[OMG 93]

"The Common Object Request Broker: Architecture and Specification", Revision 1.2, (Draft), OMG Document Number 93.12.43, 29 December 1993.

[OMG 92]

"Object Management Architecture Guide", R.M. Soley (ed.), Revision 2.0, Second Edition, OMG Document Number 92.11.1, September 1, 1992.

[OUSTERHOUT 94]

"Tcl and the TK Toolkit", John S. Ousterhout, Addison-Wesley, 1994.

[POSTEL 81]

"Transmission Control Protocol - DARPA Internet Program Protocol Specificaton", Postel, J. Internet Document RFC-793, 1981.

[POWELL 93]

"Objects, References, Indentifiers and Equality", Powell, M.L., OMG Document Number 93-7-5, July 1993.

[STALLINGS 93]

"SNMP, SNMPv2 and CMIP — The Practical Guide to Network-Management Standards", Stallings, W. Addison-Wesley, 1993.

[X/OPEN]

"Distributed Transaction Processing, The XA Specification", X/Open Document C193, X/Open Company Ltd., Reading U.K., ISBN 1-8912-05701.

References

[APM 93]

"ANSAware Programmers Manual", APM Ltd., Cambridge, U.K., 1993.

[JAVA]

"Hot Java Home Page", Sun Microsystems, 1995. <URL:http://java.sun.com/>

[DRAHOTA 94]

"DAIS and its use at Hydro-Electric", A. Drahota, D. Hutcheson, ANSAworks 94, Cambridge, UK, April 1994.

[IONA 94a]

"Orbix Programmer's Guide", IONA Technologies Ltd., Dublin, Ireland, version 1.2, February 1994.

[IONA 94b]

"Orbix Advanced Programmer's Guide", IONA Technologies Ltd., Dublin, Ireland, version 1.2, February 1994.

[MICROSOFT 94]

Common Object Model Specification, Microsoft Corporation and Digital Corporation, Draft Version 0.2, October 1994, OMG Document number 94-10-9.

[ODP]

"Open Distributed Processing", ISO/IEC 10746, International Standards Organisation, 1995.

[OMG 95a]

"The Common Object Request Broker: Architecture and Specification", Revision 2.0, OMG, 1995 (in preparation).

[OMG 95b]

"Common Facilities Architecture", Revision 4.0, OMG Document Number 95-1-2, January 1995.

[OMG 95c]

"CORBA 2.0/Interoperability - Universal Networked Objects", OMG Document Number 95-3-10, March 1995.

[OMG 95d]

"Object Models", Draft 0.3, OMG Document Number 95-1-13, January 1995.

[OMG 95e]

"CORBAservices: Common Object Services Specification", OMG Document Number 95-3-1, March 31st 1995.

7.7.1 Acknowledgements

I wish to acknowledge Mike Beasley of ICL Ltd. and ANSA for providing me with some of the code examples and for his pain staking comments on early drafts of this work. Comments of and conversations with Andrew Herbert and Andrew Watson of APM Ltd. and ANSA were extremely helpful. Finally I wish to thank James de Raeve and Paul Tanner of X/Open for providing with information on X/Open's work on conformance testing for CORBA.

Subject to being able raise the necessary sponsorship, X/Open expects by mid 1996 to have extended this work to cover the CORBA 2.0 C and C++ bindings, and subsequently to require the use of the resulting test suite for X/Open branding of CORBA implementations.

7.7 Summary

The Common Object Request Broker Architecture (CORBA) is the central component of the Object Management Group's (OMG) Object Management Architecture (OMA).

Perhaps the most important benefit of CORBA is the way it improves productivity by moving the programmer up several layers of abstraction. CORBA programmers do not require a deep understanding of protocols, Application Programming Interfaces (APIs) or platform internals to build distributed applications.

The key to this is the idea of using an Interface Definition Language (IDL) to define the service being provided by an object. From this IDL definition a stub compiler can generate IDL stubs and skeletons. The IDL stub is placed between the client and the underlying platform. For the client programmer it makes invocation of a remote object look very much like invocation of an object in the same address space. The IDL skeleton is placed between the object or server and the underlying platform. For the server programmer it makes invocations from remote clients look very much like an invocation from the same address space.

Different Object Request Brokers (ORBs) from different vendors can interoperate using the Internet Inter-ORB Protocol (IIOP). If an ORB cannot support IIOP as an internal protocol, then the Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI) make it easy to build a generic bridge application so that the ORB can interoperate using IIOP.

In addition to CORBA, the OMA also defines a number of Common Services such as naming and event channels which are useful for many applications. Common Services will be available for all ORBs. The final component of the OMA are services called Common Facilities. These services are optional, for example, they may be specialised support for a particular application domain such as finance.

CORBA is suitable for a wide variety of application domains and is particularly suited to the task of systems integration and building management applications. Currently CORBA does not provide support for timeliness or continuous media (e.g. video streams) and so could not be used to build some of the applications described elsewhere in this book.

Many of the OMG documents referenced in this chapter are available via anonymous ftp from ftp.omg.org, other documents can be obtained from:

Object Management Group Inc.,
Framingham Corporation Center,
492 Old Connecticut Path, Framingham,
MA 01701-4568, U.S.A.

tel. +1 (508) 820 4300
fax. +1 (508) 820 4303
email: request@omg.org

Some may argue that this is introducing more complexity, but the benefits would seem to outweigh the costs. It is possible that the next revision of the object model will include a notion of interface references [OMG 95d]. One issue which needs to be resolved if such a model were adopted is whether or not there is a need for both object references and interface references.

7.6.5 Memory Management and Garbage Collection

Distributed memory and resource management is a hard problem; it is very easy to end up with wasted memory and orphan objects (objects to which nothing has a reference and are hence never going to be invoked). In section 7.2.5 we saw how Orbix uses the notion of a reference count to try and avoid orphan objects. This relies on programmers calling “duplicate()” and “release()” operations at the right time. Other approaches are possible — this is an area of active research.

A related problem is local memory management. For example, when an invocation arrives at an object the IDL skeleton is responsible for making the parameters available to the objects and may have to allocate memory to do so (e.g. a string). The problem is when to free this memory. If the stubs free the memory when the operation has completed, disaster will result if the object tries to access it later (e.g. it has stored a pointer to it). On the other hand, the object may never want to access the string again. In this case the right thing to do is to free the memory when the object completes the operation.

A safe, but aggressive solution to the above problem is for the stubs to free all memory that they allocate when the invocation has completed. This requires application programmers explicitly to copy any data for which memory is dynamically allocated (e.g. a string). A better solution is to use a language and implementation which can detect when memory has gone out of scope and garbage collect it.

7.6.6 Conformance testing

At the time of writing “CORBA compliance” is by vendor assertion. While it is possible to show that an implementation is not conformant (for example the IDL compiler might not accept a known legal CORBA IDL specification), there are no official standard conformance tests to demonstrate whether an implementation conforms to CORBA.

Having no set criterion against which to measure compliance makes it impossible to evaluate whether or not an ORB is compliant. This will reduce application portability since it is likely that some ORBs which are asserted to be compliant are not compliant and will not be able to run an application which expects a CORBA compliant ORB.

It is also worth noting that there are no conformance checks for an application, so it is impossible to test whether or not an application is expecting features which are not in the standard.

Conformance testing for CORBA is now the focus of active research. In particular X/Open has developed a prototype CORBA 1.2 validation suite as part of a research and development project into automated test suite generation from formal API specifications (the “ADL” project funded by the Information Technology Promotion Agency, an agency of Japan’s MITI). The test suite is available by anonymous ftp from X/Open, but has no status as a conformance measurement tool either in X/Open or OMG.

domains will expose some internal interfaces to make it easier to plug in new protocols.

7.6.3 Object Equivalence

There is no equivalence test on object references defined in CORBA. This is often surprising to people when they first study CORBA. The reason is that only the application can decide what is meant by equivalence. Therefore equivalence operations need to be supported by the application object itself by providing an operation which takes an interface reference and tests whether it is equivalent.

For example, consider a service which is replicated. For normal clients, we would want to consider the instances of the service as being equivalent. However, a management application would need to distinguish between the various instances. So in some senses the object instances are equivalent, in another they are different.

Another example occurs when an object is handed two interface references which happen to be to the same object in another ORB domain. In what senses are these reference equivalent and in what sense are they different (e.g. an invocation might be routed through different gateways)? There is no general way of telling whether these references are equivalent: each may have been past through different gateways and subjected to different processing.

The important principle is that ORBs should not be responsible for tracking equivalence relations — it is the responsibility of applications and application services. Indeed, COS [OMG 95e] defines at least two services which have notions of object equivalence: the transaction service and the relationship service.

For a detailed discussion of object equivalence tests and the problems which can arise the reader is referred to [POWELL 93].

7.6.4 Distinguishing between Interfaces and Objects

Perhaps the most obvious difference between CORBA and ODP [ODP] is the way that ODP treats interfaces as first class entities. In both ODP and CORBA an object can support multiple interfaces of different types. However, ODP treats interfaces as first class entities: clients have references to interfaces which enable them to invoke the operations in that interface. A client needs a separate reference for each interface supported by an object to be able to invoke the operations in that interface. In ODP there is no notion of an object reference.

In contrast, CORBA has no notion of interface references. If a client has a reference to an object it can invoke any operation in any interface supported by the object.

Making interface references first class entities has some important advantages:

- Different interfaces can support different functionality, for example there could be a separate management interface (this is more consistent with IDL in which interfaces are distinguished as separate entities);
- Different interfaces may have different access controls applied, for example not all clients may have access to the management interface.

7.6 Some Practical Issues

This section discusses some issues which arise when using CORBA to build systems.

7.6.1 What applications are suited to CORBA?

A common misconceptions is that RPC systems have poor performance and are blocking. This is not so: modern ORBs are able to make RPCs at or near network speeds (Ethernet) with latencies of 1 millisecond or less. Blocking is easily avoided by using a thread to make an invocation, allowing the client to continue activity while the receiving object deals with the request.

CORBA is applicable to a wide variety of application domains, particularly those which fit the client/server or request/reply paradigm. One of the objectives of CORBA is to make it easier to build management and system integration applications. The idea is that distributed objects can act as gateways to existing heterogeneous information systems, integrating them into a uniform information space. The approach followed is similar to that used to build bridges for interoperability described in section 7.3. Although CORBA technology is still relatively new, the early evidence of its suitability for this type of application is very positive [DRAHOTA 94].

Certain classes of application are less well suited to CORBA. Security services are very necessary for certain application domains, and these are not developed fully at the time of writing. The major omission is no support for timeliness and continuous media. Many applications (for example in telecommunications) need to deal with notions of time and control the scheduling of computations.

CORBA's lack of support for continuous media means that it would be extremely difficult to build the kind of multi-media applications discussed elsewhere in this book (see chapter 8). Supporting such applications in CORBA requires enhancing IDL to describe streams and providing support in the underlying ORB to control Quality of Service or QoS, so that applications can control attributes such as jitter and latency. It is likely that support for streams and QoS will be added to CORBA eventually [OMG 95d].

7.6.2 The need for a framework based approach

In section 7.3 we argued that there will never be a protocol, platform or service which is optimal for all problems. Section7.6.1 argued that distributed objects (particularly CORBA) allow you to wrap, encapsulate and integrate this heterogeneity. This requires an ORB which can support multiple protocols and makes it easy to plug in new protocols. Unfortunately CORBA does not specify the interface between the ORB and the underlying protocol stack.

CORBA blurs the ODP [ODP] computational and engineering viewpoints. A more frameworked based approach (as advocated by ODP), in which the internal structure and interfaces of an ORB were exposed, would make it easier to replace components of the ORB, especially the protocol stacks. This would help greatly with system integration: the legacy system's native protocol could be plugged into the ORB.

The disadvantage of the OMG specifying more of the internal structure of an ORB is that it restricts implementation freedom for ORB vendors. However, doubtless some ORB vendors who are targeting system integration application

transaction context associated with each message to see if the transaction should be committed or aborted.

It is more normal for the services defined in the Common Object Service Specification [OMG 95e] to be “application objects” which do not require special facilities in the underlying ORB.

A “transactional ORB” could be built by using a different Object Adaptor, modified IDL stubs and skeletons, and reusing an existing ORB Core (see figure 7.4). Most of the ORB components would not require changing. Non transactional objects would continue to use the existing object adaptor and IDL stubs and skeletons.

7.5 Common Facilities

So far this is the least well developed of all the components of the OMA. This is not surprising, since the scope of the problem is much greater than those faced by the groups working on CORBA and Common Object Services. The idea is that common facilities will provide application level or user facilities which are interconnected by the underlying plumbing (the ORB and Common Object Services). Such services will have standard interfaces defined in IDL. [OMG 95b] defines the following categories for common facilities.

- User Interface. This includes support for the general display and printing of objects, mechanisms for storing and presenting application help information, and facilities for user desk tops.
- Information Management. This includes support for information storage and retrieval (e.g. SQL and the WorldWide Web), and mechanisms enabling objects to interoperate by exchanging data (e.g. EDI (Electronic Data Interchange), ASN.1 (Abstract Syntax Notation) and various file formats for data interchange).
- Systems Management. Services in this class will deal with the control and management of networks and objects (including physical entities such as routers and nodes, and logical entities such as users and applications). Such services may include the sort of functionality currently provided by SNMP and CMIP [STALLINGS 93].
- Task Management. Services in this category would support such concepts as workflow. Such facilities could provide support for objects migrating around the system from one user to another as a unit of work (e.g. an expense claim being processed). Other services in this category may provide support for agents - programs which migrate around the network acting on different objects. This kind of functionality is supported currently by languages such as Tcl [OUSTERHOUT 94] and Java [JAVA].
- Vertical Market. Services in this class support a specific market such as accounting, computer integrated manufacturing and distributed simulation.

At the time of writing no Common Facilities have been defined. However, the OMG working group has produced a document which scopes the problem [OMG 95b], and is working actively on compound document facilities.

the identities of two transactions (for further discussion of object equivalence see section 7.6.3).

The terminator and coordinator objects are separate to allow an object to pass a reference to the coordinator so that other objects can join a transaction, while keeping private the reference of the terminator (and hence the ability to commit or abort the transaction).

Two other important interfaces defined by the transaction service are the “Resource” interface and the “TransactionalObject” interface. The “Resource” interface contains the operations which the transaction management system will invoke on a resource during the transaction including prepare(), rollback() (or abort) and commit(). For example, the prepare() operation may cause the resource object to write into stable storage all state changes made during the transaction.

The “TransactionalObject” interface contains no operations; it is used to declare to the underlying ORB that the object is transactional. A transactional object is responsible for implementing all the object's operations. When the object is invoked it is responsible for registering a resource object with the transaction “coordinator”. For example, a bank “Account” interface could be declared to inherit from both “TransactionalObject” and “Resource”. The interface itself would declare operations like “credit()” and “debit()”.

When “credit()” is invoked on the “Account” object it would register itself as a “resource” object with the “coordinator”. When the transaction commits or rollsback, the transaction management system will invoke the operations provided by the resource interface to make permanent (or discard) any state changes made by the credit operation.

Many objects may be invoked within the scope of a single transaction. This means that the transaction context needs to be propagated to all these objects. This can be done either implicitly or explicitly. If propagation is explicit, a reference to the managing object (e.g. the “coordinator” or “control” object) is included as one of the parameters in the operation. In this case the argument would appear in the IDL definition of the operation. The alternative is implicit propagation. In this case the ORB recognises that the operation is on a “TransactionalObject” and automatically includes the transaction context as a parameter, even though it is not mentioned in the operation's IDL definition.

The transaction service defines the interface between the ORB and a “Transaction Manager”. The transaction manager is used by the transaction service to implement the transaction management objects such as the “coordinator” and “terminator”. Essentially the transaction manager registers itself with an ORB and then the ORB uses it to ensure that the transaction context is propagated and that the transaction is properly managed. [OMG 95e] defines the mapping between the X/Open DTP “TX” interface and the transaction service operations. It is intended that an application can use either the “TX” interface to drive the transaction manager directly, or the interfaces defined in the transaction service to drive it indirectly.

The transaction service is an unusual object service in that the specification requires the participation of the underlying ORB and specifies interfaces which the ORB must support. In particular the ORB is responsible for propagating the transaction context (for implicit propagation) and also informing the transaction manager each time a request or reply is sent or received within a transaction. The transaction manager will examine the

necessarily) the role object would be co-located with the person object. Assuming the company's "employer" role object already existed, the next thing to do would be to create the relationship object. This is done by invoking the create() operation of an "is employed by" relationship factory with references to the employer and employee role objects as arguments.

The "Role" interface is the most complex defined by the relationship specification. It contains various operations for managing roles:

- It can enumerate the relationships in which it participates by returning a reference to each of the corresponding relationship objects (for example an employer role object will typically be involved in many "is employed by" relationships);
- It provides an operation to return the other role object, if it is given an interface to a relationship object (for example, return the interface to the employee if given a reference to the "is employed by" object);
- It provides an operation for destroying the role object;
- An operation for linking to another role in a newly created relationship (intended for use by the relationship factory).

Relationships and role objects do not require any state to be stored by the objects with which they are associated. So, no state change needs to be made to a person object as the various roles and relationships in which it is involved are changed. This allows immutable objects to be related and those relationships to manipulated without activating the objects themselves which might be useful (for example, in a compound document system).

It is thought that one common use of the relationship service will be to manage graphs of objects (e.g. folder objects in distributed desk tops and compound document architectures). Accordingly, a special set of services are defined by [OMG 95e] to manage these graphs. [OMG 95e] also defines two specific relationships: containment (a one to many relationship) and reference (a many to many relationship). The interfaces defined for these services inherit the general "Relationship" and "Role" interfaces described above.

7.4.5 The transaction service

As explained in section 3.7, transactions provide guarantees which are useful for building dependable distributed applications. The Common Object Service Specification [OMG 95e] defines a transaction service which is designed to be compatible with the X/Open DTP standard [X/OPEN]. Indeed it is intended that applications which use the transaction service should be able to interoperate with applications which conform to the X/Open DTP model.

To create a transaction, a programmer invokes the create() operation of a "transaction factory". This returns a reference to a "control" object which in turn contains operations which return references to a "terminator" object and "coordinator" objects which are used to manage the transaction.

The "terminator" object is used to commit or abort the transaction. The "coordinator" object manages the state of the transaction; this includes registering any objects which are participating in the transaction and any sub (or nested) transactions which are created within the scope of the current transaction. The "coordinator" includes an equivalence test which takes an object reference to a coordinator object and returns true or false depending on whether the reference is for the same object. This allows objects to compare

(deep versus shallow copy). It is implementation dependent whether or not these operations are atomic.

A generic factory interface is defined to create objects. A generic factory will usually invoke application specific code to create an object of a particular type. A given factory represents an implementation at a particular location and is responsible for creating objects at that location. Factories are not special objects: their interfaces are defined in IDL, and to access a factory a client needs an object reference to it.

When a factory creates an object it is responsible for allocating resources to that object, including persistent store. The protocol between a factory and the object it creates is application specific. For example, it may or may not include transfer of state.

To find a factory, the "FactoryFinder" interface is used. This enables a client to find a factory at a particular location. It is intended that there will be different FactoryFinder interfaces defined. The only one defined at the time of writing is the Naming service (see section 7.4.1). Using the naming service as a FactoryFinder is somewhat problematic since there is not explicit support for location in the naming service. It could be supported implicitly by associating particular contexts with a location (e.g. each network host could have its own context). (A trading service could support the concept of location more easily, as it allows arbitrary properties to be associated with object references.)

The result of invoking a "create_object()" operation on a factory will be a reference to the new object. The factory interface is one way of creating new objects. There are other ways, for example, in section 7.2.8 we saw how the BOA can be used to create objects and how Orbix will run a program to create objects "on-demand".

7.4.4 The relationship service

The Common Object Services Specification [OMG 95e] defines a relationship service for managing relationships between objects. Examples of relationships include:

- "is employed by" which could relate an object representing a person and an object representing a company;
- "is contained in" which could relate a chapter object to a book object.

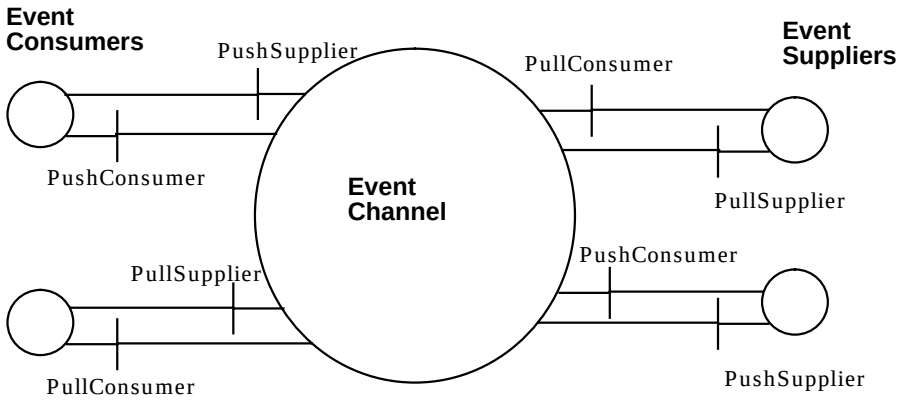
Relationships can be many to many: a company can employ many people and a person could work for more than one company. In addition a relationship can relate multiple objects. For example, [OMG 95e] describes a ternary relationship between a library, a book and a person.

One other important concept defined by the relationship specification is that of "role". Thus a person object can act in the role of "employee" and a book object can act in the role of "container".

Both relationships and roles are first class objects: [OMG 95e] defines the interfaces "RelationshipFactory", "Relationship", RoleFactory" and "Role". The use of these services is best illustrated by example.

Suppose a programmer wanted to set up an "is employed by" relationship between a person and a company. First the employee role would have to be created by invoking the create_role() operation of an employee role factory, passing an object reference to the person object as an argument. The result would be a reference to an role object of type employee. Typically (although not

Figure 7.12: An Event Channel



channel pushes the event into the consumer by calling an operation in its callback interface, informing it that the event has occurred. In the pull model a consumer has to invoke an operation on the event channel asking for the event (if no event has occurred it can either return immediately or block until one does occur). In the push model of interaction for a supplier, the supplier invokes the “push()” operation on the event channel to inform it of an event. This operation takes a single parameter: the value of the event (no results are returned). Conversely in the pull model, the event channel invokes an operation on the callback interface to see if an event has occurred or the “pull()” operation which returns the event value or blocks until one occurs.

[OMG 95e] defines the interfaces which must be supported by event channels, consumers and suppliers to support both pull and push interaction. In addition it defines the (management) interfaces needed to connect producers and consumers to event channels.

It is envisaged that different event channels will have different semantics and support different qualities of service. For example, some event channels may store an event until it is consumed. Others may discard the event if there are no consumers interested in it, or store it only for a short period of time. The later might be appropriate if the event is only valid for a short time (e.g. the latest temperature).

7.4.3 The Lifecycle Services

The lifecycle services defined in [OMG 95e] are designed to support creating, moving, copying and deleting an object. Three lifecycle interfaces are defined:

- The lifeCycleObject supporting copy, move and remove operations;
- The Factory interface used for creating objects;
- The FactoryFinder interface used for finding factories.

An object which supports the LifeCycle interface must provide operations for moving, copying and removing itself. To support the first two the object needs to understand the concept of location. The semantics of all three operations will be application specific. For example if an object is in some way linked to (or contains other) objects (perhaps it is part of a compound document), the “copy” operation may or may not copy all objects to which the object is linked

- Delete the binding between an object and a name in the context;
- Bind a new context to a name in the current context;
- Delete a given context from the current context;
- Resolve a name to a given object (reference).

The resolve operation may return a reference to a naming context. For example, consider figure 7.11 and suppose “root” is an object reference to the root naming context and “foo” is a reference to the name object “/services/printers/rainbow”, then root->resolve(foo) will return a reference to the printer service rainbow.

On the other hand, if “foo” was a reference to the name object “/services/printers”, “bar = root->resolve(foo)” would set “bar” to an object reference to the context service bound to the name “service/printers”. If “ref” is reference to the name object “rainbow”, bar->resolve(ref) would return a reference to the printer service rainbow.

As well as defining the naming context interface which is used to resolve name to object bindings, [OMG 95e] also defines the interface for name objects.

When an object is being initialised it will not have access to a naming service. Thus [OMG 95a] includes facilities for an ORB to provide an object with a way of listing a small number of object names and resolving these object names to an object reference. This facility allows the new object to discover about the rest of the world. For example, one object reference which could be provided like this, is an object reference for the naming service.

In the future the OMG may well standardise a Trading interface as part of Common Object Services. A Trader allows clients to specify constraints on the values of arbitrary properties for a service to which it wishes to bind (e.g. location, cost and ownership). The trader will return to a client one or more object references which have associated with them the specified properties satisfying the specified constraints on values. Servers export their object references to the trader telling the trader what properties they have and the values of those properties. Thus the trader acts as a “match-making” or “brokering” service.

7.4.2 The Event Service

The standard CORBA interaction model is one in which a client sends a request to an object and waits for a reply. Threads can be used to allow other activity to continue in the client before the reply has arrived. However, sometimes a more asynchronous interaction model is convenient. For example, a source program might be changed. The program might just report that it has been modified and be completely unaware of a CASE tool which is notified of the event and then rebuilds parts of the system which used the program.

To support this type of interaction [OMG 95e] defines an event service. Conceptually an event service provides an event channel between one or more consumers and one or more producers. As shown in figure 7.12, to use an event channel consumers and suppliers provide the event channel with a call back interface. The event channel is shown as a single object which is how it would appear to programmers of consumer and supplier objects. However, in practice it would probably be implemented by multiple collaborating objects.

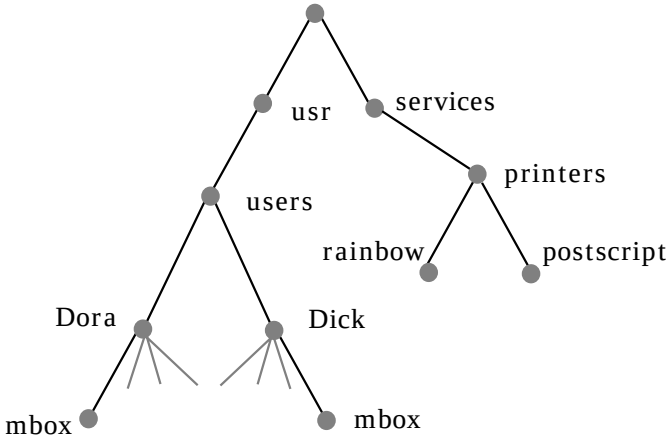
For both consumers and suppliers of events there are two basic modes of interaction: “push” and “pull”. In the push model of interaction an event

promote this the OMG is working on the specification of standard interfaces for “Common Object Services” or COS. At the time of writing the OMG has COS specifications in existence or under development for several services including: naming, event notification, lifecycle services, persistence, security, transactions, relationships and concurrency control. This section gives an overview of some of the services which are defined in [OMG 95e].

7.4.1 The Naming Service

In section 7.2.8 we saw how Orbix clients use the “_bind()” operation to obtain a reference to a named object using facilities which are provided by the ORB. A naming service can be used to keep track of the binding of names to objects. The naming service makes use of context. Contexts are rather like directories in a file system. A context contains a set of name to object bindings; in addition other contexts can be bound to a given context to form a naming graph. Consider the naming graph in figure 7.11, it shows a context “usr” to which the context “users” is bound; within the context users are two further contexts named “dick” and “dora”. Both contexts and objects can be named and can be referred to either by their simple or compound names. Thus “mbox” is the simple name of an object (ambiguous unless the context is known); “usr/users/dora” is the compound name of a context.

Figure 7.11: A naming graph



The contexts “dick” and “dora” each contain a number of name to object bindings. Both have a binding for a name mbox. Unlike in a file system there is no general way of telling if the object bound to the name “usr/users/dora/mbox” is the same as the object bound to “usr/users/dick/mbox”. The reasons for this are discussed in section 7.6.3.

Within the COS naming service names and contexts are treated as “pseudo objects”. This means that the language binding enables programmers to manipulate names and contexts exactly as if they were ordinary CORBA objects. However, there is no requirement to implement them as ordinary CORBA objects: implementations may make different choices for efficiency. This also has the benefit that the internal representation of names and contexts are hidden (object-oriented principle of encapsulation).

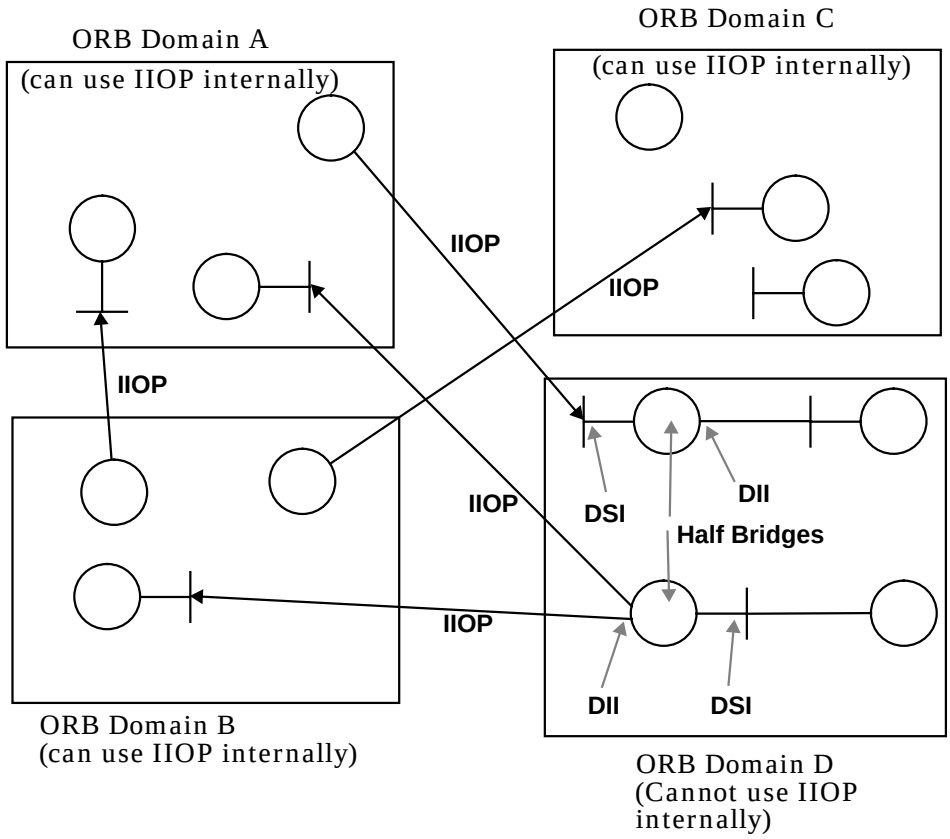
The naming service defines the context interface containing operations which:

- Bind an object to a name in the context;

returned as results. For example how is a reference to an object O2 in domain B represented in domain A? A simple answer is to say that it is actually a reference to the bridge which knows how to translate the reference to O2 in domain A to a reference in domain B. The problem is further complicated if the reference to O2 is handed out from domain A across another bridge to a third domain. [OMG 95c] lists several solutions to this problem and various trade-offs which can be made.

All CORBA 2.0 ([OMG 95a]) conformant ORBs are required to support IIOP either as a native protocol or by providing a half-bridge to IIOP. Figure 7.10 shows ORBs interoperating using a mixture of native IIOP and half-bridging to IIOP.

Figure 7.10: ORBs interoperating using IIOP and half bridges



ORBs can also interoperate with other object platforms. Indeed it is important that they can, if they are to be used a vehicle for systems integration. At the time of writing a group within the OMG is investigating CORBA / COM [MICROSOFT 94] interoperability.

7.4 Common Object Services Specification

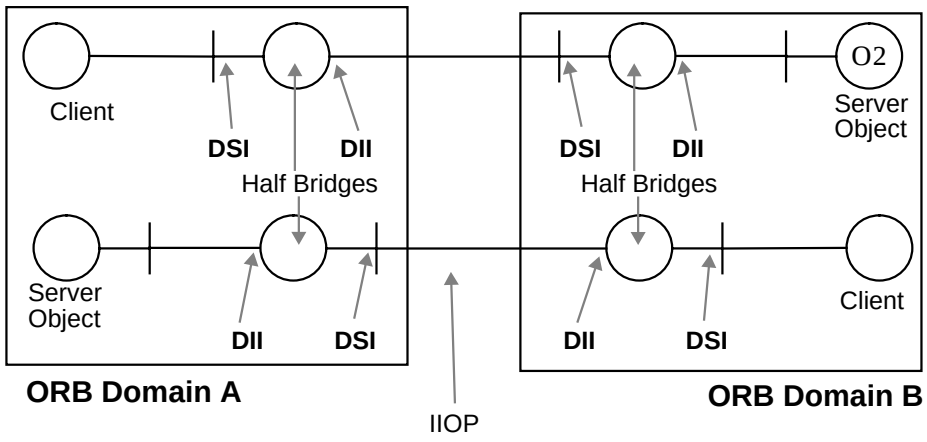
There are certain kinds of services which are generally useful, regardless of the application domain; examples include persistent storage and naming services. Rather than programmers having to re-implement such services each time they are needed, it would be better to re-use existing implementations. To

(Internet Inter-ORB Protocol). GIOP specifies a common data representation (or “on the wire” format) and also the messages and the format of those messages which can be sent between the client and server (object receiving the request) ORB. Included in the specification is a standard representation of object references. GIOP also specifies a number of requirements for its transport protocol. This include reliable at-most-once delivery of data, no re-ordering of data and support for fragmentation (to allow GIOP to send messages which otherwise might exceed the maximum packet size).

Some ORBs will be able to support IIOP as a native protocol, either as the only protocol they use for communication between objects, or by supporting multiple protocol stacks simultaneously. The latter approach requires the ORB to recognise which protocol a message is using — communication with objects in other ORBs will use IIOP. Unfortunately it is unlikely that all ORBs will be able to support IIOP as a native protocol. For such ORBs [OMG 95c] specifies a different approach to interoperability — bridges.

A bridge is an application which allows objects using one ORB to communicate with objects using another ORB even though the two ORBs do not use a common protocol. If an ORB cannot support IIOP as a native protocol, it can still interoperate by providing a so-called “half-bridge” between its internal protocol and IIOP. In this way two ORBs can interoperate even though neither can support IIOP as a native protocol. Figure 7.9 shows two ORBs interoperating using half-bridges to IIOP (the two half-bridges work together to form a full bridge). The term domain is used to distinguish between objects using the two different ORBs.

Figure 7.9: Two ORBs Interoperating using Half Bridges



Notice that the bridges in figure 7.9 use the Dynamic Invocation Interface (DII) (see section 7.2.6) and its corresponding server-side equivalent the Dynamic Skeleton Interface (DSI). This is because the bridges are generic: they can be used to invoke any object regardless of what interfaces that object has. The bridges could be built by using IDL stubs and skeletons, but in that case the bridges could not be used for any object except those whose stubs were incorporated into the bridges. Thus adding a new object to either of the domains in figure 7.9 would entail adding a new pair of bridges to make it accessible in the other domain. Using the DSI and DII avoids this problem.

One of the main difficulties with bridges is how to map between different representations of object references which are passed as parameters or

When the program is run it uses the C++ operator “new” to create objects of type AccountList, “SBankMgmt” and “SBank_i”. The code for the “SBank_i” class is shown in section 7.2.5. The server uses the name “odpBank” for the first “SBankMgt” and “SBank” objects it creates, and would used different names for other banks.

In section 7.2.5 we saw how a client uses the “_bind()” operation to obtain an object reference to a named server. To obtain a object reference to the above “SBank” server, a client would have to invoke the “_bind()” operation with the parameter “odpBank:SBank”. This would cause Orbix to run the server program registered as “SBank” (using “putit”) and return an object reference to the object named “odpBank” created by that server. Further discrimination is possible: “_bind()” can also take the name of the machine on which the object is resident (e.g. outlaw.ansa.co.uk). This will cause Orbix to run the server program on the named host.

Once the server has finished creating the bank objects it calls the “impl_is_ready()” operation; the objects created by the program are now available to clients.

The “impl_is_ready()” operation returns only after no clients have used the objects for sometime — Orbix times out. In this case the program exits and the objects would be destroyed. As explained earlier in this section, other options are possible in CORBA, including arranging for the objects to store state in a persistent store, so they can be activated when required. For an explanation of how Orbix supports this the reader is referred to The Advanced Programmer’s Guide [IONA 94b].

Although CORBA specifies the details of only a single object adapter (the BOA), some environments may contain many object adapters. Thus, [OMG 95a] includes details of how an object may select its object adapter during initialisation.

7.3 Interoperability

An important goal of CORBA is to promote interoperation between different ORBs, so that clients in one ORB can invoke objects in a different ORB. This will allow programmers to build applications which use objects running on different ORBs provided by different vendors. The early CORBA specifications (e.g. [OMG 93]) said very little about interoperability. This is being addressed in the CORBA 2.0 specification [OMG 95a] which will follow the approach described in [OMG 95c].

Before considering the approach taken to interoperability it is important to realize that there will never be a platform, protocol or service which will be suitable for all applications, thus heterogeneity is inevitable. For example some ORBs may be intended for computationally intensive applications and may use a communication protocol based on shared memory to achieve invocation times of the order of microseconds. Other ORBs may be targeted at long lived applications which are needed to maintain data which has a life of many years. Given this constraint it is not practical to specify the internal communication protocol that should be used within an ORB. Rather the approach taken in CORBA 2.0 is based on a common communication protocol which can be used to communicate with objects running on different ORBs.

The protocol specified by CORBA 2.0 is called GIOP (General Inter-ORB protocol) and its mapping to the Internet’s TCP [POSTEL 81] is called IIOP

- Activate only a single object per process;
- Activate multiple objects per process — this policy is shown in figure 7.8;
- Run a new process to execute each operation;
- Allow an entity outside the BOA to activate the object.

The last policy allows much flexibility, it requires only that the activated object registers itself with the BOA. (This would correspond to registering with the ORB daemon described in section 7.2.4.)

Objects are responsible for deactivating themselves, by invoking the BOA's "deactivate_obj()" operation.

Figure 7.8 shows how the "simple bank" example might be started using the facilities provided by Orbix [IONA 94a]. A shell script to start the program is placed in the implementation repository using the Orbix "putit" utility: arguments to putit include the full pathname of the executable file, and the name of the server program in this case "SBank". This script will be run by Orbix when any client tries to access the "SBank" service.

Figure 7.8: Creating Objects in Orbix

```
// SBankServer.cc
// The executable file generated from this code should be
// registered(under the name 'SBank') using
// the 'putit' command.

#include <unistd.h>

#include <stream.h>
#include <CORBA.h>
#include <exception.h>
#include "Account_i.h"
#include "SBankMgmt_i.h"
#include "SBank_i.h"
#include "SBankServer.h"

int main()
{

 // create an Account List object, to be shared
 //  by the SBankMgmt and SBank objects
 AccountList* odpBankList = new AccountList;

 // create SBankMgmt and SBank objects
 SBankMgmt_i* odp BankMgmt = new SBankMgmt_i("odpBank",
                                              odpBankList);
 SBank_i* odpBank = new SBank_i("odpBank", odpBankList);

 // Creation of other named banks here (code deleted)

 Orbix.impl_is_ready();

 cout << "server exiting" << endl;
}
```

7.2.8 The Basic Object Adapter

[OMG 93] specifies the details of a single object adapter called the Basic Object Adapter or BOA. An ORB may provide additional object adapters. In 7.2.4 we looked at how an object adapter is used to activate an object; recall that in practice it may well be implemented in two parts: part as a library linked in with an object along with the IDL skeleton; part as a component of an ORB daemon process running on each host. This section looks at other functions provided by the BOA as well as specific options for object activation.

The main functions of the BOA are:

- Generation and destruction of objects;
- Activation and deactivation of objects;
- Invocation of objects through the IDL skeleton.

In addition the BOA provides very minimal support for security. An object can invoke the get_principal() operation. This will return a reference to an authentication service responsible for the current invocation. The authentication interface itself and the details of any access controls are not specified in [OMG 93].

Just as interface definitions are stored in the Interface Repository, the BOA expects implementations to be stored in an Implementation Repository. A typical Implementation Repository might store compiled programs or shell scripts to start and initialise objects. The signature for the BOA's create operation is shown below:

```
Object create(in ReferenceData id, in InterfaceDef IntDef,
              in ImplementationDef ImpDef);
```

The parameters IntDef and ImpDef are respectively references to objects in the Interface and Implementation repositories. The BOA implementation is free to use the id parameter in any way it chooses.[6] The choice will be affected by the environment provided by the host computer. For example, it might be the name of the file which is used to store the persistent state of the object; it is up to the object how it stores it state in this file — the BOA operation "get_id()" can be used to obtain the name of the file.

The "create()" operation returns an object reference to the new object. The "dispose()" operation, is used to destroy the object. Once the "dispose()" operation is invoked on an object reference the ORB will behave as if the object never existed. The implementation of the object itself is responsible for deallocating any persistent storage.

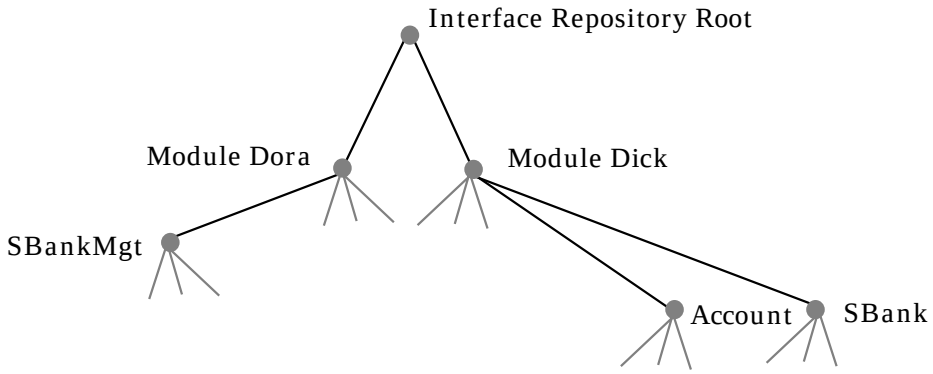
In section 7.2.4 we described how the BOA must activate a passive object before it can receive an invocation. Object activation takes place in two stages. In stage one the BOA activates the implementation. This typically corresponds to starting the program which contains the object. A program may contain many objects, so in stage two of activation the BOA activates the particular object which is needed.

The BOA's activation policy describes how an object may be mapped to a process (or in ODP terminology a capsule [ODP]). [OMG 93] requires the BOA to support four policies:

---

6. Different implementations may make different choices, so this may lead to some portability problems.

Figure 7.7: Use of modules in the interface repository



Each object in an interface repository has a unique (within the repository) identifier, as well as a name. For example, the unique identifier of if the operation object named “Debit” might be Dick::Account::Debit (the actual structure of the unique identifier is opaque). The interface repository supports operations which will return a reference to an object associated with a given unique identifier. Another operation is supported which will return a list of objects which have the same given name (e.g. all objects called “Debit”). The reader is referred to [OMG 93] for details of these and other operations provided by the interface repository.

The interface repository may be implemented as an application object, like the “SBank” object. Alternatively, it may not exist as an object at all, rather the ORB may manage all the relevant information and make it appear to the application programmer that one or more interface repository objects exist.

An ORB is required to support a “get_interface()” operation which can be applied to any object reference. Given an object reference of type X, the result of applying the “get_interface()” operation to it is a reference to an interface object X, which is contained in the interface repository and defines the interface type X.

It may seem as though the interface repository is adding extra complexity for the CORBA programmer to deal with. However, most CORBA applications can be written without the application programmer ever having to use the interface repository. Indeed, at the time of writing, most ORBs work without using the Interface Repository at all. It is there if it is needed and may be useful for certain classes of application such as CASE tools, class browsers, gateways and applications determining type equivalence.

The interface repository can also be used by the dynamic invocation interface for type checking. IDL stubs can have type checking information encoded into them by the stub compiler, since they are specific to a particular interface. In contrast, the DII can never know what interfaces it will be required to invoke, so it needs to have access to interface specifications (contained in the interface repository) if it is to do any type checking.

The early CORBA specifications (e.g. [OMG 93]) did not define any operations for installing new objects in the interface repository. However, at the time of writing, a new specification is under development [OMG 95a] which is likely to include operations for installing objects in the Interface Repository.

the “typedefs”, exceptions and two interfaces: “Account” and “SBank”. Everything declared in the scope of the interface “Account” is attached to the node representing the “Account” interface: the typedef for the struct and the three operations. Attached to the nodes representing an operation will be one node for every argument declared in the scope of the operation. These are the leaves of the AST. Each node in the AST will also have a number of attributes associated with it. Every node will have a attribute which is its name. In addition a node like the operation “Debit” will also have attributes such as its return type (void) and list of exceptions it can raise (“InsufficientFunds”).

The Interface Repository treats every node in the AST as an object and provides a set of operations to discover the relationship an object has with other objects. For example given a reference to the operation object, “Debit”, you could invoke the “within()” operation to discover in which interface this operation was declared. The result would provide an object reference to the interface object, “Account”. Operation objects support operations which return references to parameter objects, references to exception objects (if the operation can raise any) and much other useful information about the operation.

Conversely given a reference to the interface object “Account”, the “contents()” operation could be invoked to return a list of interfaces to everything declared within the scope of the interface “Account”. Included in this list would be a reference to the operation object, “Debit”.

The definition of the Interface Repository in [OMG 93] includes a primitive object type for each IDL primitive including: interface, operation, parameter, typedef and exception. It also defines the type codes for all the basic IDL types and how the type codes are constructed for complex types like structures. Thus, suppose an operation, called foo, returns a value which is a structure type. If you ask the operation object, foo, (contained in the interface repository) for the type of the result value it returns, it will return a type code for the structure. The type code is parseable so that the fields in the structure can be determined.

An Interface Repository can be responsible for managing a great many IDL definitions written by different programmers. To provide a way of managing this it uses a notion of a module. For example, suppose two programmers, Dick and Dora, are working together on the Simple Bank project. Dick is responsible for the interfaces “Account” and “SBank”, while Dora is responsible for the management interface, “SBankMgt”. Both Dick and Dora could be assigned separate modules. The interface would then be contained in their respective modules. This is shown in figure 7.7.

Thus each interface repository appears as a container object, providing a number of operations to manage objects which it contains. If its “contents()” operation is invoked it will return a list of object references to the module object it contains. Similarly if the “contents()” operation is invoked on a module object, it will return a list of object references including references to other module objects and interface objects which it contains. Thus, starting from the root object (the interface repository object), it is possible to navigate through a structured space to any specific interface object which can provide all the information contained in the IDL specification of the given interface. Operations such as “contents()” retrieve information about objects one level down from the current object.

might have to be rebuilt to incorporate an IDL stub for the new class each time a new class were added to the library. By using the DII it can browse any class without have to incorporate the IDL stub. In section 7.5. we will see how the DII is vital for building bridges or gateways between different ORB. Indeed, the next revision of CORBA (CORBA 2.0 [OMG 95a]) will include corresponding facilities for servers: the Dynamic Skeleton Interface, specifically so that bridges can be supported.

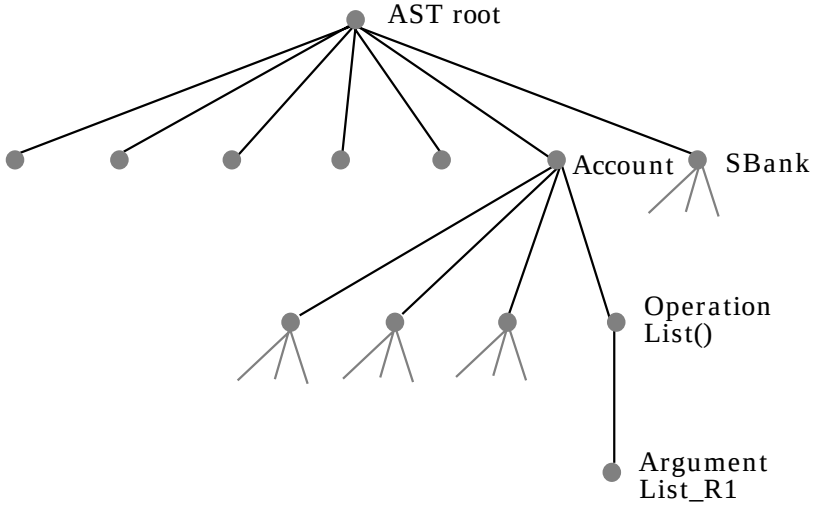
The DII's support for asynchronous and multiple asynchronous invocations might also be useful if a thread package is not available. In many cases a threads package is available, so the programmer can make multiple invocations in parallel using IDL stubs. Client side execution can be continued by creating a thread to handle each request (which will block until the reply is received) and one extra to run in the client. The latter approach to asynchrony will generally require the programmer to do less work as fewer calls to the underlying API will be required.

In summary there are certain classes of application which might be best implemented by using the DII; the DII's support for asynchrony would be useful in the absence of proper support for threads. However, the IDL stub approach to invocation requires much less work on the part of programmers: they only have to make one call to invoke an object, rather than several calls to an API. Hence there is less opportunity for programmer error using IDL stubs. So the approach to CORBA programming should be to use IDL stubs wherever possible, using the DII only when absolutely necessary.

7.2.7 The Interface Repository

The Interface Repository is a service which provides access to interface definitions. It can be used both by the internal components of the ORB and also ORB applications. The best way to understand the service provided by the Interface Repository is to regard each IDL definition as an Abstract Syntax Tree or AST.

Figure 7.6: AST Fragment for figure 7.2



Part of the AST for the IDL specification in figure 7.2 is given in figure 7.6. Attached to the root node is everything declared in the global scope in this case

language binding makes it looks as though it is implemented by the remote object itself. The inputs to the “create_request()” operation include:

- The operation name;
- The argument list;
- A variable to contain the result returned by the operation.

The result of the “create_request()” operation is an object reference to a “request” object which can be used to control the invocation. The request object supports a number of operations:

- invoke()
- send()
- get_response()
- delete()

The “invoke()” operation uses the mechanisms provided by the underlying ORB to deliver the request to a remote object. The remote object, together with its ORB core and IDL skeleton will deal with the request as described in section 7.2.4 (just as if it had come from an IDL stub). If the “invoke()” operation terminates successfully the variable containing the result will have been updated and any “out” or “inout” arguments contained in the argument list will have been changed.

The “delete()” operation destroys the request object and reclaims any memory associated with it.

The “send()” and “get_response()” operations can be used to make asynchronous invocations. When the “send()” operation is invoked on a request object, the call returns to the caller without waiting for a response to be delivered. The caller can then continue execution and retrieve the response at some later stage. The “get_response()” operation is used to discover if a reply has been received from the invocation. If “get_response()” indicates that the operation is done, the return value and “out” or “inout” parameters will be set in exactly the same way as if “invoke()” had been used to make the invocation.

The DII provides two operations for making multiple invocations simultaneously: “send_multiple_requests()” and “get_next_response()”. The “send_multiple_requests()” operation takes an array of request objects and invokes the “send()” operation on each one. Once “send_multiple_requests()” has terminated there will be multiple request executions taking place; the degree of parallelism and ordering of executions is system dependent.

The response can be retrieved by invoking the “get_next_response()” operation. This will return a pointer to a completed request object. If multiple request objects have completed there is no guarantee which one will be returned.

In some senses the dynamic invocation mechanism is less abstract that the IDL stubs. It hides fewer aspects of distribution than the IDL stubs, requiring programmers to do more work that if they had used IDL stubs. For example, programmers using the DII must explicitly build up requests, possibly invoking other (remote) objects to discover what parameters are required. However, the dynamic invocation interface is sometimes a very useful tool for building certain classes of applications.

One class of applications which can be built easily using the DII are class browsers. If the IDL stub approach to invocation were used, the class browser

provided and returns an object reference which can later be used to invoke the named object. The ORB may set various internal data structures to deal with the invocation when the reference is used. The SBank::_bind() operation can raise a number of exceptions, including standard CORBA exceptions such as NO_RESOURCES and INITIALIZE (ORB initialization failure). Any exception will be contained in the CORBA environment variable IT_X and is handled by the block of code between the CATCHANY and ENDTRY statements which prints the exception and terminates the program.

Next the "Access" operation is invoked on the SBank object, recall that this returns a reference to an account object. In addition to the standard CORBA exceptions, this operation can also raise the exceptions "NoSuchAccount" and "InvalidPin". These are handled by the block of code between CATCHANY and ENDTRY.

The rest of the program uses the same principles to invoke one of the "Account" object's operations. At the end of the program the "release()" operation is invoked on the "Account" object which will decrement the object's reference count (maintained by the ORB). This may cause the Account object to be deleted.

7.2.6 The Dynamic Invocation Interface

The Dynamic Invocation Interface (or DII) provides clients with an alternative to using IDL stubs when invoking an object. The DII is useful for certain specialised kinds of applications such as bridges (see section 7.3) and class browsers. It is anticipated that most CORBA clients will use IDL stubs. An object receiving a request cannot tell which of the two mechanisms is being used by the client.

In section 7.2.4 we saw how the IDL stubs enable a client to use the same syntax for remote invocations as they use for local invocations. Using the DII involves the client in a number of extra steps.

First the client needs to build up an argument list. Each element in the argument list consists of the following IDL structure:

```
struct NamedValue{
 identifier name; //argument name
 any argument; // the argument
 long len; // size of the argument (in bytes)
 Flags arg_modes; // in, out or inout
}
```

The CORBA IDL type "any" can express any IDL type. In C it is implemented as a structure consisting of two fields:

```
typedef struct CORBA_any {
     CORBA_TypeCode _type;
     void *_value;
} CORBA_any;
```

The type code indicates the type encoded as an "any" (e.g. a string representing the type name). The "_value" field is a pointer to a memory location contain the value of the argument.

The next step in using the DII is to invoke the "create_request()" operation on the remote object. This operation is implemented by the ORB, but the

```
    cerr << "Access(" << argv[2] << ", " << argv[3]
         << ") failed, reason: " << IT_X << endl;
    exit(1);
} ENDTRY

// Otherwise, take the appropriate action, depending on what command
// has been given.

float value;
switch(cmd_no)
{
// Credit
case 0:
    (void) sscanf(argv[5], "%f", &value);
    TRY {
         Acc->Credit(value, IT_X);
    } CATCHANY {
         cerr << "Credit(" << form("%.2f", value)
              << ") failed, reason: " << IT_X << endl;
    } ENDTRY
    break;

// Debit
case 1:
    (void) sscanf(argv[5], "%f", &value);
    TRY {
         Acc->Debit(value, IT_X);
    } CATCHANY {
         cerr << "Debit(" << form("%.2f", value)
              << ") failed, reason: " << IT_X << endl;
    } ENDTRY
    break;

// List
case 2:
{
    Account_AccountRecord lresult;
    TRY {
         Acc->List(lresult, IT_X);
    } CATCHANY {
         cerr << "List() failed, reason: " << IT_X << endl;
         break;
    } ENDTRY
    // If operation succeeds, display account details
    cout << "Details for account" << endl;
    cout << "\towner: " << lresult.owner << endl;
    cout << "\tbalance: " << form("%.2f", lresult.balance) << endl;
    cout << "\tlastaccess: " << lresult.lastaccess << endl;
}
break;
}
Acc->_release();
}
```

The first ORB operation which the program invokes is SBank::_bind() (details of this operation are discussed in section 7.2.8). If this is successful it takes the name

```
{
    PrintUsageAndDie(argv[0]);
}

// Check whether the "command" parameter is a valid command.
int cmd_no = INVALID_CMD;
for (int i = 0; cmd_list[i] != (char*) NULL; i++)
{
    if (strcmp(argv[4], cmd_list[i]) == 0)
    {
         cmd_no = i;
         break;
    }
}

// If the command was not a valid one, print error message & quit.
if (cmd_no == INVALID_CMD)
{
    PrintUsageAndDie(argv[0]);
}

// If the wrong number of arguments have been given for this command,
// give error & quit.
if (argc < minargs[cmd_no] || argc > maxargs[cmd_no])
{
    PrintUsageAndDie(argv[0]);
}

// Read in the account & PIN */
AccountNumber accno;
(void)sscanf(argv[2], "%ld", &accno);
PersonalIdentificationNumber pin;
(void)sscanf(argv[3], "%ld", &pin);

// Try to access the given account using the given PIN

// First, need to import the SimpleBank interface,
// which provides the Access operation.

SBank* Bank;

TRY {
    // bind to SBank object
    char name[20];
    sprintf(name, "%s:SBank", argv[1]);
    Bank = SBank::_bind(name, nil, IT_X);
} CATCHANY {
    cerr << "Bind to object failed" << endl;
    cerr << "Unexpected exception " << IT_X << endl;
    exit(1);
} ENDTRY

// Access the account.
Account* Acc;
TRY {
    Acc = Bank->Access(accno, pin, IT_X);
} CATCHANY {
    // If the access fails, print the reason & quit
```

incorrect. If no errors have occurred, the "_duplicate()" operation is invoked on "the_account", and finally the object referenced is returned.

The use of the "duplicate()" operation needs further explanation. An ORB provides two operations for allocating and releasing resources associated with object references: respectively "duplicate()" and "release()". Orbix [IONA 94a] keeps track of how many references there are for an object, this is called the reference count. When an object is created, its reference count is one; should its reference count ever drop to zero it is deleted. This helps to avoid orphan objects which nobody will ever use (i.e. the build up of garbage). The "duplicate()" operation increments the reference count for an object while the "release()" operation decrements the reference count. This illustrates one way in which an ORB can solve the distributed garbage collection problem; other ORBs may use different methods.

Orbix calls the "release()" operation on all object references returned from operations. So, if the "duplicate()" operation is not called on an object before returning a reference to it, the reference count could drop to zero and the object would be deleted. In the above example, although the operation always returns "the_account", "duplicate()" is called only if there is no exception raised.

The other interface defined in figure 7.2 was "Account". The implementation of the corresponding "Account_i" would have the same structure as the implementation shown for "SBank_i".

7.2.5.1 *The Simple Bank Teller Program*

The code below is for a simple bank teller which uses both the "Account" and the "SBank" interface. It also illustrates how CORBA programmers can use exceptions to deal with errors, in this case using the "TRY, CATCHANY, RETRY" macro facilities provided by Orbix™.

```
// SBankTeller.cc
#include "SimpleBank.hh"
#include <stream.h>

#include <stdio.h>
#include <stdlib.h>

// Commands that teller program can execute on an account
char *cmd_list[] = {"credit", "debit", "list", (char*) NULL};

// Min & max number of arguments for each command
int minargs[] = {6, 6, 5};
int maxargs[] = {6, 6, 5};

const int INVALID_CMD = -1;

void PrintUsageAndDie(char* prog)
{
 cerr << "usage: " << prog << " bank acct pin credit value" << endl;
 cerr << " " << prog << " bank acct pin debit value" << endl;
 cerr << " " << prog << " bank acct pin list" << endl;
 exit(1);
}

main (int argc, char **argv) {
 // Check program has been given enough parameters. If not,
 // give error & quit.
 if (argc < 4)
```

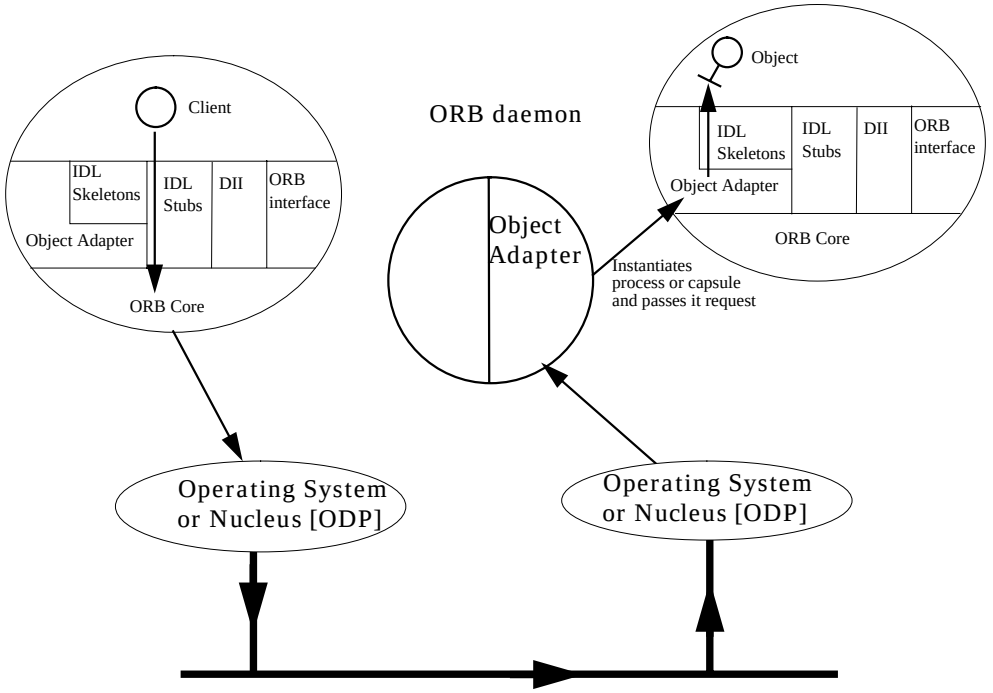
stub compiler). The IDL definition for the “SBank” interface is given in figure 7.2.

```
// SBank_i.cc
// Class SBank_i implements the SBank IDL interface

#include “SBank_i.h”
#include <stream.h>
#include <string.h>
#include <memory.h>

// constructor
SBank_i::SBank_i(char* name, AccountList* new_list) :
SBankBOAImpl(name)
{
 my_list = new_list;
}

// destructor
SBank_i::~SBank_i() {
}

// implementation of Access
Account* SBank_i::Access(AccountNumber acct,
 PersonalIdentificationNumber pin, Environment& env)
{
 Account_i* the_account = my_list->Find(acct);
 if (the_account == (Account_i*) NULL)
 {
 NoSuchAccount* e = new NoSuchAccount;
 env = e;
 }
 else if (the_account->pin != pin)
 {
 InvalidPin* e = new InvalidPin;
 env = e;
 }
 else
 {
 the_account->_duplicate();
 }
 return the_account;
}
```

The constructor for the class takes an AccountList object (containing the accounts) and a name. The name parameter allows clients to bind to a named object. This is explained further in section 7.2.8.

The “Access” operation takes an account number and a PIN, returning an object reference which can be used to access the account. The CORBA environment variable is used to raise an exception if an error occurs. First the operation looks up the account in its list of accounts, returning the exception “NoSuchAccount” in the CORBA environment variable if it cannot find an account with a matching number. Next it checks the PIN is correct, returning the exception “InvalidPin” in the CORBA environment variable if the PIN is

Figure 7.5: Use of ORB Daemon to invoke an object



is listening on a well known UDP port. The ORB daemon would tell the client ORB if the object was already active and listening on its old UDP port. Alternatively it would return a new UDP port if the object was passive and had to be reactivated (the old UDP port might not be available). Once the client's ORB core is sure of the UDP port on which it can contact the remote object, it will send the request as described above.

It is unlikely that a client would contact the ORB daemon for a remote object for each invocation. Rather it might contact the service the first time it uses an object reference, if a communication error occurs (e.g. no reply), or if it had not used the object reference for a long time. This means that if a client sends requests frequently to an object the ORB daemon will be involved infrequently.

In concluding this section we note that the IDL stub and skeleton hide many of the unpleasant details of distributed programming from the programmer, satisfying many of the transparency requirements discussed in section 7.1. For client programmers, the syntax of remote object invocation is very close to that of a local (C++) object. They have to be prepared to handle extra exceptions, but otherwise do not have to write any more code than would need to be written for local invocation. Programmers of remote objects only have to implement one operation for each of those defined in the IDL for the interface. Also they may have to write some small amount of code to deal with activation and passivation. Precisely what this code looks like will be dependent on the implementation of the ORB and specifically the Object Adapter (see 7.2.8).

7.2.5 An example CORBA Application

This section gives further details of the bank example to demonstrate how CORBA applications can be built. The C++ source code for the “SBank_i” class is shown below; this class inherits from the “SBank” class (generated by the

Once the remote object's ORB core has received the buffer and correctly re-assembled it (if it was fragmented), the object reference is unmarshalled to see to which of the objects the request should be delivered. If the object is not resident in the capsule, an error is returned to the client side ORB core. The remainder of the buffer is the passed to the IDL skeleton for the object. The IDL skeleton unmarshals the operation name, "List", and the parameters for the operation. It then invokes the object's "List" operation passing it these parameters. A C++ implementation of the "List" operation is shown below. Note that the implementation class "Account_i" inherits from its interface class "Account" which is generated by the stub compiler from the IDL definition of the interface.

```
void Account_i::List(Account_AccountRecord& List_R1,
                     Environment& env)
{
 List_R1.owner = strdup(owner);
 List_R1.balance = balance;
 List_R1.lastaccess = lastaccess;
}
```

Once the "List" operation has executed, control returns to the IDL skeleton. The events which occur to send the results back to the client are symmetrical with those which occur to send a request to an object. The IDL skeleton obtains a buffer and marshals any results which are needed into that buffer. The results needed include any "out" or "inout" parameters (in this case "List_R1"), the CORBA environment variable and the value returned by the operation. The CORBA environment variable is used to carry exception information; it should not be confused with Unix™ environment variables. The buffer is then passed to the ORB core to be sent to the client. The ORB core needs to determine the IP address and UDP port to which the buffer must be sent. This may have been contained as a field in the object reference received as part of the original request, or it may have been sent as a separate parameter.

Eventually the client's ORB core will receive the reply buffer. It checks that the client is indeed present in the capsule and passes the reply buffer to the appropriate IDL stub. The IDL stub unmarshals the results and returns control to the client. Client programmers see this as execution of "List" terminating. What is actually happening is termination of the IDL stub's "List" operation rather than the remote object's "List" operation, but this distinction is invisible to them.

Invocation of a remote object which is passive is similar to the above, but includes a number of extra steps in which the remote object is activated so that it can receive the request. In CORBA object activation is the responsibility of the object adapter. In practice the object adapter may well be implemented as a library component (linked in with the object and IDL stubs) and a second part being a component of an ORB "daemon" process running on each host. The part of the object adapter running in the ORB daemon will be shared by all objects running on a host.

To activate an object the daemon might instantiate a process, running an implementation of the object with any persistent state for the object retrieved from a persistent store.

Once the client side ORB core has determined the IP address and UDP port of the remote object, it would contact the ORB daemon on the remote host which

probably in the same process as the IDL stub and client (perhaps linked in as a runtime library). The ORB core's job is to locate the remote object, prepare it to receive the request and finally hand it the request (in the form of a buffer) so that it can execute the appropriate operation.

There are two cases to consider:

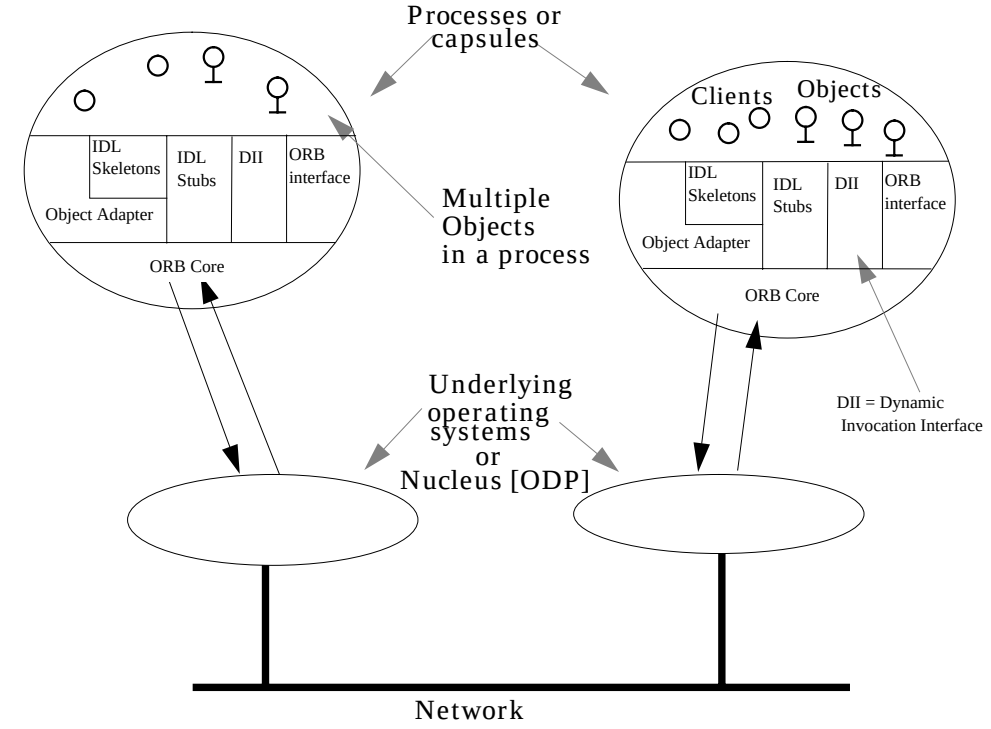
- The case when the remote object is already active;
- The case when the remote object is passive.

In the former case the task of the ORB core in the client process is to locate the IP address of the remote object's host machine and the port number on which the remote object's ORB core is listening. Once it has done this, the client side ORB core will send the remote ORB core the buffer, fragmenting it if the buffer is too large. The client side and remote object ORB cores need to deal with any failures which may occur such as lost or duplicated packets, giving the illusion of reliable communication.

The task of locating the remote object's ORB core may be quite complicated if the remote object is mobile: a sophisticated infrastructure is required to locate it. ANSAware [APM 93], demonstrates how such an infrastructure can be provided, although it is not CORBA compliant.

The arrangement of libraries and processes at the remote object will be similar to that of the client. Typically the remote object will be contained in a process linked with the IDL skeleton, the basic object adapter and some routines provided by the ORB core. A process may contain many objects and IDL skeletons (in ODP terminology it is a capsule [ODP]). This arrangement is shown in figure 7.4

Figure 7.4: Multiple Objects Contained in a process or capsule



The ORB interface provides a number of operations which can be applied to any object. Although they are provided by the ORB, the language binding (the definition of how the language is supported in CORBA) makes it look as though the operation is implemented by the object to which it is being applied. The operations provided by the ORB interface include:

- Operations to convert object references to strings and vice-versa;
- The “release()” and “duplicate()” operations for managing memory used by objects and object references (discussed in 7.2.5);
- The “get_interface()” operation needed for the Interface Repository discussed in 7.2.7;
- The “create_request()” operation used in conjunction with the Dynamic Invocation Interface discussed in 7.2.6;

A given environment may have more than one ORB. Thus [OMG 95a] includes facilities to allow objects to select an ORB at initialisation.

The next few sections discuss each of the other interfaces defined in the CORBA specification in more detail.

7.2.4 IDL Stubs and Skeletons

Both IDL stubs and skeletons are generated by a program called a “**Stub Compiler**” from the IDL definition of an interface. The OMG has defined the mapping of IDL to a number of languages (including C, C++ and Smalltalk). Thus, for a given language and for a given interface definition, the client-stub and skeleton-object interfaces are fixed (see figure 7.3).

The job of the stub and skeleton is to hide the details of the underlying ORB from the application programmer, making remote invocation look similar to local invocation.

For example the C++ code the client programmer has to write to invoke the “List” operation of “Account” is shown below (the CORBA_Environment variable “IT_X” is used to carry exception information, its use is discussed in section 7.2.5).

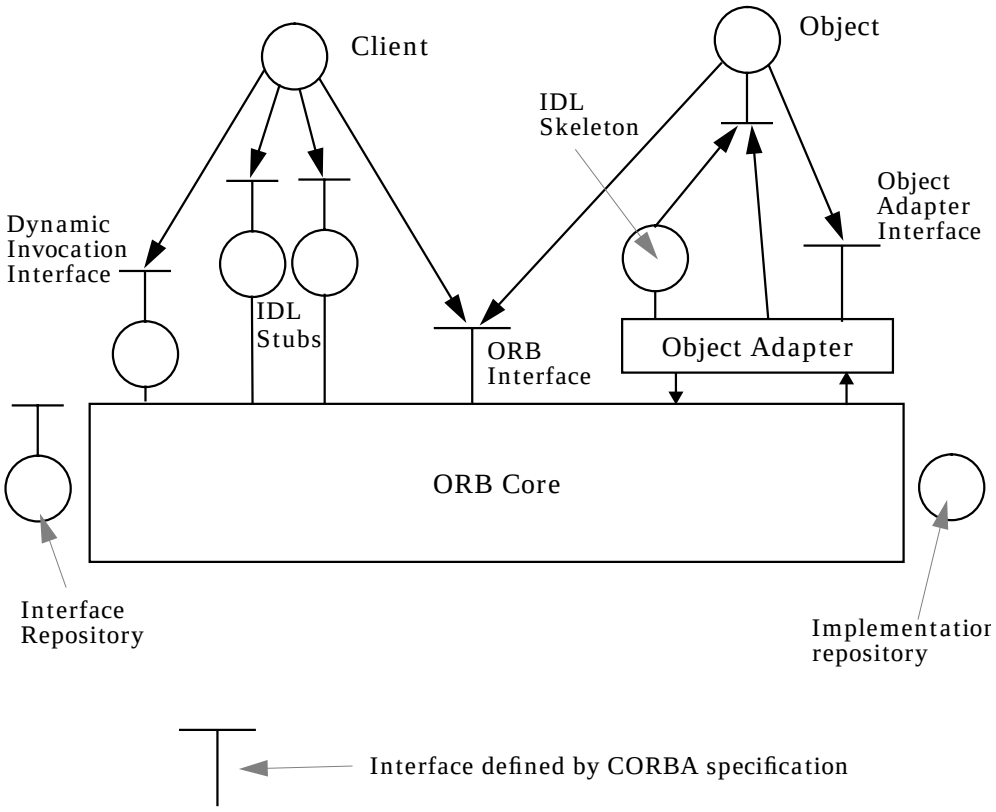
```
Acc->List(lresult, IT_X);
```

When the client invokes the “List” operation it is invoking the operation of that name provided by the IDL stub which typically resides in the same process as the client. The IDL stub drives the underlying ORB to invoke the remote object. There are many different ways in which to achieve this. In the following we describe one possible implementation. We assume the ORB is implemented using the TCP/IP protocol suite and specifically is using UDP to communicate. However, the reader should note that ORBs can be built using a variety of different protocols to communicate and the TCP/IP protocols may not always be the most appropriate protocols to use.

When the client invokes the IDL stub “List” operation, the stub obtains a buffer and writes into the buffer an encoding of the operation name and each of the parameters of the operation. This is known as marshalling; the inverse process of reading and decoding the contents of a buffer is known as unmarshalling. The operation name and possibly the object reference will be marshalled into the buffer also.

The stub will then hand the underlying ORB core the buffer together with the object reference. The routines invoked by client stub in the ORB core are

Figure 7.3: Basic Structure of an ORB



make the converse choice. The important point is that clients and objects should not observe a difference (modulo performance differences). This is because CORBA has specified the interfaces through which they interact with the ORB. Also, CORBA specifies the rules by which objects interact with each other through interfaces defined in IDL. Hence porting object implementations from one CORBA implementation to another should be trivial.

There are some things which may make porting object implementations non-trivial:

- CORBA does not define fully the interface used by the object adapter to activate objects (giving a full definition is difficult because exactly what constitutes activation will vary between different operating systems);
- CORBA allows there to be several alternative object adapters, but only defines the interface provided by one;
- The ORB interfaces are specified in IDL with an English language description of the semantics of each operation defined in an interface; different implementations may interpret the English specification differently.

Nevertheless porting between CORBA implementations will be significantly easier than if nothing was defined at all! Some of the ambiguities which may cause porting difficulties will be removed by conformance tests (see section 7.6.6)

The “SBank” interface illustrates two important points about CORBA. The first is that object references can be treated as first class entities: passed as operation parameters and returned as results. Thus clients can obtain references to and communicate with objects which were not known to them at compile time. The second point is that CORBA objects should not necessarily be thought of as large expensive entities in terms of the storage and processing resources they require. An efficient implementation on an operating system such as Unix™ would need to allow many CORBA objects to share the same process. Given such efficient support, the CORBA programmer can use distributed objects as an abstraction and encapsulation mechanism in exactly the same way as objects are used by C++ or Smalltalk programmers.

A language mapping defines how the CORBA IDL types map onto the type system of the target language. In addition the mapping also specifies how certain ORB interfaces must appear to programmers using the language. [OMG 93] specifies a mapping for C. At the time of writing several language mappings have been defined or are being defined including C++, Smalltalk, Ada and Cobol.

The full definition of IDL is given in [OMG 93] and also [OMG 95a].

7.2.3 The Object Request Broker

In this section we use the term object and client to distinguish between the entity receiving a request to execute an operation (the object) and the entity sending the request (the client). These are just roles. A client may itself support a number of operations; when these operations are invoked it will be acting in the role of an object.[5]

The most important function of an ORB is to enable a client to invoke an operation on a potentially remote object. Although this is a simple requirement, satisfying it is not so simple! In section 7.1 we described the aspects of distribution which an ORB must make transparent to the programmer making a remote invocation. To achieve this, the ORB must hide the underlying protocols and networks used to send the invocation and receive the results.

A client identifies the target object by means of an object reference. The ORB is responsible for locating the object, preparing it to receive the request (it may need to be activated) and passing the data needed for the request to the object. Once the object has executed the operation identified by the request, if there is a reply needed, the ORB is responsible for communicating the reply back to the client. An ORB consists of several logically distinct components (see figure 7.3). Everything (including the IDL Stubs and Skeletons) apart from the client and Object is regarded as part of the ORB.

The CORBA specification [OMG 93] defines the interfaces explicitly labelled in the diagram. There are several interfaces which are not defined by the specification: the interface between the ORB Core and IDL stubs; the interface between the object adapter and the IDL skeletons; the interface between the Object Adapter and ORB Core. By not defining these interfaces, the CORBA specification allows many different implementations. For example, one implementation may choose to provide more functionality in the IDL stubs and Object Adapter and less in the ORB Core. Another implementation may

5. Another name for the object role is “server”. [OMG 93] uses the term object and we have followed this convention.

Figure 7.2: IDL for SBank and Account

```
typedef unsigned long AccountNumber;
typedef unsigned long PersonalIdentificationNumber;

exception NoSuchAccount {};
exception InvalidPin {};
exception InsufficientFunds {};

interface Account
{
 struct AccountRecord {
 string owner;
 float balance;
 string lastaccess;
 };
 void Credit(in float Amount);
 void Debit(in float Amount) raises (InsufficientFunds);
 void List(out AccountRecord List_R1);
};

Interface SBank
{
 Account Access(in AccountNumber acct,
                in PersonalIdentificationNumber pin)
                raises(NoSuchAccount, InvalidPin);
};
```

“AccountRecord” can be used in the rest of the specification. The interface has three operations: “Credit”, “Debit” and “List”. “Credit” takes a single “in” parameter (the amount to be credited to the account) and returns no results: neither “out” parameters nor a return value. “Debit” is similar, but can return (or raise) the exception “InsufficientFunds” (as well as the standard system exceptions). Presumably this exception would be raised if the customer would exceed their overdraft limit by debiting the requested amount. The operation “List” takes no “in” parameters, but returns a single result of type “AccountRecord” as an “out” parameter. This result would be used to show the customer their current balance and provide some information (in the form of a string) about the last access to the account.

The interface “SBank” provides a way for customers to gain access to their accounts. Each bank account has associated with it both an account number and a Personal Identification Number or PIN. The single operation provided by the interface “SBank” takes a PIN and an account number returning a value of type “Account”. In other words it returns an object reference to the customer which can then be used to access the account.

There are two ways to implement the “Access” operation which returns a reference to an “Account” object. One assumes that “Account” objects already exist, so an implementation of “SBank” would store object references and hand them out in response to invocations of the “Access” operation. An alternative implementation would be to store the state associated with an account, creating “Account” objects on the fly when the “Access” operation is invoked. Which of these alternatives is used will be invisible to the client program.

Since none of them are preceded by the tag “oneway” (indicating best-effort), all the operations are at-most-once if an exception is returned, or exactly-once if they return successfully.

When a client invokes an operation on an object, the object may not be immediately accessible to the ORB supporting the receiving object. For example, code for the operations (methods) and persistent state may have been stored in a file system. If this is the case the object is said to be deactivated. Before the operation can be executed the object must be activated. This might require the ORB to instantiate a process containing the object's state and methods.

An interface is a set of possible operations which a client can request of an object. Interfaces are specified in IDL. In [OMG 93] an object may support multiple interfaces. Given an object reference, a client may invoke any operation within any interface supported by the object.

It is probable that the intention of the original CORBA specification was for objects to have only a single interface. However, as the specification stands, it neither explicitly allows multiple interfaces per object nor explicitly prohibits them. Hence interpreting the specification as allowing multiple interfaces per object is consistent with that specification. This may change in the future (see section on 7.6.4 which also explores some of the advantages of having multiple interfaces per object and distinguishing interfaces explicitly).

Objects can be created and destroyed as the result of operations; the mechanisms for this are transparent to the client. The result is perceived by the client as an object reference which identifies the new object.

7.2.2 The Interface Definition Language

The Interface Definition Language or IDL is used to specify the components of the ORB as well as services which (application-level) objects make available to clients. IDL uses a C++ like syntax for defining the operations supported in an interface. It completely specifies the parameters, results and exceptions for each operation, but does not specify the semantics of that operation (apart from whether it is best-effort or at-most-once). This means that two implementations may satisfy the same IDL specification, but have completely different behaviours. A simple example of an IDL specification is given in figure 7.2. It defines two interfaces: one is a bank account, the other is the interface to a bank responsible for giving customers access to their accounts by giving them an interface to an "Account" object.[4]

The specification begins with a couple of type definitions and then defines three user exceptions. Exceptions are named errors which can be returned by an operation instead of its normal return value. These help the programmer to deal with runtime failures efficiently, something which is important in building robust systems. The CORBA specification defines a number of standard exceptions which any operation can return (e.g. "`INV_OBJREF`" for invalid object reference). An operation can return a user defined exception such as "NoSuchAccount" only if it is named in the operation's signature. We will see several examples of how application programmers can use exceptions in later sections.

Next in the IDL specification is the definition of the interface "Account". This begins by defining the structure "AccountRecord". The main difference between IDL structures and structures in C is that a definition of a structure also implicitly defines a type of the same name; hence, the type

4. The specification from which this example is taken also includes a management interface for creating and deleting customer accounts in the bank.

7.2 The Common Object Request Broker Architecture (CORBA)

This section summarises the CORBA 1.2 specification as defined in [OMG 93] which is the most up to date specification available at the time of writing. The latter document is 178 pages long, so this text necessarily omits certain details. Shortly, a new specification will be published — CORBA 2.0 [OMG 95a]. The text indicates those additions and amendments which, to the best of the author's knowledge, are likely to be made in CORBA 2.0. We proceed as follows:

- Section 7.2.1 defines what is meant by an object;
- Section 7.2.2 describes the Interface Definition Language (IDL) and gives some concrete examples;
- Section 7.2.3 describes the ORB;
- Section 7.2.4 describes IDL stubs and skeletons;
- Section 7.2.5 gives an example CORBA application;
- Section 7.2.6 describes the Dynamic Invocation Interface (DII);
- Section 7.2.7 discusses the Interface Repository;
- Section 7.2.8 describes the Basic Object Adapter or BOA.

The code examples included in this section were developed using a CORBA compliant platform called Orbix™.[3]

7.2.1 The Object Model

An object is an entity which encapsulates state and provides one or more operations acting on that state. These operations can be requested by clients. An operation is defined by a signature written in IDL (see section 7.2.2) which includes:

- A specification of the parameters required;
- A specification of the results;
- A specification of the exceptions which may be raised when the operation is invoked;
- An execution specification of the semantics of the operation (not to be confused with a full semantic description).

The execution specification specifies one of two different execution semantics: at-most-once and best-effort. The former requires the operation was performed exactly once, if it returns successfully; the operation is performed at most once if it returns an exception. Best-effort operations are request-only: they do not return any results, so the requesting client receives no reply and does not synchronize with completion of the operation.

To invoke an operation, clients must identify the receiving object. Object references are used for this. An object reference always identifies the same object. However, the same object may be identified by more than one object reference (for example, each client may have a different object reference).

When an object receives an invocation from a client, it may invoke operations on other objects before returning a result to the client. The client has no way of telling if other objects were invoked.

3. Orbix is a Registered Trademark of Iona Technologies Ltd.

Object Services are basic services which other objects might find useful. Examples include:

- Naming services which, if given the name of a service, will supply a reference which can be used to invoke that service;
- Lifecycle services which can be used to control objects including creating, deleting, moving or modifying them;
- Persistence or storage services which can be used to store state.

Object services are specified using CORBA's IDL. More details and some specific examples of object services are given in section 7.4 of this chapter.

Common Facilities also provide services which other objects might find useful. Common Facilities can be thought of as an "application tool kit", whereas Common Services are an "infrastructure tool kit". Object Services will be available for all ORBs.[2] In contrast, Common Facilities are optional, but if they are provided they must conform to OMG specifications. Examples of Common Facilities include:

- User interface facilities;
- Compound document facilities;
- Services specialized for a particular application domain (e.g. financial).

Common facilities are specified in IDL; more details about these services are given in section 7.5 of this chapter.

The services provided by Application Objects are specified using IDL; however, these services are not standardized by the OMG. Examples of application objects include: email, spreadsheets, CASE tools, data querying tools, CAD tools. In OMA an application consists of a collection of interworking objects. Usually the collection will consist of one or more application objects and a number of Object Services and Common Facilities. Although Object Services and Common Facilities may provide the functionality required by an object, it is not mandatory for that object to use this functionality: it may choose to implement the functionality itself or use a non standard service provided by an Application Object.

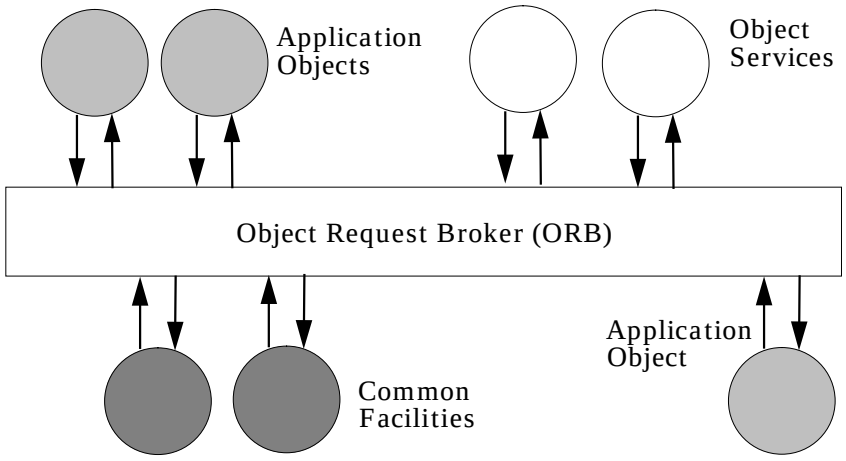
The next few sections of this chapter describes the various components of OMA in more detail:

- Section 7.2 describes the Common Object Request Broker Architecture;
- Section 7.3 discusses the CORBA approach to interoperability;
- Section 7.4 describes the Common Object Services;
- Section 7.5 describes the Common Facilities;
- Section 7.6 looks at a number of issues related to CORBA, such as what applications it is suitable for;
- Section 7.7 summarises the main points of this chapter.

We begin with CORBA.

2. [OMG 92] actually says Object services will be available on all platforms whereas common facilities are optional. Unfortunately it does not define what a platform is; we assume it means ORB.

Figure 7.1: Main components of OMA



- Access path: the route taken by messages exchanged with the other objects;
- Relocation: movement of the other objects from one machine to another;
- Representation: the format of data associated with the other objects;
- Communication mechanism: what inter process communication mechanism or protocol is used;
- Invocation mechanism: how the other objects' methods are executed (e.g. details of processes, threads, dynamically linked libraries);
- Storage mechanisms: the details of any storage that may or may not be used by the other objects;
- Machine type: any differences in machine type;
- Operating System: any differences in operating system;
- Programming language: in what programming language the other objects are implemented;
- Security Mechanisms: the specific mechanisms used to control access to the other objects.[1]

Any changes in the above for a particular object should not force the recompilation (or relinking, reloading etc.) of other objects. This allows changes to be made dynamically to the implementation of an object without affecting other objects, either its clients or its servers. Thus it is easy to introduce new implementations and replace existing services.

There are many possible different interfaces which could be provided by an ORB satisfying these requirements. The standard interface for the ORB component of the OMA is called Common Object Request Broker Architecture or CORBA. This is described in section 7.2 of this chapter.

The most important feature of CORBA is its Interface Definition Language or IDL. This language is used by the other components of the OMA to specify the services they offer to each other using the ORB.

1. It is possible to enforce simple policies transparently such as the partitioning of users into secure and insecure groups. However, certain application domains such as electronic commerce will require security to be handled explicitly by the application.

7 CORBA — An Industrial Approach to Open Distributed Computing

This chapter looks at an industry standard for open distributed processing being developed by the Object Management Group (OMG). The OMG is a consortium operated as a not-for-profit company based in the USA. The objective of the OMG is to create a standard for interoperability between independently developed applications across networks of computers. The OMG's member organisations include most of the major information technology vendor companies and many end-user companies. It works by adopting interface and protocol specifications within the context of a jointly agreed Object Management Architecture or OMA. These specifications are usually developed by a number of its members working in collaboration.

The OMG focuses on distributed objects as a vehicle for systems integration. The key benefit of building distributed systems with objects is encapsulation: data and state is accessible only through invocation of a set of defined operations rather than allowing direct access. It is much easier to cope with heterogeneity (different implementations of the same service), because differences in data representations are hidden. In turn, this makes application and system integration easier.

Objects also make system evolution easy: new services and implementations can be introduced which support the same operations as the services they replace. It does not matter that the implementation and internal state of the new services is different. Old or legacy information systems can be wrapped or encapsulated inside an object, so that in time they too may be replaced as the system evolves.

The Common Object Request Broker Architecture or CORBA is the central component of the OMA and there are already several commercial implementations available. To position CORBA properly, first we need to understand the framework within which it fits: The Object Management Architecture (OMA).

7.1 The Object Management Architecture (OMA)

Figure 7.1 shows the main components of the Object Management Architecture [OMG 92]. The Object Request Broker or ORB is the central component: it provides all the other components with the ability to communicate (make and receive requests and responses). It encapsulates the underlying platform (i.e. the operating system and network) and provides many of the transparencies discussed in chapter 1. Ideally, an ORB should make the following aspects of distribution transparent to an object which is communicating with other, potentially remote, objects:

- Location: whether or not the other objects are on the same machine;

CORBA — An Industrial Approach to Open Distributed Computing

Nigel Edwards

Abstract

Draft for inclusion in “Open Dsitributed Systems”

28th July 1995