## 9.1 CORBA

We start our study of distributed object-based systems by taking a look at the **Common Object Request Broker Architecture**, simply referred to as **CORBA**. As its name suggests, CORBA is not so much a distributed system but rather the specification of one. These specifications have been drawn up by the **Object Management Group** (**OMG**), a nonprofit organization with over 800 members, primarily from industry. An important goal of the OMG with respect to CORBA was to define a distributed system that could overcome many of the interoperability problems with integrating networked applications. The first CORBA specifications became available in the beginning of the 1990s. At present, implementations of CORBA version 2.4 are widely deployed, whereas the first CORBA version 3 systems are becoming available.

Like many other systems that are the result of the work of committees, CORBA has features and facilities in abundance. The core specifications consist of well over 700 pages, and another 1,200 are used to specify the various services that are built on top of that core. And naturally, each CORBA implementation has its own extensions because there is always something that each vendor feels cannot be missed but was not included in the specifications. CORBA illustrates again that making a distributed system that is simple may be a somewhat overwhelmingly difficult exercise.

In the following pages, we will not discuss all the things that CORBA has to offer, but instead concentrate only on the parts that are essential to it as a distributed system and that characterize it with respect to other object-based distributed systems. The specifications of CORBA can be found in (OMG, 2001b), which is publicly available at *http://www.omg.org*. A high-level overview of CORBA is described in (Vinoski, 1997), whereas Pope (1998) provides a more detailed description derived from the original specifications. Information on building applications in C++ using CORBA can be found in (Baker, 1997) and (Henning and Vinoski, 1999).

### 9.1.1 Overview of CORBA

The global architecture of CORBA adheres to a reference model of the OMG that was laid down in (OMG, 1997). This reference model, shown in Fig. 9-1, consists of four groups of architectural elements connected to what is called the **Object Request Broker** (**ORB**). The ORB forms the core of any CORBA distributed system; it is responsible for enabling communication between objects and their clients while hiding issues related to distribution and heterogeneity. In many systems, the ORB is implemented as libraries that are linked with a client and server application, and that offers basic communication services. We return to the ORB below when discussing CORBA's object model.

Besides objects that are built as part of specific applications, the reference

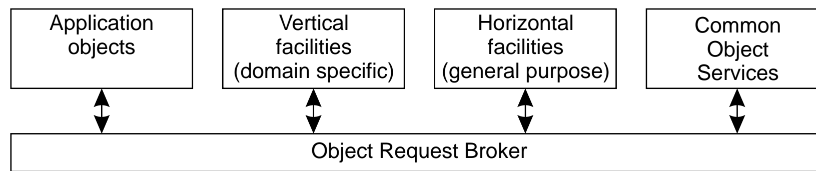


**Figure 9-1.** The global architecture of CORBA.

model also distinguishes what are known as **CORBA facilities**. Facilities are constructed as compositions of CORBA services (which we discuss below), and are split into two different groups. **Horizontal facilities** consist of general-purpose high-level services that are independent of application domains. Such services currently include those for user interfaces, information management, system management, and task management (which is used to define workflow systems). **Vertical facilities** consist of high-level services that are targeted to a specific application domain such as electronic commerce, banking, manufacturing, etc.

We will not discuss application objects and CORBA facilities in any detail, but rather concentrate on the basic services and the ORB.

### Object Model

CORBA uses the remote-object model that we described in Chap. 2. In this model, the implementation of an object resides in the address space of a server. It is interesting to note that the CORBA specifications never explicitly state that objects should be implemented only as remote objects. However, virtually all CORBA systems support only this model. In addition, the specifications often suggest that distributed objects in CORBA are actually remote objects. Later, when discussing the Globe object model, we show how a completely different model of an object could, in principle, be equally well supported by CORBA.

Objects and services are specified in the CORBA **Interface Definition Language** (**IDL**). CORBA IDL is similar to other interface definition languages in that it provides a precise syntax for expressing methods and their parameters. It is not possible to describe semantics in CORBA IDL. An interface is a collection of methods, and objects specify which interfaces they implement.

Interface specifications can be given only by means of IDL. As we shall see later, in systems such as Distributed COM and Globe, interfaces are specified at a lower level in the form of tables. These so-called **binary interfaces** are by their nature independent of any programming language. In CORBA, however, it is necessary to provide exact rules concerning the mapping of IDL specifications to existing programming languages. At present, such rules have been given for a number of languages, including C, C++, Java, Smalltalk, Ada, and COBOL.

Given that CORBA is organized as a collection of clients and object servers, the general organization of a CORBA system is shown in Fig. 9-2.

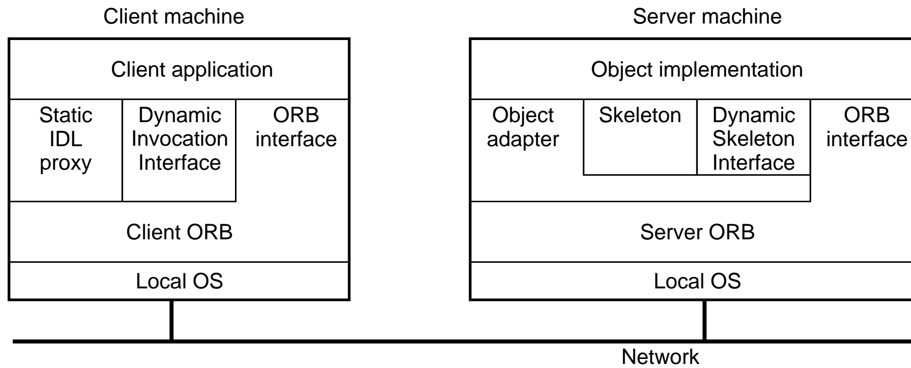


**Figure 9-2.** The general organization of a CORBA system.

Underlying any process in CORBA, be it a client or server, is the ORB. The ORB can best be seen as the runtime system that is responsible for handling the basic communication between a client and an object. This basic communication consists of ensuring that an invocation is sent to the object's server and that the reply is passed back to the client.

From the perspective of a process, the ORB offers only a few services itself. One of these services is manipulating object references. Such references are generally dependent on a particular ORB. An ORB will therefore offer operations to marshal and unmarshal object references so that they can be exchanged between processes, as well as operations for comparing references. Object references are discussed in detail below.

Other operations offered by an ORB deal with initially finding the services that are available to a process. In general, it provides a means to obtain an initial reference to an object implementing a specific CORBA service. For example, in order to make use of a naming service, it is necessary that a process knows how to refer to that service. These initialization aspects apply equally well to other services.

Besides the ORB interface, clients and servers see hardly anything of the ORB. Instead, they generally see only stubs for handling method invocations for specific objects. A client application usually has a proxy available that implements the same interface as each object it is using. As we explained in Chap. 2, a proxy is a client-side stub that merely marshals an invocation request and sends that request to the server. A response from that server is unmarshaled and passed back to the client.

Note that the interface between a proxy and the ORB does not have to be standardized. Because CORBA assumes that all interfaces are given in IDL, CORBA implementations offer an IDL compiler to developers that generates the necessary code to handle communication between the client and server ORB.

However, there are occasions in which statically defined interfaces are simply

not available to a client. Instead, what it needs is to find out during runtime what the interface to a specific object looks like, and subsequently compose an invocation request for that object. For this purpose, CORBA offers a **Dynamic Invocation Interface** (**DII**) to clients, which allows them to construct an invocation request at runtime. In essence, the DII provides a generic invoke operation, which takes an object reference, a method identifier, and a list of input values as parameters, and returns its result in a list of output variables provided by the caller.

Object servers are organized in the way we described in Chap. 3. As shown in Fig. 9-2, a CORBA system provides an object adapter, which takes care of forwarding incoming requests to the proper object. The actual unmarshaling at the server side is done by means of stubs, called skeletons in CORBA, but it is also possible that the object implementation takes care of unmarshaling. As in the case of clients, server-side stubs can either be statically compiled from IDL specifications, or be available in the form a generic dynamic skeleton. When using a dynamic skeleton, an object will have to provide the proper implementation of the invoke function as offered to the client. We return to object servers below.

### Interface and Implementation Repository

To allow the dynamic construction of invocation requests, it is important that a process can find out during runtime what an interface looks like. CORBA offers an **interface repository**, which stores all interface definitions. In many systems, the interface repository is implemented by means of a separate process offering a standard interface to store and retrieve interface definitions. An interface repository can also be viewed as that part of CORBA that assists in runtime type checking facilities.

Whenever an interface definition is compiled, the IDL compiler assigns a **repository identifier** to that interface. This repository ID is the basic means to retrieve an interface definition from the repository. The identifier is by default derived from the name of the interface and its methods, implying that no guarantees are given with respect to its uniqueness. If uniqueness is required, the default can be overridden.

Given that all interface definitions stored in an interface repository adhere to IDL syntax, it becomes possible to organize each definition in a standard way. (In database terminology, this means that the conceptual schema associated with an interface repository is the same for every repository.) As a consequence, the interface repositories in CORBA systems offer the same operations for navigating through interface definitions.

Besides an interface repository, a CORBA system generally offers also an **implementation repository**. Conceptually, an implementation repository contains all that is needed to implement and activate objects. Because such functionality is intimately related to the ORB itself and the underlying operating system, it is difficult to provide a standard implementation.

An implementation repository is also tightly coupled to the organization and implementation of object servers. As we explained in Chap. 3, an object adapter has the responsibility for activating an object by ensuring that it is running in the address space of a server so that its methods can be invoked. Given an object reference, an adapter could contact the implementation repository to find out exactly what needs to be done.

For example, the implementation repository could maintain a table specifying that a new server should be started and also to which port number the new server should be listening for the specified object. The repository would furthermore have information about which executable file (i.e., binary progra) the server should load and execute.

Alternatively, it may not be necessary to start a separate server, but the current one need merely link to a specific library containing the requested method or object. Again, such information would typically be stored in an implementation repository. These two examples illustrate that such a repository is indeed closely tied to an ORB and the platform on which it is running.

### CORBA Services

An important part of CORBA's reference model is formed by the collection of CORBA services. A CORBA service is best thought of as being general purpose and independent of the application for which CORBA is being used. As such, CORBA services strongly resemble the types of services commonly provided by operating systems. There is a whole list of services specified for CORBA, as shown in Fig. 9-3. Unfortunately, it is not always possible to draw a clear line between the different services, as they often have overlapping functionality. Let us briefly describe each service so that we can later make a better comparison to services as offered by DCOM and Globe.

The **collection service** provides the means to group objects into lists, queues, stacks, sets, and so on. Depending on the nature of the group, various access mechanisms are offered. For example, lists can be inspected elementwise through what is generally referred to as an iterator. There are also facilities to select objects by specifying a key value. In a sense, the collection service comes close to what is generally offered by class libraries for object-oriented programming languages.

There is also a separate **query service** that provides the means to construct collections of objects that can be queried using a declarative query language. A query may return a reference to an object or to a collection of objects. The query service augments the collection service with advanced queries. It differs from the collection service in that the latter offers various types of collections.

There is also a **concurrency control service**. It offers advanced locking mechanisms by which clients can access shared objects. This service can be used to implement transactions, which are offered by a separate service. The

**transaction service** allows a client to define a series of method invocations across multiple objects in a single transaction. The service supports flat and nested transactions.

| Service | Description |
|---|---|
| Collection | Facilities for grouping objects into lists, queue, sets, etc. |
| Query | Facilities for querying collections of objects in a declarative manner |
| Concurrency | Facilities to allow concurrent access to shared objects |
| Transaction | Flat and nested transactions on method calls over multiple objects |
| Event | Facilities for asynchronous communication through events |
| Notification | Advanced facilities for event-based asynchronous communication |
| Externalization | Facilities for marshaling and unmarshaling of objects |
| Life cycle | Facilities for creation, deletion, copying, and moving of objects |
| Licensing | Facilities for attaching a license to an object |
| Naming | Facilities for systemwide naming of objects |
| Property | Facilities for associating (attribute, value) pairs with objects |
| Trading | Facilities to publish and find the services an object has to offer |
| Persistence | Facilities for persistently storing objects |
| Relationship | Facilities for expressing relationships between objects |
| Security | Mechanisms for secure channels, authorization, and auditing |
| Time | Provides the current time within specified error margins |

**Figure 9-3.** Overview of CORBA services.

Normally, clients invoke methods on objects and wait for the result of that invocation. To support asynchronous communication, CORBA supports an **event service** by which clients and objects can be interrupted upon the occurrence of a specified event. Advanced facilities for asynchronous communication are provided by a separate **notification service**. We describe these services in more detail below.

**Externalization** deals with marshaling objects in such a way that they can be stored on disk or sent across a network. It is comparable to the serialization facilities offered by Java, allowing objects to be written to a data stream as a series of bytes.

The **life cycle service** provides the means to create, destroy, copy, and move objects. A key concept is that of a **factory object**, which is a special object used to create other objects (Gamma et al., 1994). Practice indicates that only the creation of objects needs to be handled by a separate service. However, destroying, copying, and moving objects is often conveniently defined by objects themselves. The reason is that these operations often affect an object's state in an object-specific way.

The **licensing service** allows developers of objects to attach a license to their object and enforce a specific licensing policy. A license expresses the rights a client has with respect to using an object. For example, a license attached to an object may enforce that the object can be used by only a single client at a time. Another license may ensure that an object is automatically disabled after a certain expiration time.

CORBA offers a separate **naming service** by which objects can be given a human-readable name that maps to the object's identifier. The basic facilities for describing objects is provided by a separate **property service**. This service allows clients to associate *(attribute, value)*-pairs with objects. Note that these attributes are not part of the object's state, but instead are used to describe the object. In other words, they provide information about the object instead of being part of it. Related to these two services is a **trading service** that allows objects to advertise what they have to offer (by means of their interfaces), and to allow clients to find services using a special language that supports the description of constraints.

A separate **persistence service** offers the facilities for storing information on disk in the form of storage objects. An important issue here is that persistence transparency is provided; a client need not explicitly transfer the data in a storage object between a disk and available main memory.

None of the services so far offer the facilities to explicitly relate two or more objects. These facilities are provided by a **relationship service**, which essentially provides support for organizing objects according to a conceptual schema like the ones used in databases.

Security is provided in a **security service**. The implementation of this service is comparable to security systems such as SESAME and Kerberos. The CORBA security service provides facilities for authentication, authorization, auditing, secure communication, nonrepudiation, and administration. We return to security in detail below.

Finally, CORBA offers a **time service** that returns the current time within specified error ranges.

As explained by Pope (1998), the CORBA services have been designed with CORBA's object model as their basis. This means that all services are specified in CORBA IDL, and that a separation between interface specification and implementation is made. Another important design principle is that services should be minimal and simple. In the following sections we discuss a number of these services in more detail. From those descriptions, it can be argued to what extent this last principle has been successfully applied.

### 9.1.2 Communication

Originally, CORBA had a simple model of communication: a client invokes a method on an object and waits for an answer. This model was thought to be too

simple and soon additional communication facilities were added. In the following, we take a closer look at invocation facilities in CORBA, and also consider its alternatives to these object invocations. As we will see, the extensions to the basic object invocation model are motivated by the need for asynchronous communication. This motivation also led to the alternative message-passing models that we discussed in Chap. 2.

### Object Invocation Models

By default, when a client invokes an object, it sends a request to the corresponding object server and blocks until it receives a response. In the absence of failures, these semantics correspond exactly to a normal method invocation when the caller and callee reside in the same address space.

However, matters become somewhat more complicated in the presence of failures. In the case of a synchronous invocation, as just described, a client will eventually receive an exception indicating that the invocation did not fully complete. CORBA specifies that in this case, the invocation should follow at-most-once semantics, implying that the invoked method may have been invoked once, or not at all. Note that it is up to an implementation to provide these semantics.

Synchronous invocation is therefore useful when the client expects an answer. If a proper response is returned, CORBA guarantees that the method has been invoked exactly once. However, in those cases where no response is needed, it would be better for the client to simply invoke the method and continue with its own execution as soon as possible. This type of invocation is similar to the asynchronous RPCs we discussed in Chap. 2.

In CORBA, this form of asynchronous invocation is called a **one-way request**. A method can be specified as being one-way only if it is specified to return no results. However, unlike the guaranteed delivery of asynchronous RPCs, one-way requests in CORBA offer only a best-effort delivery service. In other words, no guarantees are given to the caller that the invocation request is delivered to the object's server.

Besides one-way requests, CORBA also supports what is known as a **deferred synchronous request**. Such a request is, in fact, a combination of a one-way request and letting the server asynchronously send the result back to the client. As soon as the client sends its request to the server, it continues without waiting for any response from the server. In other words, the client can never know for sure whether its request is actually delivered to the server.

These three different invocation models are summarized in Fig. 9-4.

| Request type | Failure semantics | Description |
|---|---|---|
| Synchronous | At-most-once | Caller blocks until a response is returned or an exception is raised |
| One-way | Best effort delivery | Caller continues immediately without waiting for any response from the server |
| Deferred synchronous | At-most-once | Caller continues immediately and can later block until response is delivered |

**Figure 9-4.** Invocation models supported in CORBA.

### Event and Notification Services

Although the invocation models offered by CORBA should normally cover most of the communication requirements in an object-based distributed system, it was felt that having only method invocations was not enough. In particular, there was a need to provide a service that could simply signal the occurrence of an event. Clients amenable to that event could take appropriate action.

The result is the definition of an **event service**. The basic model for events in CORBA is quite simple. Each event is associated with a single data item, generally represented by means of an object reference, or otherwise an application-specific value. An event is produced by a **supplier** and received by a **consumer**. Events are delivered through an **event channel**, which is logically placed between suppliers and consumers, as shown in Fig. 9-5.

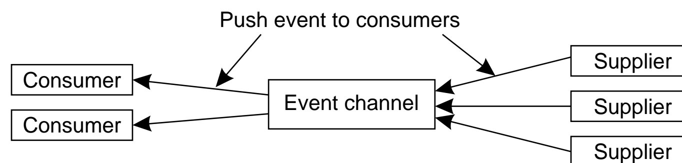


**Figure 9-5.** The logical organization of suppliers and consumers of events, following the push-style model.

Fig. 9-5 shows what is referred to as the **push model**. In this model, whenever an event occurs, the supplier produces the event and pushes it through the event channel, which then passes the event on to its consumers. The push model comes closest to the asynchronous behavior that most people associate with events. In effect, consumers passively wait for event propagation, and expect to be interrupted one way or the other when an event happens.

An alternative model, also supported by CORBA, is the **pull model** shown in Fig. 9-6. In this model, consumers poll the event channel to check whether an event has happened. The event channel, in turn, polls the various suppliers.

Although the event service provides a simple and straightforward way for

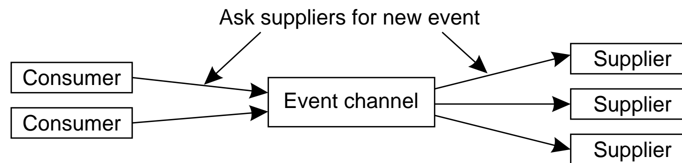


**Figure 9-6.** The pull-style model for event delivery in CORBA.

event propagation, it has a number of serious drawbacks. In order to propagate events, it is necessary that suppliers and consumers are connected to the event channel. This also means that when a consumer connects to an event channel after the occurrence of an event, that afterward, the event will be lost. In other words, CORBA's event service does not support persistence of events.

More serious is that consumers have hardly any means to filter events; each event is, in principle, passed to every consumer. If different event types need to be distinguished, it is necessary to set a separate event channel for each type. Filtering capabilities have been added to an extension called the **notification service** (OMG, 2000a). In addition, this service offers facilities to prevent event propagation when no consumers are interested in a specific event.

Finally, event propagation is inherently unreliable. The CORBA specifications state that no guarantees need to be given concerning the delivery of events. As we will discuss in Chap. 12, applications exist for which reliable event propagation is important. Such applications should not use the CORBA event service, but should resort to other means of communication.

### Messaging

All communication in CORBA as described so far is transient. This means that a message is stored by the underlying communication system only as long as both its sender and its receivers are executing. As we discussed in Chap. 2, there are many applications that require persistent communication so that a message is stored until it can be delivered. With persistent communication, it does not matter whether the sender or receiver is executing after the message has been sent; in all cases, it will be stored as long as necessary.

A well-known model for persistent communication is the message-queuing model. CORBA supports this model as an additional **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 additional forms of asynchronous method invocations.

In the **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 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 ORB 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 (which we expressed in CORBA IDL) 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 successfully complete. Transforming the original (synchronous) method invocation into an asynchronous one with call%backs 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, 2001b):

```
void sendcb_add(in int i, in int j);               // Called by the client
void replycb_add(in int ret_val, in int k);        // Called by the client's ORB
```

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 simply 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. Note that these changes do not affect the server-side implementation of the object. Using this example, the callback model is summarized in Fig. 9-7.

As an alternative to callbacks, CORBA provides a **polling model**. In this model, the client is offered a collection of operations to poll its ORB 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.

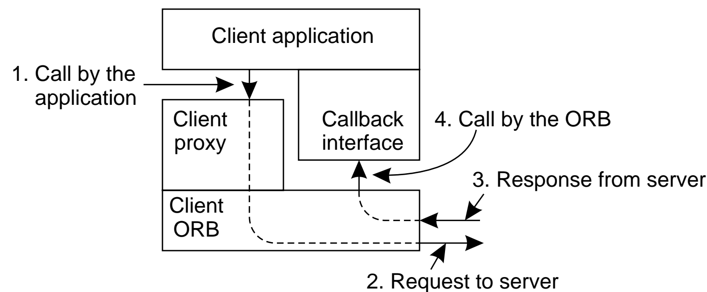


**Figure 9-7.** CORBA's callback model for asynchronous method invocation.

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
```

The most important difference with the callback model is that the method replypoll_add will have to be implemented by the client's ORB. This implementation can be automatically generated by an IDL compiler. The polling model is summarized in Fig. 9-8. Again, notice that the original implementation of the object as it appears at the server's side does not have to be changed.

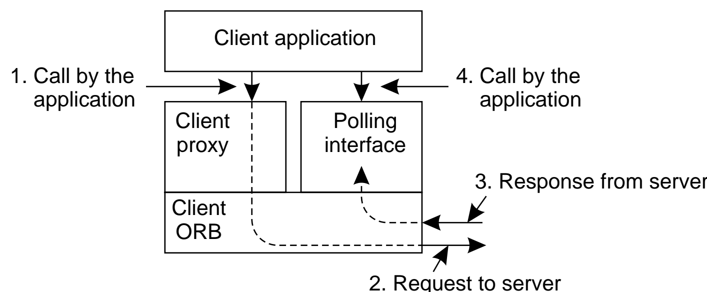


**Figure 9-8.** CORBA's polling model for asynchronous method invocation.

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 delivery can take place. The approach proposed in (OMG, 2001b) is analogous to IBM's message-

queuing system, which we discussed in Sec. 2.4.4 and will not be repeated here.

### Interoperability

The early versions of CORBA left many issues open to actual implementations. The result was that CORBA systems from different manufacturers each had their own way of enabling communication between clients and object servers. In particular, only if a client and object server were using the same ORB was it possible for the client to invoke one of the server's objects.

This lack of interoperability was solved by the introduction of a standard inter-ORB protocol, in combination with a uniform way of referencing objects. Returning to Fig. 9-2, this means that the communication shown between the client and server adheres to a standard protocol, which, in CORBA, is also known as the **General Inter-ORB Protocol** (**GIOP**).

GIOP is actually a framework for a protocol; it assumes that an actual realization is executed on top of an existing transport protocol. That transport protocol should be reliable, connection-oriented, and provide the notion of a byte stream, along with a few other features. Not entirely surprising, TCP satisfies these requirements, but so do other transport protocols. The realization of GIOP running on top of TCP is called the **Internet Inter-ORB Protocol** or simply **IIOP**.

GIOP (and thus IIOP or any other realization of GIOP), recognizes eight different message types, shown in Fig. 9-9. The two most important message types are *Request* and *Reply*, which jointly form part of the actual implementation of a remote method invocation.

A *Request* message contains a complete marshaled invocation request, comprising an object reference, the name of the method that is to be invoked, and all the necessary input parameters. The object reference and method name are part of the header. Each *Request* message also has its own request identifier, which is later used to match its corresponding incoming reply.

A *Reply* message contains the marshaled return values and output parameters associated with the method that was previously invoked. There is no need to explicitly identify the object or method; simply returning the same request identifier as used in the corresponding *Request* message is enough.

As we discuss below, a client can request an implementation repository to provide details on where a specific object can be reached. Such a request is sent by means of a *LocateRequest* message. The implementation repository will respond with a *LocateReply* message, which normally identifies the object's current server to which invocation requests can be sent.

The *CancelRequest* message can be sent by a client to a server when the client wants to cancel a previously sent *Request* or *LocateRequest* message. Canceling a request means that the client is no longer prepared to wait for a response from the server. There may be different reasons why a client wants to cancel the reply to a

| Message type | Originator | Description |
|---|---|---|
| Request | Client | Contains an invocation request |
| Reply | Server | Contains the response to an invocation |
| LocateRequest | Client | Contains a request on the exact location of an object |
| LocateReply | Server | Contains location information on an object |
| CancelRequest | Client | Indicates client no longer expects a reply |
| CloseConnection | Both | Indication that connection will be closed |
| MessageError | Both | Contains information on an error |
| Fragment | Both | Part (fragment) of a larger message |

**Figure 9-9.** GIOP message types.

request, but it is usually caused by a timeout in the client's application. It is important to note that canceling a request does not imply that the associated request will not be carried out. It is up to the client's application to cope with this situation.

In GIOP, a client always sets up a connection to a server. Servers are expected to only accept or reject incoming connection requests, but will not by themselves set up a connection to a client. However, both a client and a server are entitled to close a connection, for which they can send a *CloseConnection* message to the other communicating party.

If a failure occurs, the other party is notified by sending a message of type *MessageError.* Such a message contains the header of the GIOP message that caused the failure. (This approach is similar to ICMP messages in the Internet protocol that are used to return error information when something went wrong. In that case, the header of the IP packet that caused an error is sent as data in the ICMP message.)

Finally, GIOP also allows the various request and reply messages to be fragmented. In this way, invocations requiring that much data be shipped between a client and a server can easily be supported. Fragments are sent as special *Fragment* messages identifying the original message and allowing reassembly of that message at the receiver's side.

### 9.1.3 Processes

CORBA distinguishes two types of processes: clients and servers. An important design goal of CORBA is to make clients as simple as possible. The underlying thought is that it should be easy for application developers to make use of existing services as implemented on the servers.

Servers, on the other hand, were initially left open to a variety of

implementations and were given minimal support in the form of a basic object adapter. Unfortunately, this minimal support also led to portability problems, which by now have been solved by giving a more complete and better specification of what an object adapter should offer. Below, we take a closer look at client-side and server-side software in CORBA.

### Clients

As previously mentioned, CORBA's client-side software is kept to a minimum. The IDL specifications of an object are simply compiled into a proxy that marshals invocation requests into, for example, IIOP *Request* messages, and unmarshals the corresponding *Reply* messages into results that can are handed back to the invoking client.

Proxies in CORBA have no other task than to connect a client application to the underlying ORB. Instead of generating an object-specific proxy, a client can also dynamically invoke objects through the DII.

The consequence of this approach is that if an object requires a specific client-side implementation of its interfaces, it will have to either instruct the developer to use an IDL compiler that generates that software, or provide the client's proxy itself. For example, an object implementation could be accompanied by a set of proxies that implement an object-specific client-side caching strategy. Of course, the latter approach is totally out of line with CORBA's objective of portability and distribution transparency.

Another approach is to forget about object-specific matters and to rely on a client-side ORB that provides the necessary support. For example, instead of providing a cache contained in the client's proxy, an object implementation could assume that caching is handled in a general way by the client's ORB. Again, it should be clear that such an approach has inherent limitations.

What is needed is a mechanism to use the proxies as generated by an IDL compiler in combination with an existing client-side ORB, but nevertheless be able to adapt the client-side software when needed. CORBA's solution to this problem is to use interceptors. As its name suggests, an **interceptor** is a mechanism by which an invocation can be intercepted on its way from client to server, and adapted as necessary before letting it continue. In essence, an interceptor is a piece of code that modifies an invocation request on its way from client to server, and accordingly adapts the associated response. There may be various interceptors added to an ORB. Which one is actually activated depends on the object or server that is referenced in an invocation request.

An interceptor in CORBA can be placed at either one of two different logical levels, as shown in Fig. 9-10. A **request-level interceptor** is logically placed between a client's proxy and the ORB. Before an invocation request is passed to the ORB, it first passes through an interceptor, which may modify the request. On the server side, a request-level interceptor is placed between the ORB and the

object adapter.

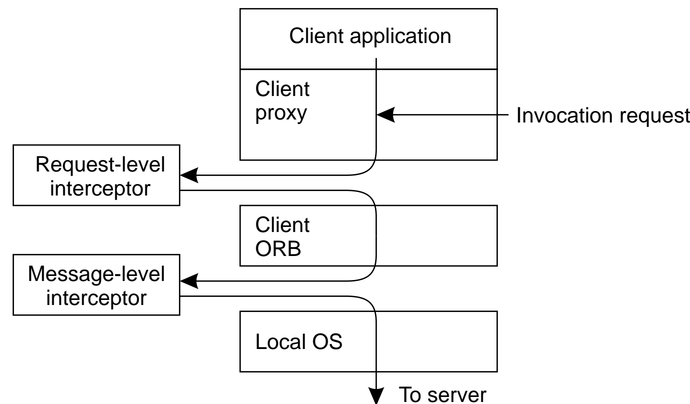


**Figure 9-10.** Logical placement of interceptors in CORBA.

In contrast, **message-level interceptors** are placed between an ORB and the underlying network. A message-level interceptor knows nothing about the message that is to be sent; it only has GIOP messages at its disposal that it may wish to modify. A typical example of a message-level interceptor is one that implements fragmentation at the sending side, and reassembly of the original GIOP message at the receiver side, for example, using the special *Fragment* messages.

Interceptors are seen only by an ORB, that is, it is the ORB's responsibility to invoke an interceptor. Clients and servers hardly ever see interceptors, except at the time a client binds to a server. Note that an ORB may make use of both types of interceptors at the same time. An overview of the use of interceptors in CORBA can be found in (Narasimhan et al., 1999), whereas the exact specifications are given in (OMG, 2001b).

Although the idea of an interceptor seems attractive at first sight, it can also be argued that it is a mechanism for breaking into an implementation to fix things that are apparently missing. For example, if CORBA had provided mechanisms to support developing and using object-specific proxies, there would be less need for interceptors. However, what is really needed are ORBs that can be easily extended. Interceptors provide a general mechanism to support extensibility, but the question is whether the two types of interceptors as provided in CORBA are really the ones we need. More on interceptors from a software architecture point of view can be found in (Schmidt et al., 2000).

### Portable Object Adapter

In Chap. 3, we described in detail the notion of an object adapter, which is a mechanism that implements a specific activation policy for a group of objects. For example, one adapter may implement a method invocation using a separate thread for each invocation, whereas another one uses only a single thread for all the

objects it manages.

In general, an object adapter does more than just call a method on an object. As its name suggests, an object adapter is responsible for providing a consistent image of what an object is; it adapts a program such that clients can see that program as an object. Adapters are also called **wrappers**.

In CORBA, the **Portable Object Adapter** (**POA**) is a component that is responsible for making server-side code appear as CORBA objects to clients. The POA has been defined in such a way that server-side code can be written independently of a specific ORB.

To support portability across different ORBs, CORBA assumes that object implementations are partly provided by what are called servants. A **servant** is that part of an object that implements the methods that clients can invoke. Servants are necessarily programming-language dependent. For example, implementing a servant in C++ or Java would typically be done by providing an instance of a class. On the other hand, a servant written in C or any other procedural language typically consists of a collection of functions operating on data structures that represent the state of an object.

How does a POA use a servant to build the image of a CORBA object? In the first place, each POA offers the following operation:

```
ObjectId activate_object(in Servant p_servant);
```

that we have specified in CORBA IDL. This operation takes a pointer to a servant as input parameter and returns a CORBA object identifier as a result. There is no universal definition of type *Servant*; instead, it is mapped to a language-dependent data type. For example, in C++, *Servant* is mapped to a pointer to the predefined *ServantBase* class. This class contains a number of method definitions that each C++ servant should implement.

The object identifier returned by the operation activate_object is generated by the POA. It is used as an index into the POA's **Active Object Map**, which points to servants as shown in Fig. 9-11(a). In this case, the POA implements a separate servant for each object it supports. To make matters more concrete, assume an application developer has written a subclass of *ServantBase*, called *My_Servant*. A C++ object that is created as an instance of the *My_Servant* class can be turned into a CORBA object as shown in Fig. 9-12.

In the code in Fig. 9-12, we first declare a reference to a C++ object. To create a CORBA identifier, we make use of *CORBA::ObjectId_var*, which is predefined C++ data type for all standard C++ implementations of CORBA. After these declarations, *my_object* can be instantiated as a true C++ object. In CORBA terminology, this C++ object corresponds to a servant. Turning the C++ object into a CORBA object proceeds by registering it at the POA (which we assume to be available through the variable *poa*). Registration returns a CORBA identifier.

It is important to note that if a second object of type *My_Servant* were created, the registration of that C++ object at the POA would lead to an almost

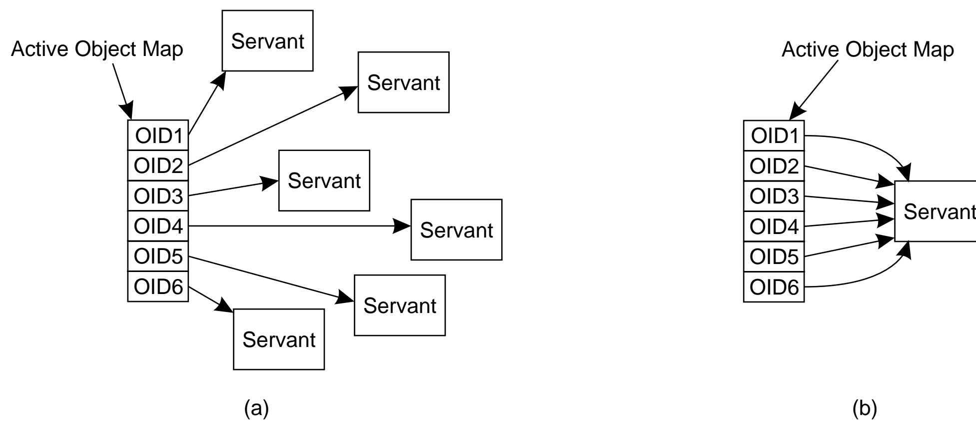


**Figure 9-11.** Mapping of CORBA object identifiers to servants. (a) The POA supports multiple servants. (b) The POA supports a single servant.

```
My_Servant *my_object;              // Declare a reference to a C++ object
CORBA::ObjectId_var oid;            // Declare a CORBA identifier
my_object = new My_Servant;         // Create a new C++ object

// Now register the C++ object as a CORBA object
oid = poa→activate_object(my_object);
```

**Figure 9-12.** Changing a C++ object into a CORBA object.

identical servant, although operating on a different state. In those cases that a POA has to support objects that are derived from the same class definition, it is more efficient to register only a single servant, and distinguish the various objects only by their state. This principle is shown in Fig. 9-11(b), where each object identifier refers to the same servant. In this case, whenever an object is invoked, the object's identifier will be passed (implicitly) to the servant so that it can operate on only that data that are uniquely associated with the identified object.

This example also illustrates another important issue: a CORBA object identifier is uniquely associated with a POA. Whenever a servant is registered with a POA, the POA returns an object identifier for that servant. An alternative that we have not shown is that an application developer first generates an identifier and passes that to a POA. However, in both cases, this identifier will be encapsulated into a larger data structure that functions as a *systemwide* object reference, and that also identifies the POA and the server where that POA is located.

Whether or not a POA supports one servant per object, or one servant for all its objects, is just one kind of policy that can be associated with a POA. There are

many other policies that can be supported. For example, a POA can support transient as well as persistent objects. Likewise, there are different policies with respect to the use of threads. We omit details on these policies, which are further described in (Henning and Vinoski, 1999)

### Agents

To facilitate agent-based applications, CORBA adopts a model in which agents from different kinds of systems can cooperate. Instead of specifying its own model of an agent, CORBA prescribes standard interfaces that an agent system should implement. This approach has the potential advantage that different types of agents can be used in a single distributed application. For example, in CORBA it is possible to let a Java applet create a Tcl agent on a D'Agents platform. (We described the D'Agents system in Chap. 3.)

In CORBA, agents are always defined with respect to an agent system. An **agent system** is defined as a platform that allows the creation, execution, transfer, and termination of agents. Each agent system has an associated profile describing exactly what the agent system looks like. For example, there will be a profile for a D'Agents agent system, prescribing its type ("D'Agents"), the supported language (e.g., Tcl), and the way agents are serialized when moving between systems.

An agent is always located at a particular **place** in an agent system. A place corresponds to a server where an agent resides. There can be multiple places in an agent system. In other words, CORBA assumes that a single agent system may consist of multiple processes, each hosting one or more agents. This organization has been adopted in order to group several agent hosts into a single administrative domain, and be able to refer to that group as a whole, namely as an agent system.

Agent systems, in turn, can be grouped into a **region**, where a region represents the administrative domain in which an agent system resides. For example, a university department could have several agent systems, each of a different type. Each agent system could be distributed across multiple hosts, or places, where each host may run several agents. This model is shown in Fig. 9-13.

An agent in CORBA is assumed to be constructed from a collection of classes, or at the very least from a file containing the necessary program text. In either case, it should be possible to name an agent's implementation, and pass that name to an agent system to allow the creation or transfer of an agent.

Each agent system in CORBA must implement a number of standard operations. For example, an agent system must offer operations to create or terminate an agent, to receive an agent for execution, to list its current set of agents, to list the places where an agent may reside, and operations to suspend and resume an agent. Note that the implementation of these operations depends entirely on the actual agent system. However, CORBA does require that each agent system adheres to the overall model. This means that if an existing agent system does not

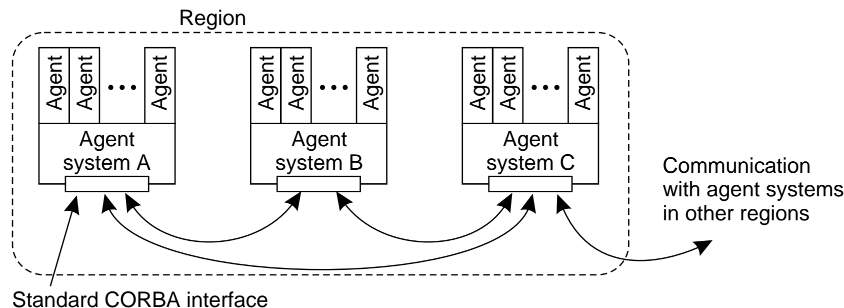


**Figure 9-13.** CORBA's overall model of agents, agent systems, and regions.

originally support a notion of place, it should nevertheless provide one.

Each region will have an associated **finder**, which allows it to find the location of the agents, places, and agent systems within that region. The finder offers a number of standard operations for registering and unregistering agents, places, and agent systems. In essence, a finder is nothing but a simple directory service for a region.

Further details on mobile agents in CORBA can be found in (OMG, 2000e).

### 9.1.4 Naming

CORBA supports different types of names. The most basic type of naming appears in the form of object references and character-based names as supported by the CORBA naming service. In addition, there are a number of advanced naming facilities, whereby objects can be found based on associated properties. In the following, we take a closer look at basic naming in CORBA, concentrating on object references. For advanced naming facilities as available through CORBA's trading service, the reader is referred to (OMG, 2001a; OMG, 2000b).

#### 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 ORB.

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 ORB. 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 ORB 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 ORB 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 IDL specifications of an object.

One of the problems that early versions of CORBA had was that each ORB 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 executing on top of the same ORB. Otherwise, the marshaled version of the reference held by process *A* would be meaningless to the ORB underlying process *B*.

Current CORBA systems all support the same language-independent representation of an object reference, which is called **Interoperable Object Reference** or **IOR**. Whether or not an ORB uses IORs internally is not important. However, when passing an object reference between two different ORBs, 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. 9-14, along with specific information for IIOP, which we next explain in detail.

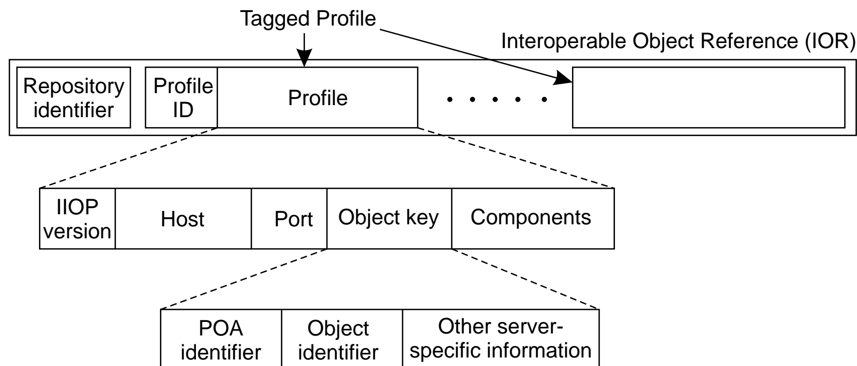


**Figure 9-14.** The organization of an IOR with specific information for IIOP.

Each IOR starts with a repository identifier. As we explained, this identifier is assigned to an interface when that interface is stored in the interface repository. It is used to retrieve information on an interface during 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.

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. Details on the profile used for IIOP are also shown in Fig. 9-14.

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, a POA-generated object identifier will generally be part of such an object key. Also, this key will identify the POA.

Finally, there is a *Components* field that optionally contains additional information needed for properly invoking the referenced object. For example, the components 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. We return to these matters below.

Now that we have explained the details of an object reference, it is not difficult to see how a client binds to an object to subsequently invoke a method. Recall from Chap. 2 that binding a client to an object is the process by which a connection is set up to an object so that the client can invoke that object's methods. Binding is possible only if the client has a reference to that object.

Anticipating our discussion on the CORBA naming service, assume a client has requested the naming service to resolve a human-readable name. The naming service will return a language-dependent implementation of the IOR as stored in the naming service. Such an implementation can be returned only if the client's ORB has carried out the complete binding process. It does so as follows.

The client ORB takes the IOR returned to it by the naming service, and by inspecting the repository ID contained in the IOR, it can place a proxy at the client and return a pointer $p$ to that proxy. Internally, the ORB will store the fact that $p$ is associated with the object's IOR.

Different ORBs have different implementations, but one typical scenario is as follows. Before passing $p$ to the client, the client's ORB inspects the tagged profiles contained in the IOR. Assume that the object can be invoked using IIOP. In that case, the client ORB may set up a TCP connection with the object's server, using the host address and port number found in the IOR. At that point, it can pass

*p* to the client.

Whenever the client invokes one of the object's methods, the client ORB marshals the invocation request into an IIOP *Request* message. This message contains the server-specific object key that was also stored in the IOR. The message is sent through the TCP connection to the server, where it can subsequently be passed to the proper POA associated with the object key. The POA will then forward the request to the proper servant where it is first unmarshaled and transformed into an actual method call.

In this scenario, an IOR refers directly to an object, leading to what is known as **direct binding**. An alternative to direct binding is **indirect binding** by which a binding request is first sent to an implementation repository. The implementation repository is just another process, identified in the object's IOR. It acts as a registry by which the referenced object can be located and activated before sending invocation requests to it. In practice, indirect binding is used primarily for persistent objects, that is, objects controlled by a POA that follows the persistent lifespan policy.

When a client ORB uses an IOR based on indirect binding, it simply starts binding to the implementation repository. The repository will notice that the request is actually intended for another server, and will look into its tables to see whether the server is already running, and if so, where it can be located. If the server is not yet running, the implementation repository can start it, although this depends on whether or not automatic start-up is supported.

When the client invokes the referenced object for the first time, the invocation request is sent to the implementation repository, which responds by giving the details where the object's server can actually be reached, as shown in Fig. 9-15. From there on, invocation requests are forwarded to the proper server.

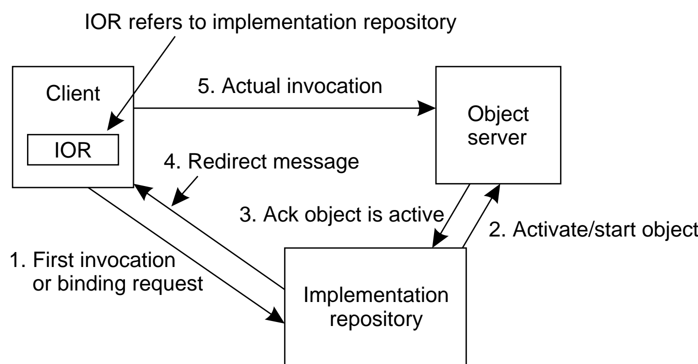


**Figure 9-15.** Indirect binding in CORBA.

**The CORBA Naming Service**

Like any other distributed system, CORBA offers a naming service that allows clients to look up object references using a character-based name. Formally, a name in CORBA is a sequence of name components, each taking the form of a *(id, kind)*-pair, where *id* and *kind* are both strings. Normally, *id* is used to name an object by means of a character string such as "steen" or "elke." The *kind* attribute is a simple indication of the named object, similar to using extensions in file names. For example, the object named "steen" could have kind "dir" telling that it is a directory object.

There is no way to represent the equivalent of a path name as a single string. In other words, CORBA does not define a separator between name components. Instead, names are to be passed explicitly as a sequence of name components. The representation of a sequence is language dependent, but remains opaque to a client that calls the naming service.

There are no restrictions with respect to the structure of a naming graph. Each node in a naming graph is treated as an object. A **naming context** is an object that stores a table mapping name components to object references. A naming context is thus the same as what we called a directory node in Chap. 4. Note that an object reference in a naming context may refer to another naming context.

A naming graph does not have a root context. However, each ORB is required to provide an initial naming context, which effectively operates as the root in a naming graph. Names are always resolved with respect to a given naming context. In other words, if a client wants to resolve a name, it is necessary that it does so by invoking the resolve method on a specific naming context. If name resolution succeeds, it always returns either a reference to a naming context or a reference to a named object. In this sense, name resolution proceeds exactly as explained in Chap. 4; for that reason we shall not repeat it here.

### 9.1.5 Synchronization

The two most important services that facilitate synchronization in CORBA are its concurrency control service and its transaction service. The two services collaborate to implement distributed and nested transactions using two-phase locking.

The model underlying transactions in CORBA is as follows. A transaction is initiated by a client and consists of a series of invocations on objects. When such an object is invoked for the first time, it automatically becomes part of the transaction. As a consequence, the object's server is notified that it is now participating in a transaction. This information is implicitly passed to the server when invoking the object.

There are essentially two types of objects that can be part of a transaction. A **recoverable object** is an object that is executed by an object server capable of

participating in a two-phase commit protocol. In particular, the server for such an object can support the abort of a transaction by rolling back all the changes that were made as the result of invoking one of its recoverable objects. However, it is also possible to invoke objects as part of a transaction that cannot be rolled back to a state before the transaction started. In particular, these **transactional objects** are executed by servers that do not participate in a transaction's two-phase commit protocol. Transactional objects are typically read-only objects.

Thus it is seen that CORBA transactions are similar to the distributed transactions and their protocols as we discussed in Chaps. 5 and 7.

Likewise, the locking services provided by the concurrency control service are very much what one would expect. In practice, the service is implemented using a central lock manager; it does not use distributed locking techniques. The service distinguishes read from write locks, and is also capable of supporting locks at different granularities as is often needed in databases. For example, it makes sense to distinguish locking an entire table from locking only a single record. See (Gray and Reuter, 1993; or Garcia-Molina et al. 2000) for further information on granular locking.

### 9.1.6 Caching and Replication

CORBA offers no support for generic caching and replication. Only the replication of objects for fault tolerance as we describe in the next section is included in version 3 of CORBA. The lack of support for generic replication implies that application developers will have to resort to an ad-hoc approach when replication is needed. In most cases, such approaches are based on using interceptors.

Let us consider one example of how replication for performance can be incorporated into CORBA. This objective is met by CASCADE. In the CASCADE system, the goal is to provide a generic, scalable mechanism that allows any kind of CORBA object to be cached (Chockler et al., 2000). CASCADE offers a caching service that is implemented as a potentially large collection of object servers each referred to as a **Domain Caching Server** (**DCS**). Each DCS is an object server running on a CORBA ORB. The collection of DCSs may be spread across a wide-area network such as the Internet.

Cached copies of the same object are organized into a hierarchy. It is assumed that a single client, such as the owner of an object, can register that object with a local DCS. That DCS becomes the root of the hierarchy. Other clients can request their local DCS to cache a copy of the object, for which that DCS will first join the current hierarchy of DCSs that already cache the object.

CASCADE supports the client-centric consistency models we discussed in Chap. 6. In addition, it supports total ordering by which all updates are guaranteed to be performed in the same order everywhere. Each object may have its own associated consistency model; there is no systemwide policy for maintaining the consistency of cached objects. As we argued in Chap. 6, when replicating for

performance it is important that different consistency models can be simultaneously supported, because the applicability of a model strongly depends on the usage and access patterns of an object. CASCADE meets this requirement.

As a CORBA service, CASCADE relies heavily on interceptors. To a client, CASCADE is virtually invisible; all issues dealing with consistency are hidden behind the interfaces that an object normally provides. The only time a client has explicit access to CASCADE is when it requests its local DCS to start caching a specific object. Whenever such an object is invoked, the invocation is intercepted at the client ORB and subsequently forwarded to that DCS.

Depending on the consistency model for the referenced object, additional information is added to the invocation request before sending it off to the DCS. For example, when a client requires read-your-writes consistency, it is necessary to tell what the last write operation was that the client has seen.

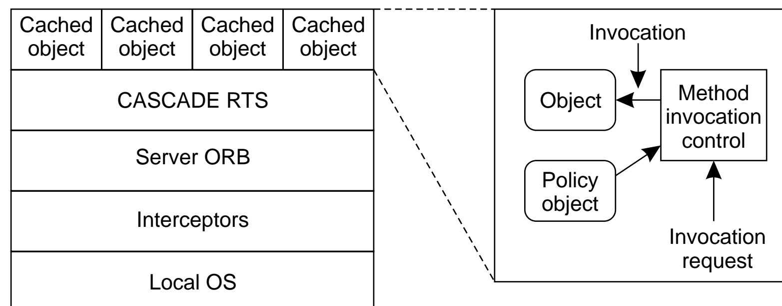


**Figure 9-16.** The (simplified) organization of a DCS.

The general organization of a DCS is shown in Fig. 9-16. A DCS manages a number of copies of an object. Such a copy consists of the state and the implementations of the operations on that state. In CORBA terminology, a DCS has the same servant as the original copy of the object. The interceptor underlying the DCS intercepts an incoming invocation request and extracts the information that, for example, the client interceptor has added. The request is then forwarded to a method invocation control module uniquely associated with the referenced object. As shown in Fig. 9-16, each cached object has its own policy object that contains specific information to control method invocations. The additional information that was extracted by the interceptor is passed separately to this policy object.

Although CASCADE offers more or less transparent caching of object servers, it is seen that the more effort is needed to implement such a caching service for CORBA. Although interceptors allow invocations to be changed as necessary to implement such a service, no other support is provided by CORBA.

### 9.1.7 Fault Tolerance

CORBA systems have long lacked real support for fault tolerance. In most cases, a failure was simply reported to the client, and the system undertook no further action. For example, if a referenced object could not be reached because its associated server was (temporarily) unavailable, a client was left on its own. In CORBA version 3, fault tolerance is explicitly addressed. The specification of a fault-tolerant CORBA can be found in (OMG, 2000d).

#### Object Groups

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. 6. There are various other properties associated with object groups, the details of which can be found in (OMG, 2000d).

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.

Whenever a client passes an IOGR to its ORB, that ORB attempts to bind to one of the referenced replicas. In the case of IIOP, the ORB 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. 9-17, by means of the separate tags *TAG_PRIMARY* and *TAG_BACKUP*, respectively.

If binding to one of the replicas fails, the client ORB 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.

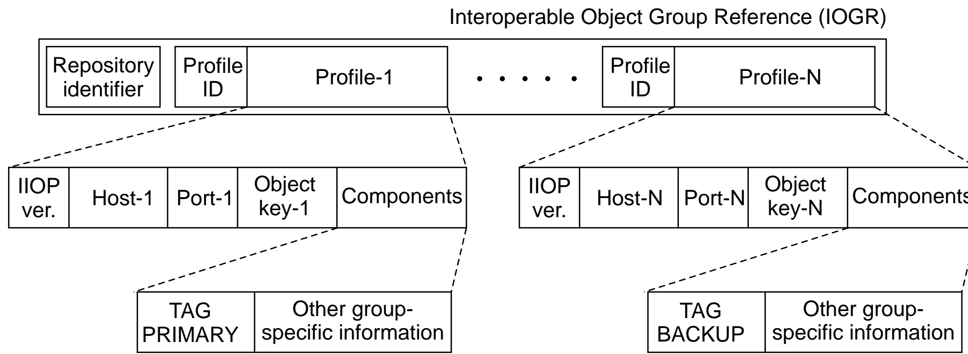


**Figure 9-17.** A possible organization of an IOGR for an object group having a primary and backups.

## 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. 9-18. This architecture is derived from the Eternal system (Moser et al., 1998; Narasimhan et al., 2000), which provides a fault tolerance infrastructure constructed on top of the Totem reliable group communication system (Moser et al., 1996).

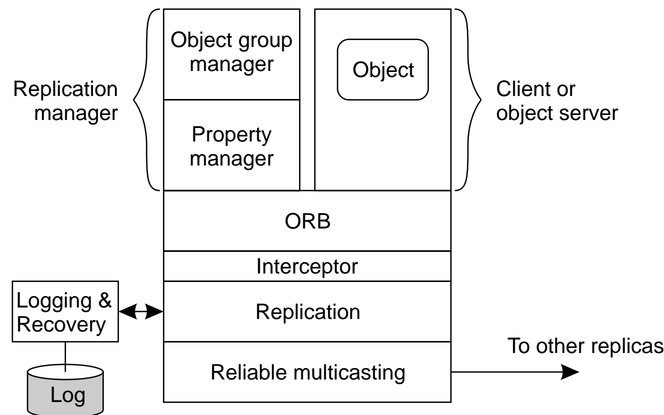


**Figure 9-18.** An example architecture of a fault-tolerant CORBA system.

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 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.

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 ORB. However, in the case of passive replication, an invocation request is passed only to the ORB 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.

### 9.1.8 Security

CORBA security has a long history. The initial versions of the CORBA specifications hardly touched upon the subject for the simple reason that several attempts to specify a security service failed. In CORBA version 2.4, the security services take over 400 pages to specify and to make clear how security should fit into CORBA systems. Let us now take a closer look at CORBA security. An overview of these matters can be found in (Blakley, 2000), which was written by one of the authors of the original CORBA security specifications.

An important specification issue for security in CORBA is that services should provide an appropriate collection of mechanisms that can be used to implement a variety of security policies. What complicates matters is that these services should be offered at different points in time and space. For example, if a client wants to securely invoke an object, we need to decide *when* security mechanisms are to be used (e.g., at binding time, invocation time, or both), and *where* these mechanisms should be used (e.g., at the level of an application, inside an ORB, or during message transfer).

The heart of CORBA security is formed by the support for secure object invocations. The underlying idea is that application-level objects should be unaware of the various security services that are used. However, if a client has specific security requirements, the client should be allowed to specify those requirements so that they can be taken into account when an object is invoked. An analogous

situation occurs for an object that is to be invoked. This approach leads to the general organization shown in Fig. 9-19.

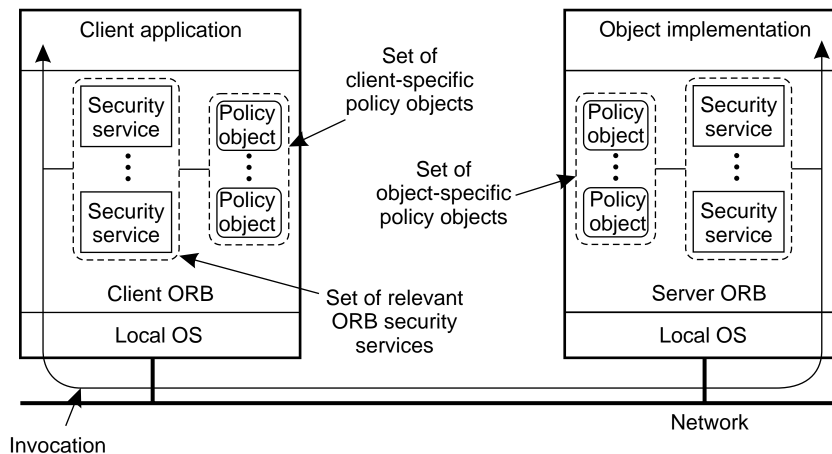


**Figure 9-19.** The general organization for secure object invocation in CORBA.

When a client binds to an object in order to invoke that object, the client ORB determines which security services are needed at the client side to support secure invocations. The selection of services is determined by the security policies associated with the administrative domain in which the client is executing, but also by the specific policies of that client.

Security policies are specified by means of **policy objects** that are associated with the client. In practice, the domain in which a client is executing will make its security policies available to the ORB of that client by means of a specific set of policy objects. Default policies are automatically associated to clients. Example policy objects include those that specify the type of message protection that is required, and objects that have a list of trusted parties. Other examples and details can be found in (Blakley, 2000).

A similar structure is followed at the server side. Again, the administrative domain in which the invoked object is executing will require that a specific set of security services is used. Likewise, the invoked object will have its own set of associated policy objects in which object-specific information is stored.

Various approaches can be followed to implement security in CORBA. In particular, to make ORBs as general as possible, it is desirable to specify different security services by means of standard interfaces that hide the implementation of those services. Services that can be specified in this way are called **replaceable** in CORBA.

Replaceable security services are assumed to be implemented in combination with two different interceptors, as shown in Fig. 9-20. The **access control interceptor** is a request-level interceptor that checks the access rights associated with

an invocation. In addition, there is also a message-level **secure invocation interceptor** that takes care of implementing the message protection. In other words, this interceptor is able to encrypt requests and responses for integrity and confidentiality.

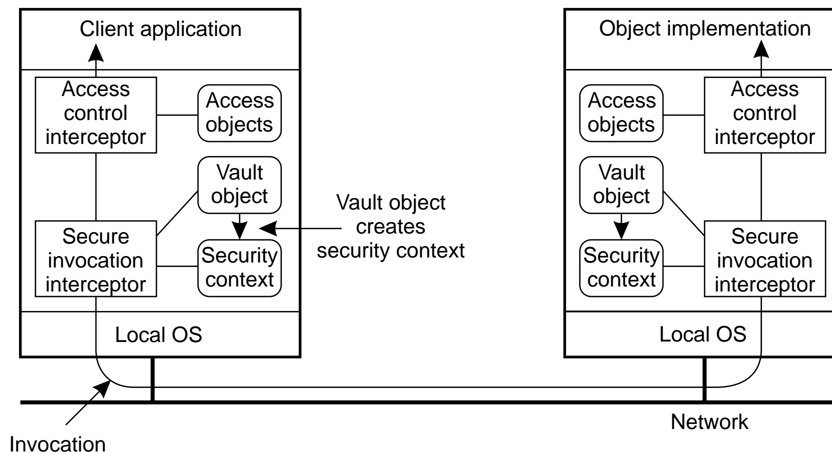


**Figure 9-20.** The role of security interceptors in CORBA.

The secure invocation interceptor plays a crucial role, as it is responsible for setting up a **security context** for the client that will allow secure invocations of the target object. This security context, represented by means of a security context object, contains all the necessary information and methods for securely invoking the target object. For example, it describes which mechanism to use, offers methods to encrypt and decrypt messages, stores references to credentials, and so on.

The object's server will also have to create its own security context object. The client interceptor will therefore generally first send a message to the object server containing the necessary information to authenticate the client and to let the server create a security context for subsequent invocations. Note that the secure invocation interceptor at the object's server will check the object-specific policy objects to see whether and how all security requirements can be met. The response returned to the client may include additional information that will allow the client to authenticate the server.

After this initial exchange of messages, the client will be bound to the target object, and the two will have established what is generally referred to as a **security association**. From there on, secure invocations can take place by which the secure invocation interceptors protect the request and response messages following the policy agreed upon between the client and object server.

A crucial role in setting up a security association is played by a separate object with a standardized interface, called the **vault object**. The vault object is called by the secure invocation interceptors to create a security context object.

The interceptor first reads the policy information from the client's associated policy objects, and passes this information to the vault object. Clearly, the vault object must be implemented as part of the ORB in a tamper-proof way and belongs to the trusted computing base of any CORBA system.