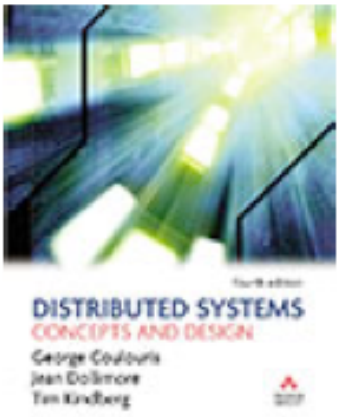


Archive material from *Edition 4* of Distributed Systems: Concepts and Design

CHAPTER CDK4–20

Originally published as pp. 827-858 of Coulouris, Dollimore and Kindberg, Distributed Systems, Edition 4, 2005.

CORBA CASE STUDY

CORBA is a middeware design that allows application programs to communicate with one another irrespective of their programming languages, their hardware and software platforms, the networks they communicate over and their implementors.

Applications are built from CORBA objects, which implement interfaces defined in CORBA's interface definition language, IDL. Clients access the methods in the IDL interfaces of CORBA objects by means of RMI. The middleware component that supports RMI is called the Object Request Broker or ORB.

The specification of CORBA has been sponsored by members of the Object Management Group (OMG). Many different ORBs have been implemented from the specification, supporting a variety of programming languages including Java and C++.

CORBA services provide generic facilities that may be of use in a wide variety of applications. They include the Naming Service, the Event and Notification Services, the Security Service, the Transaction and Concurrency Services and the Trading Service.

20.1 Introduction

The OMG (Object Management Group) was formed in 1989 with a view to encouraging the adoption of distributed object systems in order to gain the benefits of object-oriented programming for software development and to make use of distributed systems, which were becoming widespread. To achieve its aims, the OMG advocated the use of open systems based on standard object-oriented interfaces. These systems would be built from heterogeneous hardware, computer networks, operating systems and programming languages.

An important motivation was to allow distributed objects to be implemented in any programming language and to be able to communicate with one another. They therefore designed an interface language that was independent of any specific implementation language.

They introduced a metaphor, the *object request broker* (or ORB), whose role is to help a client to invoke a method on an object. This role involves locating the object, activating the object if necessary and then communicating the client's request to the object, which carries it out and replies.

In 1991, a specification for an object request broker architecture known as CORBA (Common Object Request Broker Architecture) was agreed by a group of companies. This was followed in 1996 by the CORBA 2.0 specification, which defined standards enabling implementations made by different developers to communicate with one another. These standards are called the General Inter-ORB protocol or GIOP. It is intended that GIOP can be implemented over any transport layer with connections. The implementation of GIOP for the Internet uses the TCP protocol and is called the Internet Inter-ORB Protocol or IIOP [OMG 2004a]. CORBA 3 first appeared in late 1999 and a component model has been added recently.

The main components of CORBA's language-independent RMI framework are the following:

- An interface definition language known as IDL, which is illustrated early in Section 20.2 and described more fully in Section 20.2.3.
- An architecture, which is discussed in Section 20.2.2.
- The GIOP defines an external data representation, called CDR, which is described in Section 4.3. It also defines specific formats for the messages in a request-reply protocol. In addition to request and reply messages, it specifies messages for enquiring about the location of an object, for cancelling requests and for reporting errors.
- The IIOP, an implementation of GIOP defines a standard form for remote object references, which is described in Section 20.2.4.

The CORBA architecture also allows for CORBA services – a set of generic services that can be useful for distributed applications. These are introduced in Section 20.3 which includes a more detailed discussion of the Naming Service, the Event Service, the Notification Service and the Security Service. For an interesting collection of articles on CORBA, see the *CACM* special issue [Seetharamanan 1998].

Before discussing the above components of CORBA, we introduce CORBA RMI from a programmer's point of view.

20.2 CORBA RMI

Programming in a multi-language RMI system such as CORBA RMI requires more of the programmer than programming in a single-language RMI system such as Java RMI. The following new concepts need to be learned:

- the object model offered by CORBA;
- the interface definition language and its mapping onto the implementation language.

Other aspects of CORBA programming are similar to those discussed in Chapter 5. In particular, the programmer defines remote interfaces for the remote objects and then uses an interface compiler to produce the corresponding proxies and skeletons. But in CORBA, proxies are generated in the client language and skeletons in the server language. We will use the shared whiteboard example introduced in Section 5.5 to illustrate how to write an IDL specification and to build server and client programs.

CORBA's object model ◊ The CORBA object model is similar to the one described in Section 5.2, but clients are not necessarily objects – a client can be any program that sends request messages to remote objects and receives replies. The term *CORBA object* is used to refer to remote objects. Thus, a CORBA object implements an IDL interface, has a remote object reference and is able to respond to invocations of methods in its IDL interface. A CORBA object can be implemented by a language that is not object-oriented, for example without the concept of class. Since implementation languages will have different notions of class or even none at all, the class concept does not exist in CORBA. Therefore classes cannot be defined in CORBA IDL, which means that instances of classes cannot be passed as arguments. However, data structures of various types and arbitrary complexity can be passed as arguments.

CORBA IDL ◊ A CORBA IDL interface specifies a name and a set of methods that clients can request. Figure 20.1 shows two interfaces named *Shape* (line 3) and *ShapeList* (line 5), which are IDL versions of the interfaces defined in Figure 5.12. These are preceded by definitions of two *structs*, which are used as parameter types in defining the methods. Note in particular that *GraphicalObject* is defined as a *struct*, whereas it was a class in the Java RMI example. A component whose type is a *struct* has a set of fields containing values of various types like the instance variables of an object, but it has no methods. There is more about IDL in Section 20.2.3.

Parameters and results in CORBA IDL: Each parameter is marked as being for input or output or both, using the keywords *in*, *out* or *inout*. Figure 5.2 illustrates a simple example of the use of those keywords. In Figure 20.1, line 7, the parameter of *newShape* is an *in* parameter to indicate that the argument should be passed from client to server in the request message. The return value provides an additional *out* parameter – it can be indicated as *void* if there is no *out* parameter.

Figure 20.1 IDL interfaces *Shape* and *ShapeList*

```
struct Rectangle{                                                    1
    long width;
    long height;
    long x;
    long y;
} ;
struct GraphicalObject {                                             2
    string type;
    Rectangle enclosing;
    boolean isFilled;
};
interface Shape {                                                    3
    long getVersion() ;
    GraphicalObject getAllState() ;    // returns state of the GraphicalObject
};
typedef sequence <Shape, 100> All;                                   4
interface ShapeList {                                                5
    exception FullException{ };                                      6
    Shape newShape(in GraphicalObject g) raises (FullException);     7
    All allShapes();              // returns sequence of remote object references   8
    long getVersion() ;
};
```

The parameters may be any one of the primitive types such as *long* and *boolean* or one of the constructed types such as *struct* or *array*. Primitive and structured types are described in more detail in Section 20.2.3. Our example shows the definitions of two *structs* in lines 1 and 2. Sequences and arrays are defined in *typedefs*, as shown in line 4, which shows a sequence of elements of type *Shape* of length 100. The semantics of parameter passing are as follows:

Passing CORBA objects: Any parameter whose type is specified by the name of an IDL interface, such as the return value *Shape* in line 7, is a reference to a CORBA object and the value of a remote object reference is passed.

Passing CORBA primitive and constructed types: Arguments of primitive and constructed types are copied and passed by value. On arrival, a new value is created in the recipient's process. For example, the *struct GraphicalObject* passed as argument (in line 7) produces a new copy of this *struct* at the server.

These two forms of parameter passing are combined in the method *allShapes* (in line 8), whose return type is an array of type *Shape* – that is, an array of remote object references. The return value is a copy of the array in which each of the elements is a remote object reference.

Type *Object*: *Object* is the name of a type whose values are remote object references. It is effectively a common supertype of all of IDL interface types such as *Shape* and *ShapeList*.

Exceptions in CORBA IDL: CORBA IDL allows exceptions to be defined in interfaces and thrown by their methods. To illustrate this point, we have defined our list of shapes in the server as a sequence of a fixed length (line 4) and have defined *FullException* (line 6), which is thrown by the method *newShape* (line 7) if the client attempts to add a shape when the sequence is full.

Invocation semantics: Remote invocation in CORBA has *at-most-once* call semantics as the default. However, IDL may specify that the invocation of a particular method has *maybe* semantics by using the *oneway* keyword. The client does not block on *oneway* requests, which can be used only for methods without results. For an example of a *oneway* request, see the example on callbacks at the end of Section 20.2.1.

The CORBA Naming service ◊ The CORBA Naming Service is discussed in Section 20.3.1. It is a binder that provides operations including *rebind* for servers to register the remote object references of CORBA objects by name and *resolve* for clients to look them up by name. The names are structured in a hierarchic fashion, and each name in a path is inside a structure called a *NameComponent*. This makes access in a simple example seem rather complex.

CORBA pseudo objects ◊ Implementations of CORBA provide interfaces to the functionality of the ORB that programmers need to use. In particular, they include interfaces to two of the components shown in Figure 20.6: the *ORB core* and the *Object Adaptor*. The roles of these two components are explained in Section 20.2.2

The objects representing these components are called *pseudo-objects* because they cannot be used like CORBA objects; for example, they cannot be passed as arguments in RMIs. They have IDL interfaces and are implemented as libraries. Those relevant to our simple example (which uses Java 2 version 1.4) are:

- The Object Adaptor, which has been portable since CORBA 2.2, is known as the Portable Object Adaptor (POA). Its interface includes: one method for activating a *POAmanager* and another method *servant_to_reference* for registering a CORBA object.
- The ORB, whose interface includes: the method *init*, which must be called to initialize the ORB; the method *resolve_initial_references*, which is used to find services such as the Naming Service and the root POA; and other methods, which enable conversions between remote object references and strings.

20.2.1 CORBA client and server example

This section outlines the steps necessary to produce client and server programs that use the IDL *Shape* and *ShapeList* interfaces shown in Figure 20.1. This is followed by a discussion of callbacks in CORBA. We use Java as the client and server languages, but the approach is similar for other languages. The interface compiler *idlj* can be applied to the CORBA interfaces to generate the following items:

Figure 20.2 Java interfaces generated by *idlj* from CORBA interface *ShapeList*.

```
public interface ShapeListOperations {
    Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
    Shape[] allShapes();
    int getVersion();
}
```

```
public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
        org.omg.CORBA.portable.IDLEntity { }
```

- The equivalent Java interfaces – two per IDL interface. The name of the first Java interface ends in *Operations* – this interface just defines the operations in the IDL interface. The Java second interface has the same name as the IDL interface and implements the operations in the first interface as well as those in an interface suitable for a CORBA object. For example, the IDL interface *ShapeList* results in two Java interfaces *ShapeListOperations* and *ShapeList* as shown in Figure 20.2.
- The server skeletons for each *idl* interface. The names of skeleton classes end in *POA*, for example *ShapeListPOA*.
- The proxy classes or client stubs, one for each IDL interface. The names of these classes end in *Stub*, for example *_ShapeListStub*.
- A Java class to correspond to each of the *structs* defined with the IDL interfaces. In our example, classes *Rectangle* and *GraphicalObject* are generated. Each of these classes contains a declaration of one instance variable for each field in the corresponding *struct* and a pair of constructors, but no other methods.
- Classes called helpers and holders, one for each of the types defined in the IDL interface. A helper class contains the *narrow* method, which is used to cast down from a given object reference to the class to which it belongs, which is lower down the class hierarchy. For example, the *narrow* method in *ShapeHelper* casts down to class *Shape*. The holder classes deal with *out* and *inout* arguments, which cannot be mapped directly onto Java. See Exercise 20.9 for an example of the use of holders.

Server program ◊ The server program should contain implementations of one or more IDL interfaces. For a server written in an object-oriented language such as Java or C++, these implementations are implemented as servant classes. CORBA objects are instances of servant classes.

When a server creates an instance of a servant class, it must register it with the POA, which makes the instance into a CORBA object and gives it a remote object reference. Unless this is done, the CORBA object will not be able to receive remote invocations. Readers who studied Chapter 5 carefully may realize that registering the object with the POA causes it to be recorded in the CORBA equivalent of the remote object table.

In our example, the server contains implementations of the interfaces *Shape* and *ShapeList* in the form of two servant classes, together with a server class that contains a *initialization* section (see Section 5.2.5) in its *main* method.

Figure 20.3 *ShapeListServant* class of the Java server program for CORBA interface *ShapeList*

```
import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
    private POA theRootpoa;
    private Shape theList[];
    private int version;
     private static int n=0;
    public ShapeListServant(POA rootpoa){
        theRootpoa = rootpoa;
        // initialize the other instance variables
    }
    public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException {1
        version++;
        Shape s = null;
        ShapeServant shapeRef = new ShapeServant( g, version);
        try {
            org.omg.CORBA.Object ref = theRoopoa.servant_to_reference(shapeRef);  2
            s = ShapeHelper.narrow(ref);
        } catch (Exception e) {}
         if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s;
        return s;
    }
    public  Shape[] allShapes(){ ... }
    public int getVersion() { ... }
}
```

The servant classes: Each servant class extends the corresponding skeleton class and implements the methods of an IDL interface using the method signatures defined in the equivalent Java interface. The servant class that implements the *ShapeList* interface is named *ShapeListServant*, although any other name could have been chosen. Its outline is shown in Figure 20.3. Consider the method *newShape* in line 1, which is a factory method because it creates *Shape* objects. To make a *Shape* object a CORBA object, it is registered with the POA by means of its *servant_to_reference* method, as shown in line 2, which makes use of the reference to the root POA which was passed on via the constructor when the servant was created. Complete versions of the IDL interface and the client and server classes in this example are available at www.cdk4.net/corba.

The server: The *main* method in the server class *ShapeListServer* is shown in Figure 20.4. It first creates and initializes the ORB (line 1). It gets a reference to the root POA and activates the POAManager (lines 2 & 3). Then it creates an instance of *ShapeListServant,* which is just a Java object (line 4) and in doing this, passes on a reference to the root POA. It then makes it into a CORBA object by registering it with

Figure 20.4 Java class *ShapeListServer*

```
import org.omg.CosNaming.*;
 import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class ShapeListServer {
      public static void main(String args[]) {
         try{
            ORB orb = ORB.init(args, null);                                          1
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));  2
            rootpoa.the_POAManager().activate();                                     3
            ShapeListServant SLSRef = new ShapeListServant(rootpoa);                 4
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(SLSRef);         5
            ShapeList SLRef = ShapeListHelper.narrow(ref);
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);                6
            NameComponent nc = new NameComponent("ShapeList", "");                   7
            NameComponent path[] = {nc};                                             8
            ncRef.rebind(path, SLRef);                                               9
            orb.run();                                                               10
         } catch (Exception e) { ... }
      }
}
```

the POA (line 5). After this, it registers the server with the Naming Service. It then waits for incoming client requests (line 10).

Servers using the Naming Service first get a root naming context (line 6), then make a *NameComponent* (line 7), define a path (line 8) and finally use the *rebind* method (line 9) to register the name and remote object reference. Clients carry out the same steps but use the *resolve* method as shown in Figure 20.5, line 2.

The client program ◊ An example client program is shown in Figure 20.5. It creates and initializes an ORB (line 1), then contacts the Naming Service to get a reference to the remote *ShapeList* object by using its *resolve* method (line 2). After that it invokes its method *allShapes* (line 3) to obtain a sequence of remote object references to all the *Shapes* currently held at the server. It then invokes the *getAllState* method (line 4), giving as argument the first remote object reference in the sequence returned; the result is supplied as an instance of the *GraphicalObject* class.

The *getAllState* method seems to contradict our earlier statement that objects cannot be passed by value in CORBA, because both client and server deal in instances of the class *GraphicalObject*. However, there is no contradiction: the CORBA object returns a *struct*, and clients using a different language might see it differently. For example, in the C++ language the client would see it as a *struct*. Even in Java, the generated class *GraphicalObject* is more like a *struct* because it has no methods.

Figure 20.5 Java client program for CORBA interfaces *Shape* and *ShapeList*

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                                    1
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
            ShapeListHelper.narrow(ncRef.resolve(path));                       2
            Shape[] sList = shapeListRef.allShapes();                          3
            GraphicalObject g = sList[0].getAllState();                        4
        } catch(org.omg.CORBA.SystemException e) {...}
    }
}
```

Client programs should always catch CORBA *SystemExceptions*, which report on errors due to distribution (see line 5). Client programs should also catch the exceptions defined in the IDL interface, such as the *FullException* thrown by the *newShape* method.

This example illustrates the use of the *narrow* operation: the *resolve* operation of the Naming Service returns a value of type *Object*; this type is narrowed to suit the particular type required – *ShapeList*.

Callbacks ◊ Callbacks can be implemented in CORBA in a manner similar to the one described for Java RMI in Section 5.5.1. For example, the *WhiteboardCallback* interface may be defined as follows:

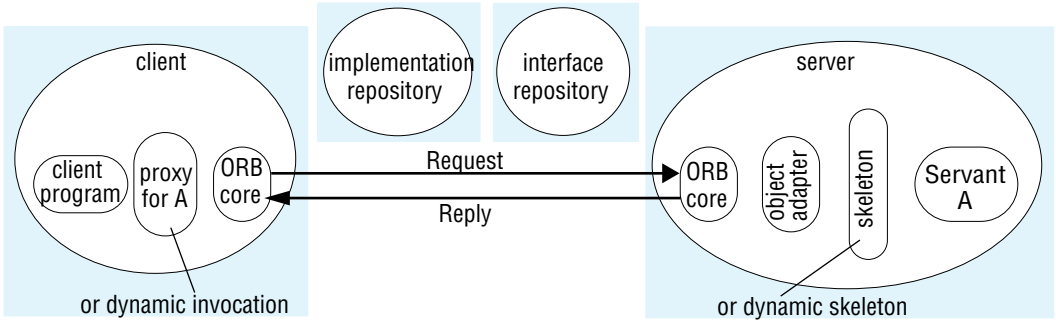
```
interface WhiteboardCallback {
    oneway void callback(in int version);
};
```

This interface is implemented as a CORBA object by the client, enabling the server to send the client a version number whenever new objects are added. But before the server can do this, the client needs to inform the server of the remote object reference of its object. To make this possible, the *ShapeList* interface requires additional methods such as *register* and *deregister*, as follows:

```
int register(in WhiteboardCallback callback);
void deregister(in int callbackId);
```

After a client has obtained a reference to the *ShapeList* object and created an instance of *WhiteboardCallback*, it uses the *register* method of *ShapeList* to inform the server that

Figure 20.6 The main components of the CORBA architecture



it is interested in receiving callbacks. The *ShapeList* object in the server is responsible for keeping a list of interested clients and notifying all of them each time its version number increases when a new object is added. The *callback* method is declared as *oneway* so that the server may use asynchronous calls to avoid delay as it notifies each client.

20.2.2 The architecture of CORBA

The architecture is designed to support the role of an object request broker that enables clients to invoke methods in remote objects, where both clients and servers can be implemented in a variety of programming languages. The main components of the CORBA architecture are illustrated in Figure 20.6.

This figure should be compared with Figure 5.7, in which case it will be noted that the CORBA architecture contains three additional components: the object adapter, the implementation repository and the interface repository.

CORBA provides for both static and dynamic invocations. Static invocations are used when the remote interface of the CORBA object is known at compile time, enabling client stubs and server skeletons to be used. If the remote interface is not known at compile time, dynamic invocation must be used. Most programmers prefer to use static invocation because it provides a more natural programming model.

We now discuss the components of the architecture, leaving those concerned with dynamic invocation until last.

ORB core ◊ The role of the ORB core is similar to that of the communication module of Figure 5.7. In addition, an ORB core provides an interface that includes the following:

- operations enabling it to be started and stopped;
- operations to convert between remote object references and strings;
- operations to provide argument lists for requests using dynamic invocation.

Object adapter ◊ The role of an *object adapter* is to bridge the gap between CORBA objects with IDL interfaces and the programming language interfaces of the

corresponding servant classes. This role also includes that of the remote reference and dispatcher modules in Figure 5.7. An object adapter has the following tasks:

- it creates remote object references for CORBA objects (see Section 20.2.4);
- it dispatches each RMI via a skeleton to the appropriate servant;
- it activates and deactivates servants.

An object adapter gives each CORBA object a unique *object name*, which forms part of its remote object reference. The same name is used each time an object is activated. The object name may be specified by the application program or generated by the object adapter. Each CORBA object is registered with its object adapter, which may keep a remote object table that maps the names of CORBA objects to their servants.

Each object adapter has its own name, which also forms part of the remote object references of all of the CORBA objects it manages. This name may either be specified by the application program or generated automatically.

Portable object adapter ◊ The CORBA 2.2 standard for object adapters is called the Portable Object Adapter. It is called portable because it allows applications and servants to be run on ORBs produced by different developers [Vinoski 1998]. This is achieved by means of the standardization of the skeleton classes and of the interactions between the POA and the servants.

The POA supports CORBA objects with two different sorts of lifetimes:

- those whose lifetimes are restricted to that of the process their servants are instantiated in;
- and those whose lifetimes can span the instantiations of servants in multiple processes.

The former have transient object references and the latter have persistent object references (see Section 20.2.4).

The POA allows CORBA objects to be instantiated transparently; and in addition, it separates the creation of CORBA objects from the creation of the *servants* that implement those objects. Server applications such as databases with large numbers of CORBA objects can create servants on demand, only when the objects are accessed. In this case, they may use database keys for the object names, Alternatively, they may use a single servant to support all of these objects.

In addition, it is possible to specify policies to the POA, for example, as to whether it should provide a separate thread for each invocation, whether the object references should be persistent or transient and whether there should be a separate servant for each CORBA object. The default is that a single servant can represent all of the CORBA objects for its POA.

Skeletons ◊ Skeleton classes are generated in the language of the server by an IDL compiler. As before, remote method invocations are dispatched via the appropriate skeleton to a particular servant, and the skeleton unmarshals the arguments in request messages and marshals exceptions and results in reply messages.

Client stubs/proxies ◊ These are in the client language. The class of a proxy (for object-oriented languages) or a set of stub procedures (for procedural languages) is generated from an IDL interface by an IDL compiler for the client language. As before, the client

stubs/proxies marshal the arguments in invocation requests and unmarshal exceptions and results in replies.

Implementation repository ◊ An implementation repository is responsible for activating registered servers on demand and for locating servers that are currently running. The object adapter name is used to refer to servers when registering and activating them.

An implementation repository stores a mapping from the names of object adapters to the pathnames of files containing object implementations. Object implementations and object adapter names are generally registered with the implementation repository when server programs are installed. When object implementations are activated in servers, the hostname and port number of the server are added to the mapping.

Implementation repository entry:

object adapter name	pathname of object implementation	hostname and port number of server

Not all CORBA objects need to be activated on demand. Some objects, for example callback objects created by clients, run once and cease to exist when they are no longer needed. They do not use the implementation repository.

An implementation repository generally allows extra information to be stored about each server, for example access control information as to who is allowed to activate it or to invoke its operations. It is possible to replicate information in implementation repositories in order to provide availability or fault tolerance.

Interface repository ◊ The role of the interface repository is to provide information about registered IDL interfaces to clients and servers that require it. For an interface of a given type it can supply the names of the methods and for each method, the names and types of the arguments and exceptions. Thus, the interface repository adds a facility for reflection to CORBA. Suppose that a client program receives a remote reference to a new CORBA object. Also suppose that the client has no proxy for it; then it can ask the interface repository about the methods of the object and the types of parameter each of them requires.

When an IDL compiler processes an interface, it assigns a type identifier to each IDL type it encounters. For each interface registered with it, the interface repository provides a mapping between the type identifier of that interface and the interface itself. Thus, the type identifier of an interface is sometimes called the *repository ID* because it may be used as a key to IDL interfaces registered in the interface repository.

Every CORBA remote object reference includes a slot that contains the type identifier of its interface, enabling clients that hold it to enquire of its type with the interface repository. Those applications that use static (ordinary) invocation with client proxies and IDL skeletons do not require an interface repository. Not all ORBs provide an interface repository.

Dynamic invocation interface ◊ As suggested in Section 5.5, in some applications, it may be necessary to construct a client program without knowing all the proxy classes it will need in the future. For example, an object browser might need to display information about all the CORBA objects available in the various servers in a distributed system. It is not feasible that such a program should have to include proxies for all of

these objects, particularly as new objects may be added to the system as time passes. CORBA does not allow classes for proxies to be downloaded at run time as in Java RMI. The dynamic invocation interface is CORBA's alternative.

The dynamic invocation interface allows clients to make dynamic invocations on remote CORBA objects. It is used when it is not practical to employ proxies. The client can obtain from the interface repository the necessary information about the methods available for a given CORBA object. The client may use this information to construct an invocation with suitable arguments and send it to the server.

Dynamic skeletons ◊ Again, as explained in Section 5.5, it may be necessary to add to a server a CORBA object whose interface was unknown when the server was compiled. If a server uses dynamic skeletons, then it can accept invocations on the interface of a CORBA object for which it has no skeleton. When a dynamic skeleton receives an invocation, it inspects the contents of the request to discover its target object, the method to be invoked and the arguments. It then invokes the target.

Legacy code ◊ The term *legacy code* refers to existing code that was not designed with distributed objects in mind. A piece of legacy code may be made into a CORBA object by defining an IDL interface for it and providing an implementation of an appropriate object adapter and the necessary skeletons.

20.2.3 CORBA Interface Definition Language

The CORBA Interface Definition Language, IDL, provides facilities for defining modules, interfaces, types, attributes and method signatures. We have shown examples of all of the above, apart from modules, in Figures 5.2 and 20.1. IDL has the same lexical rules as C++ but has additional keywords to support distribution, for example *interface*, *any*, *attribute*, *in*, *out*, *inout*, *readonly*, *raises*. It also allows standard C++ pre-processing facilities. See, for example, the *typedef* for *All* in Figure 20.7. The grammar of IDL is a subset of ANSI C++ with additional constructs to support method signatures. We give here only a brief overview of IDL. A useful overview and many examples are given in Baker [1997] and Henning and Vinoski [1999]. The full specification is available in on the OMG website [OMG 2002a].

IDL modules ◊ The module construct allows interfaces and other IDL type definitions to be grouped in logical units. A *module* defines a naming scope, which prevents names defined within a module clashing with names defined outside it. For example, the definitions of the interfaces *Shape* and *ShapeList* could belong to a module called *Whiteboard*, as shown in Figure 20.7.

IDL interfaces ◊ As we have seen, an IDL interface describes the methods that are available in CORBA objects that implement that interface. Clients of a CORBA object may be developed just from the knowledge of its IDL interface. From a study of our examples, readers will see that an interface defines a set of operations and attributes and generally depends on a set of types defined with it. For example, the *PersonList* interface in Figure 5.2 defines an attribute and three methods and depends on the type *Person*.

Figure 20.7 IDL module *Whiteboard*.

```
module Whiteboard {
    struct Rectangle{
    ...} ;
    struct GraphicalObject {
    ...};
    interface Shape {
    ...};
    typedef sequence <Shape, 100> All;
    interface ShapeList {
    ...};
};
```

IDL methods ◊ The general form of a method signature is:

[oneway] <return_type> <method_name> (parameter1,..., parameterL)
[raises (except1,..., exceptN)] [context (name1,..., nameM)]

where the expressions in square brackets are optional. For an example of a method signature that contains only the required parts, consider:

void getPerson(in string name, out Person p);

As explained in the introduction to Section 20.2, the parameters are labelled as *in, out* or *inout*, where the value of an *in* parameter is passed from the client to the invoked CORBA object and the value of an *out* parameter is passed back from the invoked CORBA object to the client. Parameters labelled as *inout* are seldom used, but they indicate that the parameter value may be passed in both directions. The return type may be specified as *void* if no value is to be returned.

The optional *oneway* expression indicates that the client invoking the method will not be blocked while the target object is carrying out the method. In addition, *oneway* invocations are executed once or not at all – that is, with *maybe* invocation semantics. We saw the following example in Section 20.2.1:

oneway void callback(in int version);

In this example, where the server calls a client each time a new shape is added, an occasional lost request is not a problem to the client, because the call just indicates the latest version number and subsequent calls are unlikely to be lost.

The optional *raises* expression indicates user-defined exceptions that can be raised to terminate an execution of the method. For example, consider the following example from Figure 20.1:

exception FullException{ };
Shape newShape(in GraphicalObject g) raises (FullException);

The method *newShape* specifies with the *raises* expression that it may raise an exception called *FullException*, which is defined within the *ShapeList* interface. In our example,

the exception contains no variables. However, exceptions may be defined to contain variables, for example:

exception FullException {GraphicalObject g };

When an exception that contains variables is raised, the server may use the variables to return information to the client about the context of the exception.

CORBA can also produce system exceptions relating to problems with servers, such as their being too busy or unable to be activated, problems with communication and client-side problems. Client programs should handle user-defined and system exceptions. The optional *context* expression is used to supply mappings from string names to string values. See Baker [1997] for an explanation of context.

IDL types ◊ IDL supports fifteen primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any primitive or constructed type). Constants of most of the primitive types and constant strings may be declared, using the *const* keyword. IDL provides a special type called *Object*, whose values are remote object references. If a parameter or result is of type *Object*, then the corresponding argument may refer to any CORBA object.

IDL's constructed types are described in Figure 20.8, all of which are passed by value in arguments and results. All arrays or sequences used as arguments must be defined in *typedefs*. None of the primitive or constructed data types can contain references.

Attributes ◊ IDL interfaces can have attributes as well as methods. Attributes are like public class fields in Java. Attributes may be defined as *readonly* where appropriate. The attributes are private to CORBA objects, but for each attribute declared, a pair of accessor methods is generated automatically by the IDL compiler, one to retrieve the value of the attribute and the other to set it. For *readonly* attributes, only the getter method is provided. For example, the *PersonList* interface defined in Figure 5.2 includes the following definition of an attribute:

readonly attribute string listname;

Inheritance ◊ IDL interfaces may be extended. For example, if interface *B* extends interface *A*, this means that it may add new types, constants, exceptions, methods and attributes to those of *A*. An extended interface can redefine types, constants and exceptions, but is not allowed to redefine methods. A value of an extended type is valid as the value of a parameter or result of the parent type. For example, the type *B* is valid as the value of a parameter or result of the type *A*.

interface A { };
interface B: A{ };
interface C {};
interface Z : B, C {};

In addition, an IDL interface may extend more than one interface. For example, interface *Z* extends both *B* and *C*. This means that *Z* has all of the components of both B and *C* (apart from those it redefines) as well as those it defines as an extension.

Figure 20.8 IDL constructed types.

Type	*Examples*	*Use*
sequence	*typedef sequence <Shape, 100> All;* *typedef sequence <Shape> All* bounded and unbounded sequences of *Shapes*	Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.
string	*string name;* *typedef string<8> SmallString;* unbounded and bounded sequences of characters	Defines a sequences of characters, terminated by the null character. An upper bound on the length may be specified.
array	*typedef octet uniqueId[12];* *typedef GraphicalObject GO[10][8]*	Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.
record	*struct GraphicalObject {* *string type;* *Rectangle enclosing;* *boolean isFilled;* *};*	Defines a type for a record containing a group of related entities. *Structs* are passed by value in arguments and results.
enumerated	*enum Rand* *(Exp, Number, Name);*	The enumerated type in IDL maps a type name onto a small set of integer values.
union	*union Exp switch (Rand) {* *case Exp: string vote;* *case Number: long n;* *case Name: string s;* *};*	The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an *enum*, which specifies which member is in use.

When an interface such as Z extends more than one interface, there is a possibility that it may inherit a type, constant or exception with the same name from two different interfaces. For example, suppose that both *B* and *C* define a type called *Q*; then the use of *Q* in the *Z* interface is ambiguous unless a scoped name such as *B*::*Q* or *C*::*Q* is given. IDL does not permit an interface to inherit methods or attributes with common names from two different interfaces.

All IDL interfaces inherit from the type *Object*, which implies that all IDL interfaces are compatible with the type *Object*. This makes it possible to define IDL operations that can takes as argument or return as a result a remote object reference of any type. The *bind* and *resolve* operations in the Naming Service are examples.

IIDL type identifiers ◊ Section 20.2.2 mentioned that type identifiers are generated by the IDL compiler for each type in an IDL interface. For example, the IDL type for the interface *Shape* type (Figure 20.7) might be:

IDL:Whiteboard/Shape:1.0

This example shows that an IDL type name has three parts – the IDL prefix, a type name and a version number. Since interface identifiers are used as keys for accessing interface definitions in the interface repository, programmers must ensure that they provide a unique mapping to the interfaces themselves. A programmer may use the IDL prefix pragma to prefix an additional string to the type name in order to distinguish their own types from those of others.

IDL pragma directives: These allow additional, non-IDL properties to be specified for components in an IDL interface (see Henning and Vinoski [1999]). These properties include, for example, specifying that an interface will be used only locally, or supplying the value of an interface repository ID. Each pragma is introduced by *#pragma* and specifies its type, for example:

#pragma version Whiteboard 2.3

Extensions to CORBA ◊ Some new features were added in version 2.3 of the CORBA specification. These include the ability to pass non-CORBA objects by value and an asynchronous variant of RMI. Both of these are discussed in the *CACM* article by Vinoski [1998].

Objects that can be passed by value: As we have seen above, IDL arguments and results of constructed and primitive types are passed by value, whereas those that refer to CORBA objects are passed by reference. Support for passing non-CORBA objects by value is now part of CORBA [OMG 2002c]. These non-CORBA objects are object-like in the sense that they possess both attributes and methods. However, they are purely local objects in that their operations cannot be invoked remotely. The pass-by-value facility provides the ability to pass a copy of a non-CORBA object between client and server.

This is achieved by the addition to IDL of a type called *valuetype* for representing non-CORBA objects. A *valuetype* is a *struct* with additional method signatures (like those of an interface). *Valuetype* arguments and results are passed by value; that is, the state is passed to the remote site and used to produce a new object at the destination.

The methods of this new object may be invoked locally, causing its state to diverge from the state of the original object. Passing the implementation of the methods is not so straightforward, since client and server may use different languages. However, if client and server are both implemented in Java, the code can be downloaded. For a common implementation in C++, the necessary code would need to be present at both client and server.

This facility is useful when it is beneficial to place a copy of an object in the client process to enable it to receive local invocations. However, it does not get us any nearer to passing CORBA objects by value.

Asynchronous RMI: CORBA now provides a form of asynchronous RMI which allows clients to make non-blocking invocation requests on CORBA objects [OMG 2004e]. It is intended to be implemented in the client. Therefore a server is generally unaware as

to whether it is invoked synchronously or asynchronously. (One exception is the Transaction Service which would need to be aware of the difference.)

Asynchronous RMI adds two new variants to the invocation semantics of RMIs:

- *callback*, in which a client uses an extra parameter to pass a reference to a callback with each invocation, so that the server can call back with the results;
- and *polling*, in which the server returns a *valuetype* object that can be used to poll or wait for the reply.

The architecture of asynchronous RMI allows an intermediate agent to be deployed to make sure that the request is carried out and if necessary to store the reply. Thus it is appropriate for use in environments where clients may become temporarily disconnected – as, for example, a client using a laptop in a train.

20.2.4 CORBA remote object references

CORBA 2.0 specifies a format for remote object references that is suitable for use, whether or not the remote object is to be activated by an implementation repository. References using this format are called interoperable object references (IORs). The following figure is based on Henning [1998], which contains a more detailed account of IORs:

IOR format

IDL interface type id	Protocol and address details			Object key	
interface repository identifier	IIOP	host domain name	port number	adapter name	object name

- The first field of an IOR specifies the type identifier of the IDL interface of the CORBA object. Note that if the ORB has an interface repository, this type name is also the interface repository identifier of the IDL interface, which allows the IDL definition for the interface to be retrieved at runtime.
- The second field specifies the transport protocol and the details required by that particular transport protocol to identify the server. In particular, the Internet Inter-ORB protocol (IIOP) uses TCP, in which the server address consists of a host domain name and a port number.
- The third field is used by the ORB to identify a CORBA object. It consists of the name of an object adapter in the server and the object name of a CORBA object specified by the object adapter.

Transient IORs for CORBA objects last only as long as the process that hosts those objects, whereas *persistent IORs* last between activations of the CORBA objects. A transient IOR contains the address details of the server hosting the CORBA object, whereas a persistent IOR contains the address details of the implementation repository with which it is registered. In both cases, the client ORB sends the request message to the server whose address details are given in the IOR. We now discuss how the IOR is used to locate the servant representing the CORBA object in the two cases.

Transient IORs: The server ORB core receives the request message containing the object adapter name and object name of the target. It uses the object adapter name to locate the object adapter, which uses the object name to locate the servant.

Persistent IORs: An implementation repository receives the request. It extracts the object adapter name from the IOR in the request. Provided that the object adapter name is in its table, it attempts if necessary to activate the CORBA object at the host address specified in its table. Once the CORBA object has been activated, the implementation repository returns its address details to the client ORB, which uses them as the destination for RMI request messages, which include the object adapter name and the object name. These enable the server ORB core to locate the object adapter, which uses the object name to locate the servant, as before.

The second field of an IOR may be repeated, so as to specify the host domain name and port number of more than one destination, to allow for an object or an implementation repository to be replicated at several different locations.

The reply message in the request-reply protocol includes header information that enables the procedure for persistent IORs to be carried out. In particular, it includes a status entry that can indicate whether the request should be forwarded to a different server, in which case the body of the reply includes an IOR that contains the address of the server of the newly activated object.

20.2.5 CORBA language mappings

We have seen from our examples that the mapping from the types in IDL to Java types is quite straightforward. The primitive types in IDL are mapped to the corresponding primitive types in Java. *Structs, enums* and *unions* are mapped to Java classes; sequences and arrays in IDL are mapped to arrays in Java. An IDL exception is mapped to a Java class that provides instance variables for the fields of the exception and constructors. The mappings in C++ are similarly straightforward.

However, we have seen that some difficulties arise with mapping the parameter passing semantics of IDL onto those of Java. In particular, IDL allows methods to return several separate values via output parameters, whereas Java can have only a single result. The *Holder* classes are provided to overcome this difference, but this requires the programmer to make use of them, which is not altogether straightforward. For example, the method *getPerson* in Figure 5.2 is defined in IDL as follows:

void getPerson(in string name, out Person p);

and the equivalent method in the Java interface would be defined as:

void getPerson(String name, PersonHolder p);

and the client must provide an instance of *PersonHolder* as the argument of its invocation. The holder class has an instance variable that holds the value of the argument for the client to access when the invocation returns. It also has methods to transmit the argument between server and client.

Although C++ implementations of CORBA can handle *out* and *inout* parameters quite naturally, C++ programmers suffer from a different set of problems with parameters, related to storage management. These difficulties arise when object

references and variable-length entities such as strings or sequences are passed as arguments.

For example, in Orbix [Baker 1997] the ORB keeps reference counts to remote objects and proxies and releases them when they are no longer needed. It provides programmers with methods for releasing or duplicating them. Whenever a server method has finished executing, the *out* arguments and results are released and the programmer must duplicate them if they will still be needed. For example, a C++ servant implementing the *ShapeList* interface will need to duplicate the references returned by the method *allShapes*. Object references passed to clients must be released when they are no longer needed. Similar rules apply to variable-length parameters.

In general, programmers using IDL not only have to learn the IDL notation itself but also have an understanding of how its parameters are mapped onto the parameters of the implementation language.

20.2.6 Integration of CORBA and Web Services

CORBA was already well established and widely used within organizations, when web services started to emerge, early in the twenty-first century. Chapter 19 argues that web services are very suitable for use for interworking between organizations over the Internet. Section 19.2.4 makes a comparison of web services and CORBA, showing that although CORBA is not suited to inter-organizational distributed applications, its main benefits are efficiency and the fact that it provides a set of services for transactions, concurrency control, security and so forth (see Section 20.3).

Many organizations rely on CORBA applications, with their associated benefits of reliability and good performance. But there could be considerable additional advantages from integrating CORBA services with web services. A useful approach would be to provide a WSDL service description (Section 19.3) to existing CORBA services. The IDL definition of a CORBA object would be expressed in XML in the abstract part of a WSDL service description and the communication protocol (for example, IIOP) would be specified in the concrete part of the description.

This would allow a CORBA object to be accessed by clients as though it were any other web service. Once that is possible, new web services may be built by combining CORBA services with web service interfaces with other web services. This would enable the users of CORBA services to benefit from the advantages of the flexibility and lightweight infrastructure associated with web services.

In autumn 2004, the OMG called for proposals for a set of CORBA bindings to WSDL, which will require:

- the mapping of CORBA interfaces defined in IDL into WSDL descriptions;
- the mapping of IDL types into XML schema types;
- a mechanism for making instances of CORBA objects accessible as web services via the communication mechanism required by those CORBA objects.

Once this work is completed, it should be possible to advertise the interfaces of existing CORBA services in WSDL. Clients will be able to access these interfaces as though they are web services, without being aware of the actions of the underlying CORBA middleware. Client programs will be compiled to use the message format, data

representation and communication protocols specified in the WSDL binding. They will appear to address CORBA objects by means of URLs but these will be translated into IORs.

20.3 CORBA services

CORBA includes specifications for services that may be required by distributed objects. In particular, the Naming Service is an essential addition to any ORB, as we saw in our programming example in Section 20.2. An index to documentation on all of the services can be found at OMG's web site at [www.omg.org]. Many of the CORBA services are described in Orfali *et al*. [1996 and 1997]. The CORBA services include the following:

Naming Service: The CORBA naming service is detailed in Section 20.3.1.

Event Service and Notification Service: The CORBA event service is discussed in 20.3.2 and the notification service in 20.3.3.

Security service: The CORBA security service is discussed in Section 20.3.4.

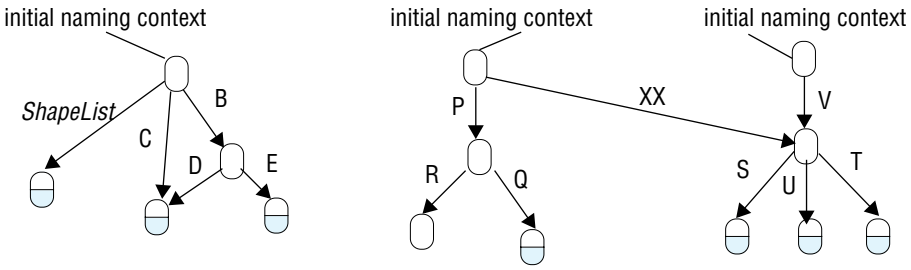
Trading service: In contrast to the Naming Service which allows CORBA objects to be located by name, the Trading Service [OMG 2000a] allows them to be located by attribute – that is, it is a directory service. Its database contains a mapping from service types and their associated attributes onto remote object references of CORBA objects. The service type is a name, and each attribute is a name-value pair. Clients make queries by specifying the type of service required, together with other arguments specifying constraints on the values of attributes, and preferences for the order in which to receive matching offers. Trading servers can form federations in which they not only use their own databases but also perform queries on behalf of one anothers' clients. For a detailed description of the Trading Service, see Henning and Vinoski [1999].

Transaction service and concurrency control service: The object transaction service [OMG 2003] allows distributed CORBA objects to participate in either flat or nested transactions. The client specifies a transaction as a sequence of RMI calls, which are introduced by *begin* and terminated by *commit* or *rollback* (*abort*). The ORB attaches a transaction identifier to each remote invocation and deals with *begin*, *commit* and *rollback* requests. Clients can also suspend and resume transactions. The transaction service carries out a two-phase commit protocol. The concurrency control service [OMG 2000b] uses locks to apply concurrency control to the access of CORBA objects. It may be used from within transactions or independently.

Persistent state service: Section 5.2.5 explained that persistent objects can be implemented by storing them in a passive form in a persistent object store while they are not in use and activating them when they are needed. Although ORBs activate CORBA objects with persistent object references, getting their implementations from the implementation repository, they are not responsible for saving and restoring the state of CORBA objects.

The CORBA Persistent State Service is intended to be suitable for use as a persistent object store for CORBA objects [OMG 2002d]. It replaces an earlier

Figure 20.9 Naming graph in CORBA Naming Service



service called the Persistent Object Service. It is based on an architecture in which servants have access to a datastore, for example a database or a file system, via an internal interface. Servants that are to represent persistent objects are called *storage objects* and they are kept in *storage homes* both within the server process and the datastore. Each storage home contains only storage objects of a particular type. A Java-like language is provided for specifying the interfaces of storage objects and associating them with storage homes. The servants can create and access storage objects within their storage homes. Storage objects may also be used transparently via programming languages including Java and C++. For transparent persistence, a pre-processor inserts operations to transfer objects between the servants and the datastore. The Persistent State Service is designed to be used in the context of transactions in the Transaction Service.

Life cycle service The life cycle service defines conventions for creating, deleting, copying and moving CORBA objects. It specifies how clients can use factories to create objects in particular locations, allowing persistent storage to be used if required. It defines an interface that allows clients to delete CORBA objects or to move or copy them to a specified location. Strategies for making shallow and deep copies are discussed [OMG 2002e].

20.3.1 CORBA Naming Service

The CORBA Naming Service is a sophisticated example of the binder described in Chapter 5. It allows names to be bound to the remote object references of CORBA objects within naming contexts.

As explained in Section 9.2, a *naming context* is the scope within which a set of names applies – each of the names within a context must be unique. A name can be associated with either an object reference for a CORBA object in an application or with another context in the naming service. Contexts may be nested so as to provide a hierarchic name space, as shown in Figure 20.9, in which CORBA objects are shown in the normal way, but naming contexts are shown as plain ovals. The graph on the left shows an entry for the *ShapeList* object described in the programming example in Section 20.2.1.

An *initial naming context* provides a root for a set of bindings. Note that more than one initial naming context can point into the same naming graph. In practice, each

instance of an ORB has a single initial naming context, but name servers associated with different ORBs can form federations, as described later in this section. Client and server programs request the initial naming context from the ORB, as shown in Figure 20.5, by invoking its method *resolve_initial_references*, giving *"NameService"* as argument. The ORB returns a reference to an object of type *NamingContext* – see Figure 20.10. This refers to the initial context of the name server for that ORB. Since there can be several initial contexts, objects do not have absolute names – names are always interpreted relative to an initial naming context.

A name with one or more components can be resolved, starting in any naming context. To resolve a name with several components, the naming service looks in the starting context for a binding that matches the first component. If one exists, it will be either a remote object reference or a reference to another naming context. If the result is a naming context, the second component of the name is resolved in that context. This procedure is repeated until all the components of a name have been resolved and a remote object reference obtained, unless the matching fails on the way.

The names used by the CORBA Naming Service are two-part names, called *NameComponent*s, each of which consists of two strings, one for the *name* and the other for the *kind* of the object. The *kind* field provides a single attribute that is intended for use by applications and may contain any useful descriptive information; it is not interpreted by the Naming Service.

Although CORBA objects are given hierarchic names by the Naming Service, these names cannot be expressed as pathnames like those of UNIX files. So, in Figure 20.9, we cannot refer to the object on the far right as /V/T. This is because names can include any characters, which precludes the possibility of having a delimiter.

Figure 20.10 shows the main operations provided by the *NamingContext* class of the CORBA Naming Service, defined in CORBA IDL. The full specification may be obtained from OMG [OMG2004b]. For simplicity, our figure does not describe the exceptions raised by the methods. For example, the *resolve* method can throw a *NotFound* exception, and *bind* can throw an *AlreadyBound* exception.

Clients use the *resolve* method to look up object references by name. Its return type is *Object*, so it can return references to any type of object belonging to applications. The result must be narrowed before it can be used to invoke a method in an application remote object, as shown in Figure 20.5, line 2. The argument of *resolve* is of type *Name*, which is defined as a sequence of name components. This means that the client must construct a sequence of name components before making the call. Figure 20.5 showed a client making an array called *path* consisting of a single name component, which it used as argument to *resolve*. This does not seem a very convenient alternative to using a normal pathname.

Servers of remote objects use the *bind* operation to register names for their objects and *unbind* to remove them. The *bind* operation binds a given name and remote object reference and is invoked on the context in which the binding is to be added. See Figure 20.4, in which the name *ShapeList* was bound in the initial naming context. In that example, the method *rebind* is used because the *bind* operation throws an exception if it is called with a name that already has a binding, whereas *rebind* allows bindings to be replaced.

The *bind_new_context* operation is used to create a new context and to bind it with the given name in the context on which it was invoked. Another method called

Figure 20.10 Part of the CORBA Naming Service *NamingContext* interface in IDL

```
struct NameComponent { string id; string kind; };

typedef sequence <NameComponent> Name;

interface NamingContext  {

    void bind (in Name n,  in Object  obj);
        binds the given name and remote object reference in my context.

    void unbind (in Name n);
        removes an existing binding with the given name.

    void bind_new_context(in Name n);
        creates a new naming context and binds it to a given name in my context.

    Object  resolve (in Name n);
         looks up the name in my context and returns its remote object reference.

    void list  (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
        returns the names in the bindings in my context.
};
```

bind_context binds a given naming context to a given name in the context on which it was invoked. The *unbind* method can be used to remove contexts as well as names.

The operation *list* is intended to be used for browsing the information available from a context in the Naming Service. It returns a list of bindings from a target *NameContext*. Each binding consists of a name and a type – an object or a context. Sometimes, a naming context may contain a very large number of bindings, in which case it would be undesirable to return them all as the result of a single invocation. For this reason, the *list* method returns some maximum number of bindings as a result of a call to *lis*t, and if further bindings remain to be sent, it arranges to return the results in batches. This is achieved by returning an iterator as a second result. The client uses the iterator to retrieve the remainder of the results a few at a time.

The method *list* is shown in Figure 20.10, but the definitions of the types of its arguments are omitted for the sake of simplicity. The type *BindingList* is a sequence of bindings, each of which contains a name and its type, which is either a context or remote object reference. The type *BindingIterator* provides a method *next_n* for accessing its next set of bindings; its first argument specifies how many bindings are wanted and its second argument receives a sequence of bindings. The client calls the method *list* giving as first argument the maximum number of bindings to be obtained immediately via the second argument. The third argument is an iterator, which can be used to obtain the remainder of the bindings, if any.

The CORBA name space allows for the federation of Naming Services, using a scheme in which each server provides a subset of the name graph. For example, in Figure 20.9, the initial naming contexts in the graphs in the middle and on the right are managed by different servers. The graph in the middle has a binding labelled 'XX' to a context in the graph on the right by which clients may access objects named in the

remote graph. To complete the federation, the graph on the right would need to add a binding to a node in the middle graph. An organization can provide access to some or all of the contexts in its name space by providing remote name servers with remote references to them.

The Java implementation of the CORBA Naming Service is very simple and is called transient because it stores all of its bindings in volatile memory. Any serious implementation would at least keep copies of its naming graph in files. As we have seen in the DNS study, replication can be used to provide better availability.

20.3.2 CORBA Event Service

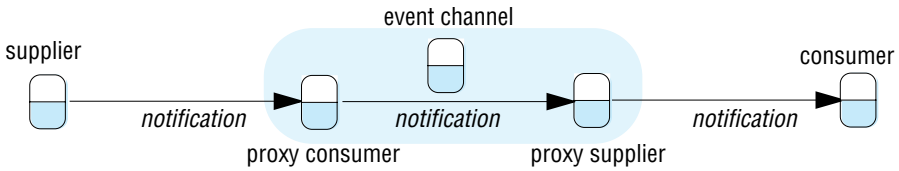
The CORBA Event Service specification defines interfaces allowing objects of interest, called *suppliers*, to communicate notifications to subscribers, called *consumers*. The notifications are communicated as arguments or results of ordinary synchronous CORBA remote method invocations. Notifications may be propagated either by being *pushed* by the supplier to the consumer or *pulled* by the consumer from the supplier. In the first case, the consumers implement the *PushConsumer* interface which includes a method *push* that takes any CORBA data type as argument. Consumers register their remote object references with the suppliers. The supplier invokes the *push* method, passing a notification as argument. In the second case, the supplier implements the *PullSupplier* interface, which includes a method *pull* that receives any CORBA data type as its return value. Suppliers register their remote object references with the consumers. The consumers invoke the *pull* method and receive a notification as result.

The notification itself is transmitted as an argument or result whose type is *any*, which means that the objects exchanging notifications must have an agreement about the contents of notifications. Application programmers, however, may define their own IDL interfaces with notifications of any desired type.

Event channels are CORBA objects that may be used to allow multiple suppliers to communicate with multiple consumers in an asynchronous manner. An event channel acts as a buffer between suppliers and consumers. It can also multicast the notifications to the consumers. Communication via an event channel may use either the push or pull style. The two styles may be mixed; for example, suppliers may push notifications to the channel and consumers may pull notifications from it.

When a distributed application needs to use asynchronous notifications, it creates an event channel, which is a CORBA object whose remote object reference may be supplied to the components of the application via the Naming Service or by means of an RMI. The suppliers in the application make themselves available for subscription by getting *proxy consumers* from the event channel and connecting the suppliers to them by passing them their remote object references. The consumers in the application subscribe to notifications by getting *proxy suppliers* from the notification channel and connecting the consumers to them. The proxy suppliers and consumers are available in both push and pull styles. When a supplier generates a notification using the push style of interaction, it calls the *push* method of a push proxy consumer. The notification passes through the channel and is given to the proxy suppliers, which pass them on to the consumers, as shown in Figure 20.11. If the consumers use the pull style of interaction, they will call the *pull* method of a pull proxy supplier.

Figure 20.11 CORBA event channels



The presence of the proxy suppliers and proxy consumers makes it possible to construct chains of event channels in which each channel supplies notifications to be consumed by the following channel. The event channels in the CORBA model are similar to the observers defined in Figure 5.11. They may be programmed to carry out some of the roles of observers discussed in Section 5.4. However, notifications do not carry any form of identifiers, and therefore the recognition of patterns or the filtering of notifications will need to be based on type information put in the notifications by the application.

A fuller explanation of the CORBA Event Service and an outline of its main interfaces is given in Farley [1998]. The full specification of the CORBA Event service is in [OMG 2004c]. However, the specification does not state how to create an event channel nor how to request the reliability required from it.

20.3.3 CORBA Notification Service

The CORBA Notification Service [OMG 2004d] extends the CORBA Event Service, retaining all of its features including event channels, event consumers and event suppliers. The event service provides no support for filtering events or for specifying delivery requirements. Without the use of filters, all the consumers attached to a channel have to receive the same notifications as one another. And without the ability to specify delivery requirements, all of the notifications sent via a channel are given the delivery guarantees built into the implementation.

The notification service adds the following new facilities:

- Notifications may be defined as data structures. This is an enhancement of the limited utility provided by notifications in the event service, whose type could only be either *any* or a type specified by the application programmer.
- Event consumers may use filters that specify exactly which events they are interested in. The filters may be attached to the proxies in a channel. The proxies will forward notifications to event consumers according to constraints specified in filters in terms of the contents of each notification.
- Event suppliers are provided with a means of discovering the events the consumers are interested in. This allows them to generate only those events that are required by the consumers.

- Event consumers can discover the event types offered by the suppliers on a channel, which enables them to subscribe to new events as they become available.
- It is possible to configure the properties of a channel, a proxy or a particular event. These properties include the reliability of event delivery, the priority of events, the ordering required (for example, FIFO or by priority) and the policy for discarding stored events.
- An event type repository is an optional extra. It will provide access to the structure of events, making it convenient to define filtering constraints.

The Structured Event type introduced by the notification service provides a data structure into which a wide variety of different types of notification can be mapped. Filters can be defined in terms of the components of the Structured Event type. A structured event consists of an event header and an event body. The following example illustrates the contents of the header:

domain type	*event type*	*event name*	*requirements*
"home"	"burglar alarm"	"21 Mar at 2pm"	"priority", 1000

The domain type refers to the defining domain (for example, *"finance"*, *"hotel"* or *"home"*). The event type categorizes the type of event uniquely within the domain (for example, *"stock quote"*, *"breakfast time"*, *"burglar alarm"*). The event name uniquely identifies the specific instance of the event being transmitted. The remainder of the header contains a list of <name, value> pairs, which are intended to be used to specify reliability and other requirements on event delivery.

The following example illustrates the information in the body of a structured event:

← *filterable part* →			
name, value	*name, value*	*name, value*	*remainder*
"bell" , "ringing"	"door" ,"open"	"cat", "outside"	

The first part of the event body contains a sequence of <name, value> pairs which are intended for use by the filters. It is expected that different industry domains will define standards for the <name, value> pairs that are used in the filterable part of the event body – the same names and values will be used when defining filters. Perhaps when the burglar alarm goes off, the event may include the state of the alarm bell, whether the front door is open and the location of the cat. The remainder of the event body is intended for transmitting data relating to the particular event; for example, when the burglar alarm goes off, it might contain a digital photograph of the interior of the premises.

Filter objects are used by proxies in making decisions as to whether to forward each notification. A filter is designed as a collection of constraints, each of which is a data structure with two components:

- A list of data structures, each of which indicates an event type in terms of its domain name and event type, for example, *"home"*, *"burglar alarm"*. The list includes all of the event types to which the constraint should apply.

- A string containing a boolean expression involving the values of the event types listed above. For example:

("domain type" == "home" && "event type" == "burglar alarm") &&
("bell" != "ringing" !! "door" == "open")

Our example uses an informal syntax. The notification service specification includes the definition of a constraint language, which is an extension of the constraint language used by the trading service.

20.3.4 CORBA Security Service

The CORBA Security Service [Blakley 1999, Baker 1997, OMG 2002b] includes the following:

- Authentication of principals (users and servers); generating credentials for principals (that is, certificates stating their rights); delegation of credentials is supported as described in Section 7.2.5.
- Access control can be applied to CORBA objects when they receive remote method invocations. Access rights may for example be specified in access control lists (ACLs).
- Security of communication between clients and objects, protecting messages for integrity and confidentiality.
- Auditing by servers of remote method invocations.
- Facilities for non-repudiation. When an object carries out a remote invocation on behalf of a principal, the server creates and stores credentials that prove that the invocation was done by that server on behalf of the requesting principal.

To guarantee that security is applied correctly to remote method invocations, the security service requires cooperation on behalf of the ORB. To make a secure remote method invocation, the client's credentials are sent in the request message. When the server receives a request message, it validates the client's credentials to see, for example, if they are fresh and signed by an acceptable authority. If the credentials are valid, they are used to make a decision as to whether the principal has the right to access the remote object using the method in the request message. This decision is made by consulting an object containing information about which principal is allowed to access each method of the target object (possibly in the form of an ACL). If the client has sufficient rights, the invocation is carried out and the result returned to the client, together with the server's credentials if needed. The target object may also record details about the invocation in an audit log or store non-repudiation credentials.

CORBA allows a variety of security policies to be specified according to requirements. A message-protection policy states whether client or server (or both) must be authenticated, and whether messages must be protected against disclosure and/or modification. Policies may also be specified with respect to auditing and non-repudiation; for example, a policy might state which methods and arguments they should be applied to.

Access control takes into account that many applications have large numbers of users and even larger numbers of objects, each with its own set of methods. Users are supplied with a special type of credential called a *privilege* according to their roles. Objects are grouped into *domains*. Each domain has a single access control policy specifying the access rights for users with particular privileges to objects within that domain. To allow for the unpredictable variety of methods, each method is classified in terms of one of four generic methods (*get*, *set*, *use* and *manage*). *Get* methods just return parts of the object state, *set* methods alter the object state, *use* methods cause the object to do some work, and *manage* methods perform special functions that are not intended to be available for general use. Since CORBA objects have a variety of different interfaces, the access rights must be specified for each new interface in terms of the above generic methods. This involves application designers being involved in the application of access control, the setting of appropriate privilege attributes (for example, groups or roles) and in helping the user to acquire the appropriate privileges for their task.

In its simplest form, security may be applied in a manner that is transparent to applications. It includes applying the required protection policy to remote method invocations, together with auditing. The security service allows users to acquire their individual credentials and privileges in return for supplying authentication data such as a password.

20.4 Summary

The main component of CORBA is the Object Request Broker or ORB, which allows clients written in one language to invoke operations in remote objects (called CORBA objects) written in another language. CORBA addresses other aspects of heterogeneity as follows:

- The CORBA General Inter-ORB protocol (GIOP) includes an external data representation called CDR, which makes it possible for clients and servers to communicate irrespective of their hardware. It also specifies a standard form for remote object references.
- GIOP also includes a specification for the operations of a request-reply protocol that can be used irrespective of the underlying operating system.
- The Internet Inter-ORB Protocol (IIOP) implements the request-reply protocol over TCP. IIOP remote object references include the domain name and port number of a server.

A CORBA object implements the operations in an IDL interface. All that clients need to know to access a CORBA object is the operations available in its interface. The client program accesses CORBA objects via proxies or stubs, which are generated automatically from their IDL interfaces in the language of the client. Server skeletons for CORBA objects are generated automatically from their IDL interfaces in the language of the client. The object adapter is an important component of CORBA

servers. Its roles include activating and deactivating servants, creating remote object references and forwarding request messages to the appropriate servants.

The CORBA architecture allows CORBA objects to be activated on demand. This is achieved by a component called the implementation repository, which keeps a database of implementations indexed by their object adapter names. When a client invokes a CORBA object, it can be activated if necessary in order to carry out the invocation.

An interface repository is a database of IDL interface definitions indexed by repository IDs. Since the IOR of a CORBA object contains the repository ID of its interface, the appropriate interface repository can be used to get the information about the methods in its interface which is required for dynamic method invocations.

CORBA services provide functionality above RMI, which may be required by distributed applications, allowing them to use additional services such as naming and directory services, event notifications, transactions or security as required.

EXERCISES

20.1 The Task Bag is an object that stores pairs of (key and value). A key is a string and a value is a sequence of bytes. Its interface provides the following remote methods:

pairOut: with two parameters through which the client specifies a *key* and a *value* to be stored.

pairIn: whose first parameter allows the client to specify the *key* of a pair to be removed from the Task Bag. The *value* in the pair is supplied to the client via a second parameter. If no matching pair is available, an exception is thrown.

readPair: is the same as *pairIn* except that the pair remains in the Task Bag.

Use CORBA IDL to define the interface of the Task Bag. Define an exception that can be thrown whenever any one of the operations cannot be carried out. Your exception should return an integer indicating the problem number and a string describing the problem. The Task Bag interface should define a single attribute giving the number of tasks in the bag. *page 839*

20.2 Define an alternative signature for the methods *pairIn* and *readPair*, whose return value indicates when no matching pair is available. The return value should be defined as an enumerated type whose values can be *ok* and *wait*. Discuss the relative merits of the two alternative approaches. Which approach would you use to indicate an error such as a key that contains illegal characters? *page 840*

20.3 Which of the methods in the Task Bag interface could have been defined as a *oneway* operation? Give a general rule regarding the parameters and exceptions of *oneway* methods. In what way does the meaning of the *oneway* keyword differ from the remainder of IDL? *page 840*

20.4 The IDL *union* type can be used for a parameter that will need to pass one of a small number of types. Use it to define the type of a parameter that is sometimes empty and sometimes has the type *Value*. *page 842*

20.5 In Figure 20.1 the type *All* was defined as a sequence of a fixed length. Redefine this as an array of the same length. Give some recommendations as to the choice between arrays and sequences in an IDL interface. *page 842*

20.6 The Task Bag is intended to be used by cooperating clients, some of which add pairs (describing tasks) and others remove them (and carry out the tasks described). When a client is informed that no matching pair is available, it cannot continue with its work until a pair becomes available. Define an appropriate callback interface for use in this situation. *page 835*

20.7 Describe the necessary modifications to the Task Bag interface to allow callbacks to be used. *page 835*

20.8 Which of the parameters of the methods in the TaskBag interface are passed by value and which are passed by reference? *page 830*

20.9 Use the Java IDL compiler to process the interface you defined in Exercise 20.1. Inspect the definition of the signatures for the methods *pairIn* and *readPair* in the generated Java equivalent of the IDL interface. Look also at the generated definition of the holder method for the value argument for the methods *pairIn* and *readPair*. Now give an example showing how the client will invoke the *pairIn* method, explaining how it will acquire the value returned via the second argument. *page 845*

20.10 Give an example to show how a Java client will access the attribute giving the number of tasks in the Task bag object. In what respects does an attribute differ from an instance variable of an object? *page 841*

20.11 Explain why the interfaces to remote objects in general and CORBA objects in particular do not provide constructors. Explain how CORBA objects can be created in the absence of constructors. *Chapter 5 and page 833*

20.12 Redefine the Task Bag interface from Exercise 20.1 in IDL so that it makes use of a *struct* to represent a *Pair*, which consists of a *Key* and a *Value*. Note that there is no need to use a *typedef* to define a *struct*. *page 842*

20.13 Discuss the functions of the implementation repository from the point of view of scalability and fault tolerance. *page 838, page 844*

20.14 To what extent may CORBA objects be migrated from one server to another? *page 838, page 844*

20.15 Discuss the benefits and drawbacks of the two-part names or *NameComponents* in the CORBA naming service. *page 848*

20.16 Give an algorithm that describes how a multipart name is resolved in the CORBA naming service. A client program needs to resolve a multipart name with components "A", "B" and "C", relative to an initial naming context. How would it specify the arguments for the *resolve* operation in the naming service? *page 848*

20.17 A virtual enterprise consists of a collection of companies who are cooperating with one another to carry out a particular project. Each company wishes to provide the others with access to only those of its CORBA objects relevant to the project. Describe an appropriate way for the group to federate their CORBA Naming Services. *page 850*

20.18 Discuss how to use directly connected suppliers and consumers of the CORBA event service in the context of the shared whiteboard application. The *PushConsumer* and *PushSupplier* interfaces are defined in IDL as follows:

```
interface PushConsumer {
        void push(in any data) raises (Disconnected);
        void disconnect_push_consumer();
}
interface PushSupplier {
        void disconnect_push_supplier();
}
```

Either the supplier or the consumer may decide to terminate the event communication by calling *disconnect_push_supplier*() or *disconnect_push_consume*r() respectively.

page 851

20.19 Describe how to interpose an Event Channel between the supplier and the consumers in your solution to Exercise 20.18. An event channel has the following IDL interface:

```
interface EventChannel {
        ConsumerAdmin for_consumers();
        SupplierAdmin for_suppliers();
};
```

where the interfaces *SupplierAdmin* and *ConsumerAdmin,* which allow the supplier and the consumer to get proxies are defined in IDL as follows:

```
interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer();
        ---
};
interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier();
        ---
};
```

The interface for the proxy consumer and procy supplier are defined in IDL as follows:

```
interface ProxyPushConsumer : PushConsumer{
        void connect_push_supplier (in PushSupplier supplier)
            raises (AlreadyConnected);
};
interface ProxyPushSupplier : PushSupplier{
        void connect_push_consumer (in PushConsumer consumer)
            raises (AlreadyConnected);
};
```

What advantage is gained by the use of the event channel? *page 851*