CORBA EXPLAINED SIMPLY

A concise book for people who want a technical understanding of the concepts and terminology of CORBA without learning the low-level details

Ciaran McHale

www.CiaranMcHale.com

Availability and Copyright

Availability

You can get this book, free of charge, from www.CiaranMcHale.com. The download page on the web site, (www.CiaranMcHale.com/download), provides links to download the book in the following formats:

- A PDF file formatted for A5 paper, which is slightly larger than a paperback novel. The small page size, combined with embedded hypertext links, makes it suitable for on-screen reading.
- A "2-up" PDF file (without any hypertext links), in which two A5 pages are placed side by side. This version will save you paper if you want to print the book on A4 paper, which is the most common size paper in Europe. If you print this document on US Letter paper (which is slightly shorter and wider than A4 paper) then the *Print...* dialog box in Adobe Acrobat Reader has a *Auto-Rotate and Center* option that you can use make the document print better.
- An archive of HTML. You can get this as a Windows-friendly zip file or, if you prefer, as a UNIX-friendly compressed tar file.

Alternatively, if you want to browse this book online before deciding to download it then go to www.CiaranMcHale.com/corba-explained-simply.

Copyright

This version of the book was produced on February 27, 2007.

Trademarks

For my mother, Alice, and my wife, Bianca
Also for JS and TUF

Contents

Preface

Intended Audience

This book provides a detailed introduction to the concepts and terminology of CORBA. It is aimed at people with a technical background who want to gain a concrete understanding of the *concepts* of CORBA without learning all the low-level details. For example:

- A chief technology officer (CTO) might read this book in order to understand the concepts of CORBA and so decide if it is suitable for the needs of his or her organization.

- A project manager might read this book so he or she can engage developers in meaningful discussions about CORBA-related issues.

- If your organization outsources the development of a CORBA application to another company then reading this book will help you when discussing your applications' specifications and requirements with your supplier company.

- Developers who are starting to learn CORBA will appreciate the high-level technical discussion of the *concepts* of CORBA. A clear understanding of these concepts makes it much easier to understand the low-level APIs that are the focus of most programmer-oriented documentation. Developers who already know some of the basic capabilities of CORBA will find that this book gives them a concise overview of some of the "advanced" capabilities of CORBA. As such, this book can provide guidance to developers with some CORBA experience on what they can learn next to enhance their skills.

- System administrators who have to manage CORBA systems will find their work much easier if they read this book to gain an understanding of what CORBA does and how it does it.

One thing that is deliberately missing from this book is code examples. This is because this book does *not* provide a programming tutorial for CORBA developers. Readers interested in learning how to develop CORBA applications are advised to browse `www.amazon.com` and pick a book that has good customer reviews. Alternatively, look at Section 26.1 on page 255 for a list of some of the author's favorite CORBA books.

Although this book is not a tutorial on CORBA programming, it is a very good complement to such tutorial books. In particular, by concisely explaining the concepts of CORBA, this book provides readers with a firm foundation that they can build upon by, afterwards, reading a CORBA programming book.

How to Read this Book

There is no need to read this book from start to finish. Instead, the information in this book is arranged in chapters (most of which are quite short), and each chapter is either self-contained or has cross-references to other chapters if concepts in the chapter rely upon concepts discussed in other chapters. This makes it possible for readers to jump around the book, reading only those chapters that interest them. The only exception to this is that all readers should read Chapter 1, which explains the most fundamental concepts of CORBA.

About the Author

Ciaran McHale holds a Ph.D. and BA in Computer Science from Trinity College, Dublin, Ireland. For the past 11 years he has been working in IONA Technologies, where he is a principal consultant. Aside from consulting with customers, his job also involves the development and teaching of training courses. He lives in Reading, England with his wife, Bianca. You can contact the author through email at Ciaran@CiaranMcHale.com.

About the Contributing Author

Donal Arundel wrote the chapter on Security (Chapter 23) for this book. Donal is a principal engineer for IONA Technologies, where he is the technical lead for CORBA Security and has been developing distributed object technology security solutions for the past seven years. Previously he worked for ICL, where he developed a secure smart-card-based Electronic Money System. He holds a BSc. in Computer Applications from Dublin City University, Dublin, Ireland.

Disclaimer

Acknowledgments

First, thank you to Marco Abbate in Vodafone Italy for his support when I started to write this book.

Second, thank you to Donal Arundel for writing the chapter on security.

Third, thank you to the numerous people who have given me feedback on draft versions of this book: Adrian Trenaman, Andres Ortiz, Arne Koschel, Brian Kelly, Francis Byrne, Jan Schaefer, John McHugh, Klaus Hofmann zur Linden, Neil Kenealy, Niall Donnelly, Örjan Petersson, Patrick Donnelly, Paul Taylor, Raffaele Giugliano, Rebecca Bergersen and Richard Bonneau.

Fourth, thank you to others within IONA who have helped, in one way or another, with this book: Klaus Hofmann zur Linden and Enda Brennan for their encouragement; Sean Flavin for spiritual guidance; and Joe McCarthy, Fintan Bolton and John O'Sullivan for miscellaneous advice and help.

Finally, In 2001 I fell head over heels in love with a wonderful woman, and part-way through writing this book I proposed to her. Bianca, thank you for turning my life upside down and for saying “Yes” when I asked you to marry me.

Part I

Introduction to CORBA

Chapter 1

Core Concepts and Terminology

1.1 Object Management Group (OMG)

The Object Management Group (OMG) is a not-for-profit organization that promotes the use of object-oriented technologies. Among other things, it defines the CORBA and UML standards. The OMG web site (`www.omg.org`) provides all of its standards documents available free-of-charge in the form of downloadable PDF files. The OMG has a relatively small staff that focuses on administrative tasks, such as maintaining the OMG web site and organizing meetings of its members.

The work of defining standards is carried out by the members of the OMG, of which there are about 600. Any organization (or individual) that is interested in the work of the OMG can become a member. Member organizations typically include universities, software vendors and software users. Members can volunteer to take part in task forces that have the goal of defining new OMG standards or enhancing existing OMG standards. It is through this work that the OMG standards evolve in directions directed by the real-world concerns of its members.

1.2 CORBA

CORBA is an acronym for *Common ORB Architecture*. The phrase *common architecture* means a *technical standard*, so CORBA is simply a technical standard for something called an ORB.

ORB is an acronym for *Object Request Broker*, which is an object-oriented version of an older technology called *Remote Procedure Call* (RPC). An ORB or RPC is a mechanism for invoking operations on an object (or calling a procedure) in a different ("remote") process that may be running on the same, or a different, computer. At a programming level, these "remote" calls look similar to "local" calls.

Many people refer to CORBA as *middleware* or *integration software*. This is because CORBA is often used to get existing, stand-alone applications communicating with each other. A tag-line used by IONA Technologies, *Making Software Work Together*™, sums up the purpose of CORBA.

Of course, CORBA is not the only middleware technology in existence. Some other brand names of middleware include Java Remote Method Invocation (RMI), IBM MQ Series, Microsoft's COM and .NET, SOAP, and TIBCO Rendezvous. Scripting languages—such as UNIX shells, Perl, Python and Tcl—can also be classified as middleware because scripts are often used to connect programs together. A famous example of this is the pipe operator in UNIX shells, as illustrated in the example below:

```
ls -l | grep ^d
```

The pipe operator sends the output of the first command to the second command. Put simply, it helps two applications communicate with each other, which is what middleware is all about.

One of CORBA's strong points is that it is *distributed* middleware. In particular, it allows applications to talk to each other even if the applications are:

- On different computers, for example, across a network.
- On different operating systems. CORBA products are available for many operating systems, including Windows, UNIX, IBM mainframes and embedded systems.
- On different CPU types, for example, Intel, SPARC, PowerPC, big-endian or little-endian, and different word sizes, for example, 32-bit and 64-bit CPUs.
- Implemented with different programming languages, such as, C, C++, Java, Smalltalk, Ada, COBOL, PL/I, LISP, Python and IDLScript[1]

[1] IDLScript is a scripting language that was invented specifically for CORBA. The people who invented IDLScript felt that CORBA would be best served by having *one official* scripting language. Some other people felt that just as CORBA supported many "systems" languages (C, C++, Java, Ada, COBOL and so on), so too it would be good for CORBA to support several existing scripting languages (Perl, Python, Tcl, Visual Basic and so on) rather than inventing a new scripting language specifically for CORBA.

CORBA is also an *object-oriented*, distributed middleware. This means that a client does not make calls to a server process. Instead, a CORBA client makes calls to *objects* (which happen to live in a server process).

1.3 Client and Server

In some computer technologies, the terms *client* and *server* have a strict meaning and an application is either one or the other. CORBA is not so strict. In CORBA terminology, a *server* is a process that contains *objects*, and a *client* is a process that makes calls to objects. A CORBA application can be both a client and a server at the same time.

1.4 Interface Definition Language (IDL)

An IDL file defines the public *application programming interface* (API) that is exposed by objects in a server application. The *type* of a CORBA object is called an `interface`, which is similar in concept to a C++ `class` or a Java `interface`. IDL interfaces support multiple inheritance.

An example IDL file is shown in Figure 1.1. An IDL `interface` may contain operations and attributes. Many people mistakingly assume that an `attribute` is similar in concept to an instance variable in C++ (a *field* in Java). This is wrong. An `attribute` is simply syntactic sugar for a pair of get- and set-style operations. An `attribute` can be `readonly`, in which case it maps to just a get-style operation.

The parameters of an operation have a specified direction, which can be `in` (meaning that the parameter is passed from the client to the server), `out` (the parameter is passed from the server back to the client) or `inout` (the parameter is passed in both directions). Operations can also have a return value. An operation can *raise* (throw) an exception if something goes wrong. There are over 30 predefined exception types, called *system* exceptions, that all operations can throw, although in practice system exceptions are raised by the CORBA runtime system much more frequently than by application code. In addition to the pre-defined system exceptions, new exception types can be defined in an IDL file. These are called *user-defined* exceptions. A `raises` clause on the signature of an operation specifies the user-defined exceptions that it might throw.

Parameters to an operation (and the return value) can be one of the built-in types—for example, `string`, `boolean` or `long`—or a “user-defined” type that is declared in an IDL file. User-defined types can be any of the following:

```
module Finance {
  typedef sequence<string> StringSeq;
  struct AccountDetails {
    string     name;
    StringSeq  address;
    long       account_number;
    double     current_balance;
  };
  exception insufficientFunds { };
  interface Account {
    void deposit(in double amount);
    void withdraw(in double amount)
                        raises(insufficientFunds);
    readonly attribute AccountDetails details;
  };
};
```

Figure 1.1: Example IDL file

- A `struct`. This is similar to a C/C++ `struct` or a Java `class` that contains only public fields.
- A `sequence`. This is a collection type. It is like a one-dimensional array that can grow or shrink.
- An array. The dimensions of an IDL array are specified in the IDL file, so an array is of fixed size, that is, it cannot grow or shrink at runtime. Arrays are rarely used in IDL. The `sequence` type is more flexible and so is more commonly used.
- A `typedef`. This defines a new name for an existing type. For example, the statement below defines `age` that is represented as a `short`:

  ```
  typedef short age;
  ```

  By default, IDL sequences and arrays are *anonymous types*, that is, they do not have a name. A common, and very important, use of `typedef` is to associate a name with a sequence or array declaration. An example of this can be seen in the definition of `StringSeq` in Figure 1.1.
- A `union`. This type can hold one of several values at runtime, for example:

```
union Foo switch(short) {
  case 1: boolean  boolVal;
  case 2: long     longVal;
  case 3: string   stringVal;
};
```

An instance of `Foo` could hold a `boolean`, a `long` or a `string`. The case label (called a *discriminant*) indicates which value is currently active. Constructs similar to an IDL `union` can be found in many procedural languages. However, they are less widely used in object-oriented languages because polymorphism usually fulfills the same purpose in a more elegant manner.

- An `enum` is conceptually similar to a collection of constant integer declarations. For example:

  ```
  enum color { red, green, blue };
  enum city { Dublin, London, Paris, Rome };
  ```

  Internally, CORBA uses integer values to represent different `enum` values. The benefit of using `enum` declarations is that many programming languages have built-in support for them (or something similar) and can perform strong type checking so that programmers cannot, for example, add a `color` to a `city`.

- A `fixed` type holds *fixed-point* numeric values, whereas the `float` and `double` types hold *floating-point* numeric values. Floating-point arithmetic is suitable for many purposes, but may result in rounding errors after a few decimal places. In contrast, fixed-point numeric values may occupy more memory space than equivalent floating-point values, but they have the virtue of avoiding rounding errors. Use of fixed-point arithmetic tends to be restricted to niche application areas, such as financial calculations and digital signal processing (DSP). Even if an application uses fixed-point numbers, it is likely that the application will use fixed-point arithmetic as an implementation detail and will *not* expose the use of fixed-point numbers in its public IDL interface. For these reasons, fixed-point types are rarely declared in IDL files.

- A `valuetype`. This is discussed in Section 9.2 on page 96.

IDL types may be grouped into a `module`. This construct has a similar purpose to a `namespace` in C++ or a `package` in Java, that is, it prepends a prefix on to the names of types in order to prevent namespace pollution. The scoping operator in IDL is `"::"`. For example, `Finance::Account` is the fully-scoped name of the `Account` type defined in the `Finance` module.

1.4.1 The C++ Preprocessor

An IDL compiler uses a C++-compatible preprocessor to preprocess input IDL files. The preprocessor removes C++-style comments and also processes directives that may be in IDL source files. An example of some of these directives is illustrated in Figure 1.2.

```
#ifndef FOO_IDL
#define FOO_IDL
#include "another-file.idl"
#pragma prefix "acme.com"
// This is a one-line comment
/* This is a multi-line
   comment
*/
module Foo {
  ...
};
#endif
```

Figure 1.2: Example `Foo.idl` file

It is common practice to have one module per IDL file and to name the IDL after after the module that it contains. For example, a file called `Foo.idl` typically contains a module called `Foo`.

The `#include` directive instructs the preprocessor to include the contents of the specified file. This makes it feasible to spread IDL definitions over several files in a modular manner rather than having to put all the definitions required for a project in one monolithic file.

The `#ifndef...#define...#endif` construct shown in Figure 1.2 is typically used to protect against the possibility that an IDL file might be `#include`d multiple times.

A discussion of the `#pragma prefix` directive is deferred until Section 9.4 on page 102.

1.4.2 Common IDL Idioms

1.4.2.1 Factory Interfaces

Many object-oriented languages have a *constructor* that is used to create and initialize an object. However, a constructor creates the object *locally*, that is, within the address space of the process that calls the constructor. Because of this, a constructor cannot be used to create an object in a *different* process, and this is the reason why you cannot define a constructor for an IDL interface.

The way for a client process to create an object in a different (server) process is for the client to invoke an operation on an existing object in the server, and for that operation (in the server process) to create a new object. The term *factory* is typically used to refer to an object that can create other objects. The operation that is used to create an object is often called `create()`—or has `"create"` embedded in its name, for example, `create_account()`—but that is just a naming convention rather than a requirement. No additional syntax is required to define factory interfaces or create-style operations. Rather, these are defined using “normal” IDL syntax. An example of a factory interface is shown in Figure 1.3.

```
interface Foo {
  void destroy();
  ...
};
interface FooFactory {
  Foo create(...);
  ...
};
```

Figure 1.3: Example of a factory interface

Just as an IDL interface does not have a constructor, neither does it have a *destructor*. Sometimes, the decision about when to destroy an object is made solely within a server, without any input from client applications. However, if there is a need for clients to control the destruction of an object then this is typically achieved by defining an operation that, when invoked, destroys the object. This operation is usually called `destroy()`, but that is just a naming convention rather than a requirement.

1.4.2.2 Callback Interfaces

Callback procedures/objects are commonly used in GUI (graphical user interface) toolkits: an application developer registers a procedure/object with the GUI toolkit runtime and the runtime can "call back" to the procedure/object whenever something relevant occurs, such as the mouse button is pressed or a key on the keyboard is typed. Callback objects are occasionally used in CORBA applications. As far as the IDL compiler is concerned, a callback interface (such as `FooCallback`, defined in Figure 1.4) is just a normal IDL `interface`, so there is no special syntax required to define a callback interface.

```
interface FooCallBack {
  void notify_something_has_happened(...);
};
interface FooCallbackRegistry {
  void register_callback(in FooCallback cb_obj);
  void unregister_callback(in FooCallback cb_obj);
  ...
};
```

Figure 1.4: Example of a callback interface

1.4.2.3 Iterator Interfaces

Let us assume that an IDL interface has a `query()` operation that uses a `sequence` to return query results. If the number of items in the returned results could potentially be quite large then it is inadvisable to return *all* the results in one monolithic lump. There are several reasons for this:

- The entire collection of results might occupy several megabytes or even gigabytes of memory. Even though the server process might run on a machine with sufficient memory to hold this amount of data, perhaps the client is running on a machine with far less memory. Returning this amount of data to the client in one lump could cause the client to run out of memory. It would be better to give the query results to the client in several smaller chunks that will not exhaust the client's memory.

- In many client-server applications that involve query-style operations, the results of a query are displayed to an interactive user and the user picks the one in which he or she is interested. If, as is frequently the case, the user happens to find the desired item near the start of the list then it is a waste of both network bandwidth and memory to have transmitted *all* the results from the server to the client. To avoid this wastage, it would be better to give the query results to the client in several smaller chunks. If the user picks an item in, say, the first or second chunk of results then further results do not have to be transmitted from the server to the client.

```
struct Data { ... };
typedef sequence<Data> DataSeq;
interface DataIterator {
  DataSeq next_n_items(in unsigned long how_many);
  void    destroy();
};
interface SearchEngine {
  DataSeq query(
            in  string          search_condition,
            in  unsigned long   how_many,
            out DataSeq         results,
            out DataIterator    iter);
};
```

Figure 1.5: An Iterator interface

Figure 1.5 shows how an iterator interface is typically used.[2] A `query()` operation initially returns up to `how_many` items in the `results`. If this holds *all* the items then the `iter` parameter is set to a nil object reference. Otherwise, the `iter` parameter contains a reference to an `DataIterator` object that can be used to obtain more results, again `how_many` at a time. When the iterator has no more results to return, `next_n_items()` returns an empty sequence and the client can then `destroy()` the iterator.

1.4.3 Limitations of IDL

The complexity of data-types that can be defined in IDL is quite limited compared to the complexity of data-types that can be defined in a programming language. The limited flexibility of IDL data-types is due mainly to the lack of *pointers*. For example, an IDL `struct` cannot contain a pointer to another IDL type. This lack of pointers makes it impossible to build arbitrary graph

[2] *Iterator* is a term denoting an object that is used to "iterate over" (traverse) a collection of items.

structures in IDL. This limitation of IDL is deliberate and is due to a combination of several reasons:

- If IDL were to support pointers then it would make it difficult, or perhaps impossible, to map IDL into programming languages that do not support pointers.

- If IDL supported pointers then this would make it possible for programmers to pass arbitrarily complex types, such as cyclic graphs, as parameters to remote calls. This flexibility would be used rarely by programmers, so supporting it would greatly increase the complexity of the marshaling engine in CORBA products for little benefit to users.

- IDL types are intended to be used to *specify* a public API rather than *implement* the API. Public APIs normally pass relatively simple datatypes as parameters so the limitations of IDL are not usually a problem in practice. Of course, it is still possible for a server to use pointers within its private implementation.

It should be noted that the relatively recent addition of *objects by value* (OBV) to IDL has finally provided IDL with some functionality similar to what C++ pointers provide. However, as I discuss in Section 9.2 on page 96, OBV has been a controversial addition to IDL.

Perhaps the most commonly-perceived limitation of IDL is that there is no inheritance of exceptions, that is, one exception type cannot be defined as a subtype of another exception type. Although this limitation is never a showstopper problem in projects, it certainly provides an irritation for developers. This is because an IDL operation may wish to report, say, 10 different types of exception, and it may be natural to arrange these into an inheritance hierarchy. Because IDL does not allow inheritance of exceptions, the designer is typically forced to either list 10 separate exceptions in the `raises` clause of the operation or to define one "generic" exception that uses, say, an `error_code` field to specify which category of error occurred. Both of these approaches can be awkward for client-side developers to handle. With the first approach, they may have 10 different `catch` clauses in a `try-catch` block surrounding an operation call. With the second approach, there will be just one `catch` clause but this will need to use a `switch` statement or a cascading `if-then-else` to determine the exact cause of failure.

1.4.4 Mapping IDL to a Programming Language

As Section 1.4 on page 5 mentioned, IDL is used to define the public API that is exposed by objects in a server application. IDL defines this API in a way that is *independent* of any particular programming language. However, for CORBA to be useful, there must be a mapping from IDL to a particular programming language. For example, the IDL-to-C++ mapping allows people to develop CORBA applications in C++ and the IDL-to-Java mapping allows people to develop CORBA applications in Java.

The CORBA standard currently defines mappings from IDL to the following programming languages: C, C++, Java, Ada, Smalltalk, COBOL, PL/I, LISP, Python and IDLScript. These officially-endorsed language mappings provide source-code portability of applications across different CORBA products (portability is discussed in Chapter 25). There are *unofficial*—or, if you prefer, *proprietary*—mappings for a few other languages, such as Eiffel, Tcl and Perl. Obviously, you could develop a CORBA application with an unofficial language mapping, but you would not have any guarantees of source-code portability to other CORBA vendor products.

1.4.5 IDL Compilers

An IDL compiler translates IDL definitions (for example, `struct`, `union`, `sequence` and so on) into similar definitions in a programming language, such as C++, Java, Ada or Cobol. In addition, for each IDL `interface`, the IDL compiler generates both *stub code*—also called *proxy types* (Section 10.3 on page 110)—and *skeleton code*. These terms are often confusing to people for whom English is not their native language, so I explain them below:

- The word *stub* has several meanings. A dictionary definition of *stub* is "the short end remaining after something bigger has been used up, for example, a pencil stub or a cigarette stub". In traditional (non-distributed) programming, a *stub procedure* is a dummy implementation of a procedure that is used to prevent "undefined label" errors at link time. In a distributed middleware system like CORBA, remote calls are implemented by the client making a local call upon a *stub* procedure/object. The stub uses an inter-process communication mechanism (such as TCP/IP sockets) to transmit the request to a server process and receive back the reply.

- The term *proxy* is often used instead of *stub*. A dictionary definition of *proxy* is "a person authorized to act for another". For example, if you would like to vote on an issue but are unable to attend the meeting where the vote will be take place then you might instruct somebody else to vote on your behalf. If you do this then you are "voting by proxy". The term *proxy* is very appropriate in CORBA (and other object-oriented middleware systems). A CORBA proxy is simply a client-side object that acts on behalf of the "real" object in a server process. When the client application invokes an operation on a proxy, the proxy uses an inter-process communication mechanism to transmit the request to the "real" object in a server process; then the proxy waits to receive the reply and passes back this reply to the application-level code in the client.

- The term *skeleton code* refers to the server-side code for reading incoming requests and dispatching them to application-level objects. The term *skeleton* may seem like a strange choice. However, use of the word *skeleton* is not limited to discussions about bones; more generally, it means a "supporting infrastructure". *Skeleton code* is so called because it provides supporting infrastructure that is required to implement server applications.

A CORBA product must provide an IDL compiler, but the CORBA specification does not state what is the *name* of the compiler or what command-line options it accepts. These details vary from one CORBA product to another.

1.5 Interoperable Object Reference (IOR)

An *object reference* is the "contact details" that a client application uses to communicate with a CORBA object. Some people refer to an object reference as an *interoperable object reference* (IOR) or *proxy*. The *interoperable* in *interoperable object reference* comes about because an IOR works (or interoperates) across different implementations of CORBA. This means that an IOR for an object in, say, an Orbix server can be used by a client that is implemented with a different CORBA product, such as Orbacus, Visibroker, TAO, omniORB or JacORB. An in-depth discussion of object references is provided in Chapter 10.

1.6 CORBA Services

Many programming languages are equipped with a standardized library of functions and/or classes that complement the core language. These standardized libraries usually provide collection data-types (for example, linked lists, sets, hash tables and so on), file input-output and other functionality that is useful for the development of a wide variety of applications. If you asked a developer to write an application in, say, Java, C or C++ but *without* making use of that language's standard library then the developer would find it very difficult.

A similar situation exists for CORBA. The core part of CORBA (an object-oriented RPC mechanism built with IDL and common on-the-wire protocols) is of limited use by itself—in the same way that a programming language stripped of its standardized library is of limited use. What greatly enhances the power of CORBA is a standardized collection of services—called *CORBA Services*—that provide functionality useful for the development of a wide variety of distributed applications. The CORBA Services have APIs that are defined in IDL. In effect, you can think of the CORBA Services as being like a standardized class library. However, one point to note is that most CORBA Services are provided as prebuilt server applications rather than as libraries that are linked into your own application. Because of this, the CORBA Services are really a *distributed*, standardized class library.

Some of the commonly-used CORBA Services are discussed in other chapters of this book:

- The Naming Service (Chapter 4) and Trading Service (Chapter 20) allow a server application to advertise its objects, thereby making it easy for client applications to find the objects.

- Most CORBA applications use *synchronous*, *one-to-one* communication. However, some applications require *many-to-many*, *asynchronous* communication, or what many people call *publish and subscribe* communication. Various CORBA Services (Chapter 22) have been defined to support this type of communication.

- Many developers are familiar with the concept of database transactions. In a distributed system, it is sometimes desirable for a transaction to span *several* databases so that when a transaction is committed, it is guaranteed that either *all* the databases are updated or *none* are updated. The *Object Transaction Service* (OTS, discussed in Chapter 21) provides this capability.

Chapter 2

Benefits of CORBA

Section 1.2 on page 3 mentioned that CORBA is a type of middleware, but that there are other types of middleware too. This naturally raises the question of why you might wish to use CORBA instead of a different middleware technology. The reason, as I discuss in this chapter, is that CORBA offers numerous important benefits. You may find *some* of these benefits in other middleware technologies, but you will be hard pressed to find another middleware technology that offers *all* of these benefits.

2.1 Maturity

The original version of the CORBA standard was defined in 1991. This first version of the specification was deliberately limited in scope. The OMG's philosophy was to define a small standard, let implementors gain experience and then slowly expand the standard to incorporate more and more capabilities. This "slow but sure" approach has been remarkably successful. In particular, there have been few backwards-incompatible changes to the CORBA specification. Instead, new versions of the specification have tended to add new functionality rather than modify existing functionality. Today, CORBA is extremely feature-rich, supporting numerous programming languages, operating systems, and a diverse range of capabilities—such as transactions, security, Naming and Trading services, messaging and publish-subscribe services—that are essential for many enterprise-level applications. Many newer middleware technologies claim to be superior to CORBA but actually have to do a lot of "catching up" just to match some of the capabilities that CORBA has had for a long time.

2.2 Open standard

CORBA is an open standard rather than a proprietary technology. This is important for a variety of reasons.

First, users can choose an implementation from a variety of CORBA vendors (or choose one of the freeware implementations). You might think that switching from one CORBA product to another would involve a lot of work. However, the amount of work involved is likely to be much less than you might think, particularly if you follow the practical advice in Chapter 25 about how to increase the portability of CORBA-based applications. In contrast, if you use a proprietary middleware system then switching to another proprietary middleware vendor is much more challenging.

Second, the competition between different CORBA vendors helps to keep software prices down.

Finally, many proprietary middleware technologies are designed with the assumption that developers will build *all* their applications using that particular middleware technology, and so they provide only limited support for integration with other technologies. In contrast, CORBA was designed with the goal of making it easy to integrate with other technologies. Indeed, the CORBA specification explicitly tackles integrations with TMN, SOAP, Microsoft's (D)COM and DCE (a middleware standard that was popular before CORBA). Furthermore, many parts of J2EE borrow heavily from concepts in CORBA, which makes it relatively easy to integrate J2EE and CORBA. Some vendors sell gateways between CORBA and J2EE that make such integration even easier. Several CORBA vendors sell COM-to-CORBA and/or .NET-to-CORBA gateways. This provides a very pragmatic solution to organizations that wish to write GUI applications in, say, Visual Basic on Windows that act as clients to server applications on a different type of computer, such as UNIX or a mainframe. The Visual Basic GUI can be written as a COM/.NET client that thinks it is talking to a COM/.NET server, but in fact communicates with a gateway that forwards on requests to a CORBA server.

2.3 Wide platform support

CORBA implementations are available for a wide variety of computers, including IBM OS/390 and Fujitsu GlobalServer mainframes, numerous variants of UNIX (including Linux), Windows, AS/400, Open VMS, Apple's OS X and several embedded operating systems. There are very few other middleware technologies that are available on such a wide range of computers.

2.4 Wide language support

CORBA defines standardized language mappings for a wide variety of programming languages, such as C, C++, Java, Smalltalk, Ada, COBOL, PL/I, LISP, Python and IDLScript. Some small organizations might use a single programming language for all their projects, but as an organization increases in size, it becomes increasingly likely that the organization will make use of several programming languages. Likewise, the older an organization is, the higher the likelihood becomes that some of its “legacy” (older) applications are implemented in one programming language and newer applications are implemented in a different programming language. For these organizational reasons, it is important for a middleware system to support many programming languages; unfortunately, not all middleware systems do so. One extreme case of this is J2EE, which supports only Java. Another extreme case is the SOAP middleware standard. SOAP applications can be built with a variety of programming languages but, at the time of writing, the SOAP standard defines only one language mapping (for Java). There may be several vendors who support, say, C++ development of SOAP applications, but each of those vendors provides their own proprietary C++ APIs. This means that there is no source-code portability of non-Java SOAP applications across different vendor products.

2.5 Efficiency

The on-the-wire protocol infrastructure of CORBA (discussed in Chapter 11) ensures that messages between clients and servers are transmitted in a compact representation. Also, most CORBA implementations *marshal* data (that is, convert data from programming-language types into a binary buffer that can be transmitted) efficiently. Many other middleware technologies also use a similarly compact format for transmitting data and have efficient marshaling infrastructure. However, there are some notable exceptions, as I now discuss.

SOAP uses XML to represent data that is to be transmitted. The verbosity of XML results in SOAP using *much more* network bandwidth than CORBA.[1] SOAP-based applications also incur considerable CPU overhead involved in

[1] The relative verbosity of SOAP messages compared to CORBA messages depends on what kind of data is transmitted. Because there is no “universal” data that is representative of all applications, it is impossible to give precise figures. However, many people would agree with the claim that some SOAP messages can require about 5 or 10 times more bandwidth than equivalent CORBA messages.

formatting programming-language types into XML format and later parsing the XML to extract the embedded programming-languages types.

Some other middleware technologies, such as IBM MQ Series, transmit only binary data, which is efficient. However, this requires that developers write the marshaling code that copies programming-language types into the binary buffers prior to transmission, and the unmarshaling code to extract the programming-language types from a binary buffer. In contrast, a CORBA IDL compiler generates the marshaling and unmarshaling code, so that developers do not need to write (and maintain) such low-level code.

2.6 Scalability

The flexible, server-side infrastructure of CORBA (Chapter 5) makes it feasible to develop servers that can scale from handling a small number of objects up to handling a virtually unlimited number of objects. Obviously, scalability varies from one CORBA implementation to another but, time and time again, real-world projects have demonstrated that a CORBA server can scale to handle not just a huge amount of server-side data, but also high communication loads from thousands of client applications. Most CORBA vendors will likely know of customers who have tried a different middleware technology, found that it could not scale sufficiently well and then switched to CORBA.

2.7 CORBA Success Stories

With such an impressive list of benefits as those discussed in this chapter, it is little wonder that CORBA is being used successfully in many industries, including aerospace, consulting, education, e-commerce, finance, government, health-care, human resources, insurance, ISVs, manufacturing, military, petrochemical, publishing, real estate, research, retail, telecommunications, and utilities.

CORBA is used in everything from billing systems and multi-media news delivery to airport runway illumination, aircraft radio control and the Hubble space telescope. Most of the world's telephone systems, as well as the truly mission-critical systems operated by the worlds biggest banks, are built on CORBA.

A discussion about real-world projects that have benefitted from the use of CORBA is outside the scope of this book. However, many CORBA success stories are available on various web sites. For example, you can find over 300 CORBA success stories on `www.corba.org`. The web sites of some CORBA vendors also contain more detailed success stories.

Part II

Application Development

Chapter 3

Development of CORBA Applications

This chapter uses pseudocode to give a brief overview of the development of a CORBA client-server application. The pseudocode is is somewhat similar to C++ or Java, but the basic principles illustrated apply to CORBA development with other languages. Pseudocode is used in order to focus on the principles rather than getting sidetracked with the details of a particular language mapping.

3.1 Development of a Traditional Application

Figure 3.1 shows the structure of a traditional (that is, non-distributed), object-oriented application. One of more types (such as `Account`) are defined in their own source code files. Then the mainline of the program (`main.cpp`) creates one or more objects and invokes operations upon them.

3.2 Development of a CORBA Application

I now discuss what is required to write a distributed client-server application (using CORBA) that has similar functionality to the traditional application of Figure 3.1.

```
// File: Account.h
class Account {
    void deposit(...) { ... }
    ... // instance variables
};
// File: main.cpp
main(...)
{
    Account obj = new Account(...);
    obj.deposit(...);
}
```

Figure 3.1: Structure of a Traditional Application

3.2.1 IDL Files and Generated Code

The first step in developing a CORBA application is to use IDL to define the public APIs of object types. This is shown in Figure 3.2. Notice that the IDL definition of `Account` is broadly similar to, say, a C++ definition. There are some minor syntactic differences, such as the `class` keyword is replaced with the `interface` keyword. However, a more important difference is that an IDL file just *declares* the public API—it does *not* contain any implementation details, such as the bodies of operations or instance variables.

```
// File: Account.idl
interface Account {
    void deposit(...);
};
```

Figure 3.2: Account.idl

Once the IDL file has been written, the developer runs it through an IDL compiler, for example:

```
idl Account.idl
```

Note that CORBA has not standardized on the names of IDL compilers or their command-line options, so the exact command used varies from one CORBA product to another. If you use a C++ CORBA product then the IDL compiler generates files containing C++ data types that correspond to the types defined in the input IDL file. Likewise, if you use, say, a Java or Cobol CORBA product then the IDL compiler generates files containing Java or Cobol data types. Among the data types generated are a *proxy class* called `Account` and a *skeleton-code* class (Section 1.4.5 on page 13) called `POA_Account`.[1] The pseudocode contents of these classes are shown in Figure 3.3.

```
// Generated code
class Account {
    void deposit(...)
    {
        marshal request details into a binary buffer
        Send request buffer message to server
        Wait to receive reply from server
        if (reply buffer contains an exception) {
            unmarshal exception and throw it
        } else {
            unmarshal "out" parameters from reply buffer
        }
    }
};
class POA_Account {
    abstract void deposit(...);
    void dispatch(...)
    {
        unmarshal "in" parameters from request buffer
        try {
            deposit(...);
            marshal "out" parameters into reply buffer
        } catch(...) {
            marshal exception into reply buffer
        }
        Send reply buffer to client
    }
};
```

Figure 3.3: Code generated by IDL compiler

A client makes a remote invocation by invoking upon a (local) proxy object. The operations on the proxy object *marshal* (Section 11.2 on page 113) the details of the invocation—the *object key* (Section 5.6.1 on page 51) that

[1] The name of the skeleton-code class varies from one language mapping to another. For example, for an interface called `Account`, the C++ class is `POA_Account` while the corresponding class in Java is called `AccountPOA`.

uniquely identifies the target object in the server, the name of the operation being invoked and `in`/`inout` parameters—into a binary buffer and the contents of this buffer are transmitted to the server process that contains the target object. Then the proxy waits to receive back a reply message from the server and unmarshals the `out`/`inout` parameters and return value, if any, from the reply buffer. Alternatively, if the reply buffer contains an exception then the proxy unmarshals this and throws it.

The generated skeleton class (Figure 3.3) contains an abstract operation (a *pure virtual member function* in C++ terminology) for each IDL operation. This operation is not implemented in the skeleton class, but rather in a sub-class that is written by a developer. The skeleton class also contains some dispatch logic that unmarshals an incoming request, calls the appropriate operation—`deposit()` in our example—with the unmarshaled `in`/`inout` parameters. When this operation returns, the skeleton class then marshals the `out`/`inout` parameters and return value, if any, (or an exception) into a reply buffer and then transmits this back to the client application.

Developers do not need to know the low-level details of *how* proxies or skeleton classes work—only that they are part of the infrastructure that is used to delegate requests from a client application across a network to the "real" object in a server process.

3.2.2 Servant Classes

CORBA uses the terminology *servant* to mean an object in the host programming language (for example, C++, Java or Cobol) that implements the functionality of a CORBA object. In CORBA, a servant is *not* a CORBA object, but a servant does *represent* a CORBA object. A detailed discussion of this subtle distinction between a servant and a CORBA object is deferred until Chapter 5.

```
// File: AccountImpl.h
class AccountImpl inherits POA_Account
{
    void deposit(...) { ... }
    ... // instance variables
};
```

Figure 3.4: Servant class

The servant class is *not* generated by the IDL compiler; instead, it is handwritten by developers. Developers can use whatever name they want for this class, but a common naming scheme is for the servant class to be composed of the name of the IDL interface combined with a suffix, such as `Impl`. For example, `AccountImpl` might be the name of the servant class for the `Account` interface. Pseudocode for this servant class is shown in Figure 3.4. The servant class inherits from the generated skeleton class and must provide an implementation for all the IDL operations, such as `deposit()` in our example.[2] The servant class may contain constructors, instance variables and extra (non-IDL) operations to support the implementation of the IDL operations.

3.2.3 Server Mainline

Pseudocode for a server mainline is shown in Figure 3.5. The most important pieces of code are indicated in **bold**.

The `ORB_init()` function creates a new `ORB` object, which represents the CORBA runtime system.[3] This function is passed command-line arguments (in C/C++ these are held in `argc` and `argv`). `ORB_init()` inspects pairs of command-line arguments of the form `-ORB<name> <value>` and interprets these name-value pairs as configuration values for the newly created `ORB` object. The recognized name-value pairs vary from one CORBA product to another, but they are typically used to specify information such as the diagnostic level for the CORBA runtime system, the port on which a server process should listen for incoming connections, or the name of a configuration file that contains additional name-value pairs of configuration information.

When a CORBA program is terminating, it must call `orb.destroy()`. This operation ensures that the CORBA runtime system is gracefully destructed before the program terminates.

If anything goes wrong when calling a CORBA API then an exception is thrown. For this reason, a `try-catch` clause surrounds most of the code in the `main()` function. This is to ensure that, even if an exception is thrown, the application can call `orb.destroy()`.

Although Figure 3.5 shows just one servant being created, CORBA allows a server process to create an arbitrary number of servants for an arbitrary number of IDL interfaces. When a server implements several IDL interfaces, the server developer might want different servants to have different *qualities of service* (QoS). For example, if some servants are implemented in a thread-safe

[2] Although this inheritance-based approach to implementing servants is common, it is not the only mechanism available. CORBA also supports a delegation-based approach to implementing servant classes, but a discussion of that is deferred until Section 3.4.3 on page 34.

[3] This function is called `ORB_init()` in most language mappings, but has the slightly different name of `ORB.init()` in Java. The `ORB_init()` name is used in the pseudocode of this book.

```
// File: server_main.cpp
int main(int argc, char* argv[])
{
    exit_status = 0;
    orb = null;
    try {
        orb = CORBA::ORB_init(argc, argv);
        ... // create POAs to contain servants
        sv = new AccountImpl(...);
        ... // activate (insert) sv into a POA
        exportObjRef(..., sv._this(), ...);
        ... // activate POA managers
        orb.run();
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
    // Terminate gracefully
    try {
        if (orb != null) { orb.destroy(); }
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
    return exit_status;
};
```

Figure 3.5: Pseudocode of server mainline

way then the CORBA runtime system in the server should allow concurrent dispatching of incoming requests to those servants. Conversely, if some other servants are implemented in a way that is *not* thread-safe then the CORBA runtime system should ensure that the dispatching of incoming requests to those servants is serialized. The way that CORBA allows one QoS to be associated with some servants and a different QoS to be associated with other servants is by letting the server create several POAs. POAs are discussed in details in Chapter 5, but, in essence, a POA is a *collection* of servants. The QoS of a POA is specified when the POA is created. When a servant is *activated* (inserted) into a POA then the servant takes on the same QoS as its containing POA.

After calling `ORB_init()`, the server typically creates one or more POAs to contain servants. Then servants can be created and activated (inserted) into these POAs. It is common for a server to initially create just one or two *factories* (Section 1.4.2.1 on page 9) that can then be used to create more servants later.

Section 3.2.2 briefly mentioned that a servant *represents* a CORBA object. The `_this()` operation (which is defined in the generated skeleton class from which servants inherit) can be invoked on a servant to obtain an object reference for its corresponding CORBA object. A server program typically advertises one or more of its objects by exporting their object references to, say, a file (Section 3.4.2 on page 34), the Naming Service (Chapter 4) or the Trading Service (Chapter 20). The `exportObjRef()` function in Figure 3.5 is not a CORBA API; rather it is just a pseudocode placeholder to denote the exporting of an object reference by some means.

When the server's initialization is complete, it activates its POA managers (a discussion of which is deferred until Section 5.7 on page 56) and then calls `orb.run()` to enter an event loop. In the event loop, the CORBA runtime system accepts connections from clients, reads incoming requests and dispatches them to the skeletons associated with servants. The `orb.run()` API does not return until `orb.shutdown()` is called. The `orb.shutdown()` operation is typically called from a signal handler or from the body of an IDL operation with a name like `shutdown()` or `kill_server()`.

3.2.4 Client Mainline

Pseudocode for a client mainline is shown in Figure 3.6. The most important pieces of code are indicated in **bold**.

Just as in a server, a client calls `ORB_init()` to initialize the CORBA runtime system, and later calls `orb.destroy()` to ensure that the CORBA runtime system is gracefully destructed before the program terminates.

The `importObjRef()` function in Figure 3.6 is not a CORBA API; rather it is just a pseudocode placeholder to denote the importing of an object reference by some means, such as from a file (Section 3.4.2 on page 34), the Naming Service (Chapter 4) or the Trading Service (Chapter 20).

Once the client has a reference to an object in the server, the client can invoke operations upon it.

3.3 Critique of CORBA Application Development

The pseudocode of a traditional application, shown in Figure 3.1 is much more concise than the pseudocode of a corresponding CORBA application, shown

```
// File: client_main.cpp
int main(int argc, char* argv[])
{
    exit_status = 0;
    orb = null;
    try {
        orb = CORBA::ORB_init(argc, argv);
        obj = importObjRef(...);
        obj.deposit();
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
    // Terminate gracefully
    try {
        if (orb != null) { orb.destroy(); }
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
    return exit_status;
};
```

Figure 3.6: Pseudocode of client mainline

in Figures 3.2, 3.4, 3.5 and 3.6. In fact, there are just 9 lines of pseudocode for the traditional application, compared to 52 lines of pseudocode for the corresponding CORBA application, which is almost a 5-fold increase in the number of lines of pseudocode. From such a comparison, it is easy to conclude that "development of CORBA is 5 times more complex than development of traditional applications". However, such a conclusion is incorrect. This is because the additional steps required in a CORBA application—such as calling `ORB_init()` and `orb.destroy()`, creating POAs, importing/exporting object references and so on—are the same regardless of whether you write a small CORBA application or a much larger one. When you write a real CORBA application (as opposed to just pseudocode), the amount of CORBA infrastructure code required is usually dwarfed by the amount of "business logic" code.

3.4 Miscellaneous Notes

3.4.1 `resolve_initial_references()`

As discussed in Section 3.2.3, the `ORB_init()` function creates a new `ORB` object, which represents the CORBA runtime system. One operation defined on the `ORB` type is `resolve_initial_references()`. Some people find the name of this operation to be unintuitive, but the name makes a lot of sense if you know the following two pieces of information:

1. In CORBA terminology, *resolve* means *lookup* or *find*, so this operation is used to "lookup" or "find" something.

2. This operation is slightly misnamed. It should actually have been called `resolve_initial_reference()` (that is, "reference" instead of "references") because it is used to lookup *one* item each time it is called.

The `resolve_initial_references()` operation is a bootstrapping API that is used to find other parts of the CORBA infrastructure. It takes a string parameter that indicates which piece of CORBA it should find. Many of the CORBA Services (Section 1.6) are found by calling this operation. For example, you can obtain a reference to the Naming Service by calling this operation with the parameter `"NameService"`. Likewise, you can obtain a reference to the Notification Service by passing `"NotificationService"` as a parameter to this operation.[4] In each case, the value returned by this operation is an object reference of the base type `Object`, and the programmer must *narrow* (CORBA equivalent of a *typecast*) this to the appropriate type before using it.

The `resolve_initial_references()` operation is used not only to find CORBA Services; it is also used to obtain references to pieces of infrastructure that are part of the same process. For example, it is used to find *Current* objects (Chapter 13), a factory to create `DynAny` objects (Section 15.3), the root POA (Section 5.5), and so on.

The `resolve_initial_references()` operation is not shown in the pseudocode of Figures 3.5 or 3.6. However, real application code would use `resolve_initial_references()` to access the root POA (which is required for creating other POAs in which to store servants) and to access, say, the Naming Service or Trading Service for importing/exporting object references.

[4] The CORBA specification defines the parameter names that are used to find different CORBA services.

3.4.2 Stringified Object References

The `ORB` provides an operation called `object_to_string()` that converts an object reference (proxy) into a stringified object reference, and another operation called `string_to_object()` that converts a stringified object reference back into a proxy. The `object_to_string()` operation works by *marshaling* (Section 11.2 on page 113) the IOR into a binary buffer, converting the contents of this buffer into hexadecimal and prefixing the result with `"IOR:"`.

These `object_to_string()` and `string_to_object()` operations provide one way for a server to advertise an object to client applications. For example, a server could stringify an object reference and write this string to a file. A client application could read this stringified IOR from the file and call `string_to_object()` to convert it back into a proxy. Other ways for a server to advertise objects are discussed in Chapters 4 and 20.

Many CORBA products provide a utility that can parse a stringified object reference and print out the details embedded within it. Although such utilities are common, they are not standardized by the OMG so the names and "look and feel" of these utilities tends to vary between different CORBA products. Here are the names of some of these utilities:

- The utilities provided with Orbix and Orbacus both happen to be called `iordump`. However, these utilities were developed independently of each other and they do not have the same "look and feel".

- The omniORB and TAO utilities both happen to be called `catior`. However, these utilities were developed independently of each other and they do not have the same "look and feel".

- The JacORB utility is called `dior`, which is short for "decode IOR".

- The web site below contains a form, into which you can copy-and-paste a stringified IOR. When you then click on the `parse` button, it displays the parsed information:

  ```
  www.parc.xerox.com/istl/projects/ILU/parseIOR/
  ```

3.4.3 The Tie Approach to Implementing Servants

Section 3.2.2 on page 28 mentioned that a servant class inherits from a generated skeleton class. Actually, CORBA supports two approaches for associating a servant class with the skeleton class. One way is for the servant class to inherit from the skeleton class, as discussed in Section 3.2.2. The other way is for the IDL compiler to generate a class—usually called a *tie* class—that inherits from the skeleton class and this tie class then *delegates* to a servant class (which, in this case, does *not* inherit from the skeleton class).

In languages (such as C++) that support multiple inheritance of classes, the inheritance approach is usually preferred. In languages (such as Java) that support only single inheritance of classes, the tie (delegation) approach is often preferred since this allows the servant class to use its single inheritance for a non-CORBA purpose. The tie approach is also useful if a servant's state is to be persisted in an object database. This is because the object database should be used to persist the instance variables of just the servant class and *not* persist any instance variables inherited from CORBA infrastructure, such as a skeleton class.

Chapter 4

The Naming Service

One way for a server application to advertise an object is for the server to stringify an object reference—by calling `object_to_string()`—and write it to, say, a file (Section 3.4.2 on page 34). A client application could then read in the string, and call `string_to_object()` to turn it back into a proxy. This mechanism works fine *as long as* the client and server applications have access to a shared file system. This is likely to be the case if the client and server are running on the same computer or on the same local area network. However, it is unlikely that a wide area network will have a shared file system. For this reason, CORBA has matured over the years to support more geographically-scalable ways for a server to advertise an object. One such way, the Naming Service, is discussed in this chapter, and another way, the Trading Service, is discussed in Chapter 20.

4.1 Basic Concepts

The white pages telephone book provides a mapping from a person's name to their contact details (address and telephone number). Likewise, the CORBA Naming Service provides a mapping from a (human-readable) name to an object's "contact details" (an IOR). However, that is where the analogy ends. The names in a telephone book are arranged alphabetically, while the names in the Naming Service are arranged as a hierarchy (similar to how the file system in UNIX or Windows is arranged as a hierarchy).[1] Each level in the Naming

[1] Strictly speaking, the contents of the Naming Service are arranged in a graph that may contain cycles, so comparing the Naming Service to the World Wide Web is more accurate than comparing

Service hierarchy is called a *naming context* (while in a UNIX file system it is called a *directory* and in Windows it is called a *folder*). A naming context is itself a CORBA object (it is defined by the `CosNaming::NamingContext` interface, as shown in Figure 4.1). Because CORBA objects can be accessed regardless of their location, this means that a Naming Service hierarchy *could* be contained inside a single server process *or* the hierarchy could be spread over multiple Naming Service server processes. The concept of linking several Naming Service server processes in this way is called *federation*. Many organizations do not make use of federation and instead use a monolithic Naming Service. However, some organizations *do* make use of a federated Naming Service. This is typically done to delegate administration responsibilities. For example, an organization might have a separate Naming Service process for each department or branch and then use federation to link these separate Naming Services into one logical unit that is the “company-wide” Naming Service.

The functionality of the Naming Service is defined through the operations of its IDL interfaces. There are some operations that can be used to create/modify/delete a naming context hierarchy. There are other operations to *bind* (that is, advertise) an IOR in the Naming Service with a specified name, and yet other operations to *resolve* (that is, lookup) an IOR associated with a specified name.

It is common for an implementation of the Naming Service to provide some command-line utilities and/or a GUI tool that can be used to do administration tasks on the Naming Service, such as create/modify/delete a naming context hierarchy. Such administration functionality is implemented by invoking corresponding IDL operations on the Naming Service.

The CORBA specification for the Naming Service does not specify what *quality of service* should be offered by an implementation. Some implementations of the Naming Service hold the details of the *name→IOR* mappings in memory, which has the drawback that such details are lost whenever the Naming Service is killed and restarted, but has the benefit of not requiring access to persistent storage (which might be important in an embedded device). Other implementations of the Naming Service persist the *name→IOR* mappings in a file or database.

Most, possibly all, implementations of CORBA have a bundled implementation of the Naming Service, which means that you do not have to purchase it separately. It is possible that the Naming Service bundled with the CORBA product you are using might have a quality of service that is unsuitable for your needs. If this is the case then you could discard that Naming Service and

it to a hierarchical file system. However, in practice, a Naming Service is usually organized in a hierarchical manner.

```
#pragma prefix "omg.org"
module CosNaming {
    typedef string Istring;
    struct NameComponent {
        Istring  id;
        Istring  kind;
    };
    typedef sequence<NameComponent> Name;
    enum BindingType {nobject, ncontext};
    struct Binding {
        Name         binding_name;
        BindingType  binding_type;
    };
    typedef sequence <Binding> BindingList;

    interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many,
                                 out BindingList bl);
        void destroy();
    };

    interface NamingContext {
        void bind(in Name n, in Object obj) raises(...);
        void rebind(in Name n, in Object obj) raises(...);
        void bind_context(in Name n, in NamingContext nc)
                        raises(...);
        void rebind_context(in Name n,
                         in NamingContext nc) raises(...);
        Object resolve(in Name n) raises(...);
        void unbind(in Name n) raises(...);
        NamingContext new_context();
        NamingContext bind_new_context(in Name n)
                        raises(...);
        void destroy() raises(...);
        void list(in  unsigned long   how_many,
                  out BindingList     bl,
                  out BindingIterator bi);
    };
    ... // interface NamingContextExt omitted for brevity
};
```

Figure 4.1: Naming Service IDL

replace it with a Naming Service from a different vendor or even implement the Naming Service yourself. However, in practice, you are likely to be content with the quality of service offered by the Naming Service bundled with the CORBA product you decide to use. The ability to "rip out and replace" an implementation of the Naming Service (or another CORBA Service) is mentioned to illustrate the flexibility of an open standard (such as CORBA) that cleanly separates specification from implementation.

4.2 Imperfections in the IDL

Although the concepts in the Naming Service are simple, unfortunately the OMG made a few poor choices when defining the IDL types. This has resulted in confusion for some developers. Such confusion rapidly disappears when the motivation for the design choices are explained. This section explains the poor design choices with the aim of helping developers avoid unnecessary confusion. Readers who will not be doing development might prefer to skip this section.

The IDL types of the Naming Service are defined in the `CosNaming` module. Within this module, the `NamingContext` interface defines the operations that can be performed on a naming context. An application can connect to the root naming context of the Naming Service by passing `"NameService"` as a parameter to the `resolve_initial_references()` operation (Section 3.4.1 on page 33). The first piece of confusion likely to strike a programmer is that the parameter to this operation is `"NameService"` rather than `"NamingService"`.

Most of the remaining confusion with the API of the Naming Service concerns the format of hierarchical names within the Naming Service. In hindsight, the OMG should probably have chosen to represent a hierarchical name as a `string` that uses `"/"` as a separator between naming contexts, for example, `"path/in/naming/service"`. However, there were several perceived problems with this:

- The OMG wanted the hierarchical names to be expressed in multi-byte character strings rather than single-byte character strings. However, at the time that the Naming Service was being defined, the `wstring` type had not been introduced to IDL.

- Using `"/"` as a hierarchical separator would be familiar to people from a UNIX background. However, hierarchical file systems in other operating systems (for example, MS-DOS, VMS and MacOS) used different syntaxes for their hierarchical separator. There was no compelling reason to arbitrarily choose one separator syntax over another.

- Many file systems have the concept of a filename being composed of a *root-name* and an *extension*. For example, in the filename `"foo.txt"`, the root-name is `"foo"` and the extension is `"txt"`, with `"."` separating the two. Some people felt that similar flexibility might be useful in a hierarchical name in a Naming Service.

The OMG tried to resolve the `string/wstring` dilemma by introducing the following definition:

```
typedef string Istring;
```

The intention was that `Istring`[2] would be used in the Naming Service API and when `wstring` was later introduced to IDL then the definition could be changed to:

```
typedef wstring Istring;
```

No doubt this seemed like a good idea at the time. However, it was fatally flawed because this proposed future change in the definition of `Istring` would have been backwards incompatible. Because of this backwards incompatibility, the change in the definition was never made. The net effect is that the IDL definitions for the Naming Service have a useless `typedef` definition, which confuses some people.

To avoid hard-coding use of an arbitrary syntax (for example, `"/"`) as a hierarchical separator, the OMG decided that a hierarchical name should be represented as a `sequence` of components. In order to allow support for `foo.txt`-style components in a name, the OMG decided that each component should be a `struct` that contains two fields. The resulting definition of a hierarchical name is shown below:

```
typedef string Istring;
struct NameComponent {
    Istring  id;   // denotes the "foo" part of "foo.txt"
    Istring  kind; // denotes the "txt" part of "foo.txt"
};
typedef sequence<NameComponent> Name;
```

The result is certainly flexible, but it is also over-engineered. Most people do not need all this flexibility and become frustrated with the complexity that it introduces to their applications. For example, consider an application that

[2] The `"I"` in `Istring` stands for *internationalization*.

reads a string of the form "path/in/naming/service" from a runtime configuration file. The developers of such an application have to write their own function to convert the string into the CosNaming::Name format. One problem is that it is a waste of time for numerous developers in different organizations around the world to re-invent the wheel in writing such a utility function. Another problem is that each of these developers must choose what hierarchical separator they want and the separator between the id and kind fields of NameComponent. As it turns out, most developers choose to use UNIX-style separators of "/" and ".".

```
#pragma prefix "omg.org"
module CosNaming {
    ...
    interface NamingContextExt : NamingContext {
        typedef string StringName;
        StringName to_string(in Name n) raises(...);
        Name to_name(in StringName sn) raises(...);
        Object resolve_str(in StringName sn) raises(...);
        ...
    };
};
```

Figure 4.2: Extract from the NamingContextExt interface

Hindsight is a wonderful thing, and a few years later the OMG decided to simplify the complexity of hierarchical names by defining a new version of the Naming Service. As I will discuss in Section 9.3 on page 100, IDL does not have a versioning mechanism, so the simplified Naming Service was defined by defining a sub-type that inherits from NamingContext.[3] An extract of this new type can be seen in Figure 4.2 (only some of the operations are shown and the raises clauses have been omitted for brevity). The to_string() and to_name() operations convert between the CosNaming::Name and "path/in/naming/service" formats. As an extra convenience, clients can call resolve_str() to resolve (that is, lookup) an IOR from the Naming Service without having to convert "path/in/naming/service" into CosNaming::Name format. However, there is no similar utility operation defined for the bind() or rebind() operations that servers use to advertise

[3] This new version of the Naming Service was defined at the same time as the corbaloc and corbaname URLs (Chapter 12). The combined result was termed the *Interoperable Naming Service* because corbaloc and corbaname addressed some bootstrapping interoperability issues.

an IOR in the Naming Service.

The final area of potential confusion for developers is the `list()` operation, which is illustrated in Figure 4.1 on page 39. An `out` parameter of this operation provides a *non-recursive* listing of the entries in a naming context. Because the listing is non-recursive, the `binding_name` field of each `Binding` *should be* of type `NameComponent`. The accidental use of type `Name` makes some developers mistakenly think that this operation provides a recursive listing.

This section has discussed the imperfections of the Naming Service API. These imperfections can cause confusion for new developers. However, once developers are aware of these imperfections, use of the Naming Service is straightforward. Certainly the imperfections of this API seem minor in comparison to imperfections in the APIs of some other (non-middleware) systems.

4.3 Practical Usage of the Naming Service

The Naming Service defines IDL operations that can be used to create, modify and delete a hierarchy of `NamingContexts`. However, performing Naming Service maintenance by writing programs that invoke these operations is tedious. Virtually all implementations of the Naming Service provide command-line utilities and/or a GUI "wrapper" around these IDL operations. These command-line utilities and/or GUI programs provide a practical way to perform administration of a Naming Service. Note, however, that such utilities are not part of the CORBA specification. Because of this, the utilities provided will vary from one CORBA product to another.

Once administration issues have been addressed, the only remaining programming aspects associated with the Naming Service are how a server application can `bind()`, that is, export, an object reference to the Naming Service, and how a client application can `resolve()`, that is, import, an object reference from the Naming Service. These are straightforward programming tasks.

The *Importing and Exporting Object References* chapter of the *CORBA Utilities* package [McH] discusses a freely available library (implemented in C++ and Java) that provides utility functions called `importObjRef()` and `exportObjRef()`. As their names suggest, these utility functions are for importing and exporting object references. A pseudocode example of their use is illustrated in Figure 4.3. These functions take an `instructions` parameter that specifies how to import or export an object reference. An `instructions` parameter that starts with `"name_service#"` indicates that the Naming Service should be used. The utility functions also support

```
try {
    instructions1 = "name_service#path/in/Naming/Service";
    instructions2 = "file#/path/to/file.ior";
    obj1 = importObjRef(orb, instructions1);
    exportObjRef(orb, obj2, instructions2);
} catch(ImportExportException ex) {
    cout << ex << endl;
}
```

Figure 4.3: `importObjRef()` and `exportObjRef()`

`"file#<filename>"` and `"exec#<command>"` for importing or exporting object references through files or by executing an external command. Ideally, an application should not hardcode the value of the `instructions` parameter, but rather should get it from a command-line option or a configuration file. Doing this provides applications with a lot of flexibility in how they import/export object references. Aside from the flexibility, these utility functions provide a simpler API than that of the Naming Service.

Chapter 5

Concepts for Server-side Programming

5.1 Object

A company is just a *concept* (with supporting, legal documentation) rather than a physical entity. For example, a building owned by the company is *not* the company. Likewise, an employee is *not* the company (but he or she can represent the company). Just as a company is a concept rather than a physical entity, so too, a CORBA object is just a *concept*, rather than a physical entity.

5.2 Servant

An *employee* represents a company. For example, let us assume that a utility company called *EasyGas* supplies gas to your home. You might say to a friend, “I talked to EasyGas yesterday to change my payment plan.” Technically, you did *not* talk to EasyGas because EasyGas is a company—a legal concept—and it is impossible to talk to a concept. What you actually did was talk to an *employee* (a representative) of EasyGas. In the same way, a programmer may say, “This CORBA client invokes upon a CORBA object in the billing server.” Technically, a CORBA object is just a concept so there is no way to invoke an operation upon it. Instead, the invocation is handled by a *servant* that *represents* the CORBA object.

A servant is simply a data-structure or object in the programming language that is used to implement a server application. For example, a servant might be a C++ object or a Java object.

5.3 Why are Object and Servant Different Concepts?

An obvious question is why does the CORBA specification make a distinction between a *CORBA object* and a *servant* (C++/Java object) that represents it? The answer can be explained by the following analogy with a company and its employees.

The lifetime of a company is independent of the lifetime of an individual employee. For example, when you telephoned EasyGas yesterday, you might have spoken to an employee called John. If John leaves EasyGas then when you telephone EasyGas again tomorrow, you might talk to another employee, say, Mary. In the same way, when a CORBA client makes an invocation upon a CORBA object, the request might be handled by a particular servant (C++ or Java object). If the server process is later killed then obviously the servant (C++ or Java object) will be destroyed. Then, if the server is restarted, a *new* servant will be created to represent the same CORBA object. What all this means is that a future invocation by the client application upon the *same* CORBA object may be handled by a different servant. In turn, this means that a CORBA object can be "long lived", that is, it can survive a stop-and-restart of a server process (just as a company can survive the resignation of one employee and the hiring of a new, replacement employee).

5.4 Object Adapters

An *object adapter* (OA) is the part of the CORBA runtime system that "adapts" the concepts of a CORBA object to the realities of the host programming language and server process. In practice, this means that the OA deals with low-level issues, such as:

- Reading incoming requests from, say, a socket connection, unmarshaling the request's parameters and dispatching to the relevant operation on a servant.

- Providing APIs that allow an object reference to be mapped to the corresponding servant, and vice versa.

- On-demand creation of servants and the later saving of servant state to a persistent store (for example, a file or a database).

The CORBA specification is deliberately vague about the capabilities of an object adapter. This is because the OMG felt that it would not be feasible to have a one-size-fits-all object adapter. Instead, the OMG expected that several different object adapters would be developed. The first version of the CORBA specification defined a Basic Object Adapter (BOA) and suggested that other object adapters—perhaps a Database Object Adapter or a Real-time Object Adapter—could be developed. Unfortunately, this goal was not achieved. The reason for this is that the BOA was under-specified so CORBA vendors *had* to add their own proprietary enhancements just to get a working system. Having done this, CORBA vendors decided to *not* define new object adapters for, say, database integration or to support real-time programming. Instead, they just added more and more proprietary APIs to the BOA to provide the desired functionality. The result was that each CORBA vendor had their own proprietary version of the BOA, and this hindered source-code portability of applications across different CORBA products.

After a few years, the OMG decided to discard the BOA and to replace it with the Portable Object Adapter (POA). The POA is superior to the BOA in two ways:

1. The POA has much more built-in functionality than the BOA. This dramatically reduces the need for CORBA vendors to add their own proprietary enhancements. It also reduces the need for programmers to *use* proprietary enhancements and hence increases source-code portability of applications across different CORBA implementations.

2. The architecture of the POA provides an open-ended way for new functionality to be added in the future. This provides a way for the OMG to incrementally built upon the capabilities of the POA, in a backwards-compatible way. It also allows CORBA vendors to provide proprietary enhancements in a way that retains the "look and feel" of existing POA-based APIs.

5.5 Portable Object Adapter (POA)

A POA is a *collection* of servants. This is a collection in the same sense that arrays, linked lists, hash tables and sets are collections. A POA is created with certain *policy* (quality of service) values. The policy values associated with a POA are applied to the servants contained within the POA. For example, a

POA might be multi-threaded or single-threaded.[1] If the POA is multi-threaded then it can dispatch requests concurrently to the servants contained inside it. Conversely, if a POA is single-threaded then it ensures that the dispatching of requests is serialized for the servants that it contains.

A server can have several POAs. Because each POA can have different policy values—and the policy values are applied to the servants contained within a POA—this means that you can apply different policy values to different (collections of) servants. For example, if some servants are implemented in a thread-safe manner then you could put them into a multi-threaded POA. Conversely, if some other servants are *not* thread-safe then you could put them into a single-threaded POA.

The CORBA specification does not place any restrictions on how many POAs you may have in an application, or on how many servants may be within one POA. For example, at one extreme, you could place *all* your servants into the same POA. At the other extreme, you could create a separate POA for each servant. A good rule of thumb is to have a separate POA for each IDL interface. This gives you the flexibility of choosing different policy values for different IDL interfaces. Each POA can be given an arbitrary name. If you have a separate POA for each IDL interface then a good convention is to use the name of an IDL interface as the name of its associated POA.

5.5.1 POA Hierarchy

Within any type of hierarchical diagram, the lines connecting the nodes in the hierarchy indicate a relationship. For example, in an organizational hierarchy the lines indicate a *reports to* relationship, while in an object-orientation type hierarchy the lines indicate an *is a* relationship.

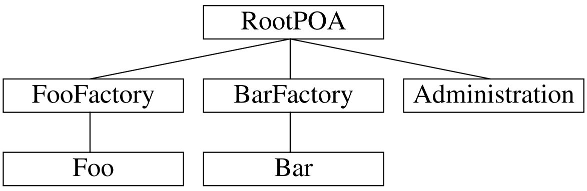


Figure 5.1: Example POA hierarchy

All the POAs in a CORBA server are arranged in a hierarchy. The lines

[1] Saying that a POA is multi-threaded or single-threaded is a slight simplification. This issue is discussed in more detail in Chapter 6.

within a POA hierarchy indicate an *order of destruction* relationship. For example, consider the POA hierarchy in Figure 5.1. The CORBA runtime system provides a built-in POA called the *root POA*.[2] As its name suggests, this node forms the root of the POA hierarchy. The order of destruction guarantee means that, at server shutdown time, the POAs will be destroyed in the following order:

- The *Foo* POA will be destroyed before the *FooFactory* POA. Likewise, *Bar* will be destroyed before *BarFactory*.

- The *FooFactory*, *BarFactory* and *Administration* POAs will be destroyed before the *RootPOA*.

No order of destruction guarantees are given for sibling nodes. For example, the relative order of destruction of the *Foo* and *Bar* POAs is not guaranteed. Neither is the relative order of destruction guaranteed for the *FooFactory*, *BarFactory* or *Administration* POAs.

The reason why the order of destruction is useful is that the programmer can optionally arrange for the servants within a POA to be destroyed when the containing POA is destroyed.[3] In essence, a programmer can use the order of destruction of the POA hierarchy to impose an order of destruction on servants.

POA hierarchies tend to be very shallow. There are two reasons for this. First, most CORBA servers tend to implement a relatively small number of IDL interfaces (typically less than 5). Since it is natural to have one POA for each IDL interface, this means that there will be only a few POAs within a server process. Second, CORBA server applications tend to have only simple order-of-destruction requirements, and this is reflected in a very flat hierarchical structure.

[2] The root POA is accessed by calling `resolve_initial_references("RootPOA")` on the ORB and then "narrowing" down to the appropriate type (`PortableServer::POA`). The `resolve_initial_references()` operation is discussed in Section 3.4.1 on page 33.

[3] In some programming languages, for example, C++, servants are reference counted. In such languages, the programmer can arrange for POAs to hold the only references to servants. Then as each POA in the POA hierarchy is being destroyed, the POA decrements (to zero) the reference count of all of its servants. This has the effect of destroying the servants at the same time as the POA in which they are contained. An alternative technique—and one which works regardless of whether or not servants are reference counted—is to use the "lazy loader" POA model (discussed in Section 5.6.2). When such a POA is being destroyed, it calls the `etherealize()` operation on its `ServantActivator` (lazy loader) for each servant. The programmer can implement `etherealize()` so it performs destruction-time behavior for servants.

5.6 Different Kinds of POA

Section 5.5 mentioned that a POA is a collection of servants. In fact, a POA can be one of several kinds of collection. This allows programmers to make various tradeoffs regarding the lifecycle of servants and the relationship between servants and the CORBA objects that they represent. For example:

- Must a servant be created *before* requests arrive for the corresponding CORBA object? Or can servants be created on an as-needed basis?
- Once a servant has been created, does it live indefinitely (until the server process terminates)? Or can a servant be destroyed and later another servant be created to take its place? If this latter approach is taken then it could be used to implement a cache that contains servants for recently-accessed CORBA objects.
- Is there a one-to-one mapping between servants and the CORBA objects that they represent? Or could one servant be used to represent *many* CORBA objects?

The possibilities listed above allow programmers to choose different techniques in order to meet a wide variety of performance and scalability requirements.

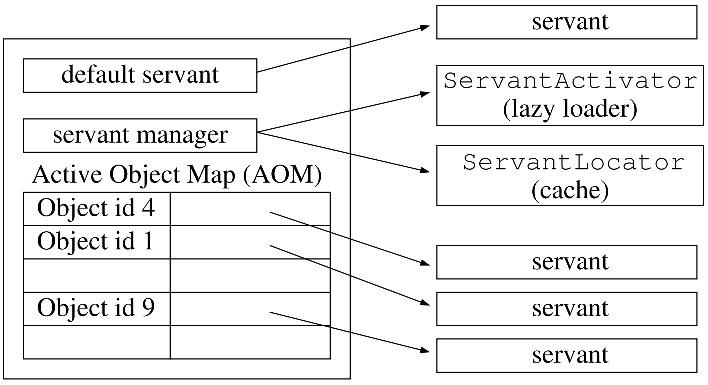


Figure 5.2: Possible components of a POA

In order to provide these possibilities, a POA must be a flexible type of container. Figure 5.2 shows the logical structure of a POA. This figure illustrates that a POA may have: a *default servant*, one of two subtypes of *servant* *manager* (either a `ServantActivator` or a `ServantLocator`), and/or an *active object map* (AOM). The meaning of these components will be made clear later. It is impossible for a POA to have all of these components at the same time. Instead, a POA has at most two of them. Which components a POA has determines the performance and scalability characteristics of the POA. The following subsections discuss four common "kinds" of POA, where each kind makes use of a certain combination of the components.

5.6.1 POA kind 1: "Simple"

The simplest kind of POA is one that has just an active object map. This is illustrated in Figure 5.3. The term *active object map* (AOM) is not very intuitive so it worthwhile explaining it. The term *map* means a lookup table that "maps" a name to a value. The term *active object* means a CORBA object that is "active", that is, *currently* has a servant associated with it.

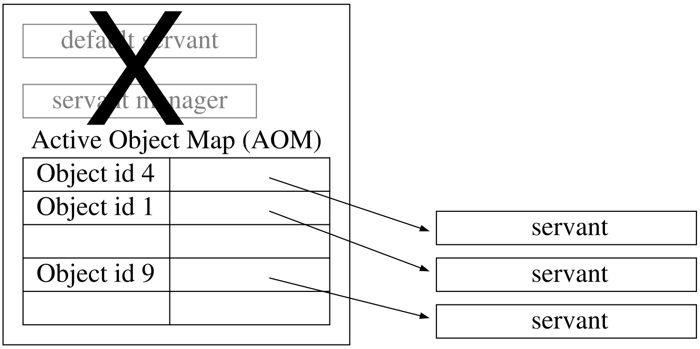


Figure 5.3: A "simple" POA

An IOR (Chapter 10) contains the "contact details" that a client application uses to communicate with a CORBA object. These contact details include the host and port of the server process, and also an *object key* that uniquely identifies the target object within the server process—the object key information is required because there may be many objects within the same server process. The format of an object key varies from one CORBA product to another, but it typically contains the following information:

- The hierarchical name of the POA that contains a servant for the CORBA object.

- An *object id* that uniquely identifies a servant within the POA. The object id is represented as arbitrary binary data,[4] which allows developers to store, say, the *primary key* of a database record in the object id. Doing this provides a convenient way to put a CORBA object "wrapper" around information in a database.

- If the POA has the `PERSISTENT` policy (discussed in Section 6.1.3 on page 62) and if the server is being deployed through an implementation repository (IMR) then the object key may contain a name that uniquely identifies the server to the IMR. The reason for this is discussed in Section 7.2.2 on page 71.

- If the POA has the `TRANSIENT` policy (discussed in Section 6.1.3 on page 62) then the object key may contain a timestamp.

When a request arrives at a server, the header of the request contains the *object key* for the target object. The CORBA runtime system in the server extracts the POA name and the *object id* information from the object key. The CORBA runtime system uses this *object id* to perform a lookup in the AOM of the specified POA. If there is a servant associated with this object id then the request is dispatched to that servant. Otherwise, an `OBJECT_NOT_EXIST` exception is thrown back to the client.

You can use a "simple" POA under the following conditions:

- You have enough memory to store all the servants in memory at the same time. A well-designed CORBA product might have an overhead as low as 30 bytes for each servant in a POA, but application-level instance variables might account for much more memory consumption. Typically, a "simple" POA can be used for up to, say, 100 or 1000 servants. Memory requirements mean that it is unlikely to scale up to one million servants.

- You can pre-create a servant for a CORBA object *before* receiving the first request for that object.

5.6.2 POA kind 2: "Lazy Loader"

In this POA model (Figure 5.4), the server programmer creates the POA and then associates it with an object that implements the `ServantActivator` interface. The name *ServantActivator* is not very intuitive; perhaps *lazy loader*

[4] The *object id* is a `sequence<octet>`. An `octet` is the built-in IDL type that denotes a byte of raw data.

would be a better name because its purpose is to lazily load (or "re-create on demand") servants. It is the programmer's responsibility to implement the lazy loader class.

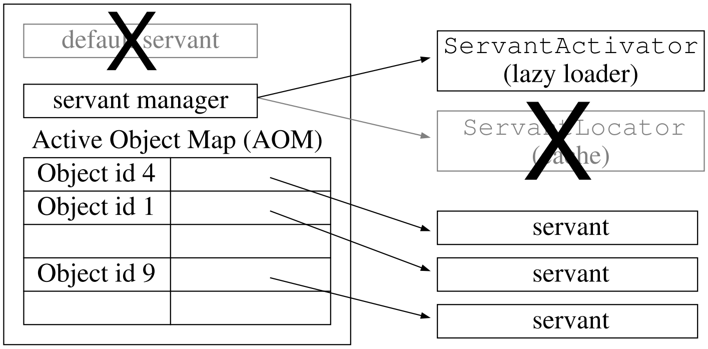


Figure 5.4: A "lazy loader" POA

When a request arrives at a server, the CORBA runtime extracts the POA name and the *object id* information from the object key. The CORBA runtime system uses this *object id* to perform a lookup in the AOM of the specified POA. If there is a servant associated with this object id then the request is dispatched to that servant. Otherwise, the POA invokes the `incarnate()` operation on the lazy loader to ask it to re-create the desired servant, and the POA then adds this servant into its AOM. If the lazy loader is unable or unwilling to re-create the servant then an `OBJECT_NOT_EXIST` exception is thrown back to the client.

You can use a "lazy loader" POA under the following conditions:

- You have enough memory to store all the servants in memory at the same time. A well-designed CORBA product might have an overhead as low as 30 bytes for each servant in a POA, but application-level instance variables might account for much more memory consumption. Typically, a "lazy loader" POA can be used for up to, say, 100 or 1000 servants. Memory requirements mean that it is unlikely to scale up to one million servants.

- You do *not* want to pre-create a servant for a CORBA object *before* receiving the first request for that object. Instead, you prefer to create the servants on an "as needed" basis. You typically do this for one of two reasons:

1. Perhaps each servant takes several seconds to be initialized. If you pre-created 10 or 20 such servants then the server application might take a few minutes to complete its initialization. Instead of suffering this long start-up time, you might prefer for the server to start up very quickly and then suffer a few seconds of delay each time a servant is accessed for the first time.
2. A CORBA server may have many objects but it is likely that only a few of them will be used during the server's uptime. In such a situation, creating the servants on an "as needed" basis helps to conserve memory.

5.6.3 POA kind 3: "Cache"

In this POA model (Figure 5.5), the server programmer creates the POA and then associates it with an object that implements the `ServantLocator` interface. The name *ServantLocator* is not very intuitive; *cache* would be a better name because its purpose is to implement a cache of servants. It is the programmer's responsibility to implement the cache class.

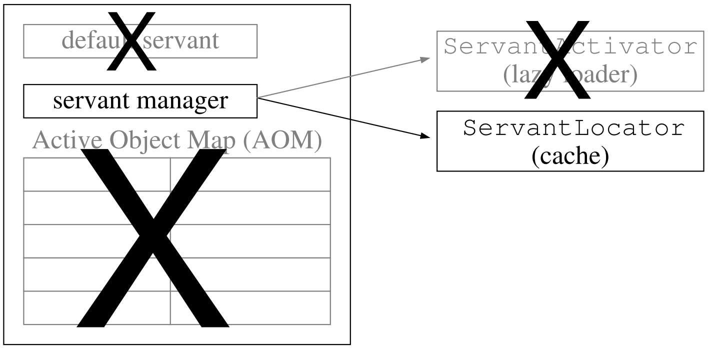


Figure 5.5: A "cache" POA

When a request arrives at a server, the CORBA runtime passes control to the POA specified in the object key in the header of the request. The POA then uses the following (pseudocode) algorithm to dispatch the request:

```
sv = cache.preinvoke(object_id,...);
sv.operation(...);
cache.postinvoke(..., sv, ...);
```

The POA does *not* maintain its own AOM. This is because it assumes that the cache will implement its own AOM-like data-structure and there would be no point in the POA replicating this effort.

You can use a "cache" POA if you have enough memory to store *some but not all* of the servants in memory at the same time. Typically, the CORBA server will be front-ending a database. To optimize access to information in the database, you decide to cache some of that database information in servants (memory). Of course, caching information in memory has associated dangers. In particular, if you update the cached/in-memory information and do not immediately flush this information to the database then you run two risks. First, if another application accesses the database directly then it will see information that is different to what is in the servants. In other words, you can have a cache inconsistency problem. Second, if your CORBA server is killed before it has a chance to flush its cached information back to the database then you will lose some data. However, the "cache" POA model works very well with data in a read-only (reference) database.

5.6.4 POA kind 4: "Default Servant"

In this POA model (Figure 5.6), the server programmer creates the POA and then associates it with just one servant. This servant is called a *default servant*. Whenever a request arrives for *any* CORBA object in that POA, the POA dispatches the request to the default servant. The default servant calls the `get_object_id()` operation on the *POACurrent* (Chapter 13) to find out which CORBA object it is representing for the current request. Typically, the servant will use this object identifier as, say, the primary key into a database table.

The "default servant" POA model minimizes memory consumption because a single servant can be used to service requests for (literally) an infinite number of CORBA objects.[5] This minimal memory consumption comes at the price of a time overhead because the default servant must use the object identifier (obtained from the *POACurrent*) to access persistent data *every* time an operation is invoked. In other words, a default servant does *not* normally cache any data in memory for faster access.

5.6.5 Other POA kinds

The four kinds of POA discussed in Sections 5.6.1 to 5.6.4 are the most common and useful POA models. You create a specific kind of POA by spec-

[5] In non-CORBA terminology, a default servant is often called a *flyweight object*.

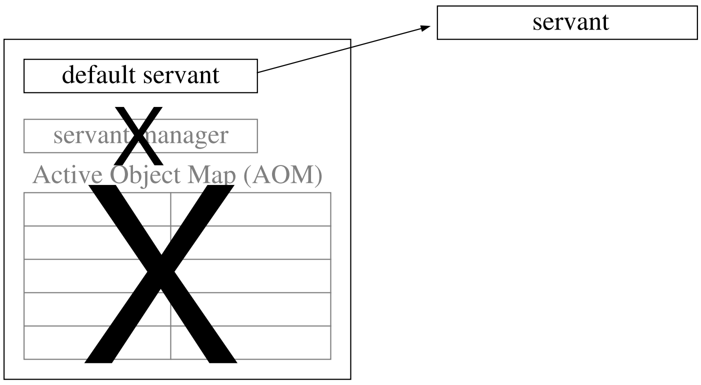


Figure 5.6: A "default servant" POA

ifying a corresponding combination of POA *policies* as a parameter to the `create_POA()` operation.[6] There are other legal combinations of POA policies that can result in other kinds of POA. However, these other kinds of POA are not always useful. For example, a "default servant" POA *may* also have an active object map. When a request arrives at such a POA, the following happens:

- If there is an entry in the active object map for the target object identifier then the request is dispatched to the corresponding servant .
- Otherwise, the request is dispatched to the POA's default servant.

It is difficult to think of a non-contrived use for such a POA. Instead, it is usually clearer to split this POA into two POAs: a "default servant" POA (without an activate object map) and a separate "simple" POA.

5.7 POA Managers

A water tap (or *faucet* as it is called in some countries) is used for turning on and off the flow of water. Conceptually, it has two states: "on" allows water to flow, and "off" prevents water from flowing.

A POA manager is similar to a water tap except that, instead of controlling the flow of water, it controls the flow of incoming requests. A POA manager can be in one of four states:

[6] POA policies are discussed in Chapter 6.

`HOLDING` This is an "off" state. Incoming requests are queued up.

`DISCARDING` This an an "off" state. Each incoming request is discarded and a `CORBA::TRANSIENT` system exception is thrown back to the client.[7]

`ACTIVE` This is the "on" state. Incoming requests are dispatched normally to servants.

`INACTIVE` This an an "off" state. This state is entered automatically when the server is shutting down. Incoming requests are rejected in a vendor-specific manner.

The name *POA manager* is somewhat undescriptive. A better name might have been *POA request valve*.

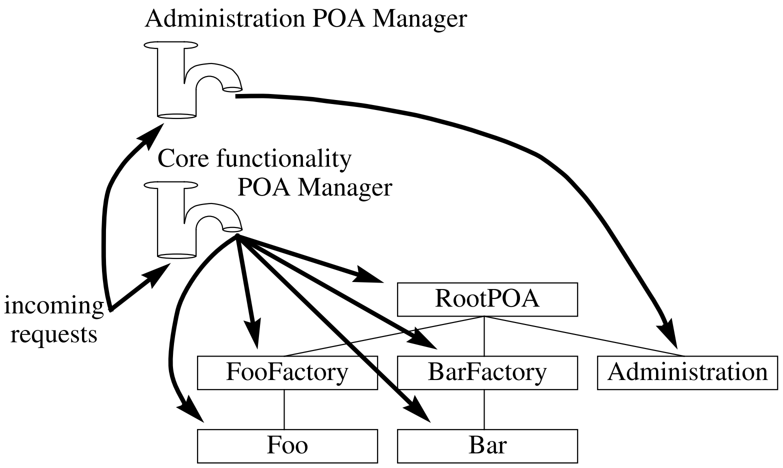


Figure 5.7: Example of POA managers

There is a one-to-many relationship between POA managers and POAs: *one* POA manager controls the dispatching of requests for all servants in *many* POAs. It is a good idea for a server application to have two POA managers, as shown in Figure 5.7. One POA manager controls the dispatching of requests for servants in an "administration" POA and another POA manager controls the dispatching of requests for servants in the "core functionality" POAs. In this way, the server can selectively enable/disable its core functionality while *always* servicing "administration" requests.

[7] A `CORBA::TRANSIENT` exception means "temporary error; please try again later'.

```
module PortableServer {
  ...
  local interface POAManager {
      enum State {HOLDING, ACTIVE, DISCARDING,
                  INACTIVE};
      State get_state();
      void activate() raises(...);
      void hold_requests(...) raises(...);
      void discard_requests(...) raises(...);
      void deactivate(...) raises(...);
  };
};
```

Figure 5.8: API for a POA manager

The API for POA managers is defined in IDL, as shown in Figure 5.8. You can query the current state of a POA manager by calling `get_state()`. You can switch a POA manager into another state by calling `activate()`, `hold_requests()`, `discard_requests()` or `deactivate()`. Initially, a POA manager is in the `HOLDING` state. This is useful because it prevents incoming requests from being dispatched while a server is performing its application-level initialization. Once initialization is complete, a server program should call `activate()` on all its POA managers and then go into an event loop (typically by calling `run()` on the `ORB`).

Chapter 6

POA Policies

Section 5.6 on page 50 discussed several different kinds of POA, such as “simple”, “lazy loader”, “cache” and “default servant”. However, that section did not explain *how* to create a POA of a specific kind. *How* to create a POA of a specific kind is the topic of this chapter.

CORBA uses the term *policy* to mean *quality of service* (QoS). A POA is created by calling the `create_POA()` operation. One of the parameters to this operation is a sequence of policy objects. Section 6.1 discusses the different sub-types of `Policy` object that are available, and how some of them combine together to create different kinds of POA. Finally, Section 6.2 briefly outlines the APIs that are used to create a policy object.

6.1 Available POA Policies

Some of the POA policies determine what kind of POA is created; these policies are discussed in Section 6.1.1. Some other POA policies are used to specify other QoS offered by the POA, and these are discussed in Sections 6.1.2–6.1.6.

6.1.1 Policies that Determine the POA’s Kind

A policy object is a “wrapper” around an `enum` value. In this section, I discuss the `enum` values; a discussion of how to create object wrappers around these values is deferred until Section 6.2.

The following three `enum` types, when combined together, determine a POA’s kind:

```
enum ServantRetentionPolicyValue {RETAIN, NON_RETAIN};
enum IdUniquenessPolicyValue {UNIQUE_ID, MULTIPLE_ID};
enum RequestProcessingPolicyValue
                            {USE_ACTIVE_OBJECT_MAP_ONLY,
                             USE_DEFAULT_SERVANT,
                             USE_SERVANT_MANAGER};
```

The first `enum` determines whether or not object ids are *retained* in the POA by an active object map (AOM). Many developers find these `enum` values difficult to remember. More meaningful names might have been `HAS_AN_AOM` and `DOES_NOT_HAVE_AN_AOM` instead of `RETAIN` and `NON_RETAIN`, respectively.

The second `enum` determines whether there is a one-to-one mapping (the `UNIQUE_ID` value) or a many-to-one mapping (`MULTIPLE_ID`) between object ids and servants. The `MULTIPLE_ID` policy allows one servant to represent many CORBA objects. You *must* use the `MULTIPLE_ID` policy for the "default servant" POA kind (Section 5.6.4 on page 55). For the other kinds of POA, you can use either policy, but the `UNIQUE_ID` policy is what is desired most of the time.

The third `enum` determines whether the POA has an AOM, a default servant and/or a servant manager:

- The `USE_ACTIVE_OBJECT_MAP_ONLY` policy value is used to obtain the "simple" kind of POA (Section 5.6.1 on page 51). You *must* combine this policy with `RETAIN`. You typically also combine this with `UNIQUE_ID`, but this is not required.

- The `USE_DEFAULT_SERVANT` policy value is used to obtain the "default servant" kind of POA (Section 5.6.4 on page 55). You *must* combine this policy with `MULTIPLE_ID`. You typically also combine this with `NON_RETAIN`, but this is not required.

- If you combine `USE_SERVANT_MANAGER` with `RETAIN` then you obtain a "lazy loader" POA (Section 5.6.2 on page 52). If, instead, you combine it with `NON_RETAIN` then you obtain a "cache" POA (Section 5.6.3 on page 54). The `USE_SERVANT_MANAGER` policy is also typically combined with `UNIQUE_ID`, but this is not required.

6.1.2 Multi- and Single-threaded Policy Values

The following `enum` is used to specify how a POA utilizes threads to dispatch incoming requests:

```
enum ThreadPolicyValue  {ORB_CTRL_MODEL,
                         SINGLE_THREAD_MODEL,
                         MAIN_THREAD_MODEL};
```

These policy values are discussed in the following sub-subsections.

6.1.2.1 The ORB_CTRL_MODEL Policy Value

The ORB_CTRL_MODEL policy specifies that the ORB runtime system has control over how incoming requests are dispatched. This policy value has very under-specified semantics:

- Most CORBA products use multiple threads to dispatch requests with this policy. However, the details of how the multiple requests are used varies widely from one CORBA product to another. For example, a CORBA product might: (1) use a pool of threads to dispatch incoming requests; (2) use a thread-per-request model; (3) use a separate thread per (socket) connection for dispatching requests; or (4) some other strategy.
- The MICO freeware implementation of CORBA was originally implemented without multi-threading support. In MICO, ORB_CTRL_MODEL used to employ a single-threaded dispatch algorithm, and this was considered to be a compliant implementation. Multi-threading support (and a multi-threaded dispatch algorithm) has since been retrofitted to MICO.

The under-specified semantics of ORB_CTRL_MODEL hinder the portability of CORBA applications. Many people hope that the OMG will add better-defined multi-threading policy values in the future.

6.1.2.2 The SINGLE_THREAD_MODEL and MAIN_THREAD_MODEL Policy Values

You would think that the SINGLE_THREAD_MODEL policy would have obvious semantics. However, it is actually ambiguous:

- Some CORBA vendors interpreted this policy to mean that all incoming requests are dispatched through the application's main thread. With this interpretation, there is serialization of request dispatch across *all* the SINGLE_THREAD_MODEL POAs within a process.
- Other CORBA vendors interpreted this policy to mean that there is concurrency *between* multiple SINGLE_THREAD_MODEL POAs, but serialization *within* a POA.

When the OMG became aware of this ambiguity, they resolved it by declaring that SINGLE_THREAD_MODEL had the latter set of semantics, and they introduced a new enum value, called MAIN_THREAD_MODEL that had the first set of semantics.

Readers should note that some CORBA products that ascribed the "wrong" semantics to SINGLE_THREAD_MODEL have not yet added support for the (newer) MAIN_THREAD_MODEL policy value. For this reason, it is important to carefully read the documentation of a CORBA product to check what semantics it provides for SINGLE_THREAD_MODEL POAs.

6.1.3 Policy Values for Object Lifetimes and Naming

The following enum types are used to specify the lifetimes of objects and how they are assigned object ids:

```
enum LifespanPolicyValue {TRANSIENT, PERSISTENT};
enum IdAssignmentPolicyValue {USER_ID, SYSTEM_ID};
```

CORBA uses the term *transient* to mean *temporary*. Thus, the TRANSIENT policy value specifies that object references (for objects within the POA) are valid *only* for the duration of the server process. In other words, the object references are *not* valid if the server is killed and restarted. How a CORBA product enforces this is an implementation detail, but it is typically done by embedding a timestamp into IORs. The CORBA runtime system can use this timestamp information to differentiate between references to objects that are "similar" but have been created in different runs of a server process.

The PERSISTENT policy value specifies that object references (for objects within the POA) are valid *even if* the server is killed and restarted.

It is important to note that the TRANSIENT and PERSISTENT policies determine the lifetimes of object *references*. These policies do *not* determine whether *data* associated with an object is maintained in volatile RAM or in a persistent store, such as a file or database. The following are typical examples of how these policy values are used:

- Some CORBA objects, such as Customer or Account, are associated with records in a database. It makes sense that such objects are kept in a PERSISTENT POA, so that a client's reference to such an object/data will be valid if the server process dies and is restarted.

- Some objects in a server process—such as an *administration* or *factory* object—are not associated with records in a database. However, these "stateless" objects are also normally held in a PERSISTENT POA. This ensures that a client's reference to such an object will be valid if the server process dies and is restarted.

- Some objects in a server process are intended to be temporary. One example is `CosNaming::BindingIterator` in the Naming Service, which is used to traverse over a collection of values. Another example is a `LoginSession` object that holds details specific to a client that is currently "logged in". Although such objects are stateful, they typically maintain their state in RAM, rather than in a database. Because of the temporary nature of these objects, there is no requirement for a client reference to such an object to remain valid if the server process dies and is restarted. In fact, it is preferable for the object reference to *not* be valid across restarts of the server. For this reason, such objects should be held in a `TRANSIENT` POA.

In CORBA terminology, programmers are referred to as *users*. Hence, the `USER_ID` policy value indicates that the programmer specifies the object id when *activating* (inserting) a servant into a POA. In this case, the programmer uses the `activate_object_with_id()` operation, as shown below:

```
poa.activate_object_with_id(obj_id, sv);
```

The `SYSTEM_ID` policy indicates that the CORBA runtime system picks a unique object id when a servant is activated into a POA. In this case, the programmer uses the `activate_object()` operation, as shown below:

```
obj_id = poa.activate_object(sv);
```

Technically, the `TRANSIENT` and `PERSISTENT` policies are independent of the `USER_ID` and `SYSTEM_ID` policies. However, `PERSISTENT` is almost always combined with `USER_ID`, for the following reasons:

- If a programmer wishes to maintain the state of an object in a database then he or she can use the primary key of a row in a database table as the object id. This provides a very simple way to map between a CORBA object and the corresponding data in the database.

- A stateless singleton—such as an *administration* or *factory* object—should be in a POA with the `USER_ID` policy so that the same object id can be used for the object every time the server process starts. In this way, a client's reference to the object will be valid across restarts of the server.

The `TRANSIENT` policy is usually combined with `SYSTEM_ID` because programmers are rarely concerned with assigning meaningful names to temporary objects.

6.1.4 Transactional Object Policy Values

Some client-server systems require the ability for a database transaction to span *multiple* operation calls from a client to one server and/or the ability for a transaction to span multiple databases. Such client-server interactions require use of the CORBA Object Transaction Service (OTS), which is discussed in Chapter 21.

A POA policy is used to indicate whether or not the objects within that POA can take part in distributed transactions.[1] The policy can have one of the following values:

`REQUIRES` This policy value indicates that all invocations on objects within the POA *must* be part of a transaction.[2]

`FORBIDS` This policy value indicates that all invocations on objects within the POA must *not* be part of a transaction.

`ADAPTS` This policy value indicates that objects are sensitive to the presence or absence of a transaction and can "adapt" to either style of invocation.

6.1.5 Implicit and Explicit Activation Policy Values

Most policy values used with POAs have some effect that either is visible to clients or affects the high-level architecture of the server program. The `ImplicitActivationPolicyValue` is different in that it controls a relatively minor aspect of coding a server.

```
enum ImplicitActivationPolicyValue
      {IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION};
```

To make a servant represent a CORBA object requires three steps:

1. You create a servant.
2. You activate (insert) the servant into a POA.
3. You call `_this()` on the servant to obtain the object reference of the CORBA object that it represents.

[1] In the first version of OTS, an object could take part in a transaction only if the object's interface inherited from a particular base interface. This approach has been deprecated by the OMG and a POA policy is now used instead.

[2] Information in a service context (Section 11.6) sent with a request indicates in which transaction, if any, the request is being executed.

If you use the NO_IMPLICIT_ACTIVATION policy then you must execute all three steps. However, if you use the IMPLICIT_ACTIVATION policy then you can optionally omit step 2 because step 3 will implicitly activate the servant if it is not already activated. So the benefit of IMPLICIT_ACTIVATION is that can optimize away one line of code associated with the creation of CORBA objects. This is a relatively minor benefit, and many people have religious feelings over whether the IMPLICIT_ACTIVATION policy is a good policy or one to be avoided. On the one hand, some people like being able to optimize away a line of code in several places within a server application. On the the hand, the steeper learning curve associated with yet-another policy value arguably outweighs this benefit.

Note that if you use IMPLICIT_ACTIVATION then you *must* combine it with SYSTEM_ID.

6.1.6 Proprietary Policy Values

The preceding subsections have discussed CORBA-compliant policy values that can be used when creating POAs. A CORBA product is allowed to define additional, proprietary policies. Some CORBA products do this in order to give programmers greater control over choosing the QoS offered by objects. You have to consult the documentation of a particular CORBA product to find out what, if any, proprietary policies it provides.

6.2 Creating Policy Objects and POAs

Policy objects were first introduced with the POA specification. The POA specification initially defined 7 types of policy and defined a separate operation for creating each of these 7 kinds of policy object. For example, you can create a ThreadPolicy object by invoking the following operation on an existing POA:

```
ThreadPolicy create_thread_policy(
                              in ThreadPolicyValue value);
```

The parameter to this operation is an enum value, and the operation creates a (subtype of) policy "wrapper" around it.

Having defined the POA specification, the OMG then realized that the concept of policy objects could be applied to other parts of CORBA too. However, having to add a new create-style operation to an existing interface for each new type of policy would create a versioning problem. Because of this, the OMG decided to define a "generic" API that could be used to create any kind of policy object. The work on defining this generic API was performed as part of CORBA Messaging (Chapter 16). Creation of transactional policy values (Section 6.1.4) is performed using this newer, generic API.

Overall, the CORBA APIs for creating POA policy objects and then using these to create POAs are quite verbose. The *Creation of POA Hierarchies Made Simple* chapter of the *CORBA Utilities* package [McH] discusses a utility class (available in C++ and Java) that dramatically reduces the amount of code required to create POAs.

Part III

Application Deployment

Chapter 7

Implementation Repository (IMR)

7.1 Introduction

The CORBA specification briefly describes the concept of an *implementation repository* (IMR). This term is not very intuitive so it can benefit from an explanation. *Implementation* is the CORBA terminology for "server application", and *repository* means a persistent storage area, such as a database or a file. This suggests that an *implementation repository* is a database/file that stores information about CORBA server applications. This is *almost* correct. An IMR usually also contains a CORBA server "wrapper" around the database/file, which makes it possible for CORBA applications to communicate with an IMR.

An IMR typically maintains the following information about each server application:

- A logical name that uniquely identifies a server, for example, "BankSrv" or "StockControlSrv".
- A command that the IMR can execute to (re)start the server process.
- Status information that indicates whether or not the server is currently running; if the server is currently running then the IMR also records the host and port on which the server process listens.

The CORBA specification provides only a *partial definition* of an IMR. In particular, CORBA states the high-level functionality that an IMR should provide, but does *not* state how this functionality should be implemented. Neither does the specification state how the IMR should be administered. The need for a partial specification is because much of the functionality of an IMR must be implemented and administered in a platform-specific manner. For example:

- An IMR should be capable of starting and stopping a server process. Different operating systems have different ways of starting and stopping processes.
- An IMR should record details of servers—such as the command used to launch a server—and whether or not the server is currently running. Some IMRs may store this information in a database. Other IMRs might record this information in a textual file. An IMR running on an embedded device might not have access to a file system or a database and hence might record server details in non-volatile RAM.

An IMR running on a mainframe would not only be *implemented* differently to an IMR running on a PC or an embedded device, it would also be *administered* differently. Put simply, one CORBA vendor's IMR running on one kind of computer might have a very different "look and feel" to another CORBA vendor's IMR running on a different kind of computer. This wide variation in IMRs is the reason why the CORBA specification contains only a high-level discussion about IMRs.

Section 7.2 illustrates the principles of an IMR through an example that is based on a hypothetical CORBA implementation. Then Section 7.3 outlines the IMRs of three different CORBA products. In this way, I illustrate how some of the details of an IMR vary from one product to another but the basic principles remain the same.

7.2 IMR Concepts

7.2.1 Registering a Server with the IMR

An IMR is typically implemented as a CORBA server "wrapper" around a database/file that persistently stores details of servers that have been registered with the IMR. CORBA products usually provide a command-line utility that can be used to register a server application with the IMR. Such a utility might be used as shown below (the "\" character indicates a line continuation):

```
reg_srv_with_imr BankSrv \
     -launch "/bin/bank_srv -ORBServerId BankSrv ..."
```

This utility takes details of a server application—such as a name ("BankSrv" in the above example) and its launch command—and somehow communicates this information to the IMR, which then persists the information in its database, as shown in Figure 7.1. Communication between the `reg_srv_with_imr` utility and the IMR is typically achieved by having this utility act as a CORBA client to the IMR. The IMR server process typically listens on a fixed port (shown as port 4000 in Figure 7.1). However, as mentioned in Section 7.1, the implementation details of the IMR are not standardized by CORBA, so the name of the `reg_srv_with_imr` utility, its command-line options, and how it communicates with the IMR vary from one CORBA product to another. Some CORBA products may provide a GUI administration program for registering servers instead of command-line utilities.

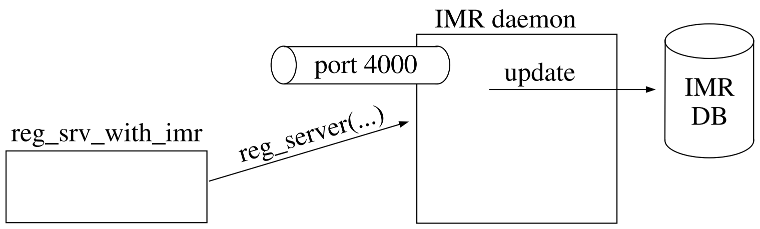


Figure 7.1: Registering a server with the IMR

The launch command specified by `reg_srv_with_imr` contains a pair of `-ORBServerId <name>` command-line arguments. Later, when the IMR launches the server, this pair of command-line arguments will be inspected by the server's call to `ORB_init()` (Section 3.2.3 on page 29) and this instructs the server process that it should identify itself to the IMR with the specified name. Note that the latest CORBA specification (3.0) defines the `-ORBServerId <name>` command-line option. Products that implement an older version of the CORBA specification might use a different command-line option to specify a unique name that identifies a server to the IMR.

7.2.2 Manually Running a Server

Having registered the server with the IMR, you can now run the server manually, for example:

```
/bin/bank_srv -ORBServerId BankSrv ...
```

Note that when you run the server manually, you use the `-ORBServerId BankSrv` command-line option that was specified in the launch command when previously registering the server with the IMR. Figure 7.2 illustrates what happens when the server runs.

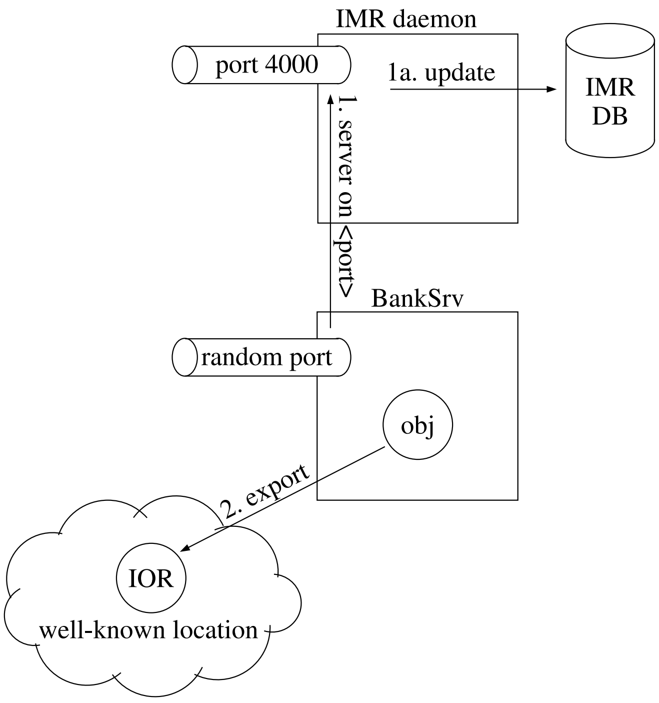


Figure 7.2: IMR interaction with manually launched server

By default, the server listens on any available port, which is often called a *random* port.[1] One of the CORBA APIs that is invoked during initialization of a server—typically `ORB_init()` or `create_POA()`—informs the IMR that the server specified by the `-ORBServerId` command-line option is running and on which port it is listening (step 1). The IMR updates its database with the supplied details (step 1a). This communication between the server and the IMR is an implementation detail of one of the CORBA APIs that is invoked during server initialization. This means that the server-IMR communication is transparent to server developers.

[1] Most/all CORBA products have a proprietary command-line option or entry in a configuration file that can be used to instruct a server process to listen on a specific port. However, the discussion in this chapter assumes that the server listens on a random port. A discussion about different ways to deploy CORBA servers can be found in Chapter 8.

An IMR *somehow* determines when a server terminates—so that the IMR can then record in its database the fact that the server is no longer running. CORBA has not standardized on the technique used by the IMR to determine when a server has terminated, but I briefly mention some of the techniques used by CORBA products. Some products have an undocumented object in the CORBA runtime system of servers and the IMR sends periodic "ping" messages to this to check if the server is still alive. Some other CORBA products open a socket/pipe connection between the IMR and a server; when the server terminates, the operating system automatically informs the IMR that the socket/pipe is being closed.

As part of a server's initialization, it is likely that the server will export an object reference (step 2) to a well-known location. Indeed, client applications will use an object reference to communicate with a server, so it is important to manually run a server once (so that it can export an object reference) before clients try to communicate with the server. The means by which the server exports the object reference—for example, to a file (Section 3.4.2 on page 34), the Naming Service (Chapter 4) or the Trading Service (Chapter 20)—are irrelevant to the present discussion.

An object reference (Chapter 10) contains the "contact details" for an object. When a server is deployed through an IMR then its exported (persistent) IORs specify how the object can be contacted through the IMR.[2] In particular, the (host, port, object-key) information within the IOR might be as follows:

host: <IMR's host>
port: 4000 (that is, the IMR's port)
object key: "BankSrv", <poa name>, <object id>

The *object key* in an IOR uniquely identifies an object within a server process. Within a server process, objects are grouped into collections called POAs (Section 5.5 on page 47), and one server may contain several POAs. Because of this, the object key contains the name of the POA in which the object resides, and also the *object id* that uniquely identifies the object within its POA. What surprises some people is that (in many CORBA products) the object key within an IOR contains the logical server name, for example "BankSrv". The reason for this is that (as shall be discussed in the next subsection) the IMR needs be able to examine the object key and determine in which server the object resides, so the IMR can redirect client requests to the appropriate server. Embedding the server's name inside the object key information of an IOR is the most obvious

[2] This discussion is applicable only to references for persistent objects. The distinction between *persistent* and *transient* objects is discussed in Sections 6.1.3 and 8.2.2.

way to link the object key to the corresponding server.[3] If (something akin to) the server name was *not* present in the object key then the IMR would know only to which POA an object belonged. In such a case, for the IMR to have the ability to redirect a client's request to the appropriate server would require that every persistent POA in every server be registered with the IMR, which would be a slight administrative burden. More importantly, it would make it impossible for several server applications that happened to have similarly-named POAs to be deployed through the same IMR, because there would be no way for the IMR to determine to which server an object belongs.[4]

7.2.3 Client Interaction with a Server and the IMR

Figure 7.3 shows what occurs when a client establishes communication with an object in a server. The client imports an IOR (step 1) from a well-known location. The first time the client tries to make a remote call to the object, it opens a socket connection to the host and port specified in the IOR. In our example, the host and port happen to be those for the server's IMR rather than for the server itself, but the client process is not aware of this. The client sends its request (step 2). In the header of each request is the *object key* information from the IOR. When the IMR receives the request, it realizes that the object key does not match one of its own objects so, the IMR extracts the server name ("BankSrv") from the object key and uses this to query its database (step 2a) for details of the server.

- If the database details indicate that the server is already running then the IMR obtains the server's host and port from the database, constructs a new IOR based on the (host, port, object key) details, and sends back a redirection message to the client that contains this new IOR (step 3).
- If the database details indicate that the server is *not* currently running then the IMR obtains the server's launch command from the database

[3] Variations on this scheme exist. For example, Orbix 6 does not embed a server name in an object key. However, a prefix (specified in a configuration file) may be attached to the name of a POA, and this prefix is embedded in the object key. Many people set the prefix to be the server's name, so the effect is as if the server's name was embedded in the object key.

[4] The CORBA specification explicitly requires a CORBA implementation to ensure that there is no namespace pollution of POA names across different server applications. Despite this, at least one implementation of CORBA neglects to embed a server name into object keys of persistent objects. This causes a scalability problem in the IMRs of such CORBA products, as it is not possible for a single IMR to deploy several servers that have similarly-named POAs. If you intend to deploy several servers through one IMR then check that your CORBA vendor's product does not suffer from this problem.

and uses this to (re)start the server (step 2b). The IMR then waits for the server to initialize itself and notify the IMR of its (probably random) port (step 2c). The IMR then updates its database with the server's "currently running" status and port number. It then constructs a new IOR based on the (host, port, object key) details, and sends back a redirection message to the client that contains this new IOR (step 3).

The redirection message (Section 11.4 on page 117) tells the CORBA runtime system in the client "The object you are looking for is not at this location, but here is a new IOR that tells you how to contact the object". The CORBA runtime system in the client then opens a new socket connection to the host and port specified in the "redirection IOR" and resends its request (step 4). This "resend" logic is handled by the CORBA runtime system in the client—it is transparent to the application-level code in the client.

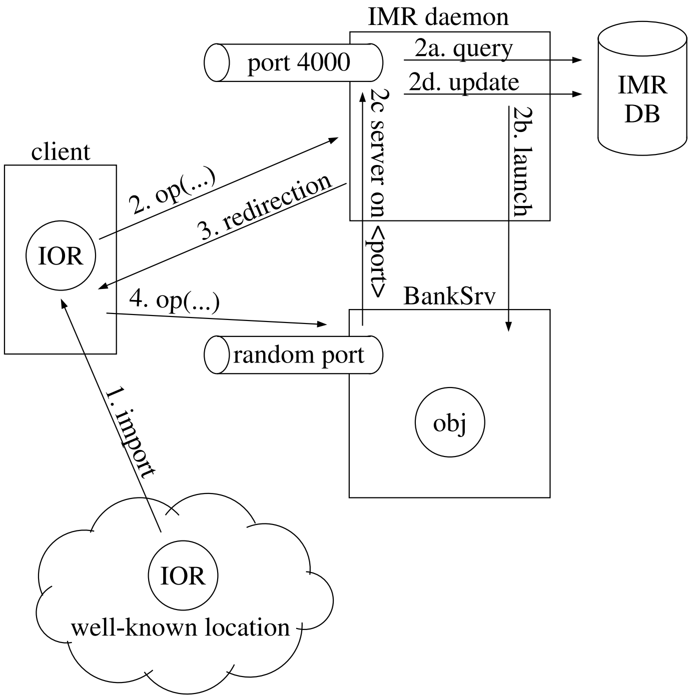


Figure 7.3: IMR interaction with client and automatically launched server

There are a few important points to note about the client initially talking to the IMR and then being redirected to the "real" server:

- The redirection occurs for just the *first invocation* from the client to an object. Subsequent requests from the client go directly to the "real" server. Because of this, any overheads associated with the redirection mechanism occur only during the initial connection establishment. Even for this redirection during the connection establishment, there is an optimization used in many CORBA products that avoids retransmission of large messages (Section 11.4 on page 117).

- The redirection mechanism means that CORBA servers do *not* have to be pre-started. Instead, servers that have been registered with an IMR can be started on an "as needed" basis by the IMR. Also, only the IMR needs to listen on a fixed port (which is usually specified in a configuration file)—servers that are deployed through the IMR can listen on random ports, which is often convenient.

- The redirection mechanism is defined as part of the CORBA specification. Because of this, use of an IMR does not affect interoperability between a client built with one CORBA product and a server built with a different CORBA product. However, the means of interaction between the server and its IMR varies from one CORBA product to another. A practical effect of this is that an Orbix IMR can launch only Orbix servers, a TAO IMR can launch only TAO servers and so on.

- Let us assume that a client is redirected via the IMR to the server and that the client successfully makes several invocations upon the object in the server. If the server later terminates, thus causing the client to lose its connection to the server, then the CORBA runtime system in the client will automatically revert to using the host and port in the original IOR. This means that the client will send its next request to the IMR, which gives the IMR a chance to restart the server and redirect the client to the newly relaunched server.

- Some CORBA products allow you to register a *replicated server* with the IMR. For example, there might be 5 servers that all share the same logical server name of "BankSrv". In such a CORBA product, the IMR could redirect some clients to one server replica, some more clients to another server replica and so on. This provides a per-client load-balancing mechanism, without the need for developers to explicitly add load-balancing logic to their applications. However, if you have a replicated server then it is your responsibility to ensure that the replicas do not suffer from cache inconsistency problems.

- It is possible to deploy a server *without* an IMR. In this case, an exported IOR contains the host and port of the server, so the client is *not* redirected through the IMR.

An IMR launches a server process by using operating system APIs to create a new process—for example, `CreateProcess()` on Windows or `fork()` and `exec()` on UNIX. On most operating system, such APIs can be used to create a new process on the *same* machine only, that is, they cannot be used to create a new process on a different machine. This suggests that you would need to have a separate IMR on each machine where you wish to launch server processes. This was a common feature of many early CORBA products. It resulted in some frustration for administrators of large CORBA deployments because they had to perform administration for several IMRs rather than than for just one IMR.

7.2.4 Distributed Implementation Repositories

Many of the modern CORBA products have split the IMR into two parts, as shown in Figure 7.4. One process (called the "IMR point of contact" in the diagram) listens on a fixed port, for example, port 4000, and this port number is embedded in IORs for objects in servers that are deployed through the IMR. Aside from providing the well-known port to be embedded in IORs, this process is also typically used to perform queries and updates to the IMR database. Whenever the IMR wants to launch a server process, it delegates this responsibility to a second process—called the "IMR launcher/monitor" in Figure 7.4—that listens on another fixed port.

The intention of splitting the IMR functionality into two components is that there may be several "IMR launcher/monitor" processes—typically one for each computer on which server processes can be started. This arrangement allows an IMR to span several computers. Obviously, the utility for registering a server with the IMR needs to specify the host on which a server process will run, as shown in the example below (the `"\"` character indicates a line continuation):

```
reg_srv_with_imr BankSrv -host pizza.acme.com \
     -launch "/bin/bank_srv -ORBServerId BankSrv ..."
```

One benefit of such a distributed IMR is that one centralized IMR database can be used to maintain the registration details of CORBA servers on several

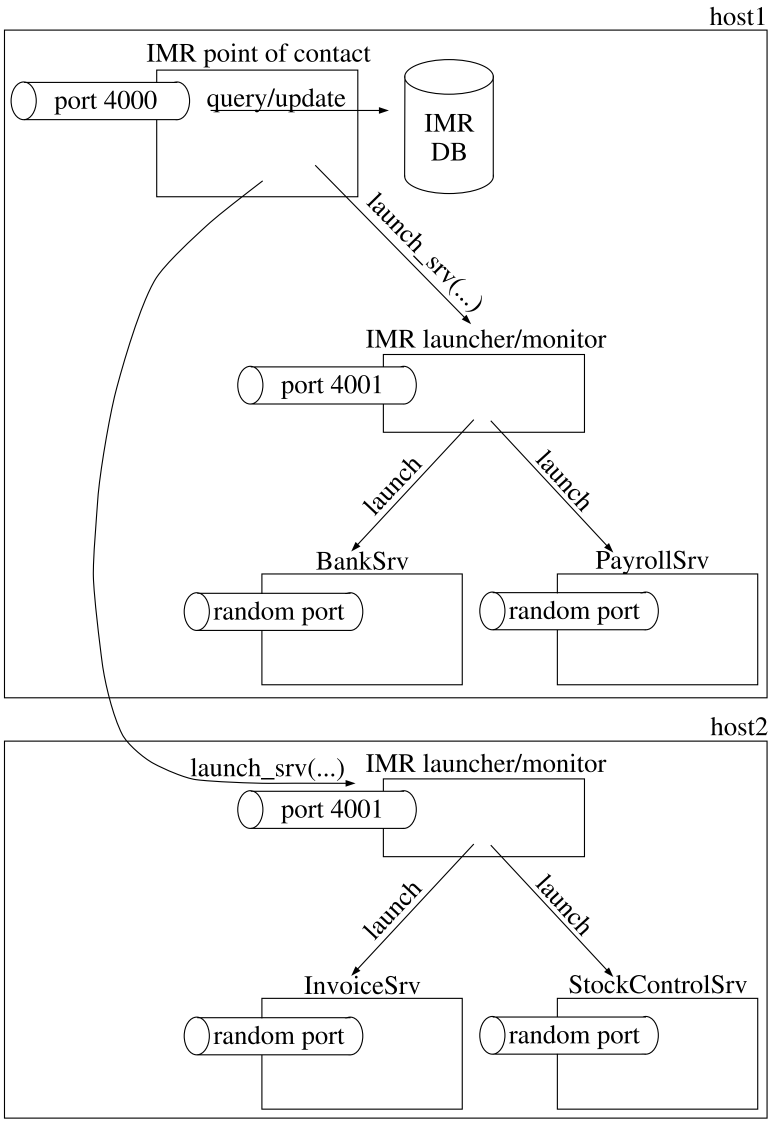


Figure 7.4: A distributed Implementation Repository

computers. Some organizations find this to be more convenient than having a separate IMR database for each computer.

Another benefit is that the host and port information in an IOR is always that of the “IMR point of contact” process, regardless of which host is used to run the server process. For example, the “BankSrv” might initially be run on one computer, but if this computer needs to undergo some maintenance work then the “BankSrv” can be stopped and re-registered/restarted on a different computer. In effect, the “BankSrv” can be migrated from one computer to another. This migration can take place *without invalidating* any previously exported IORs of the server, because the host information in the IORs remains that of the IMR’s point of contact process.

This distributed IMR architecture also offers a benefit for CORBA vendors. Much of the platform-specific code in an IMR is concerned with starting and stopping server processes. This platform-specific code can be encapsulated in the “IMR launcher/monitor” process, which enhances maintainability of the source code of the IMR.

Variations of the “distributed IMR” described above can be found in some modern CORBA products. The differences tend to concern the organization of the IMR database. For example:

- The “IMR point of contact” process might maintain a database that contains the *rarely-changing*, server registration details, such as a server’s logical name, launch command and host. Details of which servers are currently running, and the ports on which they are listening, are maintained in separate databases by each “IMR launcher/monitor” process.

- The “IMR point of contact” process might not maintain a database at all. Instead, all the information about the servers that run on a particular host is held in a database that is maintained by the “IMR launcher/monitor” process on that host.

These variations are relatively minor implementation details, and do not have any impact on the quality of service offered by a CORBA product.

CORBA products usually do not place any restriction on how many or how few IMRs you can create and whether different IMRs run on the *same* or *different* computers. Rather, the choice of the number of IMRs installed in an organization is typically due to pragmatic considerations. For example, it is common for each developer to have his or her own “private” IMR for day-to-day development work. Another IMR might be used for system testing, and yet another IMR might be used for deployed applications. An organization might find it convenient to have *several* “deployment” IMRs: perhaps a separate one for each branch or department in the organization, or perhaps one IMR for payroll applications and another IMR for stock-control applications. Obviously, if there are several IMRs running on the same computer then they need to listen on different ports. Typically, environment variables, configuration files or command-line arguments passed to a server are used to specify the host and port for the IMR that controls it.

7.3 Examples of Implementation Repositories

The following subsections give a brief overview of the IMRs in several CORBA products.

7.3.1 Orbix

The Orbix IMR is a distributed IMR, like that shown in Figure 7.4:

1. The Orbix name for the "IMR point of contact" is the *locator daemon* (`itlocator`).[5] Orbix calls this process a locator because it helps clients to locate (objects in) server processes. The Orbix name for the "IMR launcher/monitor" is the *node daemon* (`itnode_daemon`).[6] It is called this because there is one of these processes on each *node* (computer) where server processes can be deployed through the IMR.

2. The `itadmin` utility is a command-line-driven CORBA client that communicates with the locator daemon to query and update the IMR database. In effect, `itadmin` performs the work of the hypothetical `reg_srv_with_imr` utility discussed earlier in this chapter. A lot of Orbix administration is performed through various sub-commands of this utility. When a server application is registered with the IMR, one piece of registration information specifies the node daemon (computer) that should be used for launching the server.

3. In older versions of Orbix, the node daemon used to "ping" servers periodically to check if they were still alive. In more modern versions of Orbix, there is an open socket connection between the node daemon and a server process; when this socket connection closes, the node daemon realizes that a server has died.

[5] Many executables supplied with Orbix start with the prefix `"it"`. This prefix is an acronym for *IONA Technologies*, and is used to prevent namespace pollution of executables installed on a computer.

[6] The *node daemon* was called the `activation daemon` in older versions of Orbix.

When a server is being registered with the IMR, it can actually be registered as a *replicated* server that can be run on several different computers. The IMR keeps track of which server replicas are currently running, and can re-launch replicas that have crashed. The IMR can use a round-robin or random policy to redirect clients to server replicas. In this way, a per-client load-balancing mechanism is provided by the Orbix IMR without any need for extra coding in either client or server applications.

The Orbix IMR (and other other important pieces of infrastructure, such as the Naming Service) can be replicated on several computers. Doing this avoids single points of failure.

You create an Orbix IMR by running the `itconfigure` utility (which is discussed in the Orbix *Administrator's Guide*). You should note that *implementation repository* is the official CORBA terminology, but it is common for CORBA vendors to invent their own names for their IMRs. For example, the Orbix name for an IMR is a *location domain*. A location domain is simply the contents of the IMR database—that is, the details of all registered server applications—plus the locator daemon and its supporting node daemon(s).

7.3.2 Orbacus

The Orbacus IMR is a distributed IMR, like that shown in Figure 7.4:

1. The functionality of both the "IMR point of contact" and "IMR launcher/monitor" are embedded in a single executable called `imr`. In Orbacus terminology, the "IMR launcher/monitor" capability is referred to as the *object activation daemon* (OAD). By default, the `imr` executable enables *both* the "IMR point of contact" and the OAD functionality; the `-master` command-line option instructs it to enable just the "IMR point of contact" functionality, and the `-slave` command-line option instructs it to enable just the OAD functionality. If you want to have server applications running on several computers all controlled by a single IMR then start the `imr` executable on all the computers, but use the `-master` and `-slave` command-line options to ensure that you have "IMR point of contact" functionality on just one computer and OAD functionality on all the computers.

2. The `imradmin` utility is a command-line-driven CORBA client that communicates with the "IMR point of contact" process to query and update the IMR database. In effect, `imradmin` performs the work of the hypothetical `reg_srv_with_imr` utility discussed earlier in this

chapter. A lot of Orbacus administration is performed through various sub-commands of this utility.

When you run the Orbacus IMR for the first time, it creates and initializes its database. The Orbacus IMR cannot be replicated. Because of this, it is a single point of failure, which may be unacceptable in some organizations. Neither does Orbacus provide support for registering a *replicated server* with the IMR.

7.3.3 TAO

Up until and including version 1.2, the TAO IMR was a monolithic IMR. However, version 1.3 saw the start of work to turn the monolithic IMR into a distributed IMR, like that shown in Figure 7.4. This work is still ongoing. In particular, the functionality of the IMR has been split into two executables—`ImplRepo_Service` (the "IMR point of contact") and `ImR_Activator` (the "IMR launcher/monitor"). However, in versions 1.3 and 1.4, both of these processes must run on the same computer: support has not been added (yet) for one `ImplRepo_Service` to delegate to an `ImR_Activator` running on another computer.

The `tao_imr` utility is a command-line-driven CORBA client that communicates with the "IMR point of contact" process to query and update the IMR database. In effect, `tao_imr` performs the work of the hypothetical `reg_srv_with_imr` utility discussed earlier in this chapter. A lot of TAO administration is performed through various sub-commands of this utility.

When you run the TAO IMR for the first time, it creates and initializes its database. The TAO IMR cannot be replicated. Because of this, it is a single point of failure, which may be unacceptable in some organizations. Neither does TAO provide support for per-client load balancing by registering a *replicated server* with the IMR.

7.4 Comparison of Different IMRs

The discussion in Section 7.3 indicates some similarities and differences between the Orbix, Orbacus and TAO IMRs. It is worthwhile highlighting these, because there are likely to be similar likenesses and differences with the IMRs of other CORBA products:

- Each CORBA product tends to use its own terminology for the implementation repository. For example, Orbix uses the terms *locator dae-*

mon and *node daemon*, while Orbacus uses the terms *IMR* and *OAD*. Such differences in terminology are confusing for many people.

- Some early CORBA products implemented the IMR as a monolithic process. However, it is common for modern CORBA products to split the IMR functionality into two separate applications, as illustrated in Figure 7.4 on page 78.

- Some CORBA products have built-in *replication* infrastructure that allows the IMR to be replicated and also optionally allows server applications to be replicated. This replication capability can be used to avoid single points of failure in a deployed CORBA system, and may also provide per-client load balancing capabilities. Alternative replication mechanisms are discussed in Chapters 17 and 18.

Chapter 8

Deploying CORBA Applications

8.1 Deploying CORBA Clients

The details of deploying a CORBA client application vary slightly from one CORBA product to another. However, the principles are straightforward. In particular, you will need to do the following:

- Install a subset of the CORBA product (such as a configuration file, a `.jar` file or shared libraries/DLLs) on the deployment machine. Obviously, you will not need to install the development parts of the CORBA product, such as the IDL compiler. Some CORBA vendors provide a "install runtime only" and "install runtime and development tools" options for their product installation utility. If such an option is not available then it is usually easy to identify and copy the "runtime" subset from a full CORBA installation on a development machine to a deployment machine.

- Install the client executable on the deployment machine, along with required application-specific infrastructure, such as a configuration file and/or a directory for storing log files.

When you run the client application, you might need to provide it with some CORBA-related command-line arguments:

- You might need a command-line argument to tell the application where it can find the CORBA product's configuration file. Alternatively, this information might be communicated in a different manner, such as a system property for Java applications or through an environment variable for non-Java applications.

- If the diagnostics level specified in the configuration file for the CORBA product is not to your liking then you might specify a command-line option that specifies a different diagnostics level. Likewise, if the CORBA configuration file does not specify how to contact, say, the appropriate Naming Service then you might use `-ORBDefaultInitRef` or `-ORBInitRef` command-line arguments (discussed in Section 12.5 on page 127).

- Java provides an implementation of CORBA in its standard class library. If your application uses a different CORBA implementation then you may need to use command-line arguments to define some Java system properties that indicate which implementation of CORBA should be used.

If you run a client application with many command-line arguments then it becomes difficult to remember all the command-line arguments that must be typed. A simple way around this problem is to write a "wrapper" UNIX shell script or Windows batch file that runs the application with the appropriate command-line arguments.

The above requirements for deploying a CORBA client are so simple that the manuals for many CORBA products neglect to mention them and, ironically, this lack of documentation makes deployment seem more complex than it is. If your CORBA vendor does not provide documentation on deploying applications then sending an email to the vendor's technical support department should enable you to receive helpful guidance.

The deployment issues discussed in this section apply not just to clients, but also to server applications. However, there are some additional issues surrounding the deployment of servers, as I discuss in the next section.

8.2 Deploying CORBA Servers

Several concepts—POA policies (Chapter 6), the IMR (Chapter 7) and object references (Chapter 10)—interact with each other and influence how you can deploy a CORBA server. Most CORBA applications are deployed on a TCP/IP network, so the discussion in this section assumes use of TCP-based communication. Section 8.2.1 gives a brief overview of some issues in deploying a (non-CORBA) TCP server. Then Section 8.2.2 explains deployment options for a CORBA server through analogy with the deployment of a TCP server.

8.2.1 Overview of TCP Concepts

A TCP server listens on a *port* for incoming connections from clients. A TCP server has two options for specifying the port on which it listens:

1. The server can tell the operating system that it wants to listen on a *specific* (or *fixed*) port.

2. The server can tell the operating system that it wants to listen on port 0 (zero). The operating system understands this to mean that the server is happy to listen on *any* port, and so the operating system allocates an arbitrary port to the server. This arbitrary port is often called a *random*, *transient* (meaning temporary) or *ephemeral* (meaning short-lived) port.

If a TCP server is coded or configured to listen on a *fixed port* then it will listen on the *same* port whenever it is killed and restarted. Conversely, if a server is coded or configured to listen on a *random port* then it is likely to listen on a *different* port each time it is killed and restarted.

By convention, some TCP-based applications listen on well-known, fixed ports. For example, the default port for a web server is port 80. So when you type the URL `www.amazon.com` into your web browser, your web browser connects to port 80 on the specified host. This convention makes it easier to remember URLs, because you typically have to remember only the hostname on which a web server runs, rather than having to remember the hostname *and* port. For the same reason, other well-known TCP-based applications, such as POP3 mail servers (port 110), ftp servers (port 21), telnet servers (port 23) and so on also have well-known, fixed ports reserved for their use.

If a TCP server listens on a *random port* then it needs some way to communicate this random port to potential clients so that the client applications can connect to it. A simple way to do this is for the server to advertise its port by writing it to a text file, and then for clients to read this file. Of course, this solution requires that the server and clients have a shared file system, so this is not a geographically-scalable solution. A more complex, but more scalable, solution is for there to be an "advertisement server" that listens on a well-known, fixed port and maintains a list of *server-name* → *port* mappings. Whenever a "normal" server starts up and listens on a random port, it contacts the advertisement server to register its name and current port. Client applications connect to the advertisement server and request the port associated with the desired server name.

One benefit of the hypothetical “advertisement server” is that you need to allocate *only one fixed port*. This fixed port is for the advertisement server itself; all the other servers that advertise themselves through it can listen on random ports. This then reduces the administration overhead of having to choose fixed ports that are reserved for use by TCP servers.

8.2.2 Deployment Models for CORBA Servers

Most CORBA applications are deployed on a TCP/IP network, so CORBA servers are TCP servers that listen on a port. The CORBA concepts of the implementation repository (Chapter 7) and the Naming Service (Chapter 4) are *both* analogous to the TCP “advertisement server” described in Section 8.2.1. The Naming Service provides *name* → *IOR* mappings, while the IMR provides *IOR* → *IOR* mappings.[1] This functionality of the IMR seems strange, until you realize that the details that vary between the original and mapped IORs are the host and port embedded in the IOR. Because of this, the IMR provides a mapping from an IOR that contains the IMR’s well-known host and port to an IOR that contains the host and (usually random) port of a server process.

Most CORBA products can be configured (or programmed using proprietary APIs) to be deployed in some or all of the following four ways:

1. **The server listens on a *random* port and is deployed *without* an IMR**. This approach is ideal if a server has only *transient* objects, that is, all objects are in POAs that have the `TRANSIENT` policy. However, this deployment model is unsuitable for servers that have some objects in `PERSISTENT` POAs. This is because the port embedded in the IORs of the persistent objects is a random port and the server is likely to use a *different* random port if it is killed and restarted. In effect, this deployment model has the undesirable side effect of turning persistent IORs into transient IORs.

2. **The server listens on a *random* port and is deployed *with* an IMR**. In this case, the server embeds its own host and (random) port into IORs of *transient* objects, but it embeds the host and fixed port of the IMR into the IORs of *persistent* objects. When a client sends its first request to a persistent IOR, the request goes to the IMR. As discussed in Section 7.2 on page 70, the IMR uses the *object key* information in the header of the request to identify the intended server. The IMR (re)starts the server

[1] To be pedantic, the IMR provides *object-key* → *IOR* mappings. However, the object key used is part of an IOR that contained the IMR’s host and port, so conceptually the IMR provides *IOR* → *IOR* mappings.

process if it is not currently running and then redirects the client to the host and (random) port of the intended server process. This redirection through the IMR occurs only for the first request from the client to an IOR. Further requests from the client to the same IOR go directly to the server.

3. **The server listens on a *fixed* port and is deployed *without* an IMR**. In this case, the server's own host and port are embedded in both *transient* and *persistent* IORs. The embedding of a fixed port (as opposed to a random port) in *persistent* IORs ensures that the IORs are valid across restarts of the server.

4. **The server listens on a *fixed* port and is deployed *with* an IMR.** In this case, the server embeds its own host and (fixed) port into IORs of *transient* objects, but it embeds the host and fixed port of the IMR into the IORs of *persistent* objects. When a client sends its first request to a persistent IOR, the request goes to the IMR. As discussed in Section 7.2 on page 70, the IMR uses the *object key* information in the header of the request to identify the intended server. The IMR (re)starts the server process if it is not currently running and then redirects the client to the host and (fixed) port of the intended server process. This redirection through the IMR occurs only for the first request from the client to an IOR. Further requests from the client to the same IOR go directly to the server.

There are a few points worth noting about the above deployment options.

First, the server's own host and (random or fixed) port is *always* embedded in transient IORs. This means that transient objects *never* make use of the IMR. Some people make the mistake of assuming that if an IMR is running then servers automatically make use of it. However, if a server has *only* transient POAs then the server will just ignore the IMR. There is nothing preventing a CORBA vendor from designing a product in which transient IORs *do* contain the host and port of the IMR, but it would not bring any benefits. The author is not aware of any CORBA products that do this.

Second, there are some potential benefits to be had from deploying a server on a fixed port (deployment models 3 and 4 in the above list):

- Some organizations use a hardware *router* to load-balance client connections across a replicated server that is running on several machines. This approach to load balancing may require that server applications listen on fixed ports.

- The use of fixed ports is “firewall friendly”.[2]
- In general, the fewer components there are in a computer system, the more reliable the computer system will be. For this reason, some organizations prefer to deploy CORBA servers *without* an IMR. The use of fixed ports makes it feasible to deploy a server without an IMR, even if the server has persistent objects (deployment model 3 in the previous list).

  A counter argument to consider is that if the IMR is very stable then the use of an IMR can actually improve reliability because it can be used to automatically restart failed servers. Also, some CORBA products provide a fault-tolerance infrastructure through the IMR. This fault tolerance can be used to replicate the IMR and/or server processes, and so eliminate single points of failure within the computer system.

Third, some CORBA products make it easy to choose between all four of the deployment models discussed above. Other CORBA products provide *easy-to-use* support for a subset of the deployment models, but require use of proprietary APIs or complex configuration for the other deployment models. This is unfortunate because it often results in developers *hard-coding* logic into an application that, in effect, decides how a CORBA server will be deployed; it is preferable if deployment choices can be made at deployment time rather than having to be decided at development time. Furthermore, the use of proprietary APIs hinders portability. the *Creation of POA Hierarchies Made Simple* chapter of the *CORBA Utilities* package [McH] discusses how to work around portability issues associated with different server deployment models. Some CORBA products support only a *subset* of the deployment models. For example, the omniORB CORBA implementation does not contain an IMR and so cannot support deployment models 2 or 4.

Finally, as previously mentioned, some CORBA products provide a fault-tolerance infrastructure that is built into the IMR. In brief, when a client’s first request on a persistent IOR goes to the IMR, the IMR can redirect the client to one of several *replicas* of a server process. This type of fault-tolerance infrastructure (which can also provide per-client load balancing) works only for

[2] A firewall is a part of security infrastructure that prevents malicious hackers on the Internet from accessing the computers inside an organization’s network. A firewall blocks *most* TCP/IP traffic from outside the organization’s network, but can be configured to allow traffic on a set of fixed ports. In this way, the firewall can be configured to allow external access to a small number of services that are inside the organization’s network; typical examples include FTP, telnet and/or a web server. A typical requirement for exposing an internal service through a firewall is that the service listen on a fixed port .

persistent objects. It does not work for *transient* objects, because communication with transient objects never involves the IMR.

8.3 The Naming Service and the IMR

In Section 8.2.2 on page 88, I mentioned that the Naming Service provides *name* → *IOR* mappings, while the IMR provides *IOR* → *IOR* mappings. The fact that the Naming Service and the IMR *both* contain mapping functionality is often confusing to people new to CORBA. However, the purpose of the mapping provided by the Naming Service is different to the purpose of the mapping provided by the IMR.

- A stringified object reference (Section 3.4.2 on page 34) is in the form `"IOR:<hex-digits>"` so it is easy to store in a text file or database, but it is *not* easy for humans to remember. The Naming Service provides the ability to associate an easy-for-humans-to-remember name with an IOR. This makes it easier for humans to keep track of IORs. Of course, a person could decide to store stringified IORs in text files, with a separate file for each IOR. If the files have easy-to-remember names then this technique would serve a purpose similar to that of a Naming Service, except that it would not be geographically scalable, because it is rare for a file system to be accessible to computers that are geographically distant.

- Regardless of whether a client application obtains an IOR from a file or from the Naming Service, the host and port embedded in the IOR might be for the target server *or* for the server's IMR. If the host and port are for an IMR then the IMR redirects the client to the actual host and port of the desired server process. The purpose of this redirection is to allow some or all of the following benefits: (1) the IMR always listens on a fixed port, but servers deployed through the IMR can listen on random ports, so the administration overhead of allocating fixed ports for servers is reduced; (2) the IMR can (re)start a server process that is not running; and (3) some IMRs have built-in load-balancing and fault-tolerance infrastructure that allow them to redirect a client to one of several *replicas* of a server process.

Another point of confusion with the Naming Service and the IMR is that the Naming Service is itself a CORBA Server that is often deployed through the IMR. Because of this, there can be *repeated* redirection through the IMR when a client application obtains an IOR from the Naming Service and uses this to communicate with a persistent object in a server that is also also deployed through the IMR.

- When the client passes `"NameService"` as a parameter to `resolve_initial_references()` (discussed in Section 3.4.1 on page 33), it obtains an IOR for the Naming Service (this IOR is typically obtained from a configuration file). When the client invokes upon this IOR to communicate with the Naming Service, the client's first request will go to the IMR and the IMR will redirect the client to the Naming Service.
- Then when the client obtains an IOR from the Naming Service and invokes upon this IOR, the client will be redirected via the IMR to the desired server process.

Part IV

CORBA Infrastructure

Chapter 9

More Details on IDL

The basic concepts of IDL were discussed in Section 1.4. This chapter provides details on some of the more obscure or recent additions to IDL, and also discusses how to work around CORBA's lack of a versioning mechanism.

9.1 Pseudo-IDL, `local` and `native` types

In general, it is never possible to *completely* define a system in terms of itself, and CORBA is no exception. In particular, the OMG naturally decided to use IDL to define most of the APIs of CORBA, but there were some APIs that were impossible to express in legal IDL. For example, all IDL interfaces implicitly inherit from the base type `Object`. It is not possible to express the API of the `Object` interface in syntactically legal IDL because `Object` is a reserved keyword rather than an identifier. To work around this problem, the OMG used an informal notation called pseudo-IDL (PIDL) to define APIs of built-in types, such as `Object`. Pseudo-IDL is written as closely as possible to real IDL but a comment of the form `"// PIDL"` indicates that the API is not syntactically-valid IDL and hence cannot be run through an IDL compiler. PIDL was used extensively in early versions of the CORBA specification.

As CORBA matured, two new keywords—`local` and `native`—were introduced to IDL that made it possible to define a greater range of CORBA APIs in IDL. The introduction of these keywords reduced (but did not entirely eliminate) the need for pseudo-IDL.

The `local` keyword can appear in front of an `interface` definition. The effect is to define an interface that can be accessed only locally, that is, only within the same process. This keyword is not normally used by application-level developers. Rather, the intention of this keyword is to allow many local-access-only APIs of CORBA to be defined in IDL. For example, `DynAny` (Section 15.3), `Current` (Chapter 13), portable interceptors (Chapter 14), `Policy` (Section 16.1), the `ORB` itself and many of the types used for implementing server applications—`POA`, `POAManager`, `ServantManager`, `Policy`, and so on (Chapter 5)—are defined as `local interface` types.

The `native` keyword is used to indicate that a type is not an IDL type but rather is implemented in the *host language*, that is, C++/Java/Cobol or whatever programming language is used by developers to implement CORBA applications. A `native` type can be passed as a parameter only to `local` interfaces. The purpose of `native` declarations is to allow parts of CORBA to interact with the host language. For example, CORBA uses the terminology *servant* (Section 5.2) to refer to the host language object that represents a CORBA object; there is a corresponding declaration:

```
native Servant;
```

The POA infrastructure (Section 5.5) defines several `local` interfaces with operations that take `Servant` parameters.

9.2 Objects By Value (OBV)

CORBA became popular a few years before Java/J2EE became popular. When J2EE was announced, it was recognized that in some ways CORBA and J2EE complemented each other but that in other ways they were competitors. There was a lot of speculation about whether one of these apparently-competing technologies would “beat” the other. There was one particular capability present in Java that was missing from CORBA and some people within the OMG felt that CORBA should be enhanced to provide a similar capability. This feature was to become known as *objects by value* (OBV). The driving force behind OBV was not good technical innovation but rather was political and marketing pressure to defend CORBA from the perceived threat of J2EE. Quite predictably, the resulting OBV specification was (and remains) somewhat controversial because it has some technical rough edges and provides capabilities that can be misused easily.

9.2.1 The Java Equivalent of OBV

Before discussing what OBV is from a technical perspective, it is useful to discuss the Java-based technologies that it tries to emulate. Java has built-in support for *serializing* an object, that is, converting the in-memory representation of an object into a binary buffer and then later converting from the binary buffer back into an in-memory representation. This serialization capability of Java provides a convenient way to persist Java objects, by storing the binary buffer representation in, say, a file or database. It also makes it possible to serialize a Java object into a binary buffer, transmit this buffer across a socket connection to another Java process, and for the receiving process to re-create the Java object in its own address space. In effect, a Java object can be transmitted “by value” from one Java process to another Java process. Actually, the mechanism discussed so far serializes and transmits only the state (instance variables or fields) of the Java object. An object is both *state* and the *operations* that manipulate that state. However, Java is also capable of transmitting the bytecode that implements the operations of an object. In this way, Java is able to transmit *both* the state and operations of an object. Transmitting the bytecode of an object is important because the receiving Java process might not have local access to the relevant bytecode. For example, the receiving Java process might be expecting an object of type `Graphic` but might actually receive a *subtype* of `Graphic` called `Circle` for which it does not have access to the relevant bytecode.

An obvious question about this Java capability is: Is this really useful? A typical usage for this is the following interaction between a client application and server application:

1. The client invokes an operation on the server. The return value of the operation is an object (state and, if required, bytecode).

2. The client invokes many fine-grained operations upon its local (copy of the) object.

3. When the client has finished making its updates to the local object, it then makes a remote call to the server application, and passes the (updated) object as a parameter.

The main benefits offered by this usage scenario are as follows:

- Passing objects (by value) between processes can provide a significant optimization. In step 2 above, having the client make fine-grained operation calls upon a *local* object is much faster than making similar calls upon a *remote* object. This is because a remote call typically involves a few milliseconds of network latency; the local calls do not have this overhead.

- The same optimization could be achieved by passing just *data*—for example, structs and sequences—between the client and server. However, this would expose the client to the low-level data directly. It is better for these low-level implementation details to be hidden within operations, particularly if the bytecode of these operations can be transmitted automatically from the server to the client.

9.2.2 Objects By Value in CORBA

A CORBA `interface` has operations but no state variables. In contrast to this, a CORBA `struct` has state variables (fields) but no operations. A new construct, called a `valuetype`, has been introduced to IDL. A `valuetype` looks like a cross between an `interface` and a `struct` because it has both operations and state variables. Some examples of `valuetype` declarations are shown in Figure 9.1.

```
valuetype Date {
  short   year;
  short   month;
  short   date;
  void    next_day();
  void    previous_day();
};
valuetype OptionalString string;
```

Figure 9.1: Example of IDL `valuetype` definitions

When a `valuetype` is passed as a parameter, its state variables are transmitted. Operations invoked upon a `valuetype` are always invoked on the *local* (copy of the) `valuetype`. In general, it is not feasible for the code that implements the bodies of operations to be transmitted, because the client application and server application may be implemented with different programming languages and/or on different CPU types. For this reason, the client and server application developers must write and maintain separate implementations of the `valuetype`'s operations. This requirement introduces a big problem: there is no guarantee that the server-side implementation of the `valuetype` operations is semantically equivalent to the client-side implementations of the same operations. When developing the first version of the client and server applications, developers on both sides will probably take great care to ensure that the client-side and server-side operations have equivalent semantics. However, during ongoing maintenance of the applications, it is quite possible that a change in semantics (perhaps in the form of a bug-fix or a buggy optimization) will be introduced into the server-side implementation of the operations, but that a similar change will not be made in the client-side operations. During the lifetime of a project, there may be one server implemented in, say, C++, and several different kinds of clients, each of which is implemented in different languages, such as Java, Ada and Cobol. Maintaining semantic equivalence of operations implemented in multiple programming languages and used in multiple applications can quickly become a significant burden.

Opponents of `valuetype` point out that distributed applications have been successfully developed and deployed for several decades *without* the use of `valuetype` (or something similar). Because of this, `valuetype` is *not* an essential feature of CORBA and can (and probably should) be ignored.

Having discussed one of the main drawbacks of `valuetype`, I now briefly list some of the extra capabilities that they provide.

```
valuetype Base {
  long    some_data;
};
valuetype Derived : Base {
  long    more_data;
};
```

Figure 9.2: Inheritance of `valuetype` definitions

First, if you declare a `valuetype` that contains state variables but no operations then it is semantically similar to a `struct` but has one additional benefit: you can have *single* inheritance of such `valuetype`s.[1] This is shown in Figure 9.2. In effect, you can think of a `valuetype` as being a *`struct` with inheritance*.

Second, a `valuetype` is always passed in a manner similar to a C++ pointer (a *reference* in Java). For example, if a field within a `valuetype` is another `valuetype` then this field is a *pointer* to the embedded `valuetype`. It *is* legal to use a null pointer where a `valuetype` is expected. By introducing pointer semantics to IDL, `valuetype`s allow you to model cyclic graph structures. Also, if you declare a `valuetype` that has just one field, say, a `string`, then this allows you to pass a "normal" string (embedded inside a `valuetype`) *or* a null pointer as a parameter. In effect, this is a convenient

[1] You can use *multiple* inheritance if `valuetype`s have operations but no state variables.

way to pass an “optional value” as a parameter.[2] The designers of OBV felt that the “optional value” usage of `valuetype` would be useful often enough that they invented some syntactic sugar for it. This syntactic sugar is illustrated by the `OptionalString` declaration in Figure 9.1. This syntactic sugar format is usually referred to as a *valuebox*.

9.3 Versioning

CORBA does *not* have a mechanism for versioning IDL definitions. Unfortunately, there is widespread confusion about this. The confusion arises because CORBA 1 defined a syntactic place-holder for a possible future versioning mechanism. The syntactic place-holder was called `#pragma version` and it was intended to be used in IDL files as shown in the example below:

```
#pragma version "1.2"
```

The `"1.2"` was intended to indicate a version number for the following IDL construct.

A versioning mechanism requires more than just a syntactic construct: it requires additional supporting infrastructure. However, the OMG has never defined the necessary supporting infrastructure to make `#pragma version` useful. Because of this, `#pragma version` “is a historical relic and is ignored by the ORB” [HV99, Section 4.19.3]. Unfortunately, the continued presence of this syntactic place-holder leads many people to incorrectly assume that CORBA has a versioning mechanism and they then waste time and effort trying to make use of it.

Given that CORBA does not have a built-in versioning mechanism, the question then arises of whether there is any way to fake a versioning mechanism. Two (imperfect) suggestions are discussed below.

One approach is to (mis)use inheritance as a versioning mechanism. For example, let us assume that you have an existing IDL interface called `Account` and you want to create a new version that has additional functionality. You can do this by defining a new interface called, say, `Account2` that inherits from `Account` and adds new operations.[3] This approach works if the new version

[2] IDL provides two other ways to pass an “optional value”. One way is to use a `sequence` of length 1 to hold the value and a `sequence` of length 0 to indicate “no value”. The other way is to use a `union`. The union’s discriminant (case label) can indicate whether or not the intended value is provided.

[3] The OMG used this approach with the Naming Service. The first version of the Naming Service defined an interface called `NamingContext` (in module `CosNaming`). “Version 2” of the Naming Service was defined in an interface called `NamingContextExt` that inherited from `NamingContext` and added some new operations.

of the interface only *adds* new functionality; it will not work if you need to *delete* or *modify* the signatures of existing operations. Also, this approach will result in a deep inheritance hierarchy if you use it to define several versions of an interface.

```
module Finance {
    ...
    interface Account {
        ...
    };
};
```

```
module Finance2 {
    ...
    interface Account {
        ...
    };
};
```

Figure 9.3: A copy-and-modify approach to versioning

Another approach to faking versioning is to define a new, unrelated interface. This is illustrated in Figure 9.3. The original IDL types for an application are defined in module `Finance` (shown in the box on the left). When a new version of the application is being developed, a copy is made of the IDL file and the module is renamed from `Finance` to `Finance2`.[4] Then the types within `Finance2` can be modified without restriction. As far as humans are concerned, `Finance2::Account` is "similar to" `Finance::Account` and so they can think of them as being different versions of the same interface. However, this "versioning" is entirely within the minds of humans. As far as CORBA is concerned, the two interfaces are semantically unrelated.

In general, it is good coding practice to define all types inside modules, as this reduces namespace pollution. The use of modules offers another benefit for versioning: it is much more convenient to embed the version number in the name of one module rather than embed the version number in the names of the, possibly numerous, data-types defined within the module. Also, when updating version 1 of the source-code of an application to produce version 2, a single global-search-and-replace within source-code files for the name of the module is easy to perform.

It should be noted that the lack of a built-in versioning mechanism is not unique to CORBA. Most middleware systems lack a versioning mechanism, as do most programming languages.

[4] The naming scheme would be more consistent if the original module had been called `Finance1` rather than `Finance`. However, such foresight is rarely found in reality and so version numbers usually are *not* embedded in the name of the original module.

9.4 Repository IDs

A *repository id* is a slightly mangled form of the fully-scoped name of an entry in an IDL file. For example, the repository id of `Finance::Account` is `"IDL:Finance/Account:1.0"`. In general, all occurrences of `"::"` in the fully-scoped name are replaced with `"/"`. The resulting string is then prefixed with `"IDL:"` and suffixed with `"1.0"`.[5]

IDL allows a `#pragma prefix "..."` construct to be used in IDL files. An example is shown below:

```
#pragma prefix "acme.com"
module Finance {
    interface Account { ... };
};
```

If a `#pragma prefix` directive is used in an IDL file then the specified prefix (`"acme.com"` in the above example) is embedded into the repository ids for all types in that file. For example, the repository id for type `Finance::Account` is `"IDL:acme.com/Finance/Account:1.0"`.

Repository ids are a form of runtime type information. Most CORBA applications rely on compile-time type checking so repository ids are not used very frequently. However, most CORBA developers do encounter repository ids occasionally, so it is useful to know what they are and what their intended usage is. Here is an incomplete list of when repository ids are used:

- As mentioned in Section 1.5 on page 14, an interoperable object reference (IOR) contains the "contact details" for an object. However, the IOR may also contain the repository id for the object's type.[6] The presence of a repository id in an IOR is a very useful debugging aid. For example, most CORBA implementations provide a command-line utility that can print out the repository id and contact details contained inside an IOR (Section 3.4.2 on page 34). It is common for people to use such utilities to help them diagnose problems when developing and deploying a client-server system. Often a problem is due to a client being given the wrong kind of IOR, for example, an IOR for an `Employee` object

[5] The `"1.0"` suffix denotes the version number. This version number was incorporated into the repository id to support the (later abandoned) versioning mechanism discussed in Section 9.3. A `#pragma version` directive could be used to change the version number embedded in a repository id, but there is no point in doing this because, as explained in Section 9.3, CORBA does not offer a proper versioning mechanism.

[6] The CORBA specification states that an IOR is not obliged to contain a repository id; an IOR may contain an empty string instead. However, most CORBA implementations embed a repository id into IORs.

rather than an IOR for a `Finance::Account` object. Being able to see the repository id embedded in an IOR often helps people to diagnose these kinds of problems easily.

- When an operation in a server application throws an exception, the exception's repository id is marshaled (serialized) first, followed by the fields within the exception. In this way, the CORBA runtime system in the client application can use the repository id to determine the exception's type; this then tells the CORBA runtime system how it should unmarshal (deserialize) the fields of the exception. A discussion about marshaling can be found in Section 11.2 on page 113.
- Repository ids are used in programs that utilize meta-information (Chapter 15).

I mentioned earlier that use of a `#pragma prefix "..."` directive causes the specified prefix to be embedded in the repository ids for all types in that IDL file. Use of a `#pragma prefix` directive does *not* affect the public API that is generated by an IDL compiler. For example, it does not affect the public API of the C++ or Java types generated by an IDL compiler. However, it does affect the *implementation* of the generated operations that return the repository ids of IDL types. This is because the string returned by these operations must embed the string used in a `#pragma prefix` directive.

Sometimes people wonder what purpose is served by placing `#pragma prefix` directives in IDL files. The answer can be illustrated with an example. Let us assume that the Bank of America defines a module called `Finance` that contains an `Account` interface. Without use of a `#pragma prefix` directive, the repository id of this type is `"IDL:Finance/Account:1.0"`. The problem is that the Bank of America might not be the only organization in the world to define an interface called `Finance::Account`. If another organization defines an interface with the same name (and presumably with operations that have different signatures) then it might be difficult to diagnose problems if a client application that was written to communicate with a Bank of America server accidentally gets an IOR for a `Finance::Account` object in a different organization. To avoid such problems, developers are encouraged to put a `#pragma prefix` into all their IDL files. The prefix string should contain something that is unique to the developer's organization. Typically, an Internet domain name is used, as this is a globally unique identifier. For example, IDL files written by developers in the Bank of America might contain the following:

```
#pragma prefix "bankofamerica.com"
```

Now, an IOR for the `Finance::Account` interface defined in such a file is:

```
IDL:bankofamerica.com/Finance/Account:1.0
```

If a client application is developed with the Bank of America IDL files then the CORBA runtime system in this client application will throw an exception if it is mistakenly given an IOR that contains an inappropriate repository id such as `"IDL:Finance/Account:1.0"` or `"IDL:bankofengland.co.uk/Finance/Account:1.0"`.

9.5 Miscellaneous New Keywords

The following new keywords have been added to IDL in recent years.

The `typeprefix` keyword serves a purpose similar to the `#pragma prefix` construct discussed in Section 9.4. An example of its use is shown below:

```
module CosNaming {
  typeprefix CosNaming "omg.org";
  ...
};
```

In this example, the `typeprefix` command causes the `"omg.org"` prefix to be embedded in the repository ids for the `CosNaming` module and all types declared inside it.

The `import` keyword serves a similar purpose to a `#include` directive that was discussed in Section 1.4.1 on page 8. An example of its use is shown below:

```
import CosNaming;
```

As shown in this example, `import` is typically followed by the name of a module. It has the effect of including the IDL file that contains that module.

An operation can have a `raises` clause, which means that it can raise user-defined exceptions. In contrast, for many years an `attribute` (which is, in essence, syntactic sugar for a pair of get- and set-style operations) could *not* have a `raises` clause, so it could not raise exceptions. The new keywords `getraises` and `setraises` have been added to IDL to specify what exceptions can be raised by the get- and set-style operations for which an `attribute` is syntactic sugar.

```
exception X { ... };
exception Y { ... };
exception Z { ... };
interface Foo {
  attribute string name getraises(X, Y) setraises(Y, Z);
};
```

Chapter 10

Interoperable Object Reference (IOR)

10.1 Introduction

An *interoperable object reference* (IOR) is the “contact details” that a client application uses to communicate with a CORBA object. The *interoperable* in *interoperable object reference* comes about because an IOR works (or interoperates) across different implementations of CORBA. For example, an IOR for an object in an Orbix server can be used by a client that is implemented with, say, Orbacus, Visibroker, TAO, omniORB or JacORB.

Many people are content to know that object references are “contact details” for CORBA objects, and they do not seek any additional information about object references. If you are such a person then you can skip this chapter. However, CORBA IORs are remarkably flexible, and this flexibility is fundamental to the implementation of several other components of CORBA, including implementation repositories (Chapter 7), messaging (Chapter 16) and transactions (Chapter 21). Because of this, having an understanding of the information contained *inside* object references makes it much easier to understand other parts of CORBA.

10.2 IDL Definition of an IOR

CORBA uses IDL to define some of its low-level APIs. Because of this, it is not surprising that the details of an IOR are defined as an IDL `struct`, as is shown in Figure 10.1. The contents of this `struct` can be explained easily by analogy with a business card, which—because it also contains "contact details"—is the business equivalent of an IOR. A sample business card is shown in Figure 10.2.

```
module IOP {
    typedef unsigned long ProfileId;
    const ProfileId TAG_INTERNET_IOP = 0;
    struct TaggedProfile {
        ProfileId        tag;
        sequence<octet>  profile_data;
    };
    struct IOR {
        string                  type_id;
        sequence<TaggedProfile>  profiles;
    };
    ...
};
```

Figure 10.1: IDL definition of an IOR (interoperable object reference)

John Doe

Senior Consultant

Telephone:	555-493-687
Fax:	555-493-001
Email:	john.doe@acme.com

Figure 10.2: Example of a business card

The `type_id` field of the IOR struct contains the object's repository id (Section 9.4 on page 102). The equivalent of the `type_id` on a business card is the job title, such, as *senior consultant*, *sales person*, *director* or *software engineer*.

The other field in an IOR is a sequence of `TaggedProfile`. The name of this type is not very intuitive, so it deserves some explanation. *Profile* is the contact details expressed as binary data (a `sequence<octet>`)[1] and a *tag* tells us how to interpret that binary data. An analogy of a *profile* in the business card is the number 555–493–687. To make sense of this number, you

[1] An `octet` is the built-in IDL type that denotes a byte of raw data.

need to look at its *tag* to find out that it is a telephone number rather than, say, a fax number. The OMG-defined tag value `TAG_INTERNET_IOP` specifies that the binary data provides IIOP[2] contact details, and another set of IDL type definitions (which are discussed in Section 10.2.2) define the layout of the binary data. The tagged-profile approach to specifying contact details provides some future-proofing. For example, if the popularity of TCP/IP declines in the future when another transport mechanism is invented then the OMG can define a new tagged-profile format to support that transport mechanism. Furthermore, CORBA vendors can define their own proprietary tagged profiles. For example, CORBA has not yet standardized on a way of communicating using shared memory or wireless networks, but several CORBA vendors have defined their own proprietary protocols for these purposes.

An IOR contains a *sequence* of tagged profiles. This means that an IOR can list several sets of contact details. For example, perhaps one set of contact details is for a proprietary shared-memory protocol and another is for the CORBA-compliant IIOP protocol. When a client that is built with the same CORBA vendor's product uses the IOR, it will choose one of the profiles (probably the first one it finds). When a client that is built with a *different* CORBA vendor's product tries to use the IOR, the client ignores the shared-memory profile because it does not recognize the proprietary tag value and instead uses the IIOP profile. An IOR that contains several profiles—for different transport mechanisms—is analogous to a business card that lists several sets of contact details, such as a telephone number, fax number, email address and postal address.

An IOR could contain several profiles, all of which use the same underlying transport mechanism. For example, an IOR could contain 3 or 4 different sets of IIOP contact details. This could be used to provide load balancing or fault tolerance. It is important to note that CORBA *allows* this flexibility but *does not require* that a CORBA vendor exploit it. Many CORBA products embed just a single set of contact details in IORs. A few CORBA products provide load balancing and fault tolerant capabilities that are based on IORs containing several profiles. Whether a server exports single-profile or multi-profile IORs does *not* affect interoperability with clients.

10.2.1 Space Optimization

There is one interesting way in which the analogy between an IOR and a business card falls down. A business card typically has the person's *name*—in

[2] IIOP (Internet Inter-ORB Protocol) is the CORBA protocol for communication on TCP/IP networks. CORBA protocols are discussed in Chapter 11.

other words, his or her (hopefully unique) identity—written on it exactly once. A CORBA object has an identity that is unique within a server process. This identity is called its *object key*. The object key information is embedded inside the tagged profile information. Because of this, if an IOR contains several tagged profiles then the object key information *may* be repeated several times—once inside each and every tagged profile. However, the `TaggedComponent` mechanism (discussed in Section 10.2.3) provides an optional space optimization that allows a single profile to contain one object key and *several* sets of (host, port) tuples.

10.2.2 IIOP Contact Details

One issue that has not been discussed yet is this: what do the contact details inside an IOR look like? In practice, most IORs contain a single *profile* (set of contact details) for the IIOP protocol. Figure 10.3 shows the information stored in an IIOP profile. The GIOP marshaling rules (Section 11.2 on page 113) are used to marshal an IIOP profile into a binary buffer. This binary buffer is then stored in the `sequence<octet>` data of a `TaggedProfile`.[3]

As can be seen, an IIOP profile contains the `host` and `port` at which a client can connect to the server process. A server process might contain *several* objects, so the `object_key` is used to identify exactly one object within the server process. The reason for the presence of the `TaggedComponent` entries will be discussed in Section 10.2.3.

When a client wants to make a call upon an object by using its IOR, the client application opens a TCP/IP socket connection to the host and port specified in the IOR. It then sends the request. The header of the request specifies the object key of the target object and the name of the operation being invoked. The CORBA runtime system within the server application uses this header information to find the correct *servant* (Section 5.2 on page 45) and to call the appropriate operation upon that servant.

10.2.3 The use of `TaggedComponent` entries in an IOR

An IIOP profile (Figure 10.3) contains a `sequence<TaggedComponent>`. A `TaggedComponent` consists of binary data (a `sequence<octet>`), and a *tag* tells us how to interpret that binary data. The OMG has defined over 30 different types of `TaggedComponent` so far, which are used to specify additional information about an object reference, such as:

[3] CORBA uses the term *encapsulation* to refer to marshaling one type into a binary buffer and then storing this in a `sequence<octet>`.

```
module IOP {
    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId      tag;
        sequence<octet>  component_data;
    };
    ...
};
module IIOP {
    struct Version {
        octet  major;
        octet  minor;
    };
    struct ProfileBody_1_1 { // also used for 1.2 and 1.3
        Version                         iiop_version;
        string                          host;
        unsigned short                  port;
        sequence<octet>                 object_key;
        // added in 1.1; unchanged for 1.2 and 1.3
        sequence<IOP::TaggedComponent>  components;
    };
    ...
};
```

Figure 10.3: IDL definition of an IIOP Profile in an IOR

- An alternative (host, port) tuple that can be used to communicate with the object. This is a space-saving optimization, as an IOR with one profile that contains multiple (host, port) tuples is more compact than an IOR with multiple profiles.

- The code sets (character sets) that can be used to send `char`/`string` or `wchar`/`wstring` parameters (Section 11.7 on page 120).

- Information used to implement security policies.

- Whether or not the object can participate in a distributed transaction (Chapter 21).

- Details of message routers that can be used for time-independent invocations (Section 16.3 on page 149).

10.3 Proxy

The `IOR` structure (Figure 10.1 on page 106) contains the contact details for a CORBA object. However, it does not specify the signatures for operations that may be invoked upon the object. A *proxy* is a C++/Java/whatever-language-you-are-using object “wrapper” around the IOR and it provides the signatures of the IDL operations for the remote CORBA object. When a client application invokes an operation on a proxy, the proxy *marshals* (Section 11.2 on page 113) the `in`/`inout` parameters into a binary buffer and then transmits this request using the contact details specified in the IOR. The proxy waits to receive a reply message, unmarshals the `out`/`inout` parameters and return value and then returns these to the calling code.

The previous paragraph explained that a proxy is a wrapper around an IOR. However, the distinction between an IOR and a *wrapper* around an IOR is often irrelevant. Because of this, the term *proxy* is often used as a synonym for an IOR.

Chapter 11

On-the-wire Protocols

11.1 GIOP, IIOP and the Protocol Stack

Many people care that a CORBA client *can* communicate with a CORBA server, but they do not care *how* the communication happens. However, some people *do* care about (the principles that underlie) the low-level details of CORBA's communication infrastructure. There may be several reasons for their interest, for example:

- Some people want to understand how efficient CORBA's communication mechanism is so that they can compare it to the efficiency of another distributed middleware technology, for example, IBM MQ Series or SOAP.

- Most organizations make use of TCP/IP networks. However, some organizations use other transport mechanisms. People from such organizations may be curious if CORBA is flexible enough to be used with (or adapted for use with) their preferred networking technology.

The OMG decided to provide an efficient and flexible inter-application communication mechanism by designing a multi-layer protocol stack, as shown in Figure 11.1. All CORBA on-the-wire protocols are designed around some common guidelines. This makes it relatively easy to build bridges between the different protocols. One of the common guidelines includes the definition of an IOR (Chapter 10), which holds the "contact details" of an object, regardless of which on-the-wire protocol is being used.

An *Environment-Specific Inter-ORB Protocol* (ESIOP) is an on-the-wire protocol that is optimized for a specific environment. For example:

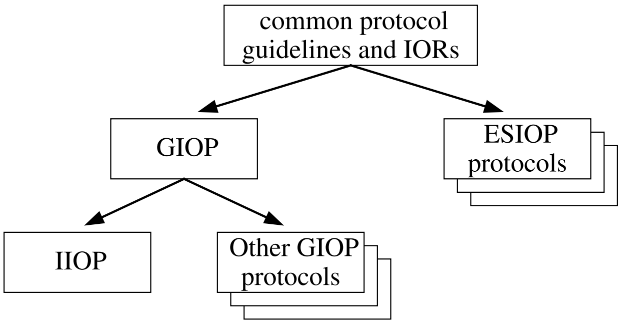


Figure 11.1: The hierarchy of CORBA protocols

- In the early days of CORBA, there were still a lot of organizations using DCE RPC (an older middleware technology). An ESIOP for DCE was defined, which made it possible for CORBA applications to interoperate efficiently with existing DCE applications.

- The Orbix product from IONA Technologies has been ported to IBM mainframes. In the mainframe environment, Orbix uses a proprietary ESIOP that allows Orbix to communicate efficiently with various components that are native to the mainframe operating system.

Aside from ESIOPs, all CORBA protocols are based on the *General Inter-ORB Protocol* (GIOP). This protocol defines the different message types (such as request and reply messages) that can be exchanged between client and server applications and also specifies a binary format for the on-the-wire representation of IDL types (`boolean`, `long`, `string`, `enum`, `struct`, `sequence`, `union` and so on).

The main thing that GIOP does *not* specify is the actual networking technology that is used to transmit messages between clients and servers. For example, GIOP does not specify if messages should be transmitted over TCP/IP, X25, ATM or some other transport. Instead, the choice of transport mechanism is decided in a *specialization* of GIOP. The most well-known GIOP specialization is the *Internet Inter-ORB Protocol* (IIOP), which is for use on TCP/IP networks. All CORBA products are obliged to support IIOP, but they may optionally support other GIOP-based protocols or ESIOPs too. An IOR contains the contact details for all the protocols that clients can use to communicate with an object in a server.

11.2 Marshaling IDL Types

The act of placing IDL types into a binary buffer (in preparation for transmission) is called *marshaling*. Conversely, extracting IDL types from a binary buffer is called *unmarshaling*.[1] The marshaling rules used by CORBA are called Common Data Representation (CDR). CDR consists of several rules, as I now discuss.

One CDR rule is that data is marshaled using the native endian format of the computer that is sending a message. A flag in the header of GIOP messages specifies whether the message is in *big-endian* or *little-endian* format.[2] If the receiving computer uses the same endian format then the data can be unmarshaled directly; otherwise byte-swapping is performed when unmarshaling numeric values. In effect, this CDR rule provides an optimization for the common case where a client and server run on machines with the same endian format.

Another CDR rule states how many bytes of memory basic data-types use and their memory-alignment requirements:

- `char`, `octet` and `boolean` occupy 1 byte of memory and have no alignment requirements.
- `short` and `unsigned short` occupy 2 bytes of memory and must be aligned on a 2 byte boundary.
- `long`, `unsigned long` and `float` occupy 4 bytes of memory and must be aligned on a 4 byte boundary.
- `long long`, `unsigned long long` and `double` occupy 8 bytes of memory and must be aligned on an 8 byte boundary.
- `long double` occupies 16 bytes of memory and must be aligned on an 8 byte boundary.

The memory-alignment rules of CDR were chosen to be similar to the memory-alignment requirements of some CPUs. It was hoped that this would facilitate optimizations in CORBA's transport layer. However, experience has shown

[1] *Marshaling* and *unmarshaling* are commonly used terms. However, some people use other terms, such as *encoding* and *decoding*, or *serializing* and *deserializing*.

[2] *Big-endian* and *little-endian* refer to the whether multi-byte integers are stored in memory with their most-significant or least-significant byte first. These terms originate from the book *Gulliver's Travels* written by Jonathan Swift in 1726. In this satirical novel, a race of people have a civil war because they cannot agree if a hard-boiled egg should be eaten starting at its big end or its little end.

that the memory-alignment requirements are mainly a small coding nuisance for people implementing CORBA. Such is the benefit of hindsight.

Finally, there are rules for how compound types are marshaled in terms of the basic types:

- An `enum` is marshaled as an `unsigned long`.
- A `string` is marshaled as the length of the string (an `unsigned long`), followed by each character of the string, and is followed with a terminating null character.
- A `struct` is marshaled by marshaling each field.
- An `exception` is marshaled as the exception's repository id (Section 9.4 on page 102), which is a string, followed by each field.
- A `union` is marshaled as its discriminant followed by the active branch (if any).
- A `sequence` is marshaled as the number of elements it contains (represented as an `unsigned long`) followed by each element.
- An array is marshaled by marshaling every element in the array.
- An `any` is marshaled by marshaling its `TypeCode` (Section 15.2 on page 139) and then its embedded value.
- An object reference is marshaled by marshaling an IOR, which is a `struct` shown in Figure 10.1 on page 106.

The above discussion is not exhaustive, for example, it does not discuss the marshaling of a `TypeCode` or a `valuetype`. However, it does provide a representative example of how CDR marshaling works. The memory-alignment rules imply that a GIOP message may contain some padding bytes, which is wasted bandwidth. However, in practice, padding bytes usually account for only a few percent of a GIOP message, so the wasted bandwidth is quite minimal.

11.3 GIOP Message Types

GIOP defines eight different types of messages that can be transmitted between client and server applications. Theoretically, a developer should not need any knowledge of these different types of messages. However, in practice, a knowledge of the different message types can be useful when debugging a CORBA application, particularly if the CORBA vendor product can print low-level diagnostics about the messages that are being sent and received.

The first 4 bytes of every GIOP message contain the letters G-I-O-P. This acts as a "magic number" that identifies the message as being in GIOP format.

The GIOP message types are: *Request*, *Reply*, *Fragment*, *CancelRequest*, *CloseConnection*, *MessageError*, *LocateRequest* and *LocateReply*. The following subsections discuss each of these message types.

11.3.1 Request and Reply Messages

As might be expected, *Request* and *Reply* messages are used to send requests from a client to a server and replies back from the server to a client, respectively.

11.3.2 LocateRequest and LocateReply Messages

A *LocateRequest* message is like a "ping" message that asks: "Is the object there?". The reply is sent back in a *LocateReply* message. CORBA introduced these messages to make its GIOP redirection mechanism more efficient, as will be discussed in Section 11.4.

11.3.3 Fragment Messages

If a *Request* or *Reply* message is very large then the CORBA runtime system might decide to transmit it in several pieces rather than as one monolithic message. In such cases, the first piece is sent as a normal *Request* or *Reply* message, but a flag in the header of the message indicates that there are more pieces to follow. The remaining pieces are transmitted as *Fragment* messages.[3]

A CORBA product is *not* obliged to split large messages into smaller fragments, but it has the *option* of doing so. If a CORBA product is capable of sending *Fragment* messages then typically a value in a runtime configuration file specifies the maximum size of an unfragmented message. Regardless of whether or not a CORBA product can *send* fragmented messages, it is obliged to be able to *receive* fragmented messages.

[3] Support for fragmentation of *Request* and *Reply* messages was introduced in GIOP 1.1. In addition, support for fragmentation of *LocateRequest* and *LocateReply* messages was added in GIOP 1.2.

11.3.4 CancelRequest Messages

A CORBA client may specify a timeout value when making a remote call. If the client does not receive a *Reply* message before the specified timeout has occurred then the CORBA runtime system in the client gives up waiting for the *Reply* message and instead throws a `TIMEOUT` exception back to the client application code. The CORBA runtime system in the client may also (but is *not* obliged to) send a *CancelRequest* message to the server, to let the server know that the client will ignore a *Reply* message if one is later sent.

The CORBA runtime system in the server can use the *CancelRequest* message as a hint to discard the previously-received *Request*. However the server can ignore this hint and, in fact, *will ignore it* if the server has already started dispatching the previously received *Request* message. This is because the CORBA runtime system in the server cannot know how to "cancel" partially-executed application-level code in the body of an operation.

11.3.5 CloseConnection Messages

CORBA allows idle socket connections to be closed. If a connection from a client to a server is closed then a new connection can be opened transparently if, later on, the client makes more calls to the server. This closing of idle socket connections enables a CORBA server to scale up to deal with thousands, or tens of thousands of interactive clients.

CORBA needs to guard against the following possibility. A client's socket connection to a server might have been idle for some time and so the server decides to close the socket connection. However, just as the server is about to close the connection, the client sends a request to the server. If the client's sending of the request overlaps with the server's closing of the socket then the client might think that the server process had crashed. To prevent this scenario from occurring, a server process sends a *CloseConnection* message to the client immediately before closing the socket connection. When the client receives this message, it assumes that the server will ignore any requests that the client had recently sent but for which it had not yet received replies. Because of this, the client would (transparently) open a new connection to the server and re-send the requests.

11.3.6 MessageError Messages

If a CORBA application receives a message that is not in GIOP format then it sends back a *MessageError* message to say "You sent me garbage!" This message should not normally be sent if a CORBA client is communicating with a CORBA server (unless there are *serious* bugs in the CORBA product(s) being used).

You can force a CORBA server to send a *MessageError* message by getting a non-CORBA application to send a message to the CORBA server. For example, if you know which port a CORBA server is listening on then you can connect to the server with `telnet`. When you type something into `telnet` and hit the ENTER/RETURN key, the non-GIOP message will be sent to the CORBA server, and the CORBA server should send a *MessageError* message back (and probably close the connection). Within `telnet`, you will see the letters G-I-O-P followed by some non-printable characters (which is the rest of the very short *MessageError* message).

11.4 GIOP Redirection

When a client sends a *Request* message to a server, it receives back a *Reply* message. The header of the *Reply* message indicates if it is:

1. A "normal reply". In this case the body of the reply contains `inout`/`out` parameters and the return value, if any.

2. An "exception reply". In this case the body of the reply contains a CORBA exception. When the CORBA runtime system in the client receives such a reply, it unmarshals the exception and re-throws it so that the calling code can catch it.

3. A"redirection reply" (actually called `LOCATION_FORWARD` in CORBA terminology). This tells the client that the target object does not live in the server process but rather lives somewhere else. The body of this reply contains an IOR that redirects the client to where the object really resides. The CORBA runtime system in the client then *transparently* resends its message using the new IOR. Furthermore, the client sends all future requests using the new IOR. In effect, the redirection occurs for only the first request; subsequent requests go directly to the target object.

The redirection reply message is used by CORBA vendors to implement an *implementation repository* (IMR), which is discussed in Chapter 7. IMRs add a lot of flexibility to CORBA, but there is *potentially* a significant overhead to be paid for the initial redirection. I now discuss this overhead and an optimization that reduces it. Let us assume that a client's first request to an object contains a `sequence<octet>` parameter that is several megabytes large (the data might be a digitized image or a large sound file). When the client receives the redirection reply, it will have to *retransmit* the multi-megabyte request using the new IOR. Obviously, having to transmit such a large request twice would be a waste of bandwidth. For this reason, CORBA defines the *LocateRequest* and *LocateReply* messages (discussed in Section 11.3.2). A *LocateRequest* message is a very compact "ping"-style message that asks: "Is the object there?". The reply is sent back in a *LocateReply* message. The intention is that the CORBA runtime system in a client application will (transparently) send a *LocateRequest* message *before* sending the first "real" request. This ensures that if a redirection occurs then the redirection will be dealt with before any (potentially large) "real" requests are transmitted.

Use of *LocateRequest* messages is an optional optimization. Some CORBA products (especially older ones) *never* send *LocateRequest* messages. Some other CORBA products *always* send a *LocateRequest* message before the first "real" invocation. It is possible to imagine a CORBA product that is implemented so it normally sends a *LocateRequest* message but optimizes it away if the first "real" request is quite small. However, it is unlikely that many CORBA products are implemented this way because the performance gain to be made from optimizing away "unnecessary" *LocateRequest* messages is insignificant in real-world deployments.

11.5 Active Connection Management (ACM)

The CORBA GIOP protocol infrastructure allows idle connections between a client and server to be closed, and transparently re-opened if the client later wishes to send another request to the server. This allows CORBA implementations to optimize their usage of network resources. The *CloseConnection* GIOP message (discussed in Section 11.3.5) provides the low-level infrastructure required for this. The CORBA standard does not define any *terminology* for this ability to close idle connections. Because of this lack of CORBA-provided terminology, some vendors have defined their own terminology for this capability. Both Orbix and Orbacus use the term *active connection management* (ACM) to refer to the automatic closing of idle socket connections. Other CORBA products might use different terminology for the same concept.

CORBA *does not require* that a product implement ACM; rather, it is an optional capability, though good-quality, modern CORBA products should implement it. Neither does CORBA specify what heuristics should be used to close idle connections. For example, in Orbacus, entries in a configuration file are used to specify that connections should be closed if they have been idle for a specified amount of time. Orbix uses a different mechanism: configuration entries specify the maximum number of open connections; once this limit has been reached, Orbix closes the connection that has been idle for the longest period of time. Other CORBA products may employ different heuristics for closing idle connections.

11.6 Service Contexts

The concept of a *service context* is easy to understand. However, the terminology is not very intuitive. The *context* part of the name arises because a service context is used to pass extra "contextual" information with request and reply messages. The *service* part of the name arises because the first uses of service contexts were to help implement some of the CORBA Services, such as the Security Service and Object Transaction Service (OTS). However, documented APIs mean that service contexts *can* also be used by application programmers.

```
module IOP {
    ...
    typedef unsigned long ContextId;
    struct ServiceContext {
        ContextId        id;
        sequence<octet>  data;
   };
   ...
};
```

Figure 11.2: IDL definition of a service context

The IDL definition of a service context is shown in Figure 11.2. As can be seen, a service context is simply a `struct` that contains binary data and an integer value. The integer value is an identifier that specifies how the binary data should be interpreted. For example, one integer value specifies that the binary data contains information used by OTS; another integer value specifies that the binary data contains information used for security; and so on. Organizations can contact the OMG to request unique integer values that they can use for service contexts specific to their own needs.

The header of each *Request* and *Reply* message contains a sequence of service contexts. Unless an application makes use of, say, OTS, it is likely that *most* messages sent and received by the application will contain an empty sequence of service contexts. An application can use a *portable interceptor* (Chapter 14) to add a service context to outgoing messages and can interrogate incoming messages to see if they contain a service context that corresponds to a specified integer identifier. In this way, if an application receives a service context that it is not expecting then the service context is simply ignored.

11.7 Codeset Negotiation

Codeset is an abbreviation of *coded character set*. Examples of codesets include ASCII and Unicode. In the early versions of CORBA, ISO Latin 1 (which is US ASCII extended with accented characters) was the hard-coded codeset used to transmit parameters of type `char` or `string`. However, CORBA has matured to support *wide characters* (IDL types `wchar` and `wstring`).[4] CORBA has also matured so that the codesets used to transmit characters and wide characters are no longer hard-coded, but rather are negotiated when a client application establishes a connection to a server. As might be expected, this is called *codeset negotiation*. Codeset negotiation is achieved through the following steps:

1. An object reference (Chapter 10) exported from a server contains “contact details” (for example, host, port and object key), but also contains details of which codesets can be used to communicate with the object. In particular, the object reference specifies the *native* `char` and `wchar` codesets used by the server application *plus* a sequence of codesets that can be used to transmit `char` or `string` types, and another sequence of codesets that can be used to transmit `wchar` or `wstring` types.[5] The codesets in the sequences are called *conversion* codesets because they are codesets for which the CORBA runtime system in the server application can convert to and from its native codeset.

2. When a client application imports an object reference, it compares the codesets in the object reference to the codesets that its own CORBA runtime understands. If the CORBA runtime in the client cannot find any overlap between its own codesets and those of the server then it throws an exception to indicate that it cannot communicate with the object. Assuming that it can find codesets common to both itself and the server, the CORBA runtime system in the client decides which codesets it will use for communication.

[4] A codeset is said to be *wide* if every character is represented by a 16-bit or 32-bit fixed-length number.

[5] To be pedantic, an object reference contains integers that uniquely identify codesets rather than the codesets themselves.

3. When a client sends its first *Request* message to the server, it uses a service context (Section 11.6) to tell the server which codesets it has chosen for communication. These are called the *transmission* codesets.

11.8 Bidirectional GIOP/IIOP

The specification of GIOP/IIOP states that it is a *unidirectional* protocol. A client transmits a message on a *channel* (GIOP terminology for what is known as a *socket* in IIOP) to a server and the server transmits its reply on the *same* channel. This *sounds* like bidirectional communication since messages are transmitted in both directions. However, the reason why it is said to be *unidirectional* is that request messages can be transmitted in only one direction. GIOP does *not* allow a server to transmit a request to a client on the same channel that the client uses to transmit requests to the server. This is illustrated in Figure 11.3. The client opens a channel to a server in order to invoke upon obj_1, and the client passes a reference to a *callback* object (obj_2) as a parameter to an invocation.[6] Later, if the server wants to invoke an operation on this callback object then GIOP does *not* allow the server to send its request on the already-open channel. Instead, the server must open a new channel to the client.

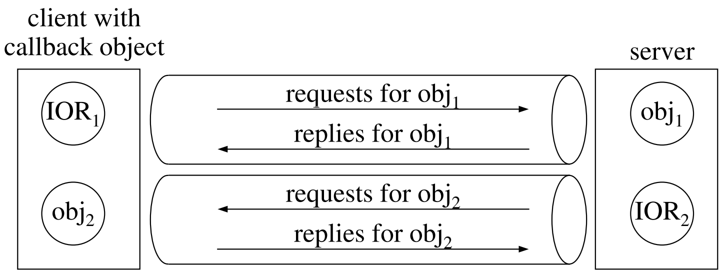


Figure 11.3: Channel usage in (unidirectional) GIOP

When GIOP was being defined, there was some debate within the OMG over whether GIOP should be a unidirectional or bidirectional protocol. However, it was decided that GIOP should be a unidirectional protocol and almost everybody has regretted this decision ever since. In particular, unidirectional protocols have two drawbacks when callback objects are used:

[6] Callback objects are discussed in Section 1.4.2.2 on page 10.

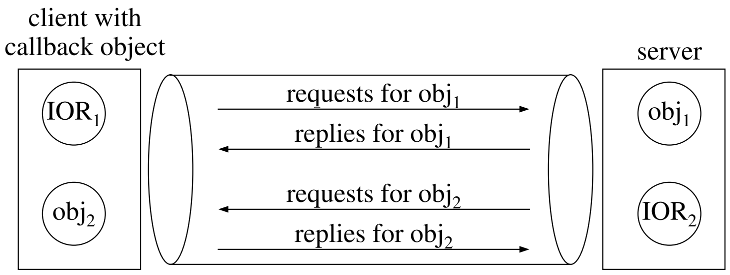


Figure 11.4: Channel usage in bidirectional GIOP

1. Opening a second channel is wasteful of network resources.
2. In some real-world deployments, there may be some firewalls between the client and server applications. Although it may be possible for the client to successfully traverse firewalls to establish a connection to a server, it is sometimes impossible to establish a *new* connection from the server back to the client.

To work around these problems, a bidirectional enhancement to the GIOP/IIOP specification (illustrated in Figure 11.4) was added to CORBA 2.4.

Chapter 12

The `corbaloc` and `corbaname` URLs

12.1 Introduction

URLs used on the world wide web (WWW) begin with the name of a protocol, followed by `":"`, for example, `"http:"`, `"ftp:"` or `"file:"`. A stringified object reference (Section 3.4.2 on page 34) begins with `"IOR:"` so this also looks similar to a URL.

In early versions of CORBA, the only kind of string parameter that could be passed to `string_to_object()` (Section 3.4.2 on page 34) was a stringified object reference. CORBA has now matured to allow *other* URL-like strings to be passed as parameters to `string_to_object()`. A CORBA product may optionally support the `"http:"`, `"ftp:"` and `"file:"` formats. The semantics of these is that they provide details of how to download a stringified IOR (or, recursively, download another URL that will eventually provide a stringified IOR).

Although support for `"http:"`, `"ftp:"` and `"file:"` is optional, all CORBA products must support `"corbaloc:"` and `"corbaname:"`, which are two URLs defined by the OMG. The purpose of these is to provide a human readable/editable way to specify a location where an IOR can be obtained.

12.2 The `corbaloc` URL

Some examples of `corbaloc` URLs are shown below:

```
corbaloc:iiop:1.2@host1:3075/NameService
corbaloc:iiop:host1:3075,iiop:host2:3075/NameService
```

The first URL specifies that an IOR can be obtained by using version 1.2 of the IIOP protocol to send a *LocateRequest* message (Section 11.3.2 on page 115) with parameter `"NameService"` to port 3075 on host `host1`.

The second URL is different in two ways. First, by omitting `"1.2@"`, it uses the default version (`1.0`) of the IIOP protocol. Second, the URL specifies two `<host>:<port>` addresses rather than one. In general, any number of `<host>:<port>` addresses can be specified, separated by commas. This second form is used to provide fault tolerance: the *LocateRequest* message will be sent to one of the addresses in the list; if that `<host>:<port>` cannot be contacted then another address in the list will be tried, and so on.

Many parts of the `corbaloc` URL have default values:

- The default protocol is `iiop`.
- If the protocol is `iiop` then the default *version* of IIOP that is used is 1.0. It is advisable to specify the most recent version of IIOP that is understood by both the client and server application. This is because more modern versions of IIOP tend to have better capabilities that might make client-server interaction more efficient.
- The default port number is 2809. This is the port that the Internet Assigned Numbers Authority (`www.iana.org`) has assigned for use with `corbaloc`.

The CORBA specification currently specifies two protocols that can be used in `corbaloc` URLs. One protocol is `iiop`, which has already been discussed. The other protocol is called `rir`, which seems like a strange name until you realize that it is an acronym for *resolve initial references*. Unsurprisingly, this protocol specifies that an object reference should be obtained by calling the `resolve_initial_references()` operation (Section 3.4.1 on page 33), passing the specified name as a parameter. For example, the `corbaloc` URL below specifies that an IOR should be obtained by calling `resolve_initial_references("NameService")`:

```
corbaloc:rir:/NameService
```

One benefit of the `rir` protocol is that it allows `string_to_object()` to subsume the functionality of `resolve_initial_references()`. For example, instead of an application being hard-coded to find the Naming Service by calling `resolve_initial_references("NameService")`, an application can now be hard-coded to find the Naming Service by obtaining a string from a command-line argument or a configuration file and passing this to `string_to_object()`. *If* the string happens to be `"corbaloc:rir:/NameService"` then it is just as if the programmer had used `resolve_initial_references()`, but now there is the flexibility for the string parameter to be a stringified IOR or a `corbaloc` URL that uses the `iiop` protocol. In this way, applications have some extra flexibility in how they find a CORBA Service.

The `rir` protocol is not used often in `corbaloc` URLs. However, it is used more commonly in `corbaname` URLs, which I now discuss.

12.3 The `corbaname` URL

A `corbaname` URL is a `corbaloc` that specifies how to contact the Naming Service, followed by `"#"` and then a name within the Naming Service. Some examples are shown below:

```
corbaname::foo.bar.com:2809/NameService#x/y
corbaname::host1,:host2,:host3/NameService#x/y
corbaname:rir:/NameService#x/y
```

Passing of the above strings as a parameter to `string_to_object()` causes the Naming Service to be located and `resolve_str()` to be invoked to obtain an IOR from the Naming Service. As the above examples illustrates, a `corbaname` URL can use either the `iiop` or `rir` protocols to locate the Naming Service.

12.4 Architectural Support for `corbaloc`

12.4.1 Client-side Support for `corbaloc`

The `string_to_object()` operation has built-in support for `corbaloc` and `corbaname` URLs:

- If the parameter to `string_to_object()` starts with `"IOR:"` then the operation treats it as a stringified object reference and builds a corresponding proxy.

- If the parameter starts with `"corbaloc:rir"` then the operation calls `resolve_initial_references()` and passes the specified name as a parameter.

- If the parameter is a `corbaloc` URL that uses the `iiop` protocol then the operation opens a socket connection to the specified host and port, and sends a *LocateRequest* message (Section 11.3.2), using the specified name as the *object key* (Section 10.2.1 on page 107) in the header of the message. The IOR embedded in the returned *LocateReply* message is used as the return value of `string_to_object()`. An important point to note is that `corbaloc` is built on top of *existing* low-level GIOP messages so the OMG did *not* have to define a new version of GIOP to support `corbaloc` URLs.

- If the parameter to `string_to_object()` is a `corbaname` URL then the embedded `corbaloc` details are use to locate a Naming Service. Then `string_to_object()` invokes `resolve_str()` on the Naming Service, passing it the string after the embedded `"#"` as a parameter. The IOR returned from `resolve_str()` is used as the return value of `string_to_object()`.

12.4.2 Server-side Support for `corbaloc`

CORBA does *not* standardize the server-side support for `corbaloc` URLs, nor even the *terminology* for this server-side support. This means that CORBA products provide proprietary mechanisms, often with proprietary terminology. For example:

- The Orbix implementation repository has built-in, server-side support for `corbaloc` URLs, and this is referred to as *named keys*. A named key is a mapping from the *name* component in a `corbaloc` URL to a stringified IOR. The `named_key` sub-commands of the `itadmin` administration utility are used to `create`, `show`, `list` and `delete` named keys. By default, the Orbix implementation repository listens on port 3075 so `corbaloc` URLs should be formatted as shown below:

  ```
  corbaloc::<host-of-IMR>:3075/<name>
  ```

  When the `itconfigure` utility is used to set up an Orbix domain, named keys are automatically created for whatever CORBA Services are added to the domain. For example, if the domain has a Naming Service then a named key called `NameService` is created.

  For a long time, Orbix did not expose APIs for embedding server-side `corbaloc` support in normal server applications. Orbix 6.1 Service Pack 1 is the first version of Orbix to expose these APIs.

- Orbacus provides some proprietary APIs (in the `BootManager` interface) that can be used by developers to embed server-side `corbaloc` support in their own server applications. These APIs are used by the Orbacus implementation repository, which looks up *name→stringified-IOR* mappings in a configuration file.

- TAO provides proprietary APIs that have different names, but similar semantics, to those of Orbacus.

- OmniORB server-side support for `corbaloc` URLs relies upon placing objects into a specific, predefined POA. OmniORB also provides a prewritten server application called `omniMapper` that listens on a specified port number and looks up *name→stringified-IOR* mappings in a configuration file.

As can be seen, each CORBA product has its own different "look and feel" for server-side support of `corbaloc` URLs. Because of this, there is *no* portable way for a CORBA server to use a `corbaloc` URL to advertise one of its own objects. Developers concerned with writing portable CORBA applications should use `corbaloc` URLs *only for* CORBA Services, for example, the Naming Service, Notification Service, Trading Service and so on.

12.5 Bootstrapping Interoperability Problems

One obvious requirement for interoperability between different CORBA products is that they must be able to speak the same on-the-wire protocol (IIOP). However, that by itself it not sufficient. Another, less obvious requirement for interoperability is for one CORBA product to be able to *find*, say, the Naming Service or the Notification Service of another CORBA product. For example, how can an Orbix client *find* (connect to) the Naming Service of an Orbacus installation. This is often called a bootstrapping problem. The `corbaloc` and `corbaname` URLs were invented to address such bootstrapping issues, as I now discuss.

A CORBA application connects to a CORBA Service—for example, the Naming Service, Transaction Service, Notification Service, and so on—by calling `resolve_initial_references()` and passing the name of the desired service as a parameter. The CORBA specification does *not* specify how `resolve_initial_references()` works (that is an implementation detail), but in most CORBA products this operation looks in a configuration file to find a *name-of-CORBA-service→stringified-IOR* mapping[1] and then passes the stringified IOR as a parameter to `string_to_object()`. These mappings are normally set up during the installation and configuration of a CORBA product. To configure, say, Orbix to use an Orbacus Naming Service is a matter of obtaining a stringified IOR of the Orbacus Naming Service (typically from the Orbacus configuration file) and copying this into the Orbix configuration file. Then the next time an Orbix client calls `resolve_initial_references("NameService")`, the client will be directed towards the Orbacus Naming Service. This technique works fine, but it is a bit cumbersome because stringified IORs are not human readable. However, with the introduction of `corbaloc` URLs, the technique becomes much easier. Now, instead of copying a stringified IOR of the Orbacus Naming Service into the Orbix configuration file, it is sufficient to copy a `corbaloc` URL into the Orbix configuration file. The fact that `corbaloc` URLs are easy to read (and edit) by humans makes it more feasible for an organization to use several different CORBA products.[2]

Sometimes, practical or organizational issues may make it awkward to update a configuration file with a stringified IOR or `corbaloc` URL for, say, the Naming Service of another CORBA product. To work around this, the OMG defined two standard command-line options that all CORBA products must support.[3]

The first command-line option takes the form:

```
-ORBInitRef <name>=<value>
```

An example is shown below:

```
-ORBInitRef NameService=corbaloc::host1:3075/NameService
```

The `<value>` in `<name>=<value>` is a stringified IOR or URL that is used if `resolve_initial_references()` is called with `"<name>"` passed as a parameter. This command-line argument takes precedence over any corresponding information in the CORBA product's configuration file. You need

[1] For example, the entry in the Orbacus configuration file is called `ooc.orb.service.<service>`. The corresponding entry in the Orbix configuration file is called `initial_references:<service>:reference`.

[2] It is rare for an organization to *deliberately decide* to use several CORBA products. However, several CORBA products may make their way into an organization if different departments or development teams make independent choices about which middleware technology they will use, or if the development of CORBA applications is outsourced to other organizations.

[3] When a CORBA application calls `ORB_init()`, it passes command-line arguments as a parameter to `ORB_init()`. This provides the mechanism by which command-line arguments are communicated to the CORBA runtime system.

to specify this command-line option *each* time you run an application, so regular use of it can get somewhat tedious. However, this command-line option is useful if, for example, restrictive file permissions prevent you from modifying the configuration file of a CORBA installation. It can be useful also when trouble-shooting a connectivity problems in a network.

The second command-line option takes the form:

```
-ORBDefaultInitRef <URL-up-to-but-not-including-final-"/">
```

Some examples are shown below:

```
-ORBDefaultInitRef corbaloc:iiop:1.2@host1:3075
-ORBDefaultInitRef corbaname::host1/NameService#x/y
```

A call to `resolve_initial_references("<name>")`, results in `"/<name>"` being appended to the string provided by the command-line argument after `-ORBDefaultInitRef`; the result of this string concatenation is then passed as a parameter to `string_to_object()`.

The intention of `-ORBDefaultInitRef` is that a user can set up a centralized store of *name* $\rightarrow$ *IOR* mappings and then applications can be started with a single `-ORBDefaultInitRef` command-line argument that points to this centralized store. This is usually more convenient than starting many applications, each with several `-ORBInitRef` command-line arguments.

You need to specify the `-ORBDefaultInitRef` command-line option *each* time you run an application so, just as with `-ORBInitRef`, regular use of it can get tedious. In general, it is usually more convenient to create or modify a configuration file for a CORBA installation than to use these command-line options every time you run a CORBA-based application.

If both `-ORBInitRef` and `-ORBDefaultInitRef` command-line arguments are used then the `-ORBInitRef` arguments take precedence.

Chapter 13

Current Objects

13.1 The Concept of Thread-local Data

The UNIX `getpid()` function returns an integer "process identifier" that uniquely identifies the process. Likewise, libraries of threading primitives provide a similar function, perhaps called `get_thread_id()`, that returns an integer to uniquely identify the current thread. The name of this function varies across different threading packages. A common coding technique used in multi-threaded programs is to use a lookup table to provide a mapping from a thread identifier to associated data. This technique allows a programmer to associate a different piece of data with different threads. The technique is often called *thread-local data* or *thread-specific data*.

13.2 Current Objects Provide Thread-local Data

CORBA uses the term *Current* to mean thread-local data.[1] Actually, CORBA defines a few different types of current object. For example, there is one for security, another for transactions, and yet another for request dispatching. All of the current interfaces are defined in IDL, and they all inherit from an empty base type called CORBA::Current.

As an example, `PortableServer::Current` inherits from `CORBA::`

[1] To be pedantic, *Current* is data associated with a *request* rather than a *thread*. However, since a request is executed within the context of a thread, the concept of request-local data is usually synonymous with thread-local data, and thinking of current as being thread-local data is "accurate enough" for most discussions.

```
module CORBA {
  local interface Current { };
  ...
};
module PortableServer {
  ...
  typedef sequence<octet> ObjectId;
  local interface Current : CORBA::Current {
    exception NoContext { };
    POA get_POA() raises(NoContext);
    ObjectId get_object_id() raises(NoContext);
  };
  ...
};
```

Figure 13.1: IDL definition of `Current` types

`Current` and is used to access information about the target object for the request being dispatched by the current thread. The definitions of these types are shown in Figure 13.1.

Current objects are accessed by specifying the relevant name as a parameter to `resolve_initial_references()`, which is discussed in Section 3.4.1 on page 33. For example, the `PortableServer::Current` singleton object is called *POACurrent* because you gain access to it by passing `"POACurrent"` as a parameter to `resolve_initial_references()`. Use of *POACurrent* is required when implementing servers with the "default servant" model (Section 5.6.4 on page 55), and is optional with the other POA models.

Chapter 14

Portable Interceptors

Early CORBA products were monolithic: when you bought a CORBA product you could use whatever capabilities were built into the product, but you (or third party companies) had little opportunity to extend the range of capabilities that it offered. CORBA has since matured to provide a "plug-in" architecture that allows people to add new code to a CORBA product. A first version of this plug-in architecture was made available in CORBA 2.2 and it was called *interceptors*. The name came about because the plug-ins could intercept some of the ORB functionality to modify the ORB's behavior. Unfortunately, the CORBA 2.2 interceptors were under-specified and this resulted in a non-portable API for their use. CORBA 2.4 provided a more complete definition and the result is now known as *portable interceptors*.

There are two types of portable interceptors: IOR interceptors and request interceptors. I provide a brief overview of each in turn in the following sections. A more substantial and very readable overview of portable interceptors can be found in the *Pure CORBA* book [Bol01].

14.1 IOR Interceptors

An IOR interceptor is called when an IOR is being created. The IOR interceptor can find out which policies were used in the object's POA (see Chapter 6 for a discussion of POA policies) and can use this information to decide if it wants to embed an extra `TaggedComponent` (Section 10.2.3 on page 108) into the IOR. For example, whether or not an object can take part in distributed transactions is determined by the use of a particular POA policy. If this POA policy is used then an IOR interceptor that is bundled as part of a transactional service will add a corresponding `TaggedComponent` to the IOR.

14.2 Request Interceptors

There are two types of request interceptor: one that deals with the client-side mechanics of sending a request and receiving a reply, and another that deals with the server-side mechanics of receiving a request and sending a reply. Obviously, if an application is both a client and a server then it could use both a client-side and serve-side interceptor. Unsurprisingly, the APIs for both client-side and server-side request interceptors have a similar look and feel.

A request interceptor is called at various points along the transmission of request and reply/exception messages. An interceptor may do the following:

- Examine the parameters of the request/reply/exception message that is being transmitted. An interceptor that does this and uses the `DynAny` APIs (Section 15.3 on page 140) could write diagnostic messages to a log file for all incoming/outgoing messages.

- A server-side interceptor can find out the object id (Section 5.6.1 on page 51) and POA (Section 5.5 on page 47) of the target object. It could use this information to keep track of when the objects have been accessed by clients and then garbage collect objects that have not been accessed in, say, the last 20 minutes.

- Add a service context (Section 11.6 on page 119) to an outgoing message or extract a service context from an incoming message.

- Examine the `TaggedComponent`s of the target object. For example, a client-side interceptor that is used as part of an transactional service may check to see if the target object can take part in a transaction and, if so, add a transactional service context to the outgoing request.

14.3 The PICurrent Object

The portable interceptors specification defines a *Current* object (Chapter 13), which is called *PICurrent* because it is accessed by passing `"PICurrent"` as the parameter to `resolve_initial_references()` (Section 3.4.1 on page 33). The main purpose of the PICurrent object is to provide a way for service contexts (Section 11.6 on page 119) to be communicated between a portable interceptor and application-level code.

Chapter 15

Meta-information Programming

15.1 What is Meta-information Programming?

In many computer languages, when you write a program, you know at compilation time what data-types you will manipulate and the compiler performs strong type checking to ensure that you access those data-types correctly. Some programming languages defer type checking until runtime. In such languages, there is some *meta information* (also known as *runtime type information*) associated with objects; the runtime system of the language uses this information to check that a program accesses objects in a legal way. Some languages that use runtime type checking even allow *programmers* to inspect an object's meta information in order to find out what operations the object has and the names and types of its instance variables. This capability is often called *introspection* or *reflection*.

The Smalltalk language is famous for its runtime type checking. Java provides compile-time type checking but also provides APIs for the `java.lang.Class` type that allow runtime type checking (and introspection) to be performed. In contrast, the C programming language does not provide meta information and C++ provides only very limited support for it with its *dynamic cast* capability.

15.1.1 Uses of Meta-information Programming

Most CORBA programs are written with compile-time knowledge of the IDL data-types that they will manipulate and the IDL interfaces that they will implement and/or invoke upon. However, CORBA also maintains meta information about objects and data-types, and makes it possible to write programs that make use of such meta information. This capability of CORBA is rarely used, but it does make it possible for some very important types of application to be written. For example:

- It is common for an organization to have some existing ("legacy") applications that were built with one middleware technology, want to build new applications with a different middleware technology, and to somehow connect the old and new applications to each other. The software that connects between different middleware technologies (or connects between different on-the-wire protocols) is usually called a *bridge* or a *gateway*.

  The basic principle of a gateway is to accept an incoming request using the on-the-wire protocol of one technology, perform data-type translation on parameters and then send a similar request using the on-the-wire protocol of the other technology. Writing a gateway for a specific set of APIs is usually tedious, highly repetitive work and if the APIs change then you have to rewrite the gateway for the new APIs. However, if the middleware technologies that are being bridged provide meta information then it usually possible to implement a *generic* gateway that can accept *any* incoming request using the on-the-wire protocol of one technology, use meta information to determine the quantity and types of parameters in the request, convert these parameters into the format of the other technology and then send a similar request using the on-the-wire protocol of this second technology.

  A generic gateway is usually a bit slower than a type-specific gateway, but it has the big advantage that you have to implement just one generic gateway and it can be used to connect systems for *any* set of APIs. Some companies developed gateways from CORBA to DCE (an older middleware technology) and several CORBA vendors sell COM-to-CORBA or .NET-to-CORBA gateways. These gateways are usually built as generic gateways that utilize meta information.

- A debugger for a traditional language uses *symbol table information* (which is a form of meta information) embedded in executables to display variables in a human-readable format rather than just as blobs of binary data. If a middleware technology provides meta information then this makes it possible for companies to develop debugging tools that are better tailored to the needs of that middleware technology. It also makes it possible to develop some other useful tools that support the software development process. For example, when writing a CORBA server, it is useful to have a *simulation client* application that can be used to perform ad-hoc testing of the server, and vice versa. Some companies sell tools that make use of meta information to produce simulation clients/servers with little, if any, coding effort.

CORBA's support for meta-information programming is spread over several distinct, but complementary, APIs. These APIs are discussed in the following sections.

15.2 TypeCodes and the Interface Repository

CORBA uses the `CORBA::TypeCode` type to represent meta information. The `kind()` operation provided by `CORBA::TypeCode` returns an enum value that specifies if the TypeCode represents one of the built-in types (`long`, `boolean` and so on) or a user-defined type (`struct`, `union`, `interface` and so on). If the TypeCode represents a user-defined type then more operations can be invoked upon it to determine the names and TypeCodes of the members of a `struct` or `union` and so on.

A TypeCode provides the meta information required by the ORB runtime system to determine how the bytes within a raw chunk of memory are laid out as the fields/components of a compound data-structure. However, the information provided by TypeCodes is *insufficient*, by itself, for general-purpose, meta-information programming. For example, a TypeCode does not specify the signatures of operations that are provided by an interface. Neither does a TypeCode provide a list of all the types (and nested modules) that are declared within a module. CORBA complements TypeCodes with the *Interface Repository* (IFR), which stores this more general meta information.

You can think of the IFR as being a database of meta information about IDL types. This database has a CORBA server "wrapper" around it; it is this wrapper that is called the IFR. The IDL interfaces of the IFR provide operations for storing meta information in the IFR and also operations for querying the contents of the IFR. The IFR organizes the meta information in a manner similar to how the internals of a compiler organize information in a parse tree. Because of this, querying meta information in the IFR is similar to traversing a parse tree in a compiler.

Although CORBA specifies the IDL interface of the IFR, CORBA does not specify *how* administration of the IFR is performed. Typically, a CORBA product provides a proprietary command-line tool that parses an IDL file and invokes operations on the IFR to store the parsed information in the IFR. The name of this command-line tool varies from one CORBA product to another. With some CORBA products, this tool is a stand-alone utility; with other CORBA products this functionality is provided as a command-line option on the IDL compiler.

The information provided by the IFR subsumes the information provided by TypeCodes. However, each call to the IFR is a remote call so it incurs the performance overhead of network latency. In contrast, querying the information in a TypeCode involves just a local operation call, so this is much faster. A common way to structure a meta information program is to use the APIs of the the IFR to navigate down to the parse-tree node of a particular type and then obtain a TypeCode that allows the details of that type to be examined much more efficiently with just local operation calls.

15.3 The `any` and `DynAny` Types

One of the IDL built-in types is called `any`. This serves a similar purpose to the `void*` type in C/C++ or the `java.lang.Object` type in Java: it is a way to pass around data when you do not have any compile-time knowledge of the type of the data. Of course, when doing this you need to have some way of finding out at runtime what is the type of the data. Internally, an `any` contains the raw data *plus* a `CORBA::TypeCode` (Section 15.2) that specifies the data's type.

The `any` type is used about as infrequently as the `void*` type is used in C/C++ or `java.lang.Object` is used in Java. In other words, it is used extensively for some specialized programming tasks, but is irrelevant for many other, more general-purpose tasks.

When `any` *is* used in IDL, it is often used to define a type that holds a name and arbitrary value, as shown in Figure 15.1.

```
struct NameValuePair {
    string    name;
    any       value;
};
```

Figure 15.1: Use of type `any` to define a name and value

The IDL compiler generates operations that can be used to insert a user-defined type into an `any`, to query the type of data inside an `any`, and a type-safe way to extract data of a specified type from an `any`. These operations can be used by applications that have been compiled with the stub code (Section 1.4.5 on page 13) of an IDL file. However, these APIs for inserting values into, and extracting values from, an `any` can be used *only* by applications that have been compiled with the stub code of the type that is embedded inside the `any`.

If an application wants to manipulate data embedded inside an `any` *without* being compiled with the relevant stub code then the application must convert the `any` into a `DynAny`, which is the base type of a hierarchy of `local` interfaces (Section 9.1 on page 95). There are sub-types of `DynAny` for each IDL construct. For example, there are types called `DynStruct`, `DynUnion`, `DynSequence` and so on.

The operations on the `DynAny` interfaces allow a programmer to recursively drill down into a compound data-structure that is contained within the `DynAny` and, in so doing, decompose the compound type into its individual components that are built-in types. Operations on the `DynAny` interface can also be used to recursively build up a compound data-structure from built-in types.

A `DynAny` object can be converted to an `any`, and back again. This is important because many CORBA APIs take parameters of type `any` instead of `DynAny`.

15.4 Dynamic Invocation Interface (DII)

The *dynamic invocation interface* (DII) is a set of APIs that allows a client application to make invocations on an object reference (Chapter 10) without the client application being compiled with the stub code for either the relevant IDL interface or the types passed as parameters. A DII-based client typically does the following:

- The client application obtains an object reference from somewhere.
- The client can invoke the `get_interface()` operation on the object reference to access meta information in the IFR (Section 15.2), and so find out the signatures of operations defined for the object.
- The client uses the `DynAny` APIs to build up parameter values and then converts these `DynAny` objects to `any` objects.

- The target object reference, the name of the operation being invoked, and a sequence of parameter values/directions are added to a `CORBA::Request` pseudo-object. The specified operation is invoked by calling `invoke()` on this `CORBA::Request` object.
- The client examines `inout`/`out` parameters and the return value by using the APIs of `DynAny`.

The DII APIs are used when a client is compiled without knowledge of the IDL interfaces upon which it will make calls. This generally means that the client application does not have much hard-coded "business logic" to dictate what parameter values it should use when making remote calls, or even which operations it should invoke. Instead, the remote invocations made by a DII-based client are typically driven by some external meta-data. Two example uses of DII-based applications—gateways and test clients—were briefly discussed in Section 15.1.1 on page 138. I now discuss the architecture of a DII-based test client in slightly more detail.

When writing a CORBA server, it is useful to have a *simulation client* application that can be used to perform ad-hoc testing of the server. A GUI-based "generic" test client could be built using the DII, and it might work as follows.

- The user specifies an object reference that the test client should use. This object reference might be obtained from the Naming Service (Chapter 4) or as a stringified IOR (Section 3.4.2 on page 34) from a file.
- The GUI client retrieves the *repository id* from the object reference. It then uses this type information to obtain the signatures of its operations from the IFR. The GUI displays a menu of the operation names.
- The user selects an operation from the menu and the GUI then displays dialog boxes to prompt the user to provide values for all `in` and `inout` parameters. If a parameter is of a compound type then the GUI client will use the TypeCode to recursively drill down into the type and display dialog boxes for each component of the compound type. The compound value is built up with the aid of the `DynAny` APIs.
- When the user has provided all the parameter values, the GUI client creates a `CORBA::Request` object and calls `invoke()` upon it. When the operation returns, the GUI displays all the `out` and `inout` parameters, and the return value.

It is possible to imagine such a GUI that would record the parameter values inputted by the user and then generate a *regression testing* program that could be rerun independently of the interactive GUI.

15.5 Dynamic Server Interface (DSI)

The *dynamic server interface* (DSI) is often described as being the server-side equivalent of the DII. It is a set of APIs that allows a server application to process incoming requests on IDL interfaces for which it does not have the relevant stub code or skeleton code. Like the DII, the DSI can be used to build gateways or testing applications. For example, a DSI-based testing server might accept incoming requests and assign values to `inout` and `out` parameters based on values it obtains from a random number generator or a configuration file/database.

Chapter 16

CORBA Messaging

The CORBA Messaging specification encompasses three separate, but complementary, topics. Each of the topics are discussed in the following sections.

16.1 Quality of Service (Policy Objects)

The first version of CORBA did not define any portable way for programmers to specify what *quality of service* (QoS) they wanted in their applications. Instead, many CORBA vendors provided proprietary APIs for this purpose.

Over time, CORBA has matured to provide a mechanism for programmers to specify what QoS they want in their applications. In CORBA terminology, a QoS value is called a *policy* object. Policy objects were first introduced with the POA specification (Section 5.5 on page 47). The POA specification initially defined 7 types of policy and separate operations for creating each of these 7 kinds of policy object. The OMG realized that the addition of new policies to CORBA would not be practical if each new policy required that a new operation be added to an existing interface because this would create a versioning nightmare. Instead, the Messaging specification (which was introduced *after* the POA specification) defined a more general-purpose mechanism for introducing new policies to CORBA. This mechanism involves the following:

- The base type `CORBA::Policy` is a local interface (Section 9.1 on page 95) from which all policy types inherit.

- The `ORB` type contains an operation called `create_policy()` that can be used to create an instance of any of the subtypes of `CORBA::`

Policy. This operation takes two parameters that specify the *type* and *value* of the policy object. For example, if the *type* is RELATIVE_RT_TIMEOUT_POLICY_TYPE (which is an integer constant that implies the RelativeRoundtripTimeoutPolicy subtype of CORBA::Policy) then the *value* expresses the desired timeout value. The *type* parameter is always a (typedef'd) integer constant, and there is a one-to-one mapping between these integer constants and subtypes of CORBA::Policy. However, the *value*'s type varies from one (sub)type of policy to another. Because of this, the *value* parameter is represented as an any (Section 15.3); this provides the flexibility for the embedded value to be an integer, a string or an arbitrary IDL data-structure. Internally, create_policy() uses its parameters to create an instance of the appropriate subtype of CORBA::Policy.

The programming steps required to create a policy object have the drawback of being verbose, but offer two benefits.

The first benefit is that this creation mechanism is general-purpose. This means that it can deal with more policy types (both CORBA-compliant and vendor-proprietary policy types) that might be introduced in the future.

The second benefit concerns the fact that the use of some policy types has the side-effect of embedding extra information in IORs (Chapter 10) or service contexts (Section 11.6 on page 119). The fact that there is an integer constant corresponding to each subtype of CORBA::Policy means that this compact integer value can be embedded inside the IORs and service contexts. Because of this, the use of policy objects does *not* incur a significant bandwidth overhead.

Aside from providing a general-purpose mechanism for defining policy types, the Messaging specification also defines some specific policy types that can be used to control the QoS for making remote calls. These policies include the following:

- CORBA allows idle socket connections to be closed (to conserve network resources) and previously-closed connections to be re-established. The RebindPolicy type is used to specify whether or not the re-establishing of previously-closed connections should take place transparently to client applications.

- The signature of an IDL operation may be prefixed with the oneway keyword. The intention is to specify that the operation should be invoked asynchronously. However, early versions of CORBA did not define clear semantics of oneway calls and so the semantics provided by different

CORBA vendors varied widely. The `SyncScopePolicy` type is now used to precisely specify the semantics of `oneway` invocations.

- There are two priority policies: one to specify the priority for delivering a request message, and another to specify the priority for delivering a reply message.
- There are various start-time policies that can be used to prevent a request/reply message from being delivered before a specified time. There are also various timeout policies that can be used to cancel the delivery of a request/reply message if it cannot be delivered by a specified time.
- There are various policies that are used in conjunction with request/reply routers (Section 16.3).

The Messaging specification also defines a service context (Section 11.6 on page 119) that is used to propagate request- or reply-related policy values between clients and servers. It is important to note that if a client application specifies a timeout value for a remote request and the timeout occurs then the CORBA runtime system in the server can cancel the request *only if* the request is still queued up to be dispatched within the server; the server *cannot* cancel the request if it has already been dispatched to the target object.

16.2 Asynchronous Messaging Interface (AMI)

By default, IDL operations provide *blocking* call-and-reply semantics, that is, when a client application makes a remote call to an object in a server, the client blocks until the server sends back a reply. These semantics are suitable for many applications, but some applications can benefit from *non-blocking* call-and-reply semantics.

For a long time, developers had just two choices for obtaining non-blocking call-and-reply behavior:

1. The client could create a new thread and get that thread to make a blocking call. In this way, the main thread of the client is not blocked. This approach cannot be used if the client application must be single-threaded. Also, even if a client can be multi-threaded, the overhead of thread creation means that this approach does not scale up to make a lot of non-blocking calls.
2. The client could use the `send_deferred()` API of the DII (Section 15.4 on page 141). However, the DII can be very tedious to use

so programmers rarely wanted to use this approach. Also, with this approach, the client application had to periodically poll to check if the reply had come back. Polling can waste a lot of CPU time if it is done too frequently. On the other hand, infrequent polling can cause delays in processing replies.

CORBA has now matured to provide a different non-blocking call-and-reply mechanism, which is called Asynchronous Messaging Interface (AMI).

AMI is an optional part of CORBA and not all CORBA products support it. Also, AMI requires that additional operations be generated into proxy classes by the IDL compiler. Unfortunately, doing this would break binary portability of Java CORBA products. For this reason, no Java CORBA product can support AMI—at least, not until the IDL-to-Java mapping is updated to support AMI.[1] For these reasons, if you are considering using AMI in a project then it is vitally important to check that the CORBA product you are using supports AMI.

If a CORBA product does support AMI then it works as follows:

- For each IDL operation, `foo()`, the IDL compiler generates: (1) a *blocking* operation called `foo()`, (2) a *non-blocking* operation called `sendc_foo()`, and (3) another *non-blocking* operation called `sendp_foo()`. The first operation is generated by all IDL compilers; the other two generated operations are new to AMI. Note that AMI support in an IDL compiler might be disabled by default, so you might have to pass a command-line option to the IDL compiler to instruct it to generate the AMI operations. The documentation provided by your CORBA vendor should mention which command-line option to use.

- The `sendc` version of the operation takes `in` and `inout` parameters. It also takes a *callback object* (Section 1.4.2.2 on page 10) as a parameter. An operation on the callback object will be invoked later with the returned `out` and `inout` parameters and return value. It is up to the client developer to implement the callback object as a servant (Section 5.2 on page 45) and activate it into a POA (Section 5.5 on page 47). It is important to note that the callback object is a "normal" CORBA object, so it could actually be located in another CORBA application. In

[1] There is one exception to this. The Java-based JacORB product has added experimental support for AMI. The experience gained by the JacORB development team in doing this is likely to be fed back into the OMG standardization process for enhancing the IDL-to-Java mapping in order to support AMI.

effect, one application can send requests to a server and the reply can be processed by either the same client *or* another application.[2]

- The `sendp` version of the operation takes `in` and `inout` parameters. It returns a *poller* object. The client can periodically invoke upon this returned poller object to determine if the reply has arrived and, if so, find out the values of the returned `out` and `inout` parameters and return value. The implementation of the poller object is generated by the IDL compiler.

It is important to note that both the callback and polling communication mechanisms are implemented entirely within the client-side ORB runtime system—a server application is unaware of, and is unaffected by, how a client has made a request.

It is also important to note that of the CORBA implementations that support AMI, most support the callback model but do *not* support the polling model. This is because most developers view the callback model as being superior to the polling model and so CORBA implementors have little motivation to implement the polling model.

16.3 Time Independent Invocations (TII)

GIOP (Chapter 11) is a synchronous protocol. This means that the asynchronous messaging infrastructure of CORBA must be layered *on top of* GIOP. For example, the *callback* and *polling* models (Section 16.2) are layered on top of GIOP by the CORBA runtime system in client applications. Another commonly desired aspect of a messaging system is *time independent invocations* (TII), which can be explained as follows. Let us assume that a client application sends a request to a server but then the client is killed before it receives the reply. TII means that the reply message is kept in some form of persistent storage until the client application is restarted and then the reply message is delivered to the client.

The CORBA Messaging specification provides TII by introducing another piece of infrastructure called message routers. A *message router* is a delegation server, that is, a server that receives an incoming message and delegates it (passes it on) to another application.

[2] As will be mentioned soon, AMI is implemented entirely within the CORBA runtime system of the client application. Because of this, the reply message will always be sent from the server to the CORBA runtime system of the client from which the request originated; it is the CORBA runtime system in the client that then invokes the reply on the callback object, which can be in the same client process or in another application.

An IOR (Chapter 10) contains the contact details of an object. However, an IOR can optionally have a `TaggedComponent` (Section 10.2.3 on page 108) that contains a sequence of embedded router IORs. The sequence of router IORs reflects the chain of delegation, from the first router to the next router and so on to the eventual target server. If a client uses such an IOR but the client does *not* have the relevant routing policies set then requests are sent in the normal manner, that is, direct to the object in the server. However, if the client *does* have the relevant routing policies set then the CORBA runtime system in the client uses a different mechanism to send requests: the client sends the request to the first router, and the IOR of a callback object is passed along with the request. Each router delegates the request message to the next router. Eventually, the last router sends the request to the target server. As far as the target server is concerned, the last router is its client, and so the server sends its reply message to this last router. The routers then pass the reply message back along the chain of routers. The router closest to the client then delivers the reply message by invoking the appropriate operation on the callback object provided by the originating client.

The "added value" provided by the routers is that each router can store request/reply messages in a persistent store, such as a database. If any process—the client, one of the routers or a server—dies during the delegation of request/reply messages then the persistent storage of the messages means that the delegation can continue when the dead process is restarted.

Several policies (Section 16.1) can be used to control the QoS of message delivery:

- The `RoutingPolicy` type controls whether or not requests to IORs with an embedded routing `TaggedComponent` are sent "normally" or via the routers. The `ROUTER_NONE` value means that the requests are sent directly to the target IOR. The `ROUTE_FORWARD` value means that the request is delivered via routers. The `ROUTE_STORE_AND_FORWARD` value means that the routers store messages persistently, in order to provide TII.

- The `MaxHopsPolicy` type is used to specify an upper limit on the number of "hops" (delegations from one router to another) that can occur when delivering a message.

- The `QueueOrderPolicy` type is used to tell routers in which order they should delegate messages. For example, they can delegate messages in first-come, first-served (FIFO) order, in order of a priority associated with each message, or in the "deadline" order (of when messages will timeout).

16.4 Further Reading

A very readable and more detailed overview of AMI can be found in some magazine articles [SV99a, SV99b].

Chapter 17

Proprietary Fault Tolerance

Some of the low-level infrastructure defined by CORBA makes it possible for a CORBA product to have fault tolerance capabilities. However, for a long time, the CORBA specification did not specify the details of *how* this fault tolerance should be provided or administered. The result was that many CORBA products provided simple-to-use (but limited) fault tolerance capabilities in a proprietary manner. More recently, the OMG has defined the (optional) CORBA Fault Tolerance specification, which provides a more standardized and more powerful (but also more complex) fault tolerance infrastructure.

This chapter discusses the proprietary fault tolerance mechanisms provided by some CORBA implementations. Section 17.1 introduces some basic issues that need to be addressed by fault tolerance mechanisms. Then Section 17.2 gives overviews of the proprietary fault tolerance mechanisms in several CORBA products. Finally, Section 17.3 discusses some miscellaneous issues that need to be considered when using proprietary fault tolerance mechanisms. A discussion of the newer CORBA Fault Tolerance specification is deferred until the next chapter.

17.1 Basic Issues in Fault Tolerance

17.1.1 Replication Granularity

One important aspect of fault tolerance is the need for replication. Because CORBA is an object-oriented middleware technology, you might think that fault tolerance in CORBA is about the need to replicate *objects*. Although this is true, CORBA objects live in (POAs within) server processes, so, at a practical level, replicating CORBA objects involves replicating server processes. Indeed, most CORBA products that support replication do so at the granularity of entire server processes rather than at the finer granularity of individual objects.[1] This is because replication at the coarser granularity of an entire process dramatically reduces the volume of information that the CORBA runtime system needs to maintain about replicated entities, and so increases the scalability of applications that utilize replication.

17.1.2 Contact Details for a Replicated Object

If a client fails to communicate with one replica of an object then the client should attempt to communicate with another replica of the object. In order to do this, the client needs to have access to the "contact details" for all the replicas of an object. The flexibility of both interoperable object references (IORs) and implementation repositories (IMRs) provide the necessary infrastructure for this, as I now discuss.

Replicated Server deployed *without* an IMR. As discussed in Chapter 10, an IOR can contain *several* sets of contact details. This makes it possible for a single IOR to contain a separate set of contact details for each replica of an object. A client attempts to communicate with a replicated object by using one of the sets of contact details in the IOR. If this fails then the client switches over to use another set of contact details in the IOR. CORBA does not specify *which* contact details should be tried first, but they are usually tried in the order they appear in the IOR.

Replicated Server deployed *with* an IMR. As discussed in Section 7.2.3 on page 74, some CORBA products allow a replicated server to be registered with the IMR. If this is done then the IOR of a replicated object contains the host and port of the IMR; when a client sends its first request to the IMR, the IMR redirects the client to one of the replicas of the desired object. The client sends future requests to that same replica. If the client's communication with the replica ever fails then the CORBA runtime system in the client switches back to using the original host and port, which is for the IMR, and the IMR then redirects the client to another replica of the desired object.

[1] At least one CORBA product, Orbix, allows replication at the mid-level granularity of individual POAs. However, in practice, very few Orbix applications take advantage of this finer level of replication granularity. Instead, most deployed Orbix applications that use replication choose to replicate *all* the POAs within a server process, so the effect is to replicate the entire server process.

Replicated IMR. In order to prevent the IMR from becoming a single point of failure, the IMR should itself be replicated. An IOR for an object in a server that is deployed through a replicated IMR contains the host and port details for *all* the IMR replicas. Because of this, the client's first request is sent to one of the IMR replicas. If the client cannot communicate with that IMR replica then the CORBA runtime system in the client switches over to use another replica of the IMR. The IMR replica then redirects the client to the desired (and possibly replicated) server.

17.1.3 Use of `PERSISTENT` POAs

The meaning of the `PERSISTENT` and `TRANSIENT` POA policies are explained in Section 6.1.3 on page 62. To recap briefly, a POA (Section 5.5 on page 47) is a container for *servants* (the host programming language objects that represent CORBA objects). The policies that are used to create a POA are applied to all the servants within a POA. If a POA has the `PERSISTENT` policy then the IORs for all servants/objects in that POA are valid even if the server process dies and is restarted. Conversely, if a POA has the `TRANSIENT` policy then the IORs for all servants/objects in that POA are valid only for the duration of the server process: once the server process dies, the `TRANSIENT` IORs automatically become invalid.

At a philosophical level, it is not sensible for `TRANSIENT`, that is, temporary, objects to be fault tolerant. At a practical level, a `TRANSIENT` IOR is automatically invalidated when a server terminates,[2] so this makes it difficult, if not impossible, for a fault tolerance infrastructure to work correctly when server processes that contain `TRANSIENT` objects die and are restarted.

If you want to implement a server that will be deployed with fault tolerance then ensure that you use `PERSISTENT` POAs in the server.

17.2 Example Products

17.2.1 OmniORB

OmniORB does not provide an implementation repository so there is no need to discuss how to replicate an omniORB server that is deployed through an

[2] The automatic invalidating of a `TRANSIENT` IOR is usually achieved by having the CORBA runtime system in a server embed a timestamp in the *object key* (Section 5.6.1 on page 52). When a server dies and is restarted, the timestamp information in a previously-exported `TRANSIENT` IOR is out of date; this ensures that a `TRANSIENT` IOR is not valid across restarts of a server.

IMR. Instead, this section focuses on how to deploy a replicated omniORB server *without* an IMR.

OmniORB has two configuration variables that, when combined, can be used to set up a replicated server.

- `endPoint = giop:tcp:<host>:<port>`
  The `giop:tcp:` portion of the value specifies that the IIOP communication protocol is to be used (omniORB supports other communication protocols too); after that the server's `host` and listening `port` are specified. Some readers might think it is redundant to have to explicitly specify the host on which the server is running. However, doing this can be useful if the server runs on a multi-homed machine, that is, if the server's machine is known by several different names or has several different IP addresses. The server listens on the specified `host:port` and also embeds that `host:port` information in IORs for objects within the server.

- `endPointNoListen = giop:tcp:<host>:<port>`
  The value of this variable is in the same format as that of `endPoint`. The server does *not* listen on the specified `host:port`, but it *does* embed that `host:port` information in IORs for objects within the server.

Let us assume that you want to have three replicas of a server: one replica will run on `host1.foo.com` and listen on port 5000, another replica will run on `host2.foo.com` and listen on port 6000, and the final replica will run on `host3.foo.com` and listen on port 7000. The omniORB configuration variables should be set for the first server replica as shown below:

```
# Extract from omniORB configuration file for replica 1
endPoint = giop:tcp:host1.foo.com:5000
endPointNoListen = giop:tcp:host2.foo.com:6000
                 = giop:tcp:host3.foo.com:7000
```

The result is that the server listens on `host1:5000`, but exported IORs indicate that objects within the server can be contacted at any of: `host1:5000`, `host2:6000` or `host3:7000` (all within the `foo.com` domain). The corresponding information in the configuration file for the second replica is shown below:

```
# Extract from omniORB configuration file for replica 2
endPoint = giop:tcp:host2.foo.com:6000
endPointNoListen = giop:tcp:host1.foo.com:5000
                 = giop:tcp:host3.foo.com:7000
```

Likewise, the information in the configuration file for the third replica is shown below:

```
# Extract from omniORB configuration file for replica 3
endPoint = giop:tcp:host3.foo.com:7000
endPointNoListen = giop:tcp:host1.foo.com:5000
                 = giop:tcp:host2.foo.com:6000
```

The overall effect is that each server listens on its own `host:port`, but exported IORs contain the `host:port` details for *all* the replicas.

Note that an omniORB application finds the omniORB configuration file by the value of the `OMNIORB_CONFIG` environment variable. If you want to run several server replicas on the same machine then you will need to have several configuration files (one for each server replica) and set the `OMNIORB_CONFIG` environment variable appropriately when starting each replica. If you prefer to not have multiple configuration files then you can specify the `endPoint` and `endPointNoListen` configuration information as command-line options when starting each server replica. The servers then pass the command-line options as a parameter to `ORB_init()`, as discussed in Section 3.2.3 on page 29.

17.2.2 Orbix

Orbix allows a replicated server to be deployed with or without an implementation repository. I discuss each of these forms of deployment in turn.

17.2.2.1 Deploying a Replicated Server *with* an IMR

Orbix administration is performed through sub-commands of the `itadmin` utility. Each sub-command performs a small amount of work so you typically need to execute several `itadmin` commands to complete a useful unit of work, such as registering an Orbix server with the IMR. However, `itadmin` has a built-in interpreter for an open-source scripting language called Tcl (pronounced *Tickle*). This makes it possible to write a Tcl script that performs the entire sequence of `itadmin` commands required to carry out a task. The *Orbix Administration Made Simple* chapter of the *CORBA Utilities* package [McH] discusses several useful task-based `itadmin` Tcl scripts. One of those scripts, `orbix_srv_admin`, simplifies the work involved in registering a server so that it can be deployed through the IMR.

When registering a server with `orbix_srv_admin`, you can specify either a single host or a list of hosts on which the server is to be run.[3] If you

[3] Technically, you specify a list of *node daemons*, rather than a list of hosts. However, there

specify a list of hosts then `orbix_srv_admin` registers the server as a *replicated server*.

As discussed in Section 17.1.2 on page 154, the IOR of a object in a replicated server contains the host and port of the IMR; when a client sends its first request to the IMR, the IMR redirects the client to one of the replicas of the desired object. The client sends future requests to that same replica. If the client's communication with the replica ever fails then the CORBA runtime system in the client switches back to using the original host and port, which is for the IMR, and the IMR can then redirect the client to another replica of the desired object.

The `itconfigure` utility is used to set up the initial configuration for an IMR. A option in this utility makes it easy to create a replicated IMR (so that the IMR is not a single point of failure). If you do this then, as discussed in Section 17.1.2, an IOR for an object in a server that is deployed through a replicated IMR contains the host and port details for *all* the IMR replicas. Because of this, the client's first request goes to one of the IMR replicas. If the client cannot communicate with that IMR replica then the CORBA runtime system in the client switches over to use another replica of the IMR. The IMR replica then redirects the client to the desired (and possibly replicated) server.

The Orbix IMR keeps track of which server replicas are currently running, and it can optionally be used to automatically restart servers that have died. When a client sends its first request to the IMR, the IMR redirects the client to one of the currently running server replicas. The IMR can use either a *round-robin* or *random* policy in choosing to which server replica a client is redirected. By doing this, the fault tolerance mechanism also provides a per-client load balancing strategy.

17.2.2.2 Deploying a Replicated Server *without* an IMR

By default, an Orbix server with persistent POAs (Section 6.1.3 on page 62) is deployed through the Orbix IMR. If you want to deploy a persistent-POA server *without* the Orbix IMR then your server has to invoke some Orbix-proprietary APIs.[4] However, the `PoaUtility` class (discussed in the *Creation of POA Hierarchies Made Simple* chapter of the *CORBA Utilities* package [McH]) encapsulates the use of these proprietary APIs and makes it easy

is normally one node daemon per host so the distinction between node daemons and hosts is not relevant to the discussion at hand.

[4] The need to use proprietary APIs to choose between different deployment models for Orbix servers is slowly disappearing. Orbix 6 allows this decision to be made through runtime configuration values for C++ applications but, at the time of writing, Java applications still require use of proprietary APIs.

for different deployment options to be chosen through, say, a command-line option.

When deploying a persistent-POA server without an IMR then the server should listen on fixed ports. The fixed ports can be specified in the Orbix runtime configuration file, as illustrated in Figure 17.1. The following points should be noted about this configuration:

- Orbix configuration is arranged into (potentially nested) scopes. For example, `BankSrv` is a scope, and `BankSrv.replica_1` is one of three scopes nested within it. When an Orbix application is started, the `-ORBname <scope>` command-line option can be used to specify the scope from which the application should obtain its configuration.

- Orbix allows different POAs within a single server to listen on the same or different ports. Proprietary APIs are required to make use of this flexibility, but the `PoaUtility` class [McH, Ch. 5] encapsulates the use of these APIs, and allows each POA manager (Section 5.7 on page 56) to use a different port. The example in Figure 17.1 assumes that the servers use the `PoaUtility` class and have two POA managers: one identified by the label `"core"` and the other identified by the label `"admin"`.

- Variables with names of the form `<label>:iiop:addr_list` specify a list of `host:port` strings, some of which may have an optional `"+"` prefix. The server listens on any `host:port` entries that are *not* preceded with `"+"`. However, *all* the `host:port` pairs are embedded in IORs for objects (within POAs controlled by the `<label>` POA manager) within the server.

If a server replica is started with the `-ORBname BankSrv.replica_1` command-line argument then that server listens on both `host1:5000` and `host1:5001` (one port per POA manager), but the server also embeds the other listed `host:port` details in exported IORs. By starting a second replica with `-ORBname BankSrv.replica_2` and starting a third replica with `-ORBname BankSrv.replica_3`, the overall effect is that each server listens on its ports but exported IORs contain the `host:port` details for *all* the replicas.

This Orbix mechanism is broadly similar to that offered by omniORB (Section 17.2.1 on page 155), with the following minor differences:

- *All* the objects in an omniORB server are accessible through the *same* port, while Orbix allows the possibility for *some* objects in a server to be accessible through one port and *other* objects in the same server to

```
BankSrv {
    replica_1 {
        core:iiop:addr_list  = [ "host1:5000",
                                "+host2:6000",
                                "+host3:7000"];

        admin:iiop:addr_list = [ "host1:5001",
                                "+host2:6001",
                                "+host3:7001"];
    };
    replica_2 {
        core:iiop:addr_list  = ["+host1:5000",
                                 "host2:6000",
                                "+host3:7000"];

        admin:iiop:addr_list = ["+host1:5001",
                                 "host2:6001",
                                "+host3:7001"];
    };
    replica_3 {
        core:iiop:addr_list  = ["+host1:5000",
                                "+host2:6000",
                                 "host3:7000"];

        admin:iiop:addr_list = ["+host1:5001",
                                "+host2:6001",
                                 "host3:7001"];
    };
};
```

Figure 17.1: Orbix fault tolerance configuration for a replicated server

be accessible through a different port. The main apparent benefit of this additional flexibility in Orbix is that one port might be accessible across a firewall while access to the other port (and hence the objects accessible via it) might be restricted to clients inside the firewall. In effect, this provides a simple security mechanism. However, this security mechanism is likely to be too basic for the needs of most organizations. The more flexible approach offered by the CORBA Security Service (Chapter 23) has more widespread appeal.

- If several omniORB applications run on the same computer but require different configuration then this is achieved by having a separate configuration file for each omniORB application. In contrast, the use of a separate scope for each application in an Orbix configuration file allows several applications to share the same configuration file. Minimizing the number of configuration files can simplify administration. Orbix allows configuration information to be stored in either a textual file or in a *configuration repository*, which is a database that is accessed via a CORBA server "wrapper". The use of a configuration repository allows configuration information to be maintained in a centralized location while being accessible from Orbix applications that run on the same or different computers. This centralized maintenance of configuration information can further simplify administration.

The above differences between omniORB and Orbix are relatively minor when deploying replicated servers *without* an IMR. As far as fault tolerance is concerned, the biggest difference between omniORB and Orbix is that Orbix *also* supports fault tolerance for servers that are deployed *with* an IMR (as discussed in Section 17.2.2.1).

17.2.3 Orbacus

Orbacus does not provide any fault tolerance capabilities within its runtime system. Instead, it provides a command-line utility called `iormerge`. An example of its usage is as follows:

```
iormerge -f replica1.ior replica2.ior replica3.ior >new.ior
```

By default, `iormerge` interprets its command-line arguments as stringified IORs. However if, as shown above, the `-f` option is given then `iormerge` interprets its command-line arguments as the names of files that contain stringified IORs. The `iormerge` utility reads the "contact details" from these IORs and writes to standard output a new stringified IOR that contains *all* the contact details. If this newly created IOR is made available to client applications then the clients can communicate with any of the replicas.

Use of `iormerge` is sufficient for replicated servers that contain only a singleton object. However, it is not appropriate for servers that have, say, a factory object (Section 1.4.2.1 on page 9) that can create new objects on demand. The reason for this is that the IOR of a newly created object contains only one set of contact details (for the server replica in which the object is created) rather than the contact details for all the replicas.

17.2.4 Server-side support for `corbaloc` URLs

As discussed in Section 12.4.2 on page 126, CORBA has not standardized the server-side support for `corbaloc` URLs. Instead, most CORBA products provide proprietary APIs that can be used to make an object accessible via a `corbaloc` URL. Let us assume you use these proprietary APIs so that a server makes an object available under the name `"foo"`. If you deploy three replicas of a server that listen on `host1:5000`, `host2:6000` and `host3:7000` then the following `corbaloc` URL can be used by clients to communicate with the replicated `"foo"` object:

```
corbaloc::host1:5000,:host2:6000,:host3:7000/foo
```

Use of such a `corbaloc` URL is sufficient for replicated servers that contain only a singleton object. However, it is not appropriate for servers that have, say, a factory object (Section 1.4.2.1 on page 9) that can create new objects on demand. The reason for this is that the IOR of a newly created object contains only one set of contact details (for the server replica in which the object is created) rather than the contact details for all the replicas.

17.2.5 Critique

As you might expect, the proprietary fault tolerance mechanisms of different CORBA products differ from one other, both in their “look and feel” and in their capabilities. However, they have some characteristics in common:

- With the exception of `corbaloc` URLs, none of the proprietary mechanisms require server-side coding. Rather, they are based on runtime configuration or the use of administrative utilities.

- The proprietary fault tolerance mechanisms are concerned *solely* with getting several sets of “contact details” into the IORs that are exported from replicated servers. The mechanisms neither help you nor hinder you in maintaining state consistency across server replicas.

- Although the mechanisms are proprietary, they do not hinder interoperability with client applications that are implemented using other CORBA products. This is because the proprietary mechanisms are concerned solely with embedding several sets of “contact details” into the IORs that are exported from servers. Since having several sets of contact details in one IOR is CORBA-compliant, a client implemented with any CORBA product can communicate with a server that uses a proprietary fault tolerance mechanism.

17.3 Miscellaneous Issues

17.3.1 Fault Tolerance is not Load Balancing

It should be noted that although fault tolerance and load balancing both rely on the use of replicas, a fault tolerance infrastructure does not necessarily imply load balancing. Most of the fault tolerance mechanisms discussed in this chapter do not provide load balancing. Instead, it is likely (though not guaranteed) that most/all clients will communicate with just one server replica; it is only when that server replica dies that clients will seek another replica with which to communicate. One exception to this concerns replicated servers that are deployed through the Orbix IMR (Section 17.2.2.1). In this case, the IMR performs per-client load balancing, that is, the IMR redirects some clients to one server replica, some other clients to another replica and so on.

Although the load balancing mechanism provided by the Orbix IMR can be useful, it is possible for the load from clients to be spread over server replicas in an unequal manner. For example, let us assume that 10 clients connect, via the Orbix IMR, to two replicas of a server, and that there are 5 clients connected to each replica. If each client sends a similar number of requests to servers then the load is balanced over the two servers, at least initially. However, let us assume that after a few minutes, 3 of the clients connected to one of the replicas terminate. We are now left with just 2 clients communicating with one server replicas while there are 5 clients communicating with the other replica. The Orbix IMR does not have any way of re-balancing the number of clients across the server replicas. Likewise, if a third server replica is now started then the IMR does not have any way of re-balancing the number of clients across the now-increased number of server replicas.

The term *adaptive load balancing* is often used to refer to load balancing mechanisms that periodically try to rebalance load from clients over servers. CORBA has not standardized on a load balancing mechanism and further discussion of load balancing is outside the scope of this book. Several CORBA products provide proprietary load balancing mechanisms. Interested readers are advised to consult documentation of CORBA products for details and to do a search with an Internet search engine for, say, “CORBA adaptive load balancing”.

An alternative approach to load balancing is to invest in a hardware load-balancing switch. The hardware switch acts as a delegation server. It receives requests from client machines and delegates them to server machines specified in the switch’s configuration. Some hardware switches utilize mechanisms that attempt to keep client load balanced across server machines.

17.3.2 Timeout Values in a Fault Tolerant System

A practical issue to keep in mind is that you may need to adjust client-side timeout values when setting up a fault tolerant system. In particular, if a server *process* is not running then a client's attempt to connect to the server will fail quickly (typically within a few milliseconds) and so the CORBA runtime system in the client can quickly fail-over to use another set of contact details in an IOR. However, if the server's *computer* is turned off or is physically disconnected from the network then the client's attempt to connect to the server may take a relatively long time (perhaps tens of seconds) to fail. Such a long delay before a fail-over occurs is often unacceptable to users. You should consult your CORBA vendor's documentation to find out how you can shorten the client's timeout for establishing connections. If your clients and servers all run on a fast local area network then you can usually shorten the connection timeout to a tiny fraction of a second. Doing this will result in a fast fail-over when a server machine dies.

Chapter 18

CORBA Fault Tolerance

The previous chapter discussed proprietary fault tolerance mechanisms provided by some CORBA implementations. This chapter discusses CORBA-FT, which is the commonly-used name for the fault tolerance functionality that was added to the CORBA specification in 2000.

18.1 Terminology and Basic Infrastructure

CORBA-FT types are defined in a module called `FT`. Most of the fault tolerance infrastructure is concentrated in the `FT::ReplicationManager` interface. That interface defines two operations but inherits most of its functionality from base interfaces called `ObjectGroupManager`, `GenericFactory` and `PropertyManager`, all of which are defined in the `FT` module. I now discuss each of these interfaces in turn.

18.1.1 The `ObjectGroupManager` Interface

CORBA-FT uses the term *object group* to mean a replicated object. An individual replica is called a *member* of an object group. An Interoperable Object Group Reference (IOGR) is an IOR (Chapter 10) for an object group. An IOGR contains multiple profiles (that is, multiple sets of contact details): typically, one profile for each member that is *currently running*.[1] An IOGR has an embedded `TaggedComponent` (Section 10.2.3 on page 108) that records

[1] Alternatively, an IOGR might contain a separate profile for each of several *gateways* that delegate requests to object replicas. More details about gateways is provided in the discussion about the `ACTIVE` replication style on page 168.

a version number for the IOGR. Whenever the set of members for an object group changes (for example, when a member dies or is restarted), the version number of the IOGR is updated.

The `FT::ObjectGroupManager` interface defines operations that manipulate information about the currently-alive members in object groups, for example, to add and remove members from an object group.

18.1.2 The `GenericFactory` Interface

`FT::GenericFactory` is a factory interface (Section 1.4.2.1 on page 9). It has *generic* as part of its name because it is used to create objects of arbitrary types.

```
module FT {
  struct  Property { ... }; // name-value pair
  typedef sequence<Property>  Properties;
  typedef  Properties          Criteria;
  typedef CORBA::RepositoryId TypeId;
  ...
  interface GenericFactory
  {
    typedef any FactoryCreationId;
    Object create_object(
              in  TypeId             type_id,
              in  Criteria           the_criteria,
              out FactoryCreationId  factory_creation_id)
                                            raises (...);
    void delete_object(
             in FactoryCreationId    factory_creation_id)
                                            raises (...);
  };
};
```

Figure 18.1: The `FT::GenericFactory` interface

The first parameter to the `create_object()` operation is a *repository id* (Section 9.4 on page 102). This parameter specifies the type of the object to be created. The next parameter specifies a sequence of name-value pairs that can be used for an application-specific purpose, for example, to provide initialization values for the object. The operation returns a reference to the newly created object, but is also uses an `out` parameter to specify an identifier that can later be used to uniquely identify the object to the factory that created it. In particular, this unique identifier can be used as a parameter to the `delete_object()` operation.

18.1.3 The `PropertyManager` Interface

The concept of "one size fits all" does not apply to fault tolerance. Instead, there are many different techniques that can be used to achieve fault tolerance, where each technique is applicable to some but not all types of application. Because of this, developers use *properties* (name-value pairs) to inform CORBA-FT which techniques are used in an application.

The `FT::PropertyManager` interface defines operations that are used to manage the fault tolerance properties. Properties can be set to be *default* properties, *type-specific* properties (that is, properties that are specific to an IDL interface) or *object group* properties (that is, properties specific to a replicated object). Object group properties override type-specific properties, which in turn override default properties.

The fault tolerance properties defined by CORBA-FT are discussed below.

Factories. Developers have to implement the `GenericFactory` interface and create an instance of it in each replica of a CORBA-FT server. When the CORBA-FT infrastructure wants to create members of an object group, it invokes `create_object()` on `GenericFactory` objects in servers.

CORBA is object-oriented rather than process-oriented. Because of this, the CORBA-FT infrastructure does not know in which server process a specific `GenericFactory` resides. To work around this, CORBA-FT defines a type called `FT::Location`, which is a `typedef` of `CosNaming::Name` (Figure 4.1 on page 39), that is, a compound string. In effect, a location is the conceptual name by which CORBA-FT knows a server process. CORBA-FT specifies the format that this compound string should have. In effect, it encodes the host on which a server runs and a conceptual name for the server process.

The factories property takes a value that is a sequence of structures, where each structure contains a reference for a `GenericFactory` object, a location and some *criteria* (like `Properties`, `Criteria` is a sequence of name-value pairs) that can be passed as a parameter to `create_object()`.

The factories property is the only property that *cannot* be set as a default property. Instead, it must be specified for each interface type and/or for each object group. Having a type-specific factories property makes it feasible for developers to have several type-specific factories within a single location (server process), if they should wish. Alternatively, if a developer wants a single factory to be used to create several types of object in a server then that factory can be registered several times: once for each type of IDL interface.

Replication style. The replication style property has one of the following values: `STATELESS`, `COLD_PASSIVE`, `WARM_PASSIVE` or `ACTIVE`. There is also a placeholder for `ACTIVE_WITH_VOTING`, which is likely to be supported in the future. All of these property values are constants defined in the `FT` module.

The `STATELESS` value indicates that the behavior of an object is independent of the history of invocations upon the object. This replication style could be used for an object that provides read-only access to a database.

In the `COLD_PASSIVE` and `WARM_PASSIVE` replication styles, only one member, called the *primary member*, executes the invocations upon the object group. The other members of the object group are called *backup members*. Periodically, a *checkpoint* (that is, a snapshot of the state) of the primary member is taken. In addition, CORBA-FT infrastructure persistently logs every request that is invoked upon the primary member. This log is truncated when a new checkpoint of the primary member's state is taken. When the primary member fails, the most recent checkpoint plus a reply of requests in the log is used to bring a backup member up-to-date and promote it to being the new primary member.

The difference between the `COLD_PASSIVE` and `WARM_PASSIVE` is the technique used to bring a backup member up-to-date. In the `COLD_PASSIVE` replication style, the most recent checkpoint of the old primary member is loaded into the new primary and then requests in the log file are re-invoked on the new primary. In the `WARM_PASSIVE` replication style, whenever a checkpoint of the primary member is taken, it is automatically loaded into the backup members, thereby enabling a failover to be handled faster.

In the `ACTIVE` and `ACTIVE_WITH_VOTING` replication styles, the IOGR contains contact details not for the group members, but rather for *gateways*, which are delegation servers. When a gateway receives a request from a client, it delegates the request to *all* the group members. The CORBA-FT specification allows the gateway to use a proprietary multicast protocol to communicate with the group members (this use of a proprietary protocol is transparent to both client and server developers). Each group member processes an invocation independently and sends its reply back to the gateway.

In the `ACTIVE` replication style, the gateway picks one of the replies and sends this back to the client, and discards the other replies.

In the `ACTIVE_WITH_VOTING` replication style, the gateway collects all the replies and compares them. They *should* all be equal, but if there is a fault somewhere then one or more of the replies might be incorrect. Assuming that a majority of the replies are identical, the gateway sends one of the identical replies to the client and discards the other replies. The `ACTIVE_WITH_VOTING` replication style is not yet supported in CORBA-FT, but it is expected to be supported in the future.

Initial and minimum number of replicas. An integer property is used to specify the *initial* number of replicas of an object that should be created. Another integer property specifies the *minimum* number of replicas of an object that are needed to maintain the desired fault tolerance. If “too many” replicas terminate so that the quantity of replicas falls below the desired minimum then more replicas are created to increase the quantity back to the desired minimum level.

Membership style. The value of the membership style property can be either `MEMB_INF_CTRL` or `MEMB_APP_CTRL`.

If the `MEMB_INF_CTRL` membership style is used then a server creates an object group by invoking `create_object()` on the replication manager, which is part of the CORBA-FT infrastructure. The replication manager then invokes `create_object()` on factories in server processes, in order to create replicas of the object.

If the `MEMB_APP_CTRL` membership style is used then a server creates an *initially empty* object group by invoking `create_object()` on the replication manager. The server then must then do one of the following to populate the object group with members:

- The server can repeatedly invoke `create_member()` on the replication manager, each time specifying a different location where a member is to be created. The replication manager invokes `create_object()` on the factories at each of the specified locations.

- Alternatively, the server can repeatedly invoke `create_object()` on factories in different locations and then invoke `add_member()` on the replication manager for each of these newly created replicas.

Consistency style and checkpoint interval. The value of the consistency style property can be either `CONS_INF_CTRL` or `CONS_APP_CTRL`.

If the `CONS_INF_CTRL` consistency style is used then the CORBA-FT infrastructure automatically performs checkpointing, logging of requests and failover when a primary member terminates. When the CONS_INF_CTRL consistency style is used then a complementary policy value is used to specify the frequency at which checkpoints are performed.

If the CONS_APP_CTRL consistency style is used then the server application code must perform checkpointing, logging of requests and failover when a primary member terminates.

Fault monitoring, interval & timeout, and granularity. The fault monitoring style property can be one of PULL, PUSH or NOT_MONITORED. The PUSH fault monitoring style is not yet supported, but support for it is anticipated in the future.

If the PULL fault monitoring style is used then the CORBA-FT infrastructure periodically invokes a "ping"-style operation, called is_alive(), to check if object members are still alive.

The fault monitoring interval and timeout property is a struct that has two fields. One field specifies the frequency of the "ping" requests, and the other specifies the time allowed for responses to those requests to determine whether an object is faulty.

The fault monitoring granularity property complements the PULL fault monitoring style. The granularity can be one of MEMB, LOC, or LOC_AND_TYPE. The MEMB (short for *member*) value indicates that the CORBA-FT infrastructure must ping each member individually. If the number of object groups (and members within those groups) is large or if the monitoring interval is very short then this property value can result in significant network overhead. The network overhead can be reduced by specifying the LOC (short for *location*) value. A *location* is, in essence, a server process. A single ping message is sent for all the object members that live at the same location (that is, within the same server process). If the pinged object is faulty then it is assumed that all the object members at that location are faulty. The LOC_AND_TYPE value is a similar to the LOC value except that one ping message is sent for all the object members of the same (IDL interface) type that live at the same location.

If the NOT_MONITORED fault monitoring style is used then CORBA-FT does not periodically check if object members are still alive. Instead, developers implement their own fault monitoring functionality and report faults to CORBA-FT's *fault notifier*, which is discussed in Section 18.5.

18.1.4 The `ReplicationManager` Interface

As stated previously, the `ReplicationManager` interface inherits from `ObjectGroupManager`, `GenericFactory` and `PropertyManager`. It also defines two new operations: `register_fault_notifier()` and `get_fault_notifier()`. A fault notifier (interface `FaultNotifier`) is an object that is used for reporting faults. A discussion of fault notifiers is provided in Section 18.5.

An implementation of CORBA-FT provides an infrastructure process that contains a `ReplicationManager` object, and application programs can connect to this by passing `"ReplicationManager"` as a parameter to `resolve_initial_references()`. The replication manager object appears to application programmers as a single object. However, in reality it is replicated so that it is not a single point of failure. The CORBA-FT specification requires that there must not be any single points of failure within an implementation of CORBA-FT.

18.2 Writing CORBA-FT Servers

18.2.1 Modifications to IDL Interfaces

The first step in making a fault-tolerant server is to ensure that the IDL interfaces of objects implemented by a server inherit from appropriate interfaces defined by CORBA-FT. As an example, let us assume that a server is to implement an interface `Foo`, and also a *factory* interface (Section 1.4.2.1 on page 9) for it called `FooFactory`. The IDL for the server might be written as shown in in Figure 18.2.

The `PullMonitorable` interface defines an `is_alive()` operation. This is a “ping”-style operation that is periodically invoked by the CORBA-FT infrastructure to check which replicas of an object are alive. A servant can trivially implement this so that it always returns *true* to indicate that is is alive. Alternatively, a servant’s implementation of this operation could perform application-specific health checks to ensure that the server process (or related hardware) is in a consistent state, and return *true* only if this is the case.

The `Checkpointable` interface defines operations, `get_state()` and `set_state()`, that are used to get and restore the “state” (for example, instance variables) of an object. The state is represented as binary data (that is, `sequence<octet>`), and it is the responsibility of the server developer to convert an object’s state to and from this binary format.

An IDL interface can optionally inherit from `Updateable`, which is a

```
interface Foo
  : FT::PullMonitorable , FT::Checkpointable
{
  ...
  void destroy();
};
interface FooFactory
  : FT::PullMonitorable , FT::Checkpointable
{
  Foo create(...);
};
```

Figure 18.2: Example IDL for a Server that uses CORBA FT

subtype of `Checkpointable`. The `Updateable` interface defines two operations called `get_update()` and `set_update()`. An "update" is a delta change in the state of an object since the last checkpoint.

Whether or not an IDL interface has to inherit from `Checkpointable` or `PullMonitorable` depends on the fault tolerance properties in effect for objects of that type. In particular, the IDL interface must inherit from `Checkpointable` if it uses the `CONS_INF_CTRL` policy along with either the `COLD_PASSIVE` or `WARM_PASSIVE` policies. Likewise, the IDL interface must inherit from `PullMonitorable` if it uses the `PULL` fault monitoring style.

18.2.2 Creating and Destroying Replicated Objects

To implement a CORBA-FT server, a developer must write servant classes that implement the server's IDL interfaces—`Foo` and `FooFactory` in our running example—and *also* write a servant class for the `GenericFactory` interface (shown previously in Figure 18.1 on page 166). This means that the `GenericFactory` interface is implemented by the replication manager infrastructure process (Section 18.1.4) and *also* by every server process.

When a server wishes to create a replicated object—for example, in the body of `FooFactory::create()`—it does *not* create the object locally. Instead, what happens depends on the membership style being used.

If the `MEMB_INF_CTRL` membership style is being used then the server calls `create_object()` on the `GenericFactory` interface that is inherited by the replication manager. The replication manager then invokes `create_object()` on the `GenericFactory` objects in some of the server replicas. The implementation of `create_object()` in a server replica creates a normal CORBA object that is a member of (that is, replica in) the object group. The replication manager then constructs an IOGR by combining the contact details of the individual members. It is this IOGR that `FooFactory::create()` returns to the client.

If the `MEMB_APP_CTRL` membership style is used then a server creates an *initially empty* object group by invoking `create_object()` on the replication manager. The server then must then do one of the following to populate the object group with members:

- The server can repeatedly invoke `create_member()` on the replication manager, each time specifying a different location where a member is to be created. The replication manager invokes `create_object()` on the factories at each of the specified locations.

- The server can repeatedly invoke `create_object()` on factories in different locations and then invoke `add_member()` on the replication manager for each of these newly created replicas.

The `create_object()` operation has an `out` parameter that is used to associate an identifier with a newly created object. This identifier is unique within the factory that creates the object. It is the responsibility of the caller to remember this identifier and to pass it as a parameter when later calling `delete_object()`.

When a server wishes to destroy a CORBA object, for example, in the body of `Foo::destroy()`, it invokes `delete_object()` on the `GenericFactory` object in the replication manager. The replication manager then invokes `delete_object()` on the `GenericFactory` objects in the server replicas so that each member of the object group can be destroyed.

18.2.3 Registering Server Replicas with CORBA-FT

The mainline of a CORBA-FT server should create one or more `GenericFactory` objects and export their object references to, say, a file or the Naming Service.

Developers need to write a utility (or perhaps use a proprietary utility provided with a CORBA-FT implementation) that registers properties for the fault tolerant application with the replication manager. Some of these properties will be details of factories and their locations for each IDL interface type. This requirement to register factories is the reason why each CORBA-FT server needs to be able to export an IOR for its own factories.

Having registered a system's properties with the replication manager, a utility could then invoke `create_object()` on the replication manager to create an object group (this is assuming that the application uses the `MEMB_INF_CTRL` membership style). The IOGR obtained from this can then be made available to clients via, say, the Naming Service.

18.3 CORBA-FT Support in Clients

An IOGR contains an embedded `TaggedComponent` (Section 10.2.3 on page 108) that indicates the object reference is for an object group rather than for a "normal" object. When the client-side CORBA runtime system encounters this `TaggedComponent` then it enables client-side CORBA-FT capabilities that enhance the end-to-end fault tolerance of the system. Clients built with a non-FT implementation of CORBA ignore the `TaggedComponent.` Such clients can still communicate with a CORBA-FT server, but they are not able to take advantage of all the fault tolerance capabilities provided by CORBA-FT.

18.3.1 Keeping IOGRs Up to Date

The `TaggedComponent` embedded in an IOGR records a version number for the IOGR. Whenever the set of members for an object group changes (for example, when a member dies or is restarted), the replication manager updates the version number of the IOGR and notifies infrastructure in CORBA-FT servers of the updated version number. Each time a CORBA-FT client makes a remote call, a *service context* (Section 11.6 on page 119) is used to transmit the IOGR's version number along with the request. The CORBA-FT infrastructure in the server compares the version number in the received service context to the version number that it has.

- If the client's version number matches the server's version number then this means that the client has an up-to-date IOGR and so the incoming request is dispatched as normal.

- If the client's version number is less than the server's version number then this means that the client has an out-of-date IOGR. In this case, the CORBA-FT infrastructure in the server does *not* dispatch the incoming request. Instead, it sends a redirection message (Section 11.4 on page 117) back to the client to provide the client with an up-to-date IOGR. The CORBA runtime system in the client then resends the request using the new IOGR.

- If the client's version number is larger than the server's version number then this means that the server's version is out of date. In this case, the CORBA-FT infrastructure in the server contacts the replication manager to obtain the most recent version of the IOGR before dispatching the incoming request.

The purpose of this version number protocol is to increase the likelihood that an IOGR held by a client contains contact details for only the currently-alive members of the object group. By omitting contact details for currently-dead members from the IOGR, CORBA-FT reduces the likelihood that clients will waste time trying to communicate with currently-dead members.

18.3.2 Making Sure Clients Invoke on Primary Members

If an object group uses the `COLD_PASSIVE` or `WARM_PASSIVE` replication styles then one of the profiles (sets of contact details) in its IOGR contains a `TaggedComponent` (Section 10.2.3 on page 108) to indicate which is the primary member of the object group. Ideally, the CORBA runtime system in a client should use this `TaggedComponent` as a strong hint regarding to which member of the object group it should send its requests. However, the client will ignore this hint if the client has been built with a non-CORBA-FT product. Even if the client *has* been built with a CORBA-FT product, the hint might be out of date, because there might have been a failover that resulted in a different member of the object group becoming the new primary member.

If a request is sent to a backup member of an object group then CORBA-FT infrastructure in the backup server uses a redirection message (Section 11.4 on page 117) to redirect the client to the primary member.

18.3.3 Transparent Retries of Failed Invocations

As will be discussed in Section 18.4, CORBA-FT infrastructure in servers log all request and reply messages. A CORBA-FT client transmits a `FT_REQUEST` service context with each request. This service context contains a string that uniquely identifies the client, an integer that uniquely identifies the request (within that client) and an expiration time. When a server receives a request that contains a `FT_REQUEST` service context, the CORBA-FT infrastructure checks to see if there a request message with an identical service context in the log. If there is then the incoming request is not executed; instead, the corresponding reply message from the log is transmitted back to the client. The purpose of this mechanism is to allow the runtime system of a CORBA-

FT client to perform automatic retries of failed invocations while preserving at-most-once invocation semantics.

18.3.4 Heartbeat Messages

TCP/IP (upon which IIOP is based) does not cope very well with some types of disruptions to a network. For example, if a client process is connected to a server process via TCP/IP then the client will be notified promptly if the server process dies but will *not* be notified promptly if the server's machine fails or is abruptly disconnected from the network. In such cases, the client process may wait for a long time (perhaps even forever) for a reply from the server. This problem can be solved by setting round-trip timeouts in the client. However, doing this for each request can be laborious, even if you know approximately how long a particular invocation should take.

CORBA-FT provides an alternative mechanism for detecting network failures in a timely manner. This mechanism involves the CORBA-FT infrastructure in a client periodically sending ping-style messages to CORBA-FT servers. When a CORBA-FT server receives one of these messages (called *heartbeat* messages in CORBA-FT terminology), the CORBA-FT infrastructure in the server does *not* dispatch the request to an object. Instead, the CORBA-FT infrastructure itself sends back a reply to the client. The intention of avoiding a dispatch to the target object is that the heartbeat messages are to check network connectivity only rather than whether or not an object is alive.

Whether or not heartbeat messages are transmitted is determined by a client-side *policy* (Section 16.1 on page 145). The policy also specifies the frequency with which heartbeat requests are transmitted and the timeout period for receiving a reply.

18.4 Logging and Recovery Infrastructure

Each CORBA-FT server has some built-in infrastructure called the *logging mechanism* and *recovery mechanism*. These pieces of infrastructure are provided by a CORBA-FT implementation, but CORBA-FT does *not* define IDL interfaces for them because they are not invoked directly by application code.

The logging mechanism is responsible for persistently logging the request messages that arrive for an object and the reply messages sent after invocations have completed. It also logs the periodic checkpoints (and possible updates) of the primary member of an object group. The log can be accessed in a distributed manner. The details of how this is achieved is an implementation detail, but one possible way is for the log to be written to a replicated database; another possible technique is for each host to maintain the log in local volatile storage and use a reliable, totally-ordered multicast protocol to send log updates to the other hosts.

If the `COLD_PASSIVE` or `WARM_PASSIVE` replication style is used and the primary member of an object group dies then a backup member is promoted to be the primary member. When this happens, the recovery mechanism is used to bring the state of that member up-to-date. The recovery mechanism is also used in the `COLD_PASSIVE` or `ACTIVE` replication styles when a new member is introduced to the group.

When the recovery mechanism is used, it analyses the log and calls `set_state()` on the relevant object member to initialize it to the last checkpoint state. Then it might call `set_update()` to load the most recent "update" to the object. Finally, it re-invokes the request messages that are more recent than the last checkpoint/update to bring the object fully up to date.

In order to conserve space, the logging mechanism compacts the log periodically. In particular, whenever a new checkpoint of an object is obtained, previous checkpoints and request/reply messages that are older than the new checkpoint can usually be removed from the log. Likewise, whenever a new update is obtained by calling `get_update()` then older updates and request/reply messages can usually be removed from the log. The only exception to this is that some request/reply messages may be retained if the expiration time in the `FT_REQUEST` service context in the request has not yet occurred. This is to support the transparent retry of failed invocations discussed in Section 18.3.3.

18.5 Fault Notifiers

An implementation of CORBA-FT contains some infrastructure called a *fault monitor* or *fault detector*—these two terms are used interchangeably within the CORBA-FT specification. A fault monitor has the task of detecting faults. CORBA-FT does not define an IDL interface for the fault monitor because it is not invoked directly by users. To aid scalability, there may be several fault monitors within a CORBA-FT deployment: typically one on each host where CORBA-FT servers run. Fault monitors can check for faults by invoking the `is_alive()` operation on object members.

When a fault monitor detects a fault, it needs to report it to the replication manager and possibly also to user-written applications that might, for example, analyze reports and show summaries of them on a graphical display. The OMG decided that the mechanism used to report faults should be based on concepts in the Notification Service (Section 22.3 on page 214). The Notification Service itself was considered to be complex enough that it would be difficult to implement all of its functionality in a fault tolerant way. Instead, the bare minimum subset of Notification Service-style functionality required for CORBA-FT was extracted and repackaged in the `FT::FaultNotifier` interface, which is shown in Figure 18.3.

```
module FT {
  ...
  interface FaultNotifier
  {
    typedef unsigned long long ConsumerId;
    void push_structured_fault(
          in CosNotification::StructuredEvent  event);
    void push_sequence_fault(
          in CosNotification::EventBatch       event);
    ConsumerId connect_structured_fault_consumer(
          in CosNotifyComm::StructuredPushConsumer
                                           push_consumer);
    ConsumerId connect_sequence_fault_consumer(
          in CosNotifyComm::SequencePushConsumer
                                           push_consumer);
    void disconnect_consumer(
          in ConsumerId  connection) raises(...);
    void replace_constraint(
          in ConsumerId                     connection,
          in CosNotification::EventTypeSeq event_types,
          in string                         constr_expr);
  };
};
```

Figure 18.3: The `FT::FaultNotifier` interface

Application code can obtain a reference to the fault notifier by invoking `resolve_initial_references("ReplicationManager")` to connect to the replication manager and then invoking `get_fault_notifier()` on it.

A fault monitor does *not* need to explicitly register with the fault notifier. Instead, once a fault monitor has a reference to the fault notifier, it can invoke `push_structured_fault()` to send a single fault event to the fault notifier. Alternatively, it can invoke `push_sequence_fault()` to send a sequence of fault events to the fault notifier.

An application that wants to receive fault events *does* need to register with the fault notifier. It can do this by invoking `connect_structured_fault_consumer()` if it wants to receive fault events one-at-a-time. If an application prefers to receive fault events batched up in a sequence then it invokes `connect_sequence_fault_consumer()`. Both of these operations return a `ConsumerId` that is later passed as a parameter to `disconnect_consumer()` to disconnect the consumer from the fault notifier.

By default, a consumer of fault events receives *all* the events. However, once connected, a consumer can invoke `replace_constraint()` to specify a constraint that the fault notifier uses to filter out unwanted events (Section 22.3.4.1 on page 218).

The CORBA-FT specification gives precise details of the information that is contained inside a fault event. The information includes the *location*, IDL interface *type* and *object group id* of the object that failed.

The replication manager registers itself with the fault notifier as a consumer of fault events. Developers can write their own applications that also register with the fault notifier as consumers of fault events. Such applications, for example, might analyze faults (based on location, frequency and so on) or show summaries of faults on a graphical display. A fault monitor does not know how many consumers are connected to the fault notifier. A fault monitor simply pushes its fault events to the fault monitor and, in turn, the fault monitor pushes the fault events to all registered consumers.

18.6 Critique

CORBA-FT defines a *lot* of infrastructure. Because of this, at first sight it appears that CORBA-FT is very complex. However, most of this infrastructure is pre-implemented by a CORBA-FT product and it works behind the scenes. Only a relatively small amount of the infrastructure is visible to, or must be implemented by, developers. This means that CORBA-FT is not as complex as it first seems. However, there is no doubt that CORBA-FT *is* more complex (and also more powerful) than the proprietary fault tolerance mechanisms discussed in Chapter 17.

Use of CORBA-FT affects the design and coding of applications. Because of this, it is best if CORBA-FT is designed into an application from the start, rather than being retrofitted to an existing application as an afterthought. In contrast, most proprietary fault tolerance mechanisms are enabled via configuration rather than via coding. This means that, quite often, use of a proprietary fault tolerance mechanism can be retrofitted to an existing application quite easily.

Perhaps the biggest drawback of CORBA-FT is that it is an optional part of the CORBA specification and, unfortunately, most CORBA products neglect to implement it. The author is aware of only one CORBA implementation, TAO, that currently implements CORBA-FT; of course, there could be other CORBA-FT implementations of which the author is not aware.

Chapter 19

Other CORBA Infrastructure

This chapter briefly mentions some other optional parts of CORBA infrastructure that have not been discussed in this book. Interested readers can find details in the CORBA specification (available from `www.omg.org`) and in manuals for products that support these capabilities.

19.1 Real-time CORBA

As its name suggests, the Real-time CORBA specification is CORBA with several extensions that make it suitable for use in real-time applications. The author is not aware of any books that specifically address real-time CORBA programming. There is, of course, the specification document for Real-time CORBA, which is available from the OMG web site, and any CORBA product that implements this standard is likely to have additional information in its manual.

19.2 CORBA for Embedded Systems

The core part of CORBA has so many capabilities that an implementation of CORBA is likely to occupy several megabytes of memory. While modern desktop computers have sufficient RAM to run CORBA-based applications, many embedded systems have much smaller amounts of memory. In order to tailor

CORBA for the constrained RAM of embedded devices, several CORBA vendors offer the ability to "subset" CORBA functionality so that if an CORBA-based application uses just a subset of CORBA's functionality then the application can be linked with a proportionally smaller CORBA library. Some vendors have managed to squeeze a useful subset of CORBA into less than 100KB.

The *Minimum CORBA* specification is an OMG-standardized subset of CORBA functionality that is thought to be useful in embedded systems. The intention of defining a standardized subset of CORBA functionality is that it provides source-code portability for applications that use a subset of CORBA. However, the Minimum CORBA specification has received criticism, as some feel that the subset it defines is still too large for many embedded systems. For this reason, CORBA vendors who specialize in embedded systems usually offer their own proprietary subsets of CORBA functionality in order to minimize memory requirements.

19.3 CORBA Component Model (CCM)

In recent years, there has been a growing realization that many applications contain not just "business logic" code but also a lot of "infrastructure code" to perform tasks such as security, transactions and persistence of data. In fact, the amount of infrastructure code in an application often outweighs the amount of business-logic code. Needless to say, this is an unfortunate imbalance because the need to write vast quantities of infrastructure code increases the cost of application development.

One of the important capabilities of J2EE (Java 2, Enterprise Edition) is an *application server*: this specializes in providing infrastructure services that are commonly used with business-logic code. The intention is that developers can focus their efforts on writing business-logic code without having to worry about the supporting infrastructure code. The business-logic code is packaged into a JavaBean[1] and this can then be deployed into a J2EE application server. The deployment is achieved by having the J2EE application server read a *deployment descriptor*, which is an XML document that specifies which JavaBean class should be dynamically loaded, and which infrastructure services (such as transactions or security) should be applied to it.

[1] A lot of terminology associated with the Java programming language is based on terminology associated with coffee, without any regard for whether or not the terminology has an obvious meaning in computers. The term *JavaBean*, which is a contraction of *Java coffee bean*, has no self-evident meaning for its use in computers. A JavaBean is a class that encapsulates some business-logic code.

The CORBA Component Model (CCM) is a generalization of the concept of a J2EE application server. It is a generalization because CCM aims to provide application server functionality for business-logic code that is written in an arbitrary language, not just Java. CCM is part of the newest CORBA specification (3.0). There are several implementations of CCM available—some from CORBA vendors and some from third-party companies—but it is still too early to predict whether CCM will achieve a lot of popularity or if it is destined to be a niche product area. A very readable and detailed overview of CCM can be found in the *Pure CORBA* book [Bol01].

Part V

CORBA Services

Chapter 20

Trading Service

CORBA provides several ways for a server to advertise an object reference. One way, discussed in Section 3.4.2 on page 34, is for the server to stringify an object reference and write it to, say, a file. Another approach is for the server to advertise the object reference in the Naming Service (Chapter 4). This chapter discusses a third approach, called the Trading Service.

Just as the Naming Service is often compared to the white pages telephone book, the trading service is often compared to the yellow pages telephone book. The yellow pages contains numerous *advertisements* organized into different *categories* (such as *Builders*, *Plumbers* and *Restaurants*), while the Trading Service contains numerous *service offers* that are organized by their *service offer types*. Each advertisement in the yellow pages provides contact details (address and telephone number) for a company along with a description of the company (for example, "open 24 hours" or "cheapest prices in town"). Similarly, each *service offer* in the Trading Service provides contact details (an IOR) along with a description of the service offered by the object.

20.1 The `ServiceTypeRepository` Interface

Figure 20.1 shows a simplified version of the `ServiceTypeRepository` IDL interface. There are several simplifications in this interface, and the interfaces shown later in this chapter:

- The `raises` clause on operations has been omitted.
- Some `typedef` statements have been removed. For example, the `name` parameter to `add_type()` is shown as being of type `string`. In

the real IDL definition, it is actually a `typedef` of a `typedef` of a `string`.

- Some of the parameters of operations and fields of structs are shown as being of an *anonymous sequence* type, such as `sequence<string>`. Use of anonymous sequences is illegal for parameter types and is deprecated for the types of fields in structs. The use of these anonymous types was done in order to avoid using extra `typedef` statements, and so to make the IDL listings more concise.

As mentioned earlier, the yellow pages telephone book groups the details of different companies into categories such as *electricians* and *plumbers*. The Trading Service equivalent of a "category" is a `CosTradingRepos::ServiceTypeRepository::TypeStruct`, although this is normally referred to as a *service offer type*. The `ServiceTypeRepository` interface is used to define service offer types.

20.1.1 The `add_type()` and `remove_type()` operations

The `add_type()` operation is used to define a new service offer type. The `name` parameter specifies the name of the category, for example, `"Printer"`. The `if_name` parameter specifies a *repository id* (Section 9.4 on page 102) for the IDL interface associated with this service offer type. Some readers may wonder why the service offer type's `name` is not hard-coded to be the same as the repository id for its interface. The reason is to allow the `name` to be easier to read than a repository id. For example, `"Printer"` is easier to read than `"IDL:acme.com/Equipment/Printer:1.0"`. Inheritance is, of course, allowed. For example, the repository id in the previous sentence denotes an IDL interface called `Equipment::Printer`. An object of this type, *or* a sub-type of it, can be used in a `"Printer"` service offer.

The `props` parameter specifies a sequence of properties that are associated with the service offer type. This is where the analogy between the Trading Service and the yellow pages telephone book starts to break down. Some advertisements in the yellow pages may boast claims about a company such as "open 24 hours", "all work has a 12-month guarantee" or "cheapest prices in town". However, the yellow pages does *not* make any requirements about what claims *should* be made in an advertisement in the yellow pages. In contrast, a service offer type in the Trading Service *does* specify what *properties* ("claims") each *service offer* ("advertisement") should have. Each property (specified by the `PropStruct` type) has a name, a type[1] (such as `long`, `boolean` or

[1] The type is specified as a `TypeCode`, which is discussed in Section 15.2 on page 139.

```
module CosTradingRepos {
  interface ServiceTypeRepository {
    enum PropertyMode {
              PROP_NORMAL, PROP_READONLY,
              PROP_MANDATORY, PROP_MANDATORY_READONLY};
    struct PropStruct {
        string          name;
        CORBA::TypeCode  value_type;
        PropertyMode    mode;
    };
    typedef sequence<PropStruct> PropStructSeq;
    struct IncarnationNumber {
        unsigned long  high;
        unsigned long  low;
    };
    struct TypeStruct {
        string            name;
        PropStructSeq     props;
        sequence<string>  super_types;
        boolean           masked;
        IncarnationNumber incarnation;
    };
    ...
    readonly attribute IncarnationNumber incarnation;
    IncarnationNumber add_type(
                  in string         name,
                  in string         if_name,
                  in PropStructSeq  props,
                  sequence<string>  super_types)
                           raises(...);
    void remove_type(in string name) raises(...);
    void mask_type(in string name) raises(...);
    void unmask_type(in string name) raises(...);
    ...
  };
};
```

Figure 20.1: Pseudo IDL Extract of ServiceTypeRepository

`string`) and a mode. The `mode` field is an `enum` that specifies whether or not the property is mandatory, and whether or not it is readonly.

The `ServiceTypeRepository` does not place any restriction on the types of properties. However, it is usually best to stick to using integers, floating point numbers, booleans, characters, strings and sequences of these types. The reason for this is that the constraint language used to make queries on the Trading Service provides full support for these types and only minimal support for other types.[2]

The `super_types` parameter lists the names of parent service offer types. A service offer type inherits the properties of its parents and is allowed to impose more restrictions on inherited properties. In particular, an optional property can be made mandatory, and a "normal" property can become readonly.

The return value of `add_type()` is an `IncarnationNumber`, which was intended to be a 64-bit integer. However, the `unsigned long long` IDL type did not yet exist when the Trading Service was being defined so a `struct` containing two `unsigned long` fields was used instead. The `IncarnationNumber` is conceptually a timestamp to indicate when a service offer type was defined. A list-style operation (not shown in Figure 20.1) can be used to list the service offer types defined since a particular incarnation number.

The `remove_type()` operation removes a service offer type.

20.1.2 The `mask_type()` and `unmask_type()` operations

The `mask_type()` operation is used to *mask* (hide) a service offer type. This has two uses. One use is to deprecate a service offer type, so that the Trading Service will not accept any more *service offers* (advertisements) for the specified service offer type. Another use is to indicate that the service offer type is an *abstract base type* that cannot be instantiated directly, but from which other service offer types can inherit. The `unmask_type()` operation unmasks a type that was previously masked with `mask_type()`.

20.2 The `Register` Interface

When a service offer type has been defined, it is then possible for applications to actually *export* (advertise) an object reference in the Trading Service. This is done through the `Register` IDL interface (Figure 20.2).

[2] If a property is of a user-defined type then a constraint can check whether or not the property exists in a service offer, but the constraint cannot examine the value of the property.

```
module CosTrading {
  struct Property {
    string  name;
    any     value;
  };
  typedef sequence<Property> PropertySeq;
  struct Offer {
    Object       reference;
    PropertySeq  properties;
  };
  typedef sequence<Offer> OfferSeq;
  interface Register
          : TraderComponents, SupportAttributes
  {
    string export(
                in Object       reference,
                in string       type,
                in PropertySeq  properties) raises(...);
    void withdraw(in string offer_id) raises(...);
    void modify(
                 in string            offer_id,
                 in sequence<string>  del_list,
                 in PropertySeq       modify_list
             ) raises(...);
    ...
  };
};
```

Figure 20.2: Pseudo IDL Extract of Register

20.2.1 The export() and withdraw() operations

The export() operation is used to create a *service offer*, that is, an advertisement for an object. Parameters to this operation specify an object reference, the service offer type that it matches, and its properties. The supplied properties must match those specified by the service offer type. The return value from export() is a unique string that denotes an offer id. This offer id can later be used to *withdraw* the advertisement or *modify* some of its (non-readonly) properties.

The withdraw() operation is used to withdraw (that is, delete) a service offer.

20.2.2 The `modify()` operation

The `modify()` operation is used to delete or modify some properties associated with a service offer. An exception is thrown if an attempt is made to delete a mandatory property. Likewise, an exception is thrown if an attempt is made to modify a readonly property.

20.3 The `Lookup` Interface

The `Lookup` interface (Figure 20.3) has just one operation, called `query()`. This operation is used to retrieve service offers (advertisements) from the Trading Service that match a specified constraint.

The `constraint` parameter is a boolean expression that refers to the properties of service offers. For example, let us assume that a `Printer` service offer type defines an integer property called `resolution` and also a `sequence<string>` property called `languages`. The following constraint can be used to obtain a list of the printers that have a resolution of at least 600 and support the PostScript printer language:

```
resolution >= 600 && "PostScript" in languages
```

The `in` operator tests if the value specified on the left (`"PostScript"`) is in the sequence specified on the right (`languages`). Constraints are specified in the syntax of the *Trader Constraint Language* (TCL), which is defined as part of the Trading Service specification.

There may be several service offers that are matched by the specified constraint. Details of these are returned through several `out` parameters. In particular, the `offers` parameter provides details of the first `how_many` matched service offers. If there are more then these can be accessed by invoking operations on the returned `offer_iter` *iterator* (Section 1.4.2.3 on page 10) object reference. The application that performs the query may be interested in seeing some of the properties of the matched service offers. The `desired_props` parameter is used to specify which properties should be accessible through `offers` and `offer_iter`.

The `preference` parameter is used to specify an ordering of the obtained service offers. For example, `"max resolution"` orders the offers by the resolution of the printers, while `"random"` provides the service offers in a random order, which might be useful if you want to use the Trading Service to load-balance many clients over several server processes.

```
module CosTrading {
  struct Property {
    string  name;
    any     value;
  };
  typedef sequence<Property> PropertySeq;
  struct Offer {
    Object reference;
    PropertySeq properties;
  };
  typedef sequence<Offer> OfferSeq;
  interface OfferIterator {
    boolean next_n(in  unsigned long  n,
                   out OfferSeq       ids) raises(...);
    ...
  };
  interface Lookup
          : TraderComponents, SupportAttributes,
            ImportAttributes
  {
    enum HowManyProps {none, some, all};
    union SpecifiedProps switch (HowManyProps) {
      case some:  sequence<string>  prop_names;
    };
    ...
    void query(
                in string           service_type_name,
                in string           constraint,
                in string           preference,
                ...
                in SpecifiedProps   desired_props,
                in unsigned long    how_many,
                out OfferSeq        offers,
                out OfferIterator   offer_iter,
                ...) raises(...);
  };
};
```

Figure 20.3: Pseudo IDL Extract of Lookup

20.4 Other Capabilities of the Trading Service

The Trading Service has several other important capabilities that I briefly mention here.

It is possible to join several Trading Services so that a query made to one Trading Service can propagate through the other Trading Services too. The Trading Service specification refers to this as *linking* or *federating* several Trading Services.

Most properties in a service offer will have static (that is, unchanging) values. For example, the `resolution` of a `Printer` is normally unchanging. However, the Trading Service allows for a property to be *dynamic*. A dynamic property is implemented with a reference to a *callback object* (Section 1.4.2.2 on page 10). When carrying out a query, the Trading Service invokes upon the callback object to obtain the current value of the desired property. Dynamic properties can be useful to denote, say, the `queue_length` of a `Printer`.

Further discussion of the above-mentioned capabilities is outside the scope of this chapter. Interested readers can find a more detailed discussion in other books [HV99, BVD01].

20.5 Using the Trading Service

The functionality of the Trading Service is spread over many IDL interfaces. This may lead you to think that there is a lot of programming involved in using the Trading Service. However, this need not be the case, as I now discuss.

Although it is not required by the Trading Service specification, each vendor is likely to provide command-line utilities and/or a graphical program that encapsulates the functionality of the IDL interfaces. These command-line utilities and/or graphical programs make it possible to do most of the administration of the Trading Service without having to do any coding. In particular:

- A command-line utility or graphical program can be used to add, remove, mask, unmask and browse service offer types. This saves you from having to write code that interacts with the `ServiceTypeRepository` interface.

- You *could* hard-code a server application to `export()` a service offer to the Trading Service. However, an alternative that involves much less coding is to have the server write a stringified object reference to a file. You can then use a command-line utility or graphical program to import this IOR, construct a service offer from it, and `export()` this service offer to the Trading Service.

This then reduces the coding burden to just applications that call `query()` on the `Lookup` interface. If an application has an interactive user then the user may wish to view the properties of the returned offers and manually choose one. For client applications in which interactive selection of returned offers is undesirable, the client application could randomly choose one of the returned offers. In fact, a command-line utility could be written that performs a query (specified as a command-line argument) and then prints out a randomly-chosen IOR from the returned offers. If a client application imports object references by using the `importObjRef()` utility function (Section 4.3 on page 43) provided in the *CORBA Utilities* package [McH, Ch. 2] then the client need not be hard-coded to invoke the APIs of the Trading Service. Instead, the client can pass an `instructions` parameter to `importObjRef()` that tells it to execute the command-line utility and interpret its standard output as an stringified object reference.

Applications connect to the `Lookup` interface of the Trading Service by calling `resolve_initial_references("TradingService")`. All the functional interfaces of the Trading Service inherit from some base interfaces. These base interfaces provide readonly attributes that provide access to the other interfaces of the Trading Service. So, for example, once an application uses `resolve_initial_references()` to connect to the `Lookup` interface, the application can then invoke an attribute to navigate to the `Register` or `ServiceTypeRepository` interface.

20.6 Quality of Service

The Trading Service specification does not make any requirements about how an implementation stores details of service offer types and service offers.

Some implementations of the Trading Service store information only in RAM. These implementations are useful in embedded systems that do not have persistent storage, but are less desirable in computers that do have persistent storage because you would have to re-populate the Trading Service every time it dies and is restarted.

Some implementations of the Trading Service that store information persistently make use of plain files. This is fine for a small deployment but it may not scale well, and there is the possibility of information being corrupted if the Trading Service is killed *while* updating information in a file.

Some other implementations of the Trading Service store information in a database. This provides greater reliability than use of plain files but may involve extra administration overhead.

The functionality of the Trading Service specification is split over many IDL interfaces, and the Trading Service specification does *not* require that all the interfaces be implemented. Instead, some of the functionality of a Trading Service is optional. This makes it possible for a vendor to make trade-offs between how much functionality is provided by an implementation of the Trading Service and the amount of resources (RAM, disk space, CPU speed and so on) that it consumes.

If you decide to use a Trading Service in your applications then it is important to ask your Trading Service vendor how it stores data and whether it implements *all* the functionality of the Trading Service or a particular subset of functionality. Be sure to pick a Trading Service that offers a quality of service that is suitable for your needs.

Chapter 21

Object Transaction Service

21.1 Associating CORBA Objects with Database Records

When designing a CORBA server that interacts with a database, you might decide to have a separate CORBA object for each record in a database table. To do this, you need to associate a CORBA object with the corresponding record in the database. This is easily achieved by storing the *primary key* of the database record in the *object id* (Section 5.6.1) of the CORBA object.

You could have a separate *servant* (Section 5.2) for each CORBA object/database record. If you do this then you might wish to store the object id in an instance variable of the servant. However, it is unusual for servants to cache any information from the database in instance variables. This is because of the risk that the cached information would become stale if the database was ever updated directly rather than being updated via the CORBA server. Having a separate servant for each record in a database table, where the *only* instance variable in each servant is an object id (acting as a primary key into a database table) is wasteful of memory. A more scalable approach is to use one *default servant* (Section 5.6.4 on page 55) to represent *all* the CORBA objects. The default servant can call the `get_object_id()` operation on the *POACurrent* (Chapter 13) to find out which CORBA object it is representing for the current request. The default servant then uses this object id as the primary key into a database table.

21.2 Per-operation Transactions

In many client-server systems, the body of IDL operations are implemented as shown in the pseudocode below:

```
void some_operation(...)
{
    begin_transaction();
    ... // query or update a record in the database
    commit_transaction();
}
```

Note that the entire transaction is contained within the body of a single IDL operation. This is often called a *per-operation* transaction. If you intend to write a CORBA server whose use of transactions is restricted to per-operation transactions then how you interact with the database is completely independent of your use of CORBA. For example, you can use whatever brand of database you want, and you can interact with that database using whatever techniques you want, such as embedded SQL, Oracle OCI, ODBC or JDBC.

The use of per-operation transactions is sufficient for a great many applications. However, some client-server systems require the ability for a transaction to span *multiple* operation calls from a client to one server. This type of interaction requires use of the CORBA Object Transaction Service (OTS). Some other client-server interactions involve access to multiple databases. These types of transactions are usually called *distributed transactions*, and they also require use of OTS.

21.3 Overview of Distributed Transactions

The concept of a distributed transaction pre-dates CORBA, and is independent of any one particular kind of middleware. Let us assume that a client wishes a transaction to span queries and updates on two databases. In order to do this, the client makes use of a transaction manager (TM), as shown in Figure 21.1.

When the client wants to begin a transaction, it sends a request to the TM (step 1). The TM sends back an identifier that uniquely identifies the newly-started transaction. The client then sends its query and/or update requests to the databases or, in a middleware system like CORBA, to the server processes that wrap the databases; this is shown in steps 2 and 3. The transaction identifier is transmitted as part of these requests. Whenever a database (or a server process that wraps the database) is accessed by the client, the database (server) uses the received transaction identifier to tell the TM that it (the database/server)

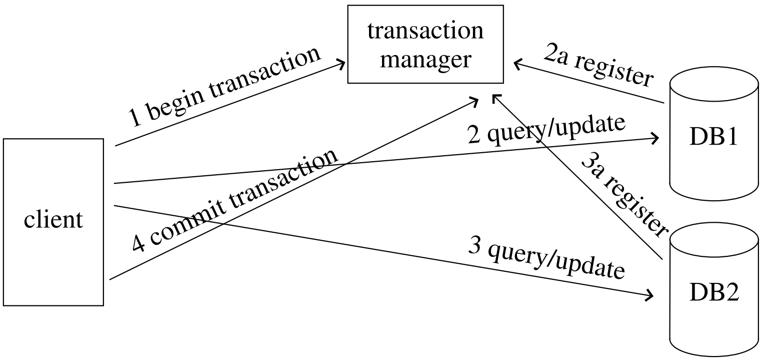


Figure 21.1: A distributed transaction

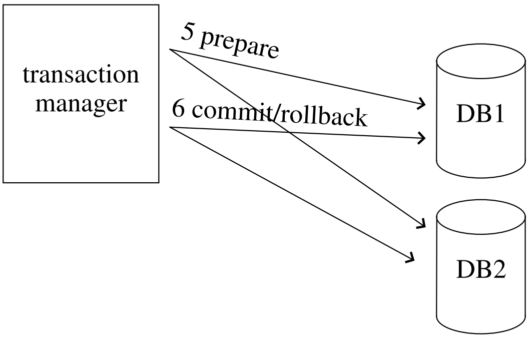


Figure 21.2: Two-phase commit

is taking part in the transaction (steps 2a and 3a). The TM uses a persistent storage area (for example, a file or its own database) to keep track of which databases/servers are taking part in which transactions. Finally, the client notifies the TM that it wishes to commit the transaction (step 4). To avoid too much clutter in a single diagram, the rest of the interaction steps are shown in Figure 21.2. The TM engages in a *two-phase commit* dialog with all the databases/servers that have taken part in the transaction.

In the first phase of the two-phase commit protocol, the TM asks each database/server to *prepare* to commit. Each database/server prepares itself by persisting any updates to disk, but in a way that it can *undo* the changes (for example, the database might persist not just changes but also details of how to undo the changes). The database/server then replies to the TM that it is voting to either *commit* or *rollback* the transaction.

In the second phase of the two-phase commit protocol, the TM analyses all the votes from the databases/servers. If *all* the votes are to *commit* then the TM instructs each database/server to commit its changes. If *any* of the votes were to *rollback* the transaction then the TM instructs *all* the databases/servers to undo any changes made during the prepare phase. Finally, the TM can delete details about the just-completed transaction from its own persistent store.

If the system ever crashes then when the system is restarted, the TM examines its persistent storage of details about transactions in progress. Using this information, the TM can instruct databases/servers to *rollback* any transactions that had not reached the commit phase before the crash occurred. The TM can also replay the final outcome of the two-phase commit protocol for any transactions that were in the process of committing when the crash occurred.

The Open Group (`www.opengroup.org`), which is sometimes referred to as X/Open, has defined some open standards for distributed transactions. One of these standards, called XA, is a C-based API that a transaction manager can use to interact with a *resource manager* (most commonly a database) to implement the two-phase commit protocol. The importance of this standard is that it allows a transaction manager to coordinate a distributed transaction not just across multiple databases, but also across multiple *brand names* of databases. Section 21.4 will discuss how CORBA leverages the XA standard to allow CORBA applications to take part in distributed transactions. However, before discussing that, there are two other points worth noting.

First, when designing a client-server system, it is comforting to know that the middleware technology you are using makes it *possible* to use distributed transactions if the need arises. However, usually it is preferable to design the system in a way that uses only per-operation transactions rather than distributed transactions. One reason for this is that recovery after a crash during a local transaction is much less burdensome than recovery after a crash during a distributed transaction. Another reason is that local transactions tend to be short-lived so database locks are held for a minimum amount of time. Such short-lived transactions increase the possibility for concurrent access of the database and so promote system scalability. In contrast, distributed transactions tend to be longer-lived, *especially* if they involve user input, and this can limit concurrent access and scalability. A final reason is that distributed transactions have a higher overhead than local transactions (such as the overhead of logging and the extra communication required by the two-phase commit protocol).

Second, the *Enterprise CORBA* book [SGR99] devotes almost 100 pages to discussing the use of databases in client-server applications. If you plan on developing a client-server system that utilizes a database then you are advised to read that book for its wealth of useful design advice.

21.4 CORBA Object Transaction Service (OTS)

The *Object Transaction Service* (OTS) is a CORBA service (Section 1.6) that enables the use of distributed, two-phase commit transactions in CORBA applications. OTS consists of: (1) several IDL interfaces (most of which are defined in a module called `CosTransactions`), (2) some additional library code that is linked into client and server applications, and (3) a transaction manager. At first sight, the OTS specification can appear to be overly complex. There are two reasons for this.

The first reason for the apparent complexity of OTS is because the OTS specification defines not just the API that is used by "normal" developers; it *also* defines the lower-level "plumbing" API that is used by vendors to *implement* OTS. The reason why the OTS specification defines the plumbing API is that doing so ensures interoperability between different implementations of OTS. This means that an OTS client built with one CORBA product can take part in distributed transactions with OTS servers that are implemented with different CORBA products.

The second reason for the apparent complexity of OTS is because its API is flexible enough to allow transactional applications to be written in several different ways. Most developers use a simple API that allows them to focus on application-level logic, while leaving OTS to automatically perform several house-keeping tasks. However, OTS does allow developers to take a more hands-on approach and *manually* handle the required house-keeping tasks. The fuller set of "hands-on" APIs makes it possible for developers to integrate OTS with non-XA-compliant databases or to implement bridges from OTS to a non-CORBA distributed transactional system.

Most of the OTS API is shown in Figure 21.3. A few details outside the scope of this chapter have been omitted from this figure. Also, the `raises` clause on operations have been omitted for brevity. Section 21.5 discusses this "raw" API of OTS. Then Section 21.6 discusses how OTS provides a simplified API by building on top of other functionality in CORBA.

```
interface TransactionFactory {
  Control create(in unsigned long time_out);
  ...
};
interface Control {
  Terminator  get_terminator();
  Coordinator get_coordinator();
};
interface Terminator {
  void commit(...);
  void rollback();
};
interface Coordinator {
  RecoveryCoordinator register_resource(in Resource r);
  ...
};
interface RecoveryCoordinator {
  Status replay_completion(in Resource r);
};
interface Resource {
  Vote prepare();
  void rollback();
  void commit();
  ...
};
local interface Current : CORBA::Current {
  void begin();
  void commit();
  void rollback();
  void set_timeout(in unsigned long seconds);
  unsigned long get_timeout();
  Control get_control();
  Control suspend();
  void resume(in Control which);
};
```

Figure 21.3: A subset of the OTS APIs

21.5 The Raw API of OTS

The `Resource` interface is a CORBA "wrapper" around a resource (database). The operations defined on this interface are similar to the C-based API of the XA standard. Implementations of OTS provide an implementation of the `Resource` interface that trivially delegates to the underlying XA C-based API.[1] This means that a server developer gets trivial integration between OTS and an XA-compliant database. If server developers are using a database that is *not* XA-compliant then they will have to implement the `Resource` interface for that database.

An implementation of OTS provides a transaction manager (TM). The specification does not state if this should be packaged as, say, a server process or as a library that can be linked into another application. However, it is common for the TM to be a stand-alone server process. Regardless of how it is packaged, the TM contains pre-written implementations of several interfaces: `TransactionFactory`, `Control`, `Terminator`, `Coordinator` and `RecoveryCoordinator`.

The CORBA specification does not state how an OTS client connects to the transaction factory in the TM, so the mechanism varies from one CORBA product to another. However, the connection is likely to be made by calling `resolve_initial_references()`. Use of this operation is discussed in Section 3.4.1 on page 33. The OTS client calls `TransactionFactory::create()` to begin a transaction. This operation returns a reference to a `Control` object.

The client must somehow communicate the `Control` object reference when invoking an operation on an OTS-aware object. This could be achieved by explicitly passing the `Control` reference as a parameter to the operation. However, it is more commonly achieved by embedding the `Control` reference (along with other information) in a *service context* (Section 11.6 on page 119) that is transmitted with the request. The OTS specification defines a service context structure for this purpose.

When the client wants to terminate a transaction, it calls `Control::get_terminator()` to obtain a reference to the `Terminator` object and then calls `commit()` or `rollback()` on this.

If an OTS server accesses an XA-compliant database then the server invokes an OTS operation (not shown in Figure 21.3) that puts a `Resource`

[1] Readers who wish to do Java-based OTS development may wonder how `Resource` interacts with a Java DataBase Connectivity (JDBC) driver. The answer is that JDBC drivers provide XA-compliant `DataSource` objects. An implementation of the `Resource` interface delegates to an underlying `DataSource` object.

wrapper around the database. If the server uses a non-XA-compliant database then the server developer must implement the `Resource` interface so that its database can take part in two-phase commit transactions.

In the original OTS specification, an object indicated that it could take part in OTS transactions by implementing an IDL interface that inherited from `CosTransactions::TransactionalObject`. The `_is_a()` operation (which is provided by the base `Object` type) was used by a client application to determine whether or not an object reference was for a transactionally aware object. However, the OMG decided that this approach was undesirable. In particular, it can result in a dramatic increase in the number of IDL interface definitions.[2] Eventually, the OMG decided that it would be be better if whether or not an object was transactionally aware could be expressed as a *quality of service*. In modern versions of the OTS, this goal is achieved by defining a new POA Policy type (Section 6.1.4 on page 64) that, if used, indicates that objects in that POA are transactionally aware. An IOR interceptor (Section 14.1 on page 133) detects the presence of this POA policy and embeds an OTS `TaggedComponent` (Section 10.2.3 on page 108) into IORs that originate from that POA. A client application can check for the presence of this `TaggedComponent` to determine if an object is transactionally aware.

When an operation in an OTS-enabled server receives a `Control` object, it can call `get_coordinator()` to gain access to the transaction's `Coordinator` object. The `Coordinator` interface is a "wrapper" around the coordination logic that implements the two-phase commit protocol. Its purpose is to interact with the `Resource` objects in OTS servers. The server calls `register_resource()` on the `Coordinator` to register its resource (this registration occurs only once per transaction). This informs the TM that the server's `Resource` is taking part in the transaction and so should be included in the two-phase commit protocol when the transaction commits. This operation returns a reference to a `RecoveryCoordinator` object for the transaction. The server stores this object reference in a persistent storage area so that if the server crashes during the two-phase commit protocol and is restarted then the server can contact the `RecoveryCoordinator` to determine if the transaction should commit or roll-back.

During the two-phase commit, the TM invokes the `prepare()` operation on all `Resource` objects that have taken part in the transaction. The return

[2] For example, the interfaces that define the Naming Service did *not* inherit from `CosTransactions::TransactionalObject`, which meant that an implementation of the Naming Service could not take part in a distributed transaction. To obtain an OTS-aware Naming Service would have required defining a new set of IDL interfaces that *did* inherit from `CosTransactions::TransactionalObject`.

value of this operation is a `Vote` that determines if the transaction will be committed or rolled back.

21.6 How OTS Builds on Top of Other Parts of CORBA

This section briefly discusses a simple subset of the API provided by OTS. This simple subset is used by most OTS developers. The focus of this discussion is *not* to act as a tutorial for developers, but rather to show how other aspects of CORBA (such as current objects, portable interceptors and service contexts) are used as building blocks for more powerful capabilities, such as OTS.

OTS defines a `Current` object (Chapter 13). This object is accessed by calling `resolve_initial_references("TransactionCurrent")` (Section 3.4.1 on page 33). The OTS Current object (defined in Figure 21.3 on page 202) lets threads in both client and server applications know with which transaction they are currently associated.

An OTS client uses the `begin()`, `commit()` and `rollback()` operations on the Current object to control the lifetime of a transaction. Internally, the Current object delegates to the corresponding operations defined on the interfaces in the transaction manager. When a client invokes an operation on an object, a portable request interceptor (Section 14.2 on page 134) provided by OTS embeds transactional context information obtained from the Current object in a service context (Section 11.6 on page 119) that is then transmitted with the request to the target object. A corresponding portable request interceptor in the server extracts this transactional context information from the service context and initializes the server's Current object before dispatching to the target operation. This means that the body of the operation executes within the context of a transaction. Because of this, the operation does *not* need to begin-and-commit or resume-and-suspend a transaction. Instead, these details are taken care of by the portable interceptor and so the body of the operation can focus on using, say, embedded SQL or JDBC to query/update the database.

The mechanism discussed above provides a simple API for developers and it is powerful enough for the majority of applications. However, developers *can*, if they so choose, avoid using the Current object and its associated portable interceptor, and instead manually execute their own OTS-infrastructure code. Although this is more complex, it provides a way for developers to integrate a non-XA-compliant database with OTS.

Chapter 22

Publish and Subscribe Services

22.1 What is Publish and Subscribe?

The default communication model in CORBA is a call from one client to (an object in) one server. This is often *one-to-one* or *point-to-point* communication. In contrast, *publish and subscribe* (often abbreviated to *pub-sub*) communication is where one application "publishes" (that is sends) a message on a particular *topic*, and *all* the other applications that have "subscribed" to this topic receive the message. This is a form of *one-to-many* communication, and it is intrinsically asynchronous because the application that publishes a message does not wait to get responses from those applications that receive the message.

Computer mailing lists are good analogy for pub-sub communication. For example, let us assume that the ACME company has mailing lists for staff in different departments: `eng-staff@acme.com` reaches the Engineering staff, `sales-staff@acme.com` reaches the Sales staff, and so on. The following points are worth noting:

- If you send an email to `eng-staff@acme.com` then you (a "publisher") have sent just *one* message, but it is received by *many* people (all the Engineering staff who are "subscribed" to the mailing list). This means that mailing lists provide a form of one-to-many communication.

- When you send an email, you do *not* wait for the email to be received by all the people subscribed to the mailing list. Instead, the email infrastructure delivers the messages in the background (and this background delivery might take hours or even days if there are network problems). This means that sending an email message to the mailing list is *asynchronous* communication.

- When you send an email, you do *not* wait for a reply. This means that email is *one-way* communication. Of course, a person who receives an email from you might send you another email in response. This shows that it is possible to *emulate* two-way communication with a pair of one-way messages, but the underlying email system is still intrinsically one-way.

- There are several different mailing lists at the ACME company. One person might send messages to both the `eng-staff@acme.com` and `sales-staff@acme.com` mailing lists. Some ACME employees might be subscribed to just one mailing list, while other employees might be subscribed to more than one mailing list. In an analogous way, a pub-sub system might have several different "topics" that an application can send messages to, or to which an application can subscribe.

- When you send an email to a mailing list, your message initially goes to the mailing list computer, and this computer then forwards on the message N times—once for each subscriber to the mailing list. Some pub-sub systems work in a similar way but some other pub-sub systems use a multicasting or broadcasting protocol so that a message is transmitted just once (rather than $N + 1$ times) and is received simultaneously by all the subscribers.

22.1.1 Emulating Different Communication Models

Some middleware systems, such as TIBCO Rendezvous, are based on pub-sub communication. Some other middleware systems, such as IBM MQ Series, are based on asynchronous, point-to-point communication. Some other middleware systems, such as CORBA, are based on synchronous, point-to-point communication.

Each of these three kinds of middleware can emulate the other two kinds. For example, CORBA can provide one-to-one, asynchronous communication with IDL `oneway` calls or CORBA Messaging (Chapter 16). And CORBA can provide pub-sub communication with the various CORBA Services discussed in this chapter.

22.1.2 CORBA Services for Publish and Subscribe

The CORBA *Event Service* (Section 22.2) provides a very basic form of pub-sub communication. The *Notification Service* (Section 22.3) extends the Event Service in ways that provide a much richer form of pub-sub communication. Finally, the *Telecom Log Service* (Section 22.4) extends the Notification Service with the ability to permanently log messages so that they can be replayed.

CORBA uses terminology that is different to what has been used in the discussion so far. Instead of *publisher* and *subscriber*, CORBA uses the terms *supplier* and *consumer*. Instead of *topic* or *mailing list*, CORBA uses the term *event channel*.

22.2 Event Service

The CORBA Event Service offers both a "push" and a "pull" model of communication. The push model is similar to how I have described pub-sub systems, so I discuss the push model first, in Section 22.2.1 and then discuss the pull model in Section 22.2.2.

22.2.1 The Push Model

Figure 22.1 shows the IDL definitions relevant to the push model of the Event Service. For conciseness, the `raises` clauses on operations have been omitted. Figure 22.2 shows graphically how the various interfaces interact with each other.

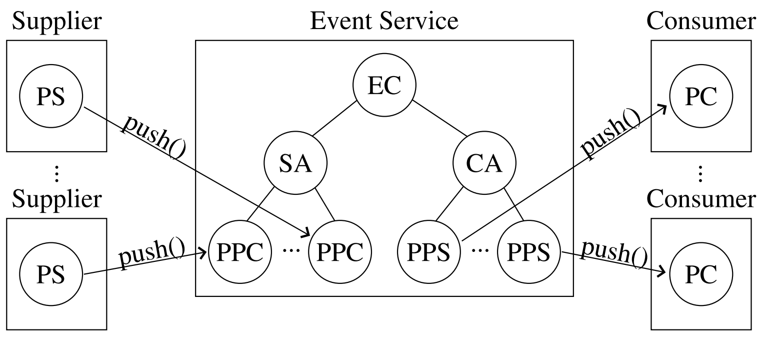
A consumer application must implement the `PushConsumer` interface. This has a `push()` operation that is invoked to pass it an `any` (Section 15.3 on page 140) containing arbitrary data related to an event. The `PushConsumer` interface also has an operation called `disconnect_push_consumer()`, which is invoked if the Event Service wants to disconnect itself from the consumer, for example, when an `EventChannel` is being `destroy()`ed.

A supplier application must implement the `PushSupplier` interface. This is, in effect, a *callback* interface (Section 1.4.2.2 on page 10). The Event Service invokes the `disconnect_push_supplier()` operation if it wants to disconnect itself from the supplier, for example, when an `EventChannel` is being `destroy()`ed.

An `EventChannel` is the initial point of contact for the Event Service. This interface just splits its functionality among the `SupplierAdmin` and `ConsumerAdmin` objects, and so provides operations that allow applications to access these objects.

```
module CosEventComm {
  interface PushConsumer {
    void push(in any data);
    void disconnect_push_consumer();
  };
  interface PushSupplier {
    void disconnect_push_supplier();
  };
};
module CosEventChannelAdmin {
  interface ProxyPushConsumer
    : CosEventComm::PushConsumer
  {
    void connect_push_supplier(
        in CosEventComm::PushSupplier push_supplier);
  };
  interface ProxyPushSupplier
    : CosEventComm::PushSupplier
  {
    void connect_push_consumer(
        in CosEventComm::PushConsumer push_consumer);
  };
  interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    ProxyPullSupplier obtain_pull_supplier();
  };
  interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ProxyPullConsumer obtain_pull_consumer();
  };
  interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
  };
};
```

Figure 22.1: IDL for the Event Service push model



EC = EventChannel	PC = PushConsumer
SA = Supplier Admin	PPC = ProxyPushConsumer
CA = ConsumerAdmin	PS = PushSupplier
	PPS = ProxyPushSupplier

Figure 22.2: The Push Model of the Event Service

`SupplierAdmin` is a *factory* interface (Section 1.4.2.1 on page 9). Its `obtain_push_supplier()` operation creates a `ProxyPushConsumer`. The use of “Proxy” in this name has nothing to do with code that is generated by an IDL compiler (Section 1.4.5 on page 13). Rather, a `ProxyPushConsumer` is a delegation object within the Event Service: a supplier invokes `push()` on a `ProxyPushConsumer` object and it, in turn, invokes (or arranges for something else within the Event Service to invoke) `push()` on each `PushConsumer` object in consumer applications.

The initialization of a supplier application involves the following steps. It first connects to the `EventChannel`. It invokes `for_suppliers()` to access the `SupplierAdmin` and then calls `obtain_push_consumer()` to create a `ProxyPushConsumer` object. Finally, the supplier calls `connect_push_supplier()` to register its own `PushSupplier` object. At this point, the supplier is fully connected to the Event Service and can call `push()` on the `ProxyPushConsumer`.

The initialization of a consumer application mirrors that of a supplier. It first connects to the `EventChannel`. Then it invokes `for_consumers()` to access the `ConsumerAdmin` and then calls `obtain_push_supplier()` to create a `ProxyPushSupplier` object. Finally, the consumer application calls `connect_push_consumer()` to register its own `PushConsumer` object. At this point, the consumer is fully connected to the Event Service and so its `push()` operation will be invoked whenever an event occurs.

22.2.2 The Pull Model

The push model is so called because it *pushes* data towards (proxy) consumers. Conversely, the pull model is so called because it *pulls* data from (proxy) suppliers. The additional IDL interfaces required for this are shown in Figure 22.3.

```
module CosEventComm {
  ...
  interface PullSupplier {
    any pull();
    any try_pull(out boolean has_event);
    void disconnect_pull_supplier();
  };
  interface PullConsumer {
    void disconnect_pull_consumer();
  };
};
module CosEventChannelAdmin {
  ...
  interface ProxyPullConsumer
    : CosEventComm::PullConsumer
  {
    void connect_pull_supplier(
        in CosEventComm::PullSupplier pull_supplier);
  };
  interface ProxyPullSupplier
    : CosEventComm::PullSupplier
  {
    void connect_pull_consumer(
        in CosEventComm::PullConsumer pull_consumer);
  };
};
```

Figure 22.3: IDL for the Event Service pull model

The `pull()` operation defined on `PullSupplier` is a blocking operation. The `try_pull()` operation is a non-blocking version. It returns immediately, and the `has_event out` parameter indicates whether the returned `any` has event data in it or is empty.

The Event Service specification defines additional interfaces that make it possible to transmit data using strongly-typed APIs rather than having to package up the data inside an `any`. However, the Typed Event Service was difficult for vendors to implement (due to some immaturity in CORBA at the time) and so there were very few implementations of it. For that reason, I do not discuss the Typed Event Service in this book.

The 80/20 principle [Koc00] applies to software products: “80% of people use just 20% of a product’s capabilities”. Of course, the percentages are not always 80 and 20, but the principle that most people use just a small subset of a product’s capabilities is true. This principle applies to the Event Service. Most people use just the push model of event communication; very few people use the pull model or the typed push/pull models.

22.2.3 Limitations of the Event Service

The Event Service suffers from several limitations, as I discuss in this section.

One limitation is that the Event Service does not define a *factory* interface (Section 1.4.2.1 on page 9) for creating event channels. This means that an Event Service implementation might have just one event channel. Conceptually, this is similar to a company deciding that it will have just one internal mailing list. The result is that subscribers receive *all* messages, even those in which they have no interest. Many implementations of the Event Service overcome this limitation by providing their own proprietary factory interface. However, use of such proprietary APIs obviously hinders source-code portability.

Another limitation is that the Event Service specification does not define what quality of service should be provided by an implementation. For example:

- Should an Event Service implementation keep track of connected suppliers and consumers only in memory, in which case details of the connections will be lost if the Event Service is killed and restarted, or should this information be maintained in a persistent store so that connections can be maintained even if the Event Service is killed and restarted?

- Should an Event Service implementation store yet-to-be-delivered messages only in memory, in which case the messages will be lost if the Event Service is killed and restarted, or should these messages be persisted in a file or a database so that they will *not* be lost if the Event Service is killed and restarted?

- If the Event Service has difficulty in delivering a message to a consumer then should the Event Service give up after the first delivery attempt? Or should the Event Service re-attempt the delivery a number of times? If the Event Service should re-attempt delivery a number of times then how long should it wait between retries? And should it finally give up after N attempts or after a particular amount of time?

The Event Service specification deliberately refrained from defining what quality of service should be offered. This was done with the hope of encouraging different vendors to compete by offering different qualities of service. However, this strategy backfired for two reasons.

First, most implementations of the Event service held all messages and information about connected suppliers/consumers in in-memory data-structures rather than in a file or a database. This meant that there was not much competition based on different qualities of service.

Second, an application might want more than one quality of service at the same time. For example, assuming that a vendor provided an `EventChannel` *factory* as a proprietary enhancement, a supplier application might want to send some messages on one `EventChannel` with a particular quality of service and some more messages on a different `EventChannel` that had another quality of service. However, most vendors offered only a *single* quality of service that applied to *all* the `EventChannel` objects.

The above limitations means that, unfortunately, the Event Service is unsuitable for the needs of most applications.

22.3 Notification Service

The Notification Service is sometimes referred to as "the Event Service on steroids", "the Event Service++" or simply "what the Event Service should have been in the first place". As I discuss in the following subsections, the Notification Service removes all the limitations of the Event Service that were discussed in Section 22.2.3, and adds additional functionality. The result is a publish-subscribe system that is very flexible, and scalable.

22.3.1 IDL Interfaces

The Notification Service is backwards compatible with the Event Service. This backwards compatibility is achieved by having the IDL interfaces of the Notification Service inherit from those of the Event Service. Furthermore, the naming conventions of the IDL interfaces for the Notification Service closely mirror those of the Event Service. This backwards compatibility and similar naming conventions provide two important benefits:

- The backwards compatibility makes it possible for developers who have already written Event Service-based applications to reuse those applications "as is" with the Notification Service and to then slowly migrate the applications so that they make use of the extra capabilities of the Notification Service.

- The backwards compatibility, combined with the similar naming conventions makes it possible for developers to use the Event Service as a stepping stone to learning the richer APIs provided by the Notification Service.

The Notification Service defines 38 IDL interfaces. This is in addition to the 11 IDL interfaces defined in the Event Service, from which interfaces in the Notification Service inherit. That is 49 interfaces in total! This is an amazingly high number of interfaces but, thankfully, most developers use just a small fraction of these interfaces. Many of the interfaces provide administration-type functionality and most vendors provide either command-line utilities or a graphical program that interacts with these administration-type interfaces, so that developers do not need to write any code to do so.

22.3.2 `StructuredEvent`

The Notification Service allows event data to be transmitted as an `any`. This is for backwards compatibility with the Event Service. However, the Notification Service allows event data to be transmitted in a different format, called a `StructuredEvent`, which is shown in Figure 22.4. Actually, this figure shows a slightly simplified IDL. In particular, some `typedef` declarations have been removed in order to make it more concise. For example, the type of the `variable_header` field in `EventHeader` is really a `typedef` of a `typedef` of the `sequence` shown.

The `EventType` embedded in the header of a `StructuredEvent` contains two `string` fields. The `domain_name` should be set to identify a particular vertical industry, for example, `"Telecomms"`, while the `type_name` should be set to uniquely identify a type of event within that domain, for example, `CommunicationsAlarm`. The rest of an `EventHeader` consists of a `sequence` of `Property`s, which are name-value pairs. Programmers can place whatever name-value pairs they want in this sequence, but it is intended to be used to express a desired *quality of service* (QoS) for controlling message delivery, for example, a message priority or a delivery timeout. The Notification Service *could have* used strongly typed fields in `EventHeader` to specify the QoS. However, there are two benefits to specifying them as weakly-typed name-value pairs. One benefit is that it reduces the size of the event header if, as is often the case, a supplier is happy with default QoS values. Another

```
module CosNotification {
  struct Property {
    string  name;
    any     value;
  };
  struct EventType {
    string  domain_name;
    string  type_name;
  };
  struct FixedEventHeader {
    EventType  event_type;
    string     event_name;
  };
  struct EventHeader {
    FixedEventHeader    fixed_header;
    sequence<Property>  variable_header;
  };
  struct StructuredEvent {
    EventHeader         header;
    sequence<Property>  filterable_data;
    any                 remainder_of_body;
  };
  typedef sequence<StructuredEvent> EventBatch;
  ...
};
```

Figure 22.4: Pseudo IDL for the Notification Service event data

benefit is that it allows future revisions of the Notification Service to define additional QoS name-value pairs in a backwards-compatible manner; likewise, it allows vendors to define additional, proprietary QoS() name-value pairs.

After the header, the `filterable_data` part of a `StructuredEvent` provides another `sequence` of name-value pairs. It is in this `sequence` that users are expected to place (most/all of) their event data. Representing the event data as name-value pairs is a trade-off between the awkwardness of a single unnamed `any` (as was provided by the Event Service) and a non-extensible but compile-time-safe `struct` with fixed fields. The intention of the OMG is that different vertical domains will define which name-value pairs should be specified for particular kinds of events. For example, representatives from different telecommunications companies might get together to define the name-value pairs for different kinds of events, such as `CommunicationsAlarm`, relevant to that industry.[1] These name-value pairs are called `filterable_data` because they can be accessed by *filters* (Section 22.3.4).

The final field in a `StructuredEvent` is an `any`. This field allows you to store any data that is unlikely to be of use to filters, possibly because the data is a large "blob" of data, such as the contents of a file.

22.3.3 `EventBatch`

The Notification Service can send and receive an `EventBatch` (Figure 22.4), which is a sequence of `StructurdEvent`. This enables applications with high throughput requirements to send or receive events in bulk. As an example of this, let us suppose a producer application produces 1000 events per second. Instead of the producer pushing 1000 events individually every second, this application could populate a sequence of 1000 structured events and push that sequence once per second.

A consumer can receive events in an `EventBatch` instead of receiving individual events. As I will discuss in Section 22.3.7, applications can specify various quality-of-services for their interactions with the Notification Service. A consumer application can use this to specify its desired batch size and *pacing interval*. The pacing interval is the maximum delay between delivery of events. For example, let us suppose a consumer sets the batch size to 1000 events and the pacing interval to 10 seconds. After 10 seconds, even if the channel has received only 300 events, it will send that batch to the consumer, rather than waiting until it has received all 1000 events before sending them to the consumer.

A discussion of how an application specifies if it wants to supply/consume events in the form of individual `any`s, individual `StructuredEvent`s or as an `EventBatch` is deferred until Section 22.3.5.

22.3.4 Filters

A *filter* is an object wrapper around a collection of *constraints* (conditions). Filters can be applied to messages as they pass through the Notification Service.

[1] Broadly speaking, this passing of data through weakly-typed, name-value pairs is similar in concept to the exchange of documents in XML format. Theoretically, an XML document *could* contain arbitrary *elements* (which are conceptually similar to name-value pairs). However, XML schemas can be defined that specify *which* elements should be present in an XML document. In both XML document exchange and transmission of messages through the Notification Service, it is people adhering to documented conventions that increases the likelihood of a particular document/message containing all the elements/name-value pairs that it is supposed to contain.

Syntactically, there are two different kinds of filter: `Filter` and `MappingFilter`. A `FilterFactory` is used to create both kinds of filter.

```
module CosNotifyFilter {
  interface FilterFactory {
    Filter        create_filter(...) raises(...);
    MappingFilter create_mapping_filter(...)
                                  raises(...);
  };
};
```

22.3.4.1 Filters to Remove Messages

The purpose of the `Filter` type is to delete messages before they are transmitted to consumers. Doing this saves consumer applications from having to examine messages to see if they are relevant. It also saves on network traffic because deleted messages are not transmitted.

The constraints within a `Filter` are expressed in *Extended Trader Constraint Language* (ETCL), which, as its name suggests, is an enhancement of the *Trader Constraint Language* (TCL) that is used with the Trading Service (Chapter 20). A constraint is a boolean expression that is written in terms of the name-value pairs within a `StructuredEvent` (Section 22.3.2). A constraint can also refer to the event's `domain_name` and `type_name`.

As previously mentioned, a `Filter` is a wrapper around a *collection* (represented as a `sequence`) of constraints. If *any* constraint within a `Filter` evaluates to *true* then the message is allowed to pass through. In other words, a message is discarded only if if *all* the constraints evaluate to *false*.

Filters can be attached to all combinations of proxy push/pull/consumer/supplier objects. However, it is more common to attach filters to the *consumer* proxies rather than the *supplier* proxies because it it more natural to want to filter what consumers receive rather than to filter what suppliers send.

Filters can also be attached to `SupplierAdmin` and `ConsumerAdmin` objects. The benefits of doing this are discussed in Section 22.3.5.

You can attach *multiple* filters to a proxy or admin object. If this you do this then a message is discarded only if *all* the constraints in *all* the filters evaluate to *false*.

22.3.4.2 Filters for Message Timeouts and Priorities

The header of a `StructuredEvent` can contain name-value pairs that specify a priority and/or deadline for message delivery. These entries, if present, are specified by the supplier of the event. The `MappingFilter` type allows consumers—actually the `ConsumerAdmin` and supplier proxies that act on behalf of consumers—to override the priority and/or deadline associated with a message. The name `MappingFilter` is not very intuitive; `OverrideFilter` might have been more intuitive.

A consumer admin or supplier proxy can have two `MappingFilter` objects associated with it. One of these is used to override a message's priority and the other is used to override a message's delivery deadline. Section 22.3.4.1 mentioned that a `Filter` is an object wrapper around a collection of constraints. In contrast, a `MappingFilter` is an object wrapper around a collection of *constraint-value* pairs. If a *constraint* evaluates to *true* then its associated *value* is used to override the message's priority or delivery deadline. Because different IDL types are used to express priorities and delivery deadlines, the *value* is wrapped inside an `any`.

22.3.5 `ConsumerAdmin` and `SupplierAdmin`

Figure 22.5 shows the create-style operations on the `SupplierAdmin` and `ConsumerAdmin` interfaces. Each of these operations takes a `ClientType` parameter that indicates if the supplier/consumer will handle event data as an `any`, `StructuredEvent` or a `EventBatch`. The create-style operation then creates a type-specific proxy object for the desired type. Each of these proxy types is a sub-type of `ProxySupplier` or `ProxyConsumer`.

The `SupplierAdmin` and `ConsumerAdmin` interfaces both inherit from `FilterAdmin`, which provides operations for associating `Filter` objects with the admin object. In addition to this, the `ConsumerAdmin` has two `MappingFilter` attributes, for overriding the priority and delivery deadline of messages. The ability to associate filters with a `ConsumerAdmin` makes it possible to do filtering for a *group* of consumers rather than individually for each consumer. Not only is this more convenient from a maintenance point of view, it is also an important optimization because a filter is evaluated *once* for the entire group of consumers rather than being evaluated *repeatedly*, once per consumer. Providing filters in a `SupplierAdmin` is of less utility (simply because filtering is typically a consumer-side issue) but the capability is provided for the sake of symmetry.

22.3.6 `EventChannel`

One of the limitations of the Event Service was that it defined just a single event channel; it did *not* define a *factory* (Section 1.4.2.1 on page 9) that could

```
module CosNotifyChannelAdmin {
  ...
  enum ClientType {ANY_EVENT, STRUCTURED_EVENT,
                              SEQUENCE_EVENT};
  interface ConsumerAdmin
    : CosNotifyFilter::FilterAdmin,
      // rest of inheritance clause omitted
  {
    attribute CosNotifyFilter::MappingFilter
                                           priority_filter;
    attribute CosNotifyFilter::MappingFilter
                                           lifetime_filter;
    ProxySupplier obtain_notification_pull_supplier(
                  in  ClientType ctype, ...) raises (...);
    ProxySupplier obtain_notification_push_supplier(
                  in  ClientType ctype, ...) raises (...);
    ...
  };
  interface SupplierAdmin
    : CosNotifyFilter::FilterAdmin,
      // rest of inheritance clause omitted
  {
    ProxyConsumer obtain_notification_pull_consumer(
                  in  ClientType ctype, ...) raises (...);
    ProxyConsumer obtain_notification_push_consumer(
                  in  ClientType ctype, ...) raises (...);
    ...
  };
};
```

Figure 22.5: IDL for Notification Service Admin objects

be used to create additional event channels. The Notification Service removes this limitation. Figure 22.6 shows the IDL definitions for `EventChannel` and `EventChannelFactory`.

The `create_channel()` operation defined on the factory interface is used to create a new `EventChannel`. The parameters passed to this operation (which have been omitted from the IDL shown in Figure 22.6) are used to specify the QoS of the newly created channel. An overview of the available QoS is provided in Section 22.3.7.

Each `EventChannel` provides a set of pre-created `ConsumerAdmin`,

```
module CosNotifyChannelAdmin {
  ...
  interface EventChannel
    : // inheritance clause omitted
  {
    readonly attribute ConsumerAdmin
                                default_consumer_admin;
    readonly attribute SupplierAdmin
                                default_supplier_admin;
    readonly attribute CosNotifyFilter::FilterFactory
                                default_filter_factory;
    ConsumerAdmin new_for_consumers(...);
    SupplierAdmin new_for_suppliers(...);
    ...
  };
  interface EventChannelFactory
  {
    EventChannel create_channel(...) raises(...);
    ...
  };
};
```

Figure 22.6: IDL for Notification Service event channel and factory

`SupplierAdmin` and `FilterFactory` objects. These objects are accessible through the `default_<...>` attributes. However, the `new_for_consumers()` and `new_for_suppliers()` operations allow additional admin objects to be created. By creating different admin objects for different groups of suppliers/consumers, it is possible for the Notification Service to offer a different QoS (Section 22.3.7) for different groups of suppliers/consumers. Also, this arrangement makes it possible for, say, a `ConsumerAdmin` object to filter messages on behalf of a group of consumers. Such group-level filtering is much more efficient (and hence more scalable) than repeated filtering at the granularity of individual proxies.

22.3.7 Quality of Service (QoS)

The Notification Service allows users to choose a quality of service for the transmission of events. The following list briefly explains some of the more important quality of services available with the Notification Service:

- Within the Notification Service, you can create many channels, where each channel can be used to transmit different types of event. This is conceptually similar to setting up multiple mailing lists within an organization, to facilitate discussion of different topics. A channel can be either *persistent* or *best effort* (which is the Notification Service terminology for non-persistent). If a channel is persistent then this means that the channel infrastructure and all its yet-to-be-delivered events are recorded in a file or database so they survive even if the Notification Service process dies and is restarted. In contrast, the infrastructure of a best-effort channel and its yet-to-be-delivered events are held only in the memory of a Notification Service, and so a best-effort channel and its events are implicitly destroyed when the Notification Service process is killed and restarted.

- A best-effort channel can be used to transmit only best-effort events. In contrast, a persistent channel can be configured to transmit either persistent or best-effort events. If best-effort events are transmitted on a persistent channel and the Notification Service dies then all the yet-to-be-delivered events are discarded.

- When the Notification Service receives an event from a supplier application, the Notification Service may not be able to deliver the event to all the consumers immediately. For example, if a consumer application is not currently running then the Notification Service will periodically try to re-send the event to the consumer. There are several quality-of-service values that can be used to control how hard the Notification Service tries to deliver an event to a consumer before it gives up and discards the event:

  – *Yet-to-be-delivered* events for a consumer can be delivered in: FIFO (first-in, first-out) order, priority order (based on an optional priority value in the header of events), deadline order (based on an optional delivery-deadline value in the header of events) or any order (which allows the Notification Service to deliver events in whatever order it wants).

  – A limit can be placed on the number of *yet-to-be-delivered* events that can be queued up for a consumer.

  – If the Notification Service is about to exceed the maximum limit of yet-to-be-delivered events for a consumer then it discards an event. The discard policy can have any of the values used for the delivery

policy, that is, FIFO order, priority order, deadline order or any order.
  - If an attempt to deliver an event fails then how long the Notification Service waits before trying to resend an event.
  - The maximum number of retries before the Notification Service gives up and discards the event.
  - A timeout value for invoking an operation on the consumer to deliver the event.

- It is possible to set maximum limits on the number of suppliers and consumers that can be connected to the Notification Service.

Quality of service for a channel is indicated through name-value pairs. QoS can be specified when creating an `EventChannel` (Figure 22.6). This default QoS can then be overridden at the level of a `ConsumerAdmin` or `SupplierAdmin` object and/or at the level of individual proxy objects.

22.4 Telecom Log Service

In the Notification Service (Section 22.3), an `EventChannel` created with the *persistent* QoS stores each event in a persistent store (for example, a file or database) until it has delivered the event to all consumers. At that point, the `EventChannel` deletes the event from the persistent store. Some organizations prefer to keep a permanent record of events. They can do this with the Telecom Log Service, so called because it was defined by companies in the telecommunications industry, but its functionality is useful to organizations in other industries too.

The most important interfaces in the Telecom Log Service are `NotifyLogFactory`, which is a factory (Section 1.4.2.1 on page 9) that creates `NotifyLog` objects, which inherit from the `EventChannel` interface of the Notification Service.

When a `NotifyLog` object (actually, one of its consumer proxy objects) receives an event from a supplier, it passes the event on to consumers, if any, and it also stores the event in a persistent store. The `NotifyLog` keeps events in its persistent store, even after it has delivered the events to all subscribed consumers.

Because the Telecom Log Service leverages the infrastructure of the Notification Service, a `NotifyLog` can have `Filter` objects (Section 22.3.4.1) associated with it. This means that a log object can be selective about *which* events it records.

The `NotifyLog` interface has operations that allow you to iterate over the events in its persistent store. You can `query()` events that match a particular constraint. You can also `retrieve()` the events that occurred since a specified time.

The Telecom Log Service gives you a lot of control over the persistent store of a `NotifyLog` object. For example:

- You can specify how much space a `NotifyLog` object should allocate in its persistent store for events.
- You can specify how a `NotifyLog` should behave when its persistent store fills up. The two options available are to use a *wrap* policy so that the `NotifyLog` overwrites the oldest events, or use a *halt* policy so that the `NotifyLog` stops recording new events.
- You can instruct the Telecom Log Service to generate an event when a `NotifyLog` reaches a specified percentage of capacity. This allows users to take appropriate action when a log is about to fill up, for example, create a new log that will be used in place of the existing log.
- You can delete individual events or events that match a specified constraint from a log.
- You can specify that events in the log should be deleted automatically after a specified number of seconds.
- You can instruct a `NotifyLog` to save only those events that are received during specified times/days.

Chapter 23

Security

The CORBA Security Service (CORBASEC) is defined in a way that is independent of any particular security technology. Instead, the specification can be applied to numerous security protocols, and the CORBASEC APIs shield application developers from differences across security technologies as much as possible. This abstraction from specific security technologies gives CORBA developers the freedom to change the underlying security technologies used in a given system without needing to redesign the applications that use CORBASEC.

23.1 Features of CORBASEC

This section discusses the most important security features that CORBASEC addresses. Constructing secure CORBA systems involves using code or configuration to specify the desired combinations of these features at an appropriate level of detail for your system. These security features are common to many different security protocols or mechanisms.

Authentication. This is the requirement that *entities* (for example, people or programs) prove their identity. In CORBASEC terminology, an entity with the ability to use the resources of a system is called a *principal*. *Authentication* is the process of verifying an entity's claimed identity. Note that clients can authenticate servers, and vice versa. Authentication can be either mandatory or optional depending on the security requirements of a given system. Successful authentication results in the principal being granted a set of *privilege attributes* (such as roles, groups, security clearance levels and so on); these are stored in a *credentials* object and are later considered during *authorization*. Some examples of different authentication mechanisms that are commonly used are X.509 certificates, usernames and paswords, and smart cards or hardware tokens. An application has separate credentials for each authentication mechanism with which it wishes to authenticate itself.

Authorization. This is the process of verifying whether or not a principal is allowed to perform a requested action in a system. An example of a commonly used authorization paradigm is Access Control Lists (ACLs). Although the flexibility of ACLs differ among CORBA Security products, ACLs typically allow access to be constrained at varying levels of granularity, such as per-process, per-object, per-interface or per-operation. Some products may also provide ACL functionality that allows access decisions to be made based on the values of parameters passed to IDL operations. Note that during authorization the set of *privilege attributes* that was determined for the principal during the authentication process is used to control access to system resources.

Data integrity. This uses techniques such as message digests (a form of cryptographic checksum) to provide protection against malicious modification of messages.

Confidentiality. This ensures the privacy of message exchanges so that only the intended recipients can read them.

Detection of replayed messages. This prevents active attackers from replaying previously stored communications.

Detection of misordering. This prevents an attacker from rearranging messages in a different order to that in which they were sent.

Auditing/logging. This involves keeping secure records of “who did what” so that access to system resources can be examined at a later time. Some security systems allow registration with a real-time management service that can perform appropriate system-defined alerts.

Delegation. This is when one user or principal authorizes another to use their identity or privileges, potentially with usage restrictions. Delegation involves controlling and recording the principal identities that are involved in executing a CORBA request across a chain of participating servers. Delegation can be transparent to the authorization process and be based on the current effective principal. Alternatively, some security products additionally allow authorization decisions to be based on delegation constraints associated with a request.

CORBASEC details a very rich model for delegation that includes support for a number of delegation modes of increasing complexity. This is discussed further in Section 23.2.2.

Non-repudiation. The verb *repudiate* means to deny, disown or reject as untrue. *Non-repudiation* means the ability to prove whether or not a principal invoked a particular operation, so that the principal cannot later deny invoking an operation that he or she did, in fact, invoke. This is an advanced feature and, in practice, depends on the existence of an appropriate supporting infrastructure to be able to persistently store and subsequently recover the relevant evidence for disputed actions. Non-repudiation support is an *optional* conformance point of the CORBASEC specification and is not supported by most current CORBASEC implementations.

23.2 CORBASEC Conformance Levels

The capabilities of CORBASEC are divided into a number of distinct conformance packages that are described in the following subsections.

CORBASEC Level 1 and Level 2 are two packages that define the most important core functionality, over which the other functional packages listed are layered to provide additional security features appropriate to specific deployment environments.

23.2.1 CORBASEC Level 1

CORBASEC Level 1 specifies how secure associations are established between client and server applications to provide authentication, confidentiality, replay and misordering detection, ORB-mediated authorization decision support and auditing. Importantly, CORBASEC Level 1 defines support for *security-unaware applications,* which involves being able to *configure* applications to communicate securely, *without* the need to write security-specific code. This is a very useful feature that removes a whole category of potential security coding errors for simple applications. CORBASEC Level 1 security also supports simple delegation. This allows the identification and subsequent authorization of an originating client when a request is sent from that client to a server and this server then acts on behalf of the originating client when making related requests to other servers. ORB-enforced access control checks must also be supported at this level, the specification gives implementors freedom as to how access decisions are performed.

While a large number of applications can be usefully configured to be secure using only CORBASEC Level 1 functionality, more complex applications may require the use of the CORBASEC Level 2 features outlined below.

23.2.2 CORBASEC Level 2

CORBASEC Level 2 is a superset of CORBASEC Level 1 functionality, all functionality specified for CORBASEC Level 1 applications is also available to CORBASEC Level 2 applications.

23.2.2.1 Security Aware APIs

CORBASEC Level 2 specifies comprehensive API support for *security-aware applications*. These APIs make it possible for both server and client applications to have finer control over security policies than can be obtained by configuration alone. These APIs allow control over most aspects of CORBASEC security such as how to combine secure and insecure communications, create and use multiple security credentials, and query the details of peer credentials. Normally CORBASEC APIs are used only when the desired functionality is not possible through the use of *security-unaware* applications. An example of where the use of CORBASEC APIs might be necessary for an application would be when some dynamic computation needs to be executed to determine whether a request should be allowed or denied.

23.2.2.2 Delegation

CORBASEC Level 2 also supports a variety of delegation modes that are important if you need more control over the chain of principals involved in a CORBA request that involves one or more intermediate CORBA servers:

No delegation. The client permits the intermediate server object to use its privileges for access control decisions, but does *not* permit them to be delegated. Because of this, the intermediate server object cannot use the client's privileges when invoking upon other objects.

Simple delegation. The client permits the intermediate object to assume its privileges, both using them for access control decisions and delegating them to others. The target object receives only the client's privileges, and does not know who the intermediate is.

Composite delegation. The client permits the intermediate object to use its credentials and delegate them. The client's privileges *and* the intermediate object's privileges are passed to the target, so that both sets of privileges can be individually checked.

Combined privileges delegation. The client permits the intermediate object to use its privileges. The intermediate object converts these privileges into credentials and combines them with its own credentials. In this case, the target cannot distinguish which privileges come from which principal.

Traced delegation. The client permits the intermediate object to use its privileges and delegate them. However, at each intermediate object in the chain, the intermediate's privileges are added, and all the privileges are propagated to provide a trace of the delegates in the chain.

A client application may not see the difference between *composite delegation*, *combined privileges delegation* and *traced delegation*; the client may just see them all as some form of composite delegation. However, the target object can obtain the credentials of intermediates and the original client separately if they have been transmitted separately.

Time periods can be applied to restrict the duration of the delegation. In some implementations, the number of invocations may also be controllable.

23.2.2.3 Access Control

In general CORBASEC implementations may use any one of a large number of authorization models. CORBASEC Level 1 does not prescribe a specific authorization model. In contrast, CORBASEC Level 2 requires support for a specific authorization model that is called *DomainAccessPolicy*. The actual *use* of this model (as opposed to its mere availability) is not mandatory; customers can (and commonly do) use alternative authorization models.

It can be argued that a specific authorization model should have been an optional part of the specification. This is because CORBA is an integration technology and, as such, it most usefully needs to integrate with the *existing* security systems that customers have already deployed. Nonetheless the DomainAccessPolicy authorization model is a powerful one; it enables the specification of the *required rights* for operations that are validated at runtime against the *effective rights* associated with the current client's *privilege attributes*.

The Administration of the mapping of rights to operations is also specified by CORBASEC Level 2. Although these administration interfaces are well designed, many customers need their security solutions to integrate with already-deployed, non-CORBA-specific enterprise security infrastructure. An already-deployed infrastructure will probably have its own administration command-line or GUI tools, and the mandated CORBASEC DomainAccessPolicy administration policy may not be of any relevance for such deployments.

23.2.3 Non-repudiation Package

This package allows for the generation and checking of evidence so that actions cannot be repudiated after the event.

23.2.4 Security Replaceability Packages

These packages standardize a way to integrate third-party Security Service implementations with a CORBA product. There are two relevant packages:

ORB Services Replaceability Package. The ORB uses *portable interceptors* (Chapter 14) to call on object services, such as transactions or security. An ORB conforming to this specification does not contain any significant transactions- or security-specific code; instead such code is contained in portable interceptors, to which the ORB delegates.

Security Service Replaceability Package. Even if the ORB does not implement the ORB Services Replaceability Package, it still makes all calls on security services via specified replaceability interfaces.

A CORBA product that supports one or both of these replaceability packages is said to be *security ready*. A security-ready CORBA product may or may not be bundled with security functionality. However, it is ready to host CORBASEC functionality, which may be implemented by a third-party vendor.

23.2.5 Secure Interoperability

23.2.5.1 Common Secure Interoperability (CSI) Feature Packages

The Common Secure Interoperability (CSI) specification ensures secure interoperability between different vendors' security products. There are three CSI feature packages, each of which provides a different level of secure interoperability:

Level 0. Identity-based policies without delegation.

Level 1. Identity-based policies with unrestricted delegation.

Level 2. Identity-based policies with controlled delegation.

Different vendors' products are interoperable if they support the same level of common secure interoperability and also share support for the same *Common Security Protocol* (discussed in Section 23.2.5.2). In general the CSI specification describes how interoperability is achieved by embedding `TaggedComponent` entries in IORs (Section 10.2.3 on page 108). The actual information embedded in the IOR is specific to the *Common Security Protocol Packages* that are being used. Clients that support CSI validate the CSI information contained in an IOR against their specified client side security policies to check:

- that their local security policies allow them to establish a secure association to the specified target for the current operation (*should* I do this?), and also
- that they have the required mechanism-specific capabilities such as the specific cryptographic profiles that may be indicated in the IOR (*can* I do this?).

CORBA implementations may simultaneously support more than one specific *Common Security Protocol Package*—this is useful since it facilitates bridging between different security technology domains.

23.2.5.2 Common Security Protocol Packages

A specific common security protocol package contains all functionality that is required for secure vendor-independent interoperability between different orbs over the specific security mechanism to which the package relates. The list of currently available *Common Security Protocol Packages* is described below. All of the items listed below except for the respective SSL and DCE-CIOP interoperability definitions depend on the implementation of the *Secure Inter-ORB Protocol* (SECIOP).

SSL protocol The popular SSL or TLS family of protocols provide Public Key based mutual authentication capabilities (using X.509v3 certificates) as well as confidentiality, integrity, replay and misordering detection capabilities.[1] These protocols inherently support identity-based policies without delegation over a secure channel to provide CSI Level 0 capabilities. Note that CSIv2 level 1 or 2 capabilities can also be layered

[1] The Transport Layer Security (TLS) protocol is the successor to Netscape's Secure Socket Layer (SSL) protocol. TLS 1.0 is based on SSL 3.0.

over the basic SSL/TLS protocols to provide a richer security solution and this is covered in Section 23.2.5.3. The TLS protocol is defined in IETF RFC 2246.

GSS Kerberos Protocol This protocol supports identity based policies with unrestricted delegation (CSI Level 1) using secret key technology for keys assigned to both principals and trusted authorities. It is also possible to use it without delegation (providing CSI Level 0). The GSS Kerberos protocol is based on the IETF GSS Kerberos V5 definition.

SPKM Protocol This protocol supports identity-based policies without delegation (CSI Level 0) using public key technology for keys assigned to both principals and trusted authorities. The SPKM protocol is based on IETF RFC 2025, “The Simple Public-Key GSS-API Mechanism”.

CSI-ECMA protocol This protocol supports identity- & privilege-based policies with controlled delegation (CSI Level 2). It can be used with identity, but no other privileges and without delegation restrictions if the administrator permits this (CSI Level 1) and can be used without delegation (CSI Level 0). The CSI-ECMA protocol is based on a SESAME profile of the ECMA GSS-API Mechanism as defined in ECMA 235. There are three CSI-ECMA variants: CSI-ECMA Public Key, CSI-ECMA Secret Key, and CSI-ECMA Hybrid.

DCE-CIOP This DCE environment specific protocol achieves secure interoperability between ORBs using the DCE-CIOP transport. Is is dependent on the security services provided by DCE and the DCE Authenticated RPC runtime that utilizes those services. DCE-CIOP is not based on the IIOP protocol and is an example of an *Environment Specific Interoperability Protocol* (Section 11.1 on page 111).

23.2.5.3 CSI Version 2 Security Attribute Service (CSIv2 SAS) Protocol

This important protocol is layered over the ORB transport functionality and provides additional client authentication, delegation, and privilege functionality that may be applied on top of any security mechanism used at the transport layer. The SAS protocol is usually layered over secure transports that are interoperable as defined by the CSI specification, but it can also be used in conjunction with insecure transports if required.

The SAS protocol is divided into two layers:

1. The *Authentication Layer* is used to perform client authentication where sufficient authentication could not be accomplished in the transport. For

example the SSL protocol by itself supports only authentication through the use of X.509 certificates. However in conjunction with CSIv2, fully interoperable username-password client authentication and delegation over SSL is possible between different vendors' products.

2. The *Security Attribute Layer* (also referred to as the Common Authorization Layer) may be used by a client to deliver privilege and identity attributes to a server where they may be used for authorization and delegation related purposes.

If you plan on integrating CORBA and J2EE applications then you should note that, since version 1.3, the J2EE Application Server specification mandates the use of the CSIv2 SAS protocol over SSL for secure interoperability between CORBA and J2EE applications.

23.3 Issues Not Covered by CORBASEC

The following subsections briefly discuss some issues that are outside the scope of CORBASEC.

23.3.1 Configuration

CORBASEC defines portable APIs for security, but it does *not* define any details related to the configuration of a security technology. There are two reasons for this.

1. All security technologies need configuration of some kind and since CORBASEC is a technology-*neutral* specification, it is impractical for CORBASEC to define any details related to configuration of security technologies.

2. CORBASEC implementations are available for a wide range of computers, from embedded devices and PDAs, through PCs and UNIX machines, all the way up to mainframes. However, different computer systems often have entirely different configuration mechanisms, even for the same security protocols. There is too much variety in computer systems for a standardized configuration mechanism to be accepted and used by developers and administrators.

A practical ramification is that the *source code* of a CORBA application that uses security can be written in a portable way, but the *configuration and administration* of the application will vary from one CORBASEC product to another.

CORBASEC *does* define administration APIs that can be used to administer some standard CORBASEC security policies. However, for the reasons discussed above, these APIs are not sufficient, by themselves, to provide complete portability of security configuration.

23.3.2 Proprietary Enhancements

Some CORBASEC vendors provide proprietary enhancements in the form of APIs that allow access to security functionality not covered by CORBASEC. Such proprietary APIs are sometimes provided to give programmers access to security technology-specific information. For example, a CORBA product that supports IIOP/TLS might provide proprietary APIs that allow access to the X.509v3 extensions associated with a peer's X.509 certificate chain. CORBASEC does not define how this is achieved; it just defines how an abstract `AccessID` value should be returned to the application, and this is sufficient for many application types.

23.4 Evaluating CORBASEC Implementations

The CORBA Security Service specification [OMGb, OMGa] covers the security requirements of an enormous amount of different types of applications. Do not jump to the conclusion that only a "fully CORBASEC Level 2 compliant" product will suffice for your requirements—you might be ruling out many products that may be more suitable for your specific system. In many cases, an easily-determined subset of CORBASEC functionality is all that is required. When selecting a CORBASEC product, a good approach is to identify what security functionality you need in your system now, and what you are likely to need in the future. Then check this required functionality against that provided by several vendors and built a small proof-of-concept application to validate your understanding of what is possible.

During your evaluation of CORBASEC implementations, you may want to consider the issues discussed in the following subsections.

23.4.1 Adherence to relevant standards

While not mandated by CORBASEC, it is anticipated that CORBASEC implementations should integrate with existing security technologies that are commonly used in industry. It is desirable for a vendor's security implementation to adhere to relevant commonly deployed industry standards wherever possible. Some reasons for this are as follows:

- You want to be sure that you are using well-understood security protocols that have undergone appropriate industry analysis and acceptance.
- You want to be sure that a server implemented with one CORBASEC product can interoperate with a client implemented with a different vendor's product.
- Sticking to standard security solutions gives you more options if/when you decide to replace part of your security solution.

23.4.2 Support for Security-unaware Applications

Does a vendor provide sufficiently powerful support for *security-unaware* applications? Such support is very important because it involves no programming. However, different vendors' implementations may vary greatly in the scope of their support for security-unaware applications. When evaluating the security-unaware support for a product, pay particular attention to the type of authorization decisions that your applications need to make, and how security credential initialization details are handled.

23.4.3 Pluggable Security Code

This issue is related to support for security-unaware applications (discussed in the previous subsection), but is sufficiently important to warrant a separate discussion. The security requirements of your application may change over time. It is time-consuming (and therefore expensive) to have to modify an application's source code when its security requirements change. Some CORBA vendors provide a proprietary mechanism by which you can write a *plug-in*, which is code that can be dynamically loaded into an application. A plug-in is typically packaged as a UNIX shared library (a DLL on Windows) for C++ applications, or as a Java class for Java application. A configuration file is used to specify which plug-ins should be loaded by an application; once loaded, a plug-in acts as an *interceptor*, though its API might not be that of a portable interceptor (Chapter 14). If a vendor supports plug-ins then you should consider using them, rather than calling CORBASEC Level 2 APIs directly from application code. Use of plug-ins increases the modularity (and hence maintainability) of your applications, simply because it keeps "business logic" code separate from security code. This modularity also increases the portability of your application because if you need to use any proprietary security APIs then their usage can be confined to the plug-in.

23.4.4 Portability

Consider the portability of your applications across different vendor products. Security-unaware applications are the easiest to port since it is just configuration information associated with security policies that needs to change. If security-related code is required by your application, does the vendor's product support standard CORBASEC API interfaces for the functionality that you require? Note that if you require access to security mechanism-specific data then it is likely that you will have to use some vendor-proprietary APIs because CORBASEC does not define this type of functionality.

23.4.5 Interoperability

Consider your current and potential interoperability requirements carefully. While every organization's interoperability requirements may be different, currently the most commonly available secure interoperability implementations are those based on CSI interoperability over the SSL/TLS protocols.

I discussed CSIv2 in Section 23.2.5.3 on page 232 but there is also a simpler Common Secure Interoperability Version 1 (CSIv1) specification for SSL. CSIv1 defines how extremely basic IIOP/SSL connectivity for SSL-enabled ORBs should be achieved. This specification enables IIOP/SSL implementations from different vendors to interoperate securely at a peer-to-peer level over the secure transport. No "higher level" semantics such as delegation or token-based client authentication are defined.

CSIv2 provides a superset of CSIv1 functionality, but note that an ORB implementation *may* support both versions simultaneously, which would allow backward compatibility with older versions.

23.4.6 Administration of Authentication & Authorization Information

How easy and scalable is the administration of the security rights for principles and separately of the required rights for specific actions? For example, if a company has hundreds of employees and thousands of customers then it would be impractical to manage the security rights for each person individually. Many different authentication and authorization models are possible but, as an example, an approach that some current security products offer involves defining several conceptual groupings of principals such as *employee*, *administrator*, *customer*, *premium customer* and so on. Administration is performed at the granularity of assigning group membership to various principals. At runtime, when access to a resource is requested the authorization component of the security system determines the group membership requirements for access to the resource and whether the principal satisfies those requirements.

23.4.7 Scalability and Fault Tolerance

Normally the security service is a critical part of the system and you need to understand if it will introduce a central bottleneck or single point of failure for your applications. For example, if the security service requires deployment of processes for authentication and authorization then you should check if these processes are replicated to provide fail-over or load balancing.

23.4.8 Integration with Enterprise Security Systems

Check if a vendor's security product provides out-of-the-box integration with whatever enterprise security system you want to use. If this functionality is not provided out-of-the-box then you may want to consider checking if the product provides a framework that allows you to implement such an integration yourself. It may also be relevant to check if the vendor's solution can help integrate multiple different enterprise security solutions at the same time (see also the related SSO subsection below).

23.4.9 Single-Sign-On Support (SSO)

Single-sign-on allows principals to use their credentials to obtain additional rights or credentials that can be used for the purposes of authentication and authorization across different security domains. Some of the main characteristics of SSO solutions are described below:

Unified logon. SSO-enabled clients do not have to logon to each distinct security domain. As well as being a convenience for clients, this also can greatly improve the security of sensitive client information. For example, a basic CSIv2 username/password deployment would result in the client providing its password to every server that it contacts. Of course, the client should be configured to use SSL to perform appropriate authentication of the servers before giving them its password, but an SSO solution will limit the visibility of the client's long-lived password to a single carefully authenticated login server and instead will use a short-lived SSO token to communicate with application servers. In general it is good security practice to always limit the visibility of sensitive information to the minimum number of system components that need it.

Bridging across Security Policy Domains. SSO capabilities can also be used to provide a bridge across different security policy domains that use the same underlying security mechanisms. For example, you could have a trusted federation of security services that serve different security policy domains; in such a deployment, authentication/issuing decisions would be referred to the security service that issued the SSO token.

Bridging across Security Technology Domains. SSO capabilities can also be used to bridge across different security technology domains. For example, a client can obtain an SSO token with one particular security mechanism and then use this SSO token in conjunction with a different security mechanism that is required by the server. The client may not be capable of performing direct authentication to the server over the second mechanism but the SSO token can provide the additional credential information required.

23.4.10 Key and Password Management

Consider what facilities the vendor's product provides for safely handling security key/password data for your chosen security mechanisms. The use of non-standard data formats or mechanisms for credential-related information would complicate any potential migration to another CORBASEC vendor's solution.

23.4.11 Client-side Security Policies

Some security solutions focus on server-side security to the exclusion of the client. Investigate what features are provided by vendor products to ensure that clients are communicating with the appropriate servers to serve their requests.

23.4.12 Secure `corbaloc`

As discussed in Chapter 12, `corbaloc` URLs provide a human-readable alternative to stringified IORs (Section 3.4.2 on page 34). Unfortunately, the `corbaloc` specification does not officially support SSL. A common misconception is that this lack of support is irrelevant since it is possible to use insecure communications in conjunction with `corbaloc` to obtain a secure object reference. However, this is not the same thing as using secure communications from the start to obtain the secure IOR. Without SSL you are obtaining an object reference from an *unauthenticated* corbaloc server; in theory what you obtain could be a different object reference than that which would have been returned by an *authenticated* corbaloc server. In general analyzing the security of your system is simpler if you use secure communications everywhere possible. Some vendors provide a proprietary SSL extension to `corbaloc` functionality to avoid this problem.

23.4.13 Secured CORBA Services

Check whether CORBA services (for example, Naming, Transactions, Notification, and various daemon services) that are supplied by the vendor are security enabled out-of-the-box, or whether they can be configured to be secure using standard security functionality provided by the product.

23.4.14 Firewall traversal

If using IIOP over the Internet is of relevance to you then you may wish to ask if your vendor provides any support for secure traversal of firewalls. Unfortunately agreement upon an OMG firewall traversal specification for IIOP has proven difficult to obtain. This is a complex area but two potential approaches are briefly mentioned in the following subsections.

23.4.15 Firewall Proxy Servers

Some vendors provide IIOP firewall proxy servers that mediate inbound and outbound IIOP requests before forwarding them on to their intended recipient. If a secure firewall traversal solution is important to you then you need to find out if the IIOP firewall supports a consistent integration with the desired enterprise security policies. For example, does it guarantee that the secure association established between the client and the firewall is compatible with the security policies of the intended target?

23.4.16 Bi-directional IIOP

The availability of a bi-directional IIOP solution can help simplify secure navigation across firewalls. It is important to note that CORBA callback objects normally require a separate connection from the server back to the client. This can complicate the port management aspect for a standard firewall. Additionally it may be desired to only support IIOP connections in one direction and not allow outbound IIOP connections at all. Bi-directional IIOP facilitates this by reusing the connection that already exists from the client to the server. If your vendor supports bi-directional IIOP then you should ask for details of how they ensure compatibility of the server's "client side" security policies with the existing connection when it makes callback invocations to the client.

23.5 Final Comments

It is critical to understand that no security product provides total security by itself. Good security software is a "necessary but not sufficient" condition for a secure system in the real world. A naive deployment of perfectly good security products can result in perfectly insecure systems. It is essential that personnel with an appropriate background in security understand the core security mechanisms that are used for a given system and review the overall security of a planned deployment. Expecting developers that have no background or appropriate training in security to "add on" security near the end of a project's development schedule is a recipe for disaster. A related point to this is that developing a secure system involves a different mindset to developing an system without security. For insecure systems, typically everybody is mostly concerned with ensuring that everything that should work does indeed work. For secure systems you also need to ensure that everything that should *not* work, indeed does not work. Sometimes developers working on the core system see security as an "obstacle that has to be overcome" as opposed to critical functionality in its own right.

23.6 Further Reading

The CORBA Security specifications [OMGb, OMGa] are sources of definitive (but at times tedious) information. A gentler introduction to CORBA security can be found in *CORBA Security* [Bla99]. Some good books that explain security issues in a non-CORBA-centric way include *Cryptography Decrypted* [MBB00], *Network Security* [KPS02] and *Applied Cryptography* [Sch95].

Chapter 24

Services not Discussed in this Book

This chapter briefly summarizes some of the CORBA Services that are *not* discussed in this book. Interested readers can find a discussion of some of these topics in other books, articles and Internet resources, many of which are listed in Chapter 26.

24.1 Persistent State Service (PSS)

Many real-world applications require the ability to maintain data in a persistent store. Unfortunately, in such applications, the low-level APIs required to access a persistent storage device, such as a file or a database, are often intermixed with the "business logic" code in the application. Furthermore, the APIs for accessing different type of persistent store vary enormously. These factors make it very time-consuming to modify an application that uses one type of persistent storage to use a different type of persistent storage instead. As an example of this, let us assume that you are given the task of implementing a CORBA server that must maintain some data in a persistent store. The development and maintenance of this application might proceed as follows:

1. The data-storage requirements of the application are quite modest so you decide to persist the data in files. The mainline of your server and the bodies of IDL operations are likely to contain a mixture of "business logic" code inter-mixed with code for opening, closing, reading and writ-

ing files. When you finish implementing the application, you deploy it and it works well.

2. During the next year, the load gradually increases and eventually it becomes apparent that there are some scalability limitations in your file-based persistence mechanism. You decide to switch over to using an object database for persisting data. This change requires that you remove a lot of the file-access code in the application and replace it with APIs that are specific to the object database. These modifications take several weeks/months of hard work, but once that work is complete, you are happy that your application can easily scale up to deal with the increased workload.

3. The next year, your company merges with another company. In the newly-merged company, a decision is made to streamline the extensive range of third-party software products being used within the company. As a result of this, you are told to remove use of the object database from your CORBA server and to replace it with, say, Oracle. You then spend several weeks/months to make the necessary modifications.

The intention of the Persistent State Service (PSS) is to eliminate the need for extensive source-code changes when switching from one persistence mechanism to another. PSS is an abstraction layer that insulates developers from the technology and APIs of the underlying persistent storage device. PSS deals *only* with persistence of data: it does *not* provide transactional or querying capabilities. Because of this, it is not a silver bullet, but a simple-to-use persistence mechanism is sufficient for the needs of many applications.

PSS provides the Persistent State Definition Language (PSDL), which is a superset of IDL. PSDL is used to define data-types that can be persisted. A PSDL compiler then translates the PSDL definitions into corresponding definitions in a programming language (such as C++ or Java). The PSDL compiler also generates the code required for these programming-language objects to be persisted in, and retrieved from, a file or database.

The long-term goal is that there will be PSS implementations for a wide variety of different storage devices, such as files, object databases and relational databases. An application developer who uses the PSS APIs will be able to easily switch from using a file-based PSS implementation to a relational database-based PSS implementation or vice versa. However, this long-term goal has not been fully met yet. Several CORBA vendors provide implementations of PSS, but it is common for a vendor to provide a mapping from PSS to only one type of persistent storage device. You can expect a CORBA vendor to be willing to implement PSS support for additional brand names of database or other storage technologies *if* there is enough customer interest for such support. This can easily become a chicken and egg situation: customers are likely to avoid PSS unless it *already* supports multiple storage technologies, and vendors will not support multiple storage technologies *until* they get sufficient PSS customers to make it worthwhile. An important point about this is that before deciding to use PSS, you should check that your CORBA vendor (or a third-party vendor) can provide you with PSS implementations for all the persistent storage technologies that you might require.

There are two final points to note. First, a good, technical overview of PSS can be found in the *Java Programming with CORBA* book [BVD01]. Second, PSS-based persistence is one of the infrastructure services that is provided by the CORBA Component Model (CCM), which is briefly discussed in Section 19.3 on page 182.

24.2 Other CORBA Services and Domain Specifications

The OMG web site (`www.omg.org`) provides free-to-download PDF documents of all the different CORBA-related specifications. It is worthwhile browsing that web site in order to see the extensive range of services defined by CORBA. The services are *optional* parts of CORBA, so your CORBA vendor is likely to sell just a subset of them.

It should be noted that the term *CORBA Services* is used to refer to optional functionality that might be useful to applications in *many* different types of organization. Naming, Events and Transactions are examples of such services. The OMG recognizes that it is useful to define standards for functionality that has a more specialized purpose. Such functionality is called an *OMG Domain Specification*. The OMG web site lists domain specifications for a diverse range of specialized areas, such as air traffic control, audio/video streaming, computer aided design, telecommunications, finance, accountancy, manufacturing, distributed simulations, and life science research. The division between what is a CORBA Service and what is an OMG Domain Specification is somewhat subjective, and several specifications are listed in both categories on the OMG web site.

If your vendor does not implement a particular CORBA Service or OMG Domain Specification that you require then you might be able to purchase an implementation of the required functionality from a third-party company. You can use an Internet search engine (for example, `www.google.com`) to locate companies that sell implementations of particular CORBA Services or OMG Domain Specifications.

Part VI

Final Issues

Chapter 25

Portability of CORBA Applications

Most of the source-code of a CORBA application is portable across different CORBA products. In fact, portability is so good that it is easier to discuss the few areas in which portability is lacking. The 80/20 principle [Koc00] applies to porting: most of the difficultly of porting an application is rooted in a relatively small number of issues. By being forewarned about these issues, a development team can pro-actively work to avoid common pitfalls and so make porting easier and cheaper. This chapter focuses on C++ and Java because those are the languages that the author has used for CORBA development.

25.1 CORBA Portability Issues

CORBA has standardized the mapping from IDL to many different programming languages, such as C++, Java, Cobol, Ada and so on. This means that the way a programmer manipulates a `struct` or `union`, makes a remote call using a proxy or implements an `interface` with a servant class is exactly the same across all C++ CORBA products, is exactly the same across all Java CORBA products, and so on. However, there are still a few CORBA-related portability issues to watch out for, as the following subsections discuss.

25.1.1 Makefile Issues

CORBA has not standardized the name of the IDL compiler or the command-line options that it takes. Neither has CORBA standardized the names of libraries with which CORBA programs must be linked. A practical ramification is that a `Makefile` (or equivalent, such as a Microsoft Visual Studio Project file) will need to be modified if you switch from using one CORBA product to another. Making such modifications is usually a straightforward, albeit somewhat tedious task.

25.1.2 Names of CORBA-related C++ Header Files

The IDL-to-C++ mapping defines the APIs of generated data-types. Unfortunately, the mapping does *not* standardize the names of the generated source-code *files* in which the data-types reside. For example, given the file `foo.idl`, the Orbix/C++ IDL compiler generates `foo.hh`, `fooS.hh`, `fooC.cxx` and `fooS.cxx`. IDL compilers provided with other CORBA products are likely to use different suffixes on the names of generated files.

Changing the names of source-code files listed in a `Makefile` is usually not a big problem. What can be a *much* bigger problem is changing numerous `#include` directives in source-code files, because most source-code files in a project may `#include` one or more CORBA-related header files. The *Portability of C++ CORBA Applications* chapter of the *CORBA Utilities* package [McH] discusses this issue in depth and explains how investing a few hours of up-front work on this issue at the start of a project can produce portable `#include` directives that can save days or even weeks of frustration later on when porting an application.

25.1.3 Configuration and Logging APIs

CORBA does not provide standardized APIs for: (1) obtaining runtime configuration values, or (2) directing diagnostic messages to a log file. However, most CORBA products provide proprietary APIs for these purposes. This is because the internals of CORBA products require such capabilities, and CORBA vendors often decide to expose these internally-required capabilities to application developers. If you avoid use of these proprietary APIs then porting to a different CORBA product will be much easier. If you *must* make use of such proprietary APIs then you should not access them directly, but rather write a thin "portability wrapper" API around them. In this way, you minimize the amount of code that must be modified when porting an application.

25.1.4 Implementation Repositories

As Chapter 7 discusses, CORBA has not standardized upon the "look and feel" of *implementation repositories* (IMRs) and, for this reason, there is considerable difference between the IMRs of different CORBA products. Put simply, there is no portability in the administration of CORBA applications across different CORBA products.

Some server deployment models (Section 8.2.2 on page 88) require use of proprietary APIs in some CORBA products. Direct use of these proprietary APIs can obviously hinder portability. The *Creation of POA Hierarchies Made Simple* chapter of the *CORBA Utilities* package [McH] discusses a Java and C++ class that provides a "portability wrapper" around such proprietary APIs, while simultaneously simplifying application development.

25.1.5 Multi-threaded Servers

Section 6.1.2.1 on page 61 explained that the `ORB_CTRL_MODEL` POA policy has under-defined semantics. Most CORBA products either implement a thread pool for this policy or offer a thread pool as one of a configurable set of alternatives for this policy. For this reason, it is reasonably portable to assume that this POA policy is implemented as a thread pool, where the size of the thread pool is determined through an entry in a configuration file, and that the thread pool is created automatically by the CORBA runtime system. However, one notable exception to this is TAO, which is a C++ freeware implementation of CORBA. TAO *does* support thread pool semantics for the `ORB_CTRL_MODEL` POA policy. However, TAO requires that the application programmer create all the threads in the thread pool, and each of these threads must call `orb->run()`, which is how the application-level threads become part of the thread pool.

The pseudocode in Figure 25.1 shows how to write a *portable* C++ server that uses a thread pool for `ORB_CTRL_MODEL` POAs. The important point is to have two portability "wrapper" functions for creating and destroying a thread pool. Usage of these is shown at lines 1 and 3, respectively. All but one of the threads for the thread pool are created at line 1; the last thread in the thread pool is the main thread that calls `orb->run()` directly (line 2).

The implementation of the portability wrapper functions is given in Figure 25.2. The implementation uses the existence of the `P_USE_TAO` preprocessor symbol (a discussion of which can be found elsewhere [McH, Ch. 4]) to choose between the TAO-specific code for creating and destroying a thread pool, or dummy implementations of the functions that are appropriate for most other C++ CORBA implementations.

```
int main(int argc, char* argv[])
{
    thread_pool_size = ...;
    exit_status = 0;
    orb = CORBA::ORB::_nil();
    try {
        orb = CORBA::ORB_init(argc, argv);
        ... // create POAs to contain servants
        ... // create servants and activate into POAs
        ... // export object references
        ... // activate POA managers
1       create_tao_thread_pool(orb, thread_pool_size - 1);
2       orb->run();
    } catch(const CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
3   wait_for_tao_thread_pool_to_terminate();
    // Terminate gracefully
    try {
        if (!CORBA::is_nil(orb)) { orb->destroy(); }
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
    return exit_status;
};
```

Figure 25.1: Server mainline using portability abstraction for TAO thread pools

25.2 Non-CORBA Portability Issues with C++

When porting an application from one CORBA product to another, developers sometimes also switch from one compiler to another or from one operating system to another. Such switches of compiler or operating system do not affect Java-based applications very much, because Java provides a “virtual machine” that is (supposed to be) portable across all Java compilers and operating systems. Actually, when Java was first released, there were promises that it was a “write once, run everywhere” language. When developers noticed that Java-based applications did not run as fast as C or C++-based applications, some cynics said that Java was a “write once, crawl everywhere” language. Some

```
#ifdef P_USE_TAO
#include "ace/Task.h"
class Worker : public ACE_Task_Base {
    CORBA::ORB_var    m_orb;
public:
    Worker(CORBA::ORB_ptr orb) {
        m_orb = CORBA::ORB::_duplicate(orb);
    }
    virtual int svc(void) {
        try {
            m_orb->run();
        } catch(...) { }
        return 0;
    }
};
static Worker * w = 0;
void create_tao_thread_pool(CORBA::ORB_ptr orb, int count)
    throw(std::string)
{
    w = new Worker(orb);
    if (w->activate(THR_NEW_LWP | THR_JOINABLE, count)!=0)
    {
        delete w;
        w = 0;
        throw std::string("Cannot create thread pool");
    }
}
void wait_for_tao_thread_pool_to_terminate() {
    if (w != 0) {
        w->thr_mgr()->wait();
        delete w;
        w = 0;
    }
}
#else // dummy implementation for other CORBA products
void create_tao_thread_pool(CORBA::ORB_ptr orb, int count)
    throw(std::string)
{ }
void wait_for_tao_thread_pool_to_terminate() { }
#endif
```

Figure 25.2: Portability wrapper for TAO thread pools

other developers encountered subtle differences in behavior in the Java Virtual Machine (JVM) on different operating systems and so said that Java was a "write once, debug everywhere" language. However, problems with performance and JVM differences have decreased over the years and, nowadays, Java applications tend to be very portable.

Switching compilers or operating systems tends to be much more troublesome for C++-based applications than for Java-based applications. The following subsections discuss some problematic areas that you should watch out for in C++ applications.

25.2.1 Cross-platform Portability

For some reason, supposedly portable applications developed on Windows often accidentally use Windows-proprietary APIs.[1] This is usually discovered only when the application is then ported to a UNIX platform. By that time, the use of Windows-proprietary APIs may be so widespread in the application that the porting becomes a major effort. For example, the Windows-proprietary `CString` class is often accidentally used in supposedly portable applications when the standard (and hence portable) C++ class `std::string` would do just as well. Some companies who have found themselves with a supposedly portable application that has been accidentally tied to `CString` have found it quicker to *reverse engineer* the `CString` class so they can write a UNIX version rather than remove the use of `CString` from their application. Some companies end up reverse engineering large parts of Microsoft's MFC class library so that they can write a UNIX version, just to port a Windows application to UNIX.

To guard against such difficulties in the future porting of applications, it is useful if at least one developer with some UNIX experience works on the team that has the task of developing a portable application initially on Windows.

If your CORBA applications will have a graphical user interface (GUI) then you might wish to consider use of a cross-platform GUI toolkit. One such toolkit is wxWindows (`www.wxwindows.org`).

25.2.2 The `iostream` Library

Although C++ dates from the early 1980s, the language was not standardized until the mid-1990s. In pre-standardized C++, many header files had `".h"`

[1] The reverse is not true. Supposedly portable applications developed on UNIX tend to be *much* easier to port to Windows than supposedly portable applications developed on Windows are to port to UNIX.

extensions, for example, `<iostream.h>`. The standardization committee decided to make two important changes to standard header files:

1. The `".h"` extension was dropped, for example, `<iostream.h>` became `<iostream>`.

2. The types and variables defined in these standard header files were defined in the `std` namespace rather than in the global scope, for example, `cout` became `std::cout`.

In many compilers, the new standardized types are *not* type compatible with the older, pre-standardized types. Many projects have wasted weeks or months in porting code originally written for use with the pre-standardized types to use the standardized types. It is not uncommon for this porting headache to raise its head only when an application is being ported from one CORBA product or operating system to another. The *Portability of C++ CORBA Applications* chapter of the *CORBA Utilities* package [McH] discusses this issue in depth and explains how investing a few hours of up-front work on this issue can result in the ability to write code that compiles correctly with either the pre-standardized or the standardized C++ library. The advice in that document can dramatically reduce the time required to port applications.

25.2.3 Synchronization in C++ Applications

The C++ language does not define standard APIs for synchronization. Instead, synchronization APIs vary a lot from one operating system to another. Many UNIX platforms now support the POSIX Threads standard, but it is still possible to use proprietary APIs on some flavors of UNIX. Also, Windows uses its own proprietary APIs instead of POSIX APIs. Because of this, porting a multi-threaded application between different operating systems can involve an enormous amount of tedious work if the application was written to use the native synchronization APIs of the original platform.

Many organizations have tried to write their own “portability wrapper” around the low-level synchronization APIs of an operating system. However, doing this is fraught with difficulties. For example, it is very difficult to *correctly and efficiently* emulate POSIX condition variables with the synchronization APIs of Windows.

Many CORBA products work on several operating systems and it is common for a CORBA product to provide its own, proprietary “portability wrapper” for synchronization APIs. Use of these libraries provides portability across different operating systems, but makes it difficult to port an application from one CORBA product to another.

The Generic Synchronization Policies (GSP) class library [McH, Ch. 7] provides cross-platform synchronization support and is not tied to a particular CORBA product. It offers an additional benefit in that it does not try to mimic low-level APIs provided by an operating system, but rather provides a high-level API that simplifies the writing of multi-threaded programs.

Chapter 26

Other CORBA Resources

26.1 Books and Articles

Most CORBA products are provided with manuals although, as you might expect, the quality of the documentation varies from one product to another. In general, you are likely to obtain less documentation with a freeware CORBA implementation than with a commercial product. Regardless of which CORBA product you choose, you may wish to supplement its documentation with a book. A good way to choose a book is to visit `www.amazon.com` and use its search engine to help you browse CORBA books. The customer reviews will help you choose a good book. Some of this author's favorite CORBA books are listed below:

- *Pure CORBA* [Bol01] is aimed at developers who are new to CORBA. It talks the reader through the concepts of CORBA and provides lots of useful code examples in both C++ *and* Java. Providing examples in C++ and Java means that the book is certainly of relevance to developers who use one of those languages. However, there is another benefit of this dual-language approach. When people learn CORBA through one specific language, often they are unable to distinguish between what is a general principle of CORBA and what is specific to the particular programming language that they use. The *Pure CORBA* approach of teaching CORBA through *two* languages helps readers to distinguish between general CORBA principles and language-specific issues.

- *Advanced CORBA Programming with C++* [HV99] is *not* an introductory book on CORBA, but rather is an excellent book for people who are already familiar with CORBA to improve their skills. Although the book uses C++ in all the code examples, the principles it teaches are relevant to CORBA developers who use other languages.

- *IIOP Complete* [RHK99] should be avoided *unless* you have a need to learn about the low-level details of the GIOP and IIOP protocols (Chapter 11). However, if you *do* have such a need then this book provides a very clear explanation of the concepts. This book is now out of date, because it discusses versions 1.0 and 1.1 of GIOP, while CORBA is now at version 1.3. However, although some of the details have changed between different versions of the protocol, the basic principles are the same. Because of this, if you need to become familiar with GIOP 1.2 or 1.3 then a good way to do so is to read this book and then download the latest GIOP specification (as a PDF file) from the OMG web site.

Douglas Schmidt, who headed the development of TAO, and Steve Vinoski, who headed the development of Orbix, have published many interesting articles on CORBA. You can obtain electronic versions of many of their papers from their web pages:

```
www.iona.com/hyplan/vinoski/
www.cs.wustl.edu/~schmidt/
```

26.2 The CORBA Utilities Package

The *CORBA Utilities* package is a collection of documented software utilities that dramatically simplify CORBA development. The collection is available as a free download from the following URL:

```
www.iona.com/devcenter/corba/utilities.htm
```

The utilities are available in both C++ and Java, and are known to work out-of-the-box with Orbix, Orbacus, TAO and omniORB. A lot of attention has been paid to portability of the utilities so it should be quite easy to get the utilities working with other CORBA products. Even if you decide to not use the utilities in your own projects, the documentation provided with the utilities is worth reading for the useful advice that it provides.

26.3 Internet Resources

The OMG web sites (`www.omg.org` and `www.corba.org`) provide free access to a lot of information on CORBA. You can download CORBA specification documents in the form of PDF files. The web sites also provides links to other CORBA-related online resources.

The following Internet newsgroups are for discussing CORBA:

```
comp.object.corba
comp.lang.java.corba
```

Contributors on those newsgroups (which include some employees of several CORBA vendors) are usually happy to answer questions and offer advice.

Some CORBA vendors have their own newsgroups and/or mailing lists dedicated to their own products. You should ask your CORBA vendor for details.

As discussed in Section 2.7 on page 20, you can find numerous examples of CORBA success stories on the OMG web sites and the web sites of CORBA vendors.

26.4 Consultancy and Training Courses

Some CORBA vendors, and also some independent companies, offer CORBA training courses and consultancy. You can contact an individual company to see what services they offer. Another option is to use an Internet search engine to find companies that offer CORBA training and consultancy.

Consultancy is often perceived as being expensive. Indeed, it can be if your organization hires a consultant for the entire duration of a long-term project. However, there is a much more cost-effective way of using consultancy services:

- When you are starting a CORBA-based project, have a consultant visit you for a few days, to help you define or “sanity check” the project’s architecture. Experienced consultants can often spot architectural flaws that, if not corrected, would have a significant performance or scalability impact later in the project.

- Later, when doing the initial coding for a project, have the same consultant return to mentor developers in the use of good coding idioms and help them avoid common coding mistakes.

- The consultant could then return for a few days every month to “sanity check” progress on the project. In this way, any deviation from good CORBA practice can be spotted and corrected relatively soon, thus saving the project from greater problems (and expense) later on.

Many organizations make the mistake of sending developers on a training course to learn new skills that will not be used until many months later. By the time the skills are finally required, the developers will have forgotten them. If feasible, it is much more productive to have an on-site, CORBA training course *at the start of a project*. In this way, the developers will be able to practice the new skills in their work straight away. Also, by timing the training course to coincide with the start of the project, the developers will be able to ask the course instructor questions about how to use what they learn from the course in their project.

A training course will be the most effective if the instructor is a consultant who is *already familiar* with your intended project. In such cases, the instructor is often able to pro-actively point out to developers how specific CORBA concepts and coding idioms can be used effectively in the project.

Much of the information in this book is taken from material in training courses offered by IONA technologies. If this book has provided you with a useful overview of the concepts of CORBA then you might want to consider attending an IONA training course to obtain the skills necessary for development of CORBA applications.

Bibliography

[Bla99] Bob Blakely. *CORBA Security: An Introduction to Safe Computing With Objects*. Addison-Wesley, 1999. ISBN 0201325659.

[Bol01] Fintan Bolton. *Pure CORBA*. Sams, 2001. 921 pages. ISBN 0672318121.

[BVD01] Gerald Brose, Andreas Vogel, and Keith Duddy. *Java Programming with CORBA, third edition*. Wiley Computer Publishing, 2001. 710 pages. ISBN 0471376817.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999. 1120 pages.

[Koc00] Richard Koch. *The 80/20 Principle: The Secret of Achieving More With Less*. Nicholas Breealey Publishing Ltd., May 2000. ISBN: 187881680. 312 pages.

[KPS02] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 2002. ISBN 0130460192.

[MBB00] H. X. Mel, Doris Burnett, and Doris M. Baker. *Cryptography Decrypted*. Addison Wesley, 2000. 256 pages. ISBN 0201616475.

[McH] Ciaran McHale. CORBA Utilities. Available at `www.CiaranMcHale.com/download/`.

[OMGa] OMG. Common Security Interoperability (CSIv2). Available for download from `www.omg.org` both as a stand-alone document and as a chapter in the CORBA Specification.

[OMGb] OMG. CORBA Security Specification (verison 1.8). Available for download from `www.omg.org`.

[RHK99] William Ruh, Thomas Herron, and Paul Klinker. *IIOP Complete: Middleware Interoperability and Distributed Object Standards*. Addison-Wesley, 1999. 288 pages. ISBN 0201379252.

[Sch95] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons Inc, 1995. ISBN 0471128457.

[SGR99] Dirk Slama, Jason Garbis, and Perry Russell. *Enterprise CORBA*. Prentice Hall PTR, 1999. 366 pages.

[SV99a] Douglas C. Schmidt and Steve Vinoski. Programming Asynchronous Method Invocations with CORBA Messasing (Object Connections, column 16). *SIGS C++ Report*, 11, February 1999. Available from Steve Vinoski's web page at `www.iona.com/hyplan/vinoski/` or from Doug Schmidt's web page at `www.cs.wustl.edu/~schmidt/`.

[SV99b] Douglas C. Schmidt and Steve Vinoski. Time-Independent Invocation Interoperable Routing (Object Connections, column 16). *SIGS C++ Report*, 11, April 1999. Available from Steve Vinoski's web page at `www.iona.com/hyplan/vinoski/` or from Doug Schmidt's web page at `www.cs.wustl.edu/~schmidt/`.