5 Distributed Object Technologies (CORBA and DCOM)

5.1 Distributed Object Technologies Overview

5.1.1 Concepts

Most of the new applications developed at present are based on the OO concepts. In addition, many legacy applications are being "wrapped" so that they appear as objects to the outside world. However, rarely, if ever, all objects of an application reside on the same machine. It is common to find that the objects of an application will be dispersed on multiple machines. A distributed application can be viewed as a collection of objects (user interfaces, databases, application modules, customers). Each object has its own attributes, and has some methods which define the behavior of the object (e.g., an order can be viewed in terms of its data and the methods which create, delete and update the order object). Interactions between the components of an application can be modeled through "messages" which invoke appropriate methods. In particular, classes and inheritance are extremely useful in modeling applications because these concepts lead to reuse and encapsulation - critical to managing the complexity of distributed systems. For example:

- A customer is defined as a class from which other business classes that define different types of customers can inherit properties.
- An inventory is defined as a class from which other properties of specific inventory items can be inherited.
- A database server is defined as a class from which other vendor specific database servers can inherit properties.
- A network is defined as a class from which other networks inherit properties (e.g., a generic network from which local and wide area networks can inherit properties).

Objects, wherever they reside, are data that can be accessed through methods and support properties such as inheritance, polymorphism, encapsulation, etc. Objects can be clients, servers, or both (see Appendix B for a tutorial on object-oriented concepts). It is quite easy to access a local object in current OO programming languages such as Java and C++. All you need to do is issue the following statement:

```
Object.method (parameters)
```

For example, consider the following object:

```
Object Name    = invoice
Attributes     = customer name, items purchased, price per item, total invoice, etc.
Methods        = prepare, send, review status, update status
```

To prepare an invoice, we say invoice.prepare (parameters for invoice preparation), and to send an invoice, we say, invoice.send (parameters for sending), etc (we are overlooking a few programming details). The "dot" between object name and the method name is of key importance because it tells the compiler to invoke the method of a given object. How can you invoke a method (with associated parameters) on a remotely located object? For example, how can you prepare an invoice object if that object is on a Unix machine in another building? The following fundamental questions arise:

- How can you find (locate) the invoice object in the network?

- How can you send it a message to invoke the "prepare" method?
- How can you pass the needed parameters?
- How can you hide the complexity and heterogeneity of distributed systems from applications?
- How can you recover from the errors?

The distributed object middleware such as CORBA provides answers to these and several other related questions. Before getting involved with details about objects in distributed systems, let us quickly review some of the major concepts. Figure 5-1 shows a conceptual view that will be expanded and refined later. It consists of the following players that are common to all distributed object technologies:

- **Distributed Objects:** Application objects behave as clients, servers, or both.
- **IDL:** Objects define their services through an Interface Definition Language (IDL) that specifies the operations to be performed by remote objects. As we will see, CORBA, DCOM, J2EE, .NET, and Web Services all support IDLs.
- **Directories:** Object advertise their services through a directory that keeps the IDLs of all operations. Directories serve as "yellow pages" for the objects. As we will also see, CORBA, DCOM, J2EE, .NET, and Web Services all use directories to locate objects.
- **Object Brokers:** Object brokers allow objects to find each other in a distributed environment and interact with each other over a network. Object brokers are the backbone of distributed object-oriented systems. They essentially provide the "dot" between the object and method in distributed environments. As we will see, CORBA and DCOM use special brokers but .NET and Web Services use the Internet as a broker.
- **Object Services:** These services allow the users to create, name, move, copy, store, delete, restore, and manage objects. As we will see, CORBA and DCOM use special object services but .NET and Web Services use the Internet Services, as much as possible, for this purpose.
- **Object Invocation Protocols.** Protocols are needed by the brokers to access and invoke objects. As we will see, CORBA and DCOM use special protocols such as IIOP but .NET and Web Services use the Internet protocols such as HTTP for this purpose.

Table 5-1 illustrates the distributed object technologies used in CORBA and Web services. Inclusion of other distributed object technologies (DCOM, J2EE, .NET) to this table is left as an exercise for the reader. Keep in mind that modeling in terms of object-oriented (OO) concepts does not necessarily imply use of object-oriented programming languages such as C++ or object-oriented database managers. It is possible to view systems in terms of OO objects and then implement them in whatever technology makes sense. For example, most systems at present view data as objects but implement the data by using relational databases.

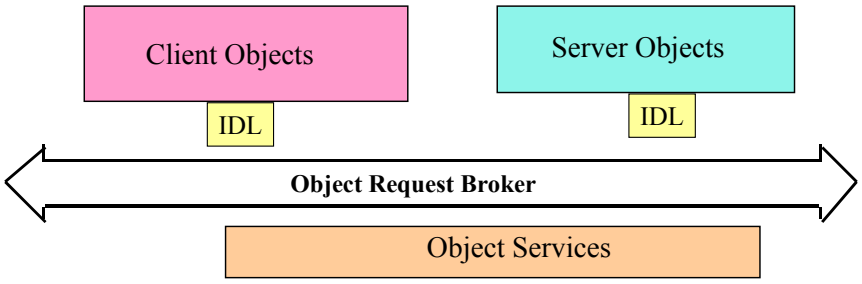


Figure 5-1: The Basic Distributed Objects Model

Due to the interest in object-oriented systems, the elegance with which complex distributed systems can be modeled by using OO concepts, and the appeal of OO technologies in developing new applications ranging from inventory control to network management, many attempts have focused on standardized middleware to support object-oriented distributed systems. The Common Object Request Broker Architecture (CORBA) from the Object Management Group is a prime example of such a standard.

Table 5-1: Distributed Objects in CORBA and Web Services

	CORBA	Web Services
Distributed Objects	CORBA Objects	Web Services Components
IDL	CORBA IDL	WSDL
Directory	CORBA Naming Services	UDDI
Object Broker	Object Request Broker	Internet
Object Services	CORBA Services	Web and Internet services such as DNS
Object Invocation Protocol	IIOP	HTTP, SOAP

5.1.2 Distributed Objects: From CORBA/DCOM to Web Services and J2EE

The trend at present is to extend the OO concepts to enterprise-wide distributed applications. Simply stated, distributed objects are objects dispersed across the network and accessed by users. Conceptually, we are talking about decomposing enterprise-wide applications into objects that can be dispersed around different machines on a network. An object on one machine can send messages to objects on other machines, thus viewing the entire network as a collection of objects. When, and if, fully realized distributed objects present a very powerful technology that has the potential of addressing many problems that have plagued the IT community for years (i.e., reuse, portability and interoperability). This is because applications constructed using reusable components that encapsulate many internal details interoperate across multiple networks and platforms.

Support of distributed object-based applications requires special purpose middleware that will allow remotely located objects to communicate with each other. A common mechanism used by such middleware is an object request broker (ORB) that receives an object invocation and delivers the message to an appropriate object. Examples of middleware for distributed objects include OMG's CORBA (Common Object Request Broker Architecture), DCOM (Distributed Computing Object Model) from Microsoft, and RMI (Remote Method Invocation) from Sun. Although, the exact implementations vary (as we will see), they all follow the conceptual view presented in Figure 5-2:

- Each remote object that wishes to provide a service defines its services through an IDL (Interface Definition Language). For example, purchasing and inventory are server objects advertising their services. Each server object specifies its service through an IDL (Interface Definition Language).
- The object request broker (ORB) is the main bus that connects object clients to the object servers.
- The clients use the ORB to locate and invoke needed services. The client uses IDL to invoke a service. For example, the client will use IDL1 for purchasing and IDL2 for inventory checking.

While object-orientation is quite fashionable, the attention has lately shifted to “components” and “business objects” that are large objects containing business functionalities. One of the main reasons for attention to component-based systems is that component-based platforms are commercially available at present. Examples of such platforms are Sun J2EE and Microsoft .NET that we will review later in this chapter (for more information about J2EE and .NET, see the Web sites (www.sun.com) and (http://msdn.microsoft.com), respectively). Figure 5-3 shows a conceptual component-based architecture platform that is a generalization of the Sun J2EE environment and the Microsoft .NET environment. The component-based architecture is composed of several components that can exist at the following tiers:

- Client-tier components run on the client machine.

- Web-tier components run on the Web server to provide server side support.
- Business-tier components run on the business tier and are the "business components" (we will discuss business components later).
- Enterprise system tier software runs on the back-end systems.

These component-based architectures, as we will see, use the distributed object middleware technologies such as IDLs.

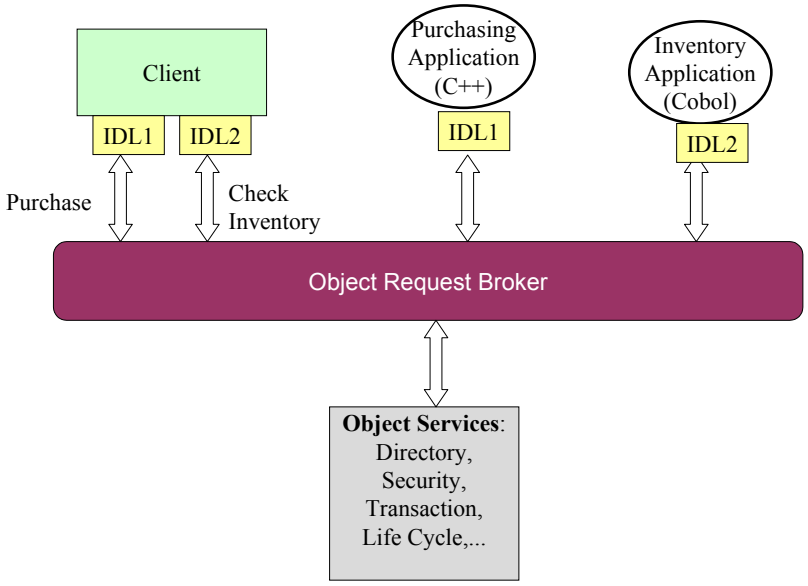


Figure 5-2: Conceptual view of Distributed Object Middleware

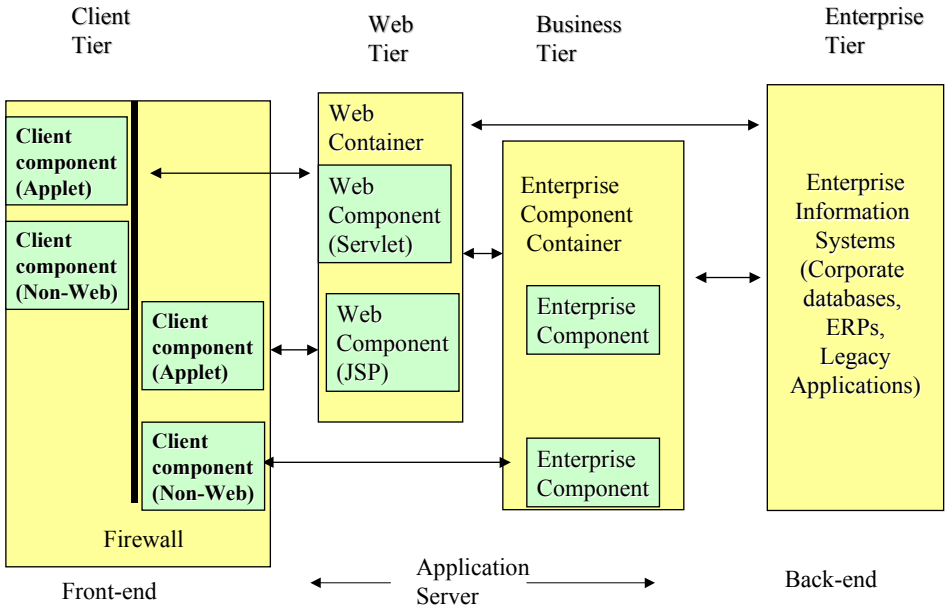


Figure 5-3: A Generalized Component-based Architecture

5.1.3 Interface Definition Languages (IDLs)

5.1.3.1 Interfaces and Interface Definition Language

Interfaces and interface definition languages (IDLs) are at the core of distributed object applications, including the new component-based applications. Simply stated, an interface specifies the API that the clients can use to invoke operations on objects. In particular, an interface describes:

- The set of operations that can be performed on an object.
- The parameters needed to perform the operations.

For distributed object applications, interface definitions are used to advertise the set of operations that an object can provide to prospective clients. Thus, the object's data is accessible only through the interface. Consequently, any server that is encapsulated by its interface can be viewed as an object. Figure 5-4 shows an interface of a simple inventory object that supports two operations: query inventory and update price (a definition of this interface is given in Table 5-2). These operations can be invoked by client programs.

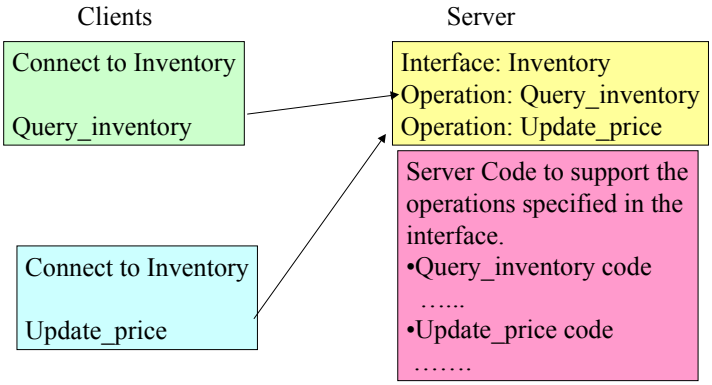


Figure 5-4: Example of an Interface

One or more interfaces may be defined for an object. For example, given an object such as inventory, you may need to define three different interfaces: one for the manager of inventory (i.e., to create, monitor and replenish the inventory), one for the order processing system (i.e., to retrieve and update the inventory), and one for the authorized customers (i.e., to browse through the available products). It is a good practice to design interfaces for different classes of users and to group related operations together. An interface definition includes some or all of the following:

- The interface header that shows an interface name and interface header attributes that uniquely identify the interface. Examples of such attributes are UUIDs (universal unique identifiers) and version numbers of the interfaces.
- Constant and data type definitions that are used to specify data properties (e.g., size) so that clients and servers can exchange data conveniently between different machines.
- A set of operations (methods) and the signatures for each operation. A signature specifies the operation's name, it's arguments, and argument types.

The interfaces are defined by using an interface definition language (IDL). Different middleware products provide IDL compilers that parse the IDL and produce header files and code segments that are used by the client and server programs (we will discuss this in more detail later). For example, CORBA, DCOM, and RMI all provide IDL compilers. In addition, middleware products support utilities and commands to store and retrieve IDLs from interface repositories. Client application developers can browse through these interface repositories to learn about the available server objects and determine the type of operations that can be invoked on an object.

IDLs are declarative languages -- they do not specify any executable code. IDL declarations (e.g., syntax, character types allowed, argument coding, etc.) must conform to the vendor provided IDL compilers. After you create the interface definition using IDL, you compile the IDL file to create header

files and "stubs" that are used in building clients and servers. Table 5-2 shows the IDL of a simple inventory system that supports query inventory and update price operations. The syntax used in this example is abstract. We will see actual IDL specifications for CORBA later on in this chapter.

Table 5-2: A Sample Abstract IDL

```
uuid 008B3C84-11c7-8580), version (1.0) /* Header */
interface inventory /* interface name is inventory */
query_inventory (/*The  operation to query inventory */
in char item_id; /* input is item id */
out integer on_hand; /output is on hand */
out integer status )  /*  output status */
update_price ( /*The operation is to update price */
in char item_id;  /* input parameter is item -id */
in integer new_price; /* input: new price */
out integer status )  /*output parameter */
```

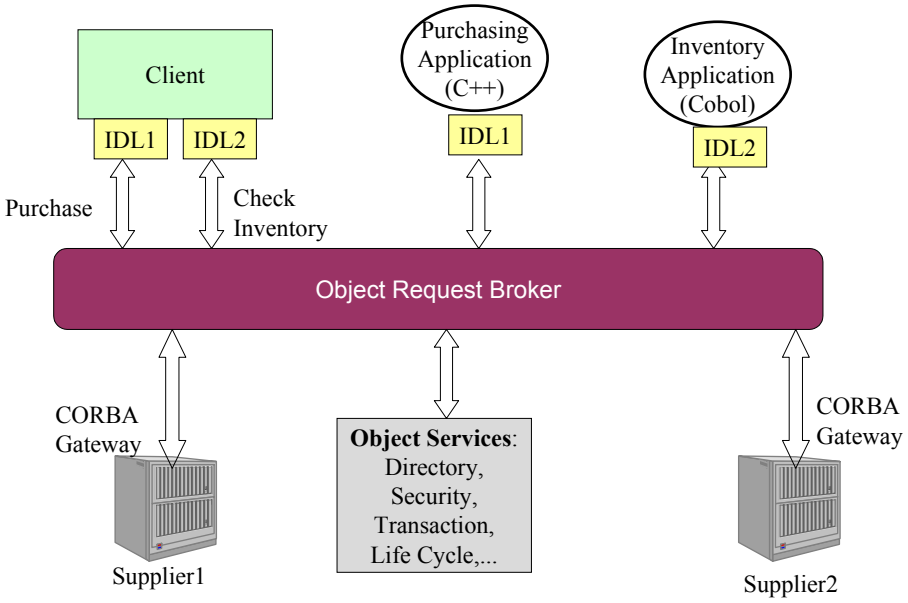
5.1.3.2 Types of Interfaces

Interfaces and interface definitions are of two kinds: operational interfaces which contain a set of named operations (i.e., procedures or methods); and stream interfaces, in which communication is organized as a set of linked directional flows. This chapter will focus primarily on operational interfaces. Stream interfaces are used to support distributed multimedia systems. Stream interfaces are used to describe unstructured communications such as voice and video streams in multimedia systems. Stream interfaces can also be used for electronic mail. The basic characteristic of stream interfaces is that they must support continuous data transfers over relatively long periods of time, e.g., real-time playout of video from a remote surveillance camera. In addition, the timeliness of such transmissions must be maintained for the duration of the media presentation. We will pick up the discussion of stream interfaces in a later chapter when we discuss distributed multimedia applications.

Interfaces significantly impact the design of distributed object applications. First, the server object is required to implement at least the operations specified in the IDL, and the client is required to accept at least the set of results generated by the IDL operations. This implies that the server will never respond with a "method/procedure not supported" message to a prospectus client. Second, the clients may not use some of the operations provided by the interface. This implies that the server interfaces can be extended to include more operations without requiring any change to the client. Finally, and most importantly, the "size" of the interface needs to be carefully examined. For example, it is not a good idea to specify an interface that supports 100 operations. In such a case, if one operation needs to be changed, then a new IDL will have to be compiled and all client programs will have to be recompiled (this could be quite irritating). It is best to design a separate interface for each group of users (e.g., one interface for end users, one for the system administrators, etc.).

Interfaces and IDL provide the basic glue for distributed object computing. IDL is used not only to define new services provided by objects, but also to "wrap" existing and legacy systems so that they behave externally as objects. For example, a legacy application written in Cobol could behave as a server object as long as it has an IDL and it provides the operations defined by the IDL. Thus, the "IDL-ized"

programs run on top of an ORB without revealing their internal details and work with CORBA gateways to work with non-CORBA systems (see Figure 5-5).



CORBA Gateways convert CORBA calls to non-CORBA

Figure 5-5: Interface Definition Language (IDL) in Action

5.1.4 Components, Business Objects, and Object Frameworks

A very wide range of OO products are commercially available from a diverse array of suppliers. The following three are the most significant because they are at the foundation of J2EE and .NET:

- Components
- Business Objects (business components)
- Object Frameworks

Components and "Applets": Components are high-level "plug and play" software modules that perform a limited set of tasks within an application. Components are essentially objects, also known as "applets", that are recognizable by the users as small applications (i.e., they do not perform internal programming tasks such as initialize internal memory locations). For example, Microsoft Draw is an applet within Microsoft applications. This particular component is high level enough to perform end-user type functions (it draws boxes, arrows, circles, etc.). However, it is not a stand-alone application, it only works with other applications components. Microsoft has many software components for Windows environments that draw, produce charts, perform calculations, etc. Depending on the size of the application, the components may be small or large. Components are very much like objects; however, the emphasis is on recognition by users (many objects are oriented towards programming tasks). Thus, component software can be used as plug and play to build complete applications. Many desktop tools are currently becoming available as components. Examples of typical desktop components are spell—checkers, SQL query builders, print managers (conceptually, each icon on your toolbar can be a separate component).

Business Objects (Business Components): The basic idea of business objects, also known as business components, is that the users can construct large objects that represent the real-world concepts of the business world. Examples of business objects are customer, order, products, and

regional office. Business components are larger than components -- components can represent a clock or a calendar but a business component can represent an inventory system. If software could be structured around such objects and other business concepts, then organizations would be able to build software that simulated current business strategy. Moreover, businesses could reuse these objects to build new applications by using the OO paradigm. Business objects started appearing in the marketplace around 1994 when OLE 2.0, OpenDoc, and CORBA-based products started emerging. Since then, OO tools designed to support creation of business objects have appeared from vendors such as Easel and applications that employ business objects have appeared from Sun, IBM, and Microsoft. Business components are at the foundation of "Business Component Factory" [Herzum 2000]. We will discuss business objects in detail in the Architecture Module.

Object Frameworks: Object frameworks are essentially descendents of object-oriented class libraries. Class libraries are collections of predefined object classes that define commonly used presentation, business processing logic, and data management structures and methods. A framework simply defines how given sets of classes are related and arranged for different applications. It is possible to think of frameworks at three levels: foundation classes, middleware frameworks, and application frameworks (or high level frameworks). The foundation classes provide fine-grained data and control statements, I/O functions, GUI structures, memory management functions, and database access functions. The middleware framework covers an extensive set of C/S middleware services such as transaction processing, database access, directory services, telephony, authentication and systems management. The application frameworks, also known as desktop frameworks, provide programmer productivity tools for compound documents, multimedia, groupware, mail, 3D graphics, and decision support applications. An example of commercial object frameworks is the IBM and Taligent object frameworks strategy that is beginning to materialize as products. The objective of this strategy is to create “seas of objects” with object frameworks as the glue to tie these objects into applications. Examples of other players in this market are ParcPlace Systems, Rogue Wave, ILOG, Next, Sun Microsystems, Easel, and others.

These, and other emerging technologies and market segments are leading towards reusable software that can be assembled to quickly build new applications. However, these technologies are introducing new terms and jargon. Due to an ever-growing list of object-oriented “things”, many groups are trying to figure out what to do. An example is the Object Management Group (OMG) that has been formed as a non-profit consortium of more than 500 software and systems manufacturers and technology information providers. OMG is specifying a set of standard terms and interfaces for interoperable software by using the object oriented concepts.

Object-Oriented Databases

Object-oriented databases allow storage and retrieval of objects to/from persistent storage (i.e., disks). Object-oriented databases, also known as object databases, allow you to store and retrieve non traditional data types such as bitmaps, icons, text, polygons, sets, arrays, and lists. The stored objects can be simple or complex, can be related to each other through complex relationships, and can inherit properties from other objects. Object-oriented database management systems (OODBMS), which can store, retrieve and manipulate objects, have been an area of active research and exploration since the mid 1980s.

Relational databases are suitable for many applications and SQL use is widespread. However, it is not easy to represent complex information in terms of relational tables. For example, a car design, a computing network layout, and software design of an airline reservation system cannot be represented easily in terms of tables. For these cases, we need to represent complex interrelationships between data elements, retrieve several versions of design, represent the semantics (meaning) of relationships, and utilize the concepts of similarities to reduce redundancies.

OODBMSs and RDBMSs both have their strengths and weaknesses. For example, RDBMSs are very mature and heavily used but cannot handle complex objects well. OODBMSs, on the other hand lack

the maturity and ease of use offered by the RDBMSs. A compromise, known as Object-Relational Databases, provides a hybrid solution where relational and object-oriented technologies are combined into a single product. Different vendors use different approaches to Object-Relational Databases. For example, Odaptor from HP uses an underlying relational database with OO front-ends while UniSQL from UniSQL is an OO database that subsumes the relational model.

Time to Take a Break
- ✓ Distributed Object Technologies
  - CORBA and DCOM
  - Web Services, .NET, J2EE
  - SOAP and EJBs

Suggested Review Questions Before Proceeding

- What are distributed object technologies and why are they important?
- What are the unique features of distributed objects that cannot be found in simple client/server systems?
- What is the role of IDL in distributed object technologies?
- What are the common features of all distributed object technologies?
- What are components and how are they different from objects?

5.2 Common Object Request Broker Architecture (CORBA)

5.2.1 Object Management Architecture

Common Object Request Broker Architecture (CORBA) is a specification proposed by the Object Management Group (OMG) - a non-profit industry consortium formed in 1989 by eight companies with the following goals [Soley 1994]:

- Solve problems of interoperability in distributed systems by using object technology.
- Use de facto standards in object technology and commercial availability of technology.
- Create a suite of standard languages, interfaces and protocols for interoperability of applications in heterogeneous distributed environments.
- Build upon, not replace, existing interfaces.

HP, IBM and Sun were among the original eight members of OMG, which has exceeded more than 800 members ranging from hardware vendors to end users. Interestingly, OMG was formed before any major products were introduced. Most standards bodies are formed to develop standards after products

are already in use. For example, ISO/OSI Reference Model for networks was introduced in 1977, almost 5 to 7 years after the introduction of SNA, Decnet, and TCP/IP.

Keep in mind that *OMG produces specifications, not implementations*. Implementations of OMG specifications can be found on over 50 operating systems. OMG solicits new specification proposals through RFI (request for information) and RFP (request for submission) process. Like other standards bodies, the proposals go through a formal evaluation, revision, review, recommendations and approval process.

OMG's first attempt at meeting its goals resulted in an Object Management Architecture (OMA), released in 1990 It was revised in 1992. OMA specified the overall object model for distributed object computing environments, including how objects are defined and created, how client applications invoke objects, and how objects can be shared and reused. The four components of this Management Architecture are (see Figure 5-6):

- **Application Objects**: These are business- aware objects specific to end-user applications. These objects can be pieces of data, software, and user artifacts that can reside on one or many machines. The application objects may be created by an OO language or encapsulated by using a "wrapper" around old systems. Applications are typically built from a large number of basic object classes.
- **Object Request Broker (ORB):** ORB is responsible for communication between objects. ORB finds an object on the network, delivers requests to the object, activates the object (if not already active), and returns any messages back to the sender. ORB is the backbone of OMA. We will discuss ORB in more detail later.
- **Object Services**: This component supports the request broker by providing services that almost every object needs. These include basic services (finding and invoking objects), thread services (create and manage threads), object life cycle services (create, destroy objects), and naming services (facilitate portable names). Additional services such as event, trading, transactions, and persistence have been added to CORBA services.
- **Horizontal Facilities**: These facilities were initially intended for common services such as user interface, task management, and information management. Examples include e-mail, database access, and compound. However, OMG has found that it was difficult to differentiate between these and CORBA services. At present, OMG has given up on these services.
- **Vertical (Domain) Facilities**: These facilities define the object models and IDLs for a very wide range of industry segments. This is one of the most active areas of OMG with work continuing in domains such as finance, business objects, healthcare, manufacturing, electronic commerce, telecommunications, transportation, and utilities.

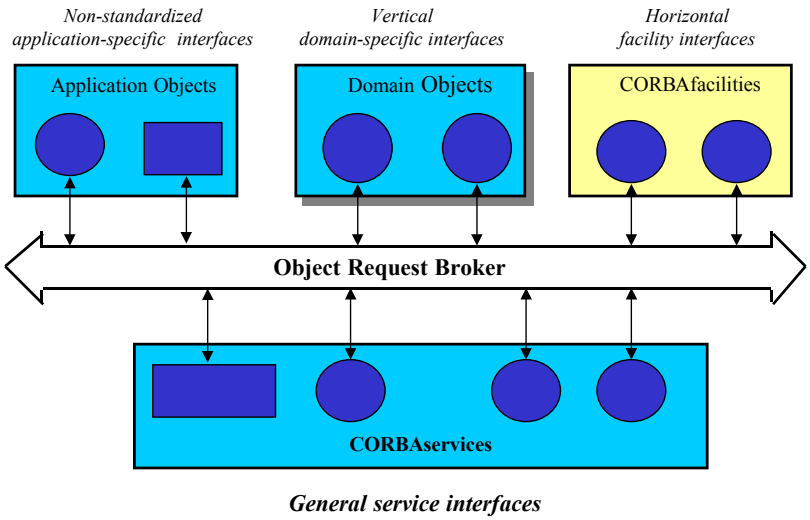


Figure 5-6: Object Management Architecture

It is important to note that the Application Objects, Vertical Facilities, and Object Services are simply categories of objects. Every piece of software in the OMA model is represented as an object that communicates with other objects via the Object Request Broker. These objects were grouped into three broad categories to ease the standardization process.

It is perhaps appropriate at this point to briefly discuss the OMG organization and structure. As shown in Figure 5-7, the OMG consists of an architecture board that oversees the developments of OMG, a Platform Technology Committee that develops the ORB and the Object Services, as well as the Domain Technology Committee that manages the growing vertical industries. Keep in mind that OMG, although well known due to CORBA specification, is actively involved in all aspects of object technologies. For example, OMG has specified the very popular UML (Universal Modeling Language) that is used to represent the models throughout the development of object-oriented systems (see Appendix B for more details on UML). OMG is also developing standards for business objects and component technologies. The adoption process consists of following steps:

Step 1: RFI (Request for Information) to establish range of commercially available software.

Step 2: RFP (Request for Proposals) to gather explicit descriptions of available software. The Architecture Board approves RFPs.

Step 3: Letters of Intent to establish corporate commitment from respondents to the RFPs (this is to assure that the RFP respondents will commit resources to implement technologies).

Step 4: Task Force evaluation & recommendation of the RFPs. Depending on the submissions and subject area, this may be a long, usually a year or more, process. There is also a simultaneous evaluation by Business Committee.

Step 5: Architecture Board consideration for consistency.

Step 6: Board decision based on recommendations from the appropriate Technology Committee & Business Committee.

A fast track process that bypasses some of the steps is also supported to introduce standards quickly. More details about OMG can be found at www.omg.org.

It is perhaps appropriate to comment on OMG's business model. OMG operates in a manner similar to a "non-profit Microsoft" instead of the typical standards bodies such as ISO and ITU. Most CORBA standards go into production within few years, sometimes months (as noted above, a submission to OMG must be accompanied by a "letter of intent" stating the intention to develop a commercial

implementation of the standard within one year.) OMG standards rarely starve for vendor implementations because it is a forum in which vendors ask users to set requirements, and users ask vendors for new features.

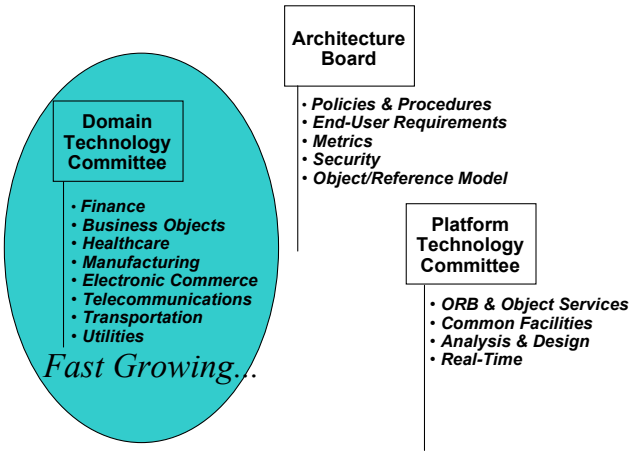


Figure 5-7: OMG Organization

5.2.2 Basic CORBA Concepts

CORBA was introduced in 1991 by OMG to go a step beyond OMA to specify the technology for interoperable distributed OO systems. CORBA specifications represent the ORB technology adopted by OMG and are published as OMG documents.

The key concepts of CORBA are (see Figure 5-8):

- Any object can be a client, server or both. For purpose of description, CORBA uses the C/S model where clients issue requests to server objects. The server objects are called "object implementations" or "servants" because they are the implementations of the invoked object.
- An interface, described in the OMG/ISO IDL (Interface Definition Language), represents contracts between client and server objects. The IDL shows the methods and the parameters being passed through the interface and is the only means of communication between clients and server objects. CORBA requires that every object's interface be represented in OMG IDL. Clients only see the object's interface, never it's implementation. Thus, as long as the interface is the same, you can substitute another implementation of the object – this is intended for plug and play. Program stubs and skeletons are produced as part of the IDL compiling.
- CORBA essentially specifies the middleware services that are used by the objects. A key part of this middleware is devoted to locating a server object, invoking the needed method on the object and returning results. A variety of other services such as naming, threading, lifecycle and event services are also provided.
- All interactions between CORBA objects are mediated by the ORB – clients cannot invoke server objects directly and server objects cannot respond to the clients directly. This requirement transfers all responsibilities to the ORB. Thus, ORB is the "Master. It finds the objects you need, wakes them up, gets them to work on your behalf to produce the results you need, and delivers the results back to you. In addition, if there is a failure, ORB is responsible for failure recovery. ORBs also hide the underlying implementation and system heterogeneity details. What a life it would be without ORBs!

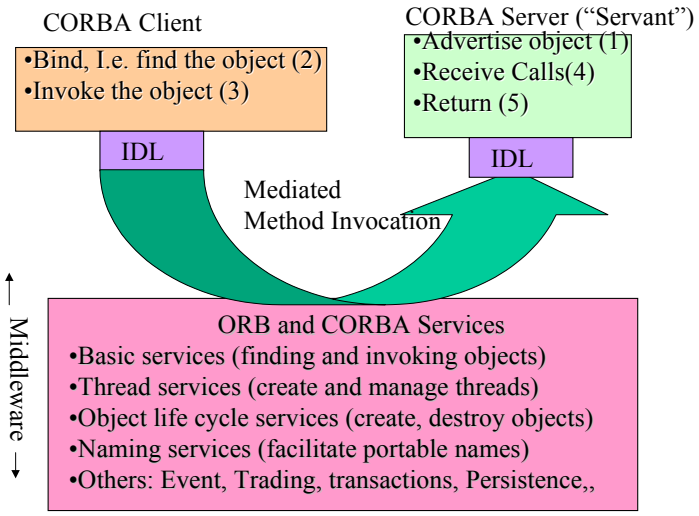


Figure 5-8: A Simple Example of CORBA

Let us now go through a simple flow of Figure 5-8 (we will discuss variants of this model later). First, the server object's interface has to be defined in IDL and the client has to write the code that invokes the server object through this IDL. At run-time, the following steps take place (the step numbers are indicated in the Figure):

1. The server object is started (this is one of the many options as we will see later). It advertises its availability to the ORB that records in its directory the location of the object. The server object then indicates that it is ready to receive calls and waits.
2. The client object issues a "bind" command for the server object (i.e., find server object). The ORB receives this call, locates the server object and returns an object reference to the client.
3. The client uses this object reference to issue the actual call to the server object.
4. The ORB passes this request to the server object that receives the request and processes it.
5. The server object returns the results to the client through the ORB.

The ORB invokes the various services to accomplish this interaction – more about it later.

5.2.3 CORBA Facilities

As defined by the OMG, the ORB provides mechanisms by which objects transparently interact with each other. It enables the objects to establish connections, communicate with one another, make requests, and receive responses. To achieve this, ORB sets up communications links and routes information between objects as needed. It literally provides brokerage services between clients and servers by determining the most efficient way for a client to receive a service and for a server to provide the service. The interfaces to the ORB and the interfaces to the objects built using the ORB are well defined. The underlying implementation of the ORB is not important to the developers building distributed object-oriented applications. Different interfaces can be defined for an object and multiple ORBs can exist in a system. Thus, different client applications can refer to the same object residing on a server but each client application can be given its own interface.

Figure 5-9 shows the architectural components of CORBA and the interfaces/flows between various components. The arrows indicate whether the ORB is called or performs an up-call across the interface. Before discussing the components, let us review some underlying concepts:

- **OMG Interface Definition Language (IDL)**: CORBA is built around a single object model embodied in IDL and uses a single specification language, the OMG Interface Definition Language (IDL), to specify the services provided by an object. The parameters needed for each task are specified and compiled using an IDL compiler. Once again, IDL is pure specification, not implementation and enables platform independence. The OMG IDL standard has been stable since 1991. OMG IDL syntax resembles C++. IDL compilers are provided by vendors. CORBA API is also defined in IDL. The IDL itself does not say anything about implementation of an interface, however all CORBA products generate bindings in languages such as Java, C, C++, and Smalltalk. IDL definitions are stored in a public interface directory. Access to this repository can be controlled through access control lists (ACLs). CORBA also specifies language bindings, i.e., mapping of IDL constructs to programming languages.
- **Synchronous and Asynchronous Support**: CORBA ORBs initially supported synchronous (i.e., client process is blocked until a reply from the server process is received by the client process) as well as delayed synchronous (i.e., the client process continues work after initiating a request and periodically polls the ORB for the response) communications. The delayed asynchronous model was not used very heavily because it was associated with dynamic binding – a nice but rarely used facility of CORBA (see below). CORBA 3.0 includes messaging services that support a variety of asynchronous communications. CORBA 3.0 also supports publish/subscribe model through its event/notification services. We will discuss CORBA 3.0 in Section 5.4. The applications can use synchronous, asynchronous, or publish/subscribe approach depending on application requirements.
- **Static Versus Dynamic Binding**: CORBA ORBs allow static as well as dynamic binding between objects. Dynamic binding between objects uses run-time identification of objects and parameters. In one sense, the static and dynamic binding of CORBA is similar to the static and dynamic binding used by SQL. From a developer's point of view, each serves the same purpose (i.e., establishes a link between a client and server object for invoking an operation). However, a developer chooses between these two options depending on how much information is available at compile time. The static binding between objects is based on compile time specification of objects and parameters. Static binding, initially proposed by HP and Sun, is more efficient at run-time because all needed libraries are included at compile time. It is also quite simple (a C call with parameters which specify the object to be invoked, the environment of the object, and any other values needed by the server object). On the other hand, dynamic binding between objects uses run-time identification of objects and parameters. Dynamic binding, initially proposed by DEC (Digital Equipment Corp.), incurs more overhead at run-time but is very flexible (it can be used when some of the information needed to complete an operation is not available at compile time). Dynamic binding needs extensive run time support (i.e. a repository that can be accessed at run time to locate objects). It is particularly useful for applications that are undergoing rapid changes or for a tool to support interactive browsing. The client can use the Dynamic Invocation interface or an IDL stub (determined at compile time).
  Editorial comment about dynamic binding: Dynamic binding, although quite appealing in concept has been used very rarely in real life situations. Due to its low use, Minimal CORBA, (a skinny version of CORBA for small applications) has excluded the dynamic feature.
- **General Inter-ORB Protocol (GIOP) and IIOP (Internet Inter-ORB Protocol):** The GIOP is specially designed for ORB-to-ORB communications. It is intended to operate over any connection-oriented transport protocol. Many mappings of GIOP have been specified. The best known is the Internet Inter-ORB Protocol (IIOP) that specifies how GIOP messages are exchanged over a TCP/IP network. IIOP allows a lightweight implementation of CORBA so that CORBA can operate directly on top of TCP/IP and is a required feature of current CORBA implementations.

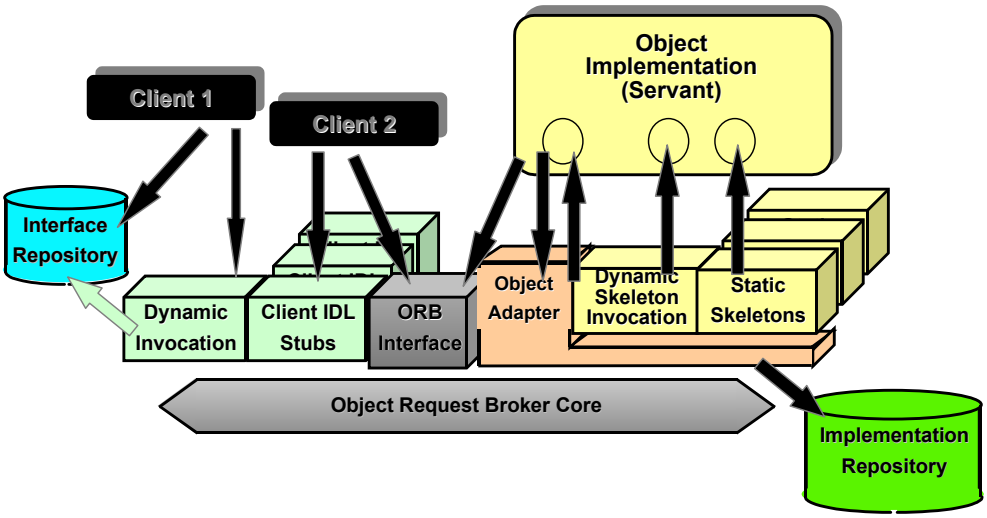


Figure 5-9: CORBA Architectural Components

Let us now go through the CORBA architectural components shown in Figure 5-9 and discuss how they interrelate with each other. Let us start from the left.

Interface Repository: A dynamic representation of available object interfaces is provided in an Interface Repository. This repository represents the interfaces (or classes) of all objects in the distributed environment. The clients access the Interface Repository to learn about the server objects and determine what type of operations can be invoked on an object. With CORBA 2.0, the Interface Repositories provide global identifiers to uniquely and globally identify a component and its interface across multivendor ORBs. This is accomplished through repository IDs. A repository ID is a unique, system-generated, string that is used across Interface Repositories. You can generate repository IDs by using the DCE Universal Unique Identifiers (UUIDs) or via a user-supplied unique prefix that is appended to IDL generated names.

Dynamic Invocation Interface (DII): This interface allows dynamic construction of object invocation. The interface details are filled in by consulting with the Interface Repository and or other run-time sources. By using the dynamic invocation, Client Application 1 can interact with Server Objects (provided descriptions of these Server Objects could be found in the Interface Repository). This component is rarely used.

Client IDL Stubs: The Client stubs make calls to the ORB Core. These precompiled stubs make it easier for the Clients to issue static requests to objects across a network. Client Application 2 uses this option.

ORB Interface: This interface goes directly to the ORB for operations that are common across all objects. This interface consists of a few APIs to local services that may be of interest to some applications. This interface is commonly used by a server object to tell the ORB that it is running and ready to accept calls. The client can also directly interact with the ORB for operations through this interface.

Object Adapters: An object adapter is essentially a scheduler that mediates between the ORB and the object implementations ("servants"). It is responsible for a) generating object references for the called servants, b) activation and de-activation of servants, and c) sending requests to servants. CORBA specifies that each ORB must support a standard adapter called the Basic Object Adapter (BOA). However, BOA was not well specified. To address these issues, CORBA 3.0 has introduced POA (Portable Object Adapter). We will discuss POA in Section 5.4.

Server IDL Stub: These stubs, also known as server skeletons, provide the code that invokes specific server methods. These stubs are generated as part of the IDL compilation and are very similar to the

client IDL stubs. They provide the interface between object adapters and the server application code. Server Objects use this stub.

Dynamic Skeleton Interface (DSI): The DSI, introduced in later versions of CORBA (CORBA 2.0), provides a run-time binding for servers that do not have IDL generated stubs. These dynamic skeletons can be very useful for scripting languages to dynamically generate server objects. When invoked, the DSI determines the server object to be invoked and the method to be invoked (the selection is based on parameters values supplied by an incoming message). In contrast, the server skeletons generated through compiled IDL are defined for a certain object class and expect a method implementation for each method specified in the IDL. The DSI can receive calls from static or dynamic client invocations.

Object Request Broker (ORB): ORB is obviously at the heart of CORBA. ORB acts as a switch in a CORBA environment — it sets up links between remote objects and routes the messages between objects. Any client object can make a request from a server object through the ORB and any server object can send responses back to the client objects through ORB. We will discuss ORB in more detail in the next section.

Implementation Repository: Implementation details of each interface, including the operating system specific information used for invocation, the attributes used for method selection, and the methods that make up the implementation are loaded into the Implementation Repository. The Implementation Repository can be implemented differently by different vendors. Some implementations of CORBA support IML (implementation mapping language) to describe the implementation details.

5.2.4 Using CORBA – An Example

Let us quickly review the overall process used in building CORBA applications to illustrate the key concepts. A more detailed example with code samples is given in Section ???). The activities involved in developing OO applications in CORBA environments involve the following major activities (see Figure 5-10):

- Create Interface definitions by using OMG IDL.
- Build the server (Object Implementation).
- Build the client application(s).
- Deploy the application.

Create CORBA Definitions: The main activity in this step is the creation of interface definitions in the CORBA IDL format by using a text editor. The IDL statements are compiled by using an IDL compiler. The IDL definitions can be kept in text files or stored in an Interface Repository so that the clients can learn about the server objects and determine what type of operations can be invoked on an object. As stated previously, the interface of an object is used to declare the operations supported by an object. It consists of a collection of the operations and their signatures, i.e., the operation's name, it's arguments, and argument types. For example, the following statements specify the interface for a bank account object in CORBA IDL (we have simplified it somewhat for illustrative purposes):

```
interface bank_account_interf /* interface name is bank_account_interf */
make_deposit  (/*The  operation is make_deposit */
in integer amount_deposited; /* input is amount_deposited represented as an integer */
in integer account_no; /* input is account_no */
out integer current_balalnce ) /*  output parameter is current_balalnce */
make-withdrawal ( /*Operation is  make-withdrawal */
in integer amount_withdrawn; /* input is amount_withdrawn */
in integer account_no; /* input is account_no */
out integer current_balalnce ) /*  output parameter is current_balalnce */
```

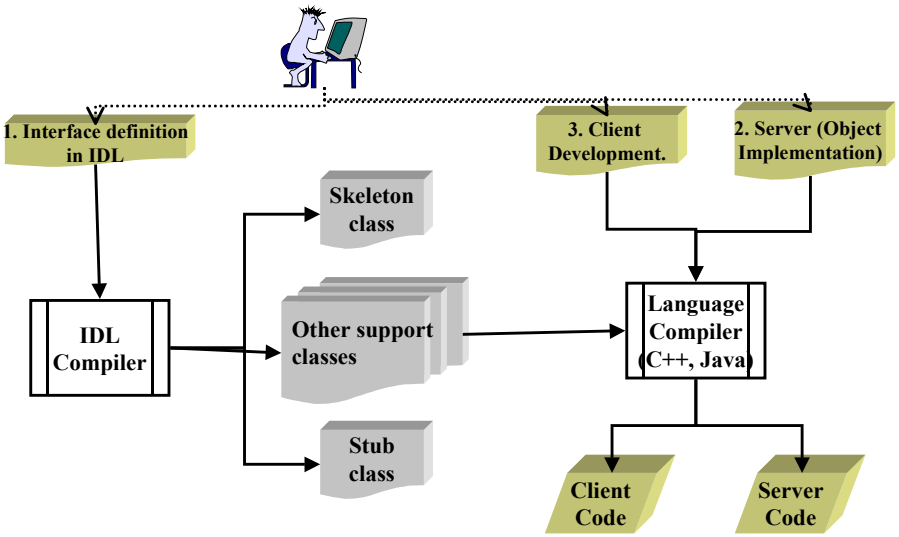


Figure 5-10: CORBA Application Development

The interface statement shows what operations can be performed on the order object. Each interface statement defines an interface and contains the descriptions of the operations (operation signatures).

In addition to IDL, the implementation details of each interface are created. You can use CORBA commands to generate a default implementation description from the interface definition. These implementations are loaded into the Implementation Repository.

After you create the interface definition using IDL, you compile the IDL file to create two very important components for building your application: the client stub and the server skeleton. The client stub and server skeleton are the code templates for building CORBA client and server programs (see Figure 7.8). The client stub is used only to build client programs that use static binding, i.e., the client code is linked with the client stub to form the client application. The client stub is not used to build client programs that use dynamic invocation. The server skeleton is always used as the framework for building the server application, regardless of the invocation type used by the client.

Build the Server (Implement the Object): CORBA servers can be quite complex and diverse. Building of the program servers requires the following steps:

- Generation of server skeletons - The IDL statements are compiled to generate a server skeleton. This compilation also uses the information contained in the IML files. The server skeleton contains method templates that show entry points for all of the implementation methods. The server skeleton also contains the server dispatcher code that makes the implementations and the methods known to the ORB (the dispatcher is called by the Basic Object Adapter). A registration routine is also generated as part of the server code (this routine is called at server start-up).
- Develop server initialization code - Each server initialization needs code to register the implementation, activate the server's implementation, enter a main loop to receive requests, and exit after un-registering and releasing resources. In addition, the server needs routines for creating objects and managing references to these objects. The server skeleton is used to develop this code.
- Develop the methods - The major activity in building a CORBA server is to write the code for the methods that execute the operations. For each method template, you must create the code for the methods. Methods can be implemented as executable code, calls to legacy applications, or scripts to integrate command line interfaces with existing applications.

Build Clients (Static): After a server has been built and registered, clients can be built to invoke the servers. As stated previously, CORBA clients use static invocations (i.e., clients know at compile time the objects and the operations on these objects) or dynamic invocation (i.e., the clients determine at

run-time the objects and the operations on these objects). We only discuss static invocation here. The main steps involved in building a static invocation CORBA client are a) generate client stub from IDL or from the interface repository, b) build the client code, and c) compile and link the client.

Deploy and Run the Application: CORBA applications can be packaged and shipped as server only, client only or a collection of clients and servers. To accomplish this, you need to send your IDL, implementation specifications, in addition to the executables. The application is installed and used in a CORBA run-time environment (see Figure 5-11). The IDLs and IML are loaded into the Interface and Implementation Repositories first. Then, the server is installed. At server start-up, it registers itself so that the invoking clients can locate it. The Dynamic Invocation Interface allows dynamic construction of object invocation. The interface details are filled in by consulting with the Interface Repository and/or other run-time sources. The Client IDL Stubs make calls to the ORB using interfaces and make it easier for the Clients to issue static requests to objects across a network. Object Adapters allow an object implementation to access the ORB services. CORBA specifies that each ORB must support a standard adapter called the Basic Object Adapter (BOA). Server Skeletons (Server IDL Stubs) provide the static interfaces to each service supported by the server.

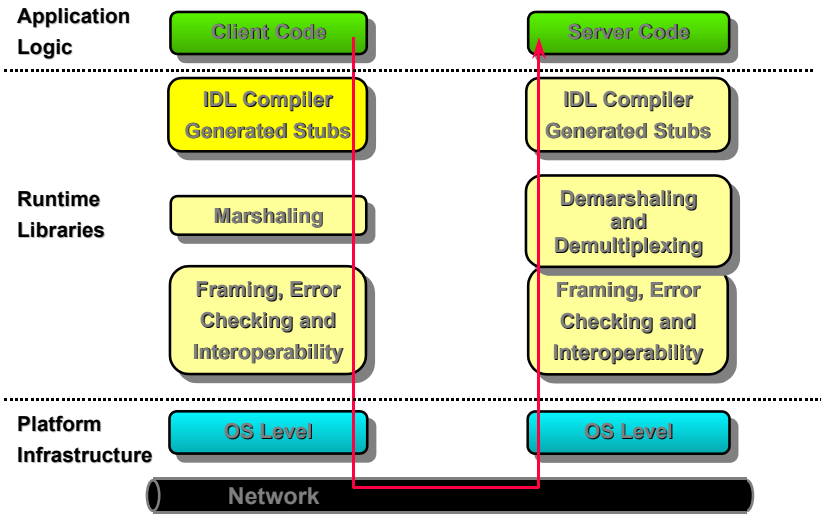


Figure 5-11: Information Flow in a CORBA Deployed Application

Script Servers

Some vendors support script servers that use operating system commands in a script (e.g., Bourne shell in UNIX), or command procedures. Building of script servers requires somewhat different steps. First, the implementation must indicate an "activation_type (script)" parameter. In addition, special techniques for handling input-output are needed because scripts are not interactive. Passing information from clients to the scripts also requires calls to special routines. Script servers also have several limitations such as data types, performance restrictions, and object creation. Despite several limitations, script servers are handy tools for quickly developing CORBA applications.

5.2.5 Combining CORBA with Web and XML

Many applications need to combine/integrate CORBA with Web and XML. Perhaps the oldest and the best known method is to invoke CORBA calls from a CGI gateway. Other approaches are:

- Invoke CORBA directly from the Web browsers (Netscape browsers can issue the CORBA IIOP calls).
- Use HTTP as a transport protocol underneath ORBs. A few small companies have implemented this option.
- Use CORBA to interact between Java applets across machines. This option is quite popular at present.

What about XML and CORBA - Well, an XML document can be used as a parameter of IDL. Thus, CORBA can be used as a transport mechanism for XML.

The main idea is to integrate CORBA with Web so that Web browsers can work directly with CORBA objects. Figure 5-12 shows a schematic for the most popular choice (i.e., through Java applets). The Figure shows the 3 major steps: 1) Download the Java applet from the server, 2) load the Java applet, and 3) invoke CORBA calls from the applet. In addition, non browser applications can also call the same server (i.e,, as long as the IDL of the CORBA server is known, the Java applet clients or any other client can call the CORBA server. The CORBA server can then invoke the back-end systems.

Please keep in mind that OMG has developed a standard called XMI (XML Interchange) that essentially uses XML as a means to exchange information between UML documents (i.e., XMI flattens UML and then passes it around to different UML sites). This could lead to other ways of integrating CORBA with XML.

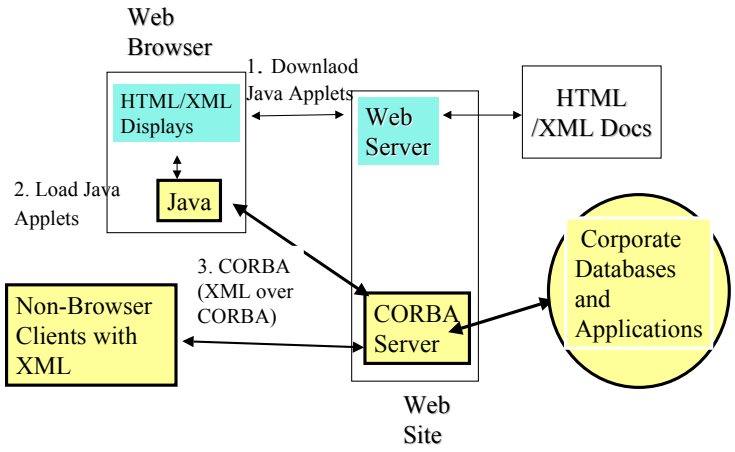


Figure 5-12; Combining CORBA with Web and XML

5.2.6 CORBA 3.0

CORBA has gone through several stages of evolution. It started with specification 1.0 and is at 3.0 at the time of this writing. Initial specifications of CORBA (i.e., CORBA 1.1 and 1.2) did not specify many important services such as security, concurrency, and transaction processing. This had two major consequences. First, the CORBA-based applications were limited in scope (e.g., they provided no security). Second, different vendors chose to plug different services to get going. This has led to CORBA implementations by vendors that do not interoperate with each other (recall that interoperability is the main goal of CORBA!). ORBs from different vendors interoperate with each other only through deliberate efforts and joint agreements between vendors.

CORBA 2.0, introduced in December 1994, addressed many problems by adding many new services. CORBA 2.0 expanded some ORB capabilities -- the Interface Repository expanded and a new interface (Dynamic Skeleton Interface) for servers was added. However, most of the new 2.0

capabilities added have been in the distributed object services of CORBA. Specifically, CORBA 2.0 specifies standards for the following object services (see Figure 5-15):

- Object Naming Service to allow different components to locate each other in a CORBA environment.
- Event Service to support notification to objects for different events.
- Persistence Service for storing components in data stores such as object databases, relational databases and flat files.
- Object Life Cycle Service for creation, modification, and deletion of objects.
- Transaction Management Service to support object-oriented transactions in distributed environments.
- Concurrency Control Service for obtaining and freeing locks.
- Security Service to protect components from unauthorized users.
- Time Service to provide universal timing service.
- Licensing Service to meter the use of components.
- Query Service to provide SQL and OQL (Object Query Language).
- Properties Service to associate properties (e.g., time and date) to components.
- Relationship Service to establish dynamic associations (e.g., referential integrity) between components.
- Externalization Service to get data in and out of a component in streams. This can be used in multimedia applications.

CORBA 2.0 also addressed the problem of ORB interoperability by defining inter-ORB protocols (IOPs) for interoperability of Object Request Brokers. Although the IOPs do not impact application software development (IOPs are too low level for applications), they play a key role for an overall middleware architecture.

CORBA 3.0 is the new kid on the block in the CORBA world. It is built upon CORBA 2.0 and 2.2 and has expanded many existing capabilities plus added new ones. Examples of CORBA 3.0 capabilities are:

- CORBA Messaging
- Real-Time CORBA
- Objects-by-value (OBV)
- CORBA Firewall
- CORBA Component Model
- CORBA Scripting Language
- JAVA-to-IDL/IDL-to-JAVA
- DCE/CORBA Interworking

CORBA 3.0, due to its inclusion/expansion of existing and introduction of new services, is a comprehensive solution for distributed object computing. In fact, CORBA 3.0 has been used as an umbrella term to refer to a suite of specifications which, taken together, add a new dimension of capability and ease-of-use to CORBA.

We will discuss the main features of CORBA 3.0 in the Tutorial Module (see the Chapter “CORBA Technologies – A Closer Look”).

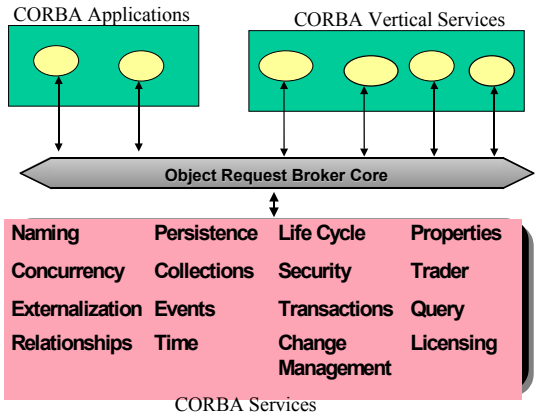


Figure 5-13: CORBA 3.0 Capabilities

5.2.7 CORBA Summary

Object-oriented technologies and techniques have natural applications in distributed systems. Entities in distributed systems can be viewed as objects exchanging messages. OMG was formed to create a suite of standard languages, interfaces and protocols for interoperability of applications in heterogeneous distributed environments. CORBA is an OMG specification for invoking objects in a distributed environment.

Initial specifications of CORBA did not specify many details. However, a wide range of capabilities have been added under the CORBA 3.0 umbrella. In addition, several Domain Task Forces and Platform Special Interest Groups (PSIGs) have been busily working on different aspects of CORBA. Examples of key CORBA developments in CORBA Services, distribution protocols, specialized models, vertical domain facilities, support for analysis and design, and basic object computing model are summarized in the sidebar "Key CORBA Developments".

Many vendors are announcing CORBA compliant software. Examples of a few products are:

- Inprise's Visibroker
- Iona's Orbix
- BEA's M3
- IBM's Object Broker
  - IBM (SOMobjects) - free for OS/2, AIX, Windows
  - PrismTech (OpenBase)
  - ParcPlace (Distributed Smalltalk)
  - TIBCO (ObjectBus)
  - ObjectSpace (Voyager)
  - Objective Interface Systems (ORBexpress) also marketed by ObjecTime
  - Bionic Buffalo
  - I-Kinetics
  - RogueWave (Nouveau)
  - Java RMI over CORBA IIOP

A wide range of free and prototype CORBA implementations are also available. For a list of such products, consult the Washington University at St. Louis Web site (www.wustl.edu/~schmidt/corba-products.html).

For a wide range of sources, including test tools, go to http://www.vex.net/~ben/corba/.

Additional details about CORBA are listed in the sidebar "Key Sources of Information for CORBA". The OMG home page (http://www.omg.org) is an excellent source of the recent activities in CORBA.

Although we have concentrated on the object request brokers so far, the concept of brokers is general and is currently being exploited in the message broker architectures (see the sidebar "Message Broker: Another Kind of Broker").

Key CORBA Developments

- **CORBA Services**
  - Naming - directory service: (svc name) → (svc object reference)
  - Trading - service discovery: (svc attributes) → (svc name)
  - Event/notification - Producers notify consumers using events
  - Transactions - distributed (2Phase Commit), flat transactional objects
  - Security - different levels (IIOP over SSL, additionally used)
  - Messaging - Support asynchronous processing for loosely coupled
- **Distribution protocol**
  - GIOP (Generalized Interorb Protocol) and its mappings (IIOP)
  - Mappings for SS7, ATM
- **Specialized Models**
  - Real-time, Fault-tolerant, Minimum CORBA
- **Vertical Domain facilities (e.g., Telecom)**
  - CORBA to TMN Interworking
  - Wireless CORBA
- **Support for Analysis and Design**
  - UML(universal Modeling Language)
- **Basic Object-Oriented Computing Model**
  - CORBA Components, ISO/OMG IDL and language mappings

Message Broker: Another Kind of Broker

A broker mediates between clients and servers (i.e., instead of a client directly connecting to a server, it first connects to a broker that in turn finds a suitable server). The concept of a broker is independent of the implementation of the broker. For example, the best known implementation of the broker architecture is the object request broker (ORB) as presented in OMG CORBA specification. In CORBA, the ORB mediates the interactions between remote objects. Another type of broker, called a message broker, is being presented as a viable implementation of the broker architecture.

A message broker is not restricted to objects. Instead, it delivers messages between disparate applications, including legacy applications. The underlying technologies used by the message broker may consist of RPCs or MOMs, although MOM does appear to fit this model quite well. The basic idea of a message broker is that it can provide brokerage services asynchronously and support a "publish/subscribe" model. The message broker can also be rule-based, i.e., you specify the rules to be used by the middleware to perform certain actions. Message brokers are at the core of “Enterprise Application Integration” platforms that are commercially available from many vendors such as Active Software, Vitria, and IBM. EAI platforms, based on message brokers, are expected to be a $11 Billion market by the early 2000s. We will discuss EAI platforms in a later chapter.

The Gartner Group proposed and advocated message brokers as key to the future success of distributed computing. The Gartner Group predictions in the mid 1990s that message brokers will be as widespread as database gateways and data warehouses have been correct. See, for example, Bort, J., "Can Message Brokers Deliver?", Applications Software Magazine, June 1996, pp.70-76).

Key Sources of Information for CORBA

- J. Seigal, "CORBA 3.0: Fundamentals and Programming", second edition, Wiley, 2000.
- D.Schmidt, *Overview of CORBA*: http://siesta.cs.wustl.edu/~schmidt/corba-overview.html.
- R.Orfali, D. Harkey, *Client/Server Programming with Java and CORBA*, Wiley 1997.
- T. Mowbray, W. Ruh, *Inside CORBA*, Addison-Weslet 1997.
- T. Mowbray, R. Zahavi, *The Essential CORBA*, Wiley 1995.
- T. Mowbray, R. Malveau, *CORBA Design Patterns*, Wiley 1997.
- J. Farley, *Java Distributed Computing*, O'Reilly 1997.
- The OMG Website: http://www.omg.org.
- "Corba Connections", Comm. of ACM, Special Issue, October 1998.
- http://www.vex.net/~ben/corba/ -- An interesting private Web site that watches CORBA ORB Core Feature Matrix, CORBA services Feature Matrix , and CORBA Vendor Platform Matrix.
- CORBA Business Objects (Workflow).
- Object Management Research in LASER (Pleiade).
- CORBA Web gateway.
- Joint Inter Domain Management.
- Towards a Web Object Model - by Frank Manola.
- Gabriel D. Minton's papers.
- CORBA Design Patterns - by Thomas Mowbray and Raphael Malveau.
- Alan Pope (The CORBA Reference Guide).
- Kate Keahey's Brief Tutorial on CORBA.
- Manfred Schneider's CORBA links.
- CORBA for Linux.
- Segue Software (was Black & White) (SilkPerformer, SilkMeter, SilkPilot, SilkObserver).
- Tom Valesky's Free CORBA page.
- Defense Information Infrastructure - Predicting CORBA Performance.
- Benchmarking some ORBs.
- GNOME - GNU Network Object Model Environment.
- Michi Henning.
- Dominique Benech's COBALT: A KQML-CORBA based Architecture for Intelligent Agents Communication.
- Live Script and Live Repository.

5.3 Microsoft's DCOM (Distributed Component Object Model)

5.3.1 Overview

In March 1996, Microsoft announced its "ActiveX" strategy that integrates the desktop services with the World Wide Web. DCOM (Distributed Component Object Model) serves as a core technology for remote communications between ActiveX components. In the late 1990s, Microsoft positioned DCOM and ActiveX as a complete environment for components and distributed objects in Microsoft environments. Almost everything coming out of Microsoft was based on ActiveX. Later, Microsoft expanded DCOM to COM+. At the time of this writing, COM+ is being phased out in favor of XML Web Services (discussed in the next section).

We briefly discuss DCOM and ActiveX because many current applications are based on this technology. Although ActiveX provides many capabilities, from a distributed objects point of view, the following features are significant (we will see the details in the following subsections):

- All ActiveX components communicate with each other by using DCOM. So a Java applet (an ActiveX component) can call a remotely located Microsoft Word document (another ActiveX component) over DCOM. See Section 5.3.2.
- The Web browser can behave as a container. For example, the Microsoft Internet Explorer can contain components such as Word documents, Java applets, C code, and Excel spreadsheets. See Section 5.3.3.
- Web technologies (browsers, HTML pages Java applets) can be intermixed with desktop tools (spreadsheets, word processors) for distributed applications. See Section 5.3.4.
- Serve facilities such as SQL servers and legacy access gateways can be invoked from ActiveX clients. See Section 5.3.4.

5.3.2 DCOM (Distributed Component Object Model) as an ORB

As discussed previously, ActiveX uses DCOM to provide communications between remote ActiveX components. In this sense, DCOM is the ORB for ActiveX. The basic scenario is that Windows will be a huge collection of ActiveX components and interfaces, with DCOM serving as the ORB. It is expected that all system services will be written as DCOM objects. These and other services can be provided by Microsoft or any third party vendors.

DCOM provides the basic brokerage services for ActiveX. It supports APIs for static as well as dynamic invocation of objects. DCOM uses DCE RPC for interactions between COM objects. DCOM's object model is somewhat limited because DCOM does not support multiple inheritance. In other words, COM supports inheritance through pointers that link different interfaces together. Figure 5-14shows the role of DCOM in ActiveX and Microsoft environments.

The following facilities of DCOM should be noted (see the sidebar "DCOM Versus CORBA" for additional discussion):

- **Interface Definition Language (IDL**): DCOM uses interfaces that are very similar, in concept, to the CORBA interfaces. An interface defines a set of related functions. The DCOM IDL is used to define an interface, the method it supports, and the parameters used by each method. DCOM IDL can be used to define your own interfaces, in addition to the Microsoft provided interfaces. For example, at the time of this writing, OLE/ActiveX consists of more than 100 interfaces, each supporting about 6 functions. Additionally, more than 100 Win32-style APIs are supported.
- **Object Definition Language (ODL) and Type Libraries**: DCOM supports an Object Definition Language (ODL) used to describe metadata. The interface specifications and metadata are stored in a repository, known as Type Library. Type Libraries are equivalent to the CORBA Interface Repositories.
- **Object Services**: DCOM provides very rudimentary object services at the time of this writing. Examples of the services provided are a basic licensing mechanism, a local directory service based on the Windows Registry, a basic life cycle facility, persistence services for file systems, and a very simple event service called connectable objects. This is in addition to the naming services provided by the Type Libraries. However, the overall ActiveX Platform is expected to support other services such as X.500 directories.

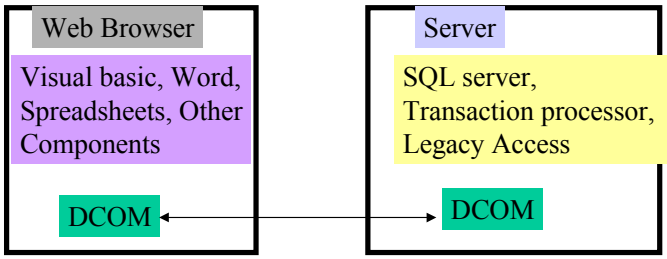


Figure 5-14: DCOM Conceptual View

DCOM Versus CORBA: Similarities and Differences

We will start with the typical disclaimer about technology comparisons, i.e., both technologies are evolving at the time of this writing and consequently the similarities/dissimilarities will also change with time. Our objective is to present, what appears to be, the philosophical and fundamental approaches being used by the two technologies.

At a high level, there are several similarities between CORBA and DCOM. However, several differences appear when you look closely.

Similarities:

- Both are based on the object model.
- Both utilize the interface concept and utilize an Interface Definition Language (IDL).
- Both use static and dynamic calls from clients to servers.
- Both use a repository to locate objects and invoke them (CORBA calls it the Interface Repository and DCOM calls it a Type Library).

Dissimilarities:

- CORBA is a specification but DCOM is an implementation.
- DCOM uses, in addition to IDL, Object Definition Language (ODL), for defining metadata. CORBA uses a single IDL for everything.
- DCOM uses the universal unique ID (UUID), based on OSF DCE, to locate and invoke objects. CORBA does not use UUIDs. It uses object references and repository to locate and invoke objects.
- DCOM uses the OSF DCE RPC as the basic transport mechanism between remote objects. CORBA uses several options such as IIOP (Internet Inter-ORB Protocol) that uses TCP/IP sockets and ESIOP (Environment Specific Inter-ORB Protocol) that runs on top of DCE.
- CORBA only uses connection-based (i.e, TCP) services while DCOM favors connectionless (i.e., UDP) services. DCOM does support TCP connections but it favors UDP for purpose of scaling (do not have to keep track of large number of open sessions).
- CORBA 2.0 has specified a very extensive set of services that include transaction management, security, concurrency control, life cycle, query, etc. In comparison, DCOM services at present are somewhat limited (these are being added through the ActiveX Platform).

Additional discussion about differences between DCOM and CORBA can be found in [Orfali 1996, Foody 1996]. The WWW Consortium held an excellent technical seminar on November 18, 1996, on trade-offs between DCOM and CORBA. Public information discussed in this seminar can be obtained from the Web site (http://www.w3.org).

5.3.3 Web Browsers as Containers of ActiveX Components

An ActiveX component is the basic unit of ActiveX applications. Different components can be combined to develop and deploy new applications. These components may be specifically written for this application or reused from some other project or even purchased off the shelf.

Components by definition cannot survive on their own — they require containers in which to execute. Visual Basic is a common example of a container in the desktop world. Visual Basic applications load needed components from a machine's local disk or a file server. In the Internet World, the Web browsers are a common example of a container — they load Java applets (Java applets are components) and provide an environment to run them (i.e., contain them). Let us focus on Web browsers as containers.

Before ActiveX, Web browsers were primarily serving as containers for Java applets. We have discussed Java applets in Chapter 4. These applets are downloaded from Web servers (embedded in HTML pages) and then the Web browser is used as a container. ActiveX has extended the scope of browsers as containers by allowing ActiveX components to be "contained" by Web browsers. At present, the Microsoft Internet Explorer is the main browser used as an ActiveX container. This can be very useful. For example, the Web browser can now contain spreadsheets, Word documents, and code written in C++, C, Java, or other programming languages. You can build powerful applications that may, for example, supply specialized viewers with the data to be viewed (the viewer and the data is loaded as needed from the network and runs inside the Web browser as a container).

5.3.4 ActiveX Controls – Building Downloadable Web-based Components

Microsoft's ActiveX Controls (formerly called either OLE controls or OCXs) are the special brand of ActiveX components that have been optimized for Internet use. ActiveX controls are, in principle, very similar to Java applets. For example, ActiveX controls, like Java applets, are self-contained pieces of functionality that run inside some kind of container (e.g., a Web browser). Thus, ActiveX controls can be embedded in Web pages and downloaded on demand. However, unlike Java applets, ActiveX controls can be written in various languages such as C, C++, and Java. Unlike Java applets, which are downloaded in a machine-independent format and usually interpreted within the browser, ActiveX controls are binaries. Another difference is that Java applets today are supported primarily by only one kind of container -- the Web browsers. ActiveX controls, on the other hand, are supported by different kinds of containers (e.g., Visual Basic applications).

Developers of downlaodable Web-based applications have two basic choices: Java applets or ActiveX Controls. See the sidebar "Java Versus ActiveX Controls" for discussion.

A plethora of different ActiveX controls already exist in the marketplace. Examples are the controls that implement spreadsheets, data viewing, mainframe connectivity, voice recognition, and the like. Many of these existing controls can be downloaded and executed within an ActiveX-capable browser. Thus, there is an instant supply of available ActiveX components for Web-based applications.

5.3.5 ActiveX Server

The ActiveX Server is based on the Microsoft Information Server (IIS) that is integrated with the Windows NT network operating system. The ActiveX Server includes the Microsoft BackOffice family that includes the Microsoft SQL Server and the Microsoft Systems Management Server. ActiveX Server provides scripting and control facilities to tie into legacy systems or to perform other specialized functions on the server side. The scripting capabilities support PERL, JavaScript and Visual Basic Script.

5.3.6 General Observations and Comments

The facilities of ActiveX have evolved over the years. However, competitors to ActiveX such as CORBA have also matured considerably in the same time period. At present, ActiveX can be combined and "bridged" to CORBA and other technologies. DCOM to CORBA bridges are available from companies such as Iona at the time of this writing. The Iona COM/CORBA bridge provides two-way mapping: it allows DCOM objects to be treated as CORBA and vice versa.

Literature on ActiveX keeps growing. The Microsoft Web site (http://www.microsoft.com) provides access to latest announcements, white papers, and frequently asked questions (FAQs). The book by David Chappell, "Understanding ActiveX and OLE", Microsoft Press, latest edition,is a good overview of the subject matter.

Java Versus ActiveX Controls

Java applets and ActiveX Controls are two valid choices for building downloadable Web applications. The leading browsers, Netscape Navigator and Microsoft's Internet Explorer, support both options. Let us discuss the choice between these two options.

Java applets should be chosen if a component must run on heterogeneous client systems, if the Java security exposures are manageable, and if you are not concerned with the performance limitations of the Java interpretive model (interpreters can be slower than binary code).

ActiveX Controls should be chosen if the component is targeted at Microsoft systems, is needed in a wider range of containers than just Web browsers, and must run as efficiently as possible (ActiveX Controls download binary code).

As expected, both of these models will evolve. For example, "just-in-time" compilers for Java will improve the performance by compiling an applet byte code on arrival. The platform independence issue may disappear because Microsoft is planning to port ActiveX on multiple platforms. Keep in mind that ActiveX Controls also support Java applets (Java environment is modified so that it uses DCOM).

Source: Chappell, D., "Component Software Meets the Web: Java Applets vs. ActiveX Controls", Network World, May 1996.

Combining Distributed Objects with the Web - Let Me Count The Ways

There are several ways to combine distributed objects with Web. Here are the principal ones that use CORBA, OLE/ActiveX, SOAP, and others).

The CORBA route:

- Invoke CORBA calls from a CGI procedure (a script or a subroutine written in C or in any other language) that resides on the Web server. In this case, the CGI procedure is the CORBA client. This is the oldest and most well-known method.
- Invoke CORBA directly from the Web browser. Netscape browsers are beginning to support the CORBA IIOP calls directly. Thus, the Web browser sites behave as CORBA clients.
- Use CORBA to interact between Java applets across machines. This option is currently supported by a few vendors (like Sun).

The ActiveX/DCOM route:

- Invoke DCOM calls from a CGI procedure (a script or a subroutine written in C or in any other language) that resides on the Web server.
- Invoke DCOM calls directly from the components contained in the Web browser (e.g., the Microsoft Internet Explorer). These components may be written in C, C++, Visual Basic, Java or other

programming languages behaving as ActiveX Controls and contained inside the browser.

- Invoke DCOM calls from the ActiveX components such as spreadsheets that may invoke Java applets or other components residing on Web servers.

Other routes:

- Use the Sun Remote Method Invocation (RMI) between remotely located Java applets. This technology is very well supported by SunSoft tools but is only restricted, at the time of this writing, to interactions between Java applets only.
- Use HTTP to invoke remote objects. A few small companies have implemented this option by using HTTP underneath ORBs. This option should be used rarely, if at all. We are mentioning it for completeness.

The SOAP Route – The New Way:

- Use XML Web Services and SOAP, the new kid on the block. SOAP is a lightweight protocol for accessing objects over HTTP.

Time to Take a Break

- ✓ Distributed Object Technologies
- ✓ CORBA and DCOM
- Web Services, .NET, J2EE
- SOAP and EJBs

Suggested Review Questions Before Proceeding

- What is CORBA and what are its basic facilities and services?
- What is Microsoft's DCOM and what are its basic facilities and services?
- Compare and contrast CORBA with DCOM

5.4 CORBA 3.0 – The Latest CORBA

5.4.1 Evolution of CORBA

CORBA started with specification 1.0 and is at 3.0 at the time of this writing. Initial specifications of CORBA (i.e., CORBA 1.1 and 1.2) did not specify many important services such as security, concurrency, and transaction processing. Many of these limitations were overcome in CORBA 2.0 and

2.2. CORBA 3.0 is built upon CORBA 2.0 and 2.2 and has expanded many existing capabilities plus added new ones. Examples of CORBA 3.0 capabilities are:

- CORBA Messaging
- Real-Time CORBA
- Objects-by-value (OBV)
- CORBA Firewall
- CORBA Component Model
- CORBA Scripting Language
- JAVA-to-IDL/IDL-to-JAVA
- DCE/CORBA Interworking

CORBA 3.0, due to its inclusion/expansion of existing and introduction of new services, is a comprehensive solution for distributed object computing. In fact, CORBA 3.0 has been used as an umbrella term to refer to a suite of specifications which, taken together, add a new dimension of capability and ease-of-use to CORBA.

We will discuss the main features of CORBA 3.0 in the next few sections. Our objective is not to discuss every service that is available under the CORBA 3.0 umbrella, instead we highlight the key developments. It is assumed that the reader is familiar with the general CORBA concepts as explained in the Distributed Object Technologies Chapter in the Middleware Module. If not, please stop and rewind.

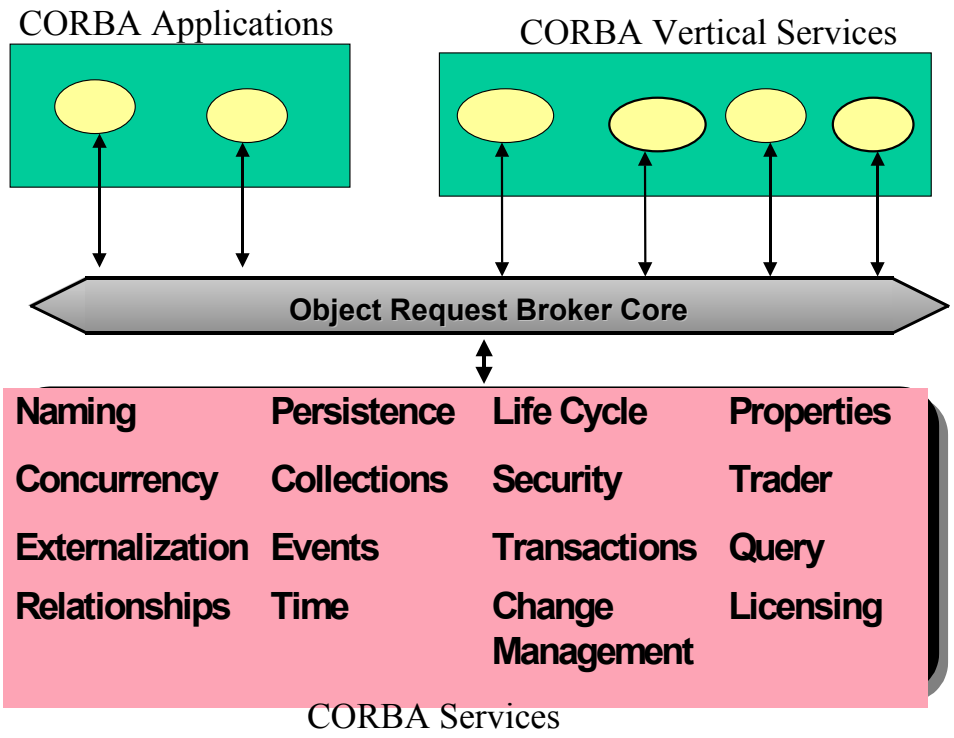


Figure 5-15: CORBA 3.0 Capabilities

5.4.2 Locating CORBA Objects – The IOR (Interoperable Object Reference)

The CORBA object model relies on the notion of locating ("binding") server objects anywhere in the network and then accessing these objects with minimum knowledge of the underlying networks and

operating systems. The Interoperable object reference (IOR) facilitates location and access of server objects and is a cornerstone of the CORBA architecture. The computer-readable IOR is the primary way to reach a server object and invoke it. The IOR is used not only to locate application objects but also to locate CORBA services. For example, you first get an IOR of the Naming Service and then use it. Let us briefly review the key ideas.

First, a terminology note. The server objects, as stated previously, are called "object implementations" or "servants" because they are the implementations of the invoked object (for a C++ or Java object, a servant is an instantiaition of the class).

The following is simplified code to access an object

```
Client issues:  ior1 = Bind ("myobject", host);    /*  The ORB creates an IOR ior1 and returns it to the client */
Client issues: ior1.op1 (p1, p2);   /* the client invokes the operation op1 with parameters p1, p3 */
```

IORs can be converted into strings, sent to other clients through arbitrary means (such as email), converted back to object references by the new clients and used to access the same servant. You can also store the IORs in a standard directory such as CORBA Naming Service or LDAP (Lightweight Directory Access Protocol). IORs can be also used in Trading Services to invoke the most suitable server.

An IOR uniquely identifies a servant and is used by clients to invoke servant operations. An IOR contains information such as name and location of the object implementation, interface type of the object, and unique key (within the scope of a server). Specifically, the IOR contains:

- Protocol and address details such as protocol version and machine and port number (transient objects)
- Object key that contains Object Adapter Name and Object ID (a.k.a Object Name)
- Type name, also called repository ID, is used to retrieve the object from the interface repository.

Once a unique IOR has been created, it can be used at any later time to access the servant. It can also be passed to other clients as parameters or results of operations. The IOR can be converted to/from strings. In CORBA, a common string format is defined for object references. You can:

- use org.omg.CORBA.ORB.object_to_string(in Object obj) to create a IOR;
- use org.omg.CORBA.ORB.string_to_object(in string str) to create an object reference from a IOR

When a client invokes a request via an IOR, the ORB run time is responsible for sending the request to the correct servant. CORBA distinguished between a transient and persistent IOR. A transient IOR continues to work only for as long as the servant is available (i.e., is up and running). A persistent IOR, on the other hand, continues to work even if the server is shut down and restarted, even on a different machine. IORs have profound impact on object scalability and migration (see Henning 1998]).

5.4.3 Object Adapters and POA (Portable Object Adapters)

An object adapter mediates between clients and the object implementations (called servants). As shown in Figure 5-16, resides between the ORB and the servants and handles the incoming client calls by passing them along to the servants. Object adapters provide the following services:

- Generating object references for objects. These references may be persistent or transient. The object types are referred through a lifespan policy.
- Demultiplexing requests to servants. This means finding the right servants and dispatching the requests to the appropriate servants.

- Activation and de-activation of servants. This includes explicit and on-demand activation; and single/ORB-controlled thread model.
- Collaborating with IDL skeletons to invoke servant operations, i.e., marshalling/un-marshalling

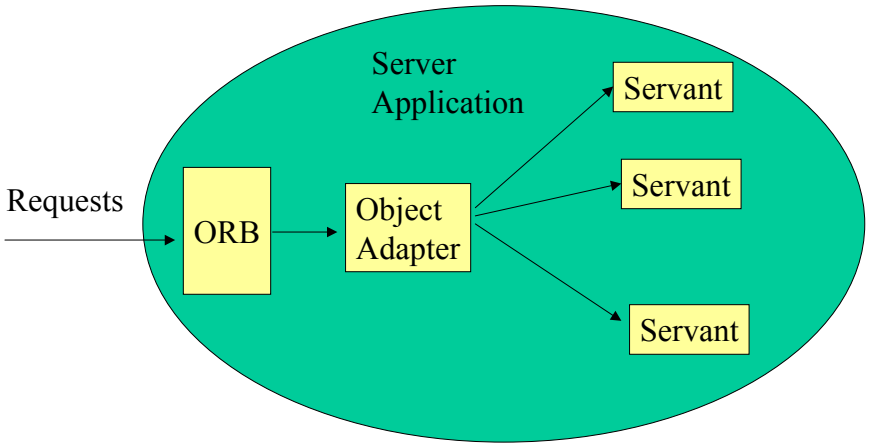


Figure 5-16: Object Adapter Conceptual View

In essence, an object adapter is a manager of the server application that hides all the complexity of the server application. A variety of object adapters can be envisioned to support different types of requests. To avoid proliferation of too many object adapters, OMG specified a Basic Object Adapter (BOA) that can be used for most ORB servants. A BOA is required by CORBA in every ORB so that a servant based on BOA can be used by any ORB. Unfortunately, the BOA was not well specified. Different vendors provided different facilities for implementing servants. For example, some allowed threading while others did not. This led to portability problem where a servant implemented on X platform did not port to Y platform. To overcome this limitation, a POA (Portable Object Adapter) has been specified in later versions of CORBA amd are included in CORBA 3.0. The design goals of POA are:

- Portability, i.e. the servant code should be portable between different ORB vendors.
- Persistent identities, i.e., create an identity that can be used to invoke a persistent object (i.e., the object can be invoked after the session has been over)
- Automation, i.e., transparent object activation and implicit servant activation
- Different resource utilization models, i.e., multiple objects IDs per servant
- Responsible for Objects, i.e., object . ID, state management, code, object status
- Servers can have multiple nested POAs

In addition, a POA is designed to support a wide range of policies such as the following:

- threading: single threaded or ORB controlled
- lifespan: persistent or transient objects
- Object ID uniqueness: one object per servant or multiple objects per servant
- Object assignment: user or POA generated Object ID
- Implicit activation: implicit activation of servant OK or not
- Retention: POA retains active servants in the Active Object Map
- Request processing: requests are serviced by
  - consulting the active object map
  - using a default servant
  - invoking a servant manager

A POA provides functions such as support of an Implementation Repository, facilities for generating and interpreting object references, identification (authentication) of the client (or principal) who submitted the request, activation and deactivation of server objects, and method invocation through the server stubs. POA supports many scheduling policies to support different type of object servers. Examples of the scheduling policies are shared server (i.e., one server handles multiple clients calls), unshared server (i.e., new server is started when a request is made for an object that is not yet active), server per method (i.e, new server is started each time a request is made), and persistent server (servers are activated by means outside BOA).

POA is quite complex and is described very well by [Villonski 1998]. Additional details can be also found in [Siegel 2000]. If you need still more information, see the OMG POA specifications.

Some object adapters are being developed for specialized CORBA situations. For example, an adapter for accessing objects in OODBMs is being considered. In addition, a "streams adapter" has been proposed to OMG to handle distributed multimedia applications over CORBA (see next chapter for additional information).

5.4.4 Objects by Value

In the earlier versions of CORBA, objects are passed by reference only. This implies that the referenced object does not move, any access to the object from a remote application causes network traffic. This can cause performance problems if an object is accessed remotely frequently. The basic idea of object by value is that an object can be, if needed, transferred to a remote site.

CORBA “values” are passed by value to a remote site. There is no correlation between the origin and destination copies, ORB creates a new copy on the receiver. However, the moved object is not registered with the ORB for remote accesses, i.e., it has no identity and can accept only local method invocations. To support objects by value, OMG defined a new *Value* type. State/Behavior passing can be implemented through the streaming interface in java or by means of user defined factories in C++

5.4.5 CORBA Messaging - Asynchronous Messaging and Quality of Service Control

The original CORBA primarily supported a synchronous request/reply model with blocking.. This model, based on the RPC (Remote Procedure Call) model, blocks the clients when they issue a call to the server. The original CORBA also supported Deferred Synchronous (i.e., client issues a call and continues processing; it can later poll or block waiting for a response) for the dynamic invocation interface (DII). However, since DII has been used very rarely, this feature has not been used widely. The original CORBA also supported Oneway (also known as “Fire and Forget”) model. However, this feature has been also used rarely.

Thus CORBA has been criticized as a restrictive specification that forces the users into an RPC paradigm. In particular, lack of asynchronous messaging to support loosely coupled systems has been considered a big hole in CORBA specifications [Vinoski 1998]. In particular, asynchronous messaging supports large scale distributed systems where clients and servers do not stay connected all the time. Asynchronous invocation allows remote requests within an asynchronous., event driven environment in which callbacks are invoked to handle events.

The CORBA Messaging Specification is a comprehensive specification that defines a number of asynchronous and time-independent invocation modes for CORBA, and allows both static and dynamic invocations to use every mode. It also contains numerous Quality of Service policies. Let us review these features briefly.

CORBA Messaging Service preserves the original three request models (synchronous, deferred synchronous, one way) and adds the following asynchronous models:

- Callbacks: The client supplies an additional object reference parameter of a "reply-handler" with each request invocation. When the response arrives, the ORB uses the reference to deliver the response back to the reply-handler. Thus the client can continue processing and ask the server to call back with responses. CallBacks are implemented using ReplyHandler, an object which is invoked when the result of the invocation is available..
- Polling. When the client invokes a request, it is immediately given a valuetype that it can use to poll the server. Based on the result of the poll, the client may decide to block furthher processing or continue. A Poller is a CORBA **value** returned by the call invocation is used to either poll or block until a response is available
- Time Independent Invocation (TII). This allows a store-and-forward model for the situations when the called objects may not be active or disconnnected at the time of call. An agent intercepts the call and keeps the message alive in the meantime. When a Time Independent Invocation is made, the destination object must be a persistent object. Minor changes to GIOP (Generalized Interoperable Protocol) have been made for the message to be delivered to an intermediate agent that can route the calls to destination objects when they become available . .

.

Quality of Service (QoS) is supported through CORBA Messaging policies. Clients can specify the QoS they require for message delivery, message queuing, and message priorities. QoS policies can be specified at the ORB level to affect all requests, at the thread level to affect only those requests made by that thread, and at the target-object reference level to affect only those requests made by the referred-to-object. Reliability aspects of QoS are provided through shared transactions (one transaction per request/reply) and unshared transactions (3 transactions per request/reply; one for client-send, one for deliver-request and receive-reply, and one for client obtain reply)

Minor changes to the OTS (Object Transaction Services) have been made for reliability aspects of QoS Messaging Specification. In the shared transaction model supported by Object Transaction Services (OTS) both the client and the server must be active at the same time. The unshared mode request a minor change to OTS and divide the request/reply process in three transactions allowing the client and the server not to be active at the same time. In the first transaction the request is delivered to the messaging mechanism, the second transaction deliver the request to the target and receive the reply, the final transaction deliver the result to the client. We will discuss OTS in a later chapter.

An Asynchronous Method Invocation (AMI) language mapping is specified to facilitate asynchronous messaging and QoS policies. The changes to the OTS (Object Transaction Services) for QoS and GIOP for time Independent Invocation (TII) are included in the CORBA Messaging Specification.

5.4.6 Minimum, Fault-Tolerant, and Real-Time CORBA – More Quality of service control

Quality of service (QoS) is a major issue at present in distributed systems. As a matter of principle, CORBA traditionally has hidden many low level details from clients such as how objects are located, how network connections are made, etc. . Unfortunately, hiding these details has made it difficult for the CORBA applications to control quality of message delivery. The CORBA Messaging Services, described previously, addresses many of these issues. Additional QoS capabilities are provided through Minimum, Fault-Tolerant, and Real-Time CORBA.

Minimum CORBA. is primarily intended for embedded systems. Embedded systems, once they are finalized and burned into chips for production, are fixed, and their interactions with the outside network are predictable - they have no need for the dynamic aspects of CORBA, such as the Dynamic Invocation Interface or the Interface Repository that supports it. Minimum CORBA.does not include the dynam support features.

Real-time CORBA is an optional extension for all ORBs. It standardizes resource control – threads, protocols, and connections - using priority models to achieve predictable behavior for both hard and statistical real-time environments. In particular, Real-time CORBA supports fixed priority scheduling,

control over ORB resources for end-to-end predictability, and flexible communications. Dynamic scheduling, has been added via a separate RFP.

Fault-tolerance for CORBA has been addressed by an RFP for a standard based on entity redundancy, and fault management control. The entity redundancy consists of: several options such as the following:

- Cold passive replication. Recovery from faults using state information and method invocations/responses recorded in a message log.
- Warm passive replication. Current state of the primary replica is periodically synchronized with the other replicas.
- Active replication. Every replica executes the invoked methods
- Active replication with majority voting. Both invocations and responses are voted.

At the time of this writing, cold passive replication and warm passive replication are commercially available. Other features will become avilable in the future.

5.4.7 Event/Notification Services

CORBA Event Service was introduced for provides an asynchronous publish/subscribe model of event distribution. The basic idea of the event-based publish subscribe model is that a supplier publishes the events that are posted on a channel. The consumers subscribe to the events and respond to the needed events.

The CORBA Event Services support pull (consumer initiated) or push (producer initiated) models. The publish-subscribe model based on push is displayed in Figure 5-17. In this case, the producer pushes the events to the channel from where they are pushed to the consumer.

The event content is packaged into an *Any data type.* The sidebar "OMG IDL CosEventComm Module" shows the IDL of COS (common object services) event model.

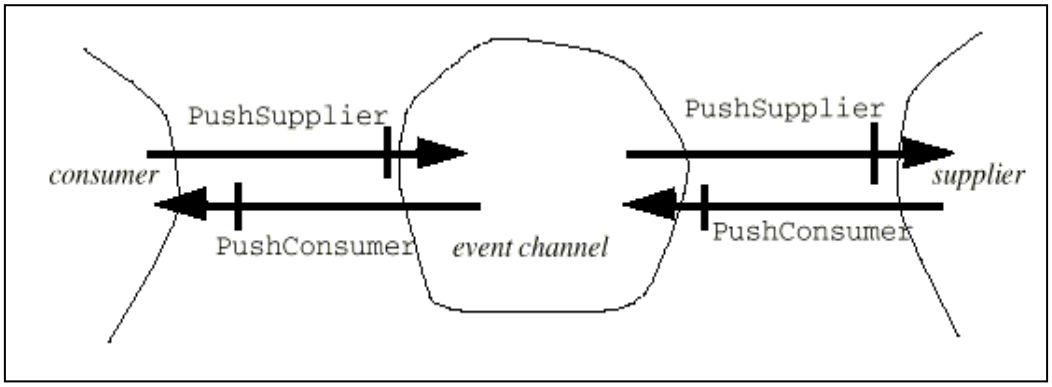


Figure 5-17: An Event-based Publish/Subscribe Model

OMG IDL CosEventComm Module

```
module CosEventComm {
    exception Disconnected{};
```

```
        interface PushConsumer {
                void push (in any data) raises(Disconnected);
                void disconnect_push_consumer();
        };
        interface PushSupplier {
                void disconnect_push_supplier();
        };
        interface PullSupplier {
                any pull () raises(Disconnected);
                any try_pull (out boolean has_event);
                raises(Disconnected);
                void disconnect_pull_supplier();
        };
        interface PullConsumer {
                void disconnect_pull_consumer();
        };
};
```

The CORBA Notification Services address the two major limitations of the Event Service, i.e., no filtering capabilities and no QoS. The Notiifcation Service is interoperable with Event Service and provides:

- Structured events (data structure which most event types can be mapped into). The need to standardize event structure is clear – if there is no standard format, the vendors will creaate their own format for events and thus create an interoperability problem (events from one vendor will not be understood by others. Figure 5-18 shows the format of structured events.
- Extensive optimized event filtering/QoS capabilities that include Event reliability, Connection Reliability, Event Priority, Expiry Time, Earliest Delivery Time, Order Policy, Discard Policy, Maximum Batch Size (only for seq), Pacing Interval (only for sequences). Event filtering is provided for per-channel/per-proxy/per-event QoS
- Sequencing of events so that the events appear in order

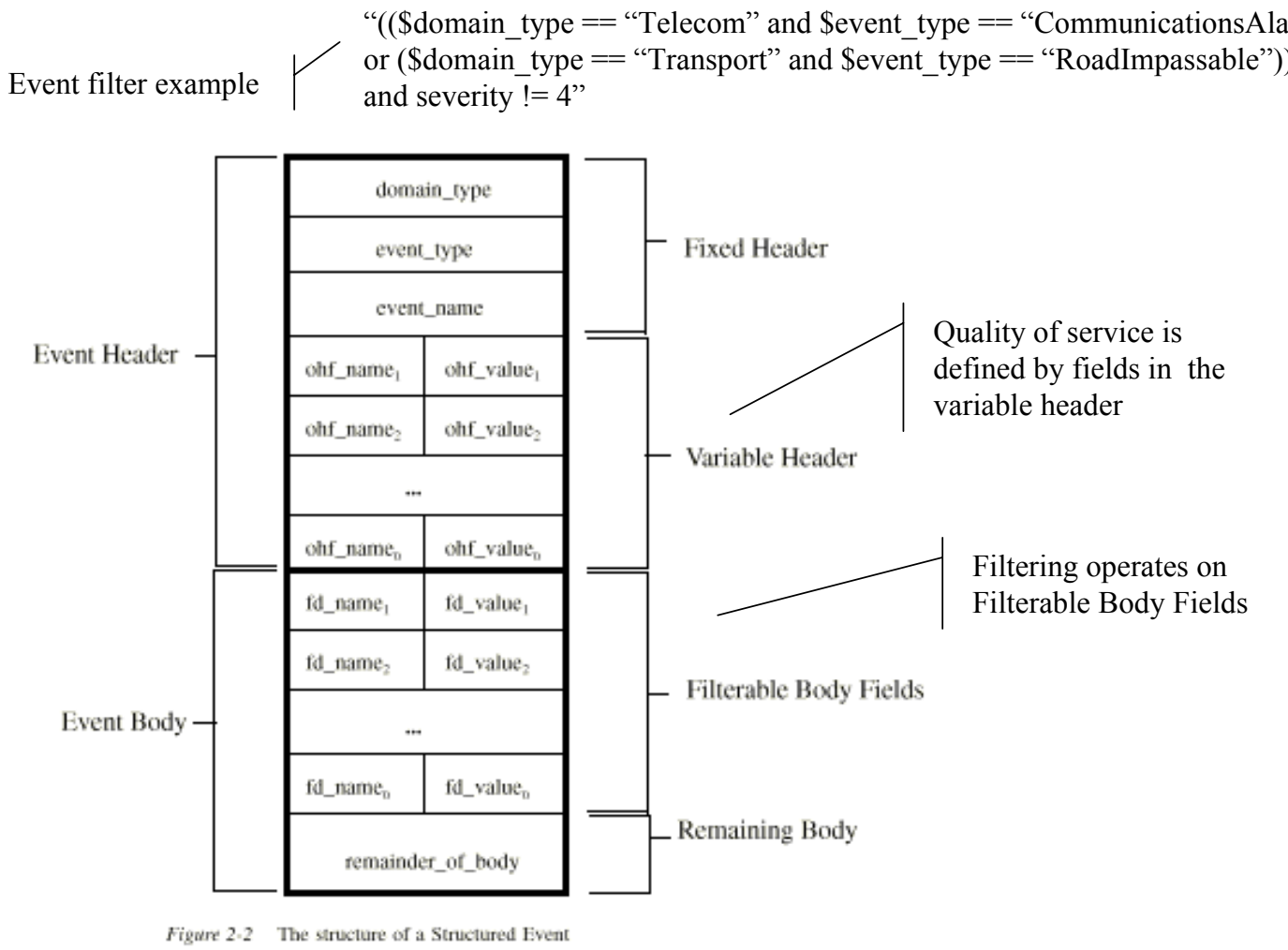


Figure 5-18: Structured Event Format

A few more words about filtering. Notification Service clients have fine-grained control over which events are forwarded by a channel, and which are discarded. This control is supported in the form of Filter objects.. A list of Filter objects can be associated with each Admin and Proxy object within each channel. Each Filter object encapsulates a set of constraints which specify events that should be forwarded. Each constraint consists of a sequence of structures indicating event types, along with a boolean expression over the fields of an event. Event instances which are of one of the types indicated in the sequence of structures, and whose contents satisfy the boolean expression, essentially *match* the constraint

CORBA event-noification is a core technology to support the loose integratiaon needed in enterprise application integration (EAI) . A variety of Event Based Integration Tools are commercialy becoming available. Examples are

- Vitria <www.vitria.com>
- BusinessWare (Connectors, Transporter, Automator, Analyzer)
- Open Horizon <java.sun.com/javareel/isv/OpenHorizon/index.html>
- CrossWorlds <www.crossworlds.com>
- New Era of Networks (NEON) <www.neonsoft.com>
- Active Software <www.activesw.com>

- IBM <www.software.ibm.com>
- MQ Series Workflow, MQ Series Integrator
- TIBCO <www.tibco.com>
- TIB/ActiveEnterprise, TIB/Rendevouz, TIB/ETX, TIB/ObjectBus

5.4.8 Internet integration

The following specifications enhance CORBA integration with the Internet:

5.4.8.1 Firewall Specification.

The CORBA 3 Firewall Specification defines transport-level firewalls, application-level firewalls, and a bi-directional GIOP connection useful for callbacks and event notifications.

Transport-level firewalls work at the TCP level. By defining well-known ports 683 for IIOP and 684 for IIOP over SSL, the specification allows administrators to configure firewalls to cope with CORBA traffic over the IIOP protocol. .

CORBA, as stated previously, CORBA supports call back where the client-side module instantiates an object that is called back in a reverse-direction invocation. Because standard CORBA connections carry invocations only one way, a callback typically requires the establishing of a second TCP connection for this traffic heading in the other direction, which is not supported by most firewalls. Under the new specification, an IIOP connection is allowed to carry invocations in the reverse direction under certain restrictive conditions that do no't compromise the security at either end of the connection.

5.4.8.2 Interoperable Name Service.

The CORBA object reference is a cornerstone of the CORBA architecture. In CORBA, there was no way to reach a remote instance unless you could get access to its object reference. The easiest way to get a reference to an object is through the Naming Service. But what if you did not have a reference for the Name Service (NS) itself (NS also needs an object refernce)?

The Interoperable Name Service defines one URL-format object reference, iioploc, that can be typed into a program to reach defined services at a remote location, including the Naming Service. A second URL format, iiopname, actually invokes the remote Naming Service using the name that the user appends to the URL, and retrieves the IOR of the named object.

For example, an iioploc identifier iioploc://www.comany.com/NameService would resolve to the CORBA Naming Service running on the machine whose IP address corresponded to the domain name www.company.com, assuming that a Name Server was running at this company.

5.4.9 Inter-ORB Architecture (GIOP and IIOP)

As specified previously, initial specifications of CORBA (i.e., CORBA 1.1 and 1.2) did not specify many implementation details such as security. This has led to CORBA implementations by vendors that do not interoperate with each other. In particular, ORBs based on CORBA 1.1 and 1.2 from different vendors interoperate with each other only through joint agreements and partnerships between vendors. CORBA 2.0 addressed this problem (CORBA 3.0 includes this feature) by defining inter-ORB protocols (IOPs) for interoperability of Object Request Brokers. For existing ORBs that do not interoperate, OMG is specifying an API for adding bridges between ORBs.

The IOP specifications are significant feature of CORBA because they allow ORBs, potentially from different suppliers, to interoperate with each other. These specifications do not impact application developers (application programmers will typically not write code at this level), but they do play an important role in how an application developed on one ORB can interoperate with an application developed on another ORB. Figure 5-19 shows the principal IOPs and depicts how they interrelate with each other.

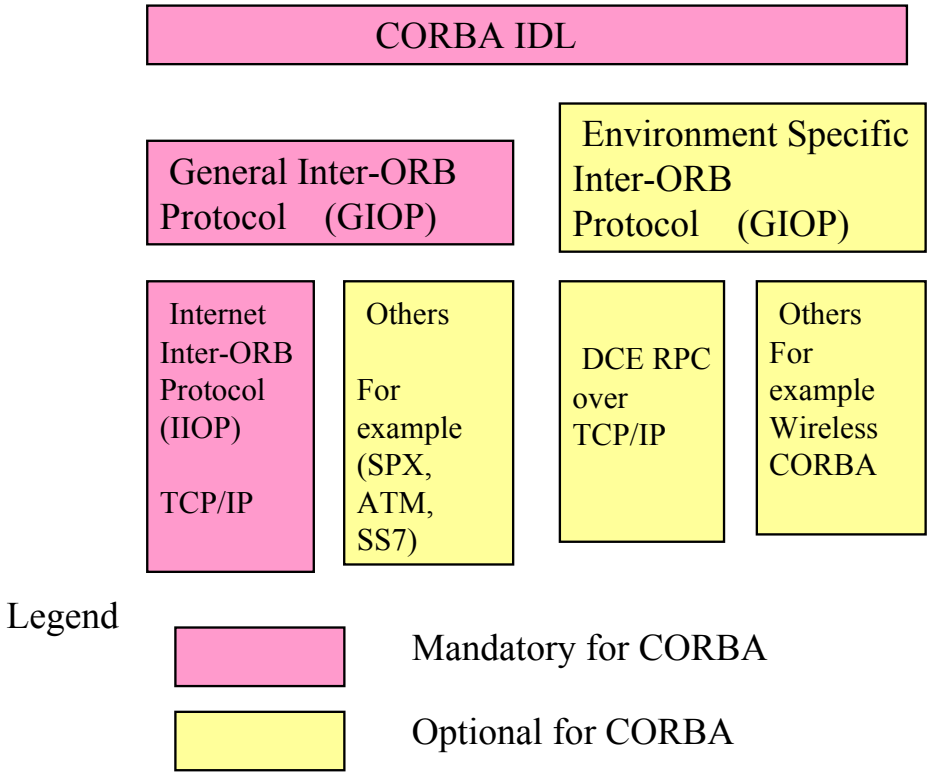


Figure 5-19: CORBA Inter-ORB Architecture

General Inter-ORB Protocol (GIOP). This IOP specifies a set of message formats and common data representations for interactions between ORBs. The GIOP is especially designed for ORB-to-ORB communications. It is intended to operate over any connection-oriented transport protocol. The Common Data Representation (CDR) is used to map OMG IDL data types (pointers and linked lists) into a "flattened" network message that can be transported over the network. GIOP also specifies a format for interoperable object references so that a given object can be accessed from different ORBs.

Internet Inter-ORB Protocol (IIOP). This IOP specifies how GIOP messages are exchanged over a TCP/IP network. IIOP allows a lightweight implementation of CORBA so that CORBA can operate directly on top of TCP/IP and not on top of DCE. IIOP is a required feature of CORBA 3.0. In other words, an ORB must support IIOP to be CORBA 3.0 compliant. GIOP specifications on top of IPX/SPX, ATM networks, and telecom SS7 networks are under different stages of development.

Environment Specific Inter-ORB Protocols (ESIOPs). These IOPs are an alternative to GIOP and are specified for specific environments. CORBA 3.0 has specified DCE as the first of many optional ESIOPs. DCE/ESIOP does not require DCE IDL (OMG IDL does the job). The DCE/ESIOP includes many features that are important for mission critical applications. Examples of these features are the DCE security, cell and global directories, authenticated RPCs, and distributed time services (all these features are part of OSF DCE). However, this makes CORBA 3.0 applications heavy weight ("CORBA Heavy"). Another ESIOP has been defined for Wireless CORBA. In the future, ESIOPs may be specified for other environments.

Bridges Between ORBs (Half and Full Bridges). CORBA provides facilities for developing generic ORB-to-ORB bridges. These bridges come in two flavors: half bridges and full bridges. Figure 5-20 shows these bridges.

- ORB Half-Bridges. An ORB half-bridge relies on a common ORB (the "backbone" ORB) to interconnect different ORBs. For example, Figure 7.13a shows how an IIOP backbone can be used to interconnect different proprietary ORBs. The key point is that your ORB needs to communicate with an IIOP through a half-bridge that translates your ORB to IIOP. After this, the IIOP can be used as a "Global ORB" bus. Keep in mind that IIOP runs on top of the Internet. The ORB half-bridges are similar to the gateways used in networks that convert different network protocols (e.g., SNA, SPX/IPX, OSI) to a backbone network protocol (e.g., TCP/IP). Half-bridges allow a federation of different ORBs around an IIOP backbone.
- ORB Full-Bridges. The full-bridges directly convert one ORB to another without requiring a common backbone ORB. The CORBA 2.0 Dynamic Skeleton Interface (DSI) is used to receive outgoing messages and the Dynamic Invocation Interface (DII) is used to receive inputs and invoke destination objects (see Figure 7.13b). Thus an ORB from one vendor can communicate with an ORB from another vendor directly. This approach is used in some bridges such as CORBA/OLE bridges [Orfali 1996].

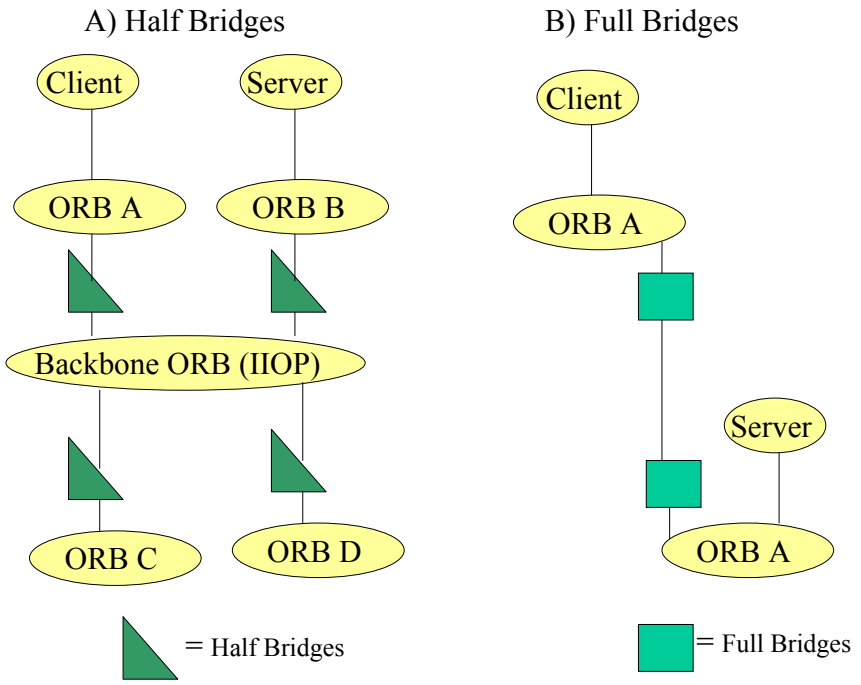


Figure 5-20: ORB-to-ORB Bridges

5.4.10 CORBAComponents Package

CORBAcomponents represents an attempt with potential benefits for programmers, users, and consumers of component software. The three major parts of CORBAcomponents are:

- A container environment that packages transactionality, security, and persistence, and provides interface and event resolution;
- Integration with Enterprise JavaBeans; and
- A software distribution format that enables a CORBAcomponent software marketplace.

The CORBA Component architecture consists of several interlocking conceptual pieces that enable a complete distributed enterprise server computing architecture. These include an Abstract Component Model, a Packaging and Deployment Model, a Container Model, a mapping to EJB and an Integration Model for Persistence and Transactions. The OMA framework for supporting the definition, code generation, packaging, assembly, and deployment of these CORBA components is collectively called the ***CORBA Component Model (CCM).*** It represents a major extension and addition to the OMA and CORBA.

Although a great deal of press has been devoted to CCM, it has not materialized in strong industrial products. The future of CCM is dubious at this time due to the popularity of other component models such as Suns's J2EE and Microsoft's .Net. Due to this reason, we will not spend a great deal of time on CCM. The interested reader should consult the OMG web site (www.omg.org) for additional details.

5.4.11 CORBA Scripting

Scripting for CORBA Components and Objects provides a uniform scripting environment for CORBA applications. No distinctions are made between CORBA objects/components and "local" objects/components in CORBA scripting. The CORBA Scripting is aimed at the creation of assembly of components, the management of servers, servants and objects, testing. The scripting languages can be Interpreted or semi/compiled. The specification contains 4 scripting-language mappings: JavaScript, CorbaScript, Pyton, and Tcl. The scripting specification is Dynamic Invocation/Skeleton based

Scripting languages, such as CorbaScript, are especially designed for testing CORBA applications. CorbaScript, in particular, is a general-purpose, object-oriented scripting language that allows any user to quickly develop test scripts by interactively accessing objects available on the CORBA bus

5.4.12 CORBA 3.0: Conclusions

After ten years of cooperative work by OMG members, the base CORBA infrastructure is complete and in constant use at thousands of sites. The extensions bundled under the banner of CORBA 3 bring ease-of-use and precise control to established architecture. These additions will ensure that CORBA continues to play an ever-increasing role in the computing world of the future.

- Powerful distributed object environment
- Reasonably complete and vendor/language independent
- Sophisticated solutions e.g., components, Trader Service, UML, MOF, ...
- Still several open issues

However, CORBA 3.0 is not complete and will continue to grow. Specifically:

- Limited number of services are available, more services need to be defined
- Some new services need completeness. For example, POA can be further improved because it lacks sophisticated threading policies. Also, CORBA Objects cannot be moved/copied in object by value.
- Interoperability needs additional work. Some interoperability issues have been solved, other are open, e.g., permanent object server interoperability and implementation repository interoperability

5.4.13 Other OMG Activities

The OMG is constantly evolving and improving. Examples are:

- The Open Group Interoperability Assurance Program. This includes ORB implementation of CORBA features, test suite for application portability, and network computer interoperability.
- Analysis and Design Platform Task Force. This Task Force is developing several facilities such as Stream-based Model Interchange, Semantics for UML, Corporate Warehouse Model (CWM) specifications, and Software Process Engineering. This process is working through its RFP/RFI cycles.
- Benchmarks, Performance and Realtime . A Platform Special Interest Group (PSIG) of OMG is considering specifcations for Benchmarks, , High Performance CORBA, and Realtime CORBA..
- Business Objects. Business objects (BOs) are objects that represent entities and processes that occur in real-world business domains, such as customers, orders, accounts, etc.They are distinct from technology objects that represent artifacts of software technology, such as name contexts, transactional resources, etc. A business domain object model consists of the specification of a set of business object types and of the relationships among them.

In addition, numerous Domain Task Forces (DTFs) are developing CORBA services for vertical markets. Examples of vertical domains are:

- The Telecommunication Task Force, that is concentrating on the Event Services, asynchronous interaction model, publish/subscribe model of event distribution, and basic QoS and filtering
- Financial Domain Task Force that is concentrating on party management and , general ledger
- CORBAmed Domain Task Force (DTF) is working on clinical observations, Healthcare resource , Healthcare data interpretation, Clinical Image Access Server, Medical Transcript Management,
- Utility Domain Task Force that is working on Utility Management System data Access Facility
- Electronic Commerce DTF exists to define and promote the specification of OMG distributed object technologies for the development and use of Electronic Commerce and Electronic Market systems. It is concentrating on electronic payment and negotiaition facility for trading

5.5 CORBA Operability (Performance, Scalbility, Fault Tolerance) and Summary

5.5.1 CORBA Reality Check – Operability (Performance, Scalbility, Fault Tolerance) Issues

You may be overwhelmed by the CORBA wide range of CORBA services and facilities – not very many applications need all these services. In fact, only needed services can be included at system configuration time. These object services provide a powerful set of capabilities for creating customized middleware. For example, a CORBA"heavy" middleware can be built for transaction management with high security and messaging requirements by including the CORBA Transaction, Security and Messaging Services. On the other hand, a CORBA "light" middleware can be created by ignoring many object services. Cost is also a consideration in using these services. Most CORBA vendors at present charge separate license fee for each service. Thus, you have to evaluate the cost implications before buying a CORBA service. In many real life situations, cost helps to clarify thinking – this may be one of those cases.

An important issue to keep in mind about CORBA is that it is a specification, not an implementation. Thus the vendor implementations of CORBA must be analyzed carefully for performance, scalability, fault tolerance, portability, and interoperability. :

Performance and scalability analysis. Approaches to performance and scalability of distributed systems vary widely (see, for example [El-Rewini 1997]}. For performance, response time is the most common metric. However, throughput (transactions per second), memory utilization, CPU utilization, and I/Os issued can also be used for performance metrics: For scalability, we need to examine how

easy it is to "expand" the system capabilities as the demands on the system increase. General scalability parameters for distributed systems may include no of clients handled per server, no of servers allowed (supported), no of active sessions (number of threads), "concentrators" (replicated identical servers) supported, total no of users (directory entries), database size, number of files, and other application specific measures.

A set of experiments, such as the ones suggested in Table 5-3, are needed to help in the analysis/evaluation of competing ORB products. These experiments, based on our own experience at Telcordia Technologies, include:

- Core Services Experiments: compare ORBs based on Bind and "null" RPC times for single-threaded, multiple-threaded client and server
- Assessment of scalability and connection management
- Nameserver Performance Experiments: Create a naming tree containing 10,000 object bindings and measure times to create contexts and bindings. Test nameserver crash recovery.
- Performance of a CORBA Server Demultiplexing varying the number of services instance in the server process space. The ORB call dispatcher was a bottleneck in the early off the shelf implementations.
- Data Marshalling/Unmarshalling Experiments. Passing different type of data structures in CORBA incurs different overhead due to marshalling/unmarshalling. Experiments used Netscape Navigator and Microsoft Internet Explorer over IIOP as well as Orbix proprietary protocol.
- Object Transaction Experiments

Operational analysis (fault tolerance, portability, interoperability). Operational and management support include fault tolerance, portability, interoperability, manageability (i..e., load balancing, monitoring capabilities, start-up/shutdown), installation and maintenance (e.g., ease of installation and support), and cost/technical support are pivotal to the successful deployment of CORBA-based applications. Table 5-4 suggests a set of operational support and management experiments .

Our main findings after running through a series of experiments listed in these two tables are:

- Performance and scalability features of different ORBs vary significantly. Thus It is important to bring in the vendor products and test theme thoroughly. The vendors appear to improve their products in response to the problems, especially in the areas of high visibility.
- The vendors appear to improve their products in response to the problems
- Portability of code between two ORBs even when both conform to the same CORBA specification is not trivial
- CORBA does perform adequately andoes scale well under several conditions.
- The developments in minimal CORBA and realtime Corba are further expected to improve the situation.
- It is important to investigate all services (event, trader, transaction) to get better insights
- Application performance depends on type of applications
- General design guidelines should be developed by the application designers

Table 5-3: Suggested Performance and Scalability Tests of ORB Services

CORBA Services	Performance Experiments	Scalability Experiments
ORB basic service	a) Binding time b) "Null" object invocation (Ping) c) Data type passing ("Timer application")	a) increase no of calls (same client, same server) b) increase no of clients c) increase no of object servers (same machine) d) increase amount of data passed
Thread Services	a) Time for thread pooling b) Time for Thread per Session	a) Scaling of thread pools b) scaling of threads per session

Life Cycle service	Time to create an object , memory limitations	Creation of objects (how many)
Name Services	a) Time to access naming service (benefit in doing handle caches) b) Time to use name contexts and context factories	a) number of entries in directory, b) handling large number of entries ("cells", or "domains"),
Security services	Time to authorize/authenticate	No. of users
Transaction Services	Time for successful and unsuccessful transactions	No of concurrent transactions(no. of locks held)
Persistence service	Time for database access	Database sizes
Trader service	Time to find the "best" server	Do traders scale for large number of servers
Event services	Time for notification	No of events

Table 5-4: Operational and Management Experiments

Operational and Support Issues	Basic Issues	Additional Issues
Fault Tolerance	*System behavior and performance under different types of failures.* *Daemon failure* Server fault. Host failure	Measure time of recovery (i.e., does it recover at all, does any thread block), exceptions (does the client detect any problem), and performance degradation (i.e., how much does the performance degrade during the transitory time, how many transaction experience large delays) for various failure scenarios.
Interoperabi-lity	Clients from vendor1 ORB access servers from vendor 2 ORB Clients in one language access servers in another	Interoperability among complex applications Fault tolerance among different ORBs (i.e., can an application across ORBs be fault tolerant?)
Portability	IDL portability Client portability Server portability	Guidelines for developing portable CORBA applications
Installation and maintenance	Ease of installation and support	Administration for large scale systems
Cost	Basic licensing fees for development and run time	Special licensing fees
Technical Support	Availability of documentation and technical staff	On-site help in special problem solving

5.5.2 CORBA Summary

Object-oriented technologies and techniques have natural applications in distributed systems. Entities in distributed systems can be viewed as objects exchanging messages. OMG was formed to create a suite of standard languages, interfaces and protocols for interoperability of applications in heterogeneous distributed environments. CORBA is an OMG specification for invoking objects in a distributed environment.

Initial specifications of CORBA did not specify many details. However, a wide range of capabilities have been added under the CORBA 3.0 umbrella. In addition, several Domain Task Forces and Platform Special Interest Groups (PSIGs) have been busily working on different aspects of CORBA. Examples of key CORBA developments in CORBA Services, distribution protocols, specialized models, vertical domain facilities, support for analysis and design, and basic object computing model are summarized in the sidebar "Key CORBA Developments".

Many vendors are announcing CORBA compliant software. Examples of a few products are:

- Inprise's Visibroker
- Iona's Orbix
- BEA's M3
- IBM's Object Broker
- IBM (SOMobjects) - free for OS/2, AIX, Windows
- PrismTech (OpenBase)
- ParcPlace (Distributed Smalltalk)
- TIBCO (ObjectBus) -
- ObjectSpace (Voyager)
- Objective Interface Systems (ORBexpress) also marketed by ObjecTime
- Bionic Buffalo
- I-Kinetics
- RogueWave (Nouveau)
- Java RMI over CORBA IIOP

A wide range of free and prototype CORBA implementations are also available. For a list of such products, consult the Washington University at St. Louis web site (www.wustl.edu/~schmidt/corba-products.html).

For a wide range of sources, including test tools, are available at http://www.vex.net/~ben/corba/

Additional details about CORBA are listed in the sidebar "Key Sources of Information for CORBA". The OMG home page (http://www.omg.org) is an excellent source of the recent activities in CORBA.

Although we have concentrated on the object request brokers so far, the concept of brokers is general and is currently being exploited in the message broker architectures (see the sidebar "Message Broker: Another Kind of Broker").

Key CORBA Developments

- **CORBA Services**
  - Naming - directory service: (svc name) → (svc object reference)
  - Trading - service discovery: (svc attributes) → (svc name)
  - Event/notifcation - Producers notify consumers using events
  - Transactions - distributed (2PC), flat transactional objects
  - Security - different levels (IIOP over SSL, additionally used)
  - Messaging - Support asynchronous processing for loosely coupled
- **Distribution protocol**
  - GIOP (Generalized Interorb Protocol) and its mappings (IIOP)
  - Mappings for SS7, ATM
- **Specialized Models**
  - Real-time, Fault-tolerant, Minimum CORBA
- **Vertical Domain facilities (e.g., Telecom)**
  - CORBA to TMN Interworking
  - Wireless CORBA
- **Support for Analysis and Design**
  - UML(universal Modeling Language)
- **Basic Object-Oriented Computing Model**
  - CORBA Components, ISO/OMG IDL and language mappings

Message Broker: Another Kind of Broker

A broker mediates between clients and servers (i.e., instead of a client directly connecting to a server, it first connects to a broker that in turn finds a suitable server). The concept of a broker is independent of the implementation of the broker. For example, the best known implementation of the broker architecture is the object request broker (ORB) i as presented in OMG CORBA specification. In CORBA, the ORB mediates the interactions between remote objects. Another type of broker, called a message broker, is being presented as a viable implementation of the broker architecture.

A message broker is not restricted to objects. Instead, it delivers messages between disparate applications, including legacy applications. The underlying technologies used by the message broker may consist of RPCs or MOMs, although MOM does appear to fit this model quite well. The basic idea of a message broker is that it can provide brokerage services asynchronously and support a "publish/subscribe" model. The message broker can also be rule-based, i.e., you specify the rules to be used by the middleware to perform certain actions.. Message brokers are at the core of “Enterprise Application Integration” platforms that are commercially available from many vendors such as Active Software, Vitria, and IBM. EAI platforms, based on message brokers, are expected to be a $11 Billion market by the early 2000s. We will discuss EAI platforms in a later chapter.

The Gartner Group proposed and advocated message brokers as key to the future success of distributed computing. The Gartner Group predictions in the mid 1990s that message brokers will be as widespread as database gateways and data warehouses have been correct. See, for example, Bort, J., "Can Message Brokers Deliver?", Applications Software Magazine, June 1996, pp.70-76).

5.6 A Detailed CORBA Example

5.6.1 Overview

Let us go through an example of developing a simple inventory system by using CORBA. The inventory system consists of a relational table that contains product information (e.g., product ID, product name, price, and quantity on-hand). This table is managed by a "product object" that responds to requests from clients to add, view, update, and delete a product. For example, a client invokes a product view operation by passing a product ID. The product object receives the request and invokes a method that reads the product information and sends it back to the client. Table 7.2 shows the object model for the customer object. In the sections that follow, we will use this model as a starting point for defining the interface, building a server and building clients in CORBA.

This example is intended to give an overview of the process needed for CORBA application development. Additional details can be found in books and araticles such as [Otte 1996, Minton 1996, Orfali 1996, Mowbray 1995].

Table 5-5: The Product Object

Object	Operations	Inputs	Outputs
Product	1. Add a product _____	Product information _____	Status _____

	___ 2. View a product information _____ ___ 3. Update a product information _____ ___ 4. Delete a product	___ Product ID of the products _____ ___ Product ID, new information _____ ___ Product ID	Product information (a structure) _____ Status and indication that a change was made _____ ___Status

Figure 7.19 shows the activities involved in developing OO applications in CORBA (this figure is repeated from an earlier discussion):

- Create CORBA definitions by using OMG IDL
- Build the server
- Build the client

5.6.2 Create CORBA Definitions

The following CORBA definitions are created as the first step in CORBA application development:

- Define the interface
- Define the implementation
- Define the method map

As stated previously, the interface of an object is used to declare the behavior of an object. For example, the following statements specify the interface for the product object in CORBA IDL. This interface only supports two operations (we have simplified it somewhat for illustrative purposes):

```
module PRODUCT_PACKAGE {
interface product_interf /* interface name is product */
insert_product  (/*Operation is insert_product */
in char product_obj; /* input is product object */
out integer status )  /*  output parameter */
view_product ( /*Operation is view_product */
in char product_id;  /* input pparameter is product-id */
out object product_obj; /* output: product object */
out integer status )  /*output parameter 2 */
}
```

The module statement is used to name a group of interfaces that relate to each other and can be used to represent a package (i.e., a group of objects). The module name becomes part of the name that the client and server use to reference an interface. The interface statement shows what operations can be performed on the customer object (we have only shown two operations, others can be filled in by the reader). Each interface statement defines an interface and contains the descriptions of the operations (operation signatures). The complete name of an operation includes the module and the interface name. For example, "PRODUCT_PACKAGE::product_interf::insert_product" is the fully qualified name of the operation that creates a new product.

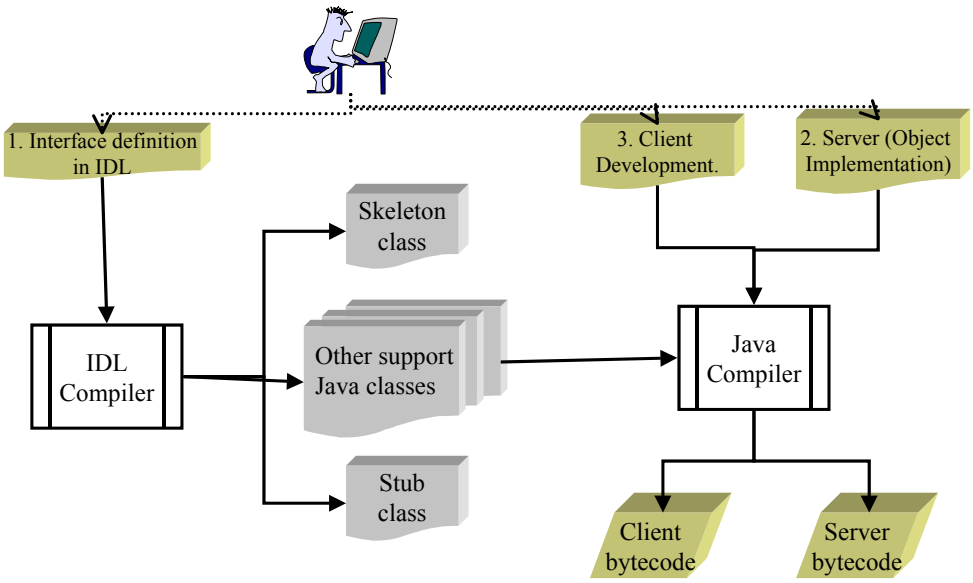


Figure 5-21: CORBA Application Development

The IDL file is compiled to create the client stub and the server skeleton. The client stub and server skeleton are the code templates for building CORBA client and server programs (see Figure 5-21). The client stub is used only to build client programs that use static binding. The server skeleton is always used as the framework for building the server application, regardless of the invocation type used by the client.

Operations on objects are impl.lemented by executable code called methods. For example, the product object must contain the code ("add-product-method") that will be executed to actually create a new product when an operation "add-product" is invoked by a client. The collection of methods that accomplishes the set of operations required for an object is called an implementation. Some CORBA implementations offer a mapping language for describing implementations called the Implementation Mapping Language (IML). The implementation descriptions can be stored in an Implementation Repository. which can be displayed by CORBA compliant commands or the Repository Manager. The following statements show the implementation of product methods (once again, we have simplified this somewhat):

```
implementation productImpl
(
activation_type (program);
implementation _identifier ("676873.0c.03.00.00.00.00");
add_product_method ()
implements (PRODUCT_PACKAGE::product_interf::add_product);
invoke_builtin ("add_p_function)
;
);
```

The implementation statement is used to specify the name of the implementation. The first few, in our case the first two, statements in an implementation specification are used at start-up time. The activation_type() statement specifies the manner in which the implementation is started once it is selected. For example, (program) indicates that a program will be started; other options indicate dynamic load and script executions. The implementation _identifier() clause is used to assign a unique identifier to an implementation. The unique identifier is automatically generated in response to the CORBA command "orbgen'. For each method in the implementation, you define the method name, the

operation it supports, and how to run it. In our example, we have defined only one method (add_product_method) that implements the add_product operation defined in the IDL and invokes a built-in module. Method specifications allow invocation options for dynamic and script executions.

In addition to IML, you may need to define method maps. These maps are used by the ORB to locate the best server method for the object operation requested when there are more than one active implementations of an interface. For example, when a client invokes the "add_product" operation, then the ORB looks at the method maps to determine the most appropriate method to invoke. The method map can be stored in the Interface repository along with the interface definitions. You can use CORBA commands to generate a default method map from the interface definitions.

5.6.3 Build the Server

Building of program servers requires the following steps:

- Generate the server skeleton
- Develop server initialization code
- Develop code for each method
- Compile, link and test the server components
- Register the server

The first step is to specify the IDL statements and then compile these statements to generate a server skeleton. This compilation also uses the information contained in the IML files. The server skeleton contains method templates that show entry points for all of the implementation methods, the server dispatcher code that makes the implementations, the methods known to the ORB, and a registration routine is also generated as part of the server code. The skeleton code simplifies the task of building a server.

Each server initialization needs code to register the implementation (e.g., make itself available for use), activate the server's implementation, enter a main loop to receive requests, and exit after unregistering and releasing resources. In addition, the server needs routines for creating objects and managing references to these objects. The server skeleton is used to develop this code. The initialization code uses CORBA run-time routines (these routines are identified as CORBA_ or ORB_). An example of the server C-type pseudo code for the product example is listed below (we have simplified this code by ignoring the error checking and by using a generic "parameter" for the CORBA provided functions):

```
#include <stdio.h>
#include <orb.h>
#include product.h   /* the IDL generated header */
main  ()
{
printf ("product server starting \n");

/* Register the implementation */
status = RegisterImpls (parameters);


/* Create object and its reference */
status = CreateObjRefs(parameters);

/*Make server ready */
CORBA_BOA_impl_is_ready (parameters); /* indicate server ready */
```

```
/* main loop to listen to  client messages */
ORB_BOA_main_loop (parameters); /* enter the main loop */

/* exit code */
CORBA_BOA_dispose(parameters); /* Frees object references */
ORB_BOA_imp_unregister (parameters); /* Unregister */

/* Methods code  templates */
void insert_product (product_obj, status);
/* .... insert code for add_product method ...*/
void view_product ( product-id, product_info, status);
/* .... insert code for view_product method ...*/
}
```

The major activity in building a CORBA server is to write the code for the methods that execute the operations. For each method template, you must develop the code for the methods. Methods can be implemented as executable code, calls to legacy applications, or scripts to integrate command line interfaces with existing applications. Methods can be written to invoke OLE (Object Linking and Embedding), DDE (Dynamic Data Exchange), and SQL database accesses. For example, the following pseudo code to access the product relational table can be added to the add_product and view_product methods:

```
void add_product  (product_obj, status);
/* code for converting product_obj object into
SQL table attributes such as pid, pname, price, on_hand */
exec sql;
insert pid, pname, price, on_hand into product_table;
end sql;
/* .. include other code */

void view_product (product-id, product_obj, status);
exec sql;
select pid, pname, price, on_hand from product_table where pid=:product_id;
end sql;
/* code for storing pid, pname, price, on_hand into an object */
}
```

The final step in building a server is to compile and link the methods, server initialization code, the generated server dispatcher and the generated registration routine. The server code can tested and debugged by using CORBA tracing facilities.

5.6.4 Build Client (Static Invocation)

After a server has been built and registered, clients can be built to invoke the servers. As stated previously, CORBA clients can use static invocations (i.e., clients know at compile time the objects and the operations on these objects) or dynamic invocation (i.e., the clients determine at run time the objects and the operations on these objects). We will focus on static invocation in this section and review dynamic invocation in the next section. The steps involved in building a static invocation CORBA client are:

- Generate client stub
- Define the context object
- Build the client code
- Compile and link the client

The client stub can be generated from IDL, IML and MML source files or from the interface repository. The generated stub consists of a header file that contains definitions, and the C language stub routines.

A context object shows a set of properties providing information about the client, the environment, or characteristics of the request. The context object is used by ORB during method resolution to identify user preferences for server selection. Basically, it provides a means of maintaining information between requests for conversational applications. The context information is difficult to pass as parameters in a distributed application. The IDL is used to specify whether the ORB should also retrieve information regarding the request from the context object (the "context" clause in IDL). If no context object is specified, the ORB uses the default context object definition. Context objects can be specified at user level (e.g., user preferences), group level (e.g., data restricted to a group of users), or system level (e.g., display types for an application).

The client code includes header file generated by IDL, "local" client code (e.g., communicate with the user), invoke object operations defined in IDL, and handle errors/exceptions.

To invoke the object operations, a client needs to first get an object reference (object references can be stored by the server at start-up in an external file or Registery), and then invoke a method on the object. The following client pseudo code illustrates the key points: of a client that invokes the add_product and view_product methods:

```
 #include <stdio.h>
 #include product.h   /. the IDL generated header ./
/* define variables, etc. */
 main ()
{
  /* code to obtain object reference. This code depends on where the server stored the reference. If object reference is in
a file, then use fget, for example, to read the object reference */

   product_interf  *pptr;   /*  *pptr is the object reference  */

/* Now invoke the add_product and view_product methods */

/* put information in product_obj */
pptr->add_product  (product_obj, status); /* invoke the object method */
printf ("product added");
product_id = "1111";
pptr->view_product ( product-id, product_obj, status); /* invoke the object method */
printf ("product information", product_info);
/* Other client code, e.g., free resources, error processing, etc. */
```

After coding the client, it is compiled, linked and debugged. by using the CORBA environment compilers and tracing facilities.

5.6.5 Building a Client (Dynamic Invocation)

CORBA's Dynamic Invocation APIs allow a client program to build and invoke requests on objects at run time. These APIs provide maximum flexibility by allowing new objects to be added at run time. The client specifies, at run time, the object to be invoked, the method to be performed, and the set of parameters through a call or a sequence of calls. The client code typically obtains this information from the Interface Repository. To invoke a dynamic method on an object, the client must perform the following steps:

- Obtain the method description from the Interface Repository

- Create the argument list
- Create the request
- Invoke the request

CORBA specifies about ten API calls for locating and obtaining objects from the Repository. An example of such an API call is lookup_name(). A describe call is issued, after an object is located, to obtain its full IDL definition. To create an argument list, CORBA specifies a NameValue list as a self--defining data structure for passing parameters. The list is created by using the create_list operation. After this, the request is created using the CORBA create_request call. Eventually, the client can invoke the request by using either an invoke call (send the request and obtain the results, i.e., a synchronous call), or a send call (an asynchronous call). The following pseudo code shows a sample dynamic invocation:

```
/* Create method description */
lookup_name()
describe ()
/* Create argument list */
create_list ()
add_arg(),,, add_arg(),,,, add_arg()
/* create the request */
create_request(Object Reference, Methods, Argument List)
/* Invoke the remote method synchronously - as an RPC */
invoke()
/* Now process the results */
Distributed Objects (CORBA and OLE/ActiveX)
7
```

5.7 Summary

Object-orientation (OO) has a great deal of promise in reducing the complexity of C/S applications. The primary standard for distributed OO systems is OMG's CORBA. CORBA is a very powerful and important specification for distributed object oriented applications. Although other technologies such as SOAP are getting more attention at present, development in CORBA continues.

Key Sources of Information for CORBA

- J. Seigal, “CORBA 3.0: Fundamentals and Programming”, second edition, Wiley, 2000
- D.Schmidt, *Overview of CORBA*: http://siesta.cs.wustl.edu/~schmidt/corba-overview.html
- R.Orfali, D. Harkey, *Client/Server Programming with Java and CORBA*, Wiley 1997
- T. Mowbray, W. Ruh, *Inside CORBA*, Addison-Weslet 1997
- T. Mowbray, R. Zahavi, *The Essential CORBA*, Wiley 1995
- T. Mowbray, R. Malveau, *CORBA Design Patterns*, Wiley 1997
- J. Farley, *Java Distributed Computing*, O’Reilly 1997
- The OMG website: http://www.omg.org
- “Corba Connections”, Comm. of ACM, Special Issue, October 1998
- http://www.vex.net/~ben/corba/ -- An interesting private Web site that watches CORBA ORB Core Feature Matrix , CORBAservices Feature Matrix , and CORBA Vendor Platform Matrix
- CORBA Business Objects (Workflow)

- Object Management Research in LASER (Pleiade)
- CORBA Web gateway
- Joint Inter Domain Management
- Towards a Web Object Model - by Frank Manola
- Gabriel D. Minton's papers
- CORBA Design Patterns - by Thomas Mowbray and Raphael Malveau.
- Alan Pope (The CORBA Reference Guide)
- Kate Keahey's Brief Tutorial on CORBA
- Manfred Schneider's CORBA links
- CORBA for Linux
- Segue Software (was Black & White) (SilkPerformer, SilkMeter, SilkPilot, SilkObserver)
- Tom Valesky's Free CORBA page
- Defense Information Infrastructure - Predicting CORBA Performance.
- Benchmarking some ORBs
- GNOME - GNU Network Object Model Environment
- Michi Henning
- Dominique Benech's COBALT: A KQML-CORBA based Architecture for Intelligent Agents Communication
- Live Script and Live Repository