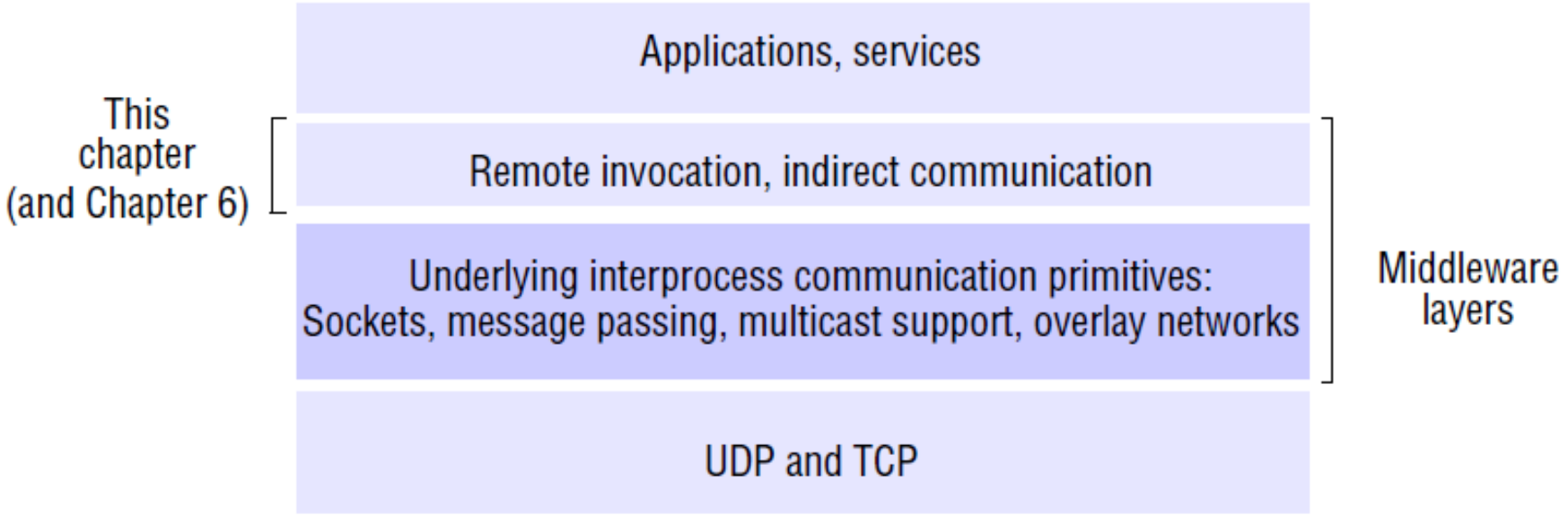
Remote Invocation

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.



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

- ***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.

Request-reply protocols

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:

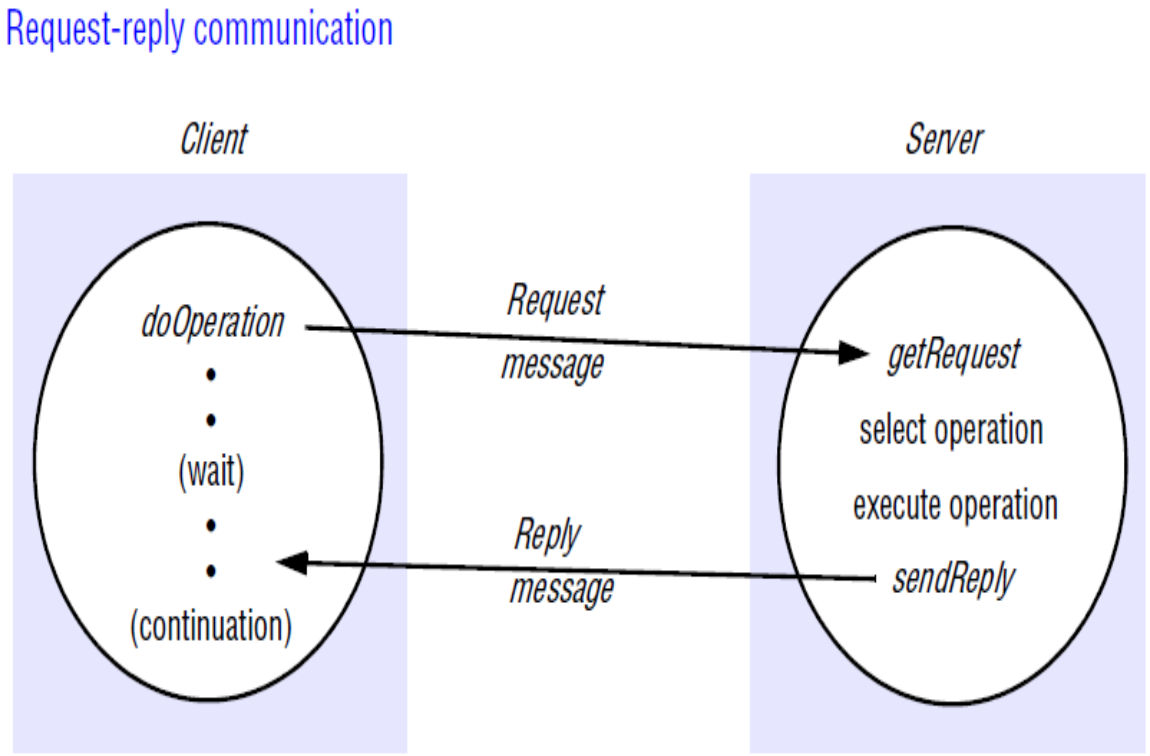
• Acknowledgements are redundant, since requests are followed by replies.

• 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.



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.

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

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, 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

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

Request-reply message structure

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

•Reflection is an API that is used to examine or modify the behavior of methods, classes, and interfaces at runtime. The required classes for reflection are provided under **java.lang.reflect** package. Reflection gives us information about the class to which an object belongs and also the methods of that class that can be executed by using the object.

•Through reflection, we can invoke methods at runtime irrespective of the access specifier used with them.

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.

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.

Omission failure happens when a crucial requirement or feature is accidentally or intentionally omitted from the design.

In addition, the protocol can suffer from the failure of processes

. We assume that processes have crash failures. That is, when they halt, they remain halted – they do not produce Byzantine behaviour

A Byzantine fault is **a state of a computer system, particularly distributed computing systems, where components may fail and there is imperfect information on whether a component has failed**.

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.
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.

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.

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

RPC exchange protocols

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

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
The RRA protocol is based on the exchange of three messages: request-replyacknowledge 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

Use of TCP streams to implement the request-reply protocol

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.

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 , 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

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.
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 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 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 tim

The GET Method

GET is used to request data from a specified resource.

Note that the query string (name/value pairs) is sent in the URL of a GET request:

```
/test/demo_form.php?name1=value1&name2=value2
```

Some notes on GET requests:

- GET requests can be cached
- GET requests remain in the browser history
- GET requests can be bookmarked
- GET requests should never be used when dealing with sensitive data
- GET requests have length restrictions
- GET requests are only used to request data (not modify)

The POST Method

POST is used to send data to a server to create/update a resource.

The data sent to the server with POST is stored in the request body of the HTTP request:

```
POST /test/demo_form.php HTTP/1.1
Host: w3schools.com

name1=value1&name2=value2
```

Some notes on POST requests:

- POST requests are never cached
- POST requests do not remain in the browser history
- POST requests cannot be bookmarked
- POST requests have no restrictions on data length

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.
The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result.

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.

With an idempotent HTTP method, **multiple invocations always leave the data on the server in the same state**.

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

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

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. 200 OK

Standard response for successful HTTP requests.

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

Remote Procedure Call

Remote Procedure Call (RPC) is a protocol that provides the high-level communications paradigm used in the operating system. RPC presumes the existence of a low-level transport protocol, such as Transmission Control Protocol/Internet Protocol (TCP/IP) or User Datagram Protocol (UDP), for carrying the message data between communicating programs. RPC implements a logical client-to-server communications system designed specifically for the support of network applications.

The RPC protocol is built on top of the eXternal Data Representation (XDR) protocol, which standardizes the representation of data in remote communications.

XDR converts the parameters and results of each RPC service provided.

The RPC protocol enables users to work with remote procedures as if the procedures were local.

The remote procedure calls are defined through routines contained in the RPC protocol.

Each call message is matched with a reply message.

A *client* is a computer or process that accesses the services or resources of another process or computer on the network.
A *server* is a computer that provides services and resources, and that implements network services.
Each network *service* is a collection of remote programs. A remote program implements remote procedures.
The procedures, their parameters, and the results are all documented in the specific program's protocol.
RPC provides an authentication process that identifies the server and client to each other.
RPC includes a slot for the authentication parameters on every remote procedure call so that the caller can identify itself to the server.
The client package generates and returns authentication parameters.
RPC supports various types of authentication such as the UNIX and Data Encryption Standard (DES) systems.

In RPC, each server supplies a program that is a set of remote service procedures. The combination of a host address, program number, and procedure number specifies one remote service procedure.

In the RPC model, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, and sends a reply back to the client.

The procedure call then returns to the client.

The RPC interface is generally used to communicate between processes on different workstations in a network.

However, RPC works just as well for communication between different processes on the same workstation.

To write network applications using RPC, programmers need a working knowledge of network theory.

For most applications, an understanding of the RPC mechanisms usually hidden by the rpcgen command's protocol compiler is also helpful. However, use of the **rpcgen** command circumvents the need for understanding the details of RPC.

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**.

Design issues for RPC

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 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
- 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

- 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.

Interface definition languages

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.

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.

```
// 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();
};
```

RPC call semantics • Request-reply protocols was discussed , 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.

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.

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

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!

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.

Difference between atleast-once and atmost-once

At most once: Messages are delivered once, and if there is a system failure, messages may be lost and are not redelivered. **At least once: This means messages are delivered one or more times**. If there is a system failure, messages are never lost, but they may be delivered more than once.

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, 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.

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.

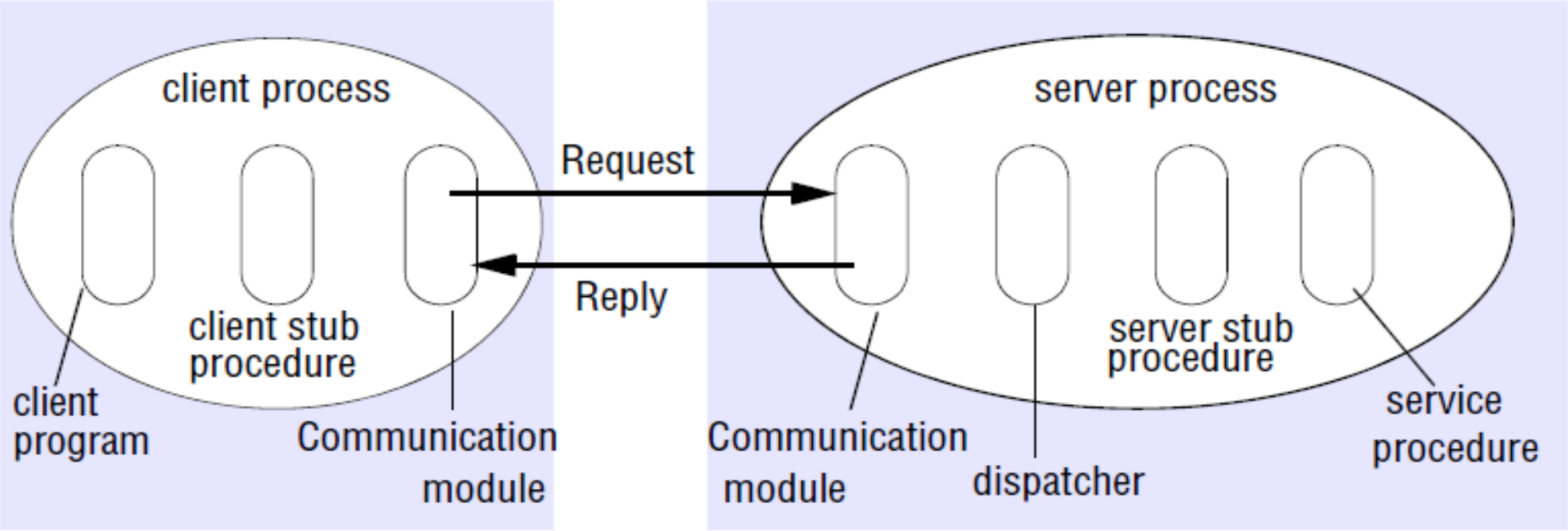
When the reply message arrives, it unmarshals the results.

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 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. A*t-leastonce* or *at-most-once* is generally chosen.

Role of client and server stub procedures in RPC



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

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;
```

Program Definition: The program FILEREADWRITE is defined with the version VERSION. This program seems to have two procedures: WRITE and READ.

Version Definition: The version VERSION of the program defines two procedures:

- *WRITE: This procedure takes writeargs as input and has a procedure number of 1.*
- *READ: This procedure takes readargs as input and has a procedure number of 2.*

Program Identifier and Version Number: The program identifier FILEREADWRITE with version VERSION is assigned a program number 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.

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.

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

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

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.

Remote Method Invocation

Remote Method Invocation (RMI) enables programmers to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap naming service provided by RMI or by receiving the reference as an argument or a return value.

A client can call a remote object in a server, and that server can also be a client of other remote objects.

RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, thereby supporting true object-oriented polymorphism.

Design issues for RMI

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.

```
class Car {
    String brand;
    Car(String brand) {
        this.brand = brand;
    }
    void display() {
        System.out.println("Brand: " + brand);
    }
}
public class Main {
    public static void main(String[] args) {
        // Create two Car objects
        Car car1 = new Car("Toyota");
        Car car2 = car1; // Both car1 and car2 now refer to the same Car object
        // Modify car2
        car2.brand = "Honda";
        // Display details of both cars
        System.out.print("Car 1: ");
        car1.display(); // Output: Car 1: Brand: Honda
        System.out.print("Car 2: ");
        car2.display(); // Output: Car 2: Brand: Honda
    }
}
```

When we display the details of both cars, we see that both car1 and car2 have the same brand ("Honda"), because they refer to the same object in memory.

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.

```
// Define an interface
interface Animal {
    void sound(); // Abstract method (does not have a body)
}
// Implement the interface
class Dog implements Animal {
    // Provide implementation for the sound method
    public void sound() {
        System.out.println("Woof");
    }
}
```

```
// Implement the interface
class Cat implements Animal {
    // Provide implementation for the sound method
    public void sound() {
        System.out.println("Meow");
    }
}
public class Main {
    public static void main(String[] args) {
        // Create objects of Dog and Cat classes
        Dog dog = new Dog();
        Cat cat = new Cat();

        // Call the sound method on both objects
        dog.sound(); // Output: Woof
        cat.sound(); // Output: Meow
    }
}
```

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.

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

```
public class Main {
    public static void main(String[] args) {
        try {
            // Call a method that may throw an exception
            divideByZero();
        } catch (ArithmeticException e) {
            // Catch and handle the exception
            System.out.println("Error: " + e.getMessage());
        }
    }

    // Method that throws an ArithmeticException
    public static void divideByZero() {
        int dividend = 10;
        int divisor = 0;

        // Attempt to divide by zero
        int result = dividend / divisor;
    }
}
```

In Java, the getMessage() method is a method of the Throwable class, which is the superclass of all classes that represent errors or exceptions.
It returns a detailed message string describing the cause of the exception.

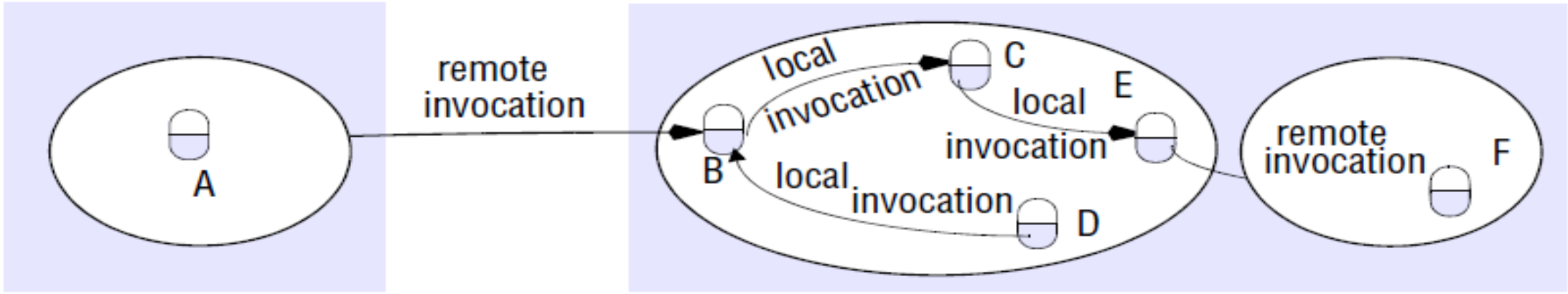
Garbage collection: It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed.
A language such as Java, that can detect automatically when an object is no longer accessible recovers the space and makes it available for allocation to other objects.
This process is called *garbage collection*.

Distributed objects

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.
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.
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.

Remote and local method invocations



The distributed object model

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* 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.

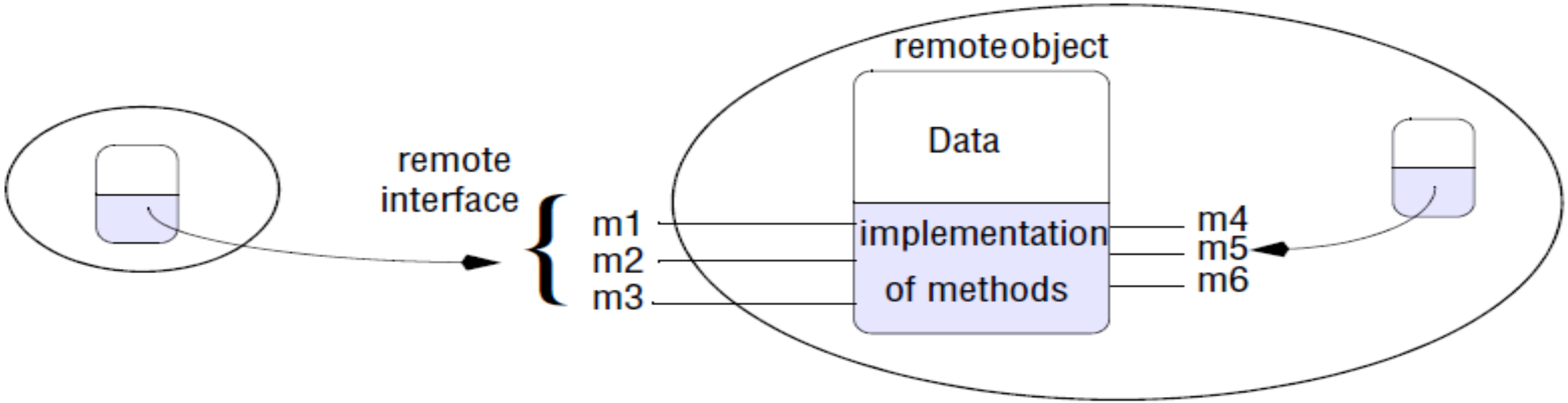
In RMI, a *remote* interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine.

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.

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.

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.

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.

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.

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

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.

Case study: Java RMI

Java 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

```
import java.rmi.Remote;
import java.rmi.RemoteException;

// Remote interface
public interface Hello extends Remote {
    // Remote method declaration
    String sayHello() throws RemoteException;
}
```

This interface defines the remote method sayHello(), which will be implemented by the server and invoked by the client.
It extends the Remote interface, indicating that it is a remote interface, and each method declares RemoteException in its throws clause to handle remote communication failures.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
// Remote implementation class
public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException {
        super();
    }
    // Implementation of the remote method
    public String sayHello() throws RemoteException {
        return "Hello, World!";
    }
}
```

In Java, super() is used to call the constructor of the superclass (the parent class) from within a subclass (the child class). It is typically used when the subclass constructor needs to initialize the inherited members of the superclass.

This class implements the Hello interface and provides the implementation for the sayHello() method.

It extends UnicastRemoteObject to export the remote object and make it available to receive remote method invocations.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Server {
    public static void main(String[] args) {
        try {
            // Create and export the remote object
            Hello hello = new HelloImpl();
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.rebind("HelloService", hello);
            System.out.println("Server started.");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

This class represents the **RMI server**. It creates an instance of HelloImpl, binds it to the RMI registry using rebind(), and starts the RMI registry on port 1099 using LocateRegistry.createRegistry(). The server is now ready to accept incoming RMI calls.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
    public static void main(String[] args) {
        try {
            // Lookup the remote object
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);
            Hello hello = (Hello) registry.lookup("HelloService");
            // Invoke the remote method
            String message = hello.sayHello();
            System.out.println("Message from server: " + message);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

This class represents the **RMI client**. It looks up the remote object HelloService in the RMI registry using registry.lookup(), obtains a stub for the remote object, and then invokes the sayHello() method on the stub. Finally, it prints the message received from the server.

Running the Example:

Compile all Java files: javac *.java
Start the RMI registry: start rmiregistry
Start the server: java Server
Run the client: java Client

Stub Generation: When the server is started, it exports the remote object (HelloImpl) using UnicastRemoteObject.exportObject(). This process generates a stub for the remote object.

Client Lookup: In the client code, the LocateRegistry.getRegistry() method is used to obtain a reference to the RMI registry running on the server host and port. Then, the registry.lookup() method is called to look up the remote object registered with the RMI registry.

Stub Invocation: The lookup() method returns a reference to the stub (proxy) object for the remote object. This stub implements the same interface (Hello) as the remote object (HelloImpl), including the sayHello() method. However, the actual implementation of the sayHello() method resides on the server.

Method Invocation: The client invokes the sayHello() method on the stub object as if it were a local method call. Internally, the stub marshals the method call and parameters into a network message and sends it to the server over the network.

Server Invocation: On the server side, the RMI runtime receives the network message containing the method call. It routes the message to the appropriate server object (HelloImpl) and invokes the sayHello() method on the server object.

Response Marshalling: The server executes the sayHello() method and returns the result (the "Hello, World!" message) to the RMI runtime.

Response Transmission: The RMI runtime marshals the return value into a network message and sends it back to the client over the network.

Client Invocation: The client receives the response from the server, unmarshals it, and returns the result to the caller as if it were a local method call.