DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN
By: Dr. Faramarz Safi
Islamic Azad University,
Najafabad Branch

Chapter 10
DISTRIBUTED OBJECT-BASED SYSTEMS

Introduction

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.

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.

Distributed Objects

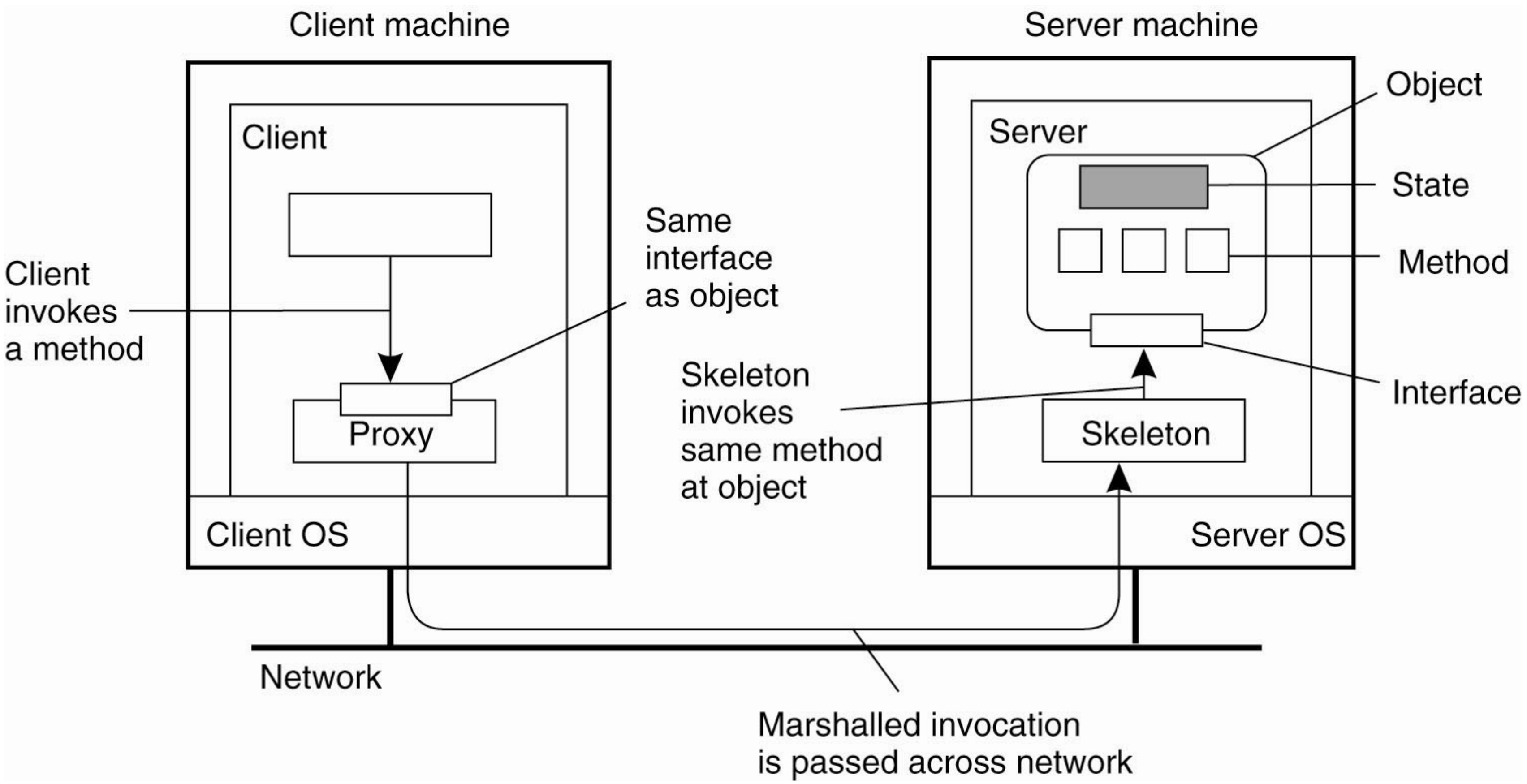


Figure 10-1. Common organization of a remote object with client-side proxy.

Distributed Objects

The Client: 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: The server stub is also responsible for marshaling replies and forwarding reply messages to the client side proxy. 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.

Distributed Objects

A characteristic 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.

Compile- Time versus Runtime Objects

An alternative way of constructing distributed objects is to do this explicitly during runtime. This approach 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. 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 adapter allows an interface to be converted into something that a client expects.

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.

Persistent and Transient Objects

A **persistent** object 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 is. As the 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.

Enterprise Java Beans (EJB)

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.

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.

Example: Enterprise Java Beans

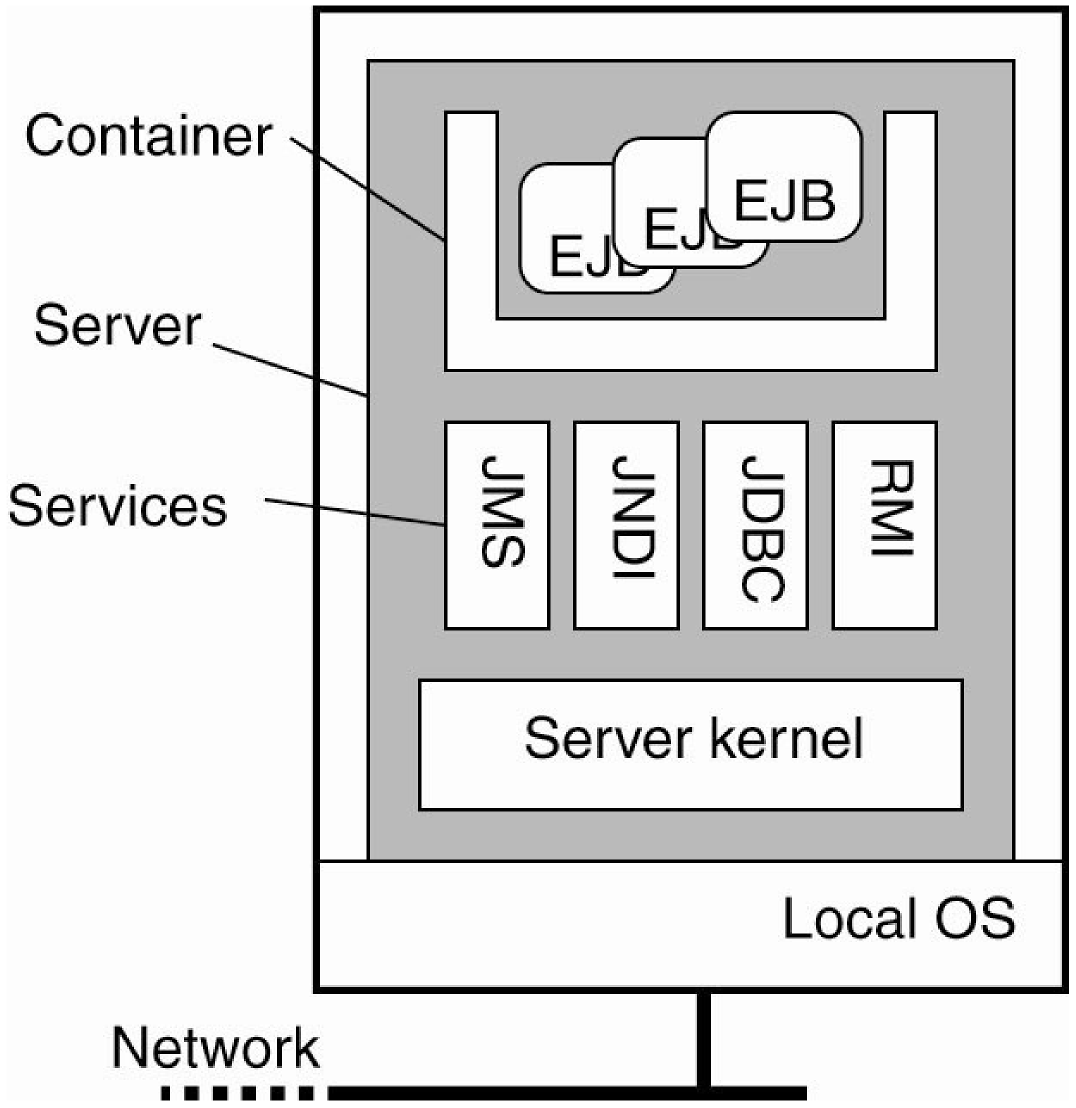


Figure 10-2. General architecture of an EJB server.

Four Types of EJBs

- **Stateless session** beans are transient objects that are invoked once, does its work, after which it discards any information it maintains to perform the service it offered to a client.
- **Stateful session** beans maintain client-related state.
- Entity beans can be considered to be a long-lived persistent object. Such an entity bean will generally be stored in a database, and likewise, will also be part of distributed transactions.
- **Message-driven** beans are used to program objects that should react to incoming messages (and likewise, be able to send messages). They cannot be invoked directly by a client, but rather fit into a publish-subscribe way of communication

Globe Distributed Shared Objects (1)

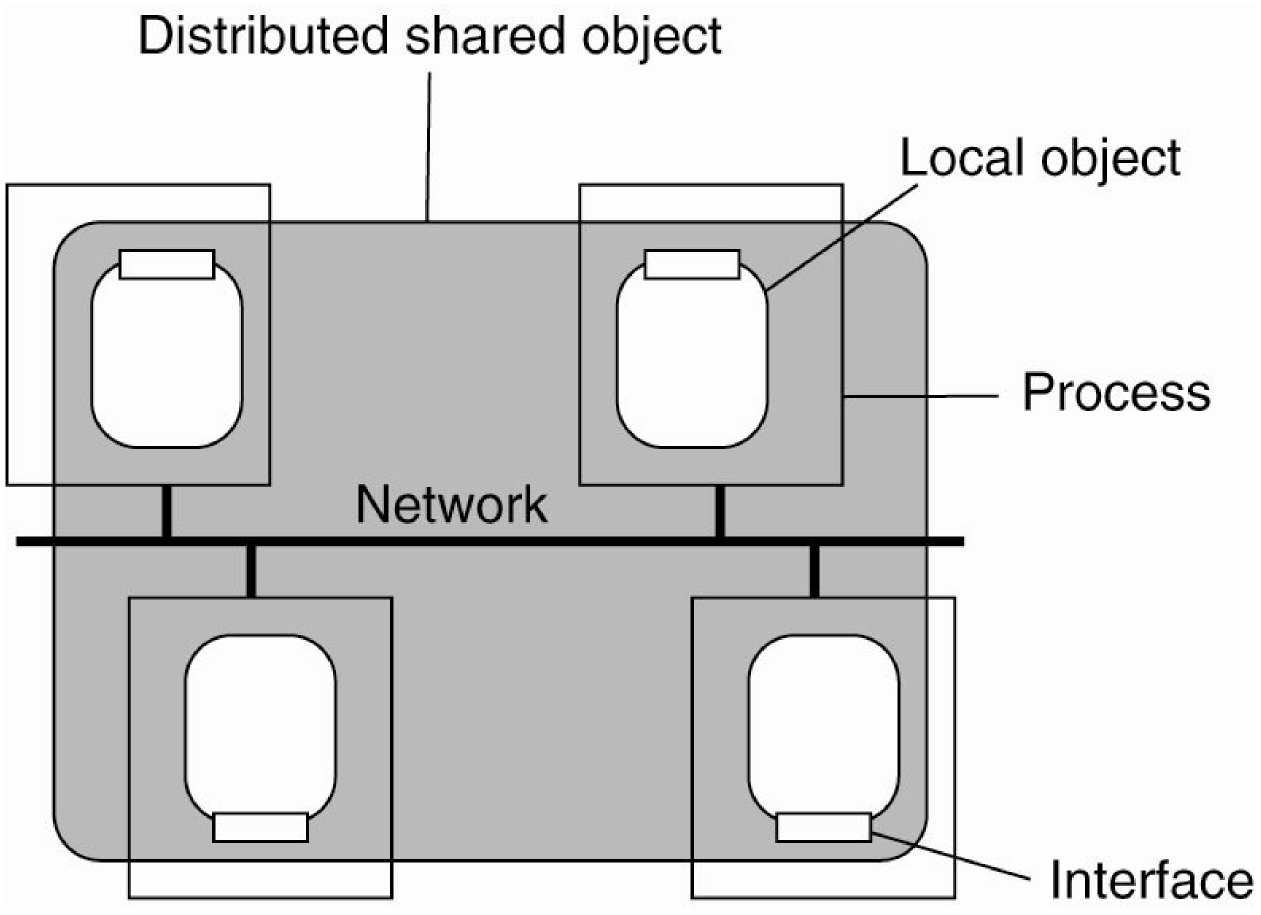


Figure 10-3. The organization of a Globe distributed shared object.

Globe Distributed Shared Objects (2)

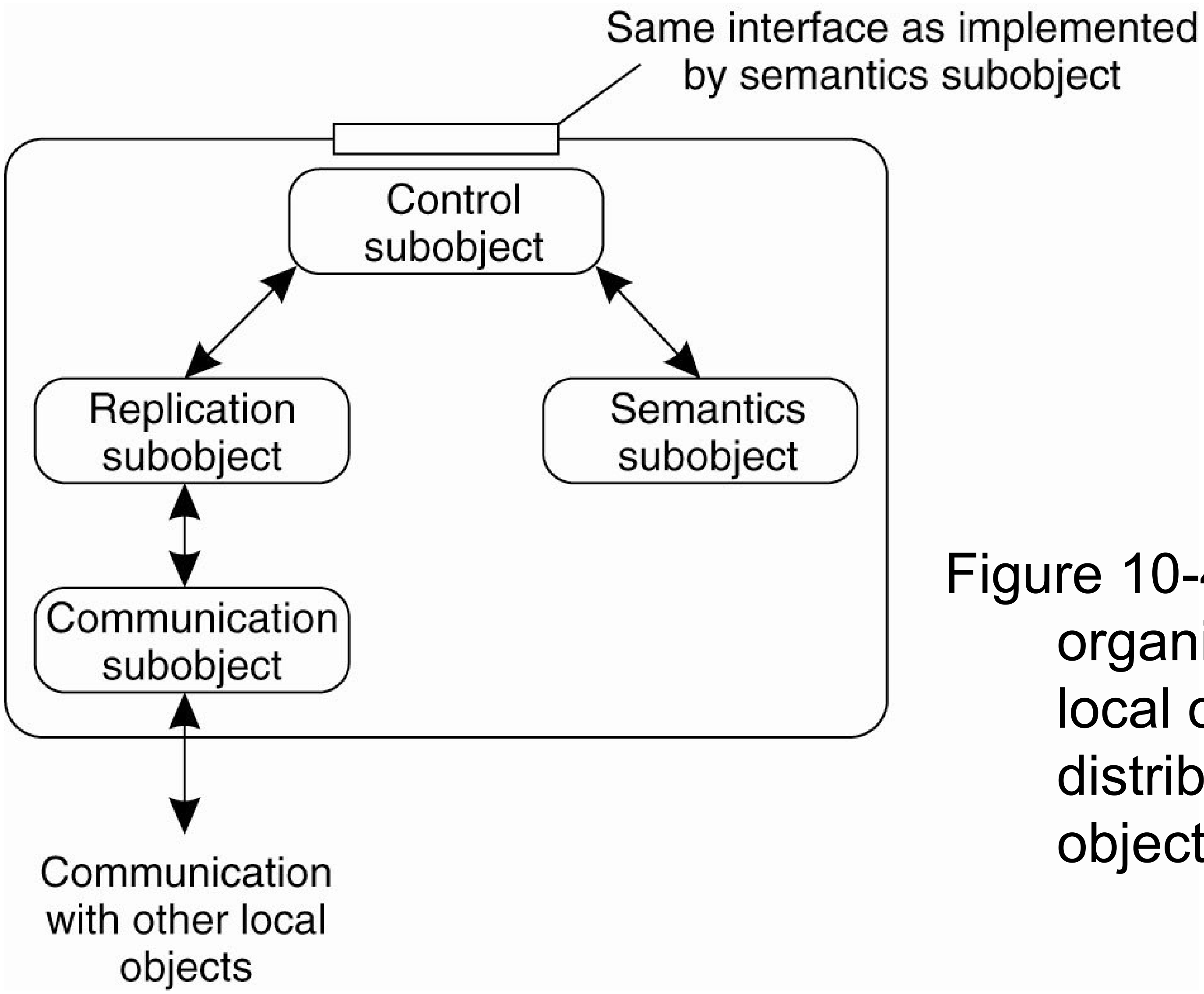

Same interface as implemented by semantics subobject
Control subobject
Replication subobject
Semantics subobject
Communication subobject
Communication with other local objects

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

Object Servers

A key role in object-based distributed systems is played by object servers, that is, the server designed to host 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. Indeed, 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.

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.

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

Object Adapter

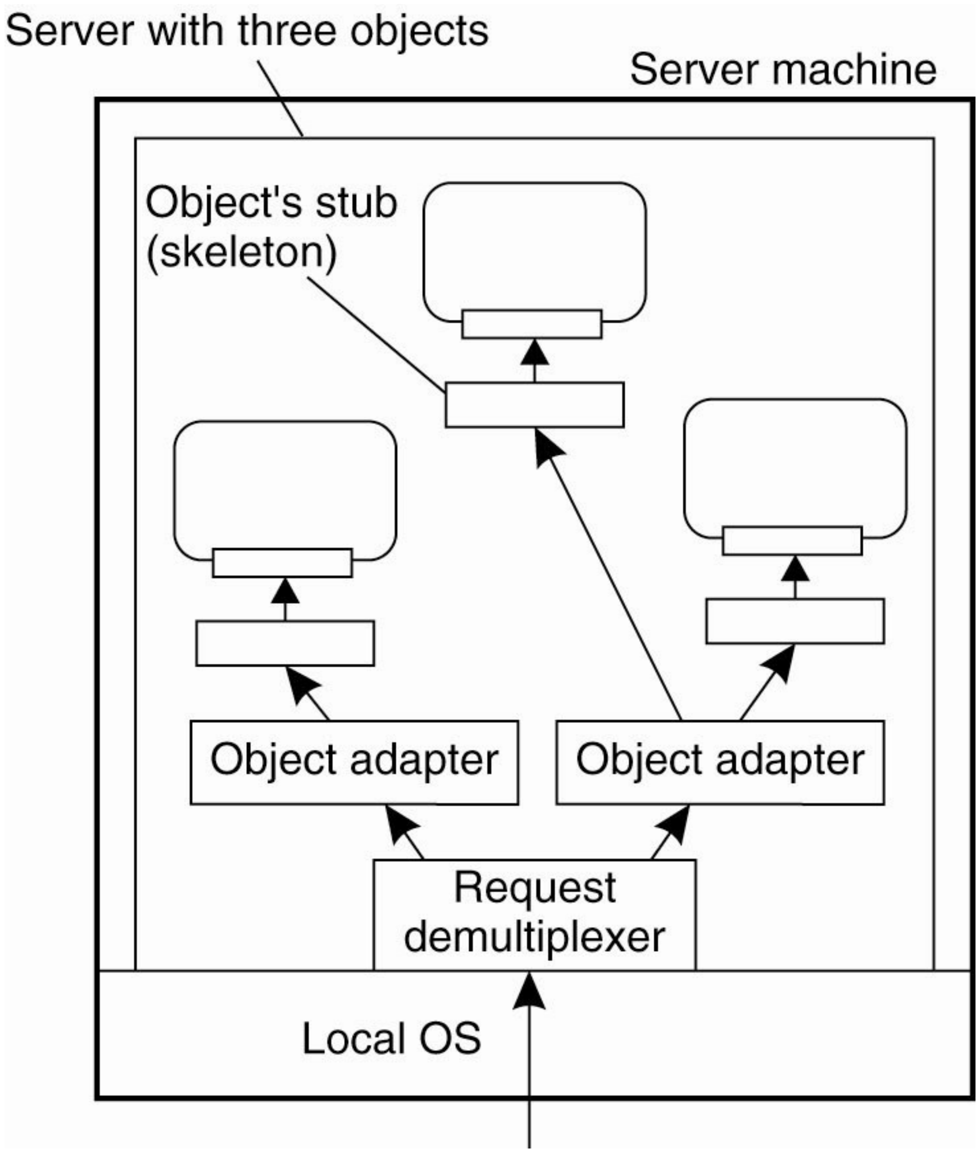


Figure 10-5. Organization of an object server supporting different activation policies.

Example: The Ice Runtime System

```
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();
}
```

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

Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall, Inc. All rights reserved. 0-13-239227-5

Communication
Binding a Client to an Object

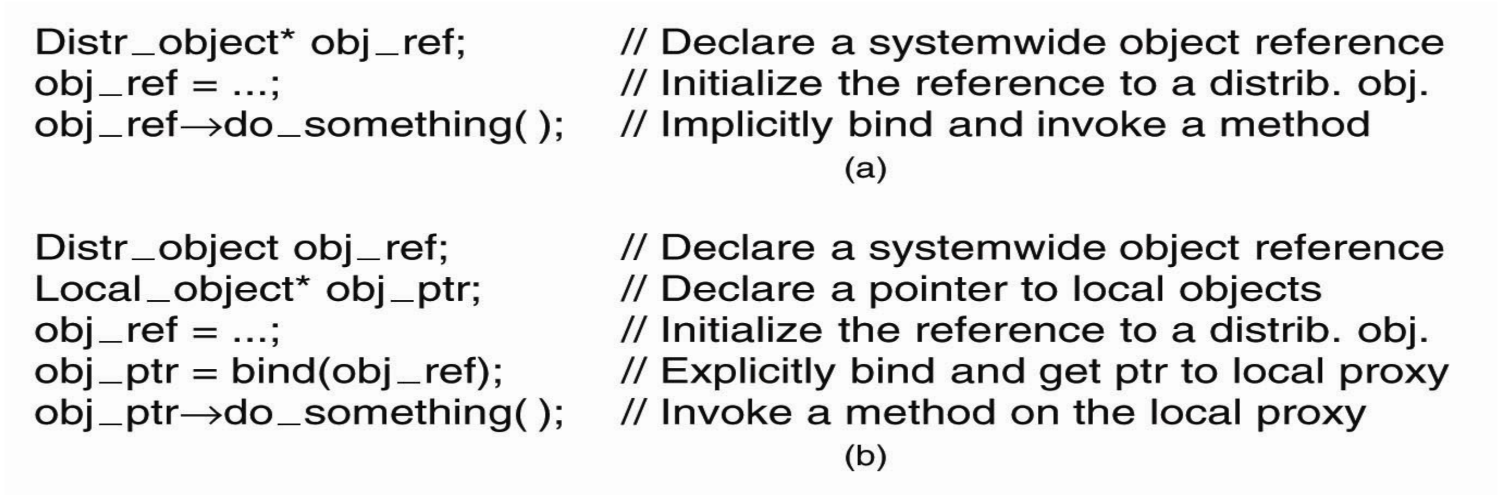
- These systems generally offer the means for a remote client to invoke an object.
- This mechanism is largely based on remote procedure calls (RPCs).
- A difference between traditional RPC systems and distributed objects is that the latter generally provides system-wide **object references**.
- Object 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, 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 an object reference is given, it needs a way to locate the server that manages the actual object, and place a proxy in the client's address space.

Communication

Binding a Client to an Object

•With **implicit binding, the client is offered a simple mechanism that allows it** to directly invoke methods using only a reference to an 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.



```
Distr_object* obj_ref;          // Declare a systemwide object reference
obj_ref = ...;                  // Initialize the reference to a distrib. obj.
obj_ref→do_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_ptr→do_something( );        // Invoke a method on the local proxy
                                           (b)
```

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

Implementation of Object References

- 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.
  - An indication of which object. Note that part of this information will be provided by an object adapter.
- Drawbacks:
  - First: if the server's machine crashes and the server is assigned a different end point after recovery, all object references have become invalid.

  Solution: DCE has 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.

Implementation of Object References

Drawbacks:
Second: 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 system-wide identifier for the server. Note that this solution comes close to implementing flat name spaces.

Implementation of Object References

So far 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.

This assumption is dropped provided that more information is added 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.

Implementation of Object References

This approach may 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-vl.la.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**.

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 system-wide object references as explained above.

Static RMI:

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.

Dynamic RMI:

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, inputParameters, outputParameters);

Parameter Passing

Let us first consider the situation that there are only distributed objects. 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 co-located 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.

Parameter Passing

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.

A client program running on machine A, and a server program on machine C. The client has a reference to a local object O1 that it uses as a parameter when calling the server program on machine C. In addition, it holds a reference to a remote object O2 residing at machine B, which is also used as a parameter. When calling the server, a copy of O1 is passed to the server on machine C, along with only a copy of the reference to O2.

Parameter Passing

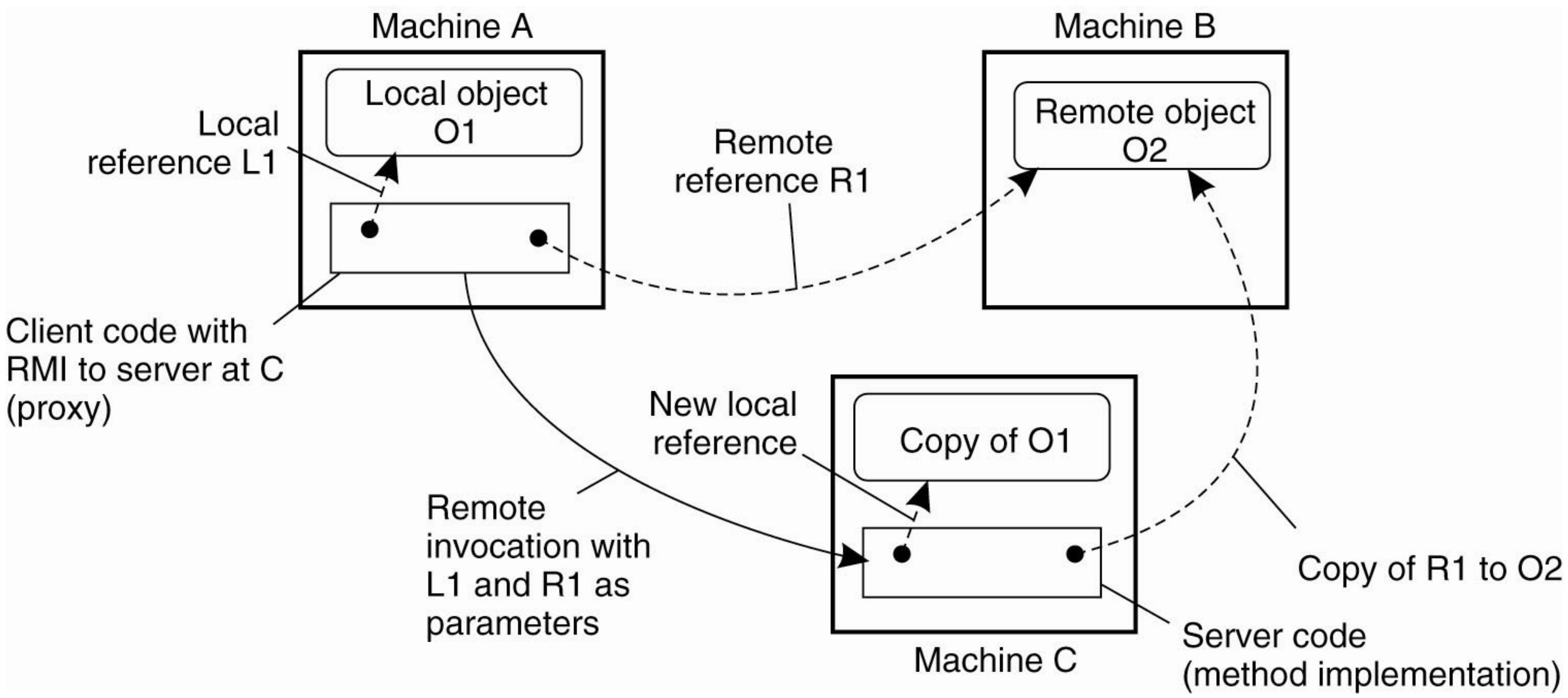


Figure 10-8. The situation when passing an object by reference or by value.

Parameter Passing

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.

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.

Object-Based Messaging (1)

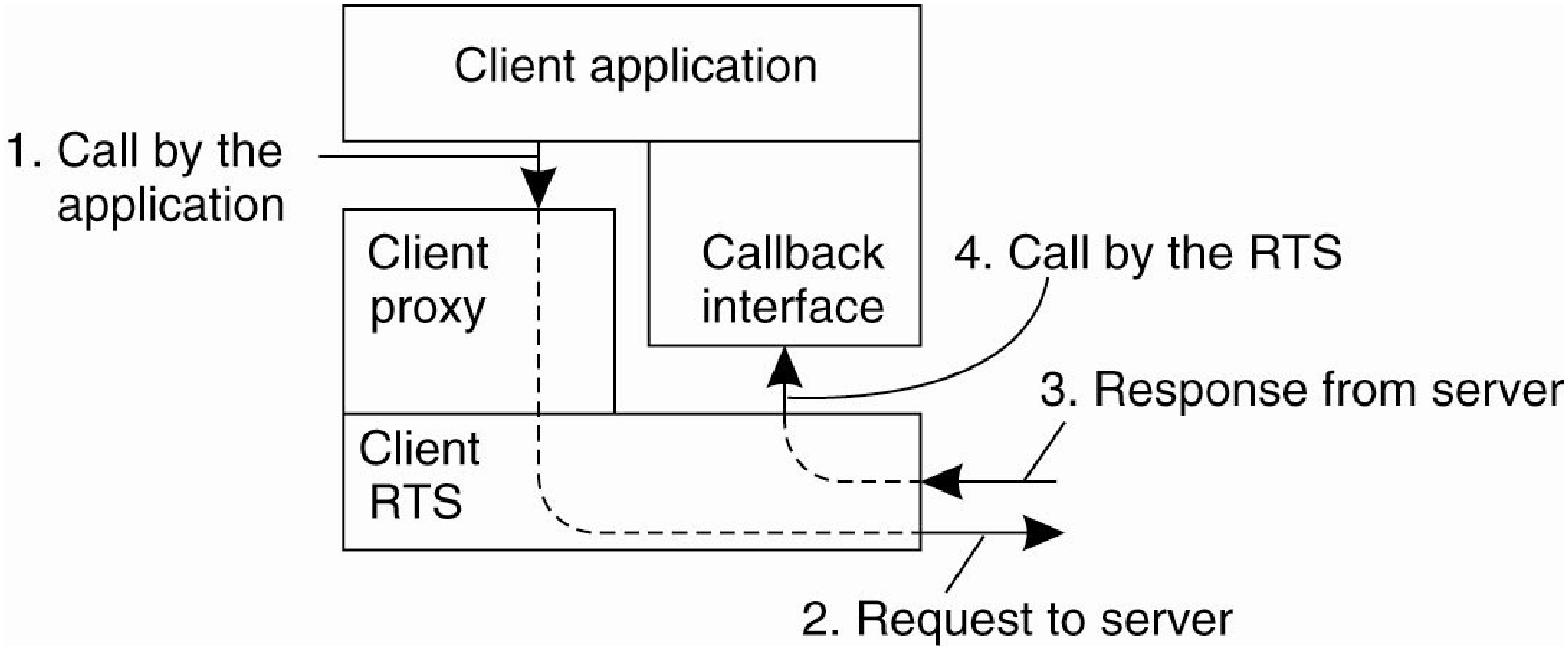


Figure 10-9. CORBA's callback model for asynchronous method invocation.

Object-Based Messaging (2)

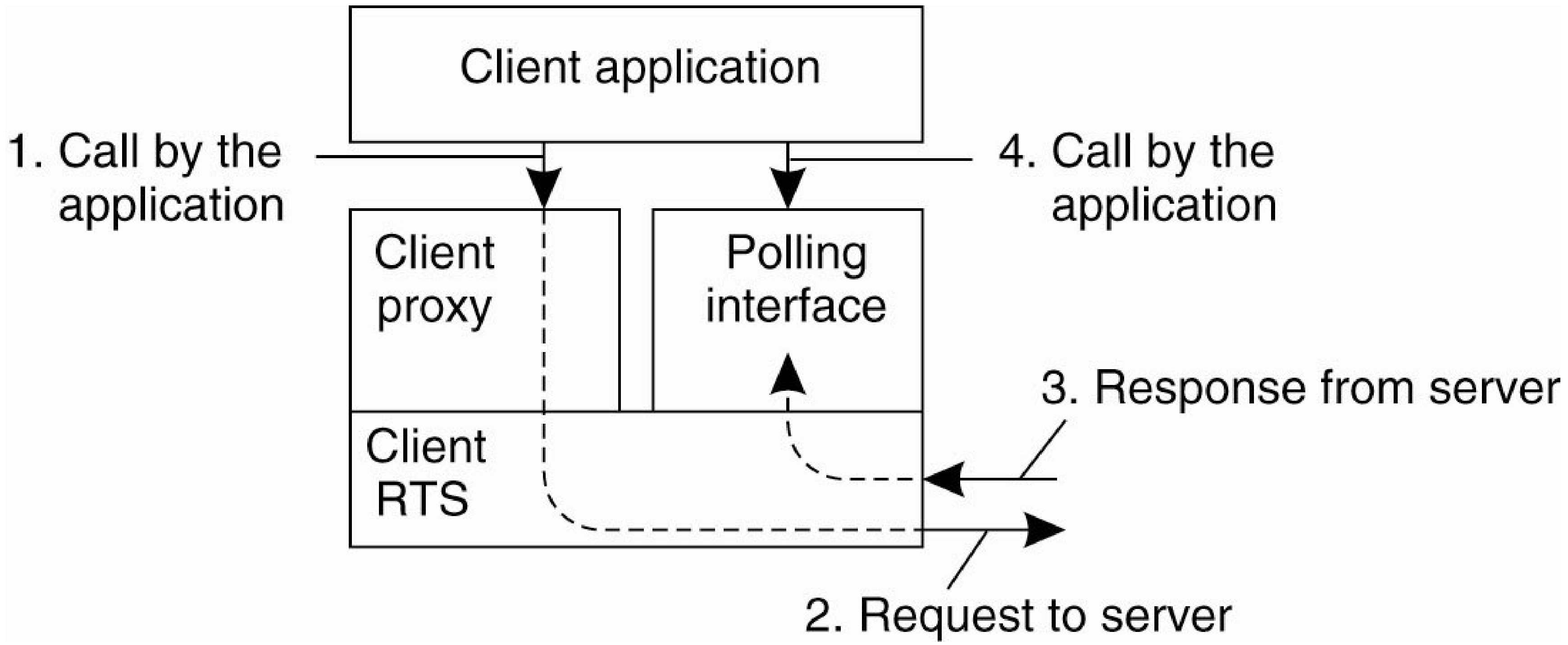


Figure 10-10. CORBA's polling model for asynchronous method invocation.

CORBA Object References

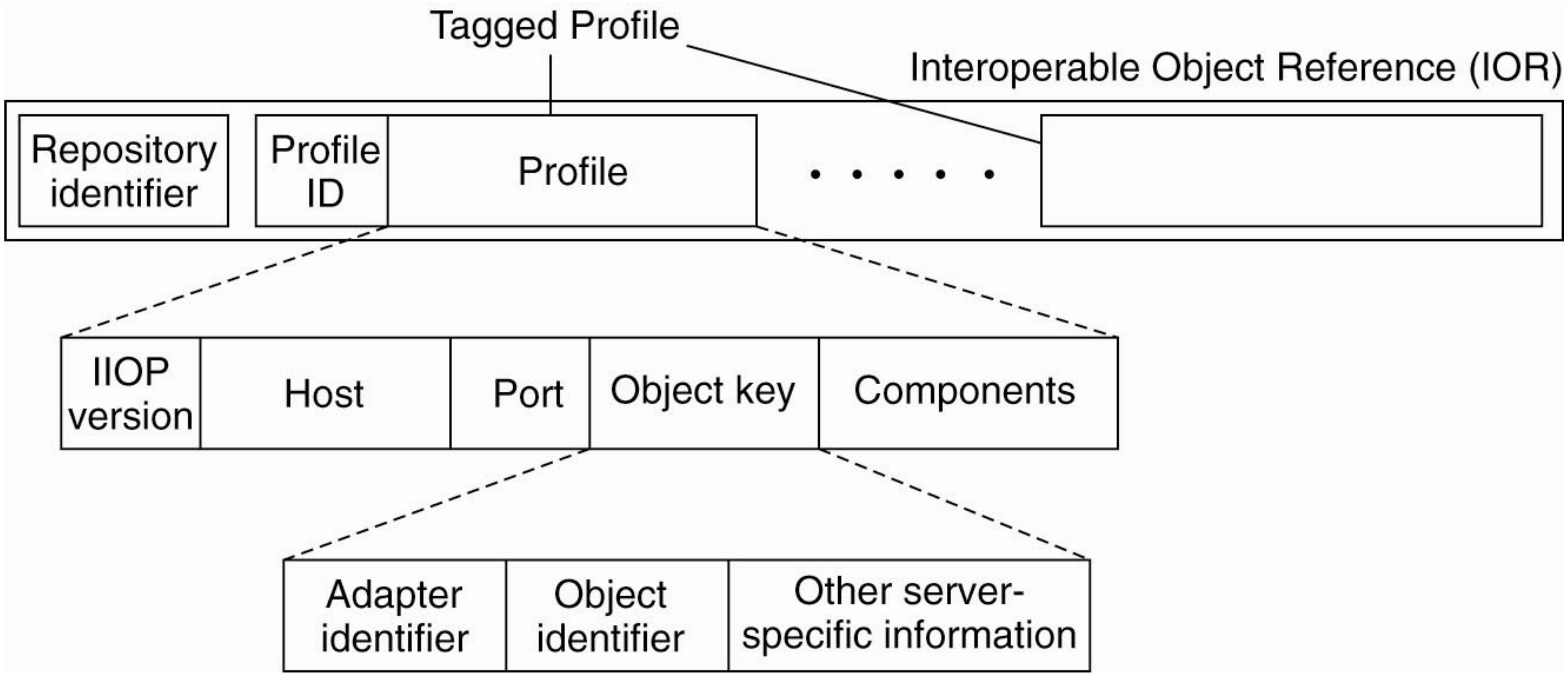
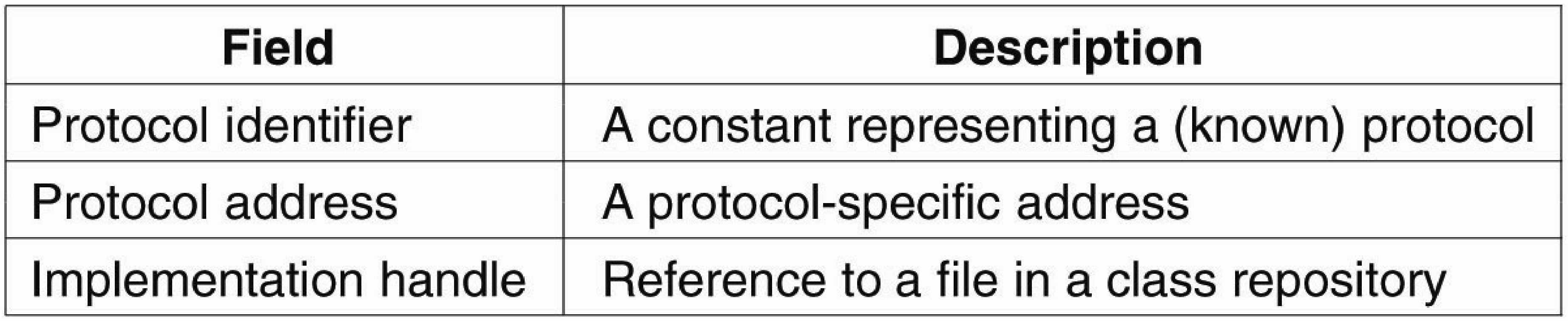


Figure 10-11. The organization of an IOR with specific information for IIOP.

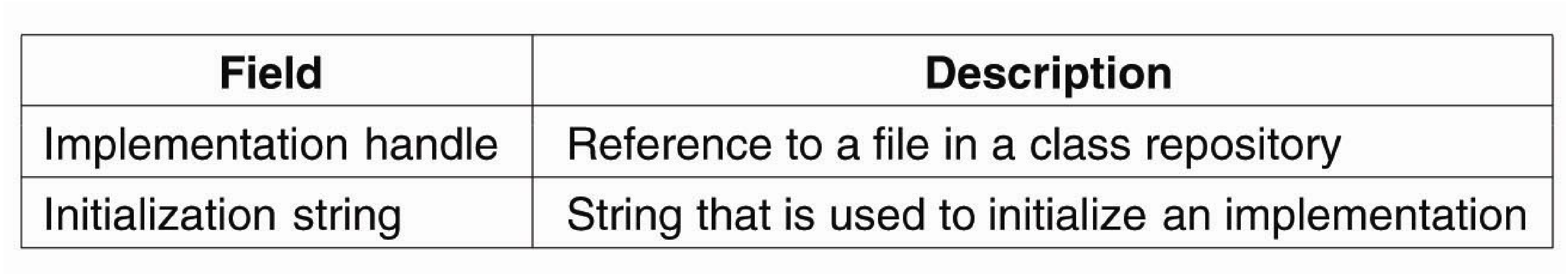
Globe Object References (1)



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

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

Globe Object References (2)



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

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

Synchronization
Locking

There are only a few issues regarding synchronization in distributed systems:

- When a process invokes a (remote) object, it has no knowledge whether that invocation will lead to invoking other objects. 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).
- 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).

Synchronization
Locking

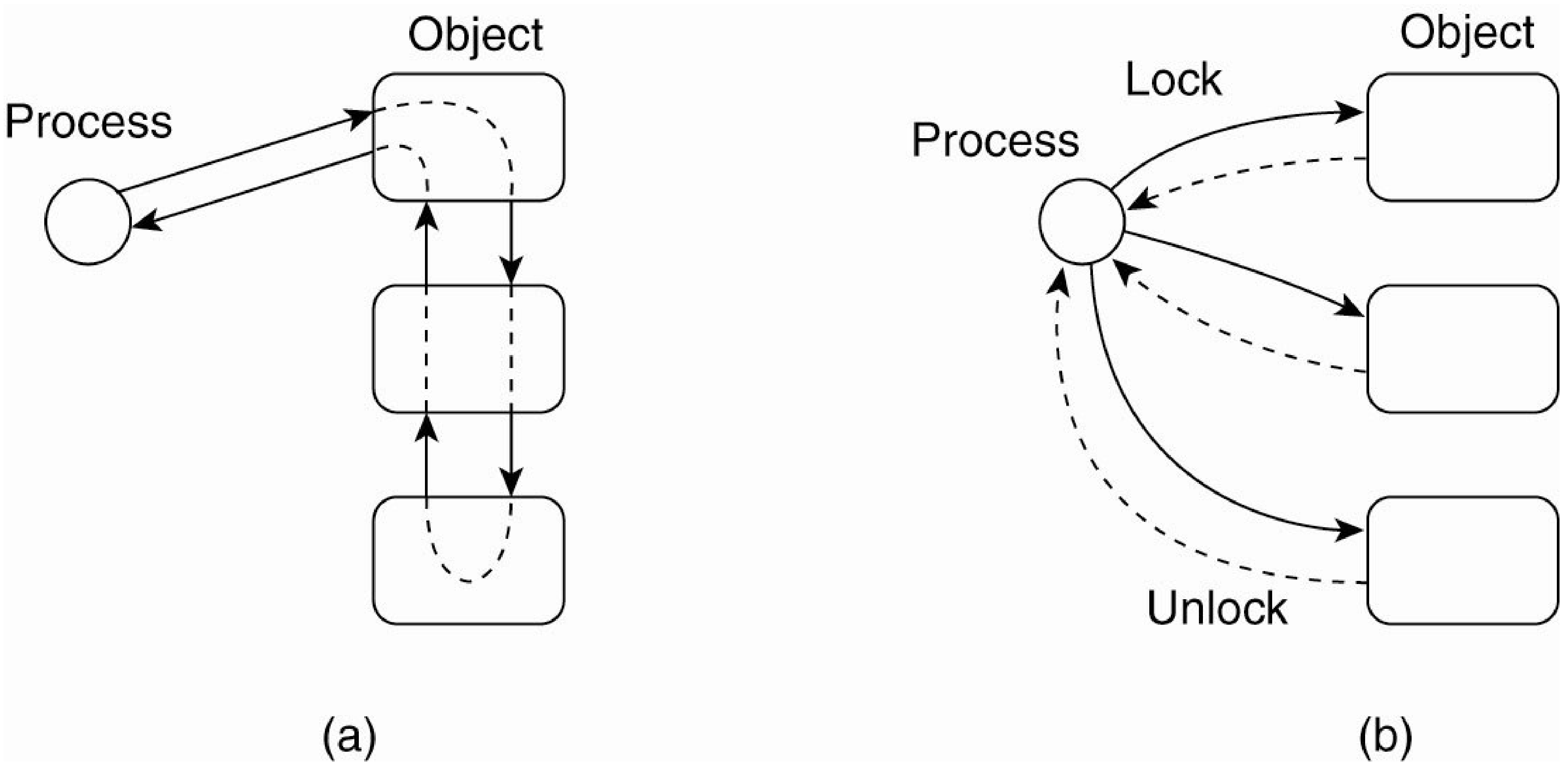


Figure 10-14. Differences in control flow for locking objects

Synchronization
Maintaining Locks

- 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.

Synchronization
Blocking in Remote Objects

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. the designers of Java RMI have chosen to restrict blocking on remote objects only to the proxies (Wollrath et aI., 1996).

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.

Consistency and Replication
Entry Consistency

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.

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.

Consistency and Replication
Entry Consistency

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 issue** 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.

Consistency and Replication
Entry Consistency

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.

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.

Consistency and Replication
Entry Consistency

• 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.
• 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 from Fig. 10-15 handle the same incoming (replicated) invocation request, they should both be scheduled before.

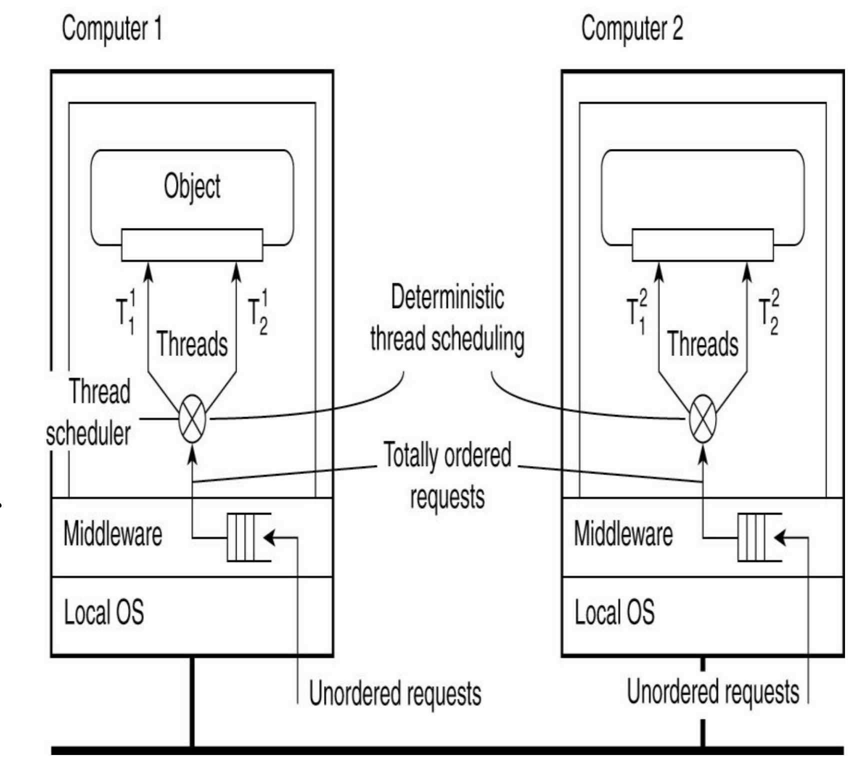


Figure 10-15. Deterministic thread scheduling for replicated object servers.

Consistency and Replication
Entry Consistency

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 aI. (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.

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 aI., 2005).

Replica 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**. A powerful mechanism to accomplish this separation is formed by interceptors.

Replication Frameworks

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.

Replication Frameworks

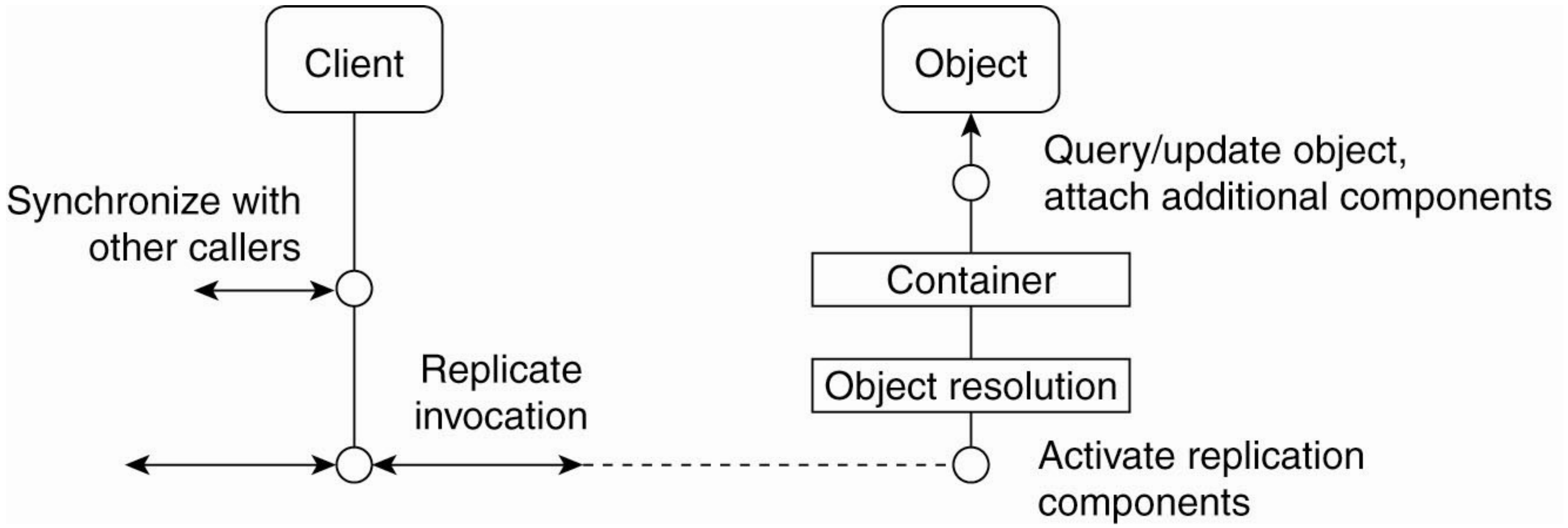


Figure 10-16. A general framework for separating replication algorithms from objects in an EJB environment.

Replication Frameworks

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.

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.

Replicated Invocations

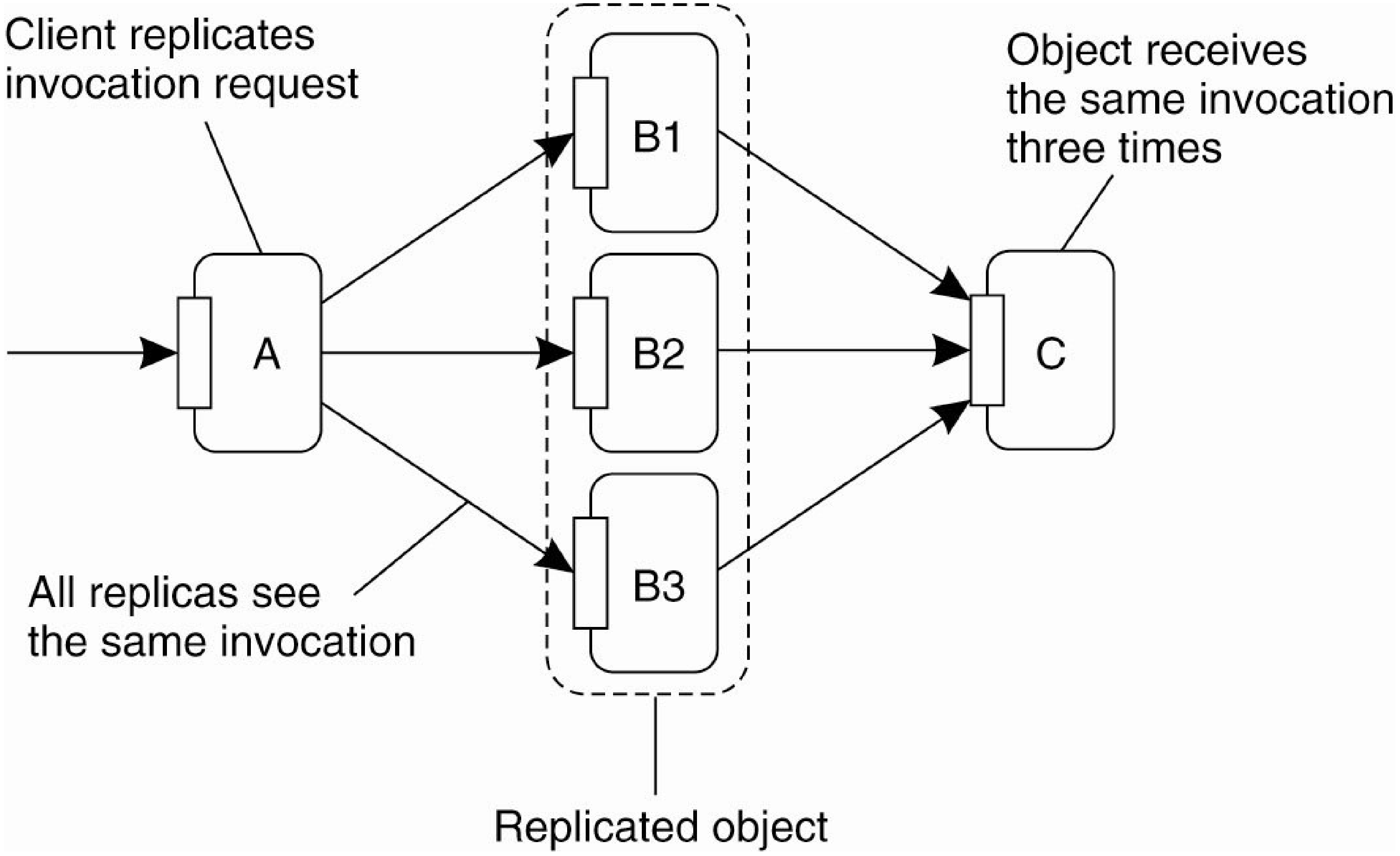

Client replicates invocation request
Object receives the same invocation three times
A
B1
B2
B3
C
All replicas see the same invocation
Replicated object

Figure 10-17. The problem of replicated method invocations.

Replicated Invocation

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.

There are not many general-purpose solutions to solve the problem of replicated invocations:

- One solution is to simply forbid it (Maassen et aI., 2001), which makes sense when performance is at stake.

- However, when replicating for fault tolerance, the following solution proposed by Mazouni et ale (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. l0-18(a). The result is that only a single request is forwarded to each replica of C.

Replication Invocation

- 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. l0-18(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. 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.

Replicated Invocations

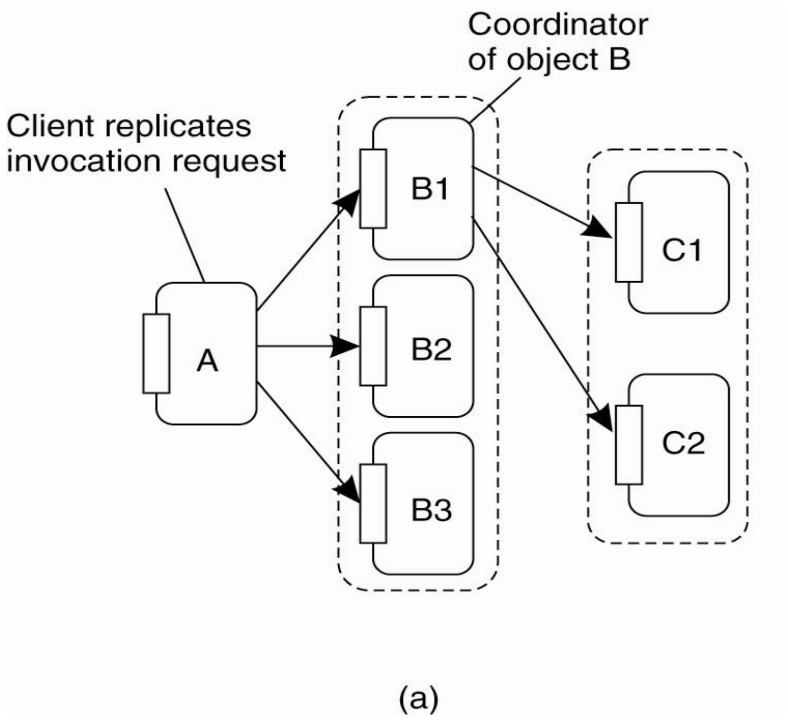


Figure 10-18. (a) Forwarding an invocation request from a replicated object to another replicated object.

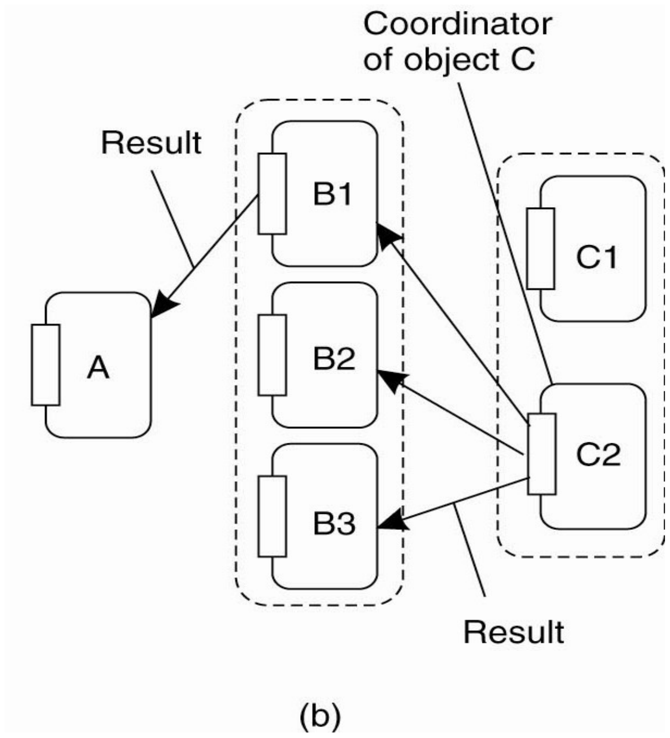


Figure 10-18. (b) Returning a reply from one replicated object to another.

Example: Fault-Tolerant CORBA

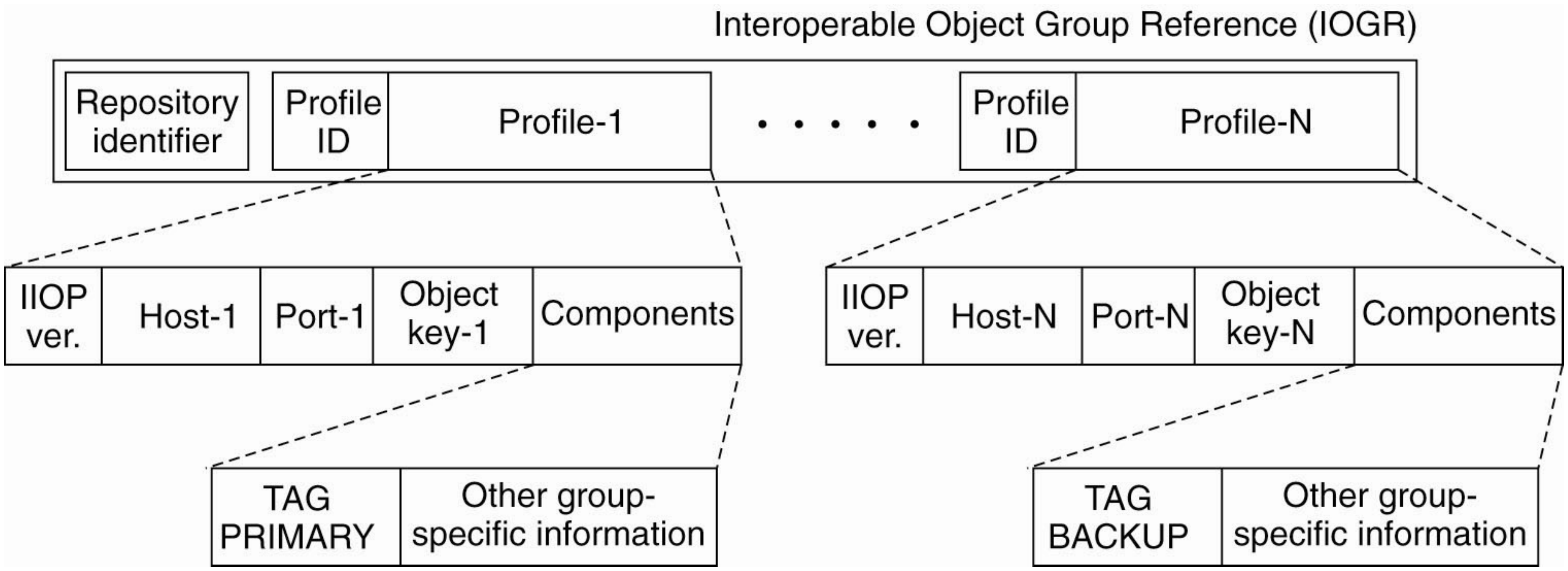


Figure 10-19. A possible organization of an IOGR for an object group having a primary and backups.

An Example Architecture

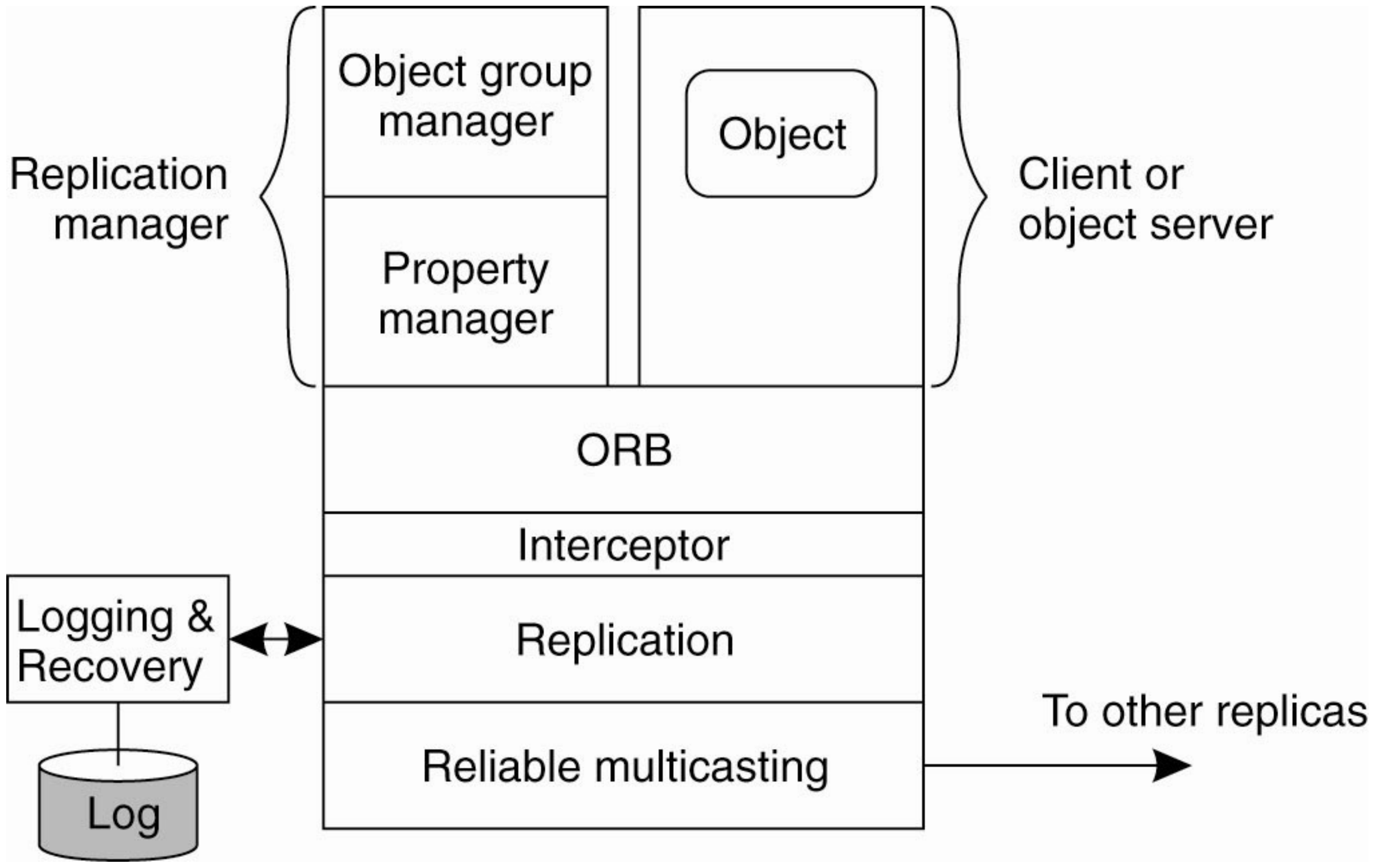


Figure 10-20. An example architecture of a fault-tolerant CORBA system.

Example: Fault-Tolerant Java

Causes for nondeterministic behavior:

1. JVM can execute native code, that is, code that is external to the JVM and provided to the latter through an interface.
2. Input data may be subject to nondeterminism.
3. In the presence of failures, different JVMs will produce different output revealing that the machines have been replicated.

Overview of Globe Security

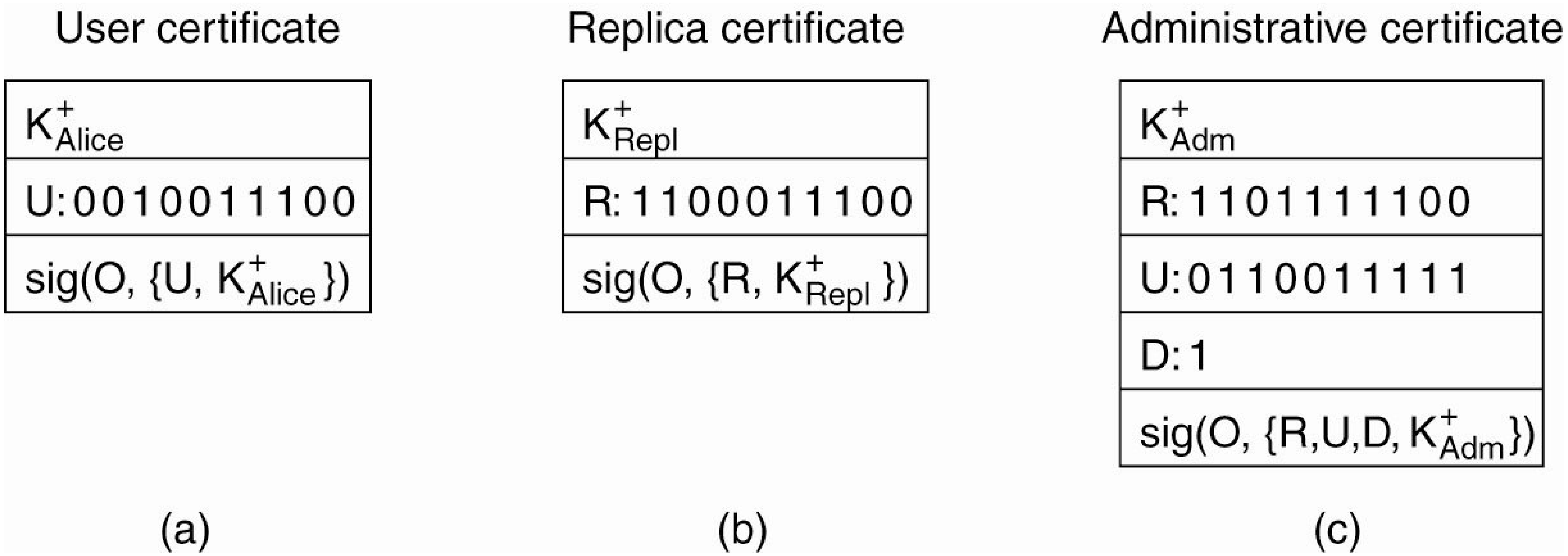


Figure 10-21. Certificates in Globe: (a) a user certificate, (b) a replica certificate, (c) an administrative certificate.

Secure Method Invocation (1)

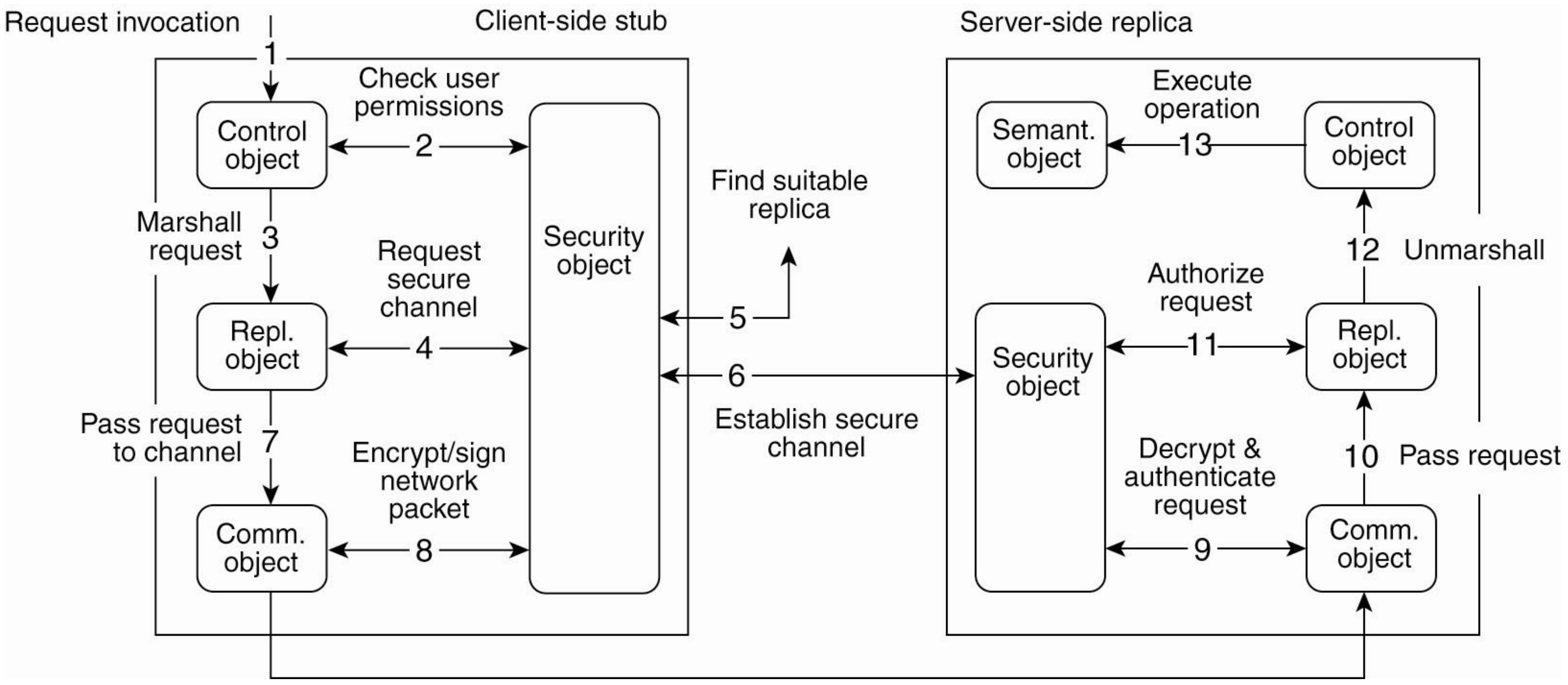


Figure 10-22. Secure method invocation in Globe.

Secure Method Invocation (2)

Steps for securely invoking a method of a Globe object:

1. Application issues a invocation request by locally calling the associated method
2. Control subobject checks the user permissions with the information stored in the local security object.
3. Request is marshaled and passed on.
4. Replication subobject requests the middleware to set up a secure channel to a suitable replica.

Secure Method Invocation (3)

5. Security object first initiates a replica lookup.
6. Once a suitable replica has been found, security subobject can set up a secure channel with its peer, after which control is returned to the replication subobject.
7. Request is now passed on to the communication subobject.
8. Subobject encrypts and signs the request so that it can pass through the channel.

Secure Method Invocation (4)

9. After its receipt, the request is decrypted and authenticated.
10. Request then passed on to the server-side replication subobject.
11. Authorization takes place:
12. Request is then unmarshaled.
13. Finally, the operation can be executed.