Principles of Object-Oriented Middleware

Learning Objectives

The communication between objects that are distributed across a network is enabled by middleware. In this chapter we will grasp the basic principles of such middleware. We will learn that midwellers is a layer between the network operating system and distributed applications. We will understand how middleware relates to the ISO/OSI reference model. We will comprehend why middleware is built upon the transport layer and how it implements the session and presentation Layer. We will acquaint ourselves with different forms of middleware and understand the origins of object-oriented middleware. We will finally investigate the process of developing distributed objects using object-oriented middleware.

Chapter Outline

3.1 Computer Networks
3.2 Types of Middleware
3.3 Object-Oriented Middleware
3.4 Developing with Object-Oriented Middleware
In the previous chapter, we discussed the design of objects. We argued that object-orientation provides a suitable component model for distributed systems. We have identified object requests as the most important concept; it enables a client object to request execution of an operation from a server object. In distributed systems, client and server objects generally reside on different hosts. Hence, object requests may have to be communicated via a network that connects the hosts.

Computer networks facilitate the physical interchange of electrical or optical signals as packets of information. The network may detect and even correct transmission errors. A network implements the routing of packets between different hosts and takes care of the composition of such packets into byte streams or fixed-length messages. Network support is built into most modern operating systems. We, therefore, refer to them also as network operating systems.

Application engineers could use network operating systems to exchange information based on sending fixed-length messages or writing into a byte stream. Many distributed systems are actually built this way and for systems that need non-standard features or that are performance-critical this is appropriate. It is, however, often not advisable to do so; there is a substantial conceptual gap between the capabilities of network operating systems - reading and writing from and to byte streams - and what designers are interested in - requesting operations from remote objects. If network operating systems are used directly the application engineer will have to bridge this conceptual gap. This is time-consuming, error-prone and distracts the designer's attention from the real problems.

Middleware is layered between network operating systems and distributed applications in order to bridge this gap. Middleware provides the application designer with a higher-level of abstraction. Middleware implements this higher level of abstraction based on primitives that are provided by network operating systems. While doing so it hides the complexity of using a network operating system from application designers. The idea of middleware is not new. Many different types of middleware have been suggested and built and are being used in industrial practice. In this book we are mainly concerned with object-oriented middleware. Object-oriented middleware facilitates object requests between distributed objects. It implements requests by using primitives provided by network operating systems.

This chapter explains the principles of middleware and then takes an object-oriented perspective. We first look at networks, the base upon which middleware is built. We investigate the different forms of middleware that exist: transaction monitors, message-oriented middleware and remote procedure calls. We then take an object-oriented perspective on middleware and finally discuss the development support that object-oriented middleware systems provide.

3.1 Computer Networks

3.1.1 ISO/OSI Reference Model

The Open Systems Interconnection (OSI) reference model was defined in 1977 by the International Organization for Standardization (ISO). It identifies the need for standardiza-
tion of the communication between hosts built by different manufacturers. We do not want to
discuss the ISO/OSI reference model in full detail but rather recall the basic principles that we
will refer to later. The ISO/OSI reference model is a layered architecture in which each layer
builds on abstractions provided by the layer below. Figure 3.1 shows a graphical overview
of the model.

The **physical layer** specifies the mechanical, electrical and optical behaviour of the plugs and
sockets of a physical interface to the network. Implementations of the physical layer perform
functions in order to signal, modulate and synchronize the transmission of bits over the
physical network. In addition, it may perform error detection by monitoring the quality of the
electrical or optical signals received.

Conceptually, a network is a graph of interconnected nodes. The **data link layer** implements
edges in these graphs. These nodes may be routers, switches or computers. It maps the physical
connection provided by the physical layer into a layer that can transmit packets from one node to
another in a relatively error-free manner. The data link layer provides error correction based on
physical error-detection primitives provided by the physical layer.

While the data link layer is concerned with node-to-node connections, the **network layer**
implements the connection of multiple nodes into a network. As such, it is concerned with
routing messages along multiple edges in the graph of nodes. Thus, the network layer
isolates the higher layers from data link considerations and implements connections between
arbitrary nodes of the network.

The **transport layer** adds the capability of mapping longer messages or even continuous
sequences of data onto these package-routing capabilities. If the network layer does no
provide reliable package routing then transport layer implementations add such reliability
Transport layer implementations also map transport layer identifiers onto network
addresses.

Some transport layer implementations are connection-less. For these transport layers, the
**session layer** provides facilities to establish and maintain connections between two or more
distributed components. If the transport layer implementation is connection-oriented, the
session layer merely uses the connection mechanisms of the transport layer.
Principles of Object-Oriented Middleware

Being able to transmit streams of bytes between two or more distributed system components might not be sufficient for the communication between two or more components. Very often these components want to communicate high-level data structures, such as records, arrays, and sequences. These higher-level data structures need to be converted into a form that can be transmitted via a connection that can transmit streams of bytes. Moreover, components might be implemented on heterogeneous hosts. Heterogeneous hosts may use considerably differing representations for atomic data structures. Mainframes, for example, use big-endian representations for integers while Unix workstations tend to use little-endian representations. Presentation layer implementations resolve such differences in the representation of information between distributed components.

The application layer is concerned with distributed components and their interaction. In the setting of this book, the application layer will be implemented using distributed objects. The object types are defined according to the meta-model of the respective object-oriented middleware. The middleware provides an interface definition language that can express all the concepts of the meta-model. Distributed objects request operation execution from each other and the presentation layer implementation is used to resolve heterogeneity and to map complex request parameters onto a form that can be transmitted via the transport layer.

We are going to review the transport layer of the model in more detail now. Transport layer implementations provide a sufficiently powerful basis for implementing distribution middleware. Middleware systems are therefore typically layered above the transport layer and they implement the presentation and session layers.

3.1.2 Transport Layer

Network operating systems may provide different implementations of the transport layer. This is motivated by the differing characteristics of network layer implementations upon which transport layers are implemented. Moreover, different kinds of application may have different and sometimes conflicting requirements for the transport layer.

We generally distinguish connection-oriented transport layer implementations from connection-less transport layer implementations. Connection-oriented transports maintain a connection between two distributed components on behalf of upper layers. They provide operations to open and close a connection, to write data into such a connection and to read data from such a connection. Connection-oriented transports, therefore, provide upper layers with a mechanism to exchange data through streams of bytes. Connection-less transports provide upper layers with the capability of sending fixed-length messages between distributed hosts. These messages are referred to as datagrams. If a connection-less transport protocol is used, implementations of the session layer are needed to associate sent and received messages with certain applications.

Different implementations of the physical, data link and network layers may provide differing degrees of reliability. An optical wire, for example, that is used with an Asynchronous Transfer Mode (ATM) network implementation would typically provide a higher degree of reliability than an analogue and noisy telephone line that is used with a Serial Line Internet Protocol (SLIP). As upper layers need to be provided with a considerable degree of
reliability, the transport layer needs to implement reliable transport even if the underlying network is unreliable. If a reliable network were used, on the other hand, implementing reliability also in the transport layer would reduce the performance of distributed applications unnecessarily. If a reliable physical or data link layer is being used, it is not necessary to implement error correction in the transport layer. However, most operating systems can interact with different data link layers and they use the same transport layer for all of them.

Unix operating system implementations generally provide two different implementations for the transport layer. These are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is connection-oriented and provides a reliable form of transport. It is now also available for Microsoft's operating systems. UDP is connection-less and it provides unreliable transport of data. Whether TCP or UDP is best used depends on whether or not an application needs to retain the connection for longer periods of time. Applications, such as who, which checks for logins on all hosts of a local area network, use UDP in order to avoid" having open connections to large numbers of computers. Applications, such as ftp, that need a reliable connection for prolonged periods of time establish TCP connections.

**Transmission Control Protocol**

TCP provides upper layers on one host with the ability to write unstructured data into a stream of bytes so that upper layers of a component on another host can read data from that stream. At the same time, the second component can write unstructured data to be returned into the stream so that upper layers of the first component can read the result.

Figure 3.2 sketches the use of bi-directional TCP byte streams for distributed requests between a client and a server components. After having opened a bi-directional channel, the presentation layer of the client maps service parameters into a form that can be transmitted via a stream of bytes. It then writes the parameters into the input stream. The presentation layer implementation in the server component reads from the input stream and reconstructs the request parameters. Upon completion of the request, the presentation layer implementation of the server component transforms the request results into a stream of bytes and writes those into the output stream. The presentation layer implementation of the client component then reads from the output byte stream and re-constructs the results. The client then closes the TCP connection.
The implementation of TCP has to map the higher-level of bi-directional byte streams onto capabilities that the network layer implementation provides. TCP is often layered on top of the Internet Protocol (IP). IP supports the routing of packets between hosts. Network layer implementations may not retain the sequence of packets as they may be sent along different routes. Two IP packets, for example, might arrive in the reverse order to that in which they were sent. Hence, the implementation of TCP for IP will have to buffer packets on the receiving host and sort them on the basis of sequence numbers, so that the overall order of bytes is retained.

TCP is often used with unreliable network protocols. To provide a higher degree of reliability to upper layers, it implements several reliability mechanisms. In order to detect transmission errors, TCP applies checksums to all transmitted data. It uses these checksums and sequence numbers to detect missing, defect and duplicated packets. TCP employs a re-transmission strategy so that it can usually recover from errors of the underlying network protocol implementation in a way that is transparent to the programmer implementing an upper layer.

Transport layer implementations are available to the middleware programmer in the form of sockets. Sockets are available on all Unix workstations and servers as well as the various Microsoft operating systems. On Unix and Windows, there are two different types of socket, SOCK_STREAM and SOCK_DGRAM. Sockets of type SOCK_STREAM provide the programming interface to TCP. A combination of an Internet domain name and a port number identifies a socket. The socket interface includes operations to create sockets and to bind port numbers and domain names to a socket.

**User Datagram Protocol**

UDP is a relatively simple transport layer implementation. It provides the capability to send messages containing blocks of bytes with a fixed length to other distributed components. UDP provides a slightly lower level of abstraction than TCP. While TCP is connection-oriented, UDP is connection-less. Moreover, UDP only provides a uni-directional flow of messages.

Figure 3.3 sketches how a middleware implementation uses UDP for the implementation of interactions between distributed system components. The middleware does not have to open a connection between client and server components. It rather decomposes the request data into a (usually) small number of request datagrams. The middleware then uses UDP to transport these request datagrams to the host of the server object, where the request information is decoded and processed. The result of the operation is then treated in the same way and sent as a UDP message back to the requester.

Depending on the message size of the UDP implementation (usually 9 KB) and the package size used in the underlying network (for example the IP maximum transfer unit is 1500 bytes for ethernet-based adapters), UDP implementations have to fragment messages into one or more network packets. UDP does not guarantee that the messages arrive in the order they were sent. UDP buffers messages for the receiver in the order of their arrival. Senders of messages Must, therefore, encode a sequence into the message contents to enable upper layers on the receiving host to re-establish the original sequence.
UDP provides a checksum field so that receivers know that the packets that they receive contain the correct data. Unlike TCP, however, UDP adds no further delivery guarantee to the network layer implementation. It is therefore, often used in local area networks in which the physical, data link and network layer implementations provide sufficient degrees of reliability already. As a direct result, UDP is an efficient protocol, as it does not spend any resources on error-detection and correction. Moreover, there is no overhead for connecting and listening to connections. As object requests can often be represented in byte packets of 9 or 18 KB, one or two UDP packets are sufficient for sending an object request. This makes UDP a particularly attractive protocol for the implementation of object requests for middleware that is used over high-quality networks, such as many local area networks.

In order to implement object requests with UDP, middleware vendors use sockets. The programming interface to UDP on Unix systems is provided by sockets of type SOCK_DGRAM. With a socket of that type, middleware implementations can use send to in order to send a datagram of a given length to a receiving socket on another host. A middleware implementation on that host might call recvfrom in order to read the datagram from the socket.

3.2 Types of Middlewares

We have now developed an understanding of the transport layer. If application engineers had to rely directly on transport layer implementations, such as TCP or UDP, they would have to implement the communication between distributed system components based on writing to and reading from byte streams or fixed-length messages. We regard this as inappropriate for the following reasons:

1. The communication will often need to send complex parameters, such as records, character strings or arrays. Application engineers would have to map those to sequences of bytes.
2. Different encodings of data types in memory might be used if clients and server objects are not deployed on the same hardware platform and if they are not written in the same programming language. Application engineers would have to map those onto each other.
3. Parameters and return values might refer to other distributed system components. Application engineers would have to implement object references in terms of Internet domain names, port numbers and additional addressing information.

4. Application developers and administrators would have to implement component activation, that is the ability of a server component to respond to a client request even when it is not currently running in an operating system process.

5. Given this referencing and encoding, type safety is an important concern. Type safety denotes the guarantee that an operation requested by a client is actually implemented by a server component and that the actual parameters supplied by the client must match the service definition. Achieving type safety manually is quite tedious and error prone, particularly if client and server objects are developed by different engineers.

6. After sending the request, the client object needs to wait for the result to come back, either from the output stream in TCP or by a reply message in UDP. Again, implementing this synchronization manually is rather tedious and error prone.

7. Sometimes there are qualities of service required that cannot be guaranteed by the network level. It may, for example, become necessary for different service requests to be implemented in an atomic fashion, that is either completely or not at all.

Fortunately, application engineers do not have to engage in these activities but can rely on distribution middleware. Middleware maps complex parameters passed to service requests into a transmittable form; resolves heterogeneity; resolves logical component addresses to Internet domain names and port addresses; and synchronizes client and server objects. While doing so, middleware implements the session and presentation layers of the ISO/OSI reference model.

The requirement to build applications in a heterogeneous and distributed setting is not new. Solutions for resolving heterogeneity and distribution in the form of middleware have been available for some time, too. The available middleware can be classified into three major categories: transaction-oriented, message-oriented and object-oriented middleware. Their differences can be explained by their initial goals.

3.2.1 Transaction-Oriented Middleware

Transaction-oriented middleware supports transactions across different distributed database systems. Hence, this type of middleware is often used in architectures where components are database applications. Transaction-oriented middleware uses the two-phase commit protocol [Bernstein et al., 1987] to implement distributed transactions. The products in this category include IBM's CICS, BEA's Tuxedo and Transarc's Eneina. These products assume that participating database management systems implement the two-phase commit protocol.

The Open Group have adopted a standard for Open Distributed Transaction Processing (DTP). DTP is widely supported by relational and object-oriented database management systems and transaction-oriented middleware. [X/Open Group, 1994] defines the XAProtocol for two-phase commit.
Being able to integrate updates in distributed databases also provides the foundation for fault-tolerance and load-balancing [Birman, 1997]. Many transaction-oriented middleware systems use distributed transactions to maintain replicated databases on different servers. Updates to these databases are synchronized using distributed transactions. Load-balancing is used to direct queries to the least-loaded replica.

In Chapter 11, we will discuss transactions between distributed objects. These distributed object transactions build on the XA-Protocol of the DTP standard and are now widely available in distributed object middleware, too.

2.2 Message-Oriented Middleware

Message-oriented middleware supports the communication between distributed system components by facilitating message exchange. Products in this category include IBM's MQSeries, Sun's Tool Talk and NCR's TopEnd. Client components use these systems to send a message in order to request execution of a service to a server component. The content of such a message includes the service parameters. Another message is sent to the client from the server to transmit the result of the service.

A strength of message-oriented middleware is that this paradigm supports asynchronous message delivery very naturally. The client continues processing as soon as the middleware has taken the message. Eventually the server will send a message including the result and the client can collect that message at an appropriate time. This achieves de-coupling of client and server and leads to more scalable systems.

A further strength of message-oriented middleware is that it supports multi-casting; it can distribute the same message to multiple receivers in a way that is transparent to clients. This is a useful primitive for implementing event notification and publish/subscribe-based architectures.

Message-oriented middleware systems achieve fault-tolerance by implementing message queues that store messages temporarily on persistent storage. The sender writes the message into the message queue and if the receiver is unavailable due to a failure, the message queue retains the message until the receiver is available again.

Object-oriented middleware initially did not achieve this degree of fault-tolerance, but only supported synchronous communication with at-most-once semantics. We will discuss in Chapter 7 the integration of message-oriented middleware and object-oriented middleware, which achieves reliable and asynchronous object requests.
3.2.3 Remote Procedure Calls

Remote Procedure Calls (RPCs) were invented in the early 1980s by Sun Microsystems as part of their Open Network Computing (ONC) platform. RPCs are operations that can be invoked remotely across different hardware and operating system platforms. Sun provided remote procedure calls as part of their Sun OS, an early Unix implementation. Sun sub
mitted RPCs as a standard to the X/Open consortium and it was adopted as part of the Distributed Computing Environment (DCE).

Remote procedure call systems are the origin of object-oriented middleware. In some systems RPCs are even used to implement distributed object requests. We now study how remote procedure calls implement the presentation and session layers so that we can, in the next section, discuss how object-oriented middleware extends remote procedure call systems.

**Interface Definition Language**

The server components that execute RPCs are called *RPC programs*. The RPC program has an interface definition that defines procedures that can be called remotely. Interface definitions also define data types of arguments that can be passed to remote procedures. The DCE standard includes an interface definition language that is used to define these exports in a way that is independent of programming languages. Example 3.1 shows an interface definition in Sun's RPC interface definition language.

**Presentation Layer**

The presentation layer implementation of RPCs maps application data structures, such as the Player data structure, into a form that can be transmitted via the transport layer. This involves two principal tasks: resolution of data heterogeneity and mapping of complex data structures into blocks of bytes. An overview of the tasks for the presentation layer implementation of middleware given in Figure 3.4.

As we have seen in Chapter 1, distributed systems are often deployed across different hardware platforms. Some of these platforms use different representations for representing data. One of the principal responsibilities of the presentation layer is to resolve this data heterogeneity between client and server objects. We will study the techniques for data heterogeneity resolution that are used in object-oriented middleware in Chapter 5.

**Example 3.1**

RPC Program that Hands' Soccer Player Data

```c
const NL=64;
struct Player {
    struct Dof {int day; int month; int year;}
    string name<NL>;
};
program PLAYERPROG {
    version PLAYERVERSION {
        void PRINT(Player)=0;
        int STORE(Player)=1;
        Player LOAD(int)=2;
    }=0;
} = 105040;
```

The above interface definition defines a struct type for soccer Player data, which may then be passed as a complex procedure parameter to an RPC program.
After having resolved heterogeneity, the presentation layer transforms the high-level data structures that are passed as request parameters into a form that can be transmitted by the transport layer implementation. The required transformation process is commonly referred to as *marshalling*. The presentation layer also implements the reverse mapping, that is the reconstruction of complex data structures from a sequence or block of bytes. This mapping is referred to as *unmarshalling*. Example 3.2 shows an example of marshalling and unmarshalling operations written in C++. In practice, this code is generated by the interface definition language compiler rather than hand-written, but we believe it is important that the reader understands what the generator needs to achieve. We have, therefore, deliberately not used the C++ iostream functionality in order to demonstrate more clearly what needs to be done.

```c
char *marshal() {
    char *msg;
    msg = new char[4*sizeof(int)+1] + strlen(name)+1];
    sprintf(msg,"%d %d %d %d %d ", dob.day, dob.month, dob.year,
             strlen(name), name);
    return(msg);
}

void unmarshal(char *msg) {
    int name_len;
    sscanf(msg,"%d %d %d %d ", &dob.day, &dob.month, &dob.year,
           &name_len);
    name = new char[name_len+1];
    sscanf(msg,"%d %d %d %d ", &dob.day, &dob.month, &dob.year,
           &name_len, name);
}
```

The operation *marshal* transforms player data into a sequence of bytes and the operation *unmarshal* performs the reverse operation. Similar operations are performed by every client and server stub.
The parameters that have to be marshalled obviously depend on the operation signatures of the server objects. Hence, marshalling is specific to the particular remote procedures. The way data heterogeneity is resolved is operation-specific for the same reason. A choice arises as to when marshalling and resolution of data heterogeneity is implemented. It may be implemented statically at compile time (that is, operations for marshalling and resolving data heterogeneity may be compiled and linked into client and server objects) or determined dynamically at run-time (that is, the particular signature of an operation exported by a server object may be interpreted).

Stubs are generated by remote procedure call systems in order to implement type-specific concerns, such as marshalling and operation dispatch. The input to that generation process is an interface definition that defines the formal parameters of the procedures exported by an RPC program. In that way, stub generation solves the problem that presentation layer implementations are needed for each object type. RPC systems on Unix provide an interface definition language compiler, called rpcgen, that generates pairs of stubs, a client stub and a server stub. The client stub resolves heterogeneity of the actual procedure parameters and marshals them. Moreover, it unmarshals the result and reproduces the native data representation for the client. Likewise, the server stub unmarshals the actual procedure parameters and maps them to the native data representation of the host where the RPC program is executed. When the procedure result becomes available it is returned to the server stub, which resolves heterogeneity and marshals it into a transmittable form.

Session Layer

The session layer implementation needs to enable clients to locate an RPC server. This can be done statically or dynamically. The binding in Example 3.3 is static because the hostname is determined at compile time. Static binding is fairly simple, but is not location transparent and thus it limits migration and replication transparency. With dynamic binding, the selection of the server is performed at run-time. This can be done in such a way that migration and replication transparency are retained. Each host of a Unix network that supports RPCs runs a particular daemon called portmap. Any RPC server registers its programs with the portmap daemon that runs on the host. Clients can contact a single

```
Example 3.3
Client Program using RPC to Invoke an Operation

print_person(char *host, Player *pers) {
  CLIENT *clnt;
  clnt = clnt_create(host, 105040, 0, "udp");
  if (clnt == (CLIENT *)NULL) exit(1);
  if (print_0(pers, clnt)==NULL)
    clnt_perror(clnt, "call failed");
  clnt_destroy(clnt);
}
```

The above C code creates a client stub for the RPC program in Example 3.1 that uses a UDP connection to connect to host. If the creation succeeds, it calls the stub with print_0.
portmap daemon, to check whether the server supports a particular program or to get a list of all programs that reside on the server. Clients can also broadcast a search for a program to all portmap daemons of a network and the portmap daemons with which the program is registered will respond. In that way the client can locate the server program without knowing the topology of the network. Programs can then easily be migrated from one server to another and be replicated over multiple hosts with full transparency for clients.

Another design choice is related to activation. The activation policy defines whether a remote procedure program is always available or has to be started on demand. For startup on demand, the RPC server is started by the inetd daemon as soon as a request arrives. The inetd requires an additional configuration table that provides for a mapping between remote procedure program names and the location of programs in the file system. If an RPC program is always available, requests will be handled more efficiently because startup overheads do not arise. On the other hand, it uses resources continuously.

### 3.3 Object-Oriented Middleware

Object-oriented middleware evolved more or less directly from the idea of remote procedure calls. The first of these systems was the OMG’s Common Object Request Broker Architecture (CORBA), and then Microsoft added distribution capabilities to its Component Object Model (COM) and Sun provided a mechanism for Remote Method Invocation (RMI) in Java. The development of object-oriented middleware mirrored similar evolutions in programming languages where object-oriented programming languages, such as C++, evolved from procedural programming languages such as C. The idea here is to make object-oriented principles, such as object identification through references and inheritance, available for the development of distributed systems.

#### Interface Definition Language

Every object-oriented middleware has an object model, which has more or less the features that we discussed in Chapter 2. It has an interface definition language (IDL) that has language constructs for all concepts of the respective object model. While the details of the object models and interface definition languages differ slightly, they have several concerns in common. We discuss the differences when we consider interface definition languages and object models in detail in Chapter 4 but now discuss the commonalities.

A disadvantage of remote procedure calls is that they are not reflexive. This means that procedures exported by one RPC program cannot return another RPC program. Object-oriented middleware provides interface definition languages, where interfaces define object types and instances of these types are objects in their own right. This means that a server object that implements an object type can return other server objects, as shown in Example 3.4.

Object-oriented middleware further extends remote procedure call programs in the way they handle failures. Sun’s RPCs return null pointers when the remote procedure call fails. This does not allow the middleware or the server object to inform the client why a failure
Principles of Object-Oriented Middleware

occurred. Object-oriented middleware has more sophisticated failure-handling mechanisms. It uses exceptions to inform clients about the details of a failure that occurs while an object request is executed. The object models and interface definition languages of object-oriented middleware all support inheritance. This means that common operations and exceptions can be defined in one interface and other interfaces can inherit these operations and exceptions. In particular, all object-oriented middleware has root types from which every interface inherits a set of common operations.

Presentation Layer Implementation

The presentation layer implementation of object-oriented middleware is very similar to that of RPCs. Object-oriented middleware also supports client and server stubs, which perform marshalling and resolve heterogeneity of data representation.

The only serious difference between the presentation layer implementation of object-oriented middleware and RPCs concerns the handling of object references. The transport layer protocol of an object-oriented middleware needs to define representations of object references. They need to be marshalled and unmarshalled in a similar way to application-level data structures.

Session Layer Implementation

The session layer implementation of object-oriented middleware is slightly more complex than that of remote procedure calls. The implementation provides the connection between multiple objects over one or several connections established by the transport layer. In more concrete terms, the tasks performed by the session layer implementation are the mapping of object references to hosts, an implementation of primitives for activation and deactivation.
of objects, the invocation of the requested operation and the synchronization of client and server.

Transport layer implementations, such as TCP and UDP, use addressing schemes based on host names and port numbers. Remote procedure call systems use the portmap daemon in order to map between service names and host names. To achieve this mapping, service names defined by system administrators have to be unique. Object-oriented middleware extends the basic idea of portmaps in several respects. First, addressing of server objects is based on object references. Secondly, these object references are generated by the middleware rather than an administrator. Finally, the session layer implementation of object-oriented middleware maps object references to hosts in a fully transparent way.

Client objects always identify the object from which they request the service by an object reference. That object reference is an essential part of the request and is always transmitted with an identification of the requested operation and the request parameters. The client side of the middleware implementation uses the object reference to locate the host where the server object resides. The session layer then connects to the server side implementation of the middleware that resides on that host. The server side implementation of the object request broker contains a component that we refer to as an object adapter. There is usually at least one object adapter on each host. Among other tasks, object adapters implement object activation and deactivation. (Please review the concepts of activation and deactivation introduced on Page 53.)

The object adapter implements the mapping of object references to active object implementations (that is, implementations that are executing). This mapping is achieved as follows. Upon arrival of a request for an object in its realm, the object adapter checks whether the object is active (that is, whether the object implementation is being executed by an operating system process). If it is not, the object adapter looks up how to start the implementation of the object in an implementation repository or registry. The actual startup may be achieved in many different ways. The adapter might launch a separate operating system process for the object, spawn a new thread of an existing process or dynamically link a library to itself. Once the adapter has started or otherwise identified the operating system process executing the object implementation, it forwards the request to that object.

The object needs to invoke the requested operation. This is referred to as operation dispatch. To dispatch the operation, the object implementation will need to partially decode the request information that has been transmitted via the network and passed on by the object adapter. That request information will include an identifier of the requested operation and so the object knows which operation to call in the implementation of the server object.

The session layer implementation on both sides also needs to achieve the synchronization between client and server objects. Object requests are, by default, synchronous forms of communication. This means that the middleware has to force the client object to wait until the server object has completed executing the request and has produced a result or notifies with an exception that it is unable to satisfy the request. We note that there are also nonsynchronous forms of communication and we have deferred their discussion to Chapter 7. The implementation of synchronization in a middleware system is conceptually part of the session layer, both at the client and the server. The client side needs to wait for requests to be
completed before it gives control back to the client; the server side needs to signal completion of the request to the client side.

We note that some of the above concerns that need to be addressed by the session layer implementation are specific to object types, while others are not. Objects of different types have different operations and each of these operations is likely to have different parameter lists and return types. Hence, the implementation of calling the requested operation on the server side has to be implemented in a different way for each different object type. The mapping of object references to hosts, the activation and deactivation and the synchronization between client and server object can therefore be provided completely by the middleware.

These considerations are summarized in Figure 3.5. The session layer implementation of an object request broker first finds the host on which to execute the request. This decision is generally based on object references which identify a host, either directly or indirectly. It then contacts the object adapter on that host which locates or activates the process in which the object resides. The server process then identifies the object and finally the session layer implementation of the object implementation ensures that the requested operation is executed. This is the point when the session layer passes control to the presentation layer, which handles complex data structures of operation parameters and the results the operation wishes to pass back to the client.

In this section, we discuss how object-oriented middleware facilitates the development of distributed objects. The organization of this section is driven by a development process for...
distributed objects, which is sketched in Figure 3.6. The rectangles represent different process activities and the arrows represent precedence relationships between activities.

The process starts by designing the server objects based on requirements that existing and future client objects may have. The design could be done using a Computer Aided Software Engineering (CASE) tool that supports an object-oriented design notation, such as the Unified Modeling Language. Hence, class diagrams are designed that show the relationship between server objects and client objects and that indicate the static structure of object types in terms of operations and attributes. Sequence diagrams might be used to model scenarios of how client objects interact with server objects. State diagrams might indicate constraints on the order in which server object operations can be requested.

The next step of the process transforms the information acquired in class diagrams into interface definitions. An interface definition governs the structure and behaviour of server objects. The way interfaces are defined depends on the particular object-oriented middleware system that is being used. The transition from class diagrams into interface definitions may be partially automated if the CASE tool supports code generation of interface definitions. The generated interfaces then need to be refined by the server object designer. We discuss the definition of interfaces in detail in Section 3.4.1.

Client and server object programmers then use the interface definitions designed in the previous step in order to generate stubs. For a client object, a client stub serves as a local proxy for a server object. This means that the client object can call the client stub with local method or procedure calls. For a server object, a server stub serves as a local proxy for client objects. The stubs implement the presentation and session layers that we have discussed in the previous section. We review the generation of stubs in Section 3.4.2.
The classes from which client objects are instantiated use client stubs in their implementation. They make object requests by calling a client stub. The client class therefore has to import the client stub. Depending on the programming language chosen for the client, this may be done using a C++ include directive or a Java import statement.

The server object programmer uses the stub in order to implement the object. Depending on the programming language chosen for the server implementation, the classes that implement the server object type might inherit from an abstract class defined in the server stub or it might implement a server stub interface.

In centralized applications, the operating system loader achieves that object implementations are available when they are needed. With distributed objects, an object adapter is responsible for that and it relies on an implementation repository. Server object implementations have to be registered with an implementation repository. Then the distribution middleware knows where to locate the objects and how to activate them.

3.4.1 Interface Definition

In Chapter 2 we argued that the object-oriented paradigm provides a suitable component model for building distributed systems. We have outlined typical constituents of a meta-model for distributed objects. A meta-model defines concepts that determine properties of object types. In fact, each object-oriented middleware defines its own meta-model. Examples include the OMG object model defined for CORBA, Microsoft's Component Object Model (COM) and the object model for Java Remote Method Invocations (RMI).

Object-oriented middleware systems need to provide facilities for designers to instantiate their meta-models. The instantiations are type definitions for distributed objects. Middleware may use a programming language for this purpose or it may provide an explicit interface definition language (IDL). IDLs are typically not computationally complete. They do not provide language constructs that are needed to implement objects.

We note that the interface definition adds considerable detail and thus refines the class diagram, such as that shown in Example 2.3. In particular, the interface definitions include exceptions, definitions of constructed types, such as Address, refined parameters of operations and implementations for the various associations.

Interface definitions play an extremely important role in the engineering of distributed objects. [Meyer, 1988b] considers type definitions as contracts between the clients that wish to use a service and the objects that provide the service. The object implementations in fact are under an obligation to implement the interfaces, provide the declared operations, respect their formal parameter lists and raise only those exceptions that are declared in the interface. Clients are under an obligation to call only those operations that have been declared in an interface, pass actual parameters that match the formal parameters declared in the interface, and to catch all the exceptions that are declared.

The declarations provided as part of an interface definition are also used by the middleware itself. In particular, for dynamic invocations when requests are only defined at run-time, middleware systems need to safe-guard that both clients and server actually obey the
contract. Moreover, there are situations where the middleware needs to translate requests between formats used by different middleware systems and it usually needs to have type information to do that. We will see in Chapter 6 that middleware systems store interface definitions for that purpose and provide a programming interface so that interface definitions can be accessed.

While all object-oriented programming languages have types or interfaces, interface definitions have a more specific role for distributed objects; they are the principle source from which middleware systems generate client and server stubs and thus they provide the basis for the automatic implementation of those presentation and session layer tasks that are typespecific.

### 3.4.2 Stub Generation

We have already discussed the role of stubs in the presentation and session layer implementations of remote procedure calls. Also object-oriented middleware use client and server stubs for the implementation of presentation and session layers. From a development point of view, we can consider them from a different perspective. For a client developer, a stub represents the server object. From a server object, the server stub represents the client object. Stubs are, in fact, incarnations of the proxy design pattern that was identified in [Gamma et al., 1995]. A proxy is placeholder that hides a certain degree of complication. Client and server stubs hide the fact that an operation execution is being requested in a distributed fashion. The client stub resides on the same host as the client object and the server stub resides on the same host as the server object. The implementor of the client object may, therefore, call the client stub using a local method call. Likewise, the server stub calls the server object using a local method call and the server and client stub implementations implement the distributed communication transparently for client and server object developers.

Figure 3.7 shows the differences between a local method call and an object request that is implemented using client and server stubs. With a local method call, parameters are passed from the caller to the called operation via mechanisms, such as the hardware stack. The call is executed using some jump operation that is supported by the processor. Object requests, which are executed remotely, utilize stubs to perform the marshalling and unmarshalling, the resolution of heterogeneity and the synchronization of client and server objects.
Stubs are always generated by a compiler that the middleware provides for its respective interface definition language. Part of every compliant CORBA implementation is a CORBA/IDL compiler that generates stubs. Note that server stubs in CORBA are referred to as server skeletons. Every COM implementation comes with a Microsoft IDL compiler that generates the COM client and server stubs. The approach taken in Java/RMI is slightly different in that it provides a compiler that generates stubs from the Java class definition, without the involvement of an intermediate interface definition.

### Example 3.5

**Stub Generation Process**

The CORBA interface definition contained in file `soccer.idl` is translated into stubs and skeletons that can be used for client and server object development in C++. The IDL compiler generates four files: two for the client stubs and two for the server stubs. With the CORBA/C++ language binding, each stub is itself a class; the first file of each pair includes the class definition and the second one the class implementation.

### 3.4.3 Implementation of Client Objects

Client objects make an object request by invoking a client stub. Hence they use a local method call for defining that request. This implies that the request is full, defined at the time the source code of the client is compiled. We therefore refer to this kind of request definition as static request definition.

The invocation of a stub implies a dependency of the client code from the stub. Hence, the stub has to be generated before the client code can be compiled. Moreover, the client code needs to be recompiled whenever the stub has been changed, for example due to a change in the interface definition of the server object.

We demand object requests to be type safe. Type safety means that at run-time the server object will be able to meet those requests that have been accepted at compile time. Without type safety we could experience run-time errors due to requests of unavailable operations or wrong actual parameters. When compiling a client object source in a type-safe setting, we expect the compiler to detect requests that are not supported by the server object and wrong actual parameter lists. This is generally achieved by using statically-typed languages, such as Ada, C++ or Java. Using these languages, we will import or include the stub class declaration and the compiler will check any static request definition against the class declaration. Because the client stub class declaration directly evolved from the interface definition,
Developing with Object-Oriented Middleware

programmers can be certain that they only make requests that are actually supported by the interface of the server object.

Because client stubs are invoked by the client, the client has to be bound with the object code for the client stub. Because the client stub generally relies on the object-oriented middleware to resolve the generic part of the presentation and session layer implementations, some middleware libraries also have to be bound with the client.

For any operation included in the interface, the operation in the client stub generally has the same signature as the operation in the server object. This supports linking the server object directly with the client when it is known at linking time that remote access will not be needed. Hence, the identity of operation signatures means that static object requests achieves access transparency. From an engineering point of view, it is a good idea to define interfaces between subsystems using an interface definition language, even though the interfaces might not be used in a distributed setting. The availability of interfaces, however, simplifies distribution of these components considerably.

The identity of the signatures of client stubs and server objects also enables the middleware to make optimizations based on the location of objects. If client and server objects reside on the same host, there is no need to use client and server stubs and the network for the communication of object requests. Indeed, that would be far too inefficient. Sophisticated object-oriented middleware implementations therefore shortcut the network and invoke the server object directly if it is located on the same host. Again, this is considerably simplified by access transparency.

The implementation of interfaces has to solve the problem that a generated server stub needs to call the object implementation in a type-safe way. We would like the compiler that we use for translating the server object to detect typing problems, such as missing parameters or parameters of the wrong type for the invocation of server objects in the same way as for the invocation of client stubs.

There are two approaches to the implementation of server objects in a type-safe way: interfaces and inheritance. Both approaches achieve that the server stub calls the server object.

Not all object-oriented programming languages support the concept of interfaces and inheritance may be used with these languages. For implementation by inheritance, the middleware generates an abstract class as part of the server stub. This abstract class has abstract operations that declare the operation signature but are not implemented by the class. Pure virtual member functions may be used in C++ and deferred operations may be used in Eiffel. The class implementing the server object is then defined as a subtype of this abstract class. The server stub will have a reference that is statically of the abstract class but refers dynamically to an instance of the server object.

The programming language compiler used for the server implementation will check that the redefinition of abstract operation meets the declarations of the respective operations in the
abstract superclass. The implementation of Player Dispatch, for example, would have a reference of static type Player Impl, which at run-time refers to an object of type Player-Server. Hence, this inheritance approach is another way to achieve type-safe implementations of server objects.

Several modern programming languages, such as Java, provide a mechanism for interfaces and a particular implementation relationship between interfaces and classes. These programming language interfaces should not be confused with the interfaces for distributed objects that are written using an IDL in a way that is independent of programming languages. Programming language interfaces are a particularly useful vehicle for the type-safe implementation of server objects. Following this approach, the IDL compiler generates a programming language interface that includes all operations exported by the IDL interface. The server object implementation then declares that it implements the programming language interface.

This approach achieves the required type safety, because the compiler used for translating the implementation of the server object checks that it correctly implements the interface that was derived by the IDL compiler. In Example 3.6, a Java compiler used for compiling Player-Server validates that the interface Player_Impl is implemented correctly. As Player Dispatch uses Player Impl, we have achieved that the generated class can safely call the hand-coded class.

3.4.5 Server Registration

object activation as discussed above might necessitate the object adapter to start a server object implementation. Once the server objects have been compiled, they need to be registered with the object-oriented middleware. The purpose of this registration is to tell the object adapter where to find the executable code for the object and how to start it.
How server implementations are registered largely depends on the particular middleware and on the operating system on which it runs. The details are not important for this textbook and we focus on the principles only.

The registration is generally done by a system administrator, who maintains an implementation repository. The implementation repository is somewhat similar to the inntd daemon of remote procedure calls. It associates objects, which are usually identified by an object reference, with some instructions on how to start up the servant in which the server object executes. There is an implementation repository on each host. The middleware provides administration tools to maintain the implementation repository. These tools enable administrators to browse through the list of servants that are registered on a particular host. Moreover, they allow administrators to register new server objects and to delete existing server objects.

There are several examples of these implementation repositories. Microsoft's COM relies on the registry that is part of the Windows operating system. Likewise, Java provides activation interfaces that are used with RMI. They are implemented in the Java RMI daemon, which plays the same role as the object adapter in CORBA. Although the CORBA specifications do not explicitly prescribe it, many CORBA implementations do include an implementation repository with respective administration tools.
We used the ISO/OSI reference model to explain where conceptually object-oriented middleware is situated and we identified that the middleware is built on top of the transport layer. We discussed non-object-oriented forms of middleware. These include transactional and message-oriented middleware as well as remote procedure calls.

- Object-oriented middleware implements the session and presentation layers of the ISO/OSI reference model.
- The session layer implementation uses connections between two hosts to establish a connection between two objects residing on that host. In order to achieve that, it implements an object adapter which knows about all active server objects on a particular host and is capable of activating and deactivating objects. Finally the session layer implements the synchronization of requests between two objects.
- The presentation layer implementation of an object-oriented middleware implements the transport of complex request parameters and results and the resolution of heterogeneous data representation.
- We discussed the main activities of the development process for distributed objects. These include the interface definition, the generation of client and server stubs, the coding of client and server objects and the registration of the server object.
- The interface is defined using an interface definition language that is an essential part of most object-oriented middleware. The middleware provides a compiler that checks interfaces on syntactic and static semantic consistency and produces client and server stubs.
- Client and server stubs contribute to achieving type safety of request. On the client side, this is concerned with ensuring that the client only requests operations that are declared in the interface and that the client passes correct actual parameters. On the server side, type safety is concerned with ensuring that the server object implements the interface definition correctly.
- We identified the need for an implementation repository that stores the information on how an object adapter can activate server objects prior to these server objects serving requests.

3.1 Name the seven layers of the ISO/OSI reference model.
3.2 On which layer does object-oriented middleware build?
3.3 Which layers does the middleware implement?
3.4 Give reasons why it is not a good idea to use the transport layer directly.
3.5 What concerns does the session layer implementation of object-oriented middleware address?
3.6 What are the two principal tasks that the presentation layer implements?
3.7 What concerns are specified in the interface definition that are typically not determined in an object-oriented design using UML?

Further Reading

3.8 What are the principal products that an interface definition language compiler produces?
3.9 What is type safety?
3.10 How do client and server stubs contribute to achieving type safety?
3.11 Name three non-object-oriented classes of middleware.
3.12 When would you use which class of middleware?

There are many books about networks and it is impossible to list them all here. [Tanenbaum, 1989] provides an excellent introduction to the ISO/OSI reference model and we highly recommend it. Though we suggested in this chapter that it is not a good idea for an application programmer to use network programming capabilities directly, it may be worth considering [Stevens, 1990] for a detailed description of TCP, UDP and the socket programming interface. The reader may then be
the use of object-oriented middleware.

We have only been able to give a very brief overview of other classes of middleware and refer the reader to the extensive literature. One of the earliest transaction monitors was IBM/CICS and we refer to [Summerville, 1987] for a discussion of its use. [Hall, 1996] provides a good introduction to the use of Tuxedo, one of the market leading transaction server products, for the construction of client-server architectures. Recently, Microsoft launched a transaction monitor product and a discussion of its application is provided by [Hillier, 1998].

There is a large body of literature about remote procedure calls. A good introduction is included in [Colouris et al., 1994]. Remote procedure calls were standardized by the Open Group as part of the Distributed Communication Environment (DCE). A good introduction to DCE is provided by [Schill, 1993].