5

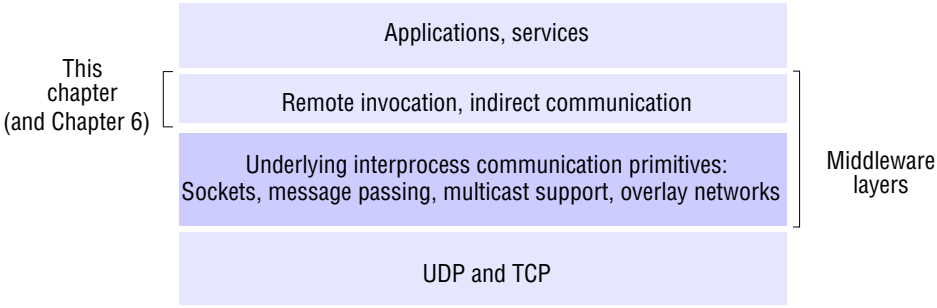
REMOTE INVOCATION

This chapter steps through the remote invocation paradigms introduced in Chapter 2 (indirect communication techniques are addressed in Chapter 6). The chapter starts by examining the most primitive service, request-reply communication, which represents relatively minor enhancements to the underlying interprocess communication primitives discussed in Chapter 4. The chapter then continues by examining the two most prominent remote invocation techniques for communication in distributed systems:

- The remote procedure call (RPC) approach extends the common programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local.
- Remote method invocation (RMI) is similar to RPC but for distributed objects, with added benefits in terms of using object-oriented programming concepts in distributed systems and also extending the concept of an object reference to the global distributed environments, and allowing the use of object references as parameters in remote invocations.

The chapter also features Java RMI as a case study of the remote method invocation approach (further insight can also be gained in Chapter 8, where we look at CORBA).

Figure 5.1 Middleware layers



5.1 Introduction

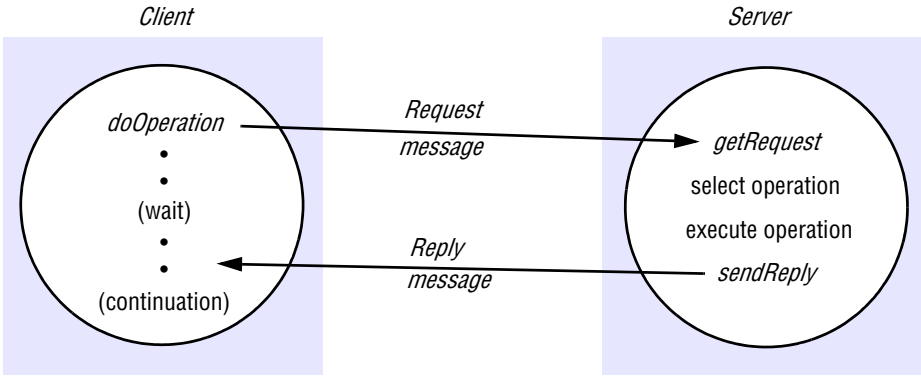
This chapter is concerned with how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system, examining, in particular, the remote invocation paradigms defined in Chapter 2:

- *Request-reply protocols* represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing. In particular, such protocols provide relatively low-level support for requesting the execution of a remote operation, and also provide direct support for RPC and RMI, discussed below.
- The earliest and perhaps the best-known example of a more programmer-friendly model was the extension of the conventional procedure call model to distributed systems (the *remote procedure call*, or RPC, model), which allows client programs to call procedures transparently in server programs running in separate processes and generally in different computers from the client.
- In the 1990s, the object-based programming model was extended to allow objects in different processes to communicate with one another by means of *remote method invocation* (RMI). RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.

Note that we use the term 'RMI' to refer to remote method invocation in a generic way – this should not be confused with particular examples of remote method invocation such as Java RMI.

Returning to the diagram first introduced in Chapter 4 (and reproduced in Figure 5.1), this chapter, together with Chapter 6, continues our study of middleware concepts by focusing on the layer above interprocess communication. In particular, Sections 5.2 through 5.4 focus on the styles of communication listed above, with Section 5.5 providing a more complex case study, Java RMI.

Figure 5.2 Request-reply communication



5.2 Request-reply protocols

This form of communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server. It can also be reliable because the reply from the server is effectively an acknowledgement to the client. Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later – see Section 7.5.2.

The client-server exchanges are described in the following paragraphs in terms of the *send* and *receive* operations in the Java API for UDP datagrams, although many current implementations use TCP streams. A protocol built over datagrams avoids unnecessary overheads associated with the TCP stream protocol. In particular:

- Acknowledgements are redundant, since requests are followed by replies.
- Establishing a connection involves two extra pairs of messages in addition to the pair required for a request and a reply.
- Flow control is redundant for the majority of invocations, which pass only small arguments and results.

The request-reply protocol • The protocol we describe here is based on a trio of communication primitives, *doOperation*, *getRequest* and *sendReply*, as shown in Figure 5.2. This request-reply protocol matches requests to replies. It may be designed to provide certain delivery guarantees. If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message. Figure 5.3 outlines the three communication primitives.

The *doOperation* method is used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation. Its result is a byte array containing the reply. It is assumed that the client calling *doOperation* marshals the

Figure 5.3 Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
Sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();
Acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
Sends the reply message *reply* to the client at its Internet address and port.

arguments into an array of bytes and unmarshals the results from the array of bytes that is returned. The first argument of *doOperation* is an instance of the class *RemoteRef*, which represents references for remote servers. This class provides methods for getting the Internet address and port of the associated server. The *doOperation* method sends a request message to the server whose Internet address and port are specified in the remote reference given as an argument. After sending the request message, *doOperation* invokes *receive* to get a reply message, from which it extracts the result and returns it to the caller. The caller of *doOperation* is blocked until the server performs the requested operation and transmits a reply message to the client process.

getRequest is used by a server process to acquire service requests, as shown in Figure 5.3. When the server has invoked the specified operation, it then uses *sendReply* to send the reply message to the client. When the reply message is received by the client the original *doOperation* is unblocked and execution of the client program continues.

The information to be transmitted in a request message or a reply message is shown in Figure 5.4. The first field indicates whether the message is a *Request* or a *Reply* message. The second field, *requestId,* contains a message identifier. A *doOperation* in the client generates a *requestId* for each request message, and the server copies these IDs into the corresponding reply messages. This enables *doOperation* to check that a reply message is the result of the current request, not a delayed earlier call. The third field is a remote reference. The fourth field is an identifier for the operation to be invoked. For example, the operations in an interface might be numbered 1, 2, 3, ... , if the client and server use a common language that supports reflection, a representation of the operation itself may be put in this field.

Figure 5.4 Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

Message identifiers • Any scheme that involves the management of messages to provide additional properties such as reliable message delivery or request-reply communication requires that each message have a unique message identifier by which it may be referenced. A message identifier consists of two parts:

1. a *requestId*, which is taken from an increasing sequence of integers by the sending process;
2. an identifier for the sender process, for example, its port and Internet address.

The first part makes the identifier unique to the sender, and the second part makes it unique in the distributed system. (The second part can be obtained independently – for example, if UDP is in use, from the message received.)

When the value of the *requestId* reaches the maximum value for an unsigned integer (for example, $2^{32} - 1$) it is reset to zero. The only restriction here is that the lifetime of a message identifier should be much less than the time taken to exhaust the values in the sequence of integers.

Failure model of the request-reply protocol • If the three primitives *doOperation*, *getRequest* and *sendReply* are implemented over UDP datagrams, then they suffer from the same communication failures. That is:

- They suffer from omission failures.
- Messages are not guaranteed to be delivered in sender order.

In addition, the protocol can suffer from the failure of processes (see Section 2.4.2). We assume that processes have crash failures. That is, when they halt, they remain halted – they do not produce Byzantine behaviour.

To allow for occasions when a server has failed or a request or reply message is dropped, *doOperation* uses a timeout when it is waiting to get the server's reply message. The action taken when a timeout occurs depends upon the delivery guarantees being offered.

Timeouts • There are various options as to what *doOperation* can do after a timeout. The simplest option is to return immediately from *doOperation* with an indication to the client that the *doOperation* has failed. This is not the usual approach – the timeout may have been due to the request or reply message getting lost and in the latter case, the operation will have been performed. To compensate for the possibility of lost messages, *doOperation* sends the request message repeatedly until either it gets a reply or it is reasonably sure that the delay is due to lack of response from the server rather than to lost messages. Eventually, when *doOperation* returns, it will indicate to the client by an exception that no result was received.

Discarding duplicate request messages • In cases when the request message is retransmitted, the server may receive it more than once. For example, the server may receive the first request message but take longer than the client's timeout to execute the command and return the reply. This can lead to the server executing an operation more than once for the same request. To avoid this, the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter out duplicates. If the server has not yet sent the reply, it need take no special action – it will transmit the reply when it has finished executing the operation.

Lost reply messages • If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result, unless it has stored the result of the original execution. Some servers can execute their operations more than once and obtain the same results each time. An *idempotent operation* is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once. For example, an operation to add an element to a set is an idempotent operation because it will always have the same effect on the set each time it is performed, whereas an operation to append an item to a sequence is not an idempotent operation because it extends the sequence each time it is performed. A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once.

History • For servers that require retransmission of replies without re-execution of operations, a history may be used. The term 'history' is used to refer to a structure that contains a record of (reply) messages that have been transmitted. An entry in a history contains a request identifier, a message and an identifier of the client to which it was sent. Its purpose is to allow the server to retransmit reply messages when client processes request them. A problem associated with the use of a history is its memory cost. A history will become very large unless the server can tell when the messages will no longer be needed for retransmission.

As clients can make only one request at a time, the server can interpret each request as an acknowledgement of its previous reply. Therefore the history need contain only the last reply message sent to each client. However, the volume of reply messages in a server's history may still be a problem when it has a large number of clients. This is compounded by the fact that, when a client process terminates, it does not acknowledge the last reply it has received – messages in the history are therefore normally discarded after a limited period of time.

Styles of exchange protocols • Three protocols, that produce differing behaviours in the presence of communication failures are used for implementing various types of request behaviour. They were originally identified by Spector [1982]:

- the *request (R)* protocol;
- the *request-reply (RR)* protocol;
- the *request-reply-acknowledge reply (RRA)* protocol.

The messages passed in these protocols are summarized in Figure 5.5. In the R protocol, a single *Request* message is sent by the client to the server. The R protocol may be used when there is no value to be returned from the remote operation and the client requires no confirmation that the operation has been executed. The client may proceed immediately after the request message is sent as there is no need to wait for a reply message. This protocol is implemented over UDP datagrams and therefore suffers from the same communication failures.

The RR protocol is useful for most client-server exchanges because it is based on the request-reply protocol. Special acknowledgement messages are not required, because a server's reply message is regarded as an acknowledgement of the client's request message. Similarly, a subsequent call from a client may be regarded as an acknowledgement of a server's reply message. As we have seen, communication

Figure 5.5 RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

failures due to UDP datagrams being lost may be masked by the retransmission of requests with duplicate filtering and the saving of replies in a history for retransmission.

The RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply. The *Acknowledge reply* message contains the *requestId* from the reply message being acknowledged. This will enable the server to discard entries from its history. The arrival of a *requestId* in an acknowledgement message will be interpreted as acknowledging the receipt of all reply messages with lower *requestId*s, so the loss of an acknowledgement message is harmless. Although the exchange involves an additional message, it need not block the client, as the acknowledgement may be transmitted after the reply has been given to the client. However it does use processing and network resources. Exercise 5.10 suggests an optimization to the RRA protocol.

Use of TCP streams to implement the request-reply protocol • Section 4.2.3 mentioned that it is often difficult to decide on an appropriate size for the buffer in which to receive datagrams. In the request-reply protocol, this applies to the buffers used by the server to receive request messages and by the client to receive replies. The limited length of datagrams (usually 8 kilobytes) may not be regarded as adequate for use in transparent RMI or RPC systems, since the arguments or results of procedures may be of any size.

The desire to avoid implementing multipacket protocols is one of the reasons for choosing to implement request-reply protocols over TCP streams, allowing arguments and results of any size to be transmitted. In particular, Java object serialization is a stream protocol that allows arguments and results to be sent over streams between the client and server, making it possible for collections of objects of any size to be transmitted reliably. If the TCP protocol is used, it ensures that request and reply messages are delivered reliably, so there is no need for the request-reply protocol to deal with retransmission of messages and filtering of duplicates or with histories. In addition the flow-control mechanism allows large arguments and results to be passed without taking special measures to avoid overwhelming the recipient. Thus the TCP protocol is chosen for request-reply protocols because it can simplify their implementation. If successive requests and replies between the same client-server pair are sent over the same stream, the connection overhead need not apply to every remote invocation. Also, the overhead due to TCP acknowledgement messages is reduced when a reply message follows soon after a request message.

Howeever, if the application does not require all of the facilities offered by TCP, a more efficient, specially tailored protocol can be implemented over UDP. For example, Sun NFS does not require support for messages of unlimited size, since it

transmits fixed-size file blocks between client and server. In addition to that, its operations are designed to be idempotent, so it does not matter if operations are executed more than once in order to retransmit lost reply messages, making it unnecessary to maintain a history.

HTTP: An example of a request-reply protocol • Chapter 1 introduced the HyperText Transfer Protocol (HTTP) used by web browser clients to make requests to web servers and to receive replies from them. To recap, web servers manage resources implemented in different ways:

- as data – for example the text of an HTML page, an image or the class of an applet;
- as a program – for example, servlets [java.sun.com III], or PHP or Python programs that run on the web server.

Client requests specify a URL that includes the DNS hostname of a web server and an optional port number on the web server as well as the identifier of a resource on that server.

HTTP is a protocol that specifies the messages involved in a request-reply exchange, the methods, arguments and results, and the rules for representing (marshalling) them in the messages. It supports a fixed set of methods (*GET*, *PUT*, *POST,* etc) that are applicable to all of the server's resources. It is unlike the previously described protocols, where each service has its own set of operations. In addition to invoking methods on web resources, the protocol allows for content negotiation and password-style authentication:

Content negotiation: Clients' requests can include information as to what data representations they can accept (for example, language or media type), enabling the server to choose the representation that is the most appropriate for the user.

Authentication: Credentials and challenges are used to support password-style authentication. On the first attempt to access a password-protected area, the server reply contains a challenge applicable to the resource. Chapter 11 explains challenges. When a client receives a challenge, it gets the user to type a name and password and submits the associated credentials with subsequent requests.

HTTP is implemented over TCP. In the original version of the protocol, each client-server interaction consisted of the following steps:

- The client requests and the server accepts a connection at the default server port or at a port specified in the URL.
- The client sends a request message to the server.
- The server sends a reply message to the client.
- The connection is closed.

However, establishing and closing a connection for every request-reply exchange is expensive, overloading the server and causing too many messages to be sent over the network. Bearing in mind that browsers generally make multiple requests to the same server – for example, to get the images in a page just supplied – a later version of the protocol (HTTP 1.1, see RFC 2616 [Fielding *et al.* 1999]) uses *persistent connections* – connections that remain open over a series of request-reply exchanges between client

Figure 5.6 HTTP *Request* message

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

and server. A persistent connection can be closed by the client or server at any time by sending an indication to the other participant. Servers will close a persistent connection when it has been idle for a period of time. It is possible that a client may receive a message from the server saying that the connection is closed while it is in the middle of sending another request or requests. In such cases, the browser will resend the requests without user involvement, provided that the operations involved are idempotent. For example, the method *GET* described below is idempotent. Where non-idempotent operations are involved, the browser should consult the user as to what to do next.

Requests and replies are marshalled into messages as ASCII text strings, but resources can be represented as byte sequences and may be compressed. The use of text in the external data representation has simplified the use of HTTP for application programmers who work directly with the protocol. In this context, a textual representation does not add much to the length of the messages.

Data resources are supplied as MIME-like structures in arguments and results. Multipurpose Internet Mail Extensions (MIME), specified in RFC 2045 [Freed and Borenstein 1996], is a standard for sending multipart data containing, for example, text, images and sound in email messages. Data is prefixed with its MIME type so that the recipient will know how to handle it. A MIME type specifies a type and a subtype, for example, *text/plain*, *text/html*, *image/gif* or *image/jpeg*. Clients can also specify the MIME types that they are willing to accept.

HTTP methods • Each client request specifies the name of a method to be applied to a resource at the server and the URL of that resource. The reply reports on the status of the request. Requests and replies may also contain resource data, the contents of a form or the output of a program resource run on the web server. The methods include the following:

GET: Requests the resource whose URL is given as its argument. If the URL refers to data, then the web server replies by returning the data identified by that URL. If the URL refers to a program, then the web server runs the program and returns its output to the client. Arguments may be added to the URL; for example, *GET* can be used to send the contents of a form to a program as an argument. The *GET* operation can be made conditional on the date a resource was last modified. *GET* can also be configured to obtain parts of the data.

With *GET*, all the information for the request is provided in the URL (see, for example, the query string in Section 1.6).

HEAD: This request is identical to *GET*, but it does not return any data. However, it does return all the information about the data, such as the time of last modification, its type or its size.

POST: Specifies the URL of a resource (for example a program) that can deal with the data supplied in the body of the request. The processing carried out on the data depends on the function of the program specified in the URL. This method is used when the action may change data on the server. It is designed to deal with:

- providing a block of data to a data-handling process such as a servlet – for example, submitting a web form to buy something from a web site;
- posting a message to a mailing list or updating details of members of the list;
- extending a database with an append operation.

PUT: Requests that the data supplied in the request is stored with the given URL as its identifier, either as a modification of an existing resource or as a new resource.

DELETE: The server deletes the resource identified by the given URL. Servers may not always allow this operation, in which case the reply indicates failure.

OPTIONS: The server supplies the client with a list of methods it allows to be applied to the given URL (for example *GET*, *HEAD*, *PUT*) and its special requirements.

TRACE: The server sends back the request message. Used for diagnostic purposes.

The operations *PUT* and *DELETE* are idempotent, but *POST* is not necessarily so because it can change the state of a resource. The others are *safe* operations in that they do not change anything.

The requests described above may be intercepted by a proxy server (see Section 2.3.1). The responses to *GET* and *HEAD* may be cached by proxy servers.

Message contents • The *Request* message specifies the name of a method, the URL of a resource, the protocol version, some headers and an optional message body. Figure 5.6 shows the contents of an HTTP *Request* message whose method is *GET*. When the URL specifies a data resource, the *GET* method does not have a message body.

Requests to proxies need the absolute URL, as shown in Figure 5.6. Requests to origin servers (the origin server is where the resource resides) specify a pathname and give the DNS name of the origin server in a *Host* header field. For example,

GET /index.html HTTP/1.1
Host: www.dcs.qmul.ac.uk

In general, the header fields contain request modifiers and client information, such as conditions on the latest date of modification of the resource or acceptable content types (for example, HTML text, audio or JPEG images). An authorization field can be used to provide the client's credentials in the form of a certificate specifying their rights to access a resource.

A *Reply* message specifies the protocol version, a status code and 'reason', some headers and an optional message body, as shown in Figure 5.7. The status code and reason provide a report on the server's success or otherwise in carrying out the request: the former is a three-digit integer for interpretation by a program, and the latter is a textual phrase that can be understood by a person. The header fields are used to pass additional information about the server or access to the resource. For example, if the request requires authentication, the status of the response indicates this and a header

Figure 5.7 HTTP *Reply* message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

field contains a challenge. Some status returns have quite complex effects. In particular, a 303 status response tells the browser to look under a different URL, which is supplied in a header field in the reply. It is intended for use in a response from a program activated by a *POST* request when the program needs to redirect the browser to a selected resource.

The message body in request or reply messages contains the data associated with the URL specified in the request. The message body has its own headers specifying information about the data, such as its length, its MIME type, its character set, its content encoding and the last date it was modified. The MIME type field specifies the type of the data, for example *image/jpeg* or *text/plain*. The content encoding field specifies the compression algorithm to be used

5.3 Remote procedure call

As mentioned in Chapter 2, the concept of a remote procedure call (RPC) represents a major intellectual breakthrough in distributed computing, with the goal of making the programming of distributed systems look similar, if not identical, to conventional programming – that is, achieving a high level of distribution transparency. This unification is achieved in a very simple manner, by extending the abstraction of a procedure call to distributed environments. In particular, in RPC, procedures on remote machines can be called as if they are procedures in the local address space. The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call. This concept was first introduced by Birrell and Nelson [1984] and paved the way for many of the developments in distributed systems programming used today.

5.3.1 Design issues for RPC

Before looking at the implementation of RPC systems, we look at three issues that are important in understanding this concept:

- the style of programming promoted by RPC – programming with interfaces;
- the call semantics associated with RPC;
- the key issue of transparency and how it relates to remote procedure calls.

Programming with interfaces • Most modern programming languages provide a means of organizing a program as a set of modules that can communicate with one another. Communication between modules can be by means of procedure calls between modules

or by direct access to the variables in another module. In order to control the possible interactions between modules, an explicit *interface* is defined for each module. The interface of a module specifies the procedures and the variables that can be accessed from other modules. Modules are implemented so as to hide all the information about them except that which is available through its interface. So long as its interface remains the same, the implementation may be changed without affecting the users of the module.

Interfaces in distributed systems: In a distributed program, the modules can run in separate processes. In the client-server model, in particular, each server provides a set of procedures that are available for use by clients. For example, a file server would provide procedures for reading and writing files. The term *service interface* is used to refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.

There are a number of benefits to programming with interfaces in distributed systems, stemming from the important separation between interface and implementation:

- As with any form of modular programming, programmers are concerned only with the abstraction offered by the service interface and need not be aware of implementation details.
- Extrapolating to (potentially heterogeneous) distributed systems, programmers also do not need to know the programming language or underlying platform used to implement the service (an important step towards managing heterogeneity in distributed systems).
- This approach provides natural support for software evolution in that implementations can change as long as long as the interface (the external view) remains the same. More correctly, the interface can also change as long as it remains compatible with the original.

The definition of service interfaces is influenced by the distributed nature of the underlying infrastructure:

- It is not possible for a client module running in one process to access the variables in a module in another process. Therefore the service interface cannot specify direct access to variables. Note that CORBA IDL interfaces can specify attributes, which seems to break this rule. However, the attributes are not accessed directly but by means of some getter and setter procedures added automatically to the interface.
- The parameter-passing mechanisms used in local procedure calls − for example, call by value and call by reference, are not suitable when the caller and procedure are in different processes. In particular, call by reference is not supported. Rather, the specification of a procedure in the interface of a module in a distributed program describes the parameters as *input* or *output*, or sometimes both. *Input* parameters are passed to the remote server by sending the values of the arguments in the request message and then supplying them as arguments to the operation to be executed in the server. *Output* parameters are returned in the reply message and are used as the result of the call or to replace the values of the corresponding

Figure 5.8 CORBA IDL example

```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
};
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

variables in the calling environment. When a parameter is used for both input and output, the value must be transmitted in both the request and reply messages.

- Another difference between local and remote modules is that addresses in one process are not valid in another remote one. Therefore, addresses cannot be passed as arguments or returned as results of calls to remote modules.

These constraints have a significant impact on the specification of interface definition languages, as discussed below.

Interface definition languages: An RPC mechanism can be integrated with a particular programming language if it includes an adequate notation for defining interfaces, allowing input and output parameters to be mapped onto the language's normal use of parameters. This approach is useful when all the parts of a distributed application can be written in the same language. It is also convenient because it allows the programmer to use a single language, for example, Java, for local and remote invocation.

However, many existing useful services are written in C++ and other languages. It would be beneficial to allow programs written in a variety of languages, including Java, to access them remotely. *Interface definition languages* (IDLs) are designed to allow procedures implemented in different languages to invoke one another. An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified.

Figure 5.8 shows a simple example of CORBA IDL. The *Person* structure is the same as the one used to illustrate marshalling in Section 4.3.1. The interface named *PersonList* specifies the methods available for RMI in a remote object that implements that interface. For example, the method *addPerson* specifies its argument as *in*, meaning that it is an *input* argument, and the method *getPerson* that retrieves an instance of *Person* by name specifies its second argument as *out*, meaning that it is an *output* argument.

Figure 5.9 Call semantics

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

The concept of an IDL was initially developed for RPC systems but applies equally to RMI and also web services. Our case studies include:

- Sun XDR as an example of an IDL for RPC (in Section 5.3.3);
- CORBA IDL as an example of an IDL for RMI (in Chapter 8 and also included above);
- the Web Services Description Language (WSDL), which is designed for an Internet-wide RPC supporting web services (see Section 9.3);
- and protocol buffers used at Google for storing and interchanging many kinds of structured information (see Section 21.4.1).

RPC call semantics • Request-reply protocols were discussed in Section 5.2, where we showed that *doOperation* can be implemented in different ways to provide different delivery guarantees. The main choices are:

Retry request message: Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

Duplicate filtering: Controls when retransmissions are used and whether to filter out duplicate requests at the server.

Retransmission of results: Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker. Figure 5.9 shows the choices of interest, with corresponding names for the semantics that they produce. Note that for local procedure calls, the semantics are *exactly once*, meaning that every procedure is executed exactly once (except in the case of process failure). The choices of RPC invocation semantics are defined as follows.

Maybe semantics: With *maybe* semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:

- omission failures if the request or result message is lost;
- crash failures when the server containing the remote operation fails.

If the result message has not been received after a timeout and there are no retries, it is uncertain whether the procedure has been executed. If the request message was lost, then the procedure will not have been executed. On the other hand, the procedure may have been executed and the result message lost. A crash failure may occur either before or after the procedure is executed. Moreover, in an asynchronous system, the result of executing the procedure may arrive after the timeout. *Maybe* semantics is useful only for applications in which occasional failed calls are acceptable.

At-least-once semantics: With *at-least-once* semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received. *At-least-once* semantics can be achieved by the retransmission of request messages, which masks the omission failures of the request or result message. *At-least-once* semantics can suffer from the following types of failure:

- crash failures when the server containing the remote procedure fails;
- arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values to be stored or returned.

Section 5.2 defines an *idempotent operation* as one that can be performed repeatedly with the same effect as if it had been performed exactly once. Non-idempotent operations can have the wrong effect if they are performed more than once. For example, an operation to increase a bank balance by $10 should be performed only once; if it were to be repeated, the balance would grow and grow! If the operations in a server can be designed so that all of the procedures in their service interfaces are idempotent operations, then *at-least-once* call semantics may be acceptable.

At-most-once semantics: With *at-most-once* semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all. *At-most-once* semantics can be achieved by using all of the fault-tolerance measures outlined in Figure 5.9. As in the previous case, the use of retries masks any omission failures of the request or result messages. This set of fault tolerance measures prevents arbitrary failures by ensuring that for each RPC a procedure is never executed more than once. Sun RPC (discussed in Section 5.3.3) provides at-least-once call semantics.

Transparency • The originators of RPC, Birrell and Nelson [1984], aimed to make remote procedure calls as much like local procedure calls as possible, with no distinction in syntax between a local and a remote procedure call. All the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call. Although request messages are retransmitted after a timeout, this is transparent to the caller to make the semantics of remote procedure calls like that of local procedure calls.

More precisely, returning to the terminology of Chapter 1, RPC strives to offer at least location and access transparency, hiding the physical location of the (potentially remote) procedure and also accessing local and remote procedures in the same way. Middleware can also offer additional levels of transparency to RPC.

However, remote procedure calls are more vulnerable to failure than local ones, since they involve a network, another computer and another process. Whichever of the above semantics is chosen, there is always the chance that no result will be received, and in the case of failure, it is impossible to distinguish between failure of the network and of the remote server process. This requires that clients making remote calls are able to recover from such situations.

The latency of a remote procedure call is several orders of magnitude greater than that of a local one. This suggests that programs that make use of remote calls need to be able to take this factor into account, perhaps by minimizing remote interactions. The designers of Argus [Liskov and Scheifler 1982] suggested that a caller should be able to abort a remote procedure call that is taking too long in such a way that it has no effect on the server. To allow this, the server would need to be able to restore things to how they were before the procedure was called. These issues are discussed in Chapter 16.

Remote procedure calls also require a different style of parameter passing, as discussed above. In particular, RPC does not offer call by reference.

Waldo *et al.* [1994] say that the difference between local and remote operations should be expressed at the service interface, to allow participants to react in a consistent way to possible partial failures. Other systems went further than this by arguing that the syntax of a remote call should be different from that of a local call: in the case of Argus, the language was extended to make remote operations explicit to the programmer.

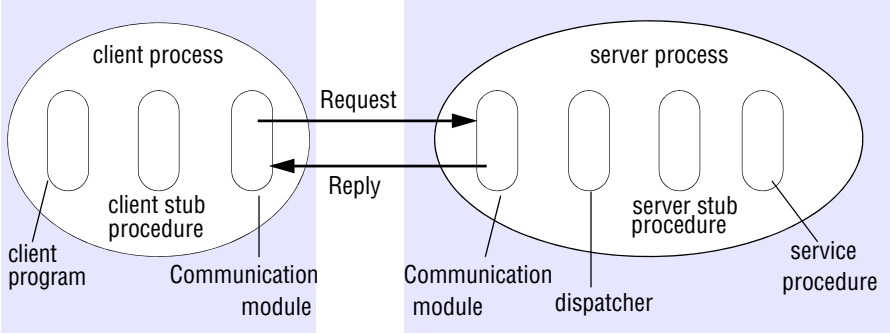
The choice as to whether RPC should be transparent is also available to the designers of IDLs. For example, in some IDLs, a remote invocation may throw an exception when the client is unable to communicate with a remote procedure. This requires that the client program handle such exceptions, allowing it to deal with such failures. An IDL can also provide a facility for specifying the call semantics of a procedure. This can help the designer of the service – for example, if *at-least-once* call semantics is chosen to avoid the overheads of *at-most-once*, the operations must be designed to be idempotent.

The current consensus is that remote calls should be made transparent in the sense that the syntax of a remote call is the same as that of a local invocation, but that the difference between local and remote calls should be expressed in their interfaces.

5.3.2 Implementation of RPC

The software components required to implement RPC are shown in Figure 5.10. The client that accesses a service includes one *stub procedure* for each procedure in the service interface. The stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server. When the reply message arrives, it unmarshals the results. The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface. The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message. The server stub procedure

Figure 5.10 Role of client and server stub procedures in RPC



then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message. The service procedures implement the procedures in the service interface. The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition of the service.

RPC is generally implemented over a request-reply protocol like the ones discussed in Section 5.2. The contents of request and reply messages are the same as those illustrated for request-reply protocols in Figure 5.4. RPC may be implemented to have one of the choices of invocation semantics discussed in Section 5.3.1 – *at-least-once* or *at-most-once* is generally chosen. To achieve this, the communication module will implement the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results, as shown in Figure 5.9.

5.3.3 Case study: Sun RPC

RFC 1831 [Srinivasan 1995a] describes Sun RPC, which was designed for client-server communication in the Sun Network File System (NFS). Sun RPC is sometimes called ONC (Open Network Computing) RPC. It is supplied as a part of the various Sun and other UNIX operating systems and is also available with NFS installations. Implementors have the choice of using remote procedure calls over either UDP or TCP. When Sun RPC is used with UDP, request and reply messages are restricted in length – theoretically to 64 kilobytes, but more often in practice to 8 or 9 kilobytes. It uses *at-least-once* call semantics. Broadcast RPC is an option.

The Sun RPC system provides an interface language called XDR and an interface compiler called *rpcgen*, which is intended for use with the C programming language.

Interface definition language • The Sun XDR language, which was originally designed for specifying external data representations, was extended to become an interface definition language. It may be used to define a service interface for Sun RPC by specifying a set of procedure definitions together with supporting type definitions. The notation is rather primitive in comparison with that used by CORBA IDL or Java. In particular:

Figure 5.11 Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
        int length;
        char buffer[MAX];
};
struct writeargs {
        FileIdentifier f;
        FilePointer position;
        Data data;
};
struct readargs {
        FileIdentifier f;
        FilePointer position;
        Length length;
};

program FILEREADWRITE {
        version VERSION {
                void WRITE(writeargs)=1;                    1
                Data READ(readargs)=2;                      2
        }=2;
} = 9999;
```

- Most languages allow interface names to be specified, but Sun RPC does not – instead of this, a program number and a version number are supplied. The program numbers can be obtained from a central authority to allow every program to have its own unique number. The version number is intended to be changed when a procedure signature changes. Both program and version number are passed in the request message, so the client and server can check that they are using the same version.
- A procedure definition specifies a procedure signature and a procedure number. The procedure number is used as a procedure identifier in request messages.
- Only a single input parameter is allowed. Therefore, procedures requiring multiple parameters must include them as components of a single structure.
- The output parameters of a procedure are returned via a single result.
- The procedure signature consists of the result type, the name of the procedure and the type of the input parameter. The type of both the result and the input parameter may specify either a single value or a structure containing several values.

For example, see the XDR definition in Figure 5.11 of an interface with a pair of procedures for writing and reading files. The program number is 9999 and the version number is 2. The *READ* procedure (line 2) takes as its input parameter a structure with three components specifying a file identifier, a position in the file and the number of bytes required. Its result is a structure containing the number of bytes returned and the file data. The *WRITE* procedure (line 1) has no result. The *WRITE* and *READ* procedures are given numbers 1 and 2. The number 0 is reserved for a null procedure, which is generated automatically and is intended to be used to test whether a server is available.

This interface definition language provides a notation for defining constants, typedefs, structures, enumerated types, unions and programs. Typedefs, structures and enumerated types use the C language syntax. The interface compiler *rpcgen* can be used to generate the following from an interface definition:

- client stub procedures;
- server *main* procedure, dispatcher and server stub procedures;
- XDR marshalling and unmarshalling procedures for use by the dispatcher and client and server stub procedures.

Binding • Sun RPC runs a local binding service called the *port mapper* at a well-known port number on each computer. Each instance of a port mapper records the program number, version number and port number in use by each service running locally. When a server starts up it registers its program number, version number and port number with the local port mapper. When a client starts up, it finds out the server's port by making a remote request to the port mapper at the server's host, specifying the program number and version number.

When a service has multiple instances running on different computers, the instances may use different port numbers for receiving client requests. If a client needs to multicast a request to all the instances of a service that are using different port numbers, it cannot use a direct IP multicast message for this purpose. The solution is that clients make multicast remote procedure calls by multicasting them to all the port mappers, specifying the program and version number. Each port mapper forwards all such calls to the appropriate local service program, if there is one.

Authentication. Sun RPC request and reply messages provide additional fields enabling authentication information to be passed between client and server. The request message contains the credentials of the user running the client program. For example, in the UNIX style of authentication the credentials include the *uid* and *gid* of the user. Access control mechanisms can be built on top of the authentication information which is made available to the server procedures via a second argument. The server program is responsible for enforcing access control by deciding whether to execute each procedure call according to the authentication information. For example, if the server is an NFS file server, it can check whether the user has sufficient rights to carry out a requested file operation. Several different authentication protocols can be supported. These include:

- none;
- UNIX style, as described above;
- a style in which a shared key is established for signing the RPC messages;
- Kerberos (see Chapter 11).

A field in the RPC header indicates which style is being used.

A more generic approach to security is described in RFC 2203 [Eisler *et al*. 1997]. It provides for the secrecy and integrity of RPC messages as well as authentication. It allows the client and server to negotiate a security context in which either no security is applied, or in the case that security is required, message integrity or message privacy or both may be applied.

Client and server programs • Further material on Sun RPC is available at www.cdk5.net/rmi. It includes example client and server programs corresponding to the interface defined in Figure 5.11.

5.4 Remote method invocation

Remote method invocation (RMI) is closely related to RPC but extended into the world of distributed objects. In RMI, a calling object can invoke a method in a potentially remote object. As with RPC, the underlying details are generally hidden from the user. The commonalities between RMI and RPC are as follows:

- They both support programming with interfaces, with the resultant benefits that stem from this approach (see Section 5.3.1).
- They are both typically constructed on top of request-reply protocols and can offer a range of call semantics such as *at-least-once* and *at-most-once*.
- They both offer a similar level of transparency − that is, local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions.

The following differences lead to added expressiveness when it comes to the programming of complex distributed applications and services.

- The programmer is able to use the full expressive power of object-oriented programming in the development of distributed systems software, including the use of objects, classes and inheritance, and can also employ related object-oriented design methodologies and associated tools.
- Building on the concept of object identity in object-oriented systems, all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.

The issue of parameter passing is particularly important in distributed systems. RMI allows the programmer to pass parameters not only by value, as input or output parameters, but also by object reference. Passing references is particularly attractive if the underlying parameter is large or complex. The remote end, on receiving an object reference, can then access this object using remote method invocation, instead of having to transmit the object value across the network.

The rest of this section examines the concept of remote method invocation in more detail, looking initially at the key issues surrounding distributed object models before looking at implementation issues surrounding RMI, including distributed garbage collection.

5.4.1 Design issues for RMI

As mentioned above, RMI shares the same design issues as RPC in terms of programming with interfaces, call semantics and level of transparency. The reader is encouraged to refer back to Section 5.3.1 for discussion of these items.

The key added design issue relates to the object model and, in particular, achieving the transition from objects to distributed objects. We first describe the conventional, single-image object model and then describe the distributed object model.

The object model • An object-oriented program, for example in Java or C++, consists of a collection of interacting objects, each of which consists of a set of data and a set of methods. An object communicates with other objects by invoking their methods, generally passing arguments and receiving results. Objects can encapsulate their data and the code of their methods. Some languages, for example Java and C++, allow programmers to define objects whose instance variables can be accessed directly. But for use in a distributed object system, an object's data should be accessible only via its methods.

Object references: Objects can be accessed via object references. For example, in Java, a variable that appears to hold an object actually holds a reference to that object. To invoke a method in an object, the object reference and method name are given, together with any necessary arguments. The object whose method is invoked is sometimes called the *target* and sometimes the *receiver*. Object references are first-class values, meaning that they may, for example, be assigned to variables, passed as arguments and returned as results of methods.

Interfaces: An interface provides a definition of the signatures of a set of methods (that is, the types of their arguments, return values and exceptions) without specifying their implementation. An object will provide a particular interface if its class contains code that implements the methods of that interface. In Java, a class may implement several interfaces, and the methods of an interface may be implemented by any class. An interface also defines types that can be used to declare the type of variables or of the parameters and return values of methods. Note that interfaces do not have constructors.

Actions : Action in an object-oriented program is initiated by an object invoking a method in another object. An invocation can include additional information (arguments) needed to carry out the method. The receiver executes the appropriate method and then returns control to the invoking object, sometimes supplying a result. An invocation of a method can have three effects:

1. The state of the receiver may be changed.
2. A new object may be instantiated, for example, by using a constructor in Java or C++.
3. Further invocations on methods in other objects may take place.

As an invocation can lead to further invocations of methods in other objects, an action is a chain of related method invocations, each of which eventually returns.

Exceptions: Programs can encounter many sorts of errors and unexpected conditions of varying seriousness. During the execution of a method, many different problems may be discovered: for example, inconsistent values in the object's variables, or failures in

attempts to read or write to files or network sockets. When programmers need to insert tests in their code to deal with all possible unusual or erroneous cases, this detracts from the clarity of the normal case. Exceptions provide a clean way to deal with error conditions without complicating the code. In addition, each method heading explicitly lists as exceptions the error conditions it might encounter, allowing users of the method to deal with them. A block of code may be defined to *throw* an exception whenever particular unexpected conditions or errors arise. This means that control passes to another block of code that *catches* the exception. Control does not return to the place where the exception was thrown.

Garbage collection: It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed. A language such as Java, that can detect automatically when an object is no longer accessible recovers the space and makes it available for allocation to other objects. This process is called *garbage collection*. When a language (for example, C++) does not support garbage collection, the programmer has to cope with the freeing of space allocated to objects. This can be a major source of errors.

Distributed objects • The state of an object consists of the values of its instance variables. In the object-based paradigm the state of a program is partitioned into separate parts, each of which is associated with an object. Since object-based programs are logically partitioned, the physical distribution of objects into different processes or computers in a distributed system is a natural extension (the issue of placement is discussed in Section 2.3.1).

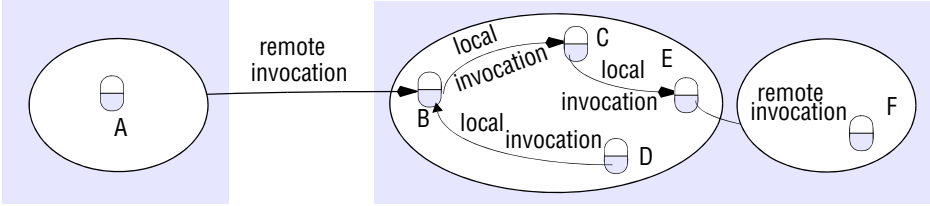
Distributed object systems may adopt the client-server architecture. In this case, objects are managed by servers and their clients invoke their methods using remote method invocation. In RMI, the client's request to invoke a method of an object is sent in a message to the server managing the object. The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message. To allow for chains of related invocations, objects in servers are allowed to become clients of objects in other servers.

Distributed objects can assume other architectural models. For example, objects can be replicated in order to obtain the usual benefits of fault tolerance and enhanced performance, and objects can be migrated with a view to enhancing their performance and availability.

Having client and server objects in different processes enforces *encapsulation*. That is, the state of an object can be accessed only by the methods of the object, which means that it is not possible for unauthorized methods to act on the state. For example, the possibility of concurrent RMIs from objects in different computers implies that an object may be accessed concurrently. Therefore the possibility of conflicting accesses arises. However, the fact that the data of an object is accessed only by its own methods allows objects to provide methods for protecting themselves against incorrect accesses. For example, they may use synchronization primitives such as condition variables to protect access to their instance variables.

Another advantage of treating the shared state of a distributed program as a collection of objects is that an object may be accessed via RMI, or it may be copied into a local cache and accessed directly, provided that the class implementation is available locally.

Figure 5.12 Remote and local method invocations



The fact that objects are accessed only via their methods gives rise to another advantage of heterogeneous systems, that different data formats may be used at different sites – these formats will be unnoticed by clients that use RMI to access the methods of the objects.

The distributed object model • This section discusses extensions to the object model to make it applicable to distributed objects. Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations, as shown in Figure 5.12. Method invocations between objects in different processes, whether in the same computer or not, are known as remote method invocations. Method invocations between objects in the same process are local method invocations.

We refer to objects that can receive remote invocations as *remote objects*. In Figure 5.12, the objects B and F are remote objects. All objects can receive local invocations, although they can receive them only from other objects that hold references to them. For example, object C must have a reference to object E so that it can invoke one of its methods. The following two fundamental concepts are at the heart of the distributed object model:

Remote object references: Other objects can invoke the methods of a remote object if they have access to its *remote object reference*. For example, a remote object reference for B in Figure 5.12 must be available to A.

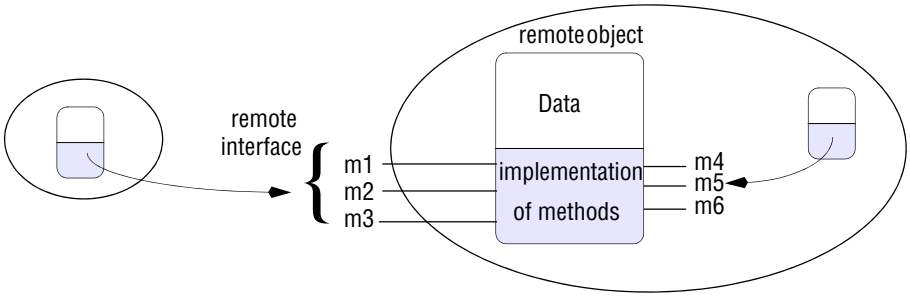
Remote interfaces: Every remote object has a *remote interface* that specifies which of its methods can be invoked remotely. For example, the objects B and F in Figure 5.12 must have remote interfaces.

We look at remote object references, remote interfaces and other aspects of the distributed object model next.

Remote object references: The notion of object reference is extended to allow any object that can receive an RMI to have a remote object reference. A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object. Its representation, which is generally different from that of a local object reference is discussed in Section 4.3.4. Remote object references are analogous to local ones in that:

1. The remote object to receive a remote method invocation is specified by the invoker as a remote object reference.
2. Remote object references may be passed as arguments and results of remote method invocations.

Figure 5.13 A remote object and its remote interface



Remote interfaces: The class of a remote object implements the methods of its remote interface, for example as public instance methods in Java. Objects in other processes can invoke only the methods that belong to its remote interface, as shown in Figure 5.13. Local objects can invoke the methods in the remote interface as well as other methods implemented by a remote object. Note that remote interfaces, like all interfaces, do not have constructors.

The CORBA system provides an interface definition language (IDL), which is used for defining remote interfaces. See Figure 5.8 for an example of a remote interface defined in CORBA IDL. The classes of remote objects and the client programs may be implemented in any language for which an IDL compiler is available, such as C++, Java or Python. CORBA clients need not use the same language as the remote object in order to invoke its methods remotely.

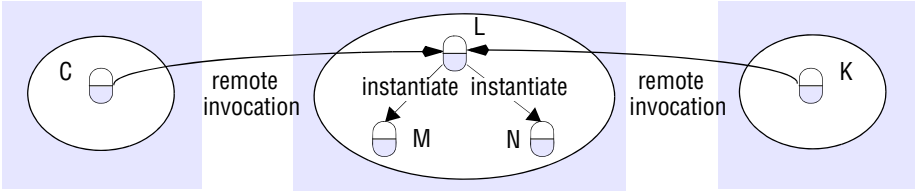
In Java RMI, remote interfaces are defined in the same way as any other Java interface. They acquire their ability to be remote interfaces by extending an interface named *Remote*. Both CORBA IDL (Chapter 8) and Java support multiple inheritance of interfaces. That is, an interface is allowed to extend one or more other interfaces.

Actions in a distributed object system • As in the non-distributed case, an action is initiated by a method invocation, which may result in further invocations on methods in other objects. But in the distributed case, the objects involved in a chain of related invocations may be located in different processes or different computers. When an invocation crosses the boundary of a process or computer, RMI is used, and the remote reference of the object must be available to the invoker. In Figure 5.12, object A needs to hold a remote object reference to object B. Remote object references may be obtained as the results of remote method invocations. For example, object A in Figure 5.12 might obtain a remote reference to object F from object B.

When an action leads to the instantiation of a new object, that object will normally live within the process where instantiation is requested – for example, where the constructor was used. If the newly instantiated object has a remote interface, it will be a remote object with a remote object reference.

Distributed applications may provide remote objects with methods for instantiating objects that can be accessed by RMI, thus effectively providing the effect of remote instantiation of objects. For example, if the object L in Figure 5.14 contains a method for creating remote objects, then the remote invocations from C and K could lead to the instantiation of the objects M and N, respectively.

Figure 5.14 Instantiation of remote objects



Garbage collection in a distributed-object system: If a language, for example Java, supports garbage collection, then any associated RMI system should allow garbage collection of remote objects. Distributed garbage collection is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting. Section 5.4.3 describes such a scheme in detail. If garbage collection is not available, then remote objects that are no longer required should be deleted.

Exceptions: Any remote invocation may fail for reasons related to the invoked object being in a different process or computer from the invoker. For example, the process containing the remote object may have crashed or may be too busy to reply, or the invocation or result message may be lost. Therefore, remote method invocation should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked. Examples of the latter are an attempt to read beyond the end of a file, or to access a file without the correct permissions.

CORBA IDL provides a notation for specifying application-level exceptions, and the underlying system generates standard exceptions when errors due to distribution occur. CORBA client programs need to be able to handle exceptions. For example, a C++ client program will use the exception mechanisms in C++.

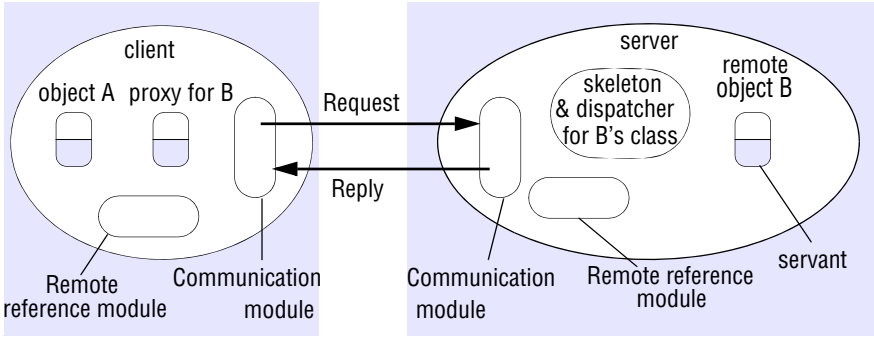
5.4.2 Implementation of RMI

Several separate objects and modules are involved in achieving a remote method invocation. These are shown in Figure 5.15, in which an application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference. This section discusses the roles of each of the components shown in that figure, dealing first with the communication and remote reference modules and then with the RMI software that runs over them.

We then explore the following related topics: the generation of proxies, the binding of names to their remote object references, the activation and passivation of objects and the location of objects from their remote object references.

Communication module • The two cooperating communication modules carry out the request-reply protocol, which transmits *request* and *reply* messages between the client and server. The contents of *request* and *reply* messages are shown in Figure 5.4. The communication module uses only the first three items, which specify the message type, its *requestId* and the remote reference of the object to be invoked. The *operationId* and

Figure 5.15 The role of proxy and skeleton in remote method invocation



all the marshalling and unmarshalling are the concern of the RMI software, discussed below. The communication modules are together responsible for providing a specified invocation semantics, for example *at-most-once*.

The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on its local reference, which it gets from the remote reference module in return for the remote object identifier in the *request* message. The role of dispatcher is discussed in the forthcoming section on RMI software.

Remote reference module • A remote reference module is responsible for translating between local and remote object references and for creating remote object references. To support its responsibilities, the remote reference module in each process has a *remote object table* that records the correspondence between local object references in that process and remote object references (which are system-wide). The table includes:

- An entry for all the remote objects held by the process. For example, in Figure 5.15 the remote object B will be recorded in the table at the server.
- An entry for each local proxy. For example, in Figure 5.15 the proxy for B will be recorded in the table at the client.

The role of a proxy is discussed in the subsection on RMI software. The actions of the remote reference module are as follows:

- When a remote object is to be passed as an argument or a result for the first time, the remote reference module is asked to create a remote object reference, which it adds to its table.
- When a remote object reference arrives in a *request* or *reply* message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object. In the case that the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.

This module is called by components of the RMI software when they are marshalling and unmarshalling remote object references. For example, when a *request* message arrives, the table is used to find out which local object is to be invoked.

Servants • A *servant* is an instance of a class that provides the body of a remote object. It is the servant that eventually handles the remote requests passed on by the corresponding skeleton. Servants live within a server process. They are created when remote objects are instantiated and remain in use until they are no longer needed, finally being garbage collected or deleted.

The RMI software • This consists of a layer of software between the application-level objects and the communication and remote reference modules. The roles of the middleware objects shown in Figure 5.15 are as follows:

Proxy: The role of a proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker; but instead of executing an invocation, it forwards it in a message to a remote object. It hides the details of the remote object reference, the marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client. There is one proxy for each remote object for which a process holds a remote object reference. The class of a proxy implements the methods in the remote interface of the remote object it represents, which ensures that remote method invocations are suitable for the type of the remote object. However, the proxy implements them quite differently. Each method of the proxy marshals a reference to the target object, its own *operationId* and its arguments into a *request* message and sends it to the target. It then awaits the *reply* message, unmarshals it and returns the results to the invoker.

Dispatcher: A server has one dispatcher and one skeleton for each class representing a remote object. In our example, the server has a dispatcher and a skeleton for the class of remote object B. The dispatcher receives *request* messages from the communication module. It uses the *operationId* to select the appropriate method in the skeleton, passing on the *request* message. The dispatcher and the proxy use the same allocation of *operationIds* to the methods of the remote interface.

Skeleton: The class of a remote object has a *skeleton*, which implements the methods in the remote interface. They are implemented quite differently from the methods in the servant that incarnates a remote object. A skeleton method unmarshals the arguments in the *request* message and invokes the corresponding method in the servant. It waits for the invocation to complete and then marshals the result, together with any exceptions, in a *reply* message to the sending proxy's method.

Remote object references are marshalled in the form shown in Figure 4.13, which includes information about the remote interface of the remote object – for example, the name of the remote interface or the class of the remote object. This information enables the proxy class to be determined so that a new proxy may be created when it is needed. For example, the proxy class name may be generated by appending *_proxy* to the name of the remote interface.

Generation of the classes for proxies, dispatchers and skeletons • The classes for the proxy, dispatcher and skeleton used in RMI are generated automatically by an interface compiler. For example, in the Orbix implementation of CORBA, interfaces of remote objects are defined in CORBA IDL, and the interface compiler can be used to generate the classes for proxies, dispatchers and skeletons in C++ or in Java [www.iona.com]. For Java RMI, the set of methods offered by a remote object is defined as a Java interface

that is implemented within the class of the remote object. The Java RMI compiler generates the proxy, dispatcher and skeleton classes from the class of the remote object.

Dynamic invocation: An alternative to proxies • The proxy just described is static, in the sense that its class is generated from an interface definition and then compiled into the client code. Sometimes this is not practical, though. Suppose that a client program receives a remote reference to an object whose remote interface was not available at compile time. In this case it needs another way to invoke the remote object. *Dynamic invocation* gives the client access to a generic representation of a remote invocation like the *doOperation* method used in Exercise 5.18, which is available as part of the infrastructure for RMI (see Section 5.4.1). The client will supply the remote object reference, the name of the method and the arguments to *doOperation* and then wait to receive the results.

Note that although the remote object reference includes information about the interface of the remote object, such as its name, this is not enough – the names of the methods and the types of the arguments are required for making a dynamic invocation. CORBA provides this information via a component called the Interface Repository, which is described in Chapter 8.

The dynamic invocation interface is not as convenient to use as a proxy, but it is useful in applications where some of the interfaces of the remote objects cannot be predicted at design time. An example of such an application is the shared whiteboard that we use to illustrate Java RMI (Section 5.5), CORBA (Chapter 8) and web services (Section 9.2.3). To summarize: the shared whiteboard application displays many different types of shapes, such as circles, rectangles and lines, but it should also be able to display new shapes that were not predicted when the client was compiled. A client that uses dynamic invocation is able to address this challenge. We shall see in Section 5.5 that the dynamic downloading of classes to clients is an alternative to dynamic invocation. This is available in Java RMI – a single-language system.

Dynamic skeletons: It is clear, from the above example, that it can also arise that a server will need to host remote objects whose interfaces were not known at compile time. For example, a client may supply a new type of shape to the shared whiteboard server for it to store. A server with dynamic skeletons would be able to deal with this situation. We defer describing dynamic skeletons until the chapter on CORBA (Chapter 8). However, as we shall see in Section 5.5, Java RMI addresses this problem by using a generic dispatcher and the dynamic downloading of classes to the server.

Server and client programs • The server program contains the classes for the dispatchers and skeletons, together with the implementations of the classes of all of the servants that it supports. In addition, the server program contains an *initialization* section (for example, in a *main* method in Java or C++). The initialization section is responsible for creating and initializing at least one of the servants to be hosted by the server. Additional servants may be created in response to requests from clients. The initialization section may also register some of its servants with a binder (see below). Generally it will register just one servant, which can be used to access the rest.

The client program will contain the classes of the proxies for all of the remote objects that it will invoke. It can use a binder to look up remote object references.

Factory methods: We noted earlier that remote object interfaces cannot include constructors. This means that servants cannot be created by remote invocation on constructors. Servants are created either in the initialization section or in methods in a remote interface designed for that purpose. The term *factory method* is sometimes used to refer to a method that creates servants, and a *factory object* is an object with factory methods. Any remote object that needs to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose. Such methods are called factory methods, although they are really just normal methods.

The binder • Client programs generally require a means of obtaining a remote object reference for at least one of the remote objects held by a server. For example, in Figure 5.12, object A would require a remote object reference for object B. A *binder* in a distributed system is a separate service that maintains a table containing mappings from textual names to remote object references. It is used by servers to register their remote objects by name and by clients to look them up. Chapter 8 contains a discussion of the CORBA Naming Service. The Java binder, RMIregistry, is discussed briefly in the case study on Java RMI in Section 5.5.

Server threads • Whenever an object executes a remote invocation, that execution may lead to further invocations of methods in other remote objects, which may take some time to return. To avoid the execution of one remote invocation delaying the execution of another, servers generally allocate a separate thread for the execution of each remote invocation. When this is the case, the designer of the implementation of a remote object must allow for the effects on its state of concurrent executions.

Activation of remote objects • Some applications require that information survive for long periods of time. However, it is not practical for the objects representing such information to be kept in running processes for unlimited periods, particularly since they are not necessarily in use all of the time. To avoid the potential waste of resources that would result from to running all of the servers that manage remote objects all of the time, the servers can be started whenever they are needed by clients, as is done for the standard set of TCP services such as FTP, which are started on demand by a service called *Inetd*. Processes that start server processes to host remote objects are called *activators*, for the following reasons.

A remote object is described as *active* when it is available for invocation within a running process, whereas it is called *passive* if is not currently active but can be made active. A passive object consists of two parts:

1. the implementation of its methods;
2. its state in the marshalled form.

Activation consists of creating an active object from the corresponding passive object by creating a new instance of its class and initializing its instance variables from the stored state. Passive objects can be activated on demand, for example when they need to be invoked by other objects.

An *activator* is responsible for:

- registering passive objects that are available for activation, which involves recording the names of servers against the URLs or file names of the corresponding passive objects;

- starting named server processes and activating remote objects in them;
- keeping track of the locations of the servers for remote objects that it has already activated.

Java RMI provides the ability to make some remote objects *activatable* [java.sun.com IX]. When an activatable object is invoked, if that object is not currently active, the object is made active from its marshalled state and then passed the invocation. It uses one activator on each server computer.

The CORBA case study in Chapter 8 describes the implementation repository – a weak form of activator that starts services containing objects in an initial state.

Persistent object stores • An object that is guaranteed to live between activations of processes is called a *persistent object*. Persistent objects are generally managed by persistent object stores, which store their state in a marshalled form on disk. Examples include the CORBA persistent state service (see Chapter 8), Java Data Objects [java.sun.com VIII] and Persistent Java [Jordan 1996; java.sun.com IV].

In general, a persistent object store will manage very large numbers of persistent objects, which are stored on disk or in a database until they are needed. They will be activated when their methods are invoked by other objects. Activation is generally designed to be transparent – that is, the invoker should not be able to tell whether an object is already in main memory or has to be activated before its method is invoked. Persistent objects that are no longer needed in main memory can be passivated. In most cases, objects are saved in the persistent object store whenever they reach a consistent state, for the sake of fault tolerance. The persistent object store needs a strategy for deciding when to passivate objects. For example, it may do so in response to a request in the program that activated the objects, either at the end of a transaction or when the program exits. Persistent object stores generally attempt to optimize passivation by saving only those objects that have been modified since the last time they were saved.

Persistent object stores generally allow collections of related persistent objects to have human-readable names such as pathnames or URLs. In practice, each human-readable name is associated with the root of a connected set of persistent objects.

There are two approaches to deciding whether an object is persistent or not:

- The persistent object store maintains some persistent roots, and any object that is reachable from a persistent root is defined to be persistent. This approach is used by Persistent Java, Java Data Objects and PerDiS [Ferreira *et al.* 2000]. They make use of a garbage collector to dispose of objects that are no longer reachable from the persistent roots.
- The persistent object store provides some classes on which persistence is based – persistent objects belong to their subclasses. For example, in Arjuna [Parrington *et al.* 1995], persistent objects are based on C++ classes that provide transactions and recovery. Unwanted objects must be deleted explicitly.

Some persistent object stores, such as PerDiS and Khazana [Carter *et al.* 1998], allow objects to be activated in multiple caches local to users, instead of in servers. In this case, a cache consistency protocol is required. Further details on consistency models can be found on the companion web site, in the chapter from the fourth edition on distributed shared memory [www.cdk5.net/dsm].

Object location • Section 4.3.4 describes a form of remote object reference that contains the Internet address and port number of the process that created the remote object as a way of guaranteeing uniqueness. This form of remote object reference can also be used as an address for a remote object, so long as that object remains in the same process for the rest of its life. But some remote objects will exist in a series of different processes, possibly on different computers, throughout their lifetime. In this case, a remote object reference cannot act as an address. Clients making invocations require both a remote object reference and an address to which to send invocations.

A *location service* helps clients to locate remote objects from their remote object references. It uses a database that maps remote object references to their probable current locations – the locations are probable because an object may have migrated again since it was last heard of. For example, the Clouds system [Dasgupta *et al.* 1991] and the Emerald system [Jul *et al.* 1988] used a cache/broadcast scheme in which a member of a location service on each computer holds a small cache of remote object reference-to-location mappings. If a remote object reference is in the cache, that address is tried for the invocation and will fail if the object has moved. To locate an object that has moved or whose location is not in the cache, the system broadcasts a request. This scheme may be enhanced by the use of forward location pointers, which contain hints as to the new location of an object. Another example is the resolution service required for resolving the URN of a resource into its current URL, mentioned in Section 9.1.

5.4.3 Distributed garbage collection

The aim of a distributed garbage collector is to ensure that if a local or remote reference to an object is still held anywhere in a set of distributed objects, the object itself will continue to exist, but as soon as no object any longer holds a reference to it, the object will be collected and the memory it uses recovered.

We describe the Java distributed garbage collection algorithm, which is similar to the one described by Birrell *et al.* [1995]. It is based on reference counting. Whenever a remote object reference enters a process, a proxy will be created and will stay there for as long as it is needed. The process where the object lives (its server) should be informed of the new proxy at the client. Then later when there is no longer a proxy at the client, the server should be informed. The distributed garbage collector works in cooperation with the local garbage collectors as follows:

- Each server process maintains a set of the names of the processes that hold remote object references for each of its remote objects; for example, *B.holders* is the set of client processes (virtual machines) that have proxies for object *B*. (In Figure 5.15, this set will include the client process illustrated.) This set can be held in an additional column in the remote object table.
- When a client *C* first receives a remote reference to a particular remote object, *B*, it makes an *addRef(B)* invocation to the server of that remote object and then creates a proxy; the server adds *C* to *B.holders.*

- When a client *C*'s garbage collector notices that a proxy for remote object *B* is no longer reachable, it makes a *removeRef(B)* invocation to the corresponding server and then deletes the proxy; the server removes *C* from *B.holders.*
- When *B.holders* is empty, the server's local garbage collector will reclaim the space occupied by *B* unless there are any local holders.

This algorithm is intended to be carried out by means of pairwise request-reply communication with *at-most-once* invocation semantics between the remote reference modules in processes – it does not require any global synchronization. Note also that the extra invocations made on behalf of the garbage collection algorithm do not affect every normal RMI; they occur only when proxies are created and deleted.

There is a possibility that one client may make a *removeRef(B)* invocation at about the same time as another client makes an *addRef(B)* invocation. If the *removeRef* arrives first and *B.holders* is empty, the remote object *B* could be deleted before the *addRef* arrives. To avoid this situation, if the set *B.holders* is empty at the time when a remote object reference is transmitted, a temporary entry is added until the *addRef* arrives.

The Java distributed garbage collection algorithm tolerates communication failures by using the following approach. The *addRef* and *removeRef* operations are idempotent. In the case that an *addRef(B)* call returns an exception (meaning that the method was either executed once or not at all), the client will not create a proxy but will make a *removeRef(B)* call. The effect of *removeRef* is correct whether or not the *addRef* succeeded. The case where *removeRef* fails is dealt with by *leases*.

The Java distributed garbage collection algorithm can tolerate the failure of client processes. To achieve this, servers lease their objects to clients for a limited period of time. The lease period starts when the client makes an *addRef* invocation to the server. It ends either when the time has expired or when the client makes a *removeRef* invocation to the server. The information stored by the server concerning each lease contains the identifier of the client's virtual machine and the period of the lease. Clients are responsible for requesting the server to renew their leases before they expire.

Leases in Jini • The Jini distributed system includes a specification for leases [Arnold *et al.* 1999] that can be used in a variety of situations when one object offers a resource to another object – for example, when remote objects offer references to other objects. Objects that offer such resources are at risk of having to maintain the resources when the users are no longer interested or their programs have exited. To avoid complicated protocols to discover whether the resource users are still interested, the resources are offered for a limited period of time. The granting of the use of a resource for a period of time is called a *lease*. The object offering the resource will maintain it until the time in the lease expires. The resource users are responsible for requesting their renewal when they expire.

The period of a lease may be negotiated between the grantor and the recipient in Jini, although this does not happen with the leases used in Java RMI. In Jini, an object representing a lease implements the *Lease* interface. It contains information about the period of the lease and methods enabling the lease to be renewed or cancelled. The grantor returns an instance of a *Lease* when it supplies a resource to another object.

5.5 Case study: Java RMI

Java RMI extends the Java object model to provide support for distributed objects in the Java language. In particular, it allows objects to invoke methods on remote objects using the same syntax as for local invocations. In addition, type checking applies equally to remote invocations as to local ones. However, an object making a remote invocation is aware that its target is remote because it must handle *RemoteException*s; and the implementor of a remote object is aware that it is remote because it must implement the *Remote* interface. Although the distributed object model is integrated into Java in a natural way, the semantics of parameter passing differ because the invoker and target are remote from one another.

The programming of distributed applications in Java RMI should be relatively simple because it is a single-language system – remote interfaces are defined in the Java language. If a multiple-language system such as CORBA is used, the programmer needs to learn an IDL and to understand how it maps onto the implementation language. However, even in a single-language system, the programmer of a remote object must consider its behaviour in a concurrent environment.

In the remainder of this introduction, we give an example of a remote interface, then discuss the parameter-passing semantics with reference to the example. Finally, we discuss the downloading of classes and the binder. The second section of this case study discusses how to build client and server programs for the example interface. The third section is concerned with the design and implementation of Java RMI. For full details of Java RMI, see the tutorial on remote invocation [java.sun.com I].

In this case study, the CORBA case study in Chapter 8 and the discussion of web services in Chapter 9, we use a *shared whiteboard* as an example. This is a distributed program that allows a group of users to share a common view of a drawing surface containing graphical objects, such as rectangles, lines and circles, each of which has been drawn by one of the users. The server maintains the current state of a drawing by providing an operation for clients to inform it about the latest shape one of their users has drawn and keeping a record of all the shapes it has received. The server also provides operations allowing clients to retrieve the latest shapes drawn by other users by polling the server. The server has a version number (an integer) that it increments each time a new shape arrives and attaches to the new shape. The server provides operations allowing clients to enquire about its version number and the version number of each shape, so that they may avoid fetching shapes that they already have.

Remote interfaces in Java RMI • Remote interfaces are defined by extending an interface called *Remote* provided in the *java.rmi* package. The methods must throw *RemoteException*, but application-specific exceptions may also be thrown. Figure 5.16 shows an example of two remote interfaces called *Shape* and *ShapeList*. In this example, *GraphicalObject* is a class that holds the state of a graphical object − for example, its type, its position, enclosing rectangle, line colour and fill colour − and provides operations for accessing and updating its state. *GraphicalObject* must implement the *Serializable* interface. Consider the interface *Shape* first: the *getVersion* method returns an integer, whereas the *getAllState* method returns an instance of the class *GraphicalObject*. Now consider the interface *ShapeList*: its *newShape* method passes an instance of *GraphicalObject* as its argument but returns an object with a remote

Figure 5.16 Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;          1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;      2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

interface (that is, a remote object) as its result. An important point to note is that both ordinary objects and remote objects can appear as arguments and results in a remote interface. The latter are always denoted by the name of their remote interface. In the next subsection, we discuss how ordinary objects and remote objects are passed as arguments and results.

Parameter and result passing • In Java RMI, the parameters of a method are assumed to be *input* parameters and the result of a method is a single *output* parameter. Section 4.3.2 describes Java serialization, which is used for marshalling arguments and results in Java RMI. Any object that is serializable – that is, that implements the *Serializable* interface – can be passed as an argument or result in Java RMI. All primitive types and remote objects are serializable. Classes for arguments and result values are downloaded to the recipient by the RMI system where necessary.

Passing remote objects: When the type of a parameter or result value is defined as a remote interface, the corresponding argument or result is always passed as a remote object reference. For example, in Figure 5.16, line 2, the return value of the method *newShape* is defined as *Shape* – a remote interface. When a remote object reference is received, it can be used to make RMI calls on the remote object to which it refers.

Passing non-remote objects: All serializable non-remote objects are copied and passed by value. For example, in Figure 5.16 (lines 2 and 1) the argument of *newShape* and the return value of *getAllState* are both of type *GraphicalObject*, which is serializable and is passed by value. When an object is passed by value, a new object is created in the receiver's process. The methods of this new object can be invoked locally, possibly causing its state to differ from the state of the original object in the sender's process.

Thus, in our example, the client uses the method *newShape* to pass an instance of *GraphicalObject* to the server; the server makes a remote object of type *Shape* containing the state of the *GraphicalObject* and returns a remote object reference to it. The arguments and return values in a remote invocation are serialized to a stream using the method described in Section 4.3.2, with the following modifications:

Figure 5.17 The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)
This method is used by a server to register the identifier of a remote object by name, as shown in Figure 5.18, line 3.

void bind (String name, Remote obj)
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)
This method removes a binding.

Remote lookup(String name)
This method is used by clients to look up a remote object by name, as shown in Figure 5.20, line 1. A remote object reference is returned.

String [] list()
This method returns an array of *Strings* containing the names bound in the registry.

1. Whenever an object that implements the *Remote* interface is serialized, it is replaced by its remote object reference, which contains the name of its (the remote object's) class.
2. When any object is serialized, its class information is annotated with the location of the class (as a URL), enabling the class to be downloaded by the receiver.

Downloading of classes • Java is designed to allow classes to be downloaded from one virtual machine to another. This is particularly relevant to distributed objects that communicate by means of remote invocation. We have seen that non-remote objects are passed by value and remote objects are passed by reference as arguments and results of RMIs. If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically. Similarly, if the recipient of a remote object reference does not already possess the class for a proxy, its code is downloaded automatically. This has two advantages:

1. There is no need for every user to keep the same set of classes in their working environment.
2. Both client and server programs can make transparent use of instances of new classes whenever they are added.

As an example, consider the whiteboard program and suppose that its initial implementation of *GraphicalObject* does not allow for text. A client with a textual object can implement a subclass of *GraphicalObject* that deals with text and pass an instance to the server as an argument of the *newShape* method. After that, other clients may retrieve the instance using the *getAllState* method. The code of the new class will be downloaded automatically from the first client to the server and then to other clients as needed.

Figure 5.18 Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();                    1
            ShapeList stub =                                                  2
                (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);3
            Naming.rebind("//bruno.ShapeList", stub );                        4
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

RMIregistry • The RMIregistry is the binder for Java RMI. An instance of RMIregistry should normally run on every server computer that hosts remote objects. It maintains a table mapping textual, URL-style names to references to remote objects hosted on that computer. It is accessed by methods of the *Naming* class, whose methods take as an argument a URL-formatted string of the form:

//computerName:port/objectName

where *computerName* and *port* refer to the location of the RMIregistry. If they are omitted, the local computer and default port are assumed. Its interface offers the methods shown in Figure 5.17, in which the exceptions are not listed – all of the methods can throw a *RemoteException*.

Used in this way, clients must direct their *lookup* enquiries to particular hosts. Alternatively, it is possible to set up a system-wide binding service. To achieve this, it is necessary to run an instance of the RMIregistry in the networked environment and then use the class *LocateRegistry*, which is in *java.rmi.registry*, to discover this registry. More specifically, this class contains a *getRegistry* method that returns an object of type *Registry* representing the remote binding service:

public static Registry getRegistry() throws RemoteException

Following this, it is then necessary to issue a call of *rebind* on this returned *Registry* object to establish a connection with the remote RMIregistry.

5.5.1 Building client and server programs

This section outlines the steps necessary to produce client and server programs that use the *Remote* interfaces *Shape* and *ShapeList* shown in Figure 5.16. The server program is a simplified version of a whiteboard server that implements the two interfaces *Shape* and *ShapeList*. We describe a simple polling client program and then introduce the callback

Figure 5.19 Java class *ShapeListServant* implements interface *ShapeList*

```
import java.util.Vector;
public class ShapeListServant implements ShapeList {
     private Vector theList;              // contains the list of Shapes
     private int version;
     public ShapeListServant(){...}
     public Shape newShape(GraphicalObject g) {                    1
          version++;
          Shape s = new ShapeServant( g, version);                 2
          theList.addElement(s);
          return s;
     }
     public  Vector allShapes(){...}
     public int getVersion()  { ... }
}
```

technique that can be used to avoid the need to poll the server. Complete versions of the classes illustrated in this section are available at www.cdk5.net/rmi.

Server program • The server is a whiteboard server: it represents each shape as a remote object instantiated by a servant that implements the *Shape* interface and holds the state of a graphical object as well as its version number; it represents its collection of shapes by using another servant that implements the *ShapeList* interface and holds a collection of shapes in a *Vector*.

The server program consists of a *main* method and a servant class to implement each of its remote interfaces. The *main* method of the server class is shown in Figure 5.18, with the key steps contained in the lines marked 1 to 4:

- In line 1, the server creates an instance of *ShapeListServant.*
- Lines 2 and 3 use the method *exportObject* (defined on *UnicastRemoteObject*) to make this object available to the RMI runtime, thereby making it available to receive incoming invocations. The second parameter of *exportObject* specifies the TCP port to be used for incoming invocations. It is normal practice to set this to zero, implying that an anonymous port will be used (one that is generated by the RMI runtime). Using *UnicastRemoteObject* ensures that the resultant object lives only as long as the process in which it is created (an alternative is to make this an *Activatable* object that is, one that lives beyond the server instance).
- Finally, line 4 binds the remote object to a name in the RMIregistry. Note that the value bound to the name is a remote object reference, and its type is the type of its remote interface – *ShapeList*.

The two servant classes are *ShapeListServant*, which implements the *ShapeList* interface, and *ShapeServant*, which implements the *Shape* interface. Figure 5.19 gives an outline of the class *ShapeListServant*.

Figure 5.20 Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");     1
            Vector sList = aShapeList.allShapes();                            2
        } catch(RemoteException e) {System.out.println(e.getMessage());
        }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

The implementation of the methods of the remote interface in a servant class is completely straightforward because it can be done without any concern for the details of communication. Consider the method *newShape* in Figure 5.19 (line 1), which could be called a factory method because it allows the client to request the creation of a servant. It uses the constructor of *ShapeServant*, which creates a new servant containing the *GraphicalObject* and version number passed as arguments. The type of the return value of *newShape* is *Shape* – the interface implemented by the new servant. Before returning, the method *newShape* adds the new shape to its vector that contains the list of shapes (line 2).

The *main* method of a server needs to create a security manager to enable Java security to apply the protection appropriate for an RMI server. A default security manager called *RMISecurityManager* is provided. It protects the local resources to ensure that the classes that are loaded from remote sites cannot have any effect on resources such as files, but it differs from the standard Java security manager in allowing the program to provide its own class loader and to use reflection. If an RMI server sets no security manager, proxies and classes can only be loaded from the local classpath, in order to protect the program from code that is downloaded as a result of remote method invocations.

Client program • A simplified client for the *ShapeList* server is illustrated in Figure 5.20. Any client program needs to get started by using a binder to look up a remote object reference. Our client sets a security manager and then looks up a remote object reference for the remote object using the *lookup* operation of the RMIregistry (line 1). Having obtained an initial remote object reference, the client continues by sending RMIs to that remote object or to others discovered during its execution according to the needs of its application. In our example, the client invokes the method *allShapes* in the remote object (line 2) and receives a vector of remote object references to all of the shapes currently stored in the server. If the client was implementing a whiteboard display, it would use the server's *getAllState* method in the *Shape* interface to retrieve each of the graphical objects in the vector and display them in a window. Each time the user finishes

drawing a graphical object, it will invoke the method *newShape* in the server, passing the new graphical object as its argument. The client will keep a record of the latest version number at the server, and from time to time it will invoke *getVersion* at the server to find out whether any new shapes have been added by other users. If so, it will retrieve and display them.

Callbacks • The general idea behind callbacks is that instead of clients polling the server to find out whether some event has occurred, the server should inform its clients whenever that event occurs. The term *callback* is used to refer to a server's action of notifying clients about an event. Callbacks can be implemented in RMI as follows:

- The client creates a remote object that implements an interface that contains a method for the server to call. We refer to this as a *callback object*.
- The server provides an operation allowing interested clients to inform it of the remote object references of their callback objects. It records these in a list.
- Whenever an event of interest occurs, the server calls the interested clients. For example, the whiteboard server would call its clients whenever a graphical object is added.

The use of callbacks avoids the need for a client to poll the objects of interest in the server and its attendant disadvantages:

- The performance of the server may be degraded by the constant polling.
- Clients cannot notify users of updates in a timely manner.

However, callbacks have problems of their own. First, the server needs to have up-to-date lists of the clients' callback objects, but clients may not always inform the server before they exit, leaving the server with incorrect lists. The *leasing* technique discussed in Section 5.4.3 can be used to overcome this problem. The second problem associated with callbacks is that the server needs to make a series of synchronous RMIs to the callback objects in the list. See Chapter 6 for some ideas about solving the second problem.

We illustrate the use of callbacks in the context of the whiteboard application. The *WhiteboardCallback* interface could be defined as follows:

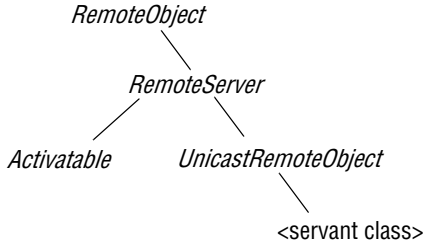
```
public interface WhiteboardCallback implements Remote {
    void callback(int version) throws RemoteException;
};
```

This interface is implemented as a remote object by the client, enabling the server to send the client a version number whenever a new object is added. But before the server can do this, the client needs to inform the server about its callback object. To make this possible, the *ShapeList* interface requires additional methods such as *register* and *deregister*, defined as follows:

```
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

After the client has obtained a reference to the remote object with the *ShapeList* interface (for example, in Figure 5.20, line 1) and created an instance of its callback object, it uses the *register* method of *ShapeList* to inform the server that it is interested in receiving

Figure 5.21 Classes supporting Java RMI



callbacks. The *register* method returns an integer (the *callbackId*) referring to the registration. When the client is finished it should call *deregister* to inform the server it no longer requires callbacks. The server is responsible for keeping a list of interested clients and notifying all of them each time its version number increases.

5.5.2 Design and implementation of Java RMI

The original Java RMI system used all of the components shown in Figure 5.15. But in Java 1.2, the reflection facilities were used to make a generic dispatcher and to avoid the need for skeletons. Prior to J2SE 5.0, the client proxies were generated by a compiler called *rmic* from the compiled server classes (not from the definitions of the remote interfaces). However, this step is no longer necessary with recent versions of J2SE, which contain support for the dynamic generation of stub classes at runtime.

Use of reflection • Reflection is used to pass information in request messages about the method to be invoked. This is achieved with the help of the class *Method* in the reflection package. Each instance of *Method* represents the characteristics of a particular method, including its class and the types of its arguments, return value and exceptions. The most interesting feature of this class is that an instance of *Method* can be invoked on an object of a suitable class by means of its *invoke* method. The *invoke* method requires two arguments: the first specifies the object to receive the invocation and the second is an array of *Object* containing the arguments. The result is returned as type *Object*.

To return to the use of the *Method* class in RMI: the proxy has to marshal information about a method and its arguments into the *request* message. For the method it marshals an object of class *Method*. It puts the arguments into an array of *Object*s and then marshals that array. The dispatcher unmarshals the *Method* object and its arguments in the array of *Object*s from the *request* message. As usual, the remote object reference of the target will have been unmarshalled and the corresponding local object reference obtained from the remote reference module. The dispatcher then calls the *Method* object's *invoke* method, supplying the target and the array of argument values. When the method has been executed, the dispatcher marshals the result or any exceptions into the *reply* message. Thus the dispatcher is generic – that is, the same dispatcher can be used for all classes of remote object, and no skeletons are required.

Java classes supporting RMI • Figure 5.21 shows the inheritance structure of the classes supporting Java RMI servers. The only class that the programmer need be aware of is *UnicastRemoteObject*, which every simple servant class needs to extend. The class *UnicastRemoteObject* extends an abstract class called *RemoteServer*, which provides

abstract versions of the methods required by remote servers. *UnicastRemoteObject* was the first example of *RemoteServer* to be provided. Another called *Activatable* is available for providing activatable objects. Further alternatives might provide for replicated objects. The class *RemoteServer* is a subclass of *RemoteObject* that has an instance variable holding the remote object reference and provides the following methods:

equals This method compares remote object references.

toString: This method gives the contents of the remote object reference as a *String*.

readObject, writeObject: These methods deserialize/serialize remote objects.

In addition, the *instanceOf* operator can be used to test remote objects.

5.6 Summary

This chapter has discussed three paradigms for distributed programming – request-reply protocols, remote procedure calls and remote method invocation. All of these paradigms provide mechanisms for distributed independent entities (processes, objects, components or services) to communicate directly with one another.

Request-reply protocols provide lightweight and minimal support for client-server computing. Such protocols are often used in environments where overheads of communication must be minimized − for example, in embedded systems. Their more common role is to support either RPC or RMI, as discussed below.

The remote procedure call approach was a significant breakthrough in distributed systems, providing higher-level support for programmers by extending the concept of a procedure call to operate in a networked environment. This provides important levels of transparency in distributed systems. However, due to their different failure and performance characteristics and to the possibility of concurrent access to servers, it is not necessarily a good idea to make remote procedure calls appear to be exactly the same as local calls. Remote procedure calls provide a range of invocation semantics, from *maybe* invocations through to *at-most-once* semantics.

The distributed object model is an extension of the local object model used in object-based programming languages. Encapsulated objects form useful components in a distributed system, since encapsulation makes them entirely responsible for managing their own state, and local invocation of methods can be extended to remote invocation. Each object in a distributed system has a remote object reference (a globally unique identifier) and a remote interface that specifies which of its operations can be invoked remotely.

Middleware implementations of RMI provide components (including proxies, skeletons and dispatchers) that hide the details of marshalling, message passing and locating remote objects from client and server programmers. These components can be generated by an interface compiler. Java RMI extends local invocation to remote invocation using the same syntax, but remote interfaces must be specified by extending an interface called *Remote* and making each method throw a *RemoteException*. This ensures that programmers know when they make remote invocations or implement remote objects, enabling them to handle errors or to design objects suitable for concurrent access.

EXERCISES

5.1 Define a class whose instances represent request and reply messages as illustrated in Figure 5.4. The class should provide a pair of constructors, one for request messages and the other for reply messages, showing how the request identifier is assigned. It should also provide a method to marshal itself into an array of bytes and to unmarshal an array of bytes into an instance. *page 188*

5.2 Program each of the three operations of the request-reply protocol in Figure 5.3, using UDP communication, but without adding any fault-tolerance measures. You should use the classes you defined in the previous chapter for remote object references (Exercise 4.13) and above for request and reply messages (Exercise 5.1). *page 187*

5.3 Give an outline of the server implementation, showing how the operations *getRequest* and *sendReply* are used by a server that creates a new thread to execute each client request. Indicate how the server will copy the *requestId* from the request message into the reply message and how it will obtain the client IP address and port. *page 187*

5.4 Define a new version of the *doOperation* method that sets a timeout on waiting for the reply message. After a timeout, it retransmits the request message *n* times. If there is still no reply, it informs the caller. *page 188*

5.5 Describe a scenario in which a client could receive a reply from an earlier call. *page 187*

5.6 Describe the ways in which the request-reply protocol masks the heterogeneity of operating systems and of computer networks. *page 187*

5.7 Discuss whether the following operations are *idempotent*:

i) pressing a lift (elevator) request button;

ii) writing data to a file;

iii) appending data to a file.

Is it a necessary condition for idempotence that the operation should not be associated with any state? *page 190*

5.8 Explain the design choices that are relevant to minimizing the amount of reply data held at a server. Compare the storage requirements when the RR and RRA protocols are used. *page 191*

5.9 Assume the RRA protocol is in use. How long should servers retain unacknowledged reply data? Should servers repeatedly send the reply in an attempt to receive an acknowledgement? *page 191*

5.10 Why might the number of messages exchanged in a protocol be more significant to performance than the total amount of data sent? Design a variant of the RRA protocol in which the acknowledgement is piggy-backed on − that is, transmitted in the same message as − the next request where appropriate, and otherwise sent as a separate message. (Hint: use an extra timer in the client.) *page 191*

5.11 An *Election* interface provides two remote methods:

> *vote*: This method has two parameters through which the client supplies the name of a candidate (a string) and the 'voter's number' (an integer used to ensure each user votes once only). The voter's numbers are allocated sparsely from the range of integers to make them hard to guess.
>
> *result*: This method has two parameters through which the server supplies the client with the name of a candidate and the number of votes for that candidate.

Which of the parameters of these two procedures are *input* and which are *output* parameters? *page 195*

5.12 Discuss the invocation semantics that can be achieved when the request-reply protocol is implemented over a TCP/IP connection, which guarantees that data is delivered in the order sent, without loss or duplication. Take into account all of the conditions causing a connection to be broken. *Section 4.2.4 and page 198*

5.13 Define the interface to the *Election* service in CORBA IDL and Java RMI. Note that CORBA IDL provides the type *long* for 32-bit integers. Compare the methods in the two languages for specifying *input* and *output* arguments. *Figure 5.8, Figure 5.16*

5.14 The *Election* service must ensure that a vote is recorded whenever any user thinks they have cast a vote.

Discuss the effect of *maybe* call semantics on the *Election* service.

Would *at-least-once* call semantics be acceptable for the *Election* service or would you recommend *at-most-once* call semantics? *page 199*

5.15 A request-reply protocol is implemented over a communication service with omission failures to provide *at-least-once* invocation semantics. In the first case the implementor assumes an asynchronous distributed system. In the second case the implementor assumes that the maximum time for the communication and the execution of a remote method is T. In what way does the latter assumption simplify the implementation? *page 198*

5.16 Outline an implementation for the *Election* service that ensures that its records remain consistent when it is accessed concurrently by multiple clients. *page 199*

5.17 Assume the *Election* service is implemented in RMI and must ensure that all votes are safely stored even when the server process crashes. Explain how this can be achieved with reference to the implementation outline in your answer to Exercise 5.16. *pages 213–214*

5.18 Show how to use Java reflection to construct the client proxy class for the *Election* interface. Give the details of the implementation of one of the methods in this class, which should call the method *doOperation* with the following signature:

byte[] doOperation (RemoteObjectRef o, Method m, byte[] arguments);

Hint: an instance variable of the proxy class should hold a remote object reference (see Exercise 4.13). *Figure 5.3, page 224*

5.19 Show how to generate a client proxy class using a language such as C++ that does not support reflection, for example from the CORBA interface definition given in your answer to Exercise 5.13. Give the details of the implementation of one of the methods in this class, which should call the method *doOperation* defined in Figure 5.3.

page 211

5.20 Explain how to use Java reflection to construct a generic dispatcher. Give Java code for a dispatcher whose signature is:

public void dispatch(Object target, Method aMethod, byte[] args)

The arguments supply the target object, the method to be invoked and the arguments for that method in an array of bytes. *page 224*

5.21 Exercise 5.18 required the client to convert *Object* arguments into an array of bytes before invoking *doOperation* and Exercise 5.20 required the dispatcher to convert an array of bytes into an array of *Object*s before invoking the method. Discuss the implementation of a new version of *doOperation* with the following signature:

Object[] doOperation (RemoteObjectRef o, Method m, Object[] arguments);

which uses the *ObjectOutputStream* and *ObjectInputStream* classes to stream the request and reply messages between client and server over a TCP connection. How would these changes affect the design of the dispatcher? *Section 4.3.2 and page 224*

5.22 A client makes remote method invocations to a server. The client takes 5 milliseconds to compute the arguments for each request, and the server takes 10 milliseconds to process each request. The local operating system processing time for each send or receive operation is 0.5 milliseconds, and the network time to transmit each request or reply message is 3 milliseconds. Marshalling or unmarshalling takes 0.5 milliseconds per message.

Calculate the time taken by the client to generate and return from two requests:

(i) if it is single-threaded;

(ii) if it has two threads that can make requests concurrently on a single processor.

You can ignore context-switching times. Is there a need for asynchronous invocation if the client and server processes are threaded? *page 213*

5.23 Design a remote object table that can support distributed garbage collection as well as translating between local and remote object references. Give an example involving several remote objects and proxies at various sites to illustrate the use of the table. Show the changes in the table when an invocation causes a new proxy to be created. Then show the changes in the table when one of the proxies becomes unreachable. *page 215*

5.24 A simpler version of the distributed garbage collection algorithm described in Section 5.4.3 just invokes *addRef* at the site where a remote object lives whenever a proxy is created and *removeRef* whenever a proxy is deleted. Outline all the possible effects of communication and process failures on the algorithm. Suggest how to overcome each of these effects, but without using leases. *page 215*