2 Distributed Systems Development

2.1 Introduction

Computer systems used to be expensive standalone self-contained entities, each with its own disk storage, line printers, terminals and other peripherals. The introduction of the minicomputer made computers cheaper and more widespread, which led to the requirement to share information between them. This requirement led to the development of early computer networks, such as the Unix-to-Unix copy program (UUCP) in 1976 and its subsequent release in AT&T Version 7 Unix in the same year.

The development of Berknet by Eric Schmidt in 1978 at the University of California, Berkeley and its subsequent distribution in Version 7 Unix for the PDP-11 minicomputer allowed users to send and receive email, transfer files and print remotely [88]. In 1980, Bolt, Beranek, and Newman were contracted by the American Department of Defense to implement the TCP/IP protocol for BSD UNIX. The release of 4.2BSD in August 1983 with its implementation of TCP/IP and the BSD sockets programming model, coupled with the growth of local area networks based mainly on Ethernet, allowed computers to connect to the ARPANET, the predecessor to the Internet, which led to the enormous growth of networked systems in the early 1980s [96].

The introduction of the personal computer and its subsequent ability to connect to TCP/IP networks using the Winsock API, based on BSD sockets, led to a huge increase in the number of networked machines and distributed systems began to become mainstream. Automated teller machines, airline reservation systems, file sharing, file transfer, centralised database access, email and various other distributed systems were introduced.

The subsequent invention of the web browser and HTTP protocol led to the World Wide Web and the enormous explosion in the number of distributed systems that we see today. With the continued increase in processing power and fall in component prices, computing is promising to become even more widespread and we may well see the vision of ubiquitous computing [120] being met in the future. Yet, while distributed systems have become mainstream, distributed systems development remains difficult and little advancement has been made since the initial concepts were developed decades ago.
This chapter develops the lineage towards the domain-specific aspect language (DSAL) approach to distributed systems development by examining previous approaches, and issues surrounding those approaches.

We begin by discussing the low-level Application Programming Interface (API) approach as exemplified by BSD sockets. We then discuss the RPC approach for both procedural languages, in the form of ONC RPC, and object-oriented systems in the form of CORBA and Java RMI and compare the network awareness and network transparent models.

We then discuss the high-level API approach as exemplified by JMS and discuss the effects and implications the different approaches have on ease of development, software reusability and maintainability.

Throughout this chapter we use the example of a simple distributed service that returns the current date.

2.2 Sockets-Based Programming

Sockets are a low-level generalised programming interface for networking and interprocess communication first provided in the BSD4.2 operating system\(^1\). Most, if not all, UNIX systems provide the socket API and various operating systems, such as Windows with its Winsock API, provide similar functionality.

Sockets are a low-level networking API modelled on the UNIX systems calls related to file I/O semantics. While there is some similarity between file and network I/O operations, network I/O has other considerations, which make the fit less than perfect. Stevens [96] identifies the following considerations for network I/O:

- The client server relationship is not symmetrical. The application needs to know which role (client or server) it is to assume.
- Network connections can be connectionless or connection-oriented. Connectionless operations do not map neatly to file operations because there is no concept of opening a connection as every network I/O operation could be to a different host.
- Names are more important in distributed systems, for example to verify security, than they are for file operations. Therefore, passing a file descriptor to a process without knowing the original name, while being acceptable for a file I/O operation, may not be acceptable for a network I/O operation.
- Additional parameters, for example the protocol and its details, are required for network operations.
- While the UNIX I/O system is stream-oriented, many network protocols are message-oriented and therefore rely on message boundaries.

\(^1\)Strictly speaking, BSD sockets were initially provided in the 4.1cBSD release and subsequently refined into their current form in 4.2BSD.
Network interfaces support multiple protocols, each with differing addressing requirements so that, for example, a 32-bit identifier is not sufficient for holding network addresses for all protocols. Network interfaces therefore need to be generalised.

Sockets provide a low-level interface to network protocols. For anything other than simple message exchanges, protocols are developed to exchange messages in specific formats. Due to the stream-oriented nature of the UNIX I/O system, this is a particularly onerous task as the programmer is required to implement packet assembly and disassembly, which differs depending on the protocol being implemented. In addition, error handling and recovery is left entirely up to the programmer, making socket development difficult and error prone.

This section provides an overview and evaluation of the BSD socket interface and illustrates its usage through a simple application.

### 2.2.1 BSD Socket Interface

As previously mentioned, there are a number of similarities between file and network I/O operations. The BSD socket interface attempts to provide as much similarity as possible while allowing additional network-based operations. Table 2.1 provides an overview of the differences and similarities between the socket and file operations.

Figure 2.1 provides an overview of the steps required for both client and server to initiate a transfer for a connection-oriented transfer. This interaction can be summarised as:

**Server interaction.** Firstly, the server obtains a socket via the `socket()` call and then `bind()` is called, which assigns a name to the socket. Next `listen()` is called to indicate that the application is willing to accept connection requests. As well as the socket, the `listen()` call also accepts a backlog parameter, which stipulates the number of connection requests that can be queued by the system while it waits for the `accept()` call to be executed. If a connection attempt arrives and the queue is full, the connection is refused. Finally `accept()` is called, which suspends the application until a connection request arrives from the client.

**Client interaction.** A socket is obtained via the `socket()` call and then `connect()` is called with a parameter stipulating the socket and the address of the server. The address is passed in the `sockaddr` structure and contains the local address, the remote address, and the protocol to use. Once a connection has been established, messages can be exchanged between the two systems. For both of the above, the `close()` call can be used to close the connection and the `shutdown()` call can be used to close part of the connection, either reads or writes.

The socket API also provides the `select()` function, which is used to provide I/O multiplexing by allowing the programmer to examine a set of file descriptors.
Table 2.1. Comparison of file and socket calls.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Files</th>
<th>Sockets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>open()</td>
<td>Client</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Server</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>close()</td>
<td>close()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shutdown()</td>
</tr>
<tr>
<td>Read</td>
<td>read()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>recv()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>recvfrom()</td>
</tr>
<tr>
<td>Write</td>
<td>write()</td>
<td>write()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sendto()</td>
</tr>
<tr>
<td>Seek</td>
<td>lseek()</td>
<td></td>
</tr>
<tr>
<td>Poll</td>
<td>select()</td>
<td>select()</td>
</tr>
<tr>
<td>Create</td>
<td>creat()</td>
<td></td>
</tr>
</tbody>
</table>

to see if they are ready for I/O or if certain events have occurred. UNIX systems also provide asynchronous I/O which is used in conjunction with `select()` thereby allowing a single process to efficiently handle a large number of open files or sockets simultaneously.

2.2.2 Socket Example

Our simple example provides the current system date to a client on demand and prints it to the console. The protocol we use between client and server is based on the exchange of C style strings between client and server. While this may sound simple, it is not, due to fundamental differences between file I/O and network I/O even though both share the same interface.

Using sockets, a read or write call may input or output fewer bytes than requested due to underlying kernel buffer limits. However, a read operation from a file is guaranteed to return the number of bytes requested providing that the number requested is less than or equal to that remaining before end-of-file. In the event a read or write call returns fewer bytes than expected, the call must be invoked again to receive or send the remaining bytes [96]. This can significantly complicate socket development, particularly if the protocol developed contains many different packet types of different sizes and advanced features such as sliding windows and
2.2 Sockets-Based Programming

Figure 2.1. Socket calls for a connection-oriented protocol.

piggybacking [107] are used. To overcome this, we provide a set of utility routines to read the remaining bytes, in the case of a read call, or output the remaining bytes, in the case of a write call.

\[
\begin{align*}
\text{int} & \text{ readline(} \text{int} \text{ socket, char *buffer, int len);} \\
& \text{int} \text{ writeln(} \text{int} \text{ socket, char * buffer, int bytes);} \\
\end{align*}
\]

Our server implementation is as follows:

extern int writeln(int socket, char * buffer, int bytes);

\[
\begin{align*}
\text{int} & \text{ main(} \text{int} \text{ argc, char ** argv) } \\
& \text{ { } } \\
& \text{int sock;} \\
& \text{struct sockaddr_in server;} \\
\end{align*}
\]
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    fprintf(stderr, "cannot open stream socket\n");
    exit(1);
}

memset(&server, 0, sizeof(server));
server.sin_family = AF_INET;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(5000);

if (bind(sock, (struct sockaddr *) &server, sizeof(server)) < 0) {
    fprintf(stderr, "bind failed\n");
    exit(1);
}

listen(sock, 10);

for(;;) {
    struct sockaddr_in client;
    int fd;
    unsigned int client_len = sizeof(client);

    fd = accept(sock, (struct sockaddr *) &client, &client_len);
    if (fd < 0) {
        fprintf(stderr, "accept failed\n");
        exit(1);
    }

    time_t now = time(NULL);
    char *s = ctime(&now);
    writeline(fd, s, strlen(s));
    close(fd);
}

The client implementation is as follows:

extern int readline(int socket, char * buffer, int bytes);

int main(int argc, char ** argv) {
    int sock; struct sockaddr_in server;

    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_port = htons(5000);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "cannot open stream socket\n");
        exit(1);
    }
}
if (connect(sock, (struct sockaddr *) &server, sizeof(server)) < 0) {
    fprintf(stderr, "connect failed\n");
    exit(1);
}

char buf[512];

readline(sock, buf, sizeof(buf));
printf("The time on the server is: %s", buf);

close(sock);
exit(0);

In both the client and the server examples above, the code that is used to either obtain the date or receive it is shaded in grey, while the other code implements the distribution concern.

2.2.3 Summary

Socket-based programming is highly complex and error prone. Developers need to implement message-based protocols on top of the socket interface with the resultant complexity directly proportional to the protocol requirements.

As illustrated in our simple example, socket-based programming is highly intrusive, even for simple applications. Programmers need to adhere to the socket API throughout the application and ensure that reads from sockets and writes to sockets return the number of bytes requested or the number of bytes required to be written respectively.

The most common protocols used in socket programming are TCP/IP and UDP/IP. UDP is a datagram protocol, which does not provide guaranteed message delivery or duplicate elimination. Although TCP/IP provides these features it does not guarantee message delivery in all circumstances [23, 115]. If guaranteed message delivery is a protocol requirement, it will need to be provided by the protocol developer. Protocol reliability is discussed in detail in Section 2.3.4.

Recovery is left up to the programmer to implement at a very low level and any significant recovery routines, for example connecting to another server in the event the current server becomes unavailable, will have to be implemented wherever a remote call is made, thereby further complicating development.

2.3 Remote Procedure Calls

In order to provide an environment with the simplicity of the then dominant procedural programming paradigm, Birrell and Nelson [15] suggested the use of remote procedure calls (RPC). The idea of RPCs is based on the observation that procedure calls are a well-known and well-understood mechanism for transfer of control and data within a program running on a single system and that this mechanism
can be extended to be used across a communications network [15]. By extending procedure calls to a distributed environment, interprocess communication is then given the syntax and semantics of a well-accepted strongly typed language abstraction [90].

According to Soares [90], the RPC mechanism has the following advantages:

- The communication mechanism has clean, general, and comprehensible semantics.
- A programmer is able to design a distributed application using the same abstraction as well-engineered software in a non-distributed application.
- It provides information hiding as information can be hidden within design components.
- The distribution of the application is transparent to the application user and all communication details are hidden.

When a remote procedure call is invoked, the calling environment is suspended, the procedure parameters are passed across the network to the callee, and the procedure is executed on the remote machine. When the procedure finishes, the results are passed back to the calling environment, where execution resumes as if returning from a local procedure call [15].

RPCs can be either asynchronous or synchronous. Asynchronous RPC calls do not block the caller and the replies can be received as and when they are needed, thus allowing the caller execution to proceed in parallel with the callee execution [4]. With synchronous RPCs, on the other hand, the caller is blocked until the callee has finished execution.

This section provides an overview of the remote procedure call paradigm and an implementation of a simple application using the ONC RPC system. We describe the features and facilities of RPC systems along with their shortcomings and compare it to BSD socket-based distributed systems.

### 2.3.1 Stubs and Skeletons

Remote procedure calls achieve a high-level of abstraction by using a system based on proxies [41]. Proxies are used on the caller side to convert local procedure calls into remote procedure calls and are used on the callee side to convert remote procedure calls to local procedure calls. The caller proxy is known as a stub and the callee proxy is known as a skeleton [116]. The interaction\(^2\) between the components in an RPC system is depicted in the diagram in Figure 2.2.

When a remote procedure call is invoked, the stub (compiled into the caller code) translates the arguments into a data representation, a process called marshalling, and transmits the data to the callee via the RPCRuntime system. On receipt of the packets, the RPCRuntime in the callee machine passes them to the skeleton (compiled into the callee code). The skeleton unpacks them into the

\(^2\)This example adapted from Birrell and Nelson [15].
appropriate data types for the machine, a process known as unmarshalling, and makes a normal local procedure call to the server process. The return value from the local procedure call is then marshalled by the skeleton and returned to the calling code [15, 116].

2.3.2 Interface Definition Language

Remote procedure calls generally make no assumptions about the architecture of the remote system or the programming language remote procedures have been written in. Key to supporting communication between these systems of unknown architectures written in unknown programming languages is the notion of an Interface Definition Language (IDL), a machine-neutral language used to describe the remote procedures, their parameters, and the call semantics (described in Section 2.3.4) in a machine neutral way. The IDL is read by an application, which generates the stubs and skeletons of the application.

The Network Interface Definition Language (NIDL) defined as part of the Network Computing Architecture (NCA) [26] provides the following data types:

- **Integers**: Both signed and unsigned integers in one, two, four, and eight byte sizes.
- **Floating point**: Single (four byte) and double (eight byte) precision floating point.
- **Scalar types**: Other scalars including signed and unsigned characters, booleans, and enumerations.
- **Type constructors**: Structures, discriminated unions, pointers, and arrays. Pointers to pointers or records containing pointers are not permitted.

Various attributes can be associated with remote procedures so that the RPC compiler can generate stubs and skeletons that are either more efficient or provide a particular feature. For example, the Distributed Computing Environment (DCE) [109] provides the following attributes:
The **idempotent attribute**: Indicates that the operation may safely be called more than once as it does not modify any state and/or yields the same result on each invocation.

The **broadcast attribute**: Specifies that the operation may be sent to multiple servers, effectively concurrently. An operation with the broadcast attribute is implicitly an idempotent operation.

The **maybe attribute**: Specifies that the operation’s caller must not require and must not receive a response or fault indication. An operation with the maybe attribute must not contain any output parameters and is implicitly an idempotent operation.

The **reflect_deletions attribute**: Specifies that memory occupied by targets of pointers on the client will be released when the corresponding targets of pointers on the server are released. This is true only for targets that are components of in-parameters of the operation.

Having a machine-neutral IDL allows multiple languages to use the RPC system as the IDL compilers can generate the stubs and skeletons for each implementation language. This also has the advantage of allowing a remote procedure developed in one language to communicate with a remote procedure developed in another language.

While this greatly simplifies the development of distributed applications, the language-neutral nature limits the kinds of data that can be exchanged between processes to the basic data types that can be represented in all the target languages [116].

While IDLs remove the complexity of network data representation from the programmer, the programmer must still control the lifecycle management of the data sent that typically require either complex conventions or reference counting. These procedures are prone to programmer error that can lead to memory leaks or referential integrity loss [116].

### 2.3.3 Data Representation

Due to the heterogeneous nature of computer networks, data transmitted between machines require a data representation protocol, which defines the way data is represented so that machines that store data in different internal formats are able to communicate [26].

There are a number of data representation standards including Sun’s XDR standard [97] and DCE’s NDR [109].

### 2.3.4 Calls Semantics

Due to their distributed nature, remote procedure calls can fail. According to Soares [90], there are three causes of RPC failure:

**Network failure**: The network is unavailable and the caller and callee cannot send or receive data.
2.3 Remote Procedure Calls

**Caller site failure:** The caller process fails or the host running the caller process fails.

**Callee site failure:** The callee process fails or the host running the callee process fails. In this case, the caller may be indefinitely suspended awaiting a response from the callee.

Most RPC systems, such as DCE RPC [109], attempt to hide their distributed nature from the programmer so that, to the programmer, the RPC system is transparent. The RPC system, and not the application code, is therefore responsible for ensuring a message reaches its intended destination and a response is received.

If, however, no response is received within a specific timeout period, one of four different conditions may have occurred [90]:

1. The callee did not receive the request.
2. The callee received the request and acted upon it but the caller did not receive the response.
3. The callee failed during the execution of the call and either resumed execution of the call upon restarting or did not.
4. The callee was still busy executing the call when the caller timed out.

According to Soares [90], a major design decision for an RPC mechanism is the choice of call semantics in the presence of failures. Spector [95] defines four different call semantics:

**Maybe:** The callee does not return a response to the caller and the caller receives no indication of success or failure.

**At least once:** The remote procedure is executed at least once.

**Only once type 1:** This is commonly referred to as at most once [108]. The remote procedure is executed at most once.

**Only once type 2:** This is commonly referred to as exactly once [108]. The call has been executed once only.

Although exactly once call semantics are generally considered to only be possible using asynchronous procedure calls [95], a number of attempts have been made to provide exactly once semantics for synchronous procedure calls.

Heindel and Kasten [48] have implemented reliable synchronous RPC calls for DCE by imposing a middleware layer between the caller and callee. This middleware layer, however, uses asynchronous messaging to achieve this reliability and therefore can be considered asynchronous in implementation.

The Encina transaction monitor attempts to implement reliable synchronous messaging using an extension of DCE’s RPC, called TRPC – transactional RPC [89]. This approach is, in many ways, similar to the asynchronous approach as transactions are written to a log file before being committed. Other approaches, such as replicated procedure calls, have been implemented [22]. Unfortunately
this results in a high overhead per operation \((O)\) as the number of messages \((M)\) required to complete a request \((N)\) is \(O(M \times N)\) \[90\].

Synchronous RPC systems can generally be considered suitable\(^3\) only for applications that can be modelled as idempotent, that is they can safely execute the same procedure more than once without any adverse effects. Probably the best known example is Sun’s Network File System (NFS) \[99\].

### 2.3.5 Binding

Binding refers to the process used to identity and address remote procedures. Bindings are either performed statically, during compilation, or dynamically, at runtime. Callees advertise their bindings, or location, in a naming service so that callers are able to find them, based on appropriate search criteria \[109\].

In order to find the callee the caller needs to provide, depending on the implementation, either the specific server process of the callee, the name of the machine where the callee is located, or just the name of the procedure to be called \[90\].

Attributes may be associated with bindings, for example a version number, so that the caller may choose which specific instance to bind to. Once bound, the remote procedure may be called.

### 2.3.6 Open Network Computing (ONC) Example

The following is an overview of the development of our simple distributed application using Sun’s Open Network Computing (ONC) system \[98\], a widely deployed RPC implementation which was originally developed for Sun’s NFS.

#### 2.3.6.1 ONC IDL

The IDL used by ONC describes remote procedures, their arguments and return values, associated version numbers and a unique program number identifier. To implement our simple example, we firstly declare the remote procedure in ONC IDL as follows:

```idl
program GETDATE_PROG {
    version DETDATE_VERS {
        string GETDATE(void) = 1;
        } = 1;
    } = 22855;
}
```

The above IDL fragment defines version .1 of a remote procedure called GETDATE, which has the parameter type `void` and returns the current date as a string.

\(^3\)Although reliable synchronous RPC systems do exist, they either require the programmer to handle timeouts, retransmissions, and the receipt and sending of messages (removing much of the advantage of using an RPC system) or rely on other mechanisms, such as replication, which significantly increase complexity and have adverse effects on performance.
The unique program number for our implementation is 22855. The `rpcgen` program is then run against the IDL file, which generates a header file `getdate.h`, and two skeleton files, `getdate_clnt.c` and `getdate_svc.c` for the client and server implementations respectively as illustrated in Figure 2.3.

The generated files contain the marshalling and unmarshalling, binding, data representation (XDR), and framework integration code required to implement the ONC protocol.

### 2.3.6.2 ONC Example Implementation

The files generated by the `rpcgen` application are required to be linked with user-supplied files for the client and server implementation. For our example, the relevant portion of our client implementation is as follows:

```c
if ((cl = clnt_create(server, GETDATE_PROG,
                      GETDATE_VERS, "udp")) == NULL) {
    clnt_pcreateerror(server);
    exit(1);
}
```
Figure 2.4. Sun ONC runtime binding.

```c
if ((message = getdate_1(NULL, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(1);
}
printf("Server Date: %s\n", *message);
clnt_destroy(cl);

and our server implementation:

char ** getdate_1_svc(void *v, struct svc_req *svc) {
    time_t now = time(NULL);
    static char *date;
    date = ctime(&now);
    return (&date);
}
```

Note that both the client and server implementation code is required to adhere to the ONC specific framework and programming conventions. For example, in the server implementation, the value that is returned to the client is required to be static and the function signature is required to use the framework specific structure svc_req and to have the program’s version number appended. For the client implementation, the client is required to call the ONC framework directly, as illustrated by the clnt_create function call in our example. However, for the server code no protocol specific code is required.

ONC servers advertise their presence in the portmap service so that clients may find them. At runtime clients bind to the portmap server, which provides the address of the server to clients as illustrated in Figure 2.4. Clients then connect directly to the server.
2.3.7 Summary

This section has provided an overview of the remote procedure call paradigm and an implementation of our simple example using the ONC RPC system. Various aspects of RPC systems have been discussed including the IDL, data representation and call semantics.

As illustrated in our example, RPC systems are highly intrusive as programmers are required to describe their remote procedures in an IDL, to implement methods generated by the IDL program generator, to use the RPC framework directly in the implementation code, and to adhere to framework specific conventions. However, compared to BSD socket-based systems, RPC systems are significantly easier to implement, understand, and subsequently maintain.

Most RPC systems attempt to provide programmer transparency so that calling a remote procedure is as simple as calling a local procedure. Indeed, this transparency is often seen as a great benefit of the RPC paradigm as Soares [90] states:

"The ideal RPC mechanism is the one that provides the application user with the same syntax and semantics for all procedure calls independently of being a local call or a remote one"

This notion of transparency, however, leads to an unfortunate situation when errors occur, as recovery is left to the RPC system to handle, not the programmer. Unfortunately, in many situations, the RPC system simply cannot recover, resulting in the application hanging while the RPC runtime attempts to reconnect to the server. Synchronous RPC systems are therefore suitable only for applications that can be modelled as idempotent, such as the NFS system. Perhaps the biggest weakness with RPC systems, however, is that their language-neutral nature limits the data types that can travel between processes to the basic static data types that can be supported by each language [116].

2.4 CORBA

One of the primary issues with early RPC systems is that they did not have an object-oriented model and client applications need to know not only how to access a server but also the location of the server. In addition, client code has to change whenever the client wants to use new services [27].

The Common Object Request Broker Architecture (CORBA) is designed as a middleware to enable distributed objects to communicate with one another via an Object Request Broker (ORB). In the CORBA model, clients communicate to a server via an ORB as illustrated in Figure 2.5.

Communicating via an ORB removes the necessity for a client to know the whereabouts of a server as clients send requests to the ORB requesting that certain
services be performed. The ORB then passes those requests on to a server, which acts upon it, and passes the result back to the client via the ORB.

The ORB is responsible for the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface [77].

Clients, therefore, only know the location of the ORB and the ORB knows the implementation details and locations of the servers. Clients and servers communicate only via component interfaces and any changes in object implementation or location are insulated from the client [27, 75, 77].

CORBA is a heterogeneous system that can be run on many different platforms and CORBA applications may be written in many different languages. For this reason CORBA uses an IDL, similar to RPC type systems. The main construct in the CORBA IDL is the interface which defines the various operations that may be called by clients. Once written, the IDL is run through a compiler to generate code for the particular implementation language [7]. By using a language and machine independent IDL, clients and servers may be written in different languages and may be run on different operating systems so that it is possible for, say, a client writing in the C language running under the UNIX operating system to communicate to a server writing in Java running under the Windows operating system.

One of the more interesting aspects of CORBA is that it uses an object-oriented architecture in that it adds a notion of inheritance. In CORBA IDL an interface may inherit from another interface as shown in Figure 2.6.

In addition, CORBA IDL supports multiple inheritance where an interface may inherit from several different interfaces. There are, however, a number of limitations to the multiple inheritance feature of the CORBA IDL [7]:

- An IDL interface cannot redefine an operation or attribute in a derived interface.
- It is illegal for an interface to inherit from two interfaces that have a common operation or attribute name.
interface Vehicle {
);

interface Car : Vehicle {
);

Figure 2.6. An example of the CORBA IDL’s support for inheritance.

Servers that implement derived interfaces, however, are considered an implementation of the basic interface, which gives a notion of polymorphism as the server is treated as though it were of both the base type and the derived type within the CORBA architecture [122].

2.4.1 CORBA Event Service

Standard CORBA requests are synchronous in nature. A request is sent from a client to a server via the ORB and the client suspends awaiting a response from the server. In this scenario, both client and server must be available.

The CORBA event service [76] decouples communication between clients and servers using either the push model or the pull model.

The push model allows the supplier of events to initiate the transfer of event data to consumers, while the pull model reverses this by allowing the event consumer to request event data from the producer.

The CORBA event architecture uses an event channel, an intervening object that allows producers and consumers to communicate asynchronously. This architecture is illustrated in Figure 2.7.

2.4.2 CORBA Example IDL

In order to implement our simple date application, we are firstly required to define the CORBA interface in an IDL file
. Figure 2.8 illustrates the IDL implementation for our simple example.

Our IDL defines a simple interface with two operations, getDate(), our method to return the server’s date and shutdown(), a method used to shut down the ORB.

Once we have defined the IDL we run the IDL through the idlj compiler to generate CORBA helper files that are necessary to implement the client and server code. Once run, the idlj compiler generates the following files:

---

4This example is derived from [101].
Figure 2.7. The CORBA event architecture.

module DateApp {
    interface Date {
        string getDate();
        oneway void shutdown();
    }
};

Figure 2.8. The IDL for our simple CORBA implementation

- **DatePOA.java.** An abstract class that provides basic CORBA functionality for the server. It extends `org.omg.PortableServer.Servant`, and implements the `InvokeHandler` interface and the `DateOperations` interface. The server class, `DateServant`, extends `DatePOA`.
- **_DateStub.java.** The client stub which is used to provide CORBA functionality for the client. It extends `org.omg.CORBA.portable.ObjectImpl` and implements the `Date` interface.
- **Date.java.** This is an interface that contains the Java version of the IDL interface and extends `org.omg.CORBA.Object`, providing standard CORBA object functionality, and the `DateOperations` interface and `org.omg.CORBA.portable.IDLEntity` class.
- **DateHelper.java.** This class provides auxiliary functionality, such as the `narrow()` method used to cast CORBA object references to their proper types.
• DateHolder.java. This class holds a public instance member of type Date and is used for all operations that have an inout IDL declaration.
• Operations.java. This interface contains the methods getDate() and shutdown() and is shared by both the stubs and skeletons.

2.4.3 CORBA Example Implementation

Our server implementation is illustrated in Figure 2.9 and our client implementation in Figure 2.10. As can be seen by these examples, Java’s CORBA framework is highly intrusive in nature, requiring the developer to implement the framework’s code alongside the applications leading to a tight coupling between the application and the distribution framework.

To run the CORBA example, the ORB needs to be started and the server bound to the ORB. Once bound, the client application connects to the ORB and requests the service it is interested in by name. The ORB contacts the server, which runs the request and sends the result back to the ORB. The ORB then returns the result back to the requesting client.

2.4.4 Summary

CORBA has similar issues to RPC type systems in that the range of values that can be passed between systems, either as arguments or return values, is limited to those that can be represented in all of the implementation languages supported. In addition, as with all IDLs, programmers have an additional language to learn and an additional artefact to deal with. A limitation with the object-oriented nature of CORBA is that objects are passed by reference not by value. There is also no way to extend the range of permissible values transmitted on the fly and still ensure that the value will be correctly interpreted upon receipt [122].

As can be seen in our simple CORBA example, Java’s CORBA framework imposes a large overhead as well as a great deal of complexity. As is common with all frameworks, the CORBA framework is highly intrusive in nature, as illustrated by the shaded areas in Figures 2.9 and 2.10, and results in CORBA specific code tangled with the application’s code making reuse of the application code extremely difficult.

2.5 Java Remote Method Invocation (RMI)

Object-oriented systems are currently the dominant programming paradigm and a number of distributed systems exist for object-oriented languages and systems. Many of these systems use the RPC mechanism even though procedure calls, as such, no longer exist in the object-oriented paradigm. Consequently many of these systems do not provide object-oriented features, such as polymorphism, because the RPC type paradigm only allows for the static representation of data [116].
import DateApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

class HelloImpl extends DatePOA {
    private ORB orb;
    public void setORB(ORB orb_val) {
        orb = orb_val;
    }

    public String getDate() {
        return new java.util.Date().toString();
    }

    public void shutdown() {
        orb.shutdown(false);
    }
}

public class DateServer {
    public static void main(String args[]) {
        try {
            ORB orb = ORB.init(args, null);
            POA rootpoa =
                POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();
            HelloImpl helloImpl = new HelloImpl();
            helloImpl.setORB(orb);
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
            Date href = DateHelper.narrow(ref);
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            String name = "Date";
            NameComponent path[] = ncRef.to_name(name);
            ncRef.rebind(path, href);
            System.out.println("HelloServer ready and waiting \ldots{}")
            orb.run();
        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
        System.out.println("DateServer Exiting \ldots{}")
    }
}

Figure 2.9. An example of a simple CORBA server. Code in the shaded area implements Java’s CORBA framework.

The Java programming language, however, provides the Remote Method Invocation (RMI) distributed system, based on Modula-3 network objects [14], that allows for the dynamic representation of data and therefore allows for polymorphic data to be transmitted and received across the network [116]. More recently, the Jini distributed system [102] builds on the idea of polymorphic data representation
import DateApp.*;
import org.omg.CosNaming.*/;
import org.omg.CORBA.*/;

public class DateClient {
    static Date dateImpl;

    public static void main(String args[]) {
        try {
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // get the root naming context
            org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");

            // Use NamingContextExt instead of NamingContext. This is
            // part of the Interoperable naming Service.
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

            // resolve the Object Reference in Naming
            String name = "Date";
            dateImpl = DateHelper.narrow(ncRef.resolve_str(name));

            System.out.println("Server's Date: "+ dateImpl.getDate());

            dateImpl.shutdown();
        } catch (Exception e) {
            System.out.println("ERROR : " + e);
            e.printStackTrace(System.out);
        }
    }
}

Figure 2.10. An example of a simple CORBA client. Code in
the shaded area implements Java’s CORBA framework.

by allowing for the discovery and spontaneous interaction between services in a
network.

As mentioned in Section 2.3, in order to reduce the programmers’ burden,
RPC systems attempt to mask the differences between local and remote procedures
so that a remote procedure call is treated the same as a local procedure call.

Most distributed systems provide a unified view of objects in terms of their
location so that all objects are considered equal regardless of their physical loca-
tion. Indeed, many distributed systems, including most RPC systems, try and
mask the differences between local and remote objects by providing programmer
transparency.
Waldo et al. [118] argue that this approach is fundamentally wrong and that non-distributed objects cannot be treated the same as distributed objects as there are fundamental differences in terms of latency, memory access, partial failure, and concurrency. They further argue that the merging of the computational models of local and distributed computing is both unwise to attempt and unable to succeed.

Java’s Remote Method Invocation (RMI) takes an entirely different approach to other types of distributed systems. RMI differs not only in the details but in the basic set of assumptions made about the distributed systems in which it operates [116].

While most distributed systems are heterogeneous, RMI assumes that the client and the server are both running in a Java virtual machine and are both written in Java. By doing so, RMI removes the need to describe remote interfaces using a language-neutral IDL. Instead, the Java interface construct is used to declare a remotely accessible interface as shown below:

```java
public interface IDateService extends java.rmi.Remote {
    Date getDate() throws RemoteException;
}
```

The RMI system architecture is illustrated in Figure 2.11. Messages from the client application to the server pass through the stub (or proxy), an implementation of the remote objects exported interface.

The stub object is generated either statically by the RMI compiler, `rmid`, or dynamically at runtime. Unlike standard RPC IDL compilers, such as those provided by CORBA systems, the stubs are generated on the implementation class of the object which the stub refers to. These stub objects therefore support all the remote methods that the remote object’s implementation supports. In a system
such as CORBA, the stub is compiled into the client and linked before runtime. In RMI, the stub originates with the client and is loaded dynamically and may therefore be different for different objects with the same apparent type. The actual type of the stub is loaded at runtime when the system is able to determine the exact type [116].

The stub forwards requests to the server using the remote reference layer. The remote reference layer implements the semantics of the type of invocation, for example unicast or multicast communication. The remote reference layer therefore provides a framework for adding additional types of remote object communication [122], although unicast communication is the only implementation that is provided by default.

The transport layer is responsible for connection setup, connection management and keeping track of and dispatching to remote objects. To dispatch to a remote object, the call is forwarded by the transport to the server specific remote reference layer. The remote reference layer hands the request off to the server’s skeleton, which in turn passes it to the remote object implementation to perform the actual method call. Return values from the call are passed back through the skeleton, the remote reference layer, and finally to the client stub [122].

The RMI system passes parameters and return values either by reference or by value. If the object to be passed is a remote object (it implements the java.rmi.Remote interface) a remote reference is passed. If, however, the object is not a remote object, a copy of that object is passed.

RMI uses Java’s object serialization mechanism to marshal and unmarshal parameters and return values, which encodes objects and any objects they refer to, into a byte stream for transmission from one virtual machine to another. Once the byte stream is received, it is converted into the original object using a process known as de-serialization. RMI therefore requires that all objects and any objects they reference, that are used as parameters or return values, implement the java.io.Serializable interface, a marker interface (one that has no methods) that indicates to the serialization system that they may be safely converted to a byte stream.

The objects that are passed are ‘real objects’ in the sense that they include both the object’s data as well as an annotation describing the type of the object. If an object of a previously unknown type is received, the RMI system fetches the bytecode for the object and loads it into the receiving process. By preserving the object’s type, RMI preserves the basic object-oriented notion of polymorphism [116, 122].

In order to fetch the bytecode of a previously unknown object, RMI uses Java’s dynamic class-loading mechanism. The following classes are loaded during an RMI call [122]:

- Classes of remote objects and their interfaces.
- Stub and skeleton classes that serve as proxies for remote objects.
Other classes used directly in an RMI application, such as parameters and return values.

The actual location of classes that may be needed to be loaded at runtime are defined by the system property `java.rmi.server.codebase`, a URL pointing to the location of the class files. Classes loaded by RMI are subject to security restrictions put in place by the `java.lang.SecurityManager` class installed for the virtual machine downloading the class. For classes downloaded into applets or applications as a result of remote calls, RMI requires a security manager to protect the application and host from potential harm [122].

Java automatically deletes objects that are no longer referenced. RMI extends this to remote objects by using a reference counting mechanism similar to that used by Modula-3 network objects [14]. RMI implements remote garbage collection by keeping track of all live remote references in all virtual machines. When a remote object is first referenced, a count is incremented and a referenced message is sent to the remote object’s RMI runtime. When a live reference is unreferenced, the count is decremented. When the count reaches 0 an unreferenced message is sent to the remote object’s RMI runtime, which is then free to garbage-collect the object.

Clients hold references to remote objects for a certain period of time, called a lease. It is the responsibility of the client to automatically renew the lease before it expires. If the lease expires, the server assumes the client is no longer referencing the remote object and is free to garbage-collect it [122]. Using this mechanism it is still possible, however, for a client to call a remote object that has been garbage-collected. For example, if the network is down for a short period of time and the client’s RMI runtime could not renew the lease, the client could, upon the network connection being restored, call a remote object that has been garbage-collected. In this instance a `java.rmi.RemoteException` exception is thrown.

RMI uses a simple naming service to bootstrap RMI applications. Servers register remote objects they are exporting with a name server called a registry. When a client wishes to obtain a reference to a remote object, a lookup is performed on a registry and a reference to the remote object is returned if the lookup succeeds.

Registry services can be used by either using the traditional RPC mechanism of a centralised registry or by each application maintaining its own registry. Because RMI services generally return remote objects, the registry only needs to be contacted when making initial contact with a remote application because once one of the remote objects on a server has been obtained, additional objects can be obtained via method calls on the first object [28].

### 2.5.1 RMI Example

In order to implement our simple date application, we are firstly required to define the remote interface, illustrated in Figure 2.12, which is required to extend from the `java.rmi.Remote` interface and each remote method is required to declare that it throws the `java.rmi.RemoteException` exception.
public interface IDateServer extends Remote {
    public Date getDate() throws RemoteException;
}

Figure 2.12. RMI interface. RMI requires an interface to be defined listing the methods that are available to remote clients.

public class DateServer implements IDateServer {
    public DateServer() {
        super();
    }

    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String[] args) {
        try {
            IdateServer server = new DateServer();
            IdateServer stub = (IDateServer) UnicastRemoteObject.exportObject(server, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind("DateServer", stub);
            System.out.println("Server Ready");
        } catch (RemoteException e) {
            System.err.println("DateServer exception: ");
            e.printStackTrace();
        }
    }
}

Figure 2.13. An example of an RMI server. Code in the shaded area implements the RMI framework.

To implement our server application we can either extend the java.rmi.server.UnicastRemoteObject, if we would like the remote object to be implicitly exported, or we can explicitly export the object using the exportObject method of the same class.
public class DateClient {

    public static void main(String args[]) {

        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            Registry registry = LocateRegistry.getRegistry();
            IDateServer dateServer = (IDateServer) registry.lookup("DateServer");
            System.out.println("Date on server: " +

                dateServer.getDate().toString());

            } catch (RemoteException e) {
                System.err.println("DateServer Exception:");
                e.printStackTrace();
            } catch (NotBoundException e) {
                System.err.println("Cannot bind to server");
                e.printStackTrace();
            }

        }
    }
}

Figure 2.14. An example of an RMI client. Code in the shaded area implements the RMI framework.

In the example in Figure 2.13, our server explicitly exports a remote object, which returns an object of type Date to the client. The date object is serialized into a byte stream and passed to the client application, where it is deserialized and accessed by the client, as shown by the client implementation in Figure 2.14:

As can be seen in our simple example, the RMI framework is highly intrusive as it requires programmers to define an interface that extends the java.rmi.Remote interface (Figure 2.12) and to implement the interface in the server (Figure 2.13) code. The client code, illustrated in Figure 2.14, contains RMI specific code to locate the server and execute the remote method. In addition, the client is required to be aware of the distributed nature of the application by ensuring that it catches a RemoteException exception should one occur.

2.5.2 Summary

RMI provides a sophisticated environment for distributed computing. However, as Hicks et al. [49] point out, programmers need to take special care to distinguish between remote and local method invocation as the argument passing conven-
tion between the two are different. As well as argument passing conventions, the equals(), hashCode(), and toString() methods of the Object class are overridden by the java.rmi.RemoteObject class to deal appropriately with remote objects, for example by displaying information about the transport of the object in the case of the toString() method. RMI applications do not, therefore, behave the same as local applications, which adds to the programmer’s burden.

The major shortcoming of RMI is that, by ensuring that programmers are aware of the differences between local and remote objects, an additional burden is placed on programmers. While Waldo et al. [118] argue that this is necessary, there are a number of significant issues with RMI’s implementation of this approach:

• Programming is far more complicated than the transparent approach adopted by most RPC type systems.
• The programmer has to mark an object as being remote by having it implement the java.rmi.Remote interface.
• Remote operations have to be declared to throw the java.rmi.RemoteException exception.
• Classes that are marked remote and have operations that are declared to throw java.rmi.RemoteException in an interface, have to be altered to be reused outside RMI.
• Clients, and servers that are also clients, are required to provide a security manager to ensure applications can only access resources they are entitled to.
• Applications are required to use the RMI framework for exporting and locating objects.

2.6 Message-Oriented Middleware

Message-oriented middleware (MOM) systems refer to a type of asynchronous communication known as message queueing [9] where middleware is commonly defined as a software layer that provides a higher level of abstraction, which considerably simplifies distributed systems development [32]. MOM systems are highly successful in industry and represent a sizeable segment of the Information and Communication Technology market [32].

As described by Eugster et al. [34], MOM systems are generally highly scalable as the decoupling of message producer from consumer improves scalability by removing all explicit dependencies between the interacting participants along the following two dimensions:

Time decoupling. The interacting parties do not need to be actively participating in the interaction at the same time. Either party may be disconnected while the other is sending messages to it. Once they become connected, they may be notified of an event sent by the other party and the other party may be currently disconnected.
Space decoupling. The interacting parties do not need to know each other as publishers publish messages through an event service and subscribers receive these events indirectly from the event service. Publishers and subscribers are not aware of each other nor do they hold references to each other.

A number of message queueing systems are widely available, such as MQSeries from IBM [51], Microsoft’s MSMQ [25] and Apache’s ActiveMQ [5] and message queueing is part of the Java Enterprise Edition (JEE) specification [104] in the form of message-driven beans and the Java Message Service (JMS) API [47].

Message queueing systems typically provide two different interaction styles, queues and publish/subscribe.

### 2.6.1 Message Queues

In the queue interaction style, also referred to as point-to-point, messages are stored in a FIFO queue. Producers append messages into the queue and consumers dequeue them at the front of the queue.

Queues typically provide transactional, ordering and timing guarantees and messages can be one way (fire-and-forget) or two way (request-response) although a response is not compulsory.

The JMS API [47] provides a simple queuing abstraction for Java applications. Implementing our simple date application is therefore straightforward. The server implementation illustrated in Figure 2.16 is developed using Apache’s ActiveMQ [5] messaging product.

In our JMS server example we use the JMS `TextMessage` type to pass a String representing the current date from the server to the client. A `Properties` object is used to set various connection parameters required by the JMS implementation and the `DateServer` class is used as a `MessageListener` so that it may receive messages asynchronously.

The JMS Server example uses a simple messaging request-reply pattern [50] where the consumer waits for a message to be sent from the producer and, upon receipt, sends a response back to the producer on a queue defined by the producer in the JMS reply header field. Once a message is received and if it is of type `TextMessage`, the request is printed and a response is sent containing the server’s current date. Our client, illustrated in Figure 2.17, creates a temporary queue for
public class DateServer implements MessageListener {
    private Queue destination;
    private Session session;

    private void initialize() throws JMSException, NamingException {
        Properties props = new Properties();
        props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                           "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
        props.setProperty(Context.PROVIDER_URL,
                           "tcp://localhost:61616");
        props.setProperty("queue.destination","TEST");
        Context ctx = new InitialContext(props);
        QueueConnectionFactory connectionFactory =
            (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
        QueueConnection c = connectionFactory.createQueueConnection();
        destination = (Queue) ctx.lookup("destination");
        session = c.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
        MessageConsumer requestConsumer = session.createConsumer(destination);
        requestConsumer.setMessageListener(this);
        c.start();
    }

    public static void main(String[] args) throws NamingException, JMSException {
        new DateServer().initialize();
    }

    public void onMessage(Message message) {
        try {
            if ((message instanceof TextMessage) &&
                (message.getJMSReplyTo() != null)) {
                TextMessage requestMessage = (TextMessage) message;
                System.out.println("Req Date: "+ requestMessage.getText());
                Destination replyDestination = message.getJMSReplyTo();
                MessageProducer replyProducer =
                    session.createProducer(replyDestination);
                replyMessage.setJMSMessageID();
                replyMessage.setJMSCorrelationID(requestMessage.
                    getJMSMessageID());
                replyProducer.send(replyMessage);
            }
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}

Figure 2.16. JMS server example. The shaded areas illustrate where usage of the JMS framework is required.

the receipt of message responses and sets the JMS reply header field to the name of the temporary queue so that the server knows which queue to use for message responses. Once a message is sent, the client suspends waiting for a response from the server.

As can be seen in Figures 2.16 and 2.17, the programmer is responsible for implementing all aspects of error recovery. In addition, as illustrated by the shad-
public class DateClient {

  private Queue destination;
  private Session session;
  private MessageProducer producer;
  private MessageConsumer consumer;
  private Queue replyQueue;

  public static void main(String[] args) throws JMSException, NamingException {

    DateClient d = new DateClient();
    d.initialize();
    System.out.println("Server’s Date: " + d.getDate());
    System.exit(0);
  }

  private String getDate() throws JMSException {
    TextMessage requestMessage = session.createTextMessage();
    requestMessage.setText(new Date().toString());
    requestMessage.setJMSReplyTo(replyQueue);
    producer.send(requestMessage);
    Message message = (TextMessage) consumer.receive();
    if (message instanceof TextMessage)
      return ((TextMessage) message).getText();
    return "Invalid Message type Received";
  }

  private void initialize() throws JMSException, NamingException {
    Properties props = new Properties();
    props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                     "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
    props.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");
    props.setProperty("queue.destination", "TEST");
    Context ctx = new InitialContext(props);
    QueueConnectionFactory connectionFactory =
      (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
    QueueConnection c = connectionFactory.createQueueConnection();
    destination = (Queue) ctx.lookup("destination");
    session = c.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
    producer = session.createProducer(destination);
    replyQueue = session.createTemporaryQueue();
    consumer = session.createConsumer(replyQueue);
    c.start();
  }
}

Figure 2.17. JMS client example. The shaded areas illustrate where usage of the JMS framework is required.

In the JMS examples, the JMS framework is highly intrusive, requiring a great deal of setup and recovery code.

2.6.2 Publish/Subscribe

In contrast to the synchronous models of communication described earlier, publish/subscribe systems provide a loosely-coupled interaction style where publish-
ers publish events and subscribers subscribe to those events and are subsequently asynchronously notified when an event occurs. Publish/subscribe systems therefore implement an *event-driven* style of communication [78].

This interaction style is illustrated in Figure 2.18, where publishers publish messages to a central event service. Subscribers register their interest in messages that may be placed in the event service by publishers and are notified asynchronously if this occurs.

As well as *time* and *space* decoupling, described in Section 2.6, publish/subscribe also provides *synchronisation* decoupling between publishers and subscribers. Eugster et al. [34] describe *synchronisation decoupling* as the ability for publishers to produce events without blocking and subscribers to receive those events asynchronously through a callback mechanism.

Subscribers are usually only interested in particular events, not all events and this has led to a number of subscription schemes. According to Eugster et al. [34], the most widely used schemes are *topic-based* and *content-based* subscription.

The topic based subscription model is based on the notion of *topics* or *subjects* and is implemented by many enterprise messaging solutions including IBM’s MQ Series [51] and Tibco’s Rendezvous message bus [78]. The JMS API [47] provides a topic abstraction mechanism and in version 1.1 of the standard, the interface between message queues and topics has converged so the API for both types of interaction styles are the same. Topic based publish/subscribe programming is, using the JMS API, the same as message queue programming.

Content-based (also known as *property-based* [87]) publish/subscribe provides a scheme where events are subscribed to, based on a filter mechanism. This is im-
implemented in the JMS standard by meta-data association in the form of *message selectors*, a string containing an expression based on a subset of the SQL92 conditional expression syntax. For example, the message selector:

$$\text{Type} = 'Football' \text{ OR Type} = 'Rugby'$$

selects any message that has a type property that is set to 'Football' or 'Rugby'.

In the JMS API, a message selector may be passed as an argument during the creation of a message consumer and the message consumer will only receive messages whose headers and properties match the selector.

### 2.6.3 Durable Topics

By default, events are only sent to consumers if the consumer is currently available. However, if the durable property is defined, then events are stored in the publish/subscribe system and will be sent to consumers once they become available.

This feature, known as *durable topics*, requires the programmer to define two additional properties so that the publish/subscribe system may uniquely identify a consumer:

- A client ID for the connection so that the system may have many different durable consumers on different topics or on the same topic with different message selectors.
- A subscription name for the consumer.

### 2.7 Chapter Summary

This chapter has discussed four common approaches to distributed systems development which are broadly indicative of current distributed systems development practises. We classify these approaches as follows:

1. The low-level API approach, which accesses the low-level protocol stack directly.
2. The RPC distribution obliviousness approach, which attempts to hide the distributed system from the programmer.
3. The RPC distribution awareness approach, which ensures the programmer is aware of the distributed nature of the application and requires the programmer to follow specific programming conventions.
4. The high-level framework or API approach, which provides a high-level library or framework that is used to hide low-level networking details from the programmer.

BSD sockets are indicative of the first approach and provide low-level access to the networking stack and therefore greater control, but requires significantly more code than other approaches. Programmers are required to implement their
own protocol on top of the socket interface and are responsible for implementing their own packet assembly and disassembly routines. In addition, BSD socket programmers cannot rely on the underlying transport mechanism to ensure message delivery. Error handling and recovery is left entirely up to the programmer, which, combined with the other requirements discussed above, makes socket programming immensely complex and error prone.

Using the second approach, RPC systems attempt to mask the differences between local and remote procedure calls so that, to the programmer, they appear identical. While this approach has its advantages, remote procedure calls do not behave in the same way as local procedure calls, and in the event of an error it is often impossible to recover unless the programmer is aware of the distributed nature of the application and is therefore in a position to take corrective action, for example by reconnecting to a different server. In addition, programmers are required to use an IDL for most RPC type systems, which is used to describe the remote procedure calls, their parameters and other information. An IDL is unique to a distributed system and a programmer is therefore required to learn an additional IDL for each type of RPC system they wish to use.

RMI, an implementation of the third approach, is a Java-centric distributed system that requires the programmer to adopt RMI specific programming conventions. Programmers are required to be aware of the distributed nature of their applications so that they may take corrective action in the event of failures, although there is no specific or general recovery mechanism in the RMI system, rather it is left to the programmer to implement one.

The JMS system uses the fourth approach to provide a high-level API to asynchronous event-driven systems. Once again, error handling and recovery is left to the programmer to resolve.

All of the above approaches require programmers to adhere to a framework or API although the level of abstraction may differ. Regardless of the approach used, programmers are required to interact with the framework or API at some level, thereby tying the application code to the framework or API.
Autonomics Development: A Domain-Specific Aspect Language Approach
Soule, P.
2010, X, 134 p. 39 illus., Softcover
ISBN: 978-3-0346-0539-7
A product of Birkhäuser Basel