.1. Abstract interface description

B.1 Local Objects

The stdObj and objMgt interfaces are defined in <p/inf/stdobj.inf>. The getInfList method in the objMgt returns the list of all interface identifiers supported by an object, and for each interface identifier an ASCII name for that interface. The stdObj interface is described in Section 5.3. The implementation of the objMgt interface is generated by the object generator (see Section 5.11.1).

interface stdObj		
id 3		
method	init	Initialize the object
in	ctx	Object's name-space context
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	cleanup	Cleanup before destruction
in	how	What kind of cleanup: full (0) or partial (1)
returns		Nothing
method	getinf	Return a pointer to an interface
in	infid	Identifier of the desired interface
out	err	Reason for failure
returns		Pointer to the interface
method	relinf	Release an interface
in	infp	Pointer to interface
returns		Pointer to object's class object if the object is to be deleted

struct infName	
infName	Pointer to interface name (ASCII string)
infId	Interface identifier

interface objMgt		
id ”39690000bb45006935093566000ebb46”		
”000000000000000000000000c01fe7ae”		
method	getInfList	Return a list of interfaces supported by the object
returns		List of infName structures

The class interface is defined in <p/inf/class.inf>. This interface is provided by all class objects. The create method creates a new instance and returns either a pointer to the newly created object's standard object interface or null in case of an error. The destroy method destroys an instance of a class. This method is only invoked by the p_inffree function. The class interface is described in Section 5.3.

interface class		
id "3969000084e7444232a80eba000b84e8" "000000000000000000000000c01fe7ae"		
method	create	Create a new object
out	err	Reason for failure
returns		Pointer to the standard object interface of the new object
method	destroy	Destroy an object
in	obj	Pointer to standard object interface of the object to be destroyed
returns		nothing

B.2 Distributed Objects

The distrObj interface is defined in <globe/inf/distr.inf>. The init method connects a local object to an existing distributed shared object. This method is invoked during binding. The gethandle returns the distributed shared object's object handle. This interface is described in Section 5.4.

interface distrObj		
id "396900009fe3139832a4246900099fe4" "000000000000000000000000c01fe7ae"		
method	init	Connect to the rest of a distributed object
in	name	Pathname of the distributed object
in	handle	Object handle
in	addrs	Contact addresses
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	gethandle	Return the object's object handle
out	err	Reason for failure
returns		Object handle or NULL

The distrCtrl interface is defined in <globe/inf/distrCtrl.inf>. This interface is used by factories and object repositories to create new distributed shared objects and to create additional replicas of an existing distributed shared object. The newLink method informs the garbage collection part of a distributed shared object about a new reference to the object. This interface is also described in Section 5.4.

interface distrCtrl		
id ”396900003bd32aa73331350d00033bd4” ”000000000000000000000000c01fe7ae”		
method	create	Initialize local object as a new distributed object
out	err	Reason for failure
returns		Set of contact addresses
method	setHandle	Tell distributed object about its object handle
in	handle	Object handle
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	newLink	Tell distributed object about a new reference from a directory object
in	parentDir	Object handle of parent directory object
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	replicate	Create a new replicate in the current address space
out	err	Reason for failure
returns		Success (0) or failure (–1)

The factory interface is defined in <globe/inf/factory.inf>. This interface is provided by factories and object repositories to allow the creation of new distributed shared objects and replicas. This interface is described in Section 5.8.4.

interface factory		
id ”39690000cc0b1785330899620009cc0c” ”0000000000000000000000000c01fe7ae”		
method	start	Start the factory
in	classname	Class of objects to produce
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	create	Create an object, and register with the name service
in	name	Name in name service
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	createRep	Create a local replica of an existing object
in	name	Name of object
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	deleteRep	Delete a local replica of an object
in	name	Name of object
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	startObj	Activate a persistent replica
in	handle	Object handle of object to activate
out	err	Reason for failure
returns		Volatile contact address for the object
method	gc	Start a garbage collection for all object in the repository
out	err	Reason for failure
returns		Success (0) or failure (–1)

The persist and persistCB interfaces are defined in <globe/inf/persist.inf>. The persistCB is a callback interface for the persist interface to provide a persistent replica with an output stream interface. These interfaces are described in Section 5.6.

interface persist		
id	”396900009c4607fd332061df00089c47” ”000000000000000000000000c01fe7ae”	
method	create	Create a persistent contact address
in	serverHandle	Object handle of object repository
out	err	Reason for failure
returns		List of contact addresses or NULL
method	setCB	Set persistence callback interface
in	pcbi	Callback interface to save state
in	ref	Callback reference to passed when calling callback interface
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	unmarshal	Recreate a previous state from a stream of data
in	state	Input stream interface to marshaled state
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	restart	Restart the replica (after a shutdown or crash)
out	err	Reason for failure
returns		List of (nonpersistent) contact addresses or NULL

interface persistCB		
id	”39690000cf6007ff3320621d0002cf61” ”000000000000000000000000c01fe7ae”	
method	newstate	Create an output stream to save the state
in	ref	Reference (the one passed in the setCB call)
out	err	Reason for failure
returns		Output stream interface or NULL

The persist2 and persist2a interfaces are defined in <globe/inf/persist2.inf>. These interfaces are an alternative to the persist and persistCB interfaces. The main difference is the persist2 interface does not deal with persistent state directly (only persistent contact addresses) and assumes the presence of an object with the state interface to save to state of the replica. These two interfaces are also described in Section 5.6.

interface persist2		
id	”3969000016f9589f33662f67000416fa” ”000000000000000000000000c01fe7ae”	
method	shutdown	Stop accepting new connections and save state
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	create	Create a persistent contact address based on the object repository’s object handle
in	handle	Object handle
out	err	Reason for failure
returns		List of persistent contact addresses or NULL
method	restart	Restart after a shutdown
out	err	Reason for failure
returns		List of transient contact addresses or NULL

interface persist2a		
id	”39690000ed9748f83365f6a80003ed98” ”000000000000000000000000c01fe7ae”	
method	idle	Report that the object is idle
out	err	Reason for failure
returns		Success (0) or failure (–1)

B.3 Replication

The replication interface is defined in <globe/inf/replication.inf>. This interface defines the following constants and four methods. A detailed description of this interface can be found in Section 5.5.2.

States		
RS_START	0x101	Start state
RS_SEND	0x102	Send state
RS_INVOKE	0x103	Invoke state
RS_RETURN	0x104	Return state
Operation types		
RO_READ	1	Read operation
RO_MODIFY	2	Modify operation

interface replication		
id ”396900006970178935b5f9b300056971”		
”000000000000000000000000c01fe7ae”		
method	init	Provide the replication object with the control callback interface
in	ci	Pointer to control interface
returns		nothing
method	start	Start the execution of a method invocation
in	o	Type of operation
returns		Next state
method	invoked	The method has been invoked on the semantics object
in	o	Type of operation
in	blocked	True if the method invocation blocked
returns		Next state
method	send	Send marshaled request and arguments
in	o	Type of operation
in	req	Request packet
out	sp	Next state
returns		Reply packet or NULL

The control interface is defined in <globe/inf/control.inf>. This interface is a callback interface for the replication interface. The handle_request method is called by the replication object to unmarshal a method invocation and to invoke the method on the semantics object. The getState and setState methods, respectively, marshal and unmarshal the state of the semantics object. The control object is described in Section 5.5.1.

interface control		
id ”3969000010f61fe935ab4bee000a10f7”		
”000000000000000000000000c01fe7ae”		
method	handle_request	Execute an operation that arrive over the network
in	req	Packet with marshaled request and arguments
returns		Packet with results or NULL
method	getState	Return the marshaled state of the semantics object
out	err	Reason for failure
returns		Packet with marshaled state
method	setState	Download state into the semantics object
in	data	Packet with marshaled state
out	err	Reason for failure
returns		Success (0) or failure (–1)

B.4 Semantics Objects

The file interface is defined in <globe/inf/file.inf>. This is a simple interface to store some amount of raw data in a distributed shared object. The file interface is described in Section 5.9.

interface file		
id	”3969000054ea16b43291f3b2000354eb” ”000000000000000000000000c01fe7ae”	
method	read	Read data from a file object
in	offset	Offset in file data
in	maxsize	Maximum amount of data to read
out	buf	Buffer filled with data
out	err	Reason for failure
returns		Number of bytes read or failure (–1)
method	write	Write data in a file object
in	offset	Offset in file to start writing
in	buf	Buffer with data to write
in	size	Amount of data to write
out	err	Reason for failure
returns		Success (0) or failure (–1)

The mutex interface is defined in <p/inf/mutex.inf>. This interface can be used to implement a simple lock object. The mutex interface is used in Section 5.5.3 to describe an example semantics object.

interface mutex		
id	”39690000e3ce12f33291dae0000be3cf” ”000000000000000000000000c01fe7ae”	
method	lock	Locks the object
returns		Nothing
method	unlock	Unlocks the object
returns		Nothing

The wwwAttr and wwwSO interfaces are defined in <globe/inf/www.inf>. These interfaces store, respectively, the attributes of a WWW document and any small related WWW documents such as inline images. These interfaces are described in Section 5.9.

interface wwwAttr		
id	”396900005a0d002935126f6a00055a0e”	
	”000000000000000000000000c01fe7ae”	
method	add	Add an attribute
in	attr	Name of the attribute
in	value	Attribute’s value (string)
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	update	Update an attribute
in	attr	Name of the attribute
in	value	Attribute’s value (string)
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	del	Delete an attribute
in	attr	Name of the attribute
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	lookup	Look up an attribute
in	attr	Name of the attribute
out	err	Reason for failure
returns		Value of attribute or NULL
method	list	List attributes
in	detail	List just attribute names(0) or attributes and values(1)
out	err	Reason for failure
returns		List of strings or NULL

interface wwwSO		
id	”396900001e706333351141ad00041e71”	
	”000000000000000000000000c01fe7ae”	
method	add	Add a subobject to a WWW object
in	name	Name of the subobject
in	value	Subobject’s value (raw data)
in	size	Size of the data
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	update	Update a subobject
in	name	Name of the subobject
in	value	Subobject’s value (raw data)
in	size	Size of the data
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	del	Delete a subobject
in	name	Name of the subobject
in	size	Size of the data
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	lookup	Look up a subobject
in	name	Name of the subobject
out	size	Size of the data
out	err	Reason for failure
returns		Buffer or NULL
method	list	List names of subobjects
out	err	Reason for failure
returns		List of names or NULL

B.5 Local Name Space

The ctx interface is defined in <p/inf/ctx.inf>. This is the interface to the local name space. However, this interface is hidden by a library. See Section 5.10.1 for a description of this interface.

interface ctx		
id	”39690000327a25c135167ea1000e327b” ”000000000000000000000000c01fe7ae”	
method	bind	Bind to an object starting in a specific context
in	ctx	Starting context
in	name	Name of object to bind to
out	err	Reason for failure
returns		Standard object interface of the object
method	register	Register an object
in	ctx	Starting context
in	name	Name of context (nil for starting context)
in	obj	Standard object interface of object to register
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	unregister	Unregister an object
in	obj	Standard object interface of object to unregister
returns		Success (0) or failure (–1)
method	override	Install an override
in	ctx	Starting context
in	name	Name of override
in	target	Target of override
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	remove	Remove a context
in	ctx	Starting context
in	name	Name of context to remove
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	context	Look up (or create) a context
in	ctx	Starting context
in	name	Name of context to look up / create
out	err	Reason for failure
returns		Context identifier
method	lookup	Look up a context
in	ctx	Starting context
in	name	Name of context to look up
out	err	Reason for failure
returns		Context identifier

The bind interface is defined in <p/inf/bind.inf>. This interface is provided by objects that implement a name-space delegation. The p_bind library function invokes the bind method in the bind interface with the remainder of the pathname. The method returns either a pointer to the object's standard object interface or null in case of an error. This interface is also described in Section 5.10.1.

interface bind		
id	”39690000518f117a3291d44600005190”	
	”000000000000000000000000c01fe7ae”	
method	bind	Bind to an object
in	name	Object name
out	err	Reason for failure
returns		Pointer object's standard object interface

B.6 Naming

The naming interface is defined in <globe/inf/naming.inf>. This interface provides access to directory objects. Directory objects are described in Section 5.8.3.

interface naming		
id	”396900003d1016f13291f48a00003d11”	
	”000000000000000000000000c01fe7ae”	
method	list	List a directory
out	err	Reason for failure
returns		List of directory entries or NULL
method	lookup	Look up a path relative to a directory
in	path	Path to look up
out	err	Reason for failure
out	rempath	Unresolved part of the path
returns		Object handle or NULL
method	add	Add an entry to a directory
in	entry	Directory entry
in	handle	Object handle
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	delete	Delete a directory entry
in	entry	Directory entry
out	err	Reason for failure
returns		Success (0) or failure (–1)

B.7 Location Service

The following interfaces are related to the location service. These interfaces are described in Section 5.8.1. The locRes interface is defined in <globe/inf/locRes.inf>. This interface provides read access to the location service. The locServ, locAdm and locMgr interfaces are defined in <globe/inf/locServ.inf>. The locServ interface allows objects to be registered with the location service. The locAdm interface provides some administrative functions, such as the registration of location service address managers and a method to trigger a garbage collection cycle. The locMgr interface is provided by address managers and is invoked by the location service to verify to validity of contact addresses and address managers.

interface locRes		
id	”39690000f0d6536334ec3dc30005f0d7” ”000000000000000000000000c01fe7ae”	
method	lookup	Look up the set of contact addresses for an object handle
in	handle	Object handle to look up
out	err	Reason for failure
returns		List of contact addresses or NULL

interface locServ		
id	”39690000a25b643234fdaad40008a25c”	
	”0000000000000000000000000c01fe7ae”	
method	regObj	Register a new object
in	objId	Object identifier
in	addrs	List of contact addresses
in	manager	Object handle of address manager
out	addrId	Address identifier
out	err	Reason for failure
returns		Object handle or NULL
method	regAddr	Register a new address for an object
in	handle	Object handle
in	addrs	List of contact addresses
in	manager	Object handle of address manager
out	err	Reason for failure
returns		Address identifier or NULL
method	update	Update a set of contact addresses
in	handle	Object handle
in	addrId	Address Identifier
in	addrs	List of contact addresses
in	manager	Object handle of address manager
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	delAddr	Delete a list of contact addresses
in	handle	Object handle
in	addrId	Address Identifier
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	delObj	Delete an object
in	handle	Object handle
out	err	Reason for failure
returns		Success (0) or failure (–1)

interface locAdm		
id	”396900007f966a4e34fdaddb00087f97” ”000000000000000000000000c01fe7ae”	
method	regMan	Register an address manager
in	manMan	Object handle of the manager’s manager
in	handle	Object handle of the manager
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	delMan	Delete an address manager
in	handle	Object handle of the manager
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	gc	Perform a garbage collection cycle
out	err	Reason for failure
returns		Success (0) or failure (–1)

interface locMgr		
id	”396900004fb63a8934f29cca00034fb7” ”000000000000000000000000c01fe7ae”	
method	validAddress	Verify a list of contact addresses
in	handle	Object handle
in	addrId	Address identifier
in	addrs	List of contact addresses
out	err	Reason for failure
returns		Valid address list (0), invalid address list (1), or failure (–1)
method	validManager	Verify an address manager
in	handle	Manager’s object handle
in	tries	Number of attempts to bind to the manager
in	secs	Amount of time passed (in seconds) since the first attempt
out	err	Reason for failure
returns		Valid manager (0), invalid manager (1), or failure (–1)

B.8 Garbage Collection

The gc interface is defined in <globe/inf/gc.inf>. This interface is described in Section 5.7.

interface gc		
id	”39690000efe52ff035053ba60002efe6” ”000000000000000000000000c01fe7ae”	
method	addRef	Tell the object about a new reference in the name space
in	handle	Object handle of the directory object that contains the reference
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	checkRef	A reference has been deleted from a directory
in	handle	Object handle of the directory object that contained the reference
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	reachable	Ask whether the object is reachable from the root set
in	interval	Desired consistency interval in seconds
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	getName	Return one of the object’s names in the name space
out	err	Reason for failure
returns		Pathname or NULL
method	dirEntry	Return the directory entry that refers to an object
in	handle	Object handle to look up
out	err	Reason for failure
returns		Directory entry or NULL
method	getPath	Return a path to the root or a (partial) ancestor graph
out	path	List of object identifiers on the path to a root object
out	graph	List of edges that represent parent-child relationships
out	err	Reason for failure
returns		Success (0) or failure (–1)

interface gc (continued)		
method	delRef	Delete a stored reference to a parent directory
in	handle	Object handle to delete
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	listRefs	Return a list of references to parent directories
out	err	Reason for failure
returns		List of object handles or NULL

B.9 Files, Streams and Local Persistent State

The pickle interface is defined in <p/inf/pickle.inf>. This interface is a standard interface to marshal and unmarshal the state of an object. This interface is implemented by semantics objects (see Section 5.5.3).

interface pickle		
id ”39690000f7b523b832de4b36000bf7b6” ”000000000000000000000000c01fe7ae”		
method	marshal	Write the marshaled state of the object to an output stream
in	ostream	Output stream interface
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	unmarshal	Read marshaled state from an input stream
in	istream	Input stream interface
out	err	Reason for failure
returns		Success (0) or failure (–1)

The inputStream, outputStream, and openStream interfaces are defined in the header file <p/inf/stream.inf>. Stream interfaces are described in Section 5.10.4.

interface inputStream		
id ”3969000075c5130e3291db29000c75c6” ”000000000000000000000000c01fe7ae”		
method	readbuf	Return a buffer with data read from a stream
out	size	Amount of data read
out	err	Reason for failure
returns		Pointer to allocated block of data or NULL
method	readall	Read data from a stream
in	buf	Pointer to user supplied buffer
in	size	Size of the buffer
out	err	Reason for failure
returns		Amount of data read or failure (–1)

interface outputStream		
id ”3969000009e7122c331ec5f2000f09e8” ”000000000000000000000000c01fe7ae”		
method	write	Write data to a stream
in	buf	Pointer to buffer with data
in	size	Amount of data to write
out	err	Reason for failure
returns		Success (0) or failure (–1)

interface openStream		
id ”396900002e4213103291db3800062e43” ”000000000000000000000000c01fe7ae”		
method	open	Open file or device
in	name	Name of file to open
in	mode	C stdio style mode string
out	err	Reason for failure
returns		Success (0) or failure (–1)

The state and openState interfaces are defined in <p/inf/state.inf>. The state interface is an extended version of the file interface. The mark and sync methods allow the user of this interface to control the consistency of the persistent state. A state interface provides access to a persistent “file”. The openState interface is used to attach an object that implements the state interface to one particular file (or create a new one), after that, the state interface can be used to access the state. These interfaces are described in Section 5.10.4.

interface state		
id ”39690000cec648ed3365f5130009cec7 ”000000000000000000000000c01fe7ae”		
method	write	Write data
in	offset	Offset in state
in	data	Pointer to block of data
in	size	Amount of data to write
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	read	Read data
in	offset	Offset in state
in	reqsize	Requested amount of data
out	actsize	Returned amount of data
out	err	Reason for failure
returns		Pointer to allocated buffer with data
method	truncate	Truncate a state object to a specified size
in	length	New size
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	mark	Mark the current state as consistent
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	sync	Make the current state persistent
out	err	Reason for failure
returns		Success (0) or failure (–1)

interface openState		
id ”39690000f2f04a76336604c00008f2f1”		
”000000000000000000000000c01fe7ae”		
method	create	Create new persistent state
in	name	Name of persistent state
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	open	Open existing persistent state
in	name	Name of persistent state
out	err	Reason for failure
returns		Success (0) or failure (–1)

B.10 Libraries and System Objects

The following interfaces are used internally by the Globe libraries. The libglobe interface is defined in <globe/inf/libglobe.inf>. The libprts interface is defined in the header file <p/inf/libprts.inf>. These interfaces are described in Section 5.10.

interface libglobe		
id ”396900001f5d189f35222cfb00051f5e”		
”000000000000000000000000c01fe7ae”		
method	bind_handle	Bind to an object using an object handle
in	name	Name of the object
in	handle	Object handle
in	ctx	Current context
out	err	Reason for failure
returns		Pointer to standard object interface or NULL
method	name_lookup	Look up a name in the global name space
in	name	Name to look up
out	err	Reason for failure
returns		Object handle or NULL
method	location_lookup	Look up contact addresses in the location service
in	handle	Object handle to look up
out	err	Reason for failure
returns		List of contact addresses or NULL

interface libprts		
id	”39690000b3322f0d32949ea60002b333” ”000000000000000000000000c01fe7ae”	
method	getuniq	Generate a unique identifier
out	id	Unique identifier
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	globallock_enter	Enter critical region protected by the global lock
returns		Nothing
method	globallock_leave	Leave critical region protected by the global lock
returns		Nothing
method	lock_alloc	Allocate a lock
out	lock	Lock identifier
in	label	ASCII description of lock purpose/category
returns		Success (0) or failure (–1)
method	lock_free	Deallocate lock
inout	lock	Lock identifier, set to 0 when deleted
returns		Nothing
method	lock_enter	Enter critical region protected by lock
in	lock	Lock identifier
returns		Nothing
method	lock_leave	Leave critical region protected by lock
in	lock	Lock identifier
returns		Nothing
method	lock_wait	Wait for signal
in	lock	Lock identifier
returns		Nothing
method	lock_signal	Signal a blocked thread
in	lock	Lock identifier
returns		Nothing
method	lock_broadcast	Signal all blocked threads
in	lock	Lock identifier
returns		Nothing

Six communication interfaces are defined in <p/inf/comm.inf>. The comm_p2p interface is used to establish point-to-point communication channels. The commConnCB interface is a callback interface which should be provided by the user of the comm_p2p interface to accept incoming connection requests. The commAddrs interface converts between ASCII strings and the 64-bit address identifiers which are used in other communication interfaces. The comm interface is the main interface to send data. The commRdCB is the callback interface associated with the comm interface to receive data. The last interface, commGroup, is an experimental interface for joining and leaving multicast groups. These interfaces are described in Section 5.10.5.

interface comm_p2p		
id "396900009b1d0ebc32f6080400089b1e"		
"000000000000000000000000c01fe7ae"		
method	bind	Bind to a local address
in	addr	Address
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	listen	Start accepting connections
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	connect	Connect to a remote process
in	dst	Destination address
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	setCallBack	Set the listen callback interface
in	ci	Callback interface
in	ref	Reference to be passed
in	ctx	Name-space context for new object
out	err	Reason for failure
returns		Success (0) or failure (–1)

interface commConnCB		
id	”396900003a4b11df3291d61b000f3a4c” ”000000000000000000000000c01fe7ae”	
method	newConn	A new connection has been established
in	ref	Reference (same as in setCallBack)
in	obj	Pointer to standard object interface of new communication object for the connection
in	addr	Remote address
returns		Context identifier for next communication object
method	error	Report an error
in	ref	Reference (same as in setCallBack)
out	err	Reason for failure
returns		Nothing

interface commAddrs		
id	”39690000ef0211e33291d6600003ef03” ”000000000000000000000000c01fe7ae”	
method	lookup	Look up (marshal) an address identifier
in	addr	Address to look up
out	err	Reason for failure
returns		Pointer to ASCII strings stored in an allocated buffer
method	install	Install an address into the address table
in	str	ASCII string representing address
out	err	Reason for failure
returns		Address identifier

interface comm		
id "39690000e5fa21aa32b004330006e5fb" "000000000000000000000000c01fe7ae"		
method	send	Send a message
in	dst	Destination address
in	data	Packet to be send
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	reply	Reply to a request
in	dst	Destination address
in	ref	Reference (passed in req_arrived)
in	data	Packet to be replied
returns		nothing
method	sendrec	Send a request and return the reply
in	dst	Destination address
in	data	Request packet
out	err	Reason for failure
returns		Reply packet
method	setCallBack	Set receive callback interface
in	rdi	Callback interface
in	ref	Reference to be passed
out	err	Reason for failure
returns		Success (0) or failure (–1)

interface commRdCB		
id ”396900005df211e13291d65400035df3” ”000000000000000000000000c01fe7ae”		
method	msg_arrived	A message has arrived
in	ref	Reference (same as in setCallBack)
in	src	Source address
in	data	Packet
returns		nothing
method	req_arrived	A request message has arrived
in	ref	Reference (same as in setCallBack)
in	src	Source address
in	data	Packet
in	reqref	Request reference
returns		Reply packet or nil to indicate that reply will be called later
method	error	Receive error
in	ref	Reference (same as in setCallBack)
out	err	Reason for failure
returns		Nothing

interface commGroup		
id ”396900009f9111dd3291d60e00049f92” ”000000000000000000000000c01fe7ae”		
method	create	Create a multicast communication group
out	err	Reason for failure
returns		Address of the group
method	join	Join a multicast group
in	dst	Address of the group
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	leave	Leave a group
out	err	Reason for failure
returns		Success (0) or failure (–1)

The memory interface is defined in <p/inf/memory.inf>. This interface is implemented by the default memory allocator object /builtin/memory. Memory allocation is described in Section 5.10.2.

interface memory		
id ”396900002f8712ef3291dac700072f88” ”000000000000000000000000c01fe7ae”		
method	alloc	Allocate a segment of memory
in	size	Amount of memory to allocate
out	err	Reason for failure
returns		Pointer to the allocated block of memory

The thread and threadCB interfaces are defined in <p/inf/thread.inf>. These interfaces are described in Section 5.10.3.

interface thread		
id ”396900000e0e13233291db54000d0e0f” ”000000000000000000000000c01fe7ae”		
method	start	Start execution
in	cbi	Pointer to threadCB interface
in	ref	Callback reference
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	join	Wait for a thread to finish
out	err	Reason for failure
returns		Success (0) or failure (–1)
method	detach	Detach the thread from the thread object
returns		nothing
method	cancel	Cancel the thread
returns		nothing

interface threadCB		
id ”39690000b34619a63308a22b000cb347” ”000000000000000000000000c01fe7ae”		
method	start	Procedure to execute in a thread
in	ref	Callback reference
returns		nothing

The debug interface is defined in <p/inf/debug.inf>. This interface allows objects to bind to the debug object (/builtin/debug) to generate debug output.

interface debug		
id	”39690000a95e12e73291da8b0000a95f”	
	”00000000000000000000000c01fe7ae”	
method	dvprint	Print a debug message
out	err	Reason for failure
in	level	Debug level of message
in	fmt	Printf style format string
in	va	Argument list
returns		Success (0) or failure (–1)

Samenvatting

De architectuur van een wereldwijd gedistribueerd systeem

Introductie Dit proefschrift beschrijft Globe, een nieuwe software-architectuur voor het uitwisselen van informatie tussen processen die draaien op door een wereldwijd netwerk verbonden computers. Globe is afgeleid van de term "**GL**obal **O**bject **B**ased **E**nvironment", een alomtegenwoordige, object gebaseerde omgeving. Het doel van het onderzoek beschreven in dit proefschrift is de ontwikkeling van een architectuur die flexibiliteit koppelt aan een uniform programmeermodel met een hoog niveau van abstractie. Bijkomende eisen zijn schaalbaarheid, veiligheid, lokatie- en replicatietransparantie, autonoom systeembeheer en uitwisselbaarheid.

Het Internet, op dit moment het grootste wereldwijde netwerk, is gebaseerd op het versturen van berichten tussen computers. De meeste communicatieprotocollen op het Internet gebruiken òf UDP (een onbetrouwbaar, niet-verbindingsgeörienteerd protocol) òf ze gebruiken TCP, dat zowel betrouwbaar is als verbindingsgeörienteerd. Deze aanpak is min of meer uniform, niet flexibel, en biedt een betrekkelijk laag niveau van abstractie.

Onze architectuur is, net als vele andere, gebaseerd op het gebruik van objecten. Het aantrekkelijke van objecten is dat zij hun implementatie geheel omvatten, en daarmee hun gebruikers toegang bieden op een hoog niveau van abstractie. Een object bevat een hoeveelheid gegevens, en biedt aan zijn gebruikers de mogelijkheid opdrachten te geven (ook wel het aanroepen van methoden genoemd) die de gegevens in het object veranderen of er over rapporteren. Twee processen kunnen communiceren door een object gezamenlijk te gebruiken. Nadat het ene proces, door de aanroep van een methode, de inhoud van het object veranderd heeft, wordt deze verandering zichtbaar voor het andere proces. Processen zien niets anders dan objecten en aanroepen van methoden; de details van het netwerk dat de computers verbindt is verborgen in de implementatie van de objecten.

De meeste object gebaseerde systemen, inclusief CORBA, bieden een uniform, hoog-niveau programmeermodel. De applicatieprogrammeur heeft echter slechts een beperkte invloed op de implementatie van de gebruikte objecten. De programmeur is verantwoordelijk voor de implementatie van de functionele aspecten van een object (wat het object doet, het verschil tussen een e-mail object en een documentobject) maar heeft geen, of slechts beperkte invloed op de niet-functionele aspecten, zoals replicatie en beveiliging. Met andere woorden, de meeste systemen zijn niet erg flexibel met betrekking tot implementaties van objecten.

Distributed Shared Objects De Globe architectuur gebruikt de term *distributed shared objects* (gezamenlijk gebruikte, gedistribueerde objecten) voor objecten die gebruikt worden om informatie uit te wisselen tussen processen die draaien op verschillende computers. Een ander soort objecten dat alleen binnen een proces bestaat, wordt lokaal object genoemd. Een distributed shared object wordt zo genoemd omdat het gezamenlijk gebruikt wordt door meerdere processen, en de implementatie fysiek gespreid kan worden over alle deelnemende processen.

Een techniek genaamd *dynamic loading* (dynamisch inladen) wordt gebruikt om de Globe architectuur flexibeler te maken dan de meeste andere architecturen. Wanneer een proces een gedistribueerd object probeert te gebruiken, wordt een deel van de implementatie van het object in het proces geladen. Dit betekent dat een programma, nadat het gecompileerd is, geen implementaties bevat van de gedistribueerde objecten die gebruikt gaan worden. Verschillende gedistribueerde objecten kunnen verschillende implementaties gebruiken, en de benodigde implementaties worden tijdens de uitvoering van het programma (*run-time*) geladen.

Onderdeel van de Globe architectuur is een architectuur voor lokale objecten. Het gedeelte van een gedistribueerd object dat in een proces geladen wordt, is de implementatie van een lokaal object. Feitelijk is de complete implementatie van een gedistribueerd object niets anders dan een verzameling lokale objecten die in verschillende processen (eigenlijk *address spaces*) geladen zijn. Vanuit het oogpunt van een programmeur wordt een gedistribueerd object altijd vertegenwoordigd door een lokaal object. Een lokaal object dat onderdeel is van een gedistribueerd object wordt *representative* (vertegenwoordiger) genoemd. Deze representatives communiceren en werken samen om de methoden van het gedistribueerde object te implementeren en de inhoud te distribueren. De implementatie van een lokaal object ligt opgeslagen in een speciaal object, een *class object* (klasse-object).

Lokale objecten Een lokaal object biedt één of meer *binary interfaces*. Een *interface* is een verzameling methoden die de door een object geboden functionaliteit beschrijft. Het gebruik van binary interfaces maakt het mogelijk om applicaties onafhankelijk van objectimplementaties te compileren.

Objecten bieden, om meerdere redenen, meer dan één interface aan. Eén reden is dat object-specifieke methoden gescheiden kunnen worden van methoden die gebruikt worden voor het beheer van objecten. De methoden die de replicatie van de inhoud van een object regelen kunnen bijvoorbeeld gescheiden worden van de methoden die de inhoud van het object lezen en schrijven. Een tweede reden is dat het gebruik van meer dan één interface de mogelijkheid biedt om verschillende functies te combineren in één object. Een enkel object kan bijvoorbeeld zowel de *file interface* als de *directory interface* implementeren. Een derde reden is compatibiliteit met eerdere versies; een object kan een nieuwe versie van een interface implementeren en de oude versie behouden voor compatibiliteit met oudere applicaties.

Het lokale objectmodel bevat ook *composite objects* (samengestelde objecten) en een lokale *name space*. Een samengesteld object is een lokaal object dat opgebouwd is uit verscheidene andere lokale objecten. Compositie wordt gebruikt om complexe lokale objecten te creëren uit primitievere objecten. Een representative is normaal gesproken een samengesteld object.

De lokale name space is een extra techniek die gebruikt wordt om meer flexibiliteit te creëren. In plaats van het doorgeven van *pointers* naar objecten, is het mogelijk om door middel van een naam aan een object te refereren. Deze flexibiliteit wordt bijvoorbeeld gebruikt om te verbergen dat, hoewel de meeste klasse-objecten dynamisch geladen worden, sommige klasse-objecten mee gecompileerd zijn met de applicatie. Voor klasse-objecten die al aanwezig zijn, levert de name space opzoekfunctie direct een pointer naar het object; in de andere gevallen wordt het object eerst ingeladen.

Tenslotte bevat het lokale objectmodel systeemobjecten. Deze objecten bieden een abstractie van de functionaliteit die geboden wordt door het onderliggende besturingssysteem. Voorbeelden van systeemobjecten zijn communicatie-objecten, *thread-objecten*, geheugenbeheerobjecten en *file-objecten*. Samen bieden deze objecten een overdraagbare interface naar het besturingssysteem.

Replicatie De implementatie van een gedistribueerd object als een verzameling lokale objecten die communiceren is prettig voor de gebruiker van het gedistribueerde object, maar vergemakkelijkt niet de taak van de programmeur die het object moet implementeren. De implementatie van een gedistribueerd object kan vereenvoudigd worden door gebruik te maken van een samengesteld object dat bestaat uit vier subobjecten. Eén subobject, *semantics object* (semantiekobject) genaamd, bevat de objectspecifieke functionaliteit. Het semantiekobject bepaalt bijvoorbeeld of het gedistribueerde object een file-object is, of een *mailboxobject*, of een WWW-pagina.

Een tweede subobject, het replicatie-object, verzorgt de replicatie van de inhoud van het gedistribueerde object, verstuurt aanroepen van methoden naar andere processen, enz. Dit object bevat het meest complexe gedeelte van de implementatie van een gedistribueerd object. Gelukkig kan een verzameling van replicatie-objecten die verschillende replicatiestrategieën implementeren, opgeslagen worden in een *class repository*. De programmeur van een gedistribueerd object hoeft slechts een geschikt replicatie-object uit de class repository te kiezen.

Replicatie-objecten gebruiken communicatie-objecten om te communiceren met replicatie-objecten in andere processen. Verschillende communicatie-objecten voor betrouwbare en onbetrouwbare *message passing*, voor groepcommunicatie, enz., zijn ook aanwezig in de class repository.

Het vierde subobject, *control object* (besturingsobject) genaamd, koppelt het semantiekobject aan het replicatie-object. De uitvoering van een methode van een gedistribueerd object begint in het besturingsobject. Op verzoek van het replicatie-object voorziet het besturingsobject het replicatie-object van de ingepakte (*marshaled*) parameters om doorgestuurd te worden naar andere processen. Het replicatie-object kan ook bepalen dat het besturingsobject de methode van het semantiekobject moet aanroepen. Het is bedoeling dat het besturingsobject door een compiler wordt gegenereerd.

Persistentie en beveiliging Het replicatie-object biedt ook persistentie. De aanpak die in Globe gebruikt wordt, is dat individuele replica's van de inhoud van het gedistribueerde object persistent gemaakt worden. Het replicatie-object bewaart en herlaadt de inhoud van het semantiekobject. Persistente replica's worden beheerd door een *object repository*. De object repository deactiveert persistente replica's die enige tijd ongebruikt zijn. Wanneer de replica op een later moment weer nodig is, wordt het weer geactiveerd door de object repository. Object repositories bieden ook ondersteuning voor de creatie van nieuwe gedistribueerde objecten.

Voor beveiliging wordt een extra beveiligingssubobject toegevoegd aan de compositie die een gedistribueerd object implementeert. Het beveiligingssubobject verzorgt versleuteling, identificatie en toegangscontrole. Speciale gedistribueerde objecten, die *principal objects* genoemd worden, vertegenwoordigen gebruikers tijdens identificatie procedures.

Naming en binding Het creëren van een nieuw lokaal object en het verbinden van dat object met een gedistribueerd object heet *binding*. Het binding proces begint normaal gesproken met de naam van een gedistribueerd object in een wereldwijde name space. De *naming service* wordt gebruikt om te bepalen waar het gedistribueerd object zich bevindt, en wat voor implementatie nodig is om het object te benaderen.

Om binding te ondersteunen biedt ieder gedistribueerd object één of meer *contact points* (contactpunten), welke gerepresenteerd worden door *contact addresses* (contactadressen). Een contactadres bestaat uit twee delen. Het eerste deel is een *protocol identifier* die specificeert welk protocol (inclusief, in termen van het OSI model, de presentatie-, sessie-, en applicatielaag) gebruikt moet worden om het gedistribueerde object te benaderen. De protocol identifier wordt gebruikt om een geschikt klasse-object te vinden dat het protocol implementeert. Het tweede deel van het contactadres bevat de eigenlijke netwerkadressen en andere informatie die nodig is om een verbinding met het gedistribueerde object op te zetten.

Een gedistribueerd object kan om twee redenen meer dan één contactadres aanbieden. De eerste reden is dat het meerdere protocollen kan gebruiken; bijvoorbeeld voor compatibiliteit met eerdere versies, of omdat het meest efficiënte protocol afhangt van de situatie. Meerdere contactadressen die hetzelfde protocol gebruiken verwijzen naar contactpunten op verschillende computers. Het gebruik van contactpunten op verschillende computers is nodig om gebruik te maken van replicatie, bijvoorbeeld om de fouttolerantie te verhogen of om de lokaliteit te verbeteren.

Het binding proces bestaat uit vier stappen: een *name lookup*, een *location lookup*, *destination selection* (bestemmingsselectie) en implementatieselectie. De eerste stap zoekt een naam op in de wereldwijde name service. Deze step levert een *object handle* op. De object handle wordt gebruikt om contactadressen op te zoeken in de *location service*. De contactadressen worden in de derde step geordend naar de verwachte bruikbaarheid. Tenslotte zoekt de laatste stap een geschikt klasse-object bij de protocol identifier in het beste contactadres.

De name service is gescheiden van de location service om het beheer van contactadressen te vereenvoudigen. In het algemeen wordt de beslissing om een contactadres toe te voegen of te verwijderen door het gedistribueerde object zelf genomen. Daarentegen zijn de gebruikers verantwoordelijk voor de naam van het object in de name service. Het gebruik van twee diensten, op basis van één object handle, maakt het mogelijk dat de gebruiker en het object onafhankelijk van elkaar opereren.

De name service biedt een gedistribueerde, hiërarchische name space vergelijkbaar met DNS. Deze name space wordt gecreëerd met behulp van *directory-objecten*. Directory-objecten zijn gedistribueerde objecten die object handles van andere objecten (inclusief andere directory-objecten) opslaan.

Location service De location service gebruikt een object handle om de contactadressen van een gedistribueerd object op te zoeken. Een object handle bestaat uit twee delen. Het eerste deel is een 256-bits, wereldwijd unieke *object identifier*. Het tweede deel wordt door de location service gecreëerd, wanneer het object voor het eerst geregistreerd wordt. Dit tweede deel hangt af van de structuur van de location service. Sommige implementaties van de location service gebruiken alleen de object identifier, en hebben geen behoefte aan aanvullende informatie. Andere implementaties negeren de object identifier en gebruiken alleen hun eigen informatie.

De location service kan op eenvoudige wijze geconstrueerd worden op basis van een gedistribueerde, hiërarchische name space (vergelijkbaar met de name service), waarbij contactadressen opgeslagen worden in plaats van object handles. In deze aanpak worden alle contactadressen van één object op één vaste plaats opgeslagen worden. Er zijn twee belangrijke problemen. Het eerste probleem is dat het op één plek opslaan van alle contactadressen van een object ongunstig is voor objecten met meerdere replica's. Het tweede probleem is dat de contactadressen achterblijven op de originele locatie wanneer het object verplaatst wordt. Om deze reden noemen we deze aanpak de *home-based approach*.

Een andere aanpak, die *distributed search tree* (gedistribueerde zoekboom) genoemd wordt, deelt de location service op in stukken (partities), en slaat een object handle met de bijbehorende contactadressen op een partitie die dicht bij het corresponderende contactpunt ligt. Voor objecten met veel replica's wordt informatie over het object opgeslagen in vele partities verspreid over de location service. Wanneer een object zich verplaatst, wordt de informatie die opgeslagen was in de buurt van het oude contactpunt verwijderd, en wordt informatie op de nieuwe locatie gecreëerd.

Om informatie over een object te kunnen vinden worden *forwarding pointers* gecreëerd die in de juiste richting wijzen. Voor ieder object dat contactpunten heeft in een bepaalde regio bevat een regionale knoop forwarding points naar de partities die de contactadressen van het object bevatten. Super-regio's bevatten forwarding pointers naar regionale knopen, enzovoorts. Een speciale *rootknoop* bevat forwarding pointers voor ieder gedistribueerd object in het systeem.

Om een contactadres voor een gedistribueerd object op te zoeken, stuurt een proces eerst een verzoek naar een nabij gelegen partitie. Als die partitie geen gegevens heeft over het object wordt een regionale knoop geprobeerd. Dit proces herhaalt zich totdat de rootknoop bereikt is. De rootknoop heeft per definitie informatie over het object (tenzij het object verwijderd is). Dit mechanisme zoekt over een steeds uitbreidend gebied. Het voordeel van deze manier van zoeken is dat contactadressen van nabij gelegen contactpunten het eerst gevonden worden. In het algemeen is dit is een gewenste vorm van lokaliteit.

Het grootste probleem van deze aanpak is de implementatie van de rootknoop. Het op één plaats opslaan van informatie over ieder gedistribueerd object in het systeem is niet mogelijk. Gelukkig is het partitioneren van de rootknoop genoeg voor opzoekoperaties. Vanwege de lokaliteit van het opzoekalgoritme bereiken verzoeken de rootknoop alleen indien het gezochte object geen contactpunten heeft in de buurt van het proces dat met binding bezig is. Dit betekent dat deze implementatie van de location service wat betreft communicatie, geen negatieve invloed op de schaalbaarheid heeft.

Garbage collection Een groot systeem dat dynamische toewijzing van *resources* toestaat, hoort resources waaraan niet gerefereerd wordt automatisch vrij te geven. In kleine, niet-persistente systemen is het mogelijk het vrijgeven van ongebruikte resources aan de programmeur toe te vertrouwen. In grote, persistente systemen, hopen kleine fouten zich op en zijn ongebruikte resources moeilijk met de hand te vinden. Het automatisch vrijgeven van resources wordt *garbage collection* genoemd.

Een probleem met garbage collection is dat algoritmen die goed werken binnen één proces, *tracing garbage collectors*, niet goed werken in grote gedistribueerde systemen. Tracing garbage collectors identificeren eerst alle objecten die bereikbaar zijn. Daarna worden alle onbereikbare objecten verwijderd. Het probleem is dat het gebruik van deze aanpak in een gedistribueerd systeem, vereist dat alle processen samenwerken en synchroniseren. Het vereiste dat alle processen samenwerken is ongewenst om vele redenen, inclusief het ontstaan van problemen met de beveiliging.

Het garbage-collectionalgoritme in Globe gebruikt het feit dat gedistribueerde objecten relatief groot (*heavyweight*) zijn. Dit maakt het mogelijk om van ieder gedistribueerd object te verwachten dat het de object handles opslaat van (directory-) objecten die naar het object terugverwijzen. Object handles die opgeslagen worden voor garbage collection worden *reverse references* (terugverwijzigingen) genoemd.

Het garbage collection algoritme van Globe gaat er vanuit dat er een collectie objecten is, die niet door de *garbage collector* verwijderd wordt. Deze collectie van objecten wordt de *root set* genoemd. Alle objecten die (indirect) bereikbaar zijn vanuit de root set worden niet verwijderd. Alle overige objecten (die dus onbereikbaar zijn) moeten verwijderd worden.

In ieder gedistribueerd object voert het garbage-collectionalgoritme twee subalgoritmen parallel uit. Het eerste subalgoritme gebruikt de reverse references om een pad te vinden naar een object in de root set. Een object mag blijven bestaan als een pad naar de root gevonden wordt. Helaas is niet zo dat het object verwijderd kan worden als het pad naar de root niet gevonden wordt.

Het tweede subalgoritme bepaald of een object onbereikbaar is of niet. Dit algoritme is een gedistribueerd graafalgoritme dat termineert als er geen pad naar de root set bestaat. Door de twee subalgoritmen parallel uit te voeren, termineert het gecombineerde algoritme wanneer òf een pad naar de root gevonden is òf het graafalgoritme bepaald heeft dat het object onbereikbaar is.

Prototype Een prototype van de Globe architectuur is geïmplementeerd om te controleren of all componenten op elkaar passen en of het systeem de gewenste flexibiliteit heeft. Schaalbaarheid was geen doel van het prototype. Het is zelfs zo dat de location service geïmplementeerd is als een gecentraliseerde service. Het prototype biedt een object gebaseerd hypertekstsysteem, dat benaderd kan worden met een standaard *WWW browser*.

Conclusies Dit proefschrift beschrijft een nieuwe architectuur voor wereldwijd gedistribueerde systemen, die gebaseerd is op gedistribueerde objecten. De architectuur bevat een location service die locatie- en replicatietransparantie ondersteund. Een nieuw garbage-collectionalgoritme biedt gedistribueerde detectie van onbereikbare objecten. De constructie van een gedistribueerd object wordt vereenvoudigd door gebruik te maken van communicatie-, replicatie-, besturings-, en semantiekobjecten die de implementatie van replicatieprotocollen afsplitsen van de implementatie van object-specifieke semantiek. De interface tussen het besturings- en het replicatieobject biedt ruimte voor meerdere replicatie strategieën.

Summary

Introduction This thesis describes a new software architecture, called Globe, for information exchange between processes that are running on computers connected by a wide-area network. Globe stands for **GL**obal **O**bject **B**ased **E**nvironment. The goal is to create an architecture that combines flexibility with a uniform, high-level programming model. Additional requirements are scalability, security, location and replication transparency, autonomous administration, and interoperability.

The Internet, at this moment the largest wide-area network, is based on a message-passing paradigm. Most protocols that are used on the Internet, are based either on an unreliable connectionless protocol (UDP) or on a reliable connection-oriented protocol (TCP). This approach is somewhat uniform, but is neither flexible nor high level.

Like many other systems, our architecture is based on objects. The attraction of objects is that they completely encapsulate their implementations, thus providing a high-level interface to their users. An object encapsulates some amount of state (information) and provides the users of the object with operations (also called methods) that change the state of the object or that report about the object's state. To communicate, two processes agree to share an object. One process invokes an operation on the object that changes the state of the object. This new state is observed by the other process. Processes care only about their objects and the methods they invoke on those objects. The details of the network that connects the computers are hidden by the objects' implementations.

Most object-based approaches (including CORBA) provide a uniform, high-level programming model. However, the application programmer has only limited control over the implementation of those objects. The programmer controls the implementation of the functional aspects of an object (i.e, what the object does, whether it is a mail object or a document object), but has no or only limited control over nonfunctional aspects, such as replication and security. In other words, those systems provide limited flexibility with respect to object implementations.

Distributed Shared Objects In the Globe architecture, objects that are used to exchange information between processes running on different computers, are called distributed shared objects. Another kind of object is called a local object. Local objects exist only within a single process. Distributed shared objects get their name from the fact that they are shared between different processes and the fact that their implementations can be physically distributed over all participating processes.

A technique called dynamic loading is used to make the Globe architecture more flexible than most other architectures. When a process tries to use a distributed shared object, part of the implementation of the distributed shared object is loaded into the process' address space. This means that after an application has been compiled, it does not contain implementations for any distributed shared objects it might use. Different distributed shared objects may use different implementations and the required implementation is loaded at run-time.

Part of the Globe architecture is an architecture for local objects. The part of a distributed shared object that is loaded into a process is the implementation of a local object. In fact, the complete implementation of a distributed shared object is nothing more than the collection of local objects that are loaded in different address spaces. From the programmer's point of view, a distributed shared object is always represented by a local object. The local objects that are part of a distributed shared object are called representatives. These representatives cooperate and communicate to implement methods that can be invoked on the distributed shared object and to distribute the state of the object. Implementations of local objects are stored in special objects, called class objects.

Local Objects Local objects provide one or more binary interfaces. An interface is a collection of methods. The use of binary interfaces allows an application to be compiled separately from object implementations. An interface describes the functionality offered by the object.

Objects provide more than one interface for multiple reasons. One reason is that object-specific operations can be separated from operations related to object management. For example, operations that control the replication of the state of the object can be separated from methods that read and write the object's state. A second reason is that the use of multiple interface allows a single object to combine different functions. For example, a single object can implement both a file and a directory interface at the same time. A third reason for providing multiple interface is backward compatibility. An object can implement a new version of an interface and retain the old version for compatibility with older applications.

The local object model also contains composite objects and a local name space. A composite object is a local object that consists of several other local objects. Composition is used to create complex local objects out of more primitive objects. A representative of a distributed shared object is usually a composite object.

The local name space is another technique to introduce more flexibility. Instead of passing pointers to objects, it is possible to refer to an object by name. For example, this flexibility is used to hide the fact that, although most class objects are loaded dynamically, some class objects are linked with the application. For class objects that are already present, the name-space lookup function simply returns a pointer to the object. For other class objects, the object is loaded first.

Finally, the local object model includes the concept of system objects. System objects abstract the functionality of the underlying operating system. Examples of system objects are communication objects, thread objects, memory allocators, and file objects. Together these system objects provide a portable operating system interface.

Replication The implementation of a distributed shared object as a collection of local objects that communicate is convenient for the user of a distributed shared object, but does not help the programmer who has to implement a distributed shared object. The implementation of a distributed shared object can be simplified by using a composite object that consists of four subobjects. One subobject, called the semantics object, contains the object-specific functionality. For example, the semantics object determines whether a distributed shared object is a file object, a mailbox, or a WWW page.

A second object, the replication object, handles the replication of the state of the distributed shared object, forwards method invocations to other processes, etc. This object is the most complex part of the implementation of a distributed shared object. Fortunately, a collection of replication objects that implement different replication strategies can be put into a class repository. The programmer of a distributed shared object selects only an appropriate replication object from the class repository.

Replication objects use communication objects to communicate with replication objects in other address spaces. Different communication objects for reliable and unreliable message passing, for group communication, etc. are also put into the class repository.

The fourth subobject, called the control object, provides the “glue” between the semantics object and the replication object. A method invocation on a distributed shared object always starts in the control object. Upon request (by the replication object), the control either provides the replication object with the marshaled arguments to forward the method invocation to a different process or the control object invokes the method on the semantics object. The control object is expected to be generated by a special compiler.

Persistence and Security The replication object also provides persistence. The approach taken in Globe is that individual replicas of the state of a distributed shared object are made persistent. The replication object saves and reloads the state of the semantics object. Persistent replicas are managed by an object repository. The object repository deactivates (passivates) persistent replicas that have been idle for some time. When a persistent replica is needed at a later time, it is activated again by the object repository. Object repositories also provide support for creating new distributed shared objects.

To support security, an extra security subobject is added to the composition that implements a distributed shared object. The security object handles security aspects such as encryption, authentication, and access control. Special distributed shared objects, called principal objects, represent users during authentication.

Naming and Binding The process of creating a new local object and connecting that object to the distributed shared object, is called binding. The binding process typically starts with a name of a distributed shared object in a worldwide name space. The name service is used to determine where the distributed shared object is located and what kind of implementation is needed to connect to the object.

To support binding, each distributed shared object provides one or more contact points, which are represented by contact addresses. A contact address consists of two parts. The first part is a protocol identifier. The protocol identifier specifies the complete protocol (including, in terms of the OSI reference model, the presentation, session, and application layers) needed to connect to the distributed shared object. The protocol identifier is used to locate a suitable loadable class object that implements the protocol. The second part of the contact address contains the actual network addresses and other demultiplexing information needed to setup a connection to the distributed shared object.

A distributed shared object may provide more than one contact address for two reasons. The first reason is that it may support more than one protocol, for example, for backward compatibility or because different protocols are more efficient in different situations. Multiple contact addresses that use the same protocol refer to different contact points on different computers. The use of contact points on different computers is needed to benefit from replication, for example, to increase fault tolerance or to improve locality.

The binding process consist of four steps: a name lookup, a location lookup, destination selection, and implementation selection. The first step looks up a name in a worldwide name service. This step returns an object handle. This object handle is used to look up a set of contact addresses in the location service. The third step orders the addresses in the set of contact addresses based on expected performance. The last step looks up a suitable class object based on the protocol identifier in the best contact address.

The reason for a separate name and location service is to simplify the management of contact addresses. In general, the decision to create or delete contact addresses is taken by the distributed shared object itself. On the other hand, users are mainly responsible for the name of the object in the name service. The use of two services that share a single object handle for each distributed shared object, allows the user and the distributed shared object to operate independently.

The name service provides a distributed, hierarchical name space, similar to DNS. This name space is created using directory objects. Directory objects are distributed shared objects that store object handles of other distributed shared objects, including other directory objects.

Location Service The location service uses an object handle to look up a set of contact addresses for a distributed shared object. An object handle consists of two parts. The first part is a 256-bit, worldwide unique object identifier. The second part is created by the location service when the object is registered for the first time. This second part depends on the structure of the location service. Some location service implementations only use the object identifier and do not need any additional information. Other implementations ignore the object identifier and rely only on their own information.

An easy way to implement a location service is to start with a distributed, hierarchical name space (similar to the name service) and insert sets of contact addresses instead of object handles. This approach is easy to implement and has nice scaling properties. Unfortunately, it does not work well as a location service. One problem is that all contact address are stored in one place. This creates a bottleneck which eliminates the increased fault tolerance and improved locality due to replication. Another problem is that when the object moves, its contact addresses continue to be stored in the same location. For this last reason, this approach is also called the home-based approach.

Another approach, called the distributed search tree approach, partitions the location service and stores the object handle that is associated with a contact address in a partition of the location service that is close to the corresponding contact point. For highly replicated objects, information about the object will be stored in many parts of the location service. When an object moves, the information that was stored near the old contact point is deleted, and information at the new location is created.

To find information about an object, records with forwarding pointers are created that point in the right direction. For each distributed shared object that has contact points in a certain region, a regional node stores forwarding pointers to the parts of the location service that store the contact addresses. Higher-level regions store forwarding pointers to regional nodes, etc. A special root node stores forwarding pointers for all distributed shared objects in the entire system.

To look up contact addresses for a distributed shared object, a process first queries a nearby partition. If that partition does not know about the object, a regional node is tried. This process repeats until the root node is reached. The root node has, by definition, a record for the object (unless the object has been deleted). This mechanism effectively implements a search with an expanding scope. The advantage of this search is that contact addresses for nearby contact points are found first. This kind of locality is often desirable.

The main problem with this approach is the implementation of the root node. Storing information about every distributed shared object in the system in one place is not possible. Fortunately, simply partitioning the root node is sufficient for lookup operations. Due to the locality of the lookup algorithm, the only queries that reach a partition of the root node are for objects that have no contact point close to the binding process. This means that the location service does not reduce the communication scalability of the system.

For the registration of objects, the root node does have a negative performance impact. Some tricks, such as using geographical information, can be used to reduce this problem. Another way to try to solve this problem is to combine the home-based approach with the distributed search tree approach. This combination provides a trade-off between higher registration costs and better support for replication and mobility, or lower registration costs for more static objects.

Garbage Collection A large system that allows dynamic allocation of resources should automatically deallocate unreferenced resources. In small, nonpersistent systems, it is possible to rely on the programmer to release resources when they are no longer needed. In large, persistent systems, small mistakes accumulate and unreferenced resources are very hard to find manually. The automatic deallocation of unreferenced resources is called garbage collection.

A problem with garbage collection is that algorithms that work well in a single process, so called tracing garbage collectors, do not work well in large distributed systems. Tracing garbage collectors first identify objects that are still reachable (called live). A second phase deletes all unreachable objects. The problem with this approach in a distributed system is that all processes have to synchronize and cooperate. Requiring global cooperation between all processes is undesirable for many reasons, including reasons related to security.

The garbage collection algorithm for Globe uses the fact that distributed shared objects are relatively heavyweight. For this reason we can require each distributed shared object to store a set of object handles that refer to (directory) objects that store references to the object. The object handles that are stored for garbage collection, are called reverse references.

The Globe garbage collection algorithm assumes the presence of a collection of objects, called the root set, that are not deleted by the garbage collector. All objects that are (indirectly) reachable from these objects are called live. All objects that are unreachable should be deleted.

For each distributed shared object, the garbage collection algorithm runs two subalgorithms in parallel. The first subalgorithm uses the reverse references to find a path to an object in the root set. The distributed shared object is live if a path to the root can be found. Unfortunately, the fact that a path cannot be found does not mean that the object is unreachable from the root set.

The second subalgorithm determines whether an object is unreachable or not. This algorithm is a distributed graph algorithm that terminates when no path to the root set exists. By running both subalgorithms in parallel, the combined algorithm can terminate when either a path to the root has been found, or when the graph algorithm has determined that the object is unreachable.

Prototype A prototype of the Globe architecture has been implemented to verify that all components fit together and that the system exhibits the desired flexibility. Scalability was not a design goal of the prototype. In fact, the location service is implemented as a single, centralized service. The prototype provides an object based hypertext system, which can be accessed using a standard WWW browser.

Conclusions This thesis describes a novel architecture for wide-area distributed systems that is based on distributed shared objects. The architecture contains a separate location service to support location and replication transparency. A new garbage collection algorithm provides distributed detection of unreachable objects. The construction of a distributed shared object is simplified by separating the implementation of the replication protocols from the implementation of the object-specific semantics using communication, replication, control and semantics objects. The interface between the control and the replication object allows different replication strategies.

Bibliografie

S. E. Abdullahi and G. A. Ringwood. Garbage Collecting the Internet: A Survey of Distributed Garbage Collection. *ACM Comput. Surv.*, 30(3):330–373, 1998.

M. D. Abrams, S. Jajodia, and H. J. Podell (eds.). *Information Security – An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

M. Accetta, R. Baron, D. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. Summer USENIX Technical Conf.*, pp. 93–112, Atlanta, GA. 1986.

A. Adya, M. Castro, B. Liskov, U. Maheshwari, and L. Shrira. Fragment Reconstruction: Providing Global Cache Coherence in a Transactional Storage System. In *Proc. 17th Int'l Conf. on Distributed Computing Systems*, Baltimore, MD. IEEE, 1997.

M. Ahamad and R. Kordale. Scalable Consistency Protocols for Distributed Services. *IEEE Trans. Par. Distr. Syst.*, 10(9):888–903, 1999.

R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, IETF, 1995.

B. Awerbuch and D. Peleg. Concurrent Online Tracking of Mobile Users. In *Proc. SIGCOMM Conf.*, pp. 221–233, Zürich, Switserland. ACM, 1991.

Ö. Babaoğlu and K. Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S. J. Mullender (ed.), *Distributed Systems*, chap. 4, pp. 55–96. ACM Press, New York, NY, 1993.

A. Bakker, M. van Steen, and A. S. Tanenbaum. From Remote Objects to Physically Distributed Objects. In *Proc. Seventh IEEE Workshop on Future Trends of Distributed Computing Systems*, pp. 47–52, Cape Town, South Africa. 1999.

H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Trans. Softw. Eng.*, 18(3):190–205, 1992a.

H. E. Bal, M. F. Kaashoek, A. S. Tanenbaum, and J. Jansen. Replication Techniques for Speeding up Parallel Applications on Distributed Systems. *Concurrency: Practice and Experience*, 4(5):337–355, 1992b.

G. Ballintijn, M. van Steen, and A. S. Tanenbaum. Exploiting Location Awareness for Scalable Location-Independent Object IDs. In *Proc. Fifth ASCI Annual Conf.*, pp. 321–328, Heijen, The Netherlands. ASCI, 1999.

E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS Wide Area Security Architecture. In *Proc. Seventh USENIX Security Symp.*, pp. 15–30, San Antonio, TX. USENIX, 1998.

D. Bell and L. LaPadula. Secure Computer Systems: A Mathematical Model. Technical Report MTR-2547, Vol 2, MITRE Corporation, Bedford, MA, 1973.

S. M. Bellovin and M. Merritt. Limitations of the Kerberos Authentication System. In *Proc. Winter USENIX Technical Conf.*, pp. 253–267, Berkeley, CA. USENIX, 1991.

T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol. RFC 1945, IETF, 1996.

T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738, IETF, 1994.

P. A. Bernstein. Middleware: A Model for Distributed System Services. *Commun. ACM*, 39(2):86–98, 1996.

B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. 38th IEEE Int'l Computer Conf.*, pp. 528–537, San Francisco, CA. IEEE, 1993.

K. P. Birman. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. *ACM Oper. Syst. Rev.*, 28(1):11–21, 1994.

K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, 1991.

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

A. D. Birrell, D. Evers, G. Nelson, S. Owicki, and E. Wobber. Distributed Garbage Collection for Network Objects. Technical Report 116, Digital Systems Research Center, Palo Alto, CA, 1993a.

A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comp. Syst.*, 2(1):39–59, 1984.

A. D. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *Proc. 14th ACM Symp. on Operating System Principles*, pp. 217–230, Asheville, NC. 1993b.

A. P. Black, N. C. Hutchinson, E. Jul, H. M. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Trans. Softw. Eng.*, 13(1):65–76, 1987.

N. Brown. Distributed Component Object Model Protocol — DCOM/1.0. Internet draft, Microsoft, Redmond, WA, 1998.

G. Brun-Cottan and M. Makpangou. Adaptable Replicated Objects in Distributed Environments. Technical Report 2593, Institut National de la Recherche en Informatique et Automatique, Rocquencourt, France, 1995.

N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The Primary–Backup Approach. In S. J. Mullender (ed.), *Distributed Systems*, chap. 8, pp. 199–216. ACM Press, New York, NY, 1993.

R. H. Campbell, N. Islam, R. Johnson, P. Kougiouris, and P. W. Madany. Choices, Frameworks, and Refinement. In *Proc. Second Int'l Workshop on Object Orientation in Operating Systems*, pp. 9–15, Palo Alto, CA. IEEE, 1991.

N. Carriero and D. Gelernter. Linda in Context. *Commun. ACM*, 32(4):444–458, 1989.

J. Carter, A. Ranganathan, and S. Susarla. Khazana: An Infrastructure for Building Distributed Service. In *Proc. 18th Int'l Conf. on Distributed Computing Systems*, pp. 562–571, Amsterdam, The Netherlands. IEEE, 1998.

M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proc. 16th ACM Symp. on Operating System Principles*, pp. 102–115, Saint-Malo, France. ACM, 1997.

V. Cate. Alex — A Global Filesystem. In *Proc. USENIX File Systems Workshop*, pp. 1–12, Ann Arbor, MI. USENIX, 1992.

K. M. Chandy, A. Rifkin, P. A. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A World-Wide Distributed Sytem Using Java and the Internet. In *Proc. Fifth IEEE Int'l Symp. on High Performance Distributed Computing*, pp. 11–18, Syracuse, NY. IEEE, 1996.

J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. *ACM Oper. Syst. Rev.*, 23(5):147–158, 1989.

J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Lightweight Shared Objects in a 64-Bit Operating System. In *Proc. Seventh Conf. on Object-Oriented Programming System, Languages, and Applications*, volume 27 of *SIGPLAN Notices*, pp. 397–413, Vancouver, Canada. ACM, 1992.

D. R. Cheriton and T. P. Mann. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Trans. Comp. Syst.*, 7(2):147–183, 1989.

D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proc. 14th ACM Symp. on Operating System Principles*, pp. 44–57, Asheville, NC. ACM, 1993.

T. Cooper and M. Wise. Critique of Orthogonal Persistence. In *Proc. Fifth Int'l Workshop on Object Orientation in Operating Systems*, pp. 122–126, Seattle, WA. IEEE, 1996.

M. Crispin. Internet Message Access Protocol — Version 4rev1. RFC 2060, IETF, 1996.

W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proc. 13th ACM Symp. on Operating System Principles*, pp. 122–136, Asilomar, CA. ACM, 1991.

M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. 13th Int'l Symp. on Computer Architecture*, pp. 434–442, Tokyo, Japan. IEEE, 1986.

G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

M. J. Feeley and H. M. Levy. Distributed Shared Memory with Versioned Objects. In *Proc. Seventh Conf. on Object-Oriented Programming System, Languages, and Applications*, volume 27 of *SIGPLAN Notices*, pp. 247–262, Vancouver, Canada. ACM, 1992.

P. J. P. Ferreira and M. Shapiro. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In *Proc. 16th Int'l Conf. on Distributed Computing Systems*, pp. 394–401, Hong Kong. IEEE, 1996.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, IETF, 1997.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, 1999.

I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proc. Seventh IPPS/SPDP Heterogeneous Computing Workshop*, pp. 4–18, Orlando, FL. IEEE, 1998.

I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proc. Seventh Int'l Workshop on Quality of Service*, London, UK. IEEE, 1999.

I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proc. Fifth ACM Conf. on Computer and Comm. Security*, pp. 83–92, San Francisco, CA. ACM, 1998.

I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Par. Distr. Computing*, 37(1):70–82, 1996.

N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions. RFC 2045, IETF, 1996.

M. Fuchs. Garbage Collection on an Open Network. In *Proc. Int'l Workshop on Memory Management*, volume 986 of *Lect. Notes Comput. Sc.*, pp. 251–265, Berlin. Springer-Verlag, 1995.

A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. Softw. Eng.*, 24(5):342–361, 1998.

K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th Int'l Symp. on Computer Architecture*, pp. 15–26, Seattle, WA. IEEE, 1990.

D. K. Gifford. Weighted Voting for replicated data. In *Proc. Seventh ACM Symp. on Operating System Principles*, pp. 150–162, Pacific Grove, CA. 1979.

I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proc. Sixth USENIX Security Symp.*, pp. 1–14, San Jose, CA. 1996.

R. A. Golding. End-to-end Performance Prediction for the Internet — Progress Report. Technical Report UCSC-CRL-92-26, Computer and Information Sciences Board, University of California, Santa Cruz, CA, 1992a.

R. A. Golding. *Weak-consistency Group Communication and Membership*. PhD thesis, University of California, Santa Cruz, CA, 1992b.

L. Gong. A Secure Identity-Based Capability System. In *Proc. IEEE Symp. on Security and Privacy*, pp. 56–63, Oakland, CA. 1989.

L. Gong. Optimal Authentication Protocols Resistant to Password Guessing Attacks. In *Proc. Eighth IEEE Computer Security Foundations Workshop*, pp. 24–29, County Kerry, Ireland. 1995.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Reading, MA, 2nd edition, 2000.

J. Gosling, D. S. H. Rosenthal, and M. J. Arden. *The NeWS Book: An Introduction to the Network/extensible Window System*. Springer-Verlag, New York, NY, 1989.

R. Gray. *Agent Tcl: A Flexible and Secure Mobile-agent System*. PhD thesis, Dept. of Computer Science, Dartmouth College, Hanover, NH, 1997.

A. S. Grimshaw. Easy to Use Object-Oriented Parallel Programming with Mentat. *Computer*, 26(5):39–51, 1993.

A. S. Grimshaw, M. J. Lewis, A. J. Ferrari, and J. F. Karpovich. Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems. Technical Report CS-98-12, University of Virginia, Department of Computer Science, Charlottesville, VA, 1998.

A. S. Grimshaw and W. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Commun. ACM*, 40(1):39–45, 1997.

A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for Specifying the Location of Services (DNS SRV). RFC 2782, IETF, 2000.

G. Hamilton, M. L. Powell, and J. Mitchell. Subcontract: A Flexible Base for Distributed Programming. In *Proc. 14th ACM Symp. on Operating System Principles*, pp. 69–79, Asheville, NC. 1993.

F. J. Hauck, M. van Steen, and A. S. Tanenbaum. Algorithmic Design of the Globe Location Service, Basic Update Operations. Technical Report IR-413, Dept. of Math. and Comp. Sci., Vrije Universiteit, Amsterdam, The Netherlands, 1996.

S. Heker, J. Reynolds, and C. Weider. Technical Overview of Directory Services Using the X.500 Protocol. RFC 1309, IETF, 1992.

M. P. Herlihy and J. M. Wing. Linearization: A Correctness Condition for Concurrent Objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.

U. Hölzle and D. Ungar. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Trans. Prog. Lang. Syst.*, 18(4):355–400, 1996.

C. Hornig. A Standard for the Transmission of IP Datagrams over Ethernet Networks. RFC 894, IETF, 1984.

J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Trans. Comp. Syst.*, 6(1):51–81, 1988.

R. L. Hudson, R. Morrison, E. B. Moss, and D. S. Munro. Garbage Collecting the World: One Car at a Time. *SIGPLAN Notices*, 32(10):162–175, 1997.

N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Trans. Softw. Eng.*, 17(1):64–76, 1991.

D. B. Ingham, M. C. Little, S. J. Caughey, and S. K. Shrivastava. W3Objects: Bringing Object-Oriented Technology to the Web. In *Proc. Fourth Int'l World-Wide Web Conf.*, pp. 89–105, Boston, MA. 1995.

ITU. The Directory: Authentication Framework. ITU-T Recommendation X.509, International Telecommunications Union, Geneva, Switzerland, 1997.

R. Jain. Reducing Traffic Impacts of PCS using Hierarchical User Location Databases. In *Proc. IEEE Int'l Conf. Comm.*, pp. 1153–1157, Dallas, TX. 1996.

E. Jul, H. M. Levy, N. C. Hutchinson, and A. P. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. Comp. Syst.*, 6(1):109–133, 1988.

M. Kamath and K. Ramamritham. Performance Characteristics of Epsilon Serializability with Hierarchical Inconsistency Bounds. In *Proc. Nineth Int'l Conf. on Data Engineering*, pp. 587–594, Vienna, Austria. IEEE, 1993.

B. Kantor and P. Lapsley. Network News Transfer Protocol. RFC 977, IETF, 1986.

J. F. Karpovich. Support for Object Placement in Wide Area Heterogeneous Distributed Systems. Technical Report CS-96-03, University of Virginia, Charlottesville, VA, 1996.

P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. 19th Int'l Symp. on Computer Architecture*, pp. 13–21, Gold Coast, Queensland, Australia. ACM, 1992.

O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *Proc. USENIX Workshop on Smartcard Technology*, pp. 9–20, Chicago, IL. 1999.

L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

B. Lang, C. Quenniac, and J. Piquer. Garbage Collecting the World. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, SIGPLAN Notices, pp. 39–50, Albequerque, NM. 1992.

K. Langendoen, R. Bhoedjang, and H. E. Bal. Models for Asynchronous Message Handling. *IEEE Concurrency*, 5(2):28–37, 1997.

M. Laubach. Classical IP and ARP over ATM. RFC 1577, IETF, 1994.

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

M. J. Lewis and A. S. Grimshaw. The Core Legion Object Model. In *Proc. Fifth IEEE Int'l Symp. on High Performance Distributed Computing*, pp. 551–561, Syracuse, NY. 1996.

K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comp. Syst.*, 7(4):321–359, 1989.

H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM*, 26(6):419–429, 1983.

J. Linn. Privacy Enhancement for Internet Electronic Mail. RFC 1421, IETF, 1993.

J. Linn. Generic Security Service Application Program Interface, Version 2. RFC 2078, IETF, 1997.

B. Liskov. Distributed Programming in Argus. *Commun. ACM*, 31(3):300–312, 1988.

B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawhat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. ACM Int. Conf. on Management of Data*, pp. 318–329, Montreal, Canada. 1996.

U. Maheshwari and B. Liskov. Collecting Distributed Garbage Cycles by Back Tracing. In *Proc. 16th ACM Symp. on Principles of Distributed Systems*, pp. 239–248, Santa Barbara, CA. 1997.

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.

Microsoft Corporation. The Component Object Model Specification, Draft Version 0.9. Technical report, Microsoft Corporation and Digital Equipment Corporation, Redmond, WA, 1995.

Microsoft Corporation. DCOM: A Business Overview. Technical report, Microsoft Corporation, Redmond, WA, 1997.

D. L. Mills. Internet Time Synchronization: The Network Time Protocol. *IEEE Trans. Commun.*, 39(10):1482–1493, 1991.

J. Mitchell, J. J. Giobbons, G. Hamilton, P. B. Kessler, M. Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. In *Proc. 39th IEEE Int'l Computer Conf.*, pp. 122–131, San Francisco, CA. 1994.

P. Mockapetris. Domain Names – Concepts and Facilities. RFC 1034, IETF, 1987.

D. Mosberger. Memory Consistency Models. *ACM Oper. Syst. Rev.*, 27(1):18–26, 1993.

MPI Forum. MPI: A Message-passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, 1995.

A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proc. 16th ACM Symp. on Operating System Principles*, pp. 129–142, Saint-Malo, France. 1997.

J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939, IETF, 1996.

R. M. Needham. Names. In S. J. Mullender (ed.), *Distributed Systems*, chap. 12, pp. 315–327. ACM Press, New York, NY, 1993.

B. C. Neuman. Prospero: A Tool for Organizing Internet Resources. *Electronic Networking: Research, Applications and Policy*, 2(1):30–37, 1992a.

B. C. Neuman. *The Virual System Model: A Scalable Approach to Organizing Large Systems*. PhD thesis, University of Washington, Seattle, WA, 1992b.

B. C. Neuman. Scale in Distributed Systems. In T. L. Casavant and M. Singhal (eds.), *Readings in Distributed Computing Systems*, pp. 463–489. IEEE Computer Society Press, Los Alamitos, CA, 1994.

A. Nguyen-Tuong and A. S. Grimshaw. Building Robust Distributed Applications With Replective Transformations. Technical Report CS-97-26, University of Virginia, Department of Computer Science, Charlottesville, VA, 1997.

B. Nowicki. NFS: Network File System Protocol specification. RFC 1094, IETF, 1989.

Object Management Group. CORBAservices: Common Object Services Specification. Technical Report 98-12-09, OMG, Needham, MA, 1998.

Object Management Group. The Common Object Request Broker: Architecture and Specification, version 2.3.1. Technical Report 99-10-07, OMG, Needham, MA, 1999.

C. Oppedahl. NSI Flawed Domain Name Policy information page. WWW document, 1997a. http://www.patents.com/nsi.sht.

C. Oppedahl. Remedies in Domain Name Lawsuits: How is a Domain Name Like a Cow? *John Marshall J. of Comp. & Inform. Law*, 15(3):437, 1997b.

OSF. Distributed Computing Environment. Technical Report OSF-DCE-PD-1090-4, Open Software Foundation, Cambridge, MA, 1992.

J. K. Ousterhout. Tcl: An Embeddable Command Language. Technical Report CSD-89-541, University of California, Berkeley, CA, 1989.

J. K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *Computer*, 31(3):23–30, 1998.

C. Perkins. IP Mobility Support. RFC 2002, IETF, 1996.

K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. 16th ACM Symp. on Operating System Principles*, pp. 288–301, Saint-Malo, France. 1997.

G. Pierre. Private communication, 2000.

E. Pitoura and G. Samatas. Locating Objects in Mobile Computing. Technical Report 98-020, University of Ioannina, Greece, 1998. To appear in IEEE Transaction on Knowledge and Data Engineering.

D. Plainfossé and M. Shapiro. A Survey of Distributed Garbage Collection Techniques. In *Proc. Int'l Workshop on Memory Management*, volume 986 of *Lect. Notes Comput. Sc.*, pp. 211–249, Berlin. Springer-Verlag, 1995.

J. B. Postel. User Datagram Protocol. RFC 768, IETF, 1980.

J. B. Postel. Internet Protocol. RFC 791, IETF, 1981a.

J. B. Postel. Transmission Control Protocol. RFC 793, IETF, 1981b.

J. B. Postel. Simple mail Transfer Protocol. RFC 821, IETF, 1982.

J. B. Postel and J. Reynolds. File Transfer Protocol. RFC 959, IETF, 1985.

D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, A Distributed System. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 31–37, Seattle, WA. 1992.

R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A General-Purpose Programming Language. *Software – Practice & Experience*, 21(1):91–118, 1991.

K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE Trans. Know. Data Eng.*, 7(6):997–1007, 1995.

D. Rand. The PPP Compression Control Protocol (CCP). RFC 1962, IETF, 1996.

H. C. Rao and L. L. Peterson. Accessing Files in an Internet: The Jade File System. *IEEE Trans. Softw. Eng.*, 19(6):613–624, 1993.

Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918, IETF, 1996.

J. E. Richardson, M. J. Carey, and D. T. Schuh. The Design of the E Programming Language. *ACM Trans. Prog. Lang. Syst.*, 15(3):494–534, 1993.

R. L. Rivest and B. Lampson. SDSI – A Simple Distributed Security Infrastructure. Technical report, MIT, Cambridge, MA, 1996.

R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris. Operating System Support for Persistant and Recoverable Computations. *Commun. ACM*, 39(9):62–69, 1996.

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. CHORUS Distributed Operating System. *Computing Systems*, 1(4):305–370, 1988.

V. F. Russo. *An Object Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, 1991.

J. H. Saltzer. Naming and Binding of Objects. In R. Bayer, R. M. Graham, and G. Seegmüller (eds.), *Operating Systems – An Advanced Course*, volume 60 of *Lect. Notes Comput. Sc.*, pp. 99–208. Springer-Verlag, Berlin, 1978.

H. S. Sandhu and S. Zhou. Cluster-Based File Replication in Large-Scale Distributed Systems. In *Proc. Int'l. Conf. on Measurement and Modeling of Computer Systems*, pp. 91–102, Newport, RI. ACM, 1992.

F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–320, 1990.

M. Shapiro. A Binding Protocol for Distributed Shared Objects. In *Proc. 14th Int'l Conf. on Distributed Computing Systems*, pp. 134–141, Poznan, Poland. IEEE, 1994.

M. Shapiro, P. Dickman, and D. Plainfossé. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Technical Report 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt, France, 1992.

M. Shapiro, S. Kloosterman, and F. Riccardi. PerDiS — a Persistent Distributed Store for Cooperative Applications. In *Proc. Third Cabernet Plenary Workshop*, Rennes, France. 1997.

W. Simpson. The Point-to-Point Protocol (PPP). RFC 1661, IETF, 1994.

K. Sklower, B. Lloyd, G. McGregor, D. Carr, and T. Coradetti. The PPP Multilink Protocol (MP). RFC 1990, IETF, 1996.

K. Sollins and L. Masinter. Functional Requirements for Uniform Resource Names. RFC 1737, IETF, 1994.

H. Soulard and M. Makpangou. A Generic Fragmented Object Structured Framework for Distributed Storage Support. In *Proc. Third Int'l Workshop on Object Orientation in Operating Systems*, pp. 57–65, Dourdan, France. IEEE, 1992.

M. Spasojevic, M. Bowman, and A. Spector. Using a Wide-Area File System Within the World Wide Web. In *Proc. Second Int'l World-Wide Web Conf.*, Chicago, IL. NCSA, 1994.

J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proc. Winter USENIX Technical Conf.*, pp. 191–202, Dallas, TX. USENIX, 1988.

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

Sun Microsystems, Inc. Java Remote Method Invocation Specification, version 1.50. Available by FTP, 1998.

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

A. Taivalsaari. On the Notion of Inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

A. S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, N.J., 1995.

A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *Proc. Sixth Int'l Conf. on Distributed Computing Systems*, pp. 558–563, Cambridge, MA. IEEE, 1986.

A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. J. Mullender, A. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Commun. ACM*, 33(12):46–63, 1990.

D. B. Terry, K. Petersen, M. J. Spreitzer, and M. M. Theimer. The Case for Non-transparent Replication: Examples from Bayou. *IEEE Data Engineering*, 21(4): 12–20, 1998.

A. Vahdat. *Operating System Services For Wide Area Applications*. PhD thesis, University of California, Berkeley, CA, 1998.

A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proc. Seventh IEEE Int'l Symp. on High Performance Distributed Computing*, pp. 52–63, Chicago, IL. 1998.

L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure Network Objects. In *Proc. IEEE Security and Privacy Conf.*, volume 1603 of *Lect. Notes Comput. Sc.*, pp. 395–412, Berlin. Springer-Verlag, 1999.

L. van Doorn, P. Homburg, and A. S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Proc. IEEE Workshop on Hot Topics in Operating Systems*, pp. 86–89, Orcas Island, WA. IEEE, 1995.

R. van Renesse. Causal Controversy at Le Mont St.-Michel. *ACM Oper. Syst. Rev.*, 27 (2):44– 53, 1993.

R. van Renesse. Goal-Oriented Programming, or Composition using Events, or Threads Considered Harmful. In *Proc. Eighth SIGOPS European Workshop*, pp. 82–87, Sintra, Portugal. ACM, 1998.

R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible Group Communication System. *Commun. ACM*, 39(4):76–83, 1996.

J. Z. Wang. A Fully Distributed Location Registration Strategy for Universal Personal Communication Systems. *IEEE J. Sel. Areas in Commun.*, 11(6):850–860, 1993.

M. J. Wiener. Efficient DES Key Search. In *Proc. Crypto '93*, Practical Cryptography for Data Internetworks, pp. 31–79, Los Alamitos, CA. IEEE Computer Society Press, 1996.

R. J. Wieringa and W. de Jonge. Object Identifiers, Keys, and Surrogates — Object Identifiers Revisited. *Theory and Practice of Object Systems*, 1(2):101–114, 1995.

C. J. Wilkenloh, U. Ramachandran, S. Menon, R. J. LeBlanc, M. Y. A. Khalidi, P. W. Hutto, P. Dasgupta, R. C. Chen, J. M. Bernabeu, W. F. Appelbe, and M. Ahamad. The Clouds Experience: Building an Object-Based Distributed Operating System. In *Proc. USENIX Distributed & Multiprocessor Systems Workshop*, pp. 333–347, Fort Lauderdale, FL. 1989.

P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. Int'l Workshop on Memory Management*, volume 637 of *Lect. Notes Comput. Sc.*, Berlin. Springer-Verlag, 1992.

P. R. Wilson. Uniprocessor Garbage Collection Techniques. Accepted for publication in ACM Comput. Surv., 2000.

E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. *ACM Trans. Comp. Syst.*, 12(1):3–32, 1994.

W. A. Wulf, C. Wang, and D. Kienzle. A New Model of Security for Distributed Systems. In *Proc. ACM New Security Paradigms Workshop*, pp. 34–43, Lake Arrowhead, CA. 1996.

A. K. Yeo, A. L. Ananda, and E. K. Koh. A Taxonomy of Issues in Name Systems Design and Implementation. *ACM Oper. Syst. Rev.*, 27(3):4–18, 1993.

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

List of Citations

Index

Publications

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

P. Homburg, M. van Steen, and A. S. Tanenbaum. An Architecture for a Wide Area Distributed System. In *Proc. Seventh SIGOPS European Workshop*, pp. 75–82, Connemara, Ireland. ACM, 1996a.

P. Homburg, M. van Steen, and A. S. Tanenbaum. Communication in Globe: An Object-Based Worldwide Operating System. In *Proc. Fifth Int'l Workshop on Object Orientation in Operating Systems*, pp. 43–47, Seattle, WA. IEEE, 1996b.

P. Homburg, M. van Steen, and A. S. Tanenbaum. Distributed Shared Objects as a Communication Paradigm. In *Proc. Second ASCI Annual Conf.*, pp. 132–137, Lommel, Belgium. ASCI, 1996c.

J. Leiwo, C. Hänle, P. Homburg, C. Gamage, and A. S. Tanenbaum. A Security design for a wide-area distributed system. In *Proc. Second Int'l Conf. Information Security and Cryptography*, volume 1787 of *Lect. Notes Comput. Sc.*, pp. 236–256, Berlin. Springer-Verlag, 1999.

J. Leiwo, C. Hänle, P. Homburg, and A. S. Tanenbaum. Disallowing Unauthorized State Changes in Distributed Shared Objects. In S. Qing and J. H. P. Eloff (eds.), *16th Annual Working Conf. on Information Security (IFIP TC11)*, pp. 381–390, Beijing, PRC. Kluwer Academic Publishers, 2000.

L. van Doorn, P. Homburg, and A. S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Proc. IEEE Workshop on Hot Topics in Operating Systems*, pp. 86–89, Orcas Island, WA. IEEE, 1995.

M. van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*, 36(1):104–109, 1998.

M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, 1999.

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