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 94.

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 100.

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 94, 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 official language mappings provide source-code portability of CORBA 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 108)—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.

Firstly, 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.

Secondly, 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 does *not* define any language mappings. Instead, 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 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 formatting programming-language types into XML format and later parsing the XML to extract the embedded programming-languages types.

[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.

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.