Chapter 10. Distributed Object-Based Systems

With this chapter, we switch from our discussion of principles to an examination of various paradigms that are used to organize distributed systems. The first paradigm consists of distributed objects. In distributed object-based systems, the notion of an object plays a key role in establishing distribution transparency. In principle, everything is treated as an object and clients are offered services and resources in the form of objects that they can invoke.

Distributed objects form an important paradigm because it is relatively easy to hide distribution aspects behind an object's interface. Furthermore, because an object can be virtually anything, it is also a powerful paradigm for building systems. In this chapter, we will take a look at how the principles of distributed systems are applied to a number of well-known object-based systems. In particular, we cover aspects of CORBA, Java-based systems, and Globe.

10.1. Architecture

Object orientation forms an important paradigm in software development. Ever since its introduction, it has enjoyed a huge popularity. This popularity stems from the natural ability to build software into well-defined and more or less independent components. Developers could concentrate on implementing specific functionality independent from other developers.
[Page 444]

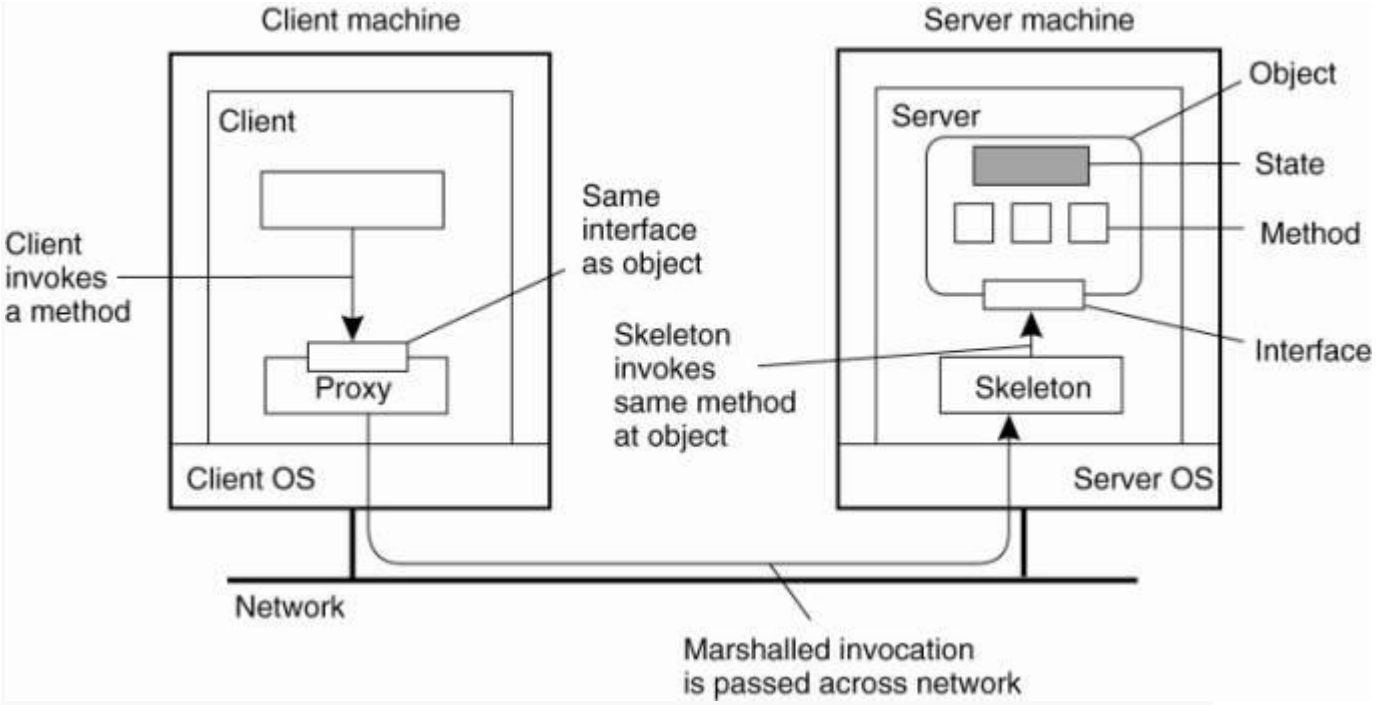
Object orientation began to be used for developing distributed systems in the 1980s. Again, the notion of an independent object hosted by a remote server while attaining a high degree of distribution transparency formed a solid basis for developing a new generation of distributed systems. In this section, we will first take a deeper look into the general architecture of object-based distributed systems, after which we can see how specific principles have been deployed in these systems.

10.1.1. Distributed Objects

The key feature of an object is that it encapsulates data, called the state, and the operations on those data, called the methods. Methods are made available through an interface. It is important to understand that there is no "legal" way a process can access or manipulate the state of an object other than by invoking methods made available to it via an object's interface. An object may implement multiple interfaces. Likewise, given an interface definition, there may be several objects that offer an implementation for it.

This separation between interfaces and the objects implementing these interfaces is crucial for distributed systems. A strict separation allows us to place an interface at one machine, while the object itself resides on another machine. This organization, which is shown in Fig. 10-1, is commonly referred to as a distributed object.

Figure 10-1. Common organization of a remote object with client-side proxy.



When a client binds to a distributed object, an implementation of the object's interface, called a proxy, is then loaded into the client's address space. A proxy is analogous to a client stub in RPC systems. The only thing it does is marshal method invocations into messages and unmarshal reply messages to return the result of the method invocation to the client. The actual object resides at a server machine, where it offers the same interface as it does on the client machine. Incoming invocation requests are first passed to a server stub, which unmarshals them to make method invocations at the object's interface at the server. The server stub is also responsible for marshaling replies and forwarding reply messages to the client-side proxy.
[Page 445]

The server-side stub is often referred to as a skeleton as it provides the bare means for letting the server middleware access the user-defined objects. In practice, it often contains incomplete code in the form of a language-specific class that needs to be further specialized by the developer.

A characteristic, but somewhat counterintuitive feature of most distributed objects is that their state is not distributed: it resides at a single machine. Only the interfaces implemented by the object are made available on other machines. Such objects are also referred to as remote objects. In a general distributed object, the state itself may be physically distributed across multiple machines, but this distribution is also hidden from clients behind the object's interfaces.

Compile-Time versus Runtime Objects

Objects in distributed systems appear in many forms. The most obvious form is the one that is directly related to language-level objects such as those supported by Java, C++, or other object-oriented languages, which are referred to as compile-time objects. In this case, an object is

defined as the instance of a class. A class is a description of an abstract type in terms of a module with data elements and operations on that data (Meyer, 1997).

Using compile-time objects in distributed systems often makes it much easier to build distributed applications. For example, in Java, an object can be fully defined by means of its class and the interfaces that the class implements. Compiling the class definition results in code that allows it to instantiate Java objects. The interfaces can be compiled into client-side and server-side stubs, allowing the Java objects to be invoked from a remote machine. A Java developer can be largely unaware of the distribution of objects: he sees only Java programming code.

The obvious drawback of compile-time objects is the dependency on a particular programming language. Therefore, an alternative way of constructing distributed objects is to do this explicitly during runtime. This approach is followed in many object-based distributed systems, as it is independent of the programming language in which distributed applications are written. In particular, an application may be constructed from objects written in multiple languages.

When dealing with runtime objects, how objects are actually implemented is basically left open. For example, a developer may choose to write a C library containing a number of functions that can all work on a common data file. The essence is how to let such an implementation appear to be an object whose methods can be invoked from a remote machine. A common approach is to use an object adapter, which acts as a wrapper around the implementation with the sole purpose to give it the appearance of an object. The term adapter is derived from a design pattern described in Gamma et al. (1994), which allows an interface to be converted into something that a client expects. An example object adapter is one that dynamically binds to the C library mentioned above and opens an associated data file representing an object's current state.
[Page 446]

Object adapters play an important role in object-based distributed systems. To make wrapping as easy as possible, objects are solely defined in terms of the interfaces they implement. An implementation of an interface can then be registered at an adapter, which can subsequently make that interface available for (remote) invocations. The adapter will take care that invocation requests are carried out, and thus provide an image of remote objects to its clients. We return to the organization of object servers and adapters later in this chapter.

Persistent and Transient Objects

Besides the distinction between language-level objects and runtime objects, there is also a distinction between persistent and transient objects. A persistent object is one that continues to exist even if it is currently not contained in the address space of any server process. In other words, a persistent object is not dependent on its current server. In practice, this means that the server that is currently managing the persistent object, can store the object's state on secondary storage and then exit. Later, a newly started server can read the object's state from storage into its own address space, and handle invocation requests. In contrast, a transient object is an object that exists only as long as the server that is hosting the object. As soon as that server exits, the object ceases to exist as well. There used to be much controversy about having persistent objects; some people believe that transient objects are enough. To take the discussion away from middleware issues, most object-based distributed systems simply support both types.

10.1.2. Example: Enterprise Java Beans

The Java programming language and associated model has formed the foundation for numerous distributed systems and applications. Its popularity can be attributed to the straightforward support for object orientation, combined with the inherent support for remote method invocation. As we will discuss later in this chapter, Java provides a high degree of access transparency, making it easier to use than, for example, the combination of C with remote procedure calling.

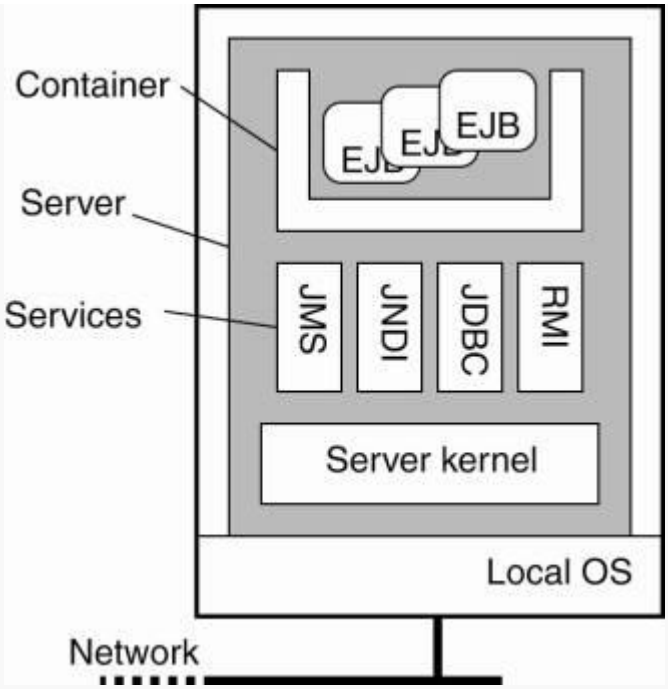
Ever since its introduction, there has been a strong incentive to provide facilities that would ease the development of distributed applications. These facilities go well beyond language support, requiring a runtime environment that supports traditional multitiered client-server architectures. To this end, much work has been put into the development of (Enterprise) Java Beans (EJB).
[Page 447]

An EJB is essentially a Java object that is hosted by a special server offering different ways for remote clients to invoke that object. Crucial is that this server provides the support to separate application functionality from systems-oriented functionality. The latter includes functions for looking up objects, storing objects, letting objects be part of a transaction, and so on. How this separation can be realized is discussed below when we concentrate on object servers. How to develop EJBs is described in detail by Monson-Hafael et al. (2004). The specifications can be found in Sun Microsystems (2005a).

With this separation in mind, EJBs can be pictured as shown in Fig. 10-2. The important issue is that an EJB is embedded inside a container which effectively provides interfaces to underlying services that are implemented by the application server. The container can more or less automatically bind the EJB to these services, meaning that the correct references are readily available to a programmer. Typical services include those for remote method invocation (RMI), database access (JDBC), naming (JNDI), and messaging (JMS). Making use of these services is more or less automated, but does require that the programmer makes a distinction between four kinds of EJBs:

1. Stateless session beans
2. Stateful session beans
3. Entity beans
4. Message-driven beans

Figure 10-2. General architecture of an EJB server.



[Page 448]

As its name suggests, a stateless session bean is a transient object that is invoked once, does its work, after which it discards any information it needed to perform the service it offered to a client. For example, a stateless session bean could be used to implement a service that lists the top-ranked books. In this case, the bean would typically consist of an SQL query that is submitted to an database. The results would be put into a special format that the client can handle, after which its work would have been completed and the listed books discarded.

In contrast, a stateful session bean maintains client-related state. The canonical example is a bean implementing an electronic shopping cart like those widely deployed for electronic commerce. In this case, a client would typically be able to put things in a cart, remove items, and use the cart to go to an electronic checkout. The bean, in turn, would typically access databases for getting current prices and information on number of items still in stock. However, its lifetime would still be limited, which is why it is referred to as a session bean: when the client is finished (possibly having invoked the object several times), the bean will automatically be destroyed.

An entity bean can be considered to be a long-lived persistent object. As such, an entity bean will generally be stored in a database, and likewise, will often also be part of distributed transactions. Typically, entity beans store information that may be needed a next time a specific client access the server. In settings for electronic commerce, an entity bean can be used to record customer information, for example, shipping address, billing address, credit card information, and so on. In these cases, when a client logs in, his associated entity bean will be restored and used for further processing.

Finally, message-driven beans are used to program objects that should react to incoming messages (and likewise, be able to send messages). Message-driven beans cannot be invoked directly by a client, but rather fit into a publish-subscribe way of communication, which we

briefly discussed in Chap. 4. What it boils down to is that a message-driven bean is automatically called by the server when a specific message m is received, to which the server (or rather an application it is hosting) had previously subscribed. The bean contains application code for handling the message, after which the server simply discards it. Message-driven beans are thus seen to be stateless. We will return extensively to this type of communication in Chap. 13.

10.1.3. Example: Globe Distributed Shared Objects

Let us now take a look at a completely different type of object-based distributed system. Globe is a system in which scalability plays a central role. All aspects that deal with constructing a large-scale wide-area system that can support huge numbers of users and objects drive the design of Globe. Fundamental to this approach is the way objects are viewed. Like other object-based systems, objects in Globe are expected to encapsulate state and operations on that state. [Page 449]

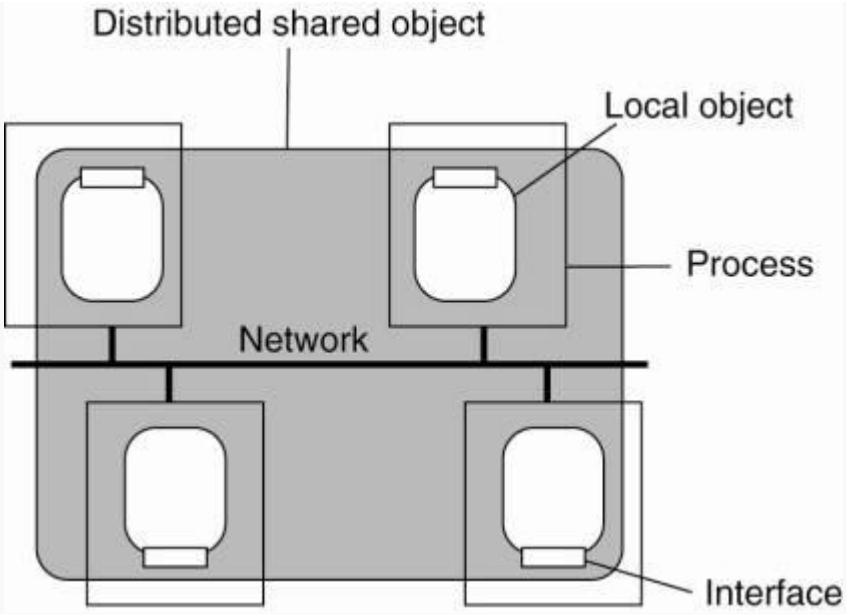
An important difference with other object-based systems is that objects are also expected to encapsulate the implementation of policies that prescribe the distribution of an object's state across multiple machines. In other words, each object determines how its state will be distributed over its replicas. Each object also controls its own policies in other areas as well.

By and large, objects in Globe are put in charge as much as possible. For example, an object decides how, when, and where its state should be migrated. Also, an object decides if its state is to be replicated, and if so, how replication should take place. In addition, an object may also determine its security policy and implementation. Below, we describe how such encapsulation is achieved.

Object Model

Unlike most other object-based distributed systems, Globe does not adopt the remote-object model. Instead, objects in Globe can be physically distributed, meaning that the state of an object can be distributed and replicated across multiple processes. This organization is shown in Fig. 10-3, which shows an object that is distributed across four processes, each running on a different machine. Objects in Globe are referred to as distributed shared objects, to reflect that objects are normally shared between several processes. The object model originates from the distributed objects used in Orca as described in Bal (1989). Similar approaches have been followed for fragmented objects (Makpangou et al., 1994).

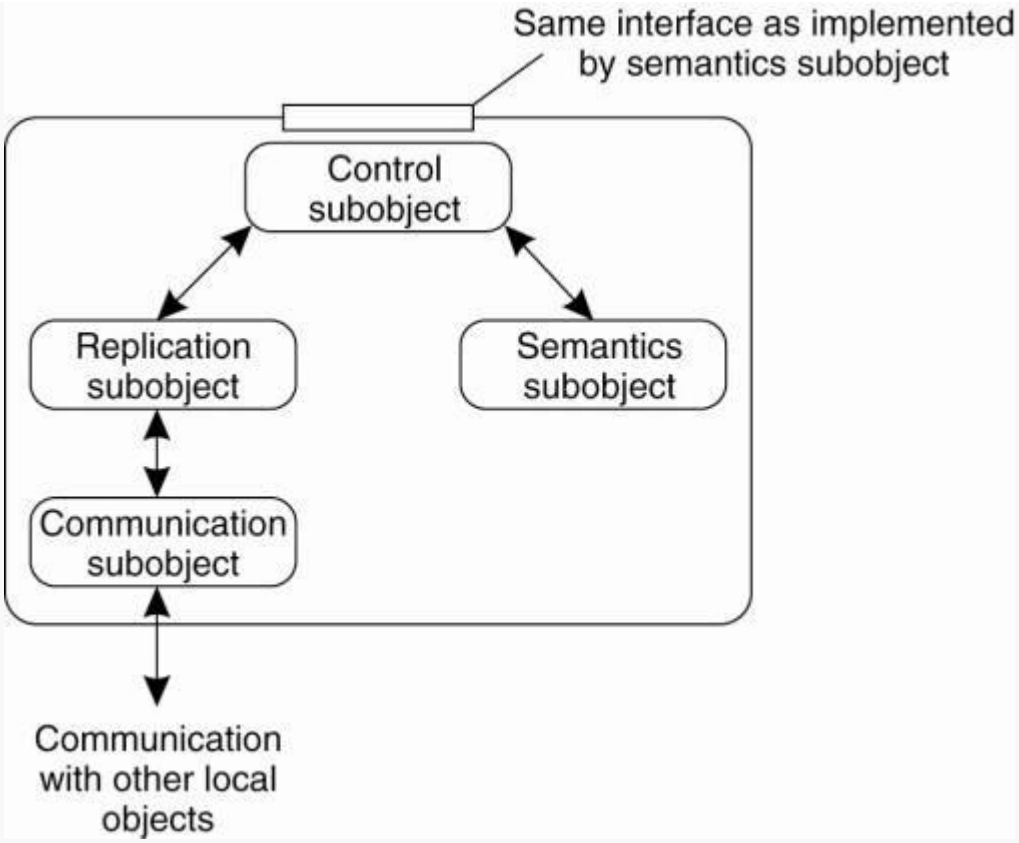
Figure 10-3. The organization of a Globe distributed shared object.



A process that is bound to a distributed shared object is offered a local implementation of the interfaces provided by that object. Such a local implementation is called a local representative, or simply local object. In principle, whether or not a local object has state is completely transparent to the bound process. All implementation details of an object are hidden behind the interfaces offered to a process. The only thing visible outside the local object are its methods. [Page 450]

Globe local objects come in two flavors. A primitive local object is a local object that does not contain any other local objects. In contrast, a composite local object is an object that is composed of multiple (possibly composite) local objects. Composition is used to construct a local object that is needed for implementing distributed shared objects. This local object is shown in Fig. 10-4 and consists of at least four subobjects.

Figure 10-4. The general organization of a local object for distributed shared objects in Globe.



The semantics subobject implements the functionality provided by a distributed shared object. In essence, it corresponds to ordinary remote objects, similar in flavor to EJBs.

The communication subobject is used to provide a standard interface to the underlying network. This subobject offers a number of message-passing primitives for connection-oriented as well as connectionless communication. There are also more advanced communication subobjects available that implement multicasting interfaces. Communication subobjects can be used that implement reliable communication, while others offer only unreliable communication.

Crucial to virtually all distributed shared objects is the replication subobject. This subobject implements the actual distribution strategy for an object. As in the case of the communication subobject, its interface is standardized. The replication subobject is responsible for deciding exactly when a method as provided by the semantics subobject is to be carried out. For example, a replication subobject that implements active replication will ensure that all method invocations are carried out in the same order at each replica. In this case, the subobject will have to communicate with the replication subobjects in other local objects that comprise the distributed shared object.
[Page 451]

The control subobject is used as an intermediate between the user-defined interfaces of the semantics subobject and the standardized interfaces of the replication subobject. In addition, it is responsible for exporting the interfaces of the semantics subobject to the process bound to

the distributed shared object. All method invocations requested by that process are marshaled by the control subobject and passed to the replication subobject.

The replication subobject will eventually allow the control subobject to carry on with an invocation request and to return the results to the process. Likewise, invocation requests from remote processes are eventually passed to the control subobject as well. Such a request is then unmarshaled, after which the invocation is carried out by the control subobject, passing results back to the replication subobject.

10.2. Processes

A key role in object-based distributed systems is played by object servers, that is, the server designed to host distributed objects. In the following, we first concentrate on general aspects of object servers, after which we will discuss the open-source JBoss server.

10.2.1. Object Servers

An object server is a server tailored to support distributed objects. The important difference between a general object server and other (more traditional) servers is that an object server by itself does not provide a specific service. Specific services are implemented by the objects that reside in the server. Essentially, the server provides only the means to invoke local objects, based on requests from remote clients. As a consequence, it is relatively easy to change services by simply adding and removing objects.

An object server thus acts as a place where objects live. An object consists of two parts: data representing its state and the code for executing its methods. Whether or not these parts are separated, or whether method implementations are shared by multiple objects, depends on the object server. Also, there are differences in the way an object server invokes its objects. For example, in a multithreaded server, each object may be assigned a separate thread, or a separate thread may be used for each invocation request. These and other issues are discussed next.

Alternatives for Invoking Objects

For an object to be invoked, the object server needs to know which code to execute, on which data it should operate, whether it should start a separate thread to take care of the invocation, and so on. A simple approach is to assume that all objects look alike and that there is only one way to invoke an object. Unfortunately, such an approach is generally inflexible and often unnecessarily constrains developers of distributed objects.
[Page 452]

A much better approach is for a server to support different policies. Consider, for example, transient objects. Recall that a transient object is an object that exists only as long as its server exists, but possibly for a shorter period of time. An in-memory, read-only copy of a file could typically be implemented as a transient object. Likewise, a calculator could also be implemented as a transient object. A reasonable policy is to create a transient object at the first invocation request and to destroy it as soon as no clients are bound to it anymore.

The advantage of this approach is that a transient object will need a server's resources only as long as the object is really needed. The drawback is that an invocation may take some time to

complete, because the object needs to be created first. Therefore, an alternative policy is sometimes to create all transient objects at the time the server is initialized, at the cost of consuming resources even when no client is making use of the object.

In a similar fashion, a server could follow the policy that each of its objects is placed in a memory segment of its own. In other words, objects share neither code nor data. Such a policy may be necessary when an object implementation does not separate code and data, or when objects need to be separated for security reasons. In the latter case, the server will need to provide special measures, or require support from the underlying operating system, to ensure that segment boundaries are not violated.

The alternative approach is to let objects at least share their code. For example, a database containing objects that belong to the same class can be efficiently implemented by loading the class implementation only once into the server. When a request for an object invocation comes in, the server need only fetch that object's state from the database and execute the requested method.

Likewise, there are many different policies with respect to threading. The simplest approach is to implement the server with only a single thread of control. Alternatively, the server may have several threads, one for each of its objects. Whenever an invocation request comes in for an object, the server passes the request to the thread responsible for that object. If the thread is currently busy, the request is temporarily queued.

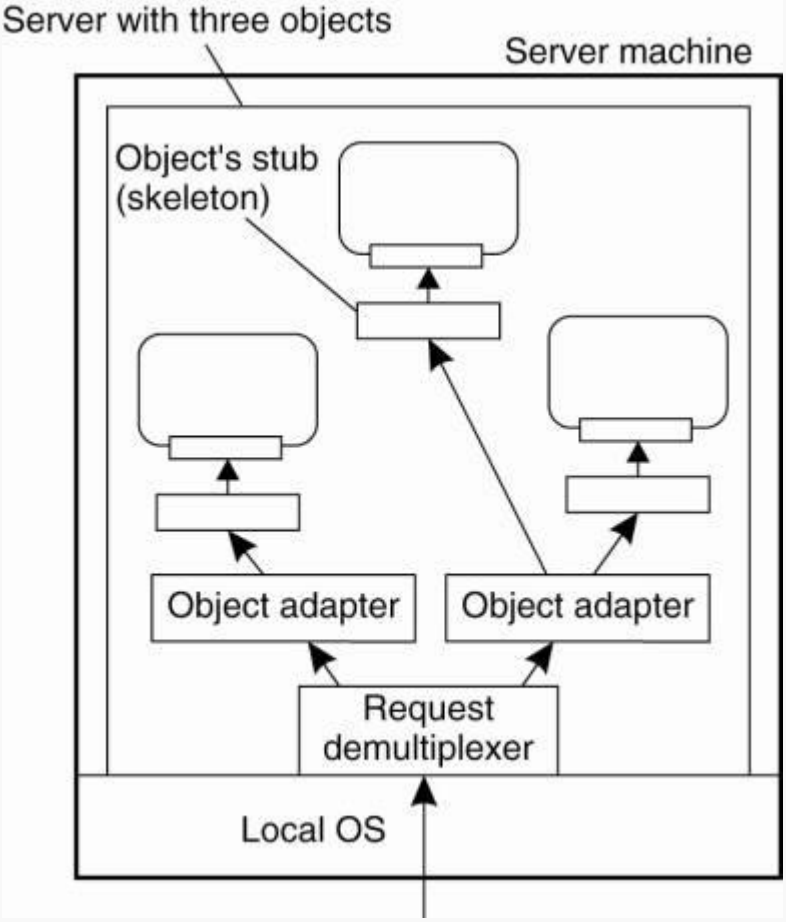
The advantage of this approach is that objects are automatically protected against concurrent access: all invocations are serialized through the single thread associated with the object. Neat and simple. Of course, it is also possible to use a separate thread for each invocation request, requiring that objects should have already been protected against concurrent access. Independent of using a thread per object or thread per method is the choice of whether threads are created on demand or the server maintains a pool of threads. Generally there is no single best policy. Which one to use depends on whether threads are available, how much performance matters, and similar factors.

Object Adapter

Decisions on how to invoke an object are commonly referred to as activation policies, to emphasize that in many cases the object itself must first be brought into the server's address space (i.e., activated) before it can actually be invoked. What is needed then is a mechanism to group objects per policy. Such a mechanism is sometimes called an object adapter, or alternatively an object wrapper. An object adapter can best be thought of as software implementing a specific activation policy. The main issue, however, is that object adapters come as generic components to assist developers of distributed objects, and which need only to be configured for a specific policy.

An object adapter has one or more objects under its control. Because a server should be capable of simultaneously supporting objects that require different activation policies, several object adapters may reside in the same server at the same time. When an invocation request is delivered to the server, that request is first dispatched to the appropriate object adapter, as shown in Fig. 10-5.

Figure 10-5. Organization of an object server supporting different activation policies.



An important observation is that object adapters are unaware of the specific interfaces of the objects they control. Otherwise, they could never be generic. The only issue that is important to an object adapter is that it can extract an object reference from an invocation request, and subsequently dispatch the request to the referenced object, but now following a specific activation policy. As is also o;lustrated in Fig. 10-5, rather than passing the request directly to the object, an adapter hands an invocation request to the server-side stub of that object. The stub, also called a skeleton, is normally generated from the interface definitions of the object, unmarshals the request and invokes the appropriate method.
[Page 454]

An object adapter can support different activation policies by simply configuring it at runtime. For example, in CORBA-compliant systems (OMG, 2004a), it is possible to specify whether an object should continue to exist after its associated adapter has stopped. Likewise, an adapter can be configured to generate object identifiers, or to let the application provide one. As a final example, an adapter can be configured to operate in single-threaded or multithreaded mode as we explained above.

As a side remark, note that although in Fig. 10-5 we have spoken about objects, we have said nothing about what these objects actually are. In particular, it should be stressed that as part of the implementation of such an object the server may (indirectly) access databases or call special library routines. The implementation details are hidden for the object adapter who

communicates only with a skeleton. As such, the actual implementation may have nothing to do with what we often see with language-level (i.e., compile-time) objects. For this reason, a different terminology is generally adopted. A servant is the general term for a piece of code that forms the implementation of an object. In this light, a Java bean can be seen as nothing but just another kind of servant.

10.2.2. Example: The Ice Runtime System

Let us take a look at how distributed objects are handled in practice. We briefly consider the Ice distributed-object system, which has been partly developed in response to the intricacies of commercial object-based distributed systems (Henning, 2004). In this section, we concentrate on the core of an Ice object server and defer other parts of the system to later sections.

An object server in Ice is nothing but an ordinary process that simply starts with initializing the Ice runtime system (RTS). The basis of the runtime environment is formed by what is called a communicator. A communicator is a component that manages a number of basic resources, of which the most important one is formed by a pool of threads. Likewise, it will have associated dynamically allocated memory, and so on. In addition, a communicator provides the means for configuring the environment. For example, it is possible to specify maximum message lengths, maximum invocation retries, and so on.

Normally, an object server would have only a single communicator. However, when different applications need to be fully separated and protected from each other, a separate communicator (with possibly a different configuration) can be created within the same process. At the very least, such an approach would separate the different thread pools so that if one application has consumed all its threads, then this would not affect the other application.
[Page 455]

A communicator can also be used to create an object adapter, such as shown in Fig. 10-6. We note that the code is simplified and incomplete. More examples and detailed information on Ice can be found in Henning and Spruiell (2005).

Figure 10-6. Example of creating an object server in Ice.

```
main(int argc, char* argv[]) {
    Ice::Communicator       ic;
    Ice::ObjectAdapter      adapter;
    Ice::Object             object;
    ic = Ice::initialize(argc, argv);
    adapter =
      ic->createObjectAdapterWithEnd Points( "MyAdapter","tcp -p 10000");
    object = new MyObject;
    adapter->add(object, objectID);
    adapter->activate();
    ic->waitForShutdown();
}
```

In this example, we start with creating and initializing the runtime environment. When that is done, an object adapter is created. In this case, it is instructed to listen for incoming TCP connections on port 10000. Note that the adapter is created in the context of the just created communicator. We are now in the position to create an object and to subsequently add that object to the adapter. Finally, the adapter is activated, meaning that, under the hood, a thread is activated that will start listening for incoming requests.

This code does not yet show much differentiation in activation policies. Policies can be changed by modifying the properties of an adapter. One family of properties is related to maintaining an adapter-specific set of threads that are used for handling incoming requests. For example, one can specify that there should always be only one thread, effectively serializing all accesses to objects that have been added to the adapter.

Again, note that we have not specified MyObject. Like before, this could be a simple C++ object, but also one that accesses databases and other external services that jointly implement an object. By registering MyObject with an adapter, such implementation details are completely hidden from clients, who now believe that they are invoking a remote object.

In the example above, an object is created as part of the application, after which it is added to an adapter. Effectively, this means that an adapter may need to support many objects at the same time, leading to potential scalability problems. An alternative solution is to dynamically load objects into memory when they are needed. To do this, Ice provides support for special objects known as locators. A locator is called when the adapter receives an incoming request for an object that has not been explicitly added. In that case, the request is forwarded to the locator, whose job is to further handle the request.
[Page 456]

To make matters more concrete, suppose a locator is handed a request for an object of which the locator knows that its state is stored in a relational database system. Of course, there is no magic here: the locator has been programmed explicitly to handle such requests. In this case, the object's identifier may correspond to the key of a record in which that state is stored. The locator will then simply do a lookup on that key, fetch the state, and will then be able to further process the request.

There can be more than one locator added to an adapter. In that case, the adapter would keep track of which object identifiers would belong to the same locator. Using multiple locators allows supporting many objects by a single adapter. Of course, objects (or rather their state) would need to be loaded at runtime, but this dynamic behavior would possibly make the server itself relatively simple.

10.3. Communication

We now draw our attention to the way communication is handled in object-based distributed systems. Not surprisingly, these systems generally offer the means for a remote client to invoke an object. This mechanism is largely based on remote procedure calls (RPCs), which we discussed extensively in Chap. 4. However, before this can happen, there are numerous issues that need to be dealt with.

10.3.1. Binding a Client to an Object

An interesting difference between traditional RPC systems and systems supporting distributed objects is that the latter generally provides systemwide object references. Such object references can be freely passed between processes on different machines, for example as parameters to method invocations. By hiding the actual implementation of an object reference, that is, making it opaque, and perhaps even using it as the only way to reference objects, distribution transparency is enhanced compared to traditional RPCs.

When a process holds an object reference, it must first bind to the referenced object before invoking any of its methods. Binding results in a proxy being placed in the process's address space, implementing an interface containing the methods the process can invoke. In many cases, binding is done automatically. When the underlying system is given an object reference, it needs a way to locate the server that manages the actual object, and place a proxy in the client's address space.

With implicit binding, the client is offered a simple mechanism that allows it to directly invoke methods using only a reference to an object. For example, C++ allows overloading the unary member selection operator ("$\rightarrow$") permitting us to introduce object references as if they were ordinary pointers as shown in Fig. 10-7(a). With implicit binding, the client is transparently bound to the object at the moment the reference is resolved to the actual object. In contrast, with explicit binding, the client should first call a special function to bind to the object before it can actually invoke its methods. Explicit binding generally returns a pointer to a proxy that is then become locally available, as shown in Fig. 10-7(b).
[Page 457]

Figure 10-7. (a) An example with implicit binding using only global references. (b) An example with explicit binding using global and local references.

```
Distr_object* obj_ref;      // Declare a systemwide object reference
obj_ref = ...;              // Initialize the reference to a distrib. obj.
obj_refdo_something( );     // Implicitly bind and invoke a method
```

(a)

```
Distr_object obj_ref;       // Declare a systemwide object reference
Local_object* obj_ptr;      // Declare a pointer to local objects
obj_ref = ...;              // Initialize the reference to a distrib. obj.
obj_ptr = bind(obj_ref);    // Explicitly bind and get ptr to local proxy
obj_ptrdo_something( );     // Invoke a method on the local proxy
```

(b)

Implementation of Object References

It is clear that an object reference must contain enough information to allow a client to bind to an object. A simple object reference would include the network address of the machine where the actual object resides, along with an end point identifying the server that manages the object, plus an indication of which object. Note that part of this information will be provided by an object adapter. However, there are a number of drawbacks to this scheme.

First, if the server's machine crashes and the server is assigned a different end point after recovery, all object references have become invalid. This problem can be solved as is done in DCE: have a local daemon per machine listen to a well-known end point and keep track of the server-to-end point assignments in an end point table. When binding a client to an object, we first ask the daemon for the server's current end point. This approach requires that we encode a server ID into the object reference that can be used as an index into the end point table. The server, in turn, is always required to register itself with the local daemon.

However, encoding the network address of the server's machine into an object reference is not always a good idea. The problem with this approach is that the server can never move to another machine without invalidating all the references to the objects it manages. An obvious solution is to expand the idea of a local daemon maintaining an end point table to a location server that keeps track of the machine where an object's server is currently running. An object reference would then contain the network address of the location server, along with a systemwide identifier for the server. Note that this solution comes close to implementing flat name spaces as we discussed in Chap. 5.
[Page 458]

What we have tacitly assumed so far is that the client and server have somehow already been configured to use the same protocol stack. Not only does this mean that they use the same transport protocol, for example, TCP; furthermore it means that they use the same protocol for marshaling and unmarshaling parameters. They must also use the same protocol for setting up an initial connection, handle errors and flow control the same way, and so on.

We can safely drop this assumption provided we add more information in the object reference. Such information may include the identification of the protocol that is used to bind to an object and of those that are supported by the object's server. For example, a single server may simultaneously support data coming in over a TCP connection, as well as incoming UDP datagrams. It is then the client's responsibility to get a proxy implementation for at least one of the protocols identified in the object reference.

We can even take this approach one step further, and include an implementation handle in the object reference, which refers to a complete implementation of a proxy that the client can dynamically load when binding to the object. For example, an implementation handle could take the form of a URL pointing to an archive file, such as ftp://ftp.clientware.org/proxies/java/proxy-v1.1a.zip. The binding protocol would then only need to prescribe that such a file should be dynamically downloaded, unpacked, installed, and subsequently instantiated. The benefit of this approach is that the client need not worry about whether it has an implementation of a specific protocol available. In addition, it gives the object developer the freedom to design object-specific proxies. However, we do need to take special security measures to ensure the client that it can trust the downloaded code.

10.3.2. Static versus Dynamic Remote Method Invocations

After a client is bound to an object, it can invoke the object's methods through the proxy. Such a remote method invocation, or simply RMI, is very similar to an RPC when it comes to issues such as marshaling and parameter passing. An essential difference between an RMI and an RPC is that RMIs generally support systemwide object references as explained above. Also, it is not necessary to have only general-purpose client-side and server-side stubs available. Instead, we can more easily accommodate object-specific stubs as we also explained.

The usual way to provide RMI support is to specify the object's interfaces in an interface definition language, similar to the approach followed with RPCs. Alternatively, we can make use of an object-based language such as Java, that will handle stub generation automatically. This approach of using predefined interface definitions is generally referred to as static invocation. Static invocations require that the interfaces of an object are known when the client application is being developed. It also implies that if interfaces change, then the client application must be recompiled before it can make use of the new interfaces.
[Page 459]

As an alternative, method invocations can also be done in a more dynamic fashion. In particular, it is sometimes convenient to be able to compose a method invocation at runtime, also referred to as a dynamic invocation. The essential difference with static invocation is that an application selects at runtime which method it will invoke at a remote object. Dynamic invocation generally takes a form such as

```
invoke(object, method, input_parameters, output_parameters);
```

where object identifies the distributed object, method is a parameter specifying exactly which method should be invoked, input_parameters is a data structure that holds the values of that method's input parameters, and output_parameters refers to a data structure where output values can be stored.

For example, consider appending an integer int to a file object fobject, for which the object provides the method append. In this case, a static invocation would take the form

```
fobject.append(int)
```

whereas the dynamic invocation would look something like

```
invoke(fobject, id(append), int)
```

where the operation id(append) returns an identifier for the method append.

To illustrate the usefulness of dynamic invocations, consider an object browser that is used to examine sets of objects. Assume that the browser supports remote object invocations. Such a browser is capable of binding to a distributed object and subsequently presenting the object's interface to its user. The user could then be asked to choose a method and provide values for its parameters, after which the browser can do the actual invocation. Typically, such an object browser should be developed to support any possible interface. Such an approach requires that

interfaces can be inspected at runtime, and that method invocations can be dynamically constructed.

Another application of dynamic invocations is a batch processing service to which invocation requests can be handed along with a time when the invocation should be done. The service can be implemented by a queue of invocation requests, ordered by the time that invocations are to be done. The main loop of the service would simply wait until the next invocation is scheduled, remove the request from the queue, and call invoke as given above.
[Page 460]

10.3.3. Parameter Passing

Because most RMI systems support systemwide object references, passing parameters in method invocations is generally less restricted than in the case of RPCs. However, there are some subtleties that can make RMIs trickier than one might initially expect, as we briefly discuss in the following pages.

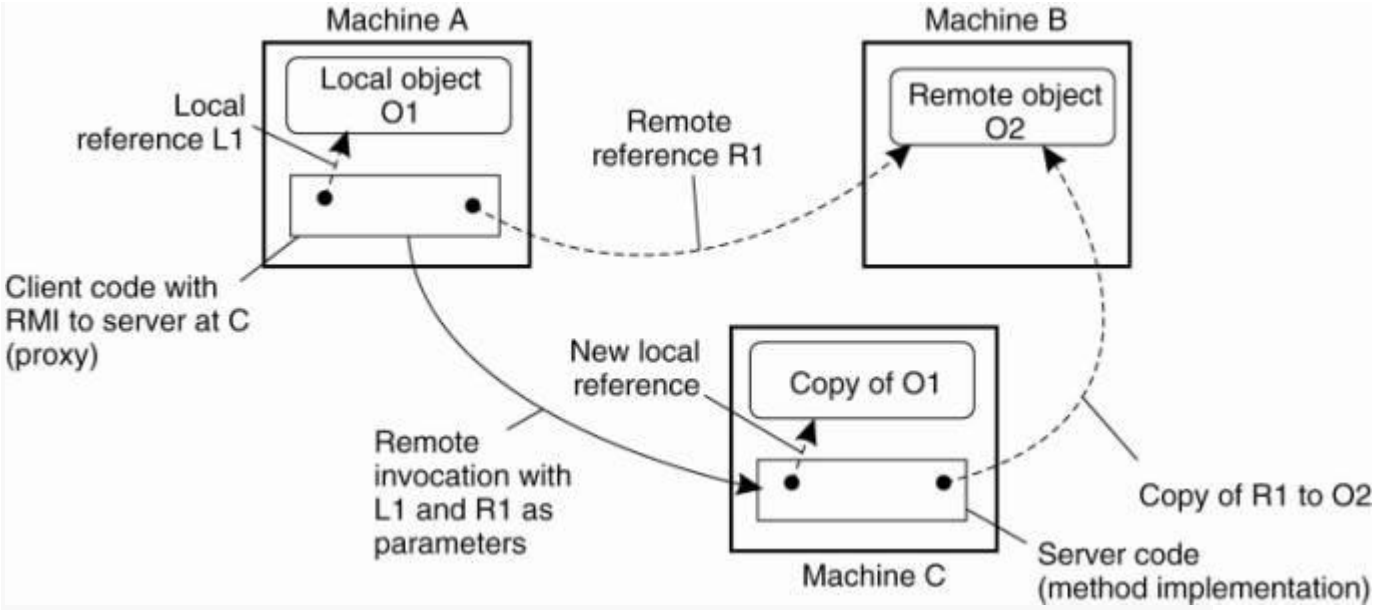
Let us first consider the situation that there are only distributed objects. In other words, all objects in the system can be accessed from remote machines. In that case, we can consistently use object references as parameters in method invocations. References are passed by value, and thus copied from one machine to the other. When a process is given an object reference as the result of a method invocation, it can simply bind to the object referred to when needed later.

Unfortunately, using only distributed objects can be highly inefficient, especially when objects are small, such as integers, or worse yet, Booleans. Each invocation by a client that is not colocated in the same server as the object, generates a request between different address spaces or, even worse, between different machines. Therefore, references to remote objects and those to local objects are often treated differently.

When invoking a method with an object reference as parameter, that reference is copied and passed as a value parameter only when it refers to a remote object. In this case, the object is literally passed by reference. However, when the reference refers to a local object, that is an object in the same address space as the client, the referred object is copied as a whole and passed along with the invocation. In other words, the object is passed by value.

These two situations are illustrated in Fig. 10-8, which shows a client program running on machine A, and a server program on machine C. The client has a reference to a local object O 1 that it uses as a parameter when calling the server program on machine C. In addition, it holds a reference to a remote object O 2 residing at machine B, which is also used as a parameter. When calling the server, a copy of O 1 is passed to the server on machine C, along with only a copy of the reference to O 2.

Figure 10-8. The situation when passing an object by reference or by value.
(This item is displayed on page 461 in the print version)



Note that whether we are dealing with a reference to a local object or a reference to a remote object can be highly transparent, such as in Java. In Java, the distinction is visible only because local objects are essentially of a different data type than remote objects. Otherwise, both types of references are treated very much the same [see also Wollrath et al. (1996)]. On the other hand, when using conventional programming languages such as C, a reference to a local object can be as simple as a pointer, which can never be used to refer to a remote object.

The side effect of invoking a method with an object reference as parameter is that we may be copying an object. Obviously, hiding this aspect is unacceptable, so that we are consequently forced to make an explicit distinction between local and distributed objects. Clearly, this distinction not only violates distribution transparency, but also makes it harder to write distributed applications.

10.3.4. Example: Java RMI

In Java, distributed objects have been integrated into the language. An important goal was to keep as much of the semantics of nondistributed objects as possible. In other words, the Java language developers have aimed for a high degree of distribution transparency. However, as we shall see, Java's developers have also decided to make distribution apparent where a high degree of transparency was simply too inefficient, difficult, or impossible to realize.

The Java Distributed-Object Model

Java also adopts remote objects as the only form of distributed objects. Recall that a remote object is a distributed object whose state always resides on a single machine, but whose interfaces can be made available to remote processes. Interfaces are implemented in the usual way by means of a proxy, which offers exactly the same interfaces as the remote object. A proxy itself appears as a local object in the client's address space.

There are only a few, but subtle and important, differences between remote objects and local objects. First, cloning local or remote objects are different. Cloning a local object O results in a new object of the same type as O with exactly the same state. Cloning thus returns an exact copy of the object that is cloned. These semantics are hard to apply to a remote object. If we were to make an exact copy of a remote object, we would not only have to clone the actual object at its server, but also the proxy at each client that is currently bound to the remote object. Cloning a remote object is therefore an operation that can be executed only by the server. It results in an exact copy of the actual object in the server's address space. Proxies of the actual object are thus not cloned. If a client at a remote machine wants access to the cloned object at the server, it will first have to bind to that object again.

Java Remote Object Invocation

As the distinction between local and remote objects is hardly visible at the language level, Java can also hide most of the differences during a remote method invocation. For example, any primitive or object type can be passed as a parameter to an RMI, provided only that the type can be marshaled. In Java terminology, this means that it must be serializable. Although, in principle, most objects can be serialized, serialization is not always allowed or possible. Typically, platform-dependent objects such as file descriptors and sockets, cannot be serialized.

The only distinction made between local and remote objects during an RMI is that local objects are passed by value (including large objects such as arrays), whereas remote objects are passed by reference. In other words, a local object is first copied after which the copy is used as parameter value. For a remote object, a reference to the object is passed as parameter instead of a copy of the object, as was also shown in Fig. 10-8.

In Java RMI, a reference to a remote object is essentially implemented as we explained in Sec. 10.3.3. Such a reference consists of the network address and end point of the server, as well as a local identifier for the actual object in the server's address space. That local identifier is used only by the server. As we also explained, a reference to a remote object also needs to encode the protocol stack that is used by a client and the server to communicate. To understand how such a stack is encoded in the case of Java RMI, it is important to realize that each object in Java is an instance of a class. A class, in turn, contains an implementation of one or more interfaces.

In essence, a remote object is built from two different classes. One class contains an implementation of server-side code, which we call the server class. This class contains an implementation of that part of the remote object that will be running on a server. In other words, it contains the description of the object's state, as well as an implementation of the methods that operate on that state. The server-side stub, that is, the skeleton, is generated from the interface specifications of the object.

The other class contains an implementation of the client-side code, which we call the client class. This class contains an implementation of a proxy. Like the skeleton, this class is also generated from the object's interface specification. In its simplest form, the only thing a proxy does is to convert each method call into a message that is sent to the server-side implementation of the remote object, and convert a reply message into the result if a method call. For each call, it sets up a connection with the server, which is subsequently torn down when the call is finished. For this purpose, the proxy needs the server's network address and end point as

mentioned above. This information, along with the local identifier of the object at the server, is always stored as part of the state of a proxy.
[Page 463]

Consequently, a proxy has all the information it needs to let a client invoke methods of the remote object. In Java, proxies are serializable. In other words, it is possible to marshal a proxy and send it as a series of bytes to another process, where it can be unmarshaled and used to invoke methods on the remote object. In other words, a proxy can be used as a reference to a remote object.

This approach is consistent with Java's way of integrating local and distributed objects. Recall that in an RMI, a local object is passed by making a copy of it, while a remote object is passed by means of a systemwide object reference. A proxy is treated as nothing else but a local object. Consequently, it is possible to pass a serializable proxy as parameter in an RMI. The side effect is that such a proxy can be used as a reference to the remote object.

In principle, when marshaling a proxy, its complete implementation, that is, all its state and code, is converted to a series of bytes. Marshaling the code like this is not very efficient and may lead to very large references. Therefore, when marshaling a proxy in Java, what actually happens is that an implementation handle is generated, specifying precisely which classes are needed to construct the proxy. Possibly, some of these classes first need to be downloaded from a remote site. The implementation handle replaces the marshaled code as part of a remote-object reference. In effect, references to remote objects in Java are in the order of a few hundred bytes.

This approach to referencing remote objects is highly flexible and is one of the distinguishing features of Java RMI (Waldo, 1998). In particular, it allows for object-specific solutions. For example, consider a remote object whose state changes only once in a while. We can turn such an object into a truly distributed object by copying the entire state to a client at binding time. Each time the client invokes a method, it operates on the local copy. To ensure consistency, each invocation also checks whether the state at the server has changed, in which case the local copy is refreshed. Likewise, methods that modify the state are forwarded to the server. The developer of the remote object will now have to implement only the necessary client-side code, and have it dynamically downloaded when the client binds to the object.

Being able to pass proxies as parameters works only because each process is executing the same Java virtual machine. In other words, each process is running in the same execution environment. A marshaled proxy is simply unmarshaled at the receiving side, after which its code can be executed. In contrast, in DCE for example, passing stubs is out of the question, as different processes may be running in execution environments that differ with respect to language, operating system, and hardware. Instead, a DCE process first needs to (dynamically) link in a locally-available stub that has been previously compiled specifically for the process's execution environment. By passing a reference to a stub as parameter in an RPC, it is possible to refer to objects across process boundaries.

10.3.5. Object-Based Messaging

Although RMI is the preferred way of handling communication in object-based distributed systems, messaging has also found its way as an important alternative. There are various object-based messaging systems available, and, as can be expected, offer very much the same

functionality. In this section we will take a closer look at CORBA messaging, partly because it also provides an interesting way of combining method invocation and message-oriented communication.

CORBA is a well-known specification for distributed systems. Over the years, several implementations have come to existence, although it remains to be seen to what extent CORBA itself will ever become truly popular. However, independent of popularity, the CORBA specifications are comprehensive (which to many also means they are very complex). Recognizing the popularity of messaging systems, CORBA was quick to include a specification of a messaging service.

What makes messaging in CORBA different from other systems is its inherent object-based approach to communication. In particular, the designers of the messaging service needed to retain the model that all communication takes place by invoking an object. In the case of messaging, this design constraint resulted in two forms of asynchronous method invocations (in addition to other forms that were provided by CORBA as well).

An asynchronous method invocation is analogous to an asynchronous RPC: the caller continues after initiating the invocation without waiting for a result. In CORBA's callback model, a client provides an object that implements an interface containing callback methods. These methods can be called by the underlying communication system to pass the result of an asynchronous invocation. An important design issue is that asynchronous method invocations do not affect the original implementation of an object. In other words, it is the client's responsibility to transform the original synchronous invocation into an asynchronous one; the server is presented with a normal (synchronous) invocation request.

Constructing an asynchronous invocation is done in two steps. First, the original interface as implemented by the object is replaced by two new interfaces that are to be implemented by client-side software only. One interface contains the specification of methods that the client can call. None of these methods returns a value or has any output parameter. The second interface is the callback interface. For each operation in the original interface, it contains a method that will be called by the client's runtime system to pass the results of the associated method as called by the client.

As an example, consider an object implementing a simple interface with just one method:

```
int add(in int i, in int j, out int k);
```

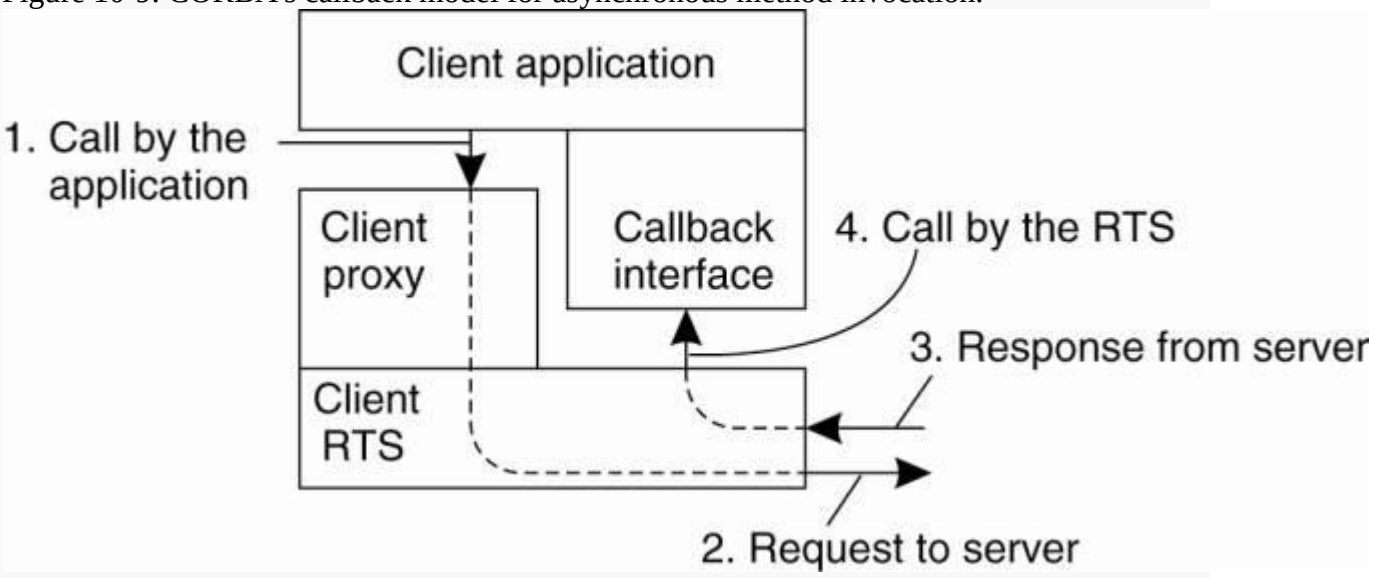
Assume that this method takes two nonnegative integers i and j and returns i + j as output parameter k. The operation is assumed to return-1 if the operation did not complete successfully. Transforming the original (synchronous) method invocation into an asynchronous one with callbacks is achieved by first generating the following pair of method specifications (for our purposes, we choose convenient names instead of following the strict rules as specified in OMG (2004a):
[Page 465]

```
void sendcb_add(in int i, in int j);  // Downcall by the client
void replycb_add(in int ret_val, in int k); // Upcall to the client
```

In effect, all output parameters from the original method specification are removed from the method that is to be called by the client, and returned as input parameters of the callback operations. Likewise, if the original method specified a return value, that value is passed as an input parameter to the callback operation.

The second step consists of compiling the generated interfaces. As a result, the client is offered a stub that allows it to asynchronously invoke sendcb_add. However, the client will need to provide an implementation for the callback interface, in our example containing the method replycb_add. This last method is called by the client's local runtime system (RTS), resulting in an upcall to the client application. Note that these changes do not affect the server-side implementation of the object. Using this example, the callback model is summarized in Fig. 10-9.

Figure 10-9. CORBA's callback model for asynchronous method invocation.



As an alternative to callbacks, CORBA provides a polling model. In this model, the client is offered a collection of operations to poll its local RTS for incoming results. As in the callback model, the client is responsible for transforming the original synchronous method invocations into asynchronous ones. Again, most of the work can be done by automatically deriving the appropriate method specifications from the original interface as implemented by the object.

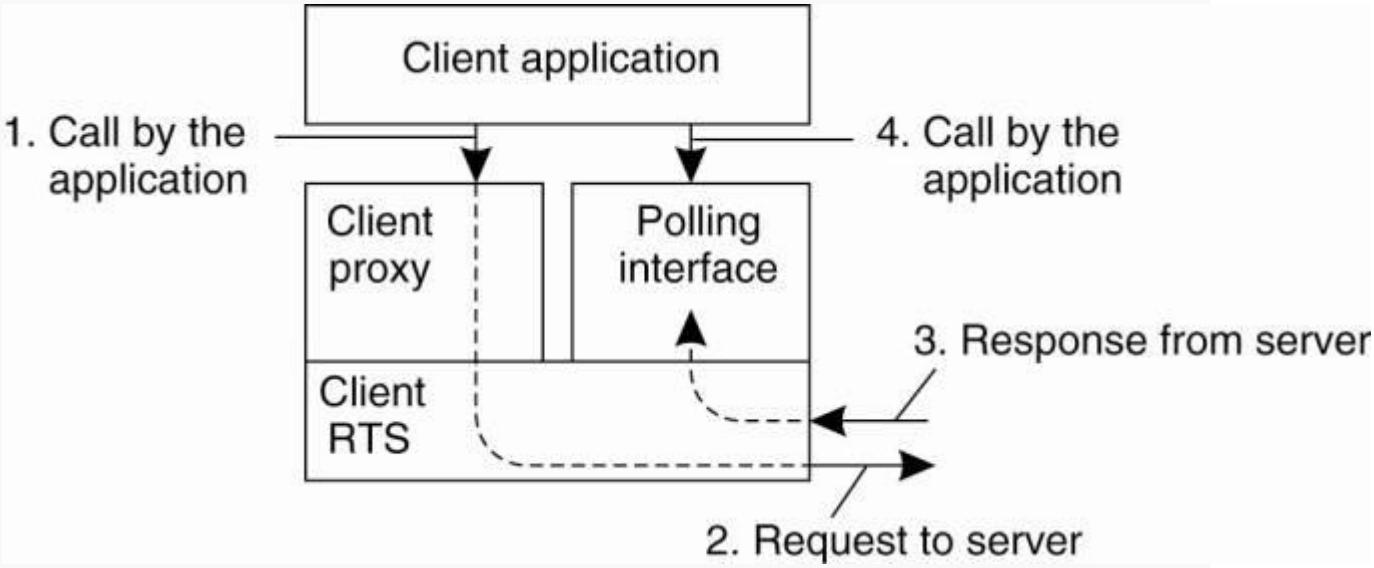
Returning to our example, the method add will lead to the following two generated method specifications (again, we conveniently adopt our own naming conventions):
void sendpoll_add(in int i, in int j); // Called by the client
void replypoll_add(out int ret_val, out int k); // Also called by the client

[Page 466]

The most important difference between the polling and callback models is that the method replypoll_add will have to be implemented by the client's RTS. This implementation can be automatically generated from interface specifications, just as the client-side stub is

automatically generated as we explained for RPCs. The polling model is summarized in Fig. 10-10. Again, notice that the implementation of the object as it appears at the server's side does not have to be changed.

Figure 10-10. CORBA's polling model for asynchronous method invocation.



What is missing from the models described so far is that the messages sent between a client and a server, including the response to an asynchronous invocation, are stored by the underlying system in case the client or server is not yet running. Fortunately, most of the issues concerning such persistent communication do not affect the asynchronous invocation model discussed so far. What is needed is to set up a collection of message servers that will allow messages (be they invocation requests or responses), to be temporarily stored until their delivery can take place.

To this end, the CORBA specifications also include interface definitions for what are called routers, which are analogous to the message routers we discussed in Chap. 4, and which can be implemented, for example, using IBM's WebSphere queue managers.

Likewise, Java has its own Java Messaging Service (JMS) which is again very similar to what we have discussed before [see Sun Microsystems (2004a)]. We will return to messaging more extensively in Chap. 13 when we discuss the publish/subscribe paradigm.

10.4. Naming

The interesting aspect of naming in object-based distributed systems evolves around the way that object references are supported. We already described these object references in the case of Java, where they effectively correspond to portable proxy implementations. However, this a language-dependent way of being able to refer to remote objects. Again taking CORBA as an example, let us see how basic naming can also be provided in a language and platform-

independent way. We also discuss a completely different scheme, which is used in the Globe distributed system.
[Page 467]

10.4.1. CORBA Object References

Fundamental to CORBA is the way its objects are referenced. When a client holds an object reference, it can invoke the methods implemented by the referenced object. It is important to distinguish the object reference that a client process uses to invoke a method, and the one implemented by the underlying RTS.

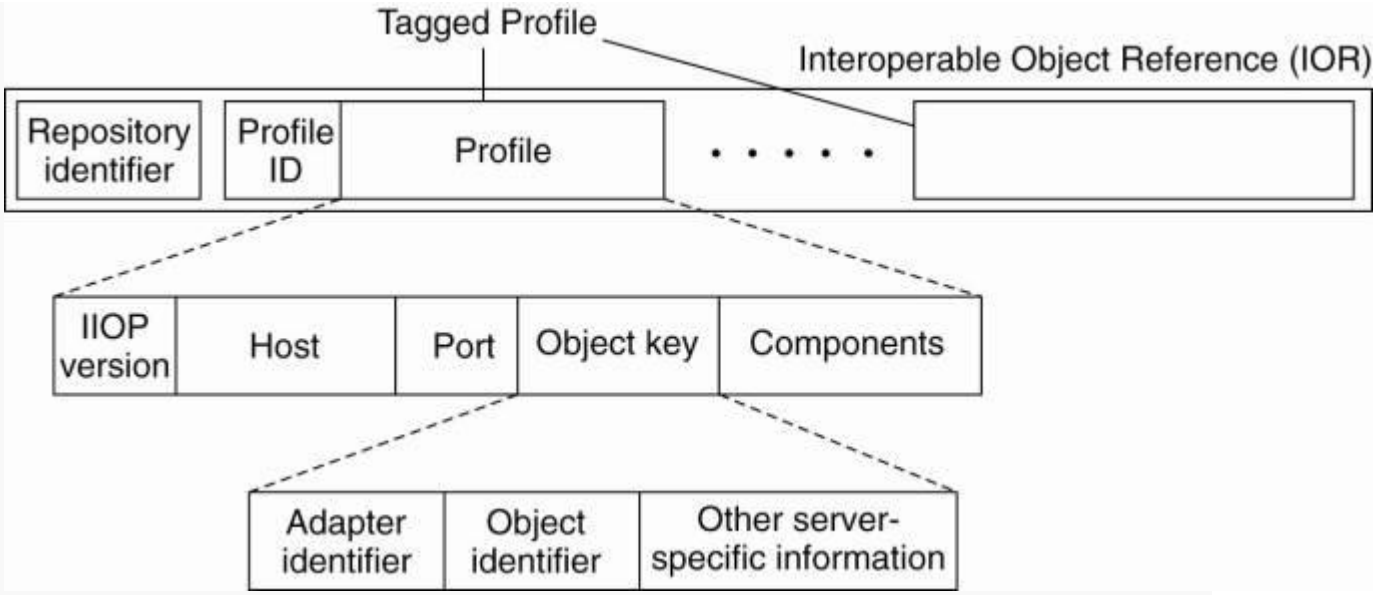
A process (be it client or server) can use only a language-specific implementation of an object reference. In most cases, this takes the form of a pointer to a local representation of the object. That reference cannot be passed from process A to process B, as it has meaning only within the address space of process A. Instead, process A will first have to marshal the pointer into a process-independent representation. The operation to do so is provided by its RTS. Once marshaled, the reference can be passed to process B, which can unmarshal it again. Note that processes A and B may be executing programs written in different languages.

In contrast, the underlying RTS will have its own language-independent representation of an object reference. This representation may even differ from the marshaled version it hands over to processes that want to exchange a reference. The important thing is that when a process refers to an object, its underlying RTS is implicitly passed enough information to know which object is actually being referenced. Such information is normally passed by the client and server-side stubs that are generated from the interface specifications of an object.

One of the problems that early versions of CORBA had was that each implementation could decide on how it represented an object reference. Consequently, if process A wanted to pass a reference to process B as described above, this would generally succeed only if both processes were using the same CORBA implementation. Otherwise, the marshaled version of the reference held by process A would be meaningless to the RTS used by process B.

Current CORBA systems all support the same language-independent representation of an object reference, which is called an Interoperable Object Reference or IOR. Whether or not a CORBA implementation uses IORs internally is not all that important. However, when passing an object reference between two different CORBA systems, it is passed as an IOR. An IOR contains all the information needed to identify an object. The general layout of an IOR is shown in Fig. 10-11, along with specific information for the communication protocol used in CORBA.

Figure 10-11. The organization of an IOR with specific information for IIOP.
(This item is displayed on page 468 in the print version)



Each IOR starts with a repository identifier. This identifier is assigned to an interface so that it can be stored and looked up in an interface repository. It is used to retrieve information on an interface at runtime, and can assist in, for example, type checking or dynamically constructing an invocation. Note that if this identifier is to be useful, both the client and server must have access to the same interface repository, or at least use the same identifier to identify interfaces. [Page 468]

The most important part of each IOR is formed by what are called tagged profiles. Each such profile contains the complete information to invoke an object. If the object server supports several protocols, information on each protocol can be included in a separate tagged profile. CORBA used the Internet Inter-ORB Protocol (IIOP) for communication between nodes. (An ORB or Object Request Broker is the name used by CORBA for their object-based runtime system.) IIOP is essentially a dedicated protocol for supported remote method invocations. Details on the profile used for IIOP are also shown in Fig. 10-11.

The IIOP profile is identified by a ProfileID field in the tagged profile. Its body consists of five fields. The IIOP version field identifies the version of IIOP that is used in this profile.

The Host field is a string identifying exactly on which host the object is located. The host can be specified either by means of a complete DNS domain name (such as soling.cs.vu.nl), or by using the string representation of that host's IP address, such as 130.37.24.11.

The Port field contains the port number to which the object's server is listening for incoming requests.

The Object key field contains server-specific information for demultiplexing incoming requests to the appropriate object. For example, an object identifier generated by a CORBA object adapter will generally be part of such an object key. Also, this key will identify the specific adapter.

Finally, there is a Components field that optionally contains more information needed for properly invoking the referenced object. For example, this field may contain security information indicating how the reference should be handled, or what to do in the case the referenced server is (temporarily) unavailable.

10.4.2. Globe Object References

Let us now take a look at a different way of referencing objects. In Globe, each distributed shared object is assigned a globally unique object identifier (OID), which is a 256-bit string. A Globe OID is a true identifier as defined in Chap. 5. In other words, a Globe OID refers to at most one distributed shared object; it is never reused for another object; and each object has at most one OID.

Globe OIDs can be used only for comparing object references. For example, suppose processes A and B are each bound to a distributed shared object. Each process can request the OID of the object they are bound to. If and only if the two OIDs are the same, then A and B are considered to be bound to the same object.

Unlike CORBA references, Globe OIDs cannot be used to directly contact an object. Instead, to locate an object, it is necessary to look up a contact address for that object in a location service. This service returns a contact address, which is comparable to the location-dependent object references as used in CORBA and other distributed systems. Although Globe uses its own specific location service, in principle any of the location services discussed in Chap. 5 would do.

Ignoring some minor details, a contact address has two parts. The first one is an address identifier by which the location service can identify the proper leaf node to which insert or delete operations for the associated contact address are to forwarded. Recall that because contact addresses are location dependent, it is important to insert and delete them starting at an appropriate leaf node.

The second part consists of actual address information, but this information is completely opaque to the location service. To the location service, an address is just an array of bytes that can equally stand for an actual network address, a marshaled interface pointer, or even a complete marshaled proxy.

Two kinds of addresses are currently supported in Globe. A stacked address represents a layered protocol suite, where each layer is represented by the three-field record shown in Fig. 10-12.

Figure 10-12. The representation of a protocol layer in a stacked contact address.

Field	Description
Protocol identifier	A constant representing a (known) protocol
Protocol address	A protocol-specific address
Implementation handle	Reference to a file in a class repository

The Protocol identifier is a constant representing a known protocol. Typical protocol identifiers include TCP, UDP, and IP. The Protocol address field contains a protocol-specific address, such as TCP port number, or an IPv4 network address. Finally, an Implementation handle can be optionally provided to indicate where a default implementation for the protocol can be found. Typically, an implementation handle is represented as a URL.
[Page 470]

The second type of contact address is an instance address, which consists of the two fields shown in Fig. 10-13. Again, the address contains an implementation handle, which is nothing but a reference to a file in a class repository where an implementation of a local object can be found. That local object should be loaded by the process that is currently binding to the object.

Figure 10-13. The representation of an instance contact address.

Field	Description
Implementation handle	Reference to a file in a class repository
Initialization string	String that is used to initialize an implementation

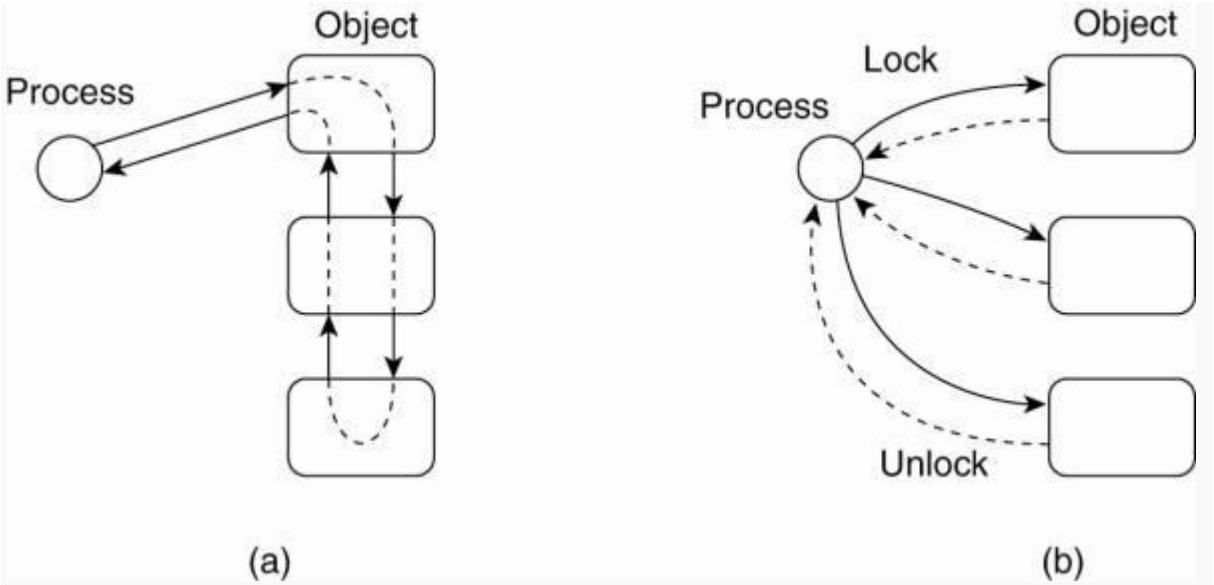
Loading follows a standard protocol, similar to class loading in Java. After the implementation has been loaded and the local object created, initialization takes place by passing the initialization string to the object. At that point, the object identifier has been completed resolved.

Note the difference in object referencing between CORBA and Globe, a difference which occurs frequently in distributed object-based systems. Where CORBA references contain exact information where to contact an object, Globe references require an additional lookup step to retrieve that information. This distinction also appears in systems such as Ice, where the CORBA equivalent is referred to as a direct reference, and the Globe equivalent as an indirect reference (Henning and Spruiell, 2005).

10.5. Synchronization

There are only a few issues regarding synchronization in distributed systems that are specific to dealing with distributed objects. In particular, the fact that implementation details are hidden behind interfaces may cause problems: when a process invokes a (remote) object, it has no knowledge whether that invocation will lead to invoking other objects. As a consequence, if an object is protected against concurrent accesses, we may have a cascading set of locks that the invoking process is unaware of, as sketched in Fig. 10-14(a).

Figure 10-14. Differences in control flow for locking objects
(This item is displayed on page 471 in the print version)



In contrast, when dealing with data resources such as files or database tables that are protected by locks, the pattern for the control flow is actually visible to the process using those resources, as shown in Fig. 10-14(b). As a consequence, the process can also exert more control at runtime when things go wrong, such as giving up locks when it believes a deadlock has occurred. Note that transaction processing systems generally follow the pattern shown in Fig. 10-14(b).
[Page 471]

In object-based distributed systems it is therefore important to know where and when synchronization takes place. An obvious location for synchronization is at the object server. If multiple invocation requests for the same object arrive, the server can decide to serialize those requests (and possibly keep a lock on an object when it needs to do a remote invocation itself).

However, letting the object server maintain locks complicates matters in the case that invoking clients crash. For this reason, locking can also be done at the client side, an approach that has been adopted in Java. Unfortunately, this scheme has its own drawbacks.

As we mentioned before, the difference between local and remote objects in Java is often difficult to make. Matters become more complicated when objects are protected by declaring its methods to be synchronized. If two processes simultaneously call a synchronized method, only one of the processes will proceed while the other will be blocked. In this way, we can ensure that access to an object's internal data is completely serialized. A process can also be blocked inside an object, waiting for some condition to become true.

Logically, blocking in a remote object is simple. Suppose that client A calls a synchronized method of a remote object. To make access to remote objects look always exactly the same as to local objects, it would be necessary to block A in the client-side stub that implements the object's interface and to which A has direct access. Likewise, another client on a different machine would need to be blocked locally as well before its request can be sent to the server. The consequence is that we need to synchronize different clients at different machines. As we discussed in Chap. 6, distributed synchronization can be fairly complex.

An alternative approach would be to allow blocking only at the server. In principle, this works fine, but problems arise when a client crashes while its invocation is being handled by the server. As we discussed in Chap. 8, we may require relatively sophisticated protocols to handle this situation, and which that may significantly affect the overall performance of remote method invocations.
[Page 472]

Therefore, the designers of Java RMI have chosen to restrict blocking on remote objects only to the proxies (Wollrath et al., 1996). This means that threads in the same process will be prevented from concurrently accessing the same remote object, but threads in different processes will not. Obviously, these synchronization semantics are tricky: at the syntactic level (i.e., when reading source code) we may see a nice, clean design. Only when the distributed application is actually executed, unanticipated behavior may be observed that should have been dealt with at design time. Here we see a clear example where striving for distribution transparency is not the way to go.

10.6. Consistency and Replication

Many object-based distributed systems follow a traditional approach toward replicated objects, effectively treating them as containers of data with their own special operations. As a result, when we consider how replication is handled in systems supporting Java beans, or CORBA-compliant distributed systems, there is not really that much new to report other than what we have discussed in Chap. 7.

For this reason, we focus on a few particular topics regarding consistency and replication that are more profound in object-based distributed systems than others. We will first consider consistency and move to replicated invocations.

10.6.1. Entry Consistency

As we mentioned in Chap. 7, data-centric consistency for distributed objects comes naturally in the form of entry consistency. Recall that in this case, the goal is to group operations on shared data using synchronization variables (e.g., in the form of locks). As objects naturally combine data and the operations on that data, locking objects during an invocation serializes access and keeps them consistent.

Although conceptually associating a lock with an object is simple, it does not necessarily provide a proper solution when an object is replicated. There are two issues that need to be solved for implementing entry consistency. The first one is that we need a means to prevent concurrent execution of multiple invocations on the same object. In other words, when any method of an object is being executed, no other methods may be executed. This requirement ensures that access to the internal data of an object is indeed serialized. Simply using local locking mechanisms will ensure this serialization.

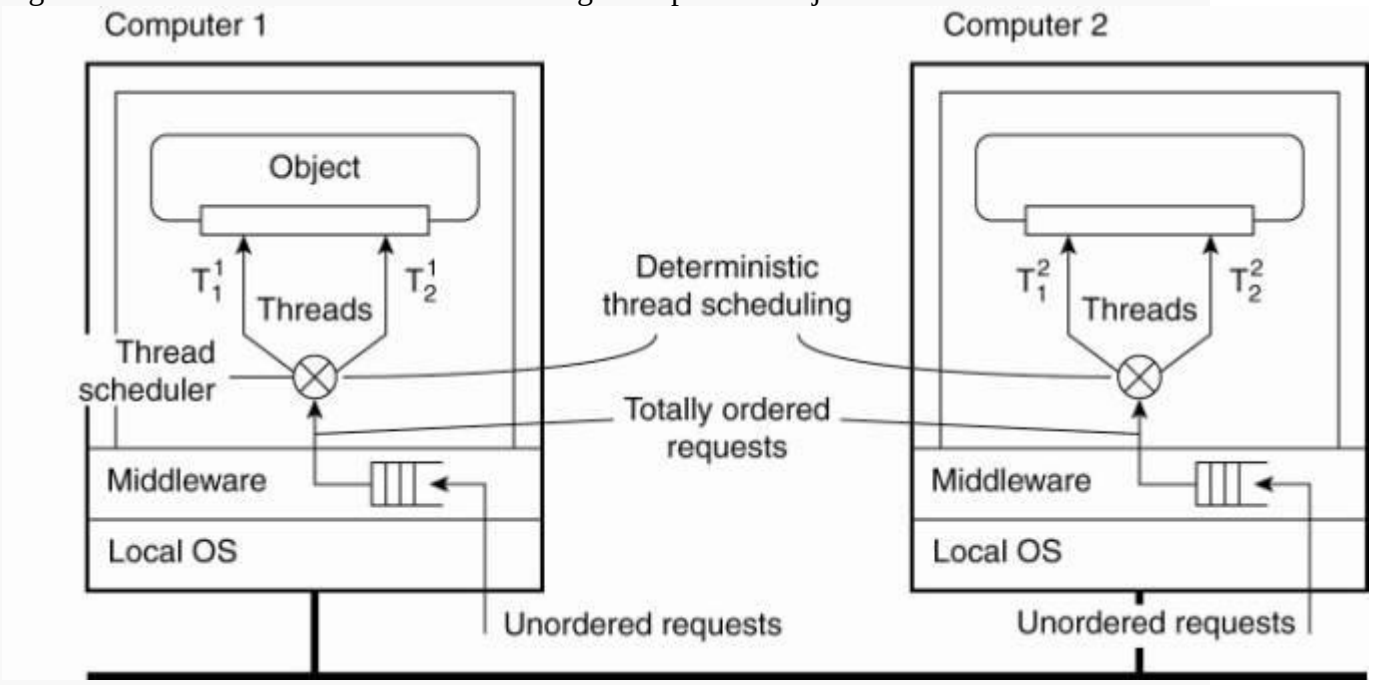
The second issue is that in the case of a replicated object, we need to ensure that all changes to the replicated state of the object are the same. In other words, we need to make sure that no two independent method invocations take place on different replicas at the same time. This

requirement implies that we need to order invocations such that each replica sees all invocations in the same order. This requirement can generally be met in one of two ways: (1) using a primary-based approach or (2) using totally-ordered multicast to the replicas.
[Page 473]

In many cases, designing replicated objects is done by first designing a single object, possibly protecting it against concurrent access through local locking, and subsequently replicating it. If we were to use a primary-based scheme, then additional effort from the application developer is needed to serialize object invocations. Therefore, it is often convenient to assume that the underlying middleware supports totally-ordered multicasting, as this would not require any changes at the clients, nor would it require additional programming effort from application developers. Of course, how the totally ordered multicasting is realized by the middleware should be transparent. For all the application may know, its implementation may use a primary-based scheme, but it could equally well be based on Lamport clocks.

However, even if the underlying middleware provides totally-ordered multicasting, more may be needed to guarantee orderly object invocation. The problem is one of granularity: although all replicas of an object server may receive invocation requests in the same order, we need to ensure that all threads in those servers process those requests in the correct order as well. The problem is sketched in Fig. 10-15.

Figure 10-15. Deterministic thread scheduling for replicated object servers.



Multithreaded (object) servers simply pick up an incoming request, pass it on to an available thread, and wait for the next request to come in. The server's thread scheduler subsequently allocates the CPU to runnable threads. Of course, if the middleware has done its best to provide a total ordering for request delivery, the thread schedulers should operate in a deterministic fashion in order not to mix the ordering of method invocations on the same object. In other

words, If threads and from Fig. 10-15 handle the same incoming (replicated) invocation request, they should both be scheduled before and , respectively.
[Page 474]

Of course, simply scheduling all threads deterministically is not necessary. In principle, if we already have totally-ordered request delivery, we need only to ensure that all requests for the same replicated object are handled in the order they were delivered. Such an approach would allow invocations for different objects to be processed concurrently, and without further restrictions from the thread scheduler. Unfortunately, only few systems exist that support such concurrency.

One approach, described in Basile et al. (2002), ensures that threads sharing the same (local) lock are scheduled in the same order on every replica. At the basics lies a primary-based scheme in which one of the replica servers takes the lead in determining, for a specific lock, which thread goes first. An improvement that avoids frequent communication between servers is described in Basile et al. (2003). Note that threads that do not share a lock can thus operate concurrently on each server.

One drawback of this scheme is that it operates at the level of the underlying operating system, meaning that every lock needs to be managed. By providing application-level information, a huge improvement in performance can be made by identifying only those locks that are needed for serializing access to replicated objects (Taiani et al., 2005). We return to these issues when we discuss fault tolerance for Java.

Replication Frameworks

An interesting aspect of most distributed object-based systems is that by nature of the object technology it is often possible to make a clean separation between devising functionality and handling extra-functional issues such as replication. As we explained in Chap. 2, a powerful mechanism to accomplish this separation is formed by interceptors.

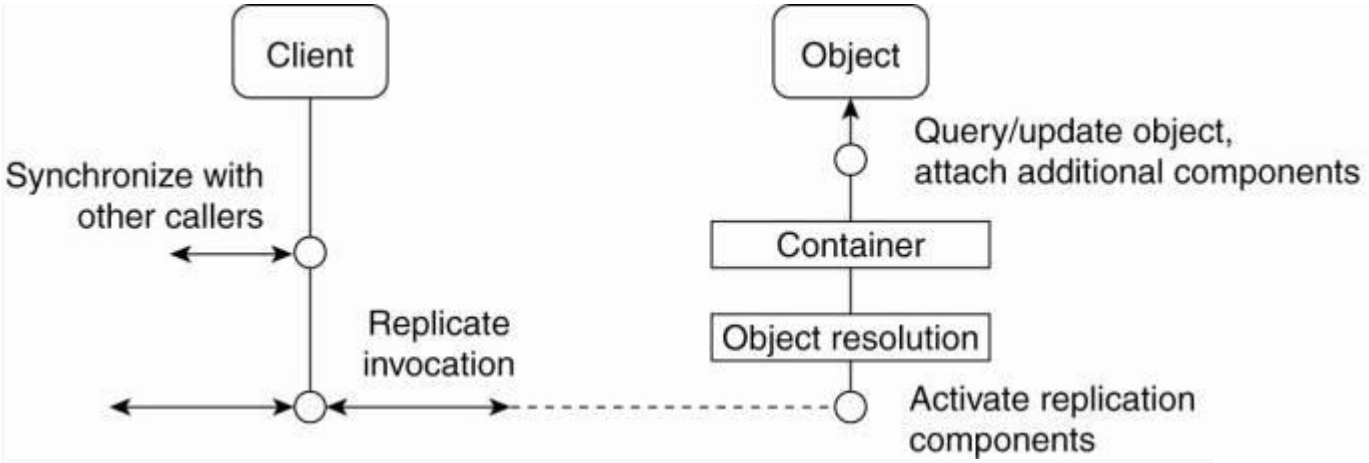
Babaoglu et al. (2004) describe a framework in which they use interceptors to replicate Java beans for J2EE servers. The idea is relatively simple: invocations to objects are intercepted at three different points, as also shown in Fig. 10-16:

At the client side just before the invocation is passed to the stub.

Inside the client's stub, where the interception forms part of the replication algorithm.

At the server side, just before the object is about to be invoked.

Figure 10-16. A general framework for separating replication algorithms from objects in an EJB environment.
(This item is displayed on page 475 in the print version)



The first interception is needed when it turns out that the caller is replicated. In that case, synchronization with the other callers may be needed as we may be dealing with a replicated invocation as discussed before.
[Page 475]

Once it has been decided that the invocation can be carried out, the interceptor in the client-side stub can take decisions on where to be forward the request to, or possibly implement a fail-over mechanism when a replica cannot be reached.

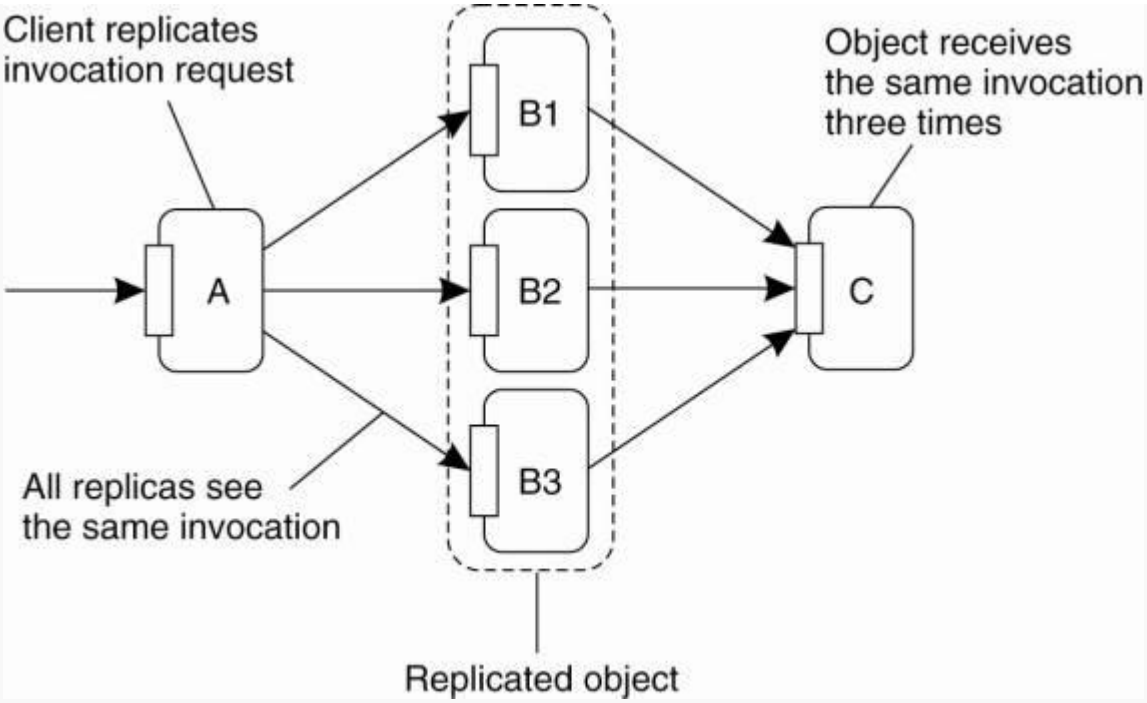
Finally, the server-side interceptor handles the invocation. In fact, this interceptor is split into two. At the first point, just after the request has come in and before it is handed over to an adapter, the replication algorithm gets control. It can then analyze for whom the request is intended allowing it to activate, if necessary, any replication objects that it needs to carry out the replication. The second point is just before the invocation, allowing the replication algorithm to, for example, get and set attribute values of the replicated object.

The interesting aspect is that the framework can be set up independent of any replication algorithm, thus leading to a complete separation of object functionality and replication of objects.

10.6.2. Replicated Invocations

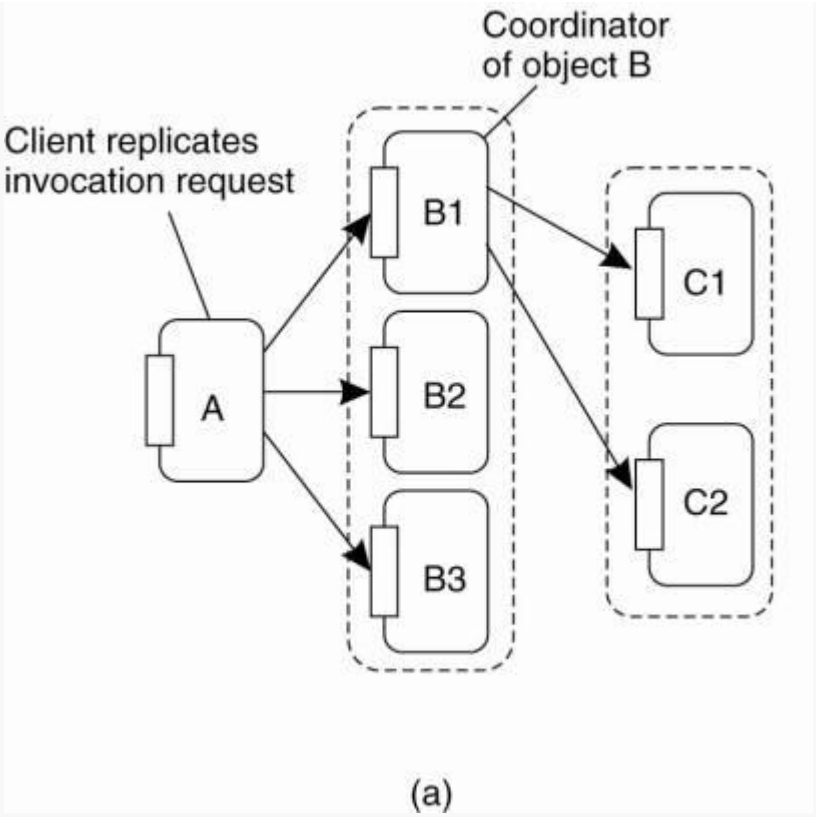
Another problem that needs to be solved is that of replicated invocations. Consider an object A calling another object B as shown in Fig. 10-17. Object B is assumed to call yet another object C. If B is replicated, each replica of B will, in principle, call C independently. The problem is that C is now called multiple times instead of only once. If the called method on C results in the transfer of $100,000, then clearly, someone is going to complain sooner or later.

Figure 10-17. The problem of replicated method invocations.
(This item is displayed on page 476 in the print version)

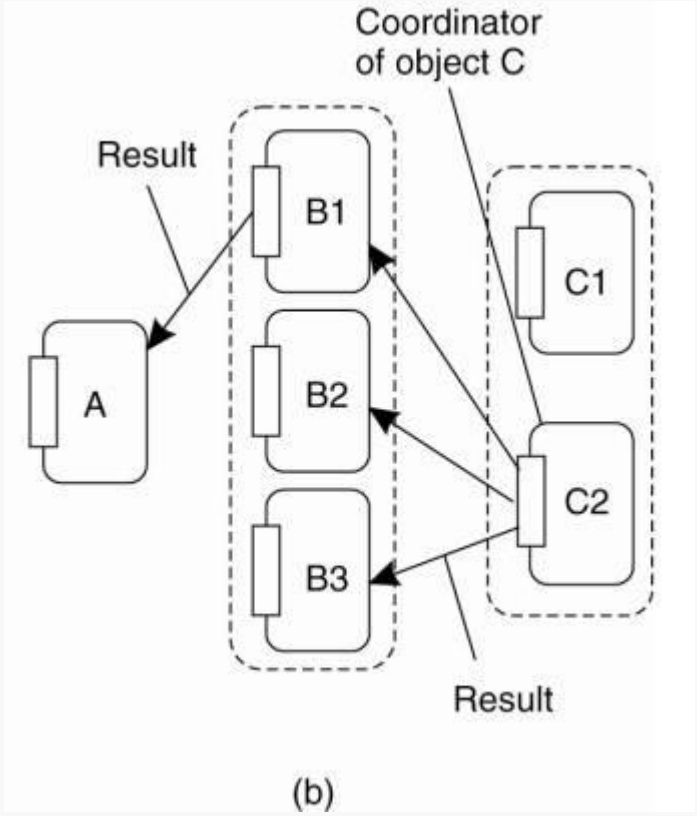


There are not many general-purpose solutions to solve the problem of replicated invocations. One solution is to simply forbid it (Maassen et al., 2001), which makes sense when performance is at stake. However, when replicating for fault tolerance, the following solution proposed by Mazouni et al. (1995) may be deployed. Their solution is independent of the replication policy, that is, the exact details of how replicas are kept consistent. The essence is to provide a replication-aware communication layer on top of which (replicated) objects execute. When a replicated object B invokes another replicated object C, the invocation request is first assigned the same, unique identifier by each replica of B. At that point, a coordinator of the replicas of B forwards its request to all the replicas of object C, while the other replicas of B hold back their copy of the invocation request, as shown in Fig. 10-18(a). The result is that only a single request is forwarded to each replica of C.
[Page 476]

Figure 10-18. (a) Forwarding an invocation request from a replicated object to another replicated object. (b) Returning a reply from one replicated object to another.



(a)



(b)

The same mechanism is used to ensure that only a single reply message is returned to the replicas of B. This situation is shown in Fig. 10-18(b). A coordinator of the replicas of C notices

it is dealing with a replicated reply message that has been generated by each replica of C. However, only the coordinator forwards that reply to the replicas of object B, while the other replicas of C hold back their copy of the reply message.
[Page 477]

When a replica of B receives a reply message for an invocation request it had either forwarded to C or held back because it was not the coordinator, the reply is then handed to the actual object.

In essence, the scheme just described is based on using multicast communication, but in preventing that the same message is multicast by different replicas. As such, it is essentially a sender-based scheme. An alternative solution is to let a receiving replica detect multiple copies of incoming messages belonging to the same invocation, and to pass only one copy to its associated object. Details of this scheme are left as an exercise.

10.7. Fault Tolerance

Like replication, fault tolerance in most distributed object-based systems use the same mechanisms as in other distributed systems, following the principles we discussed in Chap. 8. However, when it comes to standardization, CORBA arguably provides the most comprehensive specification.

10.7.1. Example: Fault-Tolerant CORBA

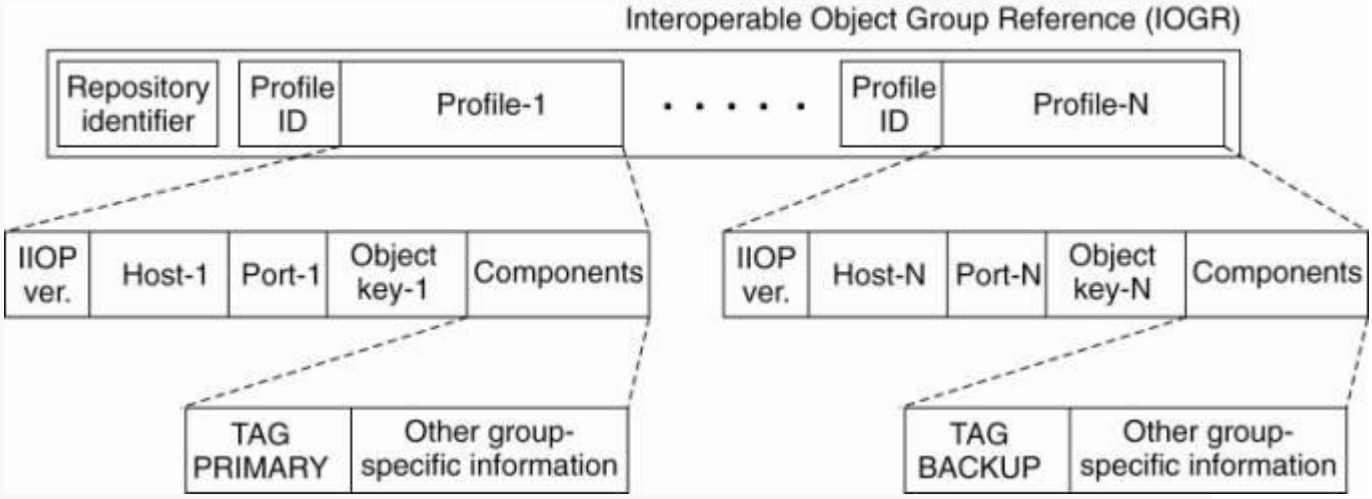
The basic approach for dealing with failures in CORBA is to replicate objects into object groups. Such a group consists of one or more identical copies of the same object. However, an object group can be referenced as if it were a single object. A group offers the same interface as the replicas it contains. In other words, replication is transparent to clients. Different replication strategies are supported, including primary-backup replication, active replication, and quorum-based replication. These strategies have all been discussed in Chap. 7. There are various other properties associated with object groups, the details of which can be found in OMG (2004a).

To provide replication and failure transparency as much as possible, object groups should not be distinguishable from normal CORBA objects, unless an application prefers otherwise. An important issue, in this respect, is how object groups are referenced. The approach followed is to use a special kind of IOR, called an Interoperable Object Group Reference (IOGR). The key difference with a normal IOR is that an IOGR contains multiple references to different objects, notably replicas in the same object group. In contrast, an IOR may also contain multiple references, but all of them will refer to the same object, although possibly using a different access protocol.
[Page 478]

Whenever a client passes an IOGR to its runtime system (RTS), that RTS attempts to bind to one of the referenced replicas. In the case of IIOP, the RTS may possibly use additional information it finds in one of the IIOP profiles of the IOGR. Such information can be stored in the Components field we discussed previously. For example, a specific IIOP profile may refer

to the primary or a backup of an object group, as shown in Fig. 10-19, by means of the separate tags TAG_PRIMARY and TAG_BACKUP, respectively.

Figure 10-19. A possible organization of an IOGR for an object group having a primary and backups.

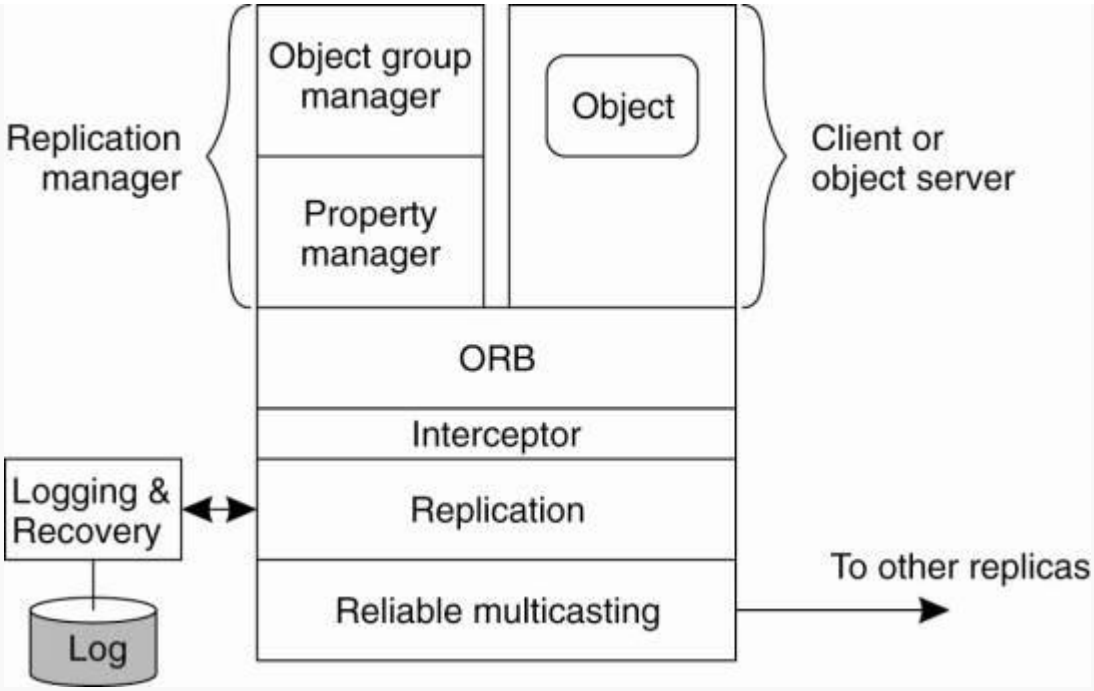


If binding to one of the replicas fails, the client RTS may continue by attempting to bind to another replica, thereby following any policy for next selecting a replica that it suits to best. To the client, the binding procedure is completely transparent; it appears as if the client is binding to a regular CORBA object.

An Example Architecture

To support object groups and to handle additional failure management, it is necessary to add components to CORBA. One possible architecture of a fault-tolerant version of CORBA is shown in Fig. 10-20. This architecture is derived from the Eternal system (Moser et al., 1998; and Narasimhan et al., 2000), which provides a fault tolerance infrastructure constructed on top of the Totem reliable group communication system (Moser et al., 1996).

Figure 10-20. An example architecture of a fault-tolerant CORBA system.
(This item is displayed on page 479 in the print version)



There are several components that play an important role in this architecture. By far the most important one is the replication manager, which is responsible for creating and managing a group of replicated objects. In principle, there is only one replication manager, although it may be replicated for fault tolerance.

As we have stated, to a client there is no fundamental difference between an object group and any other type of CORBA object. To create an object group, a client simply invokes the normal create_object operation as offered, in this case, by the replication manager, specifying the type of object to create. The client remains unaware of the fact that it is implicitly creating an object group. The number of replicas that are created when starting a new object group is normally determined by the system-dependent default value. The replica manager is also responsible for replacing a replica in the case of a failure, thereby ensuring that the number of replicas does not drop below a specified minimum.
[Page 479]

The architecture also shows the use of message-level interceptors. In the case of the Eternal system, each invocation is intercepted and passed to a separate replication component that maintains the required consistency for an object group and which ensures that messages are logged to enable recovery.

Invocations are subsequently sent to the other group members using reliable, totally-ordered multicasting. In the case of active replication, an invocation request is passed to each replica object by handing it to that object's underlying runtime system. However, in the case of passive replication, an invocation request is passed only to the RTS of the primary, whereas the other servers only log the invocation request for recovery purposes. When the primary has completed the invocation, its state is then multicast to the backups.

This architecture is based on using interceptors. Alternative solutions exist as well, including those in which fault tolerance has been incorporated in the runtime system (potentially affecting interoperability), or in which special services are used on top of the RTS to provide fault tolerance. Besides these differences, practice shows that there are other problems not (yet) covered by the CORBA standard. As an example of one problem that occurs in practice, if replicas are created on different implementations, there is no guarantee that this approach will actually work. A review of the different approaches and an assessment of fault tolerance in CORBA is discussed in Felber and Narasimhan (2004).

10.7.2. Example: Fault-Tolerant Java

Considering the popularity of Java as a language and platform for developing distributed applications, some effort has also been into adding fault tolerance to the Java runtime system. An interesting approach is to ensure that the Java virtual machine can be used for active replication.

Active replication essentially dictates that the replica servers execute as deter-ministic finite-state machines (Schneider, 1990). An excellent candidate in Java to fulfill this role is the Java Virtual Machine (JVM). Unfortunately, the JVM is not deterministic at all. There are various causes for nondeterministic behavior, identified independently by Napper et al. (2003) and Friedman and Kama (2003):

JVM can execute native code, that is, code that is external to the JVM and provided to the latter through an interface. The JVM treats native code like a black box: it sees only the interface, but has no clue about the (potentially nondeterministic) behavior that a call causes. Therefore, in order to use the JVM for active replication, it is necessary to make sure that native code behaves in a deterministic way.

Input data may be subject to nondeterminism. For example, a shared variable that can be manipulated by multiple threads may change for different instances of the JVM as long as threads are allowed to operate concurrently. To control this behavior, shared data should at the very least be protected through locks. As it turned out, the Java runtime environment did not always adhere to this rule, despite its support for multithreading.

In the presence of failures, different JVMs will produce different output revealing that the machines have been replicated. This difference may cause problems when the JVMs need to brought back into the same state. Matters are simplified if one can assume that all output is idempotent (i.e., can simply be replayed), or is testable so that one can check whether output was produced before a crash or not. Note that this assumption is necessary in order to allow a replica server to decide whether or not it should re-execute an operation.

Practice shows that turning the JVM into a deterministic finite-state machine is by no means trivial. One problem that needs to be solved is the fact that replica servers may crash. One possible organization is to let the servers run according to a primary-backup scheme. In such a scheme, one server coordinates all actions that need to be performed, and from time to time instructs the backup to do the same. Careful coordination between primary and backup is required, of course.
[Page 481]

Note that despite the fact that replica servers are organized in a primary-backup setting, we are still dealing with active replication: the replicas are kept up to date by letting each of them execute the same operations in the same order. However, to ensure the same nondeterministic behavior by all of the servers, the behavior of one server is taken as the one to follow.

In this setting, the approach followed by Friedman and Kama (2003) is to let the primary first execute the instructions of what is called a frame. A frame consists of the execution of several context switches and ends either because all threads are blocking for I/O to complete, or after a predefined number of context switches has taken place. Whenever a thread issues an I/O operation, the thread is blocked by the JVM put on hold. When a frame starts, the primary lets all I/O requests proceed, one after the other, and the results are sent to the other replicas. In this way, at least deterministic behavior with respect to I/O operations is enforced.

The problem with this scheme is easily seen: the primary is always ahead of the other replicas. There are two situations we need to consider. First, if a replica server other than the primary crashes, no real harm is done except that the degree of fault tolerance drops. On the other hand, when the primary crashes, we may find ourselves in a situation that data (or rather, operations) are lost.

To minimize the damage, the primary works on a per-frame basis. That is, it sends update information to the other replicas only after completion of its current frame. The effect of this approach is that when the primary is working on the k-th frame, that the other replica servers have all the information needed to process the frame preceding the k-th one. The damage can be limited by making frames small, at the price of more communication between the primary and the backups.

10.8. Security

Obviously, security plays an important role in any distributed system and object-based ones are no exception. When considering most object-based distributed systems, the fact that distributed objects are remote objects immediately leads to a situation in which security architectures for distributed systems are very similar. In essence, each object is protected through standard authentication and authorization mechanisms, like the ones we discussed in Chap. 9.

To make clear how security can fit in specifically in an object-based distributed system, we shall discuss the security architecture for the Globe system. As we mentioned before, Globe supports truly distributed objects in which the state of a single object can be spread and replicated across multiple machines. Remote objects are just a special case of Globe objects. Therefore, by considering the Globe security architecture, we can also see how its approach can be equally applied to more traditional object-based distributed systems. After discussing Globe, we briefly take a look at security in traditional object-based systems.
[Page 482]

10.8.1. Example: Globe

As we said, Globe is one of the few distributed object-based systems in which an object's state can be physically distributed and replicated across multiple machines. This approach also

introduces specific security problems, which have led to an architecture as described in Popescu et al. (2002).

Overview

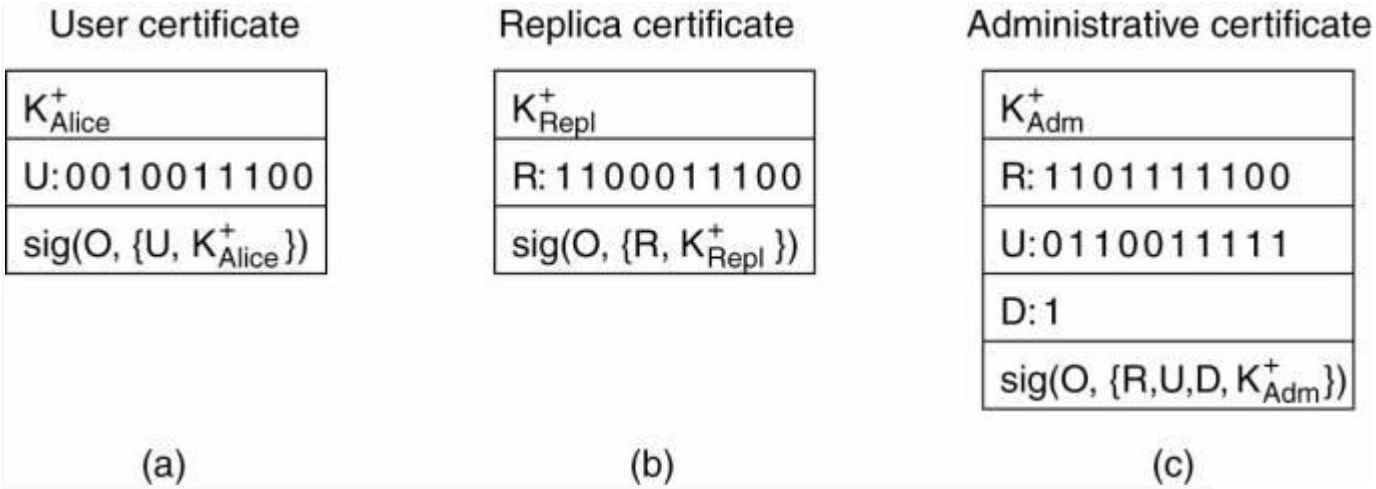
When we consider the general case of invoking a method on a remote object, there are at least two issues that are important from a security perspective: (1) is the caller invoking the correct object and (2) is the caller allowed to invoke that method. We refer to these two issues as secure object binding and secure method invocation, respectively. The former has everything to do with authentication, whereas the latter involves authorization. For Globe and other systems that support either replication or moving objects around, we have an additional problem, namely that of platform security. This kind of security comprises two issues. First, how can the platform to which a (local) object is copied be protected against any malicious code contained in the object, and secondly, how can the object be protected against a malicious replica server.

Being able to copy objects to other hosts also brings up another problem. Because the object server that is hosting a copy of an object need not always be fully trusted, there must be a mechanism that prevents that every replica server hosting an object from being allowed to also execute any of an object's methods. For example, an object's owner may want to restrict the execution of update methods to a small group of replica servers, whereas methods that only read the state of an object may be executed by any authenticated server. Enforcing such policies can be done through reverse access control, which we discuss in more detail below.

There are several mechanisms deployed in Globe to establish security. First, every Globe object has an associated public/private key pair, referred to as the object key. The basic idea is that anyone who has knowledge about an object's private key can set the access policies for users and servers. In addition, every replica has an associated replica key, which is also constructed as a public/private key pair. This key pair is generated by the object server currently hosting the specific replica. As we will see, the replica key is used to make sure that a specific replica is part of a given distributed shared object. Finally, each user is also assumed to have a unique public/private key pair, known as the user key.

These keys are used to set the various access rights in the form of certificates. Certificates are handed out per object. There are three types, as shown in Fig. 10-21. A user certificate is associated with a specific user and specifies exactly which methods that user is allowed to invoke. To this end, the certificate contains a bit string U with the same length as the number of methods available for the object. U [i] = 1 if and only if the user is allowed to invoke method Mi. Likewise, there is also a replica certificate that specifies, for a given replica server, which methods it is allowed to execute. It also has an associated bit string R, where R [i] = i if and only if the server is allowed to execute method Mi.
[Page 483]

Figure 10-21. Certificates in Globe: (a) a user certificate, (b) a replica certificate, (c) an administrative certificate.



For example, the user certificate in Fig. 10-21(a) tells that Alice (who can be identified through her public key), has the right to invoke methods M2, M5, M6, and M7 (note that we start indexing U at 0). Likewise, the replica certificate states that the server owning is allowed to execute methods M0, M1, M5, M6, and M7.

An administrative certificate can be used by any authorized entity to issue user and replica certificates. In the case, the R and U bit strings specify for which methods and which entities a certificate can be created. Moreover, there is bit indicating whether an administrative entity can delegate (part of) its rights to someone else. Note that when Bob in his role as administrator creates a user certificate for Alice, he will sign that certificate with his own signature, not that of the object. As a consequence, Alice's certificate will need to be traced back to Bob's administrative certificate, and eventually to an administrative certificate signed with the object's private key.

Administrative certificates come in handy when considering that some Globe objects may be massively replicated. For example, an object's owner may want to manage only a relatively small set of permanent replicas, but delegate the creation of server-initiated replicas to the servers hosting those permanent replicas. In that case, the owner may decide to allow a permanent replica to install other replicas for read-only access by all users. Whenever Alice wants to invoke a read-only method, she will succeed (provided she is authorized). However, when wanting to invoke an update method, she will have to contact one of the permanent replicas, as none of the other replica servers is allowed to execute such methods.

As we explained, the binding process in Globe requires that an object identifier (OID) is resolved to a contact address. In principle, any system that supports flat names can be used for this purpose. To securely associate an object's public key to its OID, we simply compute the OID as a 160-bit secure hash of the public key. In this way, anyone can verify whether a given public key belongs to a given OID. These identifier are also known as self-certifying names, a concept pioneered in the Secure File System (Mazieres et al., 1999), which we will discuss in Chap. 11.
[Page 484]

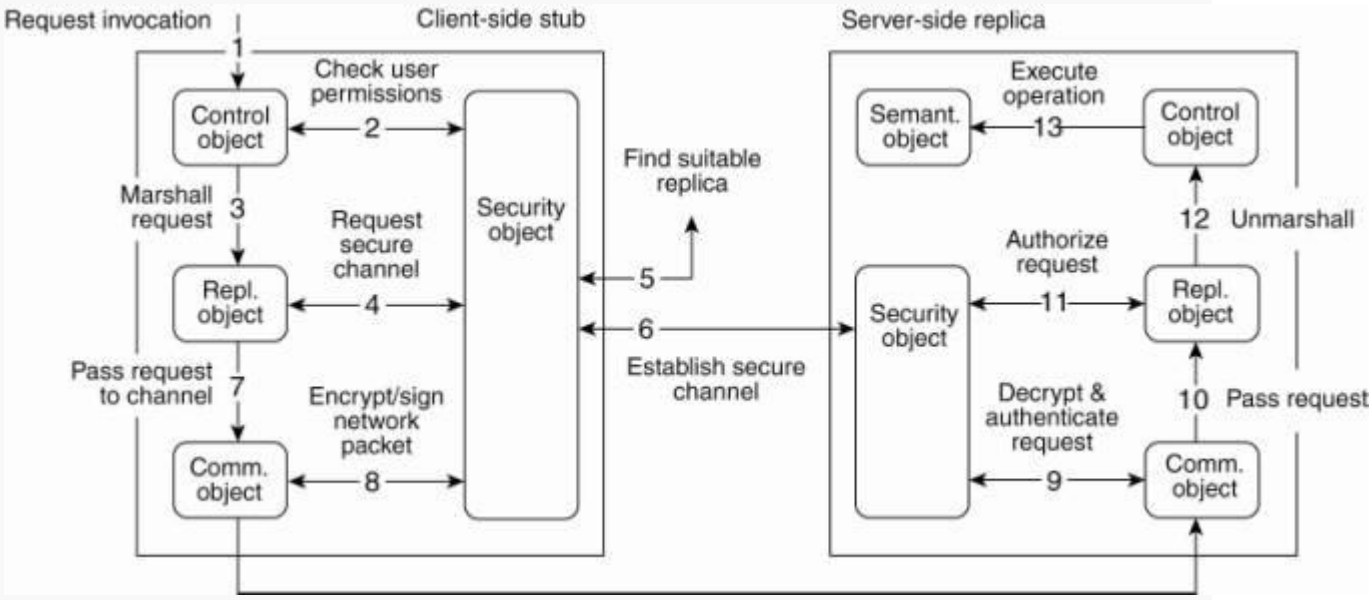
We can also check whether a replica R belongs to an object O. In that case, we merely need to inspect the replica certificate for R, and check who issued it. The signer may be an entity with administrative rights, in which case we need to inspect its administrative certificate. The bottom line is that we can construct a chain of certificates of which the last one is signed using the object's private key. In that case, we know that R is part of O.

To mutually protect objects and hosts against each other, techniques for mobile code, as described in Chap. 9 are deployed. Detecting that objects have been tampered with can be done with special auditing techniques which we will describe in Chap. 12.

Secure Method Invocation

Let us now look into the details of securely invoking a method of a Globe object. The complete path from requesting an invocation to actually executing the operation at a replica is sketched in Fig. 10-22. A total of 13 steps need to be executed in sequence, as shown in the figure and described in the following text.

Figure 10-22. Secure method invocation in Globe.



1. First, an application issues a invocation request by locally calling the associated method, just like calling a procedure in an RPC.

2. The control subobject checks the user permissions with the information stored in the local security object. In this case, the security object should have a valid user certificate.

3. The request is marshaled and passed on.

4. The replication subobject requests the middleware to set up a secure channel to a suitable replica.

5. The security object first initiates a replica lookup. To achieve this goal, it could use any naming service that can look up replicas that have been specified to be able to execute certain methods. The Globe location service has been modified to handle such lookups (Ballintijn, 2003).

6. Once a suitable replica has been found, the security subobject can set up a secure channel with its peer, after which control is returned to the replication subobject. Note that part of this establishment requires that the replica proves it is allowed to carry out the requested invocation.

7. The request is now passed on to the communication subobject.

8. The subobject encrypts and signs the request so that it can pass through the channel.

9. After its receipt, the request is decrypted and authenticated.

10. The request is then simply passed on to the server-side replication subobject.

11. Authorization takes place: in this case the user certificate from the client-side stub has been passed to the replica so that we can verify that the request can indeed be carried out.

12. The request is then unmarshaled.

13. Finally, the operation can be executed.

Although this may seem to be a relatively large number of steps, the example shows how a secure method invocation can be broken down into small units, each unit being necessary to ensure that an authenticated client can carry out an authorized invocation at an authenticated replica. Virtually all object-based distributed systems follow these steps. The difference with Globe is that a suitable replica needs to be located, and that this replica needs to prove it may execute the method call. We leave such a proof as an exercise to the reader.

10.8.2. Security for Remote Objects

When using remote objects we often see that the object reference itself is implemented as a complete client-side stub, containing all the information that is needed to access the remote object. In its simplest form, the reference contains the exact contact address for the object and uses a standard marshaling and communication protocol to ship an invocation to the remote object.

However, in systems such as Java, the client-side stub (called a proxy) can be virtually anything. The basic idea is that the developer of a remote object also develops the proxy and subsequently registers the proxy with a directory service. When a client is looking for the object, it will eventually contact the directory service, retrieve the proxy, and install it. There are obviously some serious problems with this approach.

First, if the directory service is hijacked, then an attacker may be able to return a bogus proxy to the client. In effect, such a proxy may be able to compromise all communication between the client and the server hosting the remote object, damaging both of them.

Second, the client has no way to authenticate the server: it only has the proxy and all communication with the server necessarily goes through that proxy. This may be an undesirable situation, especially because the client now simply needs to trust the proxy that it will do its work correctly.

Likewise, it may be more difficult for the server to authenticate the client. Authentication may be necessary when sensitive information is sent to the client. Also, because client authentication is now tied to the proxy, we may also have the situation that an attacker is spoofing a client causing damage to the remote object.

Li et al. (2004b) describe a general security architecture that can be used to make remote object invocations safer. In their model, they assume that proxies are indeed provided by the developer of a remote object and registered with a directory service. This approach is followed in Java RMI, but also Jini (Sun Microsystems, 2005).

The first problem to solve is to authenticate a remote object. In their solution, Li and Mitchell propose a two-step approach. First, the proxy which is downloaded from a directory service is signed by the remote object allowing the client to verify its origin. The proxy, in turn, will authenticate the object using TLS with server authentication, as we discussed in Chap. 9. Note that it is the object developer's task to make sure that the proxy indeed properly authenticates the object. The client will have to rely on this behavior, but because it is capable of authenticating the proxy, relying on object authentication is at the same level as trusting the remote object to behave decently.

To authenticate the client, a separate authenticator is used. When a client is looking up the remote object, it will be directed to this authenticator from which it downloads an authentication proxy. This is a special proxy that offers an interface by which the client can have itself authenticated by the remote object. If this authentication succeeds, then the remote object (or actually, its object server) will pass on the actual proxy to the client. Note that this approach allows for authentication independent of the protocol used by the actual proxy, which is considered an important advantage.
[Page 487]

Another important advantage of separating client authentication is that it is now possible to pass dedicated proxies to clients. For example, certain clients may be allowed to request only execution of read-only methods. In such a case, after authentication has taken place, the client will be handed a proxy that offers only such methods, and no other. More refined access control can easily be envisaged.