Lecture 5
Interprocess Communications
(June-22,23,24,2009. Chapter 4, 5)

Lecture Outline

1. Introduction
2. TCP/IP API and Java UDP/TCP API
3. External data representation and marshalling
4. Client-server communication
5. Remote procedure call
6. Distributed objects and RMI (remote method invocation)
7. Group communications

1. Introduction

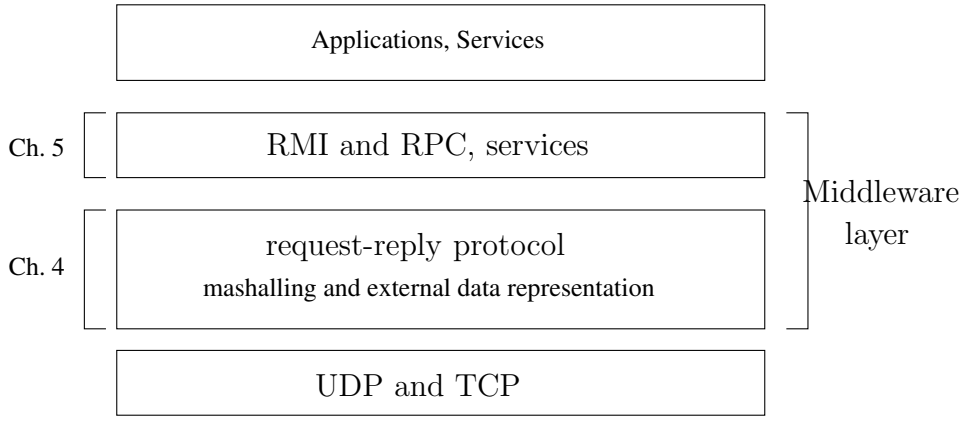

Figure 26: Middleware layers (Fig.4.1, p.132)

(1) Facilities for interprocess communications are essential for DSs.

(2) Facilities for interprocess communications can be classified in terms of abstraction levels:

   a. Basic TCP/IP APIs: *Berkeley sockets* and *SVR4 TLIs*.

   b. RMI (remote method invocation) supported by certain object oriented programming languages or development tools such as $C++$, Java, and CORBA.

   c. Remote procedure calls: SUN RPC, OSF DCE (distributed computing environment) RPC

(3) Characteristics of IPC

a. IPC involves message exchanges among processes

b. Messages

   (a) A message consists of two parts: the data part, provided by the user, and the control part, attached by the system.

   (b) Messages can be structured or unstructured

   (c) Basic message passing primitives

      i. **send** *expression-list* **to** *destination-identifier*

      ii. **receive** *variable-list* **from** *source-identifier*

c. Synchronous (blocking) and asynchronous (non-blocking) communications.

   (a) Two queues, a *send queue* and a *receive queue* may be associated with each process.

   (b) The action of sending a message causes the message to be transmitted and eventually to be queued to the receive message queue of destination process.

   (c) A communication is *synchronous* if the the sender and receiver synchronize at every message. Both *send* and *receive* are block operations.

      i. A *send* operation will block the issuing process until the corresponding *receive* operation is performed at the receiver.

      ii. A *receive* operation will block the issuing process until the a message from the specified destination arrives.

   (d) A communication is *asynchronous* if the *send* operation will not block the issuing process: the process can proceed as soon as the the message to be transmitted is copied to a local buffer. The *receive* operation can be either blocking or non-blocking.

      i. The blocking receive operation behaves the same as in synchronous comm.

      ii. The non-blocking receive, if a desired message hasn't arrived yet, after a receiving process issues a *receive* operation, the process will continue to proceed with its other operations. The system will fill a buffer (provided by the receiving process in the receive operation) later and notify the process after that.

   (e) Discussions

      i. Non-blocking communication is more flexible and efficient. However implementation of receiving processes is much harder. Buffers *may be* needed.

      ii. Blocking communications are just the opposite.

      iii. For the non-blocking receiving variant of the non-blocking communication, an extra notification is needed, which interrupts the running process. Few DSs adopts this variant.

      iv. Parallelism may be lost in blocking communications.

v. In systems where support of threads is available, *threads* can be used in synchronous communications to enhance performance.

(4) Quality of IPC

a. *Reliability*. An IPC facility that delivers messages un-corrupted and un-duplicated is *reliable*.

b. *Ordering*. Messages sent by a sending process may or may not arrive same as their sending order.

c. *Usability*. An IPC facility is *highly usable* it supports high degree of *location transparency*.

d. *Timeness and efficiency*. This demands quick and efficient delivery of messages.

e. Notes:

(a) Quality of IPC facilities is closely related to the quality of services at the transport layer.

(b) IPC facilities can improve their service qualities on top of services by the transport layer.

2. TCP/IP API and Java UDP/TCP API

(1) Identification of message destinations

a. Destinations of messages are identified by the *IP address* and *port number*: (*remote IP address, remote port*).

(a) In most cases a message also has to carry (*local IP address, local port*) to allow a remote destination to send replies.

(b) A process may use multiple ports to send receive messages simultaneously.

(c) Port numbers of services should be publicized by servers to clients.

b. Other means of destination identification. The use of a (*IP address, port*) pair voids the location transparency. Several other methods support location transparency:

(a) A *binder* may be supported in some DSs (cf. Sect.5.2.5). A binder is a *service resolver*: given the name of service, it will return the server locations.

(b) Some OSs such Mach provide location independent identifiers for message destinations, translating identifiers into lower-level addresses such as (IP address, port).

(2) The Berkeley socket API

a. The *socket* API: a socket is an end point of a communication. *socket* API originated from Berkeley UNIX and are now available on Windows and Macintosh OSs.

b. Socket supports both reliable communications (connection oriented) through TCP API and un-reliable comm. (connectionless) through UDP API.

c. Illustration of *ports* and *sockets*: Fig.4.2, p.135.

d. UDP datagram communications

   (a) Characteristics

      i. UDP is a connectionless transport layer comm. protocol. It provides unreliable datagram delivery service.

      ii. *Message sizes*: a message can be up to 2^{16} bytes (however, many implementations impose a smaller size 8 kbytes). A receiving process has to specify the maximum size of messages to be received.

      iii. UDP employs *non-blocking send* and *blocking receive*.

      iv. UDP relies IP to deliver datagrams. It does not handle timeouts by itself. Applications should handle timeout according to their own requirements.

      v. UDP API provides several different primitives (functions) to send and receive messages. Several pairs of them allow *receive from any* semantics: they do not specify an origin of messages to be received through a socket. Any messages can be delivered and the system will fill the (IP address, port) pair of the sending process.

   (b) Major UDP API functions:

      i. *socket, bind* – for comm. management.

      ii. *sendto, recvfrom, send, recv, read, write, sendv, writev, sendmsg, recvmsg* – for sending and receiving messages.

   (c) Failure model:

      i. Cf. Lecture 3 for failure model: it defines the ways in which failures may occur. Two properties of reliable communications: *validity* (messages are eventually delivered) and *integrity* (corrupted messages and duplicate messages can be detected).

      ii. UDP provides an optional checksum that lets a receiving application verify if the contents of a datagram is corrupted. Notice checksum may not guarantee all possible corruptions to be found.

      iii. *Omission failures*: messages (i.e. UDP datagrams) may not arrive at their destinations, validity is not met.

      iv. *Ordering*: as individual datagrams may be delayed, they could arrive out of sender order. Since in general UDP does not do automatic retransmission, duplicate datagrams will not appear.

UDP applications that require reliable communications must provide their own acknowledgments, timeouts, and sequence numbers.

e. TCP stream communications.

(a) Characteristics

i. TCP is a connection-oriented transport layer comm. protocol. It provides reliable message delivery service.

ii. *Message sizes*: applications can decide the number of bytes to write to a stream or read from it. The underlying implementation maintains local write buffers and read buffers. The buffer sizes determines how much data to collect from a application before actually transmitting it, or how much data to accumulate from an incoming stream before delivering to an application. Applications can force data to be transmitted immediately.

iii. *Lost messages*: TCP employs the well-known *slide window* protocol to handle message transmissions. Messages are acknowledged and lost messages are retransmitted. In terms of slide window protocol concept, each byte is counted as a TCP packet. For example, when a client declares to a server window size $8k$, that means the server can send $8k$ bytes continuously before having to wait for acks from the client.

iv. *Flow control*: TCP attempts to *synchronize* the transmission speeds of a client and a server involved in a comm. in the sense that if one of them is slow, TCP will try to inform the other to slow down.

v. *Message duplication and ordering*: use of slide window protocol by TCP ensures that duplicated messages will be detected and discarded. In addition, messages delivering order will be the same as they are sent.

(b) Major TCP API functions:

i. *socket, bind, listen, connect, accept, close* – for comm. management.

ii. *read, write, readv, sendv* – for sending and receiving messages.

(c) Failure model:

i. TCP employs checksums, timeouts on individual messages, and retransmissions, in attempt to provide reliable communications.

ii. TCP also uses an overall timeout for a connection. If TCP cannot communicate with its peer for a prolonged period, it will assume that connection is broken. Hence, TCP *does not* provide reliable communications under our failure model, because it cannot guarantee delivery of messages in the faces of all possible problems.

iii. When a TCP connection is broken, the sender process does not know if its messages have been delivered; and both the sender and receiver processes do not know whether disconnection is caused by a network failure or failure of its peer.

(3) Java UDP and TCP API

a. The package that provides TCP/IP API in Java is *java.net*.

(a) Java TCP/IP API (called networking API in java) is implemented using sockets. It can be regarded as on a little higher level than the original socket API. For instance, programmers do not have to know the socket structures and applications do not have to explicitly initialize individual fields of such structures.

(b) For both UDP and TCP, Java TCP/IP API provides a sequence of methods corresponding to those in original socket API.

b. Java UDP API

(a) A Java UDP datagram packet: consisting four fields (p.138):

array of bytes containing messages	length of message	Internet address	port number

(b) An Java UDP example (Fig.4.3, p.138, Fig.4.4, p.139): a client sends a char string to a server, which sends it back as reply. The client is invoked with syntax

java UDPClient char_string server_IP_name

(c) The UDP client code (Fig.4.3, p.138)

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
    // args give message contents and server hostname
      DatagramSocket aSocket = null;
      try {
         aSocket = new DatagramSocket();
         byte [] m = args[0].getBytes();
         InetAddress aHost = InetAddress.getByName(args[1]);
         int serverPort = 6789;
         DatagramPacket request = new
              DatagramPacket(m, args[0].length(), aHost, serverPort);
         aSocket.send(request);
         byte[] buffer = new byte[1000];
         DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
         aSocket.receive(reply);
         System.out.println("Reply: " + new String(reply.getData()));
      }catch (SocketException e)
                {System.out.println("Socket: " + e.getMessage());
      }catch (IOException e)
                {System.out.println("IO: " + e.getMessage());}
      }finally { if (aSocket != null) aSocket.close();}
   }
}
```

(d) Major Java UDP API classes and methods:

i. Class *DatagramSocket*: represents a datagram socket. There are three constructors for this class:

(i) *DatagramSocket()*;

(ii) *DatagramSocket(int port)*;

(iii) *DatagramSocket(int port, InetAddress laddr)*.

ii. Major methods in class *DatagramSocket*:

(i) *void close()* closes the datagram socket.

(ii) *void send(DatagramPacket p)*: sends a datagram packet through this socket.

(iii) *void receive(DatagramPacket p)*: receives a datagram packet from this socket;

(iv) *void setSoTimeout(int timeout p)*: enable/disable SO_TIMEOUT with the specified timeout in units of milliseconds. A zero value means infinite timeout (disabled). When a timeout occurs, a *java.io.InterruptedIOException* is thrown;

iii. class *DatagramPacket*: represents a datagram packet. There are four constructors for this class:

(i) *DatagramPacket(byte[] buf, int length)*: constructs a DatagramPacket for receiving packets of length *length*.

(ii) *DatagramPacket(byte[] buf, int length, InetAddress address, int port)*: constructs a DatagramPacket for sending to the specified port number on the specified host.

(iii) *DatagramPacket(byte[] buf, int offset, int length)*: constructs a DatagramPacket for receiving packets of length *length*, specifying an offset into the buffer.

(iv) *DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)*: this method combines method 2 and 3 above.

iv. Major methods in class *DatagramPacket*:

(i) *byte[] getData()*: returns the data in bytes from the packet.

(ii) *int getLength()*: returns the length of the data to be sent or length of data received.

The data and length fields are illustrated by the structure of an Java UDP packet on p.138.

(e) Other components in the class.

i. The *getByName* method in class *InetAddress* takes an IPv4 address or name as argument and return the IPv4 address as return value (a NULL argument will return the local computer's IP address).

ii. A *try* block encloses a section of code that will be monitored for exceptions that the corresponding *catch* clauses will try to catch.

iii. An *IOException* object is thrown by both the client and server programs when there is an error caused by failed or interrupted I/O operations.

iv. Similarly a *SocketException* object is thrown when there is any error in the underlying protocol of the corresponding socket. *IOException* class is a superclass of *SocketException* class. Therefore if a program does not catch a SocketException, an IOException will be raised.

(f) The UDP server code (Fig.4.4,p.139)

```
import java.net.*;
import java.io.*;
public class UDPServer{
   public static void main(String args[]){
      DatagramSocket aSocket = null;
      try{
         aSocket = new DatagramSocket(6789);
         // create socket at agreed port
         byte[] buffer = new byte[1000];
         while(true){
            DatagramPacket request = new DatagramPacket(buffer,
                  buffer.length);
            aSocket.receive(request);
            DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(),
                    request.getPort());
            aSocket.send(reply);
         }
      }catch (SocketException e)
                {System.out.println("Socket: " + e.getMessage());
      }catch (IOException e)
                {System.out.println("IO: " + e.getMessage());}
      }finally { if (aSocket != null) aSocket.close();}
  }
}
```

c. Java TCP API

   (a) Unlike Java UDP, Java TCP will not create datagram packets for sending or preparing a datagram packet buffer for receiving. Java TCP creates *DataInputStream* and *DataOutputStream* for receiving and sending stream of bytes.

   (b) An Java TCP example (Fig.4.5, p.142, Fig.4.6, p.143):

      i. The client

      (i) It is invoked through a command line like:

      java TCPClient char_string server_IP_name

      (ii) The client creates a *communication socket* by calling a constructor of the *Socket* class.

      (iii) The client opens a *DataInputStream* for reading and a *DataOutputStream* for writing.

      (iv) The client sends the first command-line parameter to the server by calling the *writeUTF* method of the DataOutputStream. It attempts

to receive reply from the server by calling the *readUTF* method of the DataInputStream.

ii. The server

(i) The server calls *ServerSocket* class to create a server listen socket instance. It then enters *listen* mode and calls the *accept* method of the listen socket.

(ii) When a new connection request arrives, the server creates a new thread by calling the application defined *Connection* class, passing a client socket obtained from return of the accept method.

(iii) The constructor of the Connection class creates two data streams using the socket passed by the main process. It then calls *this.start()* method to put the thread in the *ready* state (when a thread is created, it is initially in *born* state).

(iv) The thread just reads a line of input from the client and sends it back to the client. It then terminates by calling the *close* method of the sockets it back to the client. It then terminates by calling the *close* method of the socket.

(c) The TCP client code (Fig.4.5, p.142)

```
import java.net.*;
import java.io.*;
public class TCPClient {
   public static void main (String args[]) {
   // arguments supply message and hostname
      Socket s = null;
      try{
         int serverPort = 7896;
         s = new Socket(args[1], serverPort);
         DataInputStream in = new
                     DataInputStream(s.getInputStream());
         DataOutputStream out = new
                     DataOutputStream(s.getOutputStream());
         out.writeUTF(args[0]);  // UTF is a string encoding (cf.4.3)
         String data = in.readUTF();
         System.out.println("Received: "+ data);
      }catch (UnknownHostException e){
                System.out.println("Socket:"+e.getMessage());
      }catch (EOFException e)
                {System.out.println("EOF:"+e.getMessage());
      }catch (IOException e)
                {System.out.println("readline:"+e.getMessage());}
      } finally {if (s!=null) try {
```

```
            s.close();} catch (IOException e){/*close failed */}}
   }
```

(d) The TCP server code (Fig.4.6, p.143)

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
      try{
         int serverPort = 7896; // the server port
         ServerSocket listenSocket = new ServerSocket(serverPort);
         while(true) {
            Socket clientSocket = listenSocket.accept();
            Connection c = new Connection(clientSocket);
         }
      } catch(IOException e)
             {System.out.println("Listen:"+e.getMessage());}
   }
}

class Connection extends Thread {
      DataInputStream in;
      DataOutputStream out;
      Socket clientSocket;
      public Connection (Socket aClientSocket) {
         try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream());
            out = new DataOutputStream(clientSocket.getOutputStream());
            this.start();
          } catch(IOException e)
                {System.out.println("Connection:"+e.getMessage());}
      }
      public void run(){
         try {       // an echo server
            String data = in.readUTF();
            // read a line of data from the stream
            out.writeUTF(data);
         }catch (EOFException e)
                   {System.out.println("EOF:"+e.getMessage());
         }catch (IOException e)
                   {System.out.println("IO:"+e.getMessage());}
         } finally { try { clientSocket.close();}
                         catch (IOException e) {/* close failed */}}
```

```
        }
    }
```

3. External data representation and marshalling

(1) The need of external data representation

a. The need of EDR

(a) Different programming languages support different primitive data types and data structures. Even for the same data type, such as floating point numbers, two different languages on two different platform may implement them differently.

(b) The set of codes used to present characters can be different according to platforms. UNIX systems use ASCII coding and take one byte per character. UTF-8 (universal transfer format) code takes two bytes per character. For integers, there are also the so called *big-endian* (most significant byte first) verse *little-endian* issue.

(c) These different primitive data types and data structures, plus potential different set of codes, have to be communicated through messages.

b. Two approaches to handle the above diversities

(a) The values are converted to an agreed external format before transmission and converted to the local format upon receipt.

(b) The values are transmitted in the sender's format together with an indication of the format used, and the recipient converts the values if necessary.

The first approach is commonly used.

c. *External data representation*: an agreed standard for the representation of primitive data types and other data structures.

d. *Marshalling* and *unmarshalling*:

(a) *Marshalling* is the action of taking a collection of data items to be used in a communication and transform them into items in a form suitable for transmission in a message.

(b) *Unmarshalling* is the opposite action of *marshalling*. It is the action that takes a received message, extract individual data items from the message, and produce equivalent data items that can be used by the receiving process.

Clearly marshalling and unmarshalling are based on EDR.

e. EDRs and marshalling discussed here:

(a) SUN RPC XDR

(b) DCE RPC interface

(c) CORBA's common data representation

(d) JAVA RMI

(e) JAVA IDL

(2) SUN RPC

a. SUN RPC's XDR (External Data Representation) provides a uniform type conversion mechanism. SUN RPC can handle arbitrary data types/structures, regardless of different machines' byte order and structure layout conventions, by always converting them to a network standard called *External Data Representation* (XDR) before sending them over the network.

b. *Serializing and deserializing*: In SUN RPC's terms, the action of converting from a particular machine representation to XDR format is called *serializing*. The reverse action is called *deserializing*.

c. SUN RPC provides three level interfaces to applications. XDR is used at the intermediate and low levels.

d. *Marshalling* (i.e. *serializing*) in SUN RPC:

(a) Each data type that is allowed to be transmitted in an RPC call must have a corresponding function (called an *XDR filter*) defined for it. The XDR filter handles the serializing and deserializing of that data type.

(b) XDR filters are written in XDR language, which is similar to C language in syntax and semantics. The XDR compiler *rpcgen* compiles an XDR file (each ends with suffix *xdr*) into C files, which can be compiled and linked with clients and servers.

(c) XDR filters can be built in a structured fashion. SUN RPC provides 11 built-in type XDR filters: xdr_int(), xdr_u_int(), xdr_long(), ... Users can define an XDR filter based on these built-in XDR filters or any existing XDR filters.

e. Example: For an application defined structure:

```
struct simple {
   int a;
   short b;
} simple;
```

then you would call callrpc() as

callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple, &simple ...);

where xdr_simple() is written as:

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)

XDR *xdrsp; struct simple *simplep; {

   if (!xdr_int(xdrsp, &simplep->a))
      return (0);
   if (!xdr_short(xdrsp, &simplep->b))
      return (0);
   return (1);
}
```

(3) DCE RPC

a. DCE RPC applications must first define their interfaces which are written in the IDL (interface definition language) language. An IDL file ends with suffix .idl and has syntax similar to the C language.

b. An IDL interface definition defines all data types that can be used by both client and server and all remote functions/procedures available as services. Each remote function/procedure declared in the interface contains complete specification of their arguments, their orderings, and return values.

c. All servers that support the interface must implement the remote function/procedures using the same function/procedure names, data types, arguments, and ordering of arguments. All clients must call a remote function/procedure consistent with its declaration in the interface.

d. *Marshalling* in DCE RPC:

   (a) Compilation of an interface file produces a C header file, a client stub file in C, and a server stub file in C. The C header file produced contains for each remote function/procedure declared in the IDL file its corresponding C declaration.

   (b) The client stub file performs all marshalling/unmarshalling actions for the client. It should be compiled and linked with other client files (such as the file that contains the client's main function) to produce an executable client program.

   (c) Similar to the server stub file.

   The DCE IDL compiler is supported by the IDL library in producing client and server stubs. Thus IDL compiler provides transparency in EDR and EDR marshalling.

(4) CORBA's common data representation (CDR)

a. CORBA CDR is an EDR.

(a) It defines 15 primitive data types (such as *short* (16-bit), *long* (32-b), *unsigned short*, *unsigned long*, *float* (32-b), *double* (64-b), *char*, *boolean*, *octet* (8-b), and *any* which can represent any basic or constructed data type.

(b) It also defines 6 composite data types (Fig.4.7, p.146) that are commonly used in programming languages:

Type	Representation
sequence	length(unsigned long) followed by elements in order
string	length(unsigned long) followed by characters in order (can also have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declarations of the components
enumerated	unsigned long(the values are specified by the order declared)
union	type tag followed by the selected member

Figure 4.7 CORBA CDR for constructed types

(c) For primitive data types, CDR supports both big-endian and little-endian orderings. The specific ordering of each value is attached in the containing message.

(d) A value of the composite type will be represented as a sequence of bytes according to the ordering as specified by Fig.4.7, and the ordering of its underline primitive data types.

b. *Marshalling* in CDR.

(1) Like in DCE RPC, CORBA supports its own IDL. Interfaces written in CORBA IDL can be compiled by its IDL compiler to produce client and server stubs that will perform marshalling and unmarshalling automatically with no need of interferences from users.

(2) For a structure defined by

```
struct Person {
   string name;
   string place;
   long year;
};
```

CORBA IDL will flatten a Person struct with value: {'Smith', 'London', 1934} to a sequence of $4 \times 7 = 28$ bytes (Fig.4.8, p.147):

index in sequence of bytes	← 4 bytes →	*notes on representation*
0-3	5	*length of string*
4-7	"Smit"	'Smith'
8-11	"h___"	
12-15	6	*length of string*
16-19	"Lond"	'London'
20-23	"on__"	
24-27	1934	*unsigned long*

Figure 4.8 CORBA CDR message

(5) Java RMI (remote method invocation, since JDK1.1)

a. Basics

(a) Java RMI is Java's implementation of RPC for Java-object-to-Java-object distributed communications.

(b) A method of a Java object can be registered through RMI and will become remotely accessible by other Java methods thereafter.

i. The syntax of invoking a remote method is identical to that of invoking a method in the same program.

ii. Parameter marshalling/unmarshalling is done by RMI.

(c) The *java.rmi* package contains classes that support RMI.

b. *Marshalling* in Java RMI

(a) Marshalling in RMI is based on Java's concept of serialization and *deserialization*.

i. *Serialization* refers to the activity of *flattening* an object into certain form of serial bytes so that it can be transmitted as an argument or result of RMI invocation.

ii. As the recipient of a serialized object does not know the original object, additional information has to be attached in the serialized form of an object to help the deserialization operation.

iii. For a class, the information is the name of the class and its version number. The version number can be supplied by a programmer, or can be calculated through class name hash.

iv. Any object referenced by an object will be serialized when the latter is serialized. Referenced objects are serialized as *handles*, which is a reference to an object within the serialized form.

(b) Example. For a structure

Person p = new Person("Smith", "London", 1934)

The corresponding serialized form is shown in Fig.4.9, p.150.

Serialized values				Explanation
Person	8-byte version number		h0	*class name,version number*
3	int year	java.lang.String name:	java.lang.String place:	*number,type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h_0 and h_1 are handles

Figure 4.9 Indication of Java serialized form

(c) Serialization and deserialization actions on arguments and results of remote invocations are performed by Java RMI support (which is part of the middleware).

(d) Programmers may write special methods to fulfill special serialization requirements if desired.

i. The *writeObject* method in an instance of the *ObjectOutputStream* class can be invoked to serialize an object (such as the *Person* structure) that is passed to it.

ii. Similarly, the *readObject* method in an instance of the *ObjectInputStream* class can be invoked to deserialize (restore) an object.

(6) Java IDL (since JDK1.2)

a. Java IDL is intended for Java applications and applets to communicate with objects written in any languages that support CORBA. RMI on the other hand is restricted to Java-to-Java communications.

b. Like DCE IDL, Java IDL is an interface definition language. Java IDL supports a compiler (idlj on UNIX, idlj.exe on Windows) to compile Java IDL files into Java files. The use of a standard interface definition language provides transparency to marshalling/unmarshalling operations.

c. *Marshalling* in Java IDL applications. Similar to DCE IDL, the first step of writing a Java IDL application is developing a Java IDL interface. For an interface named *Xyz* and an IDL file named *Xyz.idl*, the IDL compiler idlj.exe will compile it and produce the following files:

   (a) A *server skeleton* file _XyzImplBase.java , which provides basic CORBA functionality for the server. It implements the interface. This is the server stub file.

   (b) A *client stub* file _XyzStub.java, which provides basic CORBA functionality for the client.

   (c) A file Xyz.java which is the Java version of the IDL interface.

   (d) A file XyzHelper.java which provides auxiliary functionality required to cast CORBA object references to their proper types.

   (e) XyzHolder.java which holds a public instance member of type Xyz/. It provides operations for *out* and *inout* arguments, which CORBA has but which do not map easily to Java's semantics.

   A sample Java IDL file *Hello.idl*:

```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

4. Client-server communications

(1) Client-server communications are typical in DSs.

a. Client-server communications usually employ the *request-reply* protocol.

b. Request-reply communications can be synchronous or asynchronous. But most of time it is synchronous – a client normally suspends its operations while waiting for reply from server.

c. Request-reply communications are independent underlying transport layer protocols. Both UDP and TCP can be used.

d. Structure of request-reply messages (Fig.4.16, p.157)

messageType	*int (0=Request, 1=Reply)*
messageId	
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

Figure 4.16 Request-reply message structure

(a) *messageId*: (Note: the book showed a *requestId* in Fig.4.16. It then said that a *message identifier* contains a *requestId*) it consists of two parts: a *requestId* and an identifier of the sender process (its IP address and port number). *requestId* is taken from an increasing sequences of integers. A retransmitted request will carry the same requestId as in the original request.

(b) *methodId*: this is the identity of the remote method to be invoked.

(2) UDP datagram request-reply protocol implementation for RMI (Java RMI is used an example RMI)

a. Advantages: less comm. overhead

(a) No explicit ack is needed, as replies can serve as acks.

(b) TCP connection and termination involves seven message exchanges.

(c) There is no need of flow control (available in TCP) as most remote invocations are short in duration.

b. The request-reply protocol (Fig.4.11,p.145)

(a) The presented protocol is generic. However most RPC and RMI implementations support similar request-reply protocols.

(b) Three comm. primitives (Fig.4.15, p.157):

i. *doOperation* method: performs the action of invoking a remote method (i.e. sending a request).

ii. *getRequest* method: is used by a remote server process to receive service requests.

iii. *sendReply* method: sends the reply message to the client.

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments);

`sends a request message to the remote object and returns the reply. The arguments specify the remote object, the method to be invoked, and the arguments of that method.`

public byte[] getRequest ();

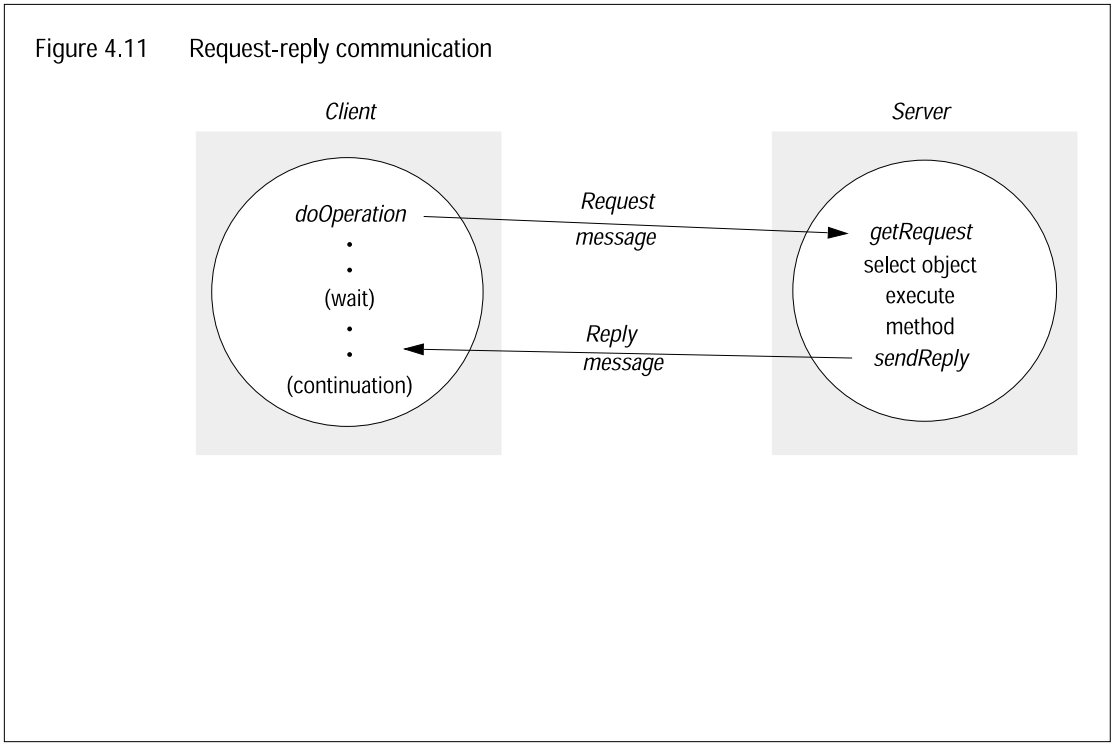

Figure 27: Request-reply communications (Fig.4.14, p.156)

`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.`

c. Possible errors. For each request, the following events might happen:

(a) Request message is not received by the server:

i. The request message is lost during comm.

ii. The server process is not running.

iii. The computer hosting the server process is not running.

(b) The reply message does not reach the client

i. The reply message is lost during comm.

(c) The server process crashes and restarts

(d) The client process crashes and restarts

(e) The request message is delayed

(f) The request message is corrupted

(g) The reply message is corrupted

* If (a), or (c), or (f) occurs, the request has not been served.
* If (b) or (g) occurs, the request has been served.
* If (c) or (d), the request may or may not have been served.

The critical observation is that, from a client's perspective, it has no way to know what exactly has happened that has caused it not to be able to receive the reply.

d. *Idempotent operations* and semantics of request-reply protocols

   (a) An operation is *idempotent* if the effect of performing the operation repeatedly is same as that of performing it exactly once. Example idempotent operations:
      i. Access a WEB page;
      ii. The operation of adding an element to a set. The property of sets guarantee idempotency.

   (b) *At-least-once, at-most-once, maybe*, and *exactly-once* semantics of request-reply protocol (Cf. Fig.5.6 also):
      i. *At-lease-once*: a request is guaranteed to be served by a server *at least once*. Example application: client may request a remote email server to send a email message to a specific user to notify approval of the user's application. This semantics is suitable for idempotent operations.
      ii. *At-most-once*: a request is guaranteed to be served by a server at most once. But it may not be served at all. Not very common.
      iii. *Maybe*: a request may or may not be served. Not very common. It arises when no any fault tolerance mechanism is taken.
      iv. *Exactly-once*: a request is guaranteed to be served by a server *exactly once*. Too many applications require this semantics.

   (c) Comments:
      i. Exactly-once semantics is from local procedure calls. Combination of at-least-once and at-most-once semantics provides exactly-once semantics.
      ii. Most IPC implementations do not support exactly-once semantics due to its overhead. It's left to applications to apply additional fault tolerance mechanisms to achieve exactly-once semantics.

   This topic will be re-examined in Section 6 "Distributed objects and RMI" in this lecture.

e. Failure model of the request-reply protocol.

   (a) A request-reply protocol based on UDP that does not add any mechanism to enhance reliability by itself suffers from two well-know failures:
      i. Omission failures;
      ii. Non-delivery of messages (violation of *validity* in our failure model).

   Such a protocol can only support *at-most-once* or *maybe* semantics. It cannot provide *at-least-once* or *exactly-once* semantics.

(b) *Timeout* and *retransmission*.

i. Timeout and retransmission are essential for a request-reply protocol to provide at-least-once or exactly-once semantics (although for the latter, additional mechanism is needed). The *doOperation* uses timeout to detect loss of requests or replies.

ii. For at-most-once and maybe semantics, timeout alone is sufficient.

(c) *Detecting and discarding duplicate request messages*.

i. For non-idempotent operations, a server in a client-server comm. must be able to detect a duplicate request message and discard it. Otherwise inconsistency may arise.

ii. A server should also re-transmit the reply when it detects a duplicate request message. This will keep the client from performing pair of time-out/retransmit operations repeatedly.

(d) *History*.

i. In order for a server to able to detect duplicate requests and retransmit replies, it must maintain, a record of executions of requests. Such a record is called *history*.

ii. When a request from a specific client arrives, the server will check the newly arrived request against the history. If that request is new, it will be served. If that request is a duplicate, a stored copy of reply will be fetched from the history and retransmitted to the client.

iii. If the client-server comm is synchronous, the server only has to keep one copy of completed request for each client. A new request from a client signals that that client has received reply to its previous request. Hence the server can purge the history to delete reply for that client.

(3) RPC exchange protocols (Fig.4.17, p.159)

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

Figure 4.17 RPC exchange protocols

a. The three RPC exchange protocols are:

(a) The request (R) protocol

(b) The request/reply (RR) protocol

(c) The request/reply/acknowledge (RRA) protocol

b. The three RPC exchange protocols were special request-reply protocols originally designed for RPC. But its semantics also applies to request-reply protocols in general.

c. **R** protocol. Used when no value has to be returned in response to a request and the client does not require/care confirmation from the server. It implements the *maybe* and *at-most-once* semantics. This protocol is the least reliable and least expensive.

d. **RR** protocol. Used in most RPC systems. This protocols implements the *exactly-once* semantics.

   (a) In order to filter out duplicate requests a server must keep three separate histories (also called *pools*) containing information of all outstanding requests:

      i. *Completion pool*: which contains those requests, together with their replies, that have been *recently* completed;

      ii. *Execution pool*: which contains those requests that are currently being served;

      iii. *Waiting pool*: which contains those requests that were received but are waiting their turn to be served.

   (b) A newly arrived RPC request is checked against the three pools as follows:

      i. If there is a request in the waiting pool or execution pool with the same *messageId* as the newly arrived request, the newly arrived request message is simply discarded. This is a duplicate request whose original request has been received but has not been completed.

      ii. If the messageId of the newly received request matches that of a request in the completion pool, this newly received request is also a duplicate, hence discarded. However, the reply for that request in the completion pool will be fetched and retransmitted to the client. This is a duplicate request whose original request has been completed.

      iii. If the newly arrived request came from some process with some *SourceAddress* $= x$ such that the *messageId* is not in the completion pool, then the newly arrived request is either put into the waiting pool or given the CPU for execution. In addition, entry in the completion pool with the *SourceAddress* equal to x is purged, because the client must have received the reply to its previous request.

e. **RRA** protocol.

(a) Problems with RR protocols: the waiting pool may grow very large, consuming system resources (a completed request is kept in the completion pool until the initiating client makes a new request. However, a client may never send new requests after initial interactions). RRA is a protocol that can remedy this problem.

(b) The RRA protocol:

- An acknowledgment message is required from the client to which a reply message has been sent (by the server). The ack message contains messageId of the original request message.
- After receiving the ack message the server can free all memory associated with that particular client's request.

(c) Alternatively, the server can adopt the strategy of holding completed requests in its histories for a certain period τ of time and deletes them after τ time. Deciding the value of τ is critical to the correctness and performance of a request-reply protocol. Too short: may not ensure at-most-once semantics. Too long: large waiting pool.

(4) Request-reply protocols implementation using TCP streams

a. Problems of datagram communications: (besides unreliability) the buffers sizes at the sender and receiver are difficult to decide.

(a) Certain applications require large buffers that are difficult to support by clients or servers.

(b) If a client supports small buffers, an application request that is significantly larger than size of the buffers will result in multiple datagram packets. An UDP based request-reply protocol has to deal with the issue of disassembling and reassembling these scrambled packets.

b. Benefits of TCP based request-reply protocols

(a) TCP is a stream protocol, hence buffer sizes no longer present a problem.

(b) TCP is reliable. Therefore retransmission is not an issue here.

(c) TCP significantly simplifies the implementation of request-reply protocols. If a client intends to communicate with a server for a relatively long time period, the overhead of TCP connection establishment and termination is negligible.

c. The request-reply protocol in Java RMI is TCP based. A server *extends* the *UnicastRemoteObject* class, which provides support for point-to-point active object references (invocations, parameters, and results) using TCP stream.

(5) Example: the HTTP (hypertext transfer protocol) protocol

a. A Web server manages different resources in different ways:

(a) as data such as texts and images of an HTML page.

(b) as program such *cgi* scripts/programs and *servlets*, which run on the web server.

b. HTTP is a request-reply protocol. It specifies formats of messages involved in a request-reply exchange, the supported methods on Web servers, arguments and results and rules for representing (i.e. marshalling) them in messages.

(a) HTTP supports a fixed set of methods that are applicable to all resources managed by a Web server. This is different from other protocols where different resources (objects) have different methods.

(b) HTTP supports content negotiations (what data format/representation a client can accept) and password based authentication mechanism.

(c) HTTP is based on TCP.

i. HTTP 1.0 would open/close a TCP session for every pair of request/reply exchange between a client and server (high overhead).

ii. HTTP 1.1 (and newer) uses *persistent connections* which allows a connection to be up through a sequence of request/reply exchanges. A persistent connection can be explicitly closed by either a client or server.

c. *Marshalling* in HTTP

(a) Requests and replies are marshalled into ASCII text strings unless specified otherwise.

(b) Requests and replies can also be communicated as byte sequences and compressed.

(c) Multimedia data are sent and received as MIME formed structures, which has a prefix about its MIME data type and subtype (such as *text/html, image/gif*). This information allows the receiving process to use appropriate applications to handle the data.

d. HTTP *messages*

i. Request messages (Fig.4.18, p.162)

method	*URL*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/1.1		

Figure 4.18 HTTP *request* message

ii. Reply messages (Fig.4.19, p.163)

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

Figure 4.19 HTTP *reply* message

iii. *header* field: contains request modifiers and client information: for reload, a condition (date) of retrieve a web page; for password based authentication, a encrypted password or other credentials.

iv. *message body* field: itself may also be structured and contain its own headers so that the client may invoke different applications to handle different types of data in it.

e. HTTP *methods*

(a) *GET*: requests all resources specified by the given URL. An HTML page located by an URL may refer to programs (CGI or Servlet), which will be invoked and results of the invocation will be returned.

(b) *HEAD*: Similar to GET. But it will not retrieve message body. It can be used by a proxy server or cache server to check freshness of a resource.

(c) *POST*: Places a document to a Web server. The URL that will be used to process the document is specified. Example applications that uses this method include:

i. Posting a message to a discussion group or message board.

ii. Submitting an online application, an online electronic payment, or an online order request.

A POST request normally will invoke a CGI script or Servlet on the Web server.

(d) *PUT*: specifies that the data supplied in the request should be stored with the given URL as identifier.

(e) *DELETE*: requests the remote Web server to delete the resource identified by the specified URL. Potentially insecure and disabled on many Web server.

(f) *OPTIONS*: mainly used by a server to declare to a client the list of methods that are allowed to be applied to the resources with the specified URL.

(g) *TRACE*: the server will echo the request sent by a client. For debugging purpose.

5. Remote procedure calls

(1) Motivations

a. A process that wants to access to a service hosted on a remote computer should be able to make a service request that specifies the *service name* and possibly some needed parameters, just like making a local procedure call (LPC). The calling process may not and does not want to know the physical location of the service.

b. Advantages of RPC.

The concept of RPC and its implementation are important steps toward *access transparency*. RPC relieves applications from the details of locations of available services, communications, transmission errors, and various failures.

(2) Syntax and semantics of local procedure calls

a. Before a procedure is actually invoked, a new environment is created, which includes the actual parameters passed, local variables, and variables in the program enclosing the called procedure.

b. The caller is suspended upon the call.

c. When the called procedure completes, it returns the results to the calling procedure and terminates.

d. The calling procedure resumes its execution.

(3) Syntax and semantics of RPCs:

a. The local process makes a RPC, providing the name of a remote procedure, and possibly a list of parameters.

b. The RPC will be caught by a communication daemon, which calls the *client stub* procedure. The client stub prepares a *request message* and sends it to the remote host.

c. A corresponding server stub procedure at the remote host accepts the request message, and prepares a local procedure call that implements the requested service. It then initiates the local procedure. The client and server stubs are illustrated in Fig.5.8, p.198.

d. The LPC at the remote host completes, and the results are given to the server stub at the remote host.

e. The server stub at the remote host prepares a *reply message* and sends it back to the client stub at the local host.

f. Upon receiving the reply message, the client stub will unpack the message and send the results back to the local process.

Figure 5.7 Role of client and server stub procedures in RPC

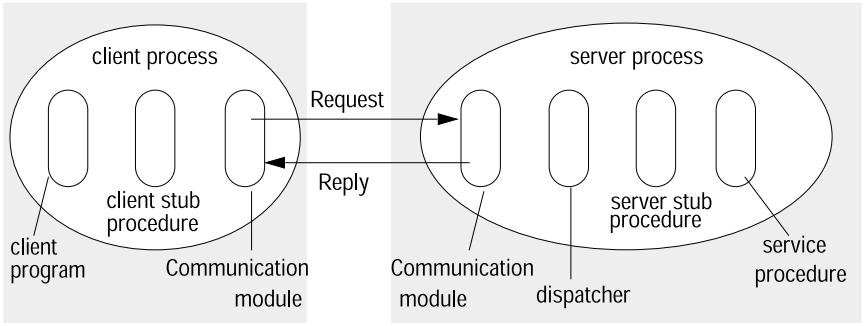

Figure 28: Role of client and server stub procedures in RPC (Fig.5.8, p.198)

(4) Major issues of RPCs

a. *Uniform call semantics* – transparency problem. Should communication failures and long delays be hidden?

b. *Type checking* – strong type checking available in local procedure call implementation should also be used for RPCs.

c. *Full parameter functionality* – all primitive data types should be allowed. Supports to structured and application defined data types must also be considered.

d. *Concurrency control and exception handling* – not fundamental, but should be considered and implemented if possible.

e. *Binding* – a remote procedure name called by a client will be eventually associated with a local procedure at a remote host. When will this association occur?

   (a) *Static binding*: bind it at compilation time. when a client program is compiled, every remote procedure name is bound for each RPC.

      i. Advantages: simple and easy to implement.

      ii. Problems: programs need to be recompiled when the service locations change.

(b) *Dynamic binding*: delay the binding until run-time

i. Advantages: location transparency is maintained because no recompilation is necessary.

ii. Problems: each site must maintain a table that maps every remote procedure name to a list of possible servers. The table must consulted for each RPC at run-time and updated frequently to reflect the real configuration.

(c) Binding based on name servers

i. A server name of a remote service must be registered

ii. Remote procedure name is resolved through name servers

iii. Flexible. Could be dynamic or static.

(5) External data representation

a. Methods of passing parameters and results in RPCs

a. *Input* parameters are passed by value

b. *Output* parameters are passed by reference

b. Marshalling. The client and server stub processes will perform marshalling/unmarshalling for applications.

c. Interface declaration: an interface must be defined for a RPC system. Each remote procedure must be assigned a unique procedure id and have an entry in the interface which must contain the following information:

(a) The types of parameters and results, for type checking. This is just like in LPC.

(b) Different languages have different notions of data types. The interface must contain adequate information so that stubs can perform appropriate type conversions if necessary (real in Pascal, float and double in C).

d. Stubs hide the remote nature of RPC. A stub has the following responsibilities:

(a) For each RPC creates an appropriate message that contains the arguments, message id, client id, message type, request id, and procedure id. A typical message is of the form. Note: each retransmitted RPC request message will carry the same messageId but different requestId fields.

```
type message record
                MessageType: {request, reply};
                MessageId: integer; (* one per message request*)
                RequestId: integer; (* one per RPC request *)
                SourceAddress: port; (* network address of client process *)
                ProcedureId: integer; (* unique server id in interface *)
                Argument: flattenedList;
```

end;

(b) Perform type checking and conversions according to the interface specification and declaration.
Example. The remote procedure has as its parameters an array of integers. The client calls the stub with an array name. Then the stub must get the values of the array and pass them as individual arguments.

(c) Initiate a message exchange request. Usually the message exchange will be done by a system *message module* on behalf of the stub.

(d) At the server end, the server stub will unpack the messages. Each host has a *dispatcher*, which, for each incoming request message, will look at the message for a *procedureId*, and *dispatches* the request to the intended server.

(e) When the remote procedure completes, the results are submitted to the server stub. The results are marshalled and a reply message is prepared by the server stub. It then sends the reply message to the client host through the message module at the remote host.

(f) The client stub at the client host receives and unpack the reply message and delivers the results to the client.

Stub hides and encapsulates all the details of type conversions and communications, making RPCs just like LPCs.

e. Special data structures: pointers, arrays, linked lists and etc.

(a) Prohibited by many RPC systems.

(b) Even if supported by some RPC systems, their semantics are much more restrictive than in LPCs.
Example. Linked lists: copy it and pass the whole; created at remote node; send back as results.

(6) Transparency

a. RPC is a high level client-server IPC facility. As such it faces all possible errors a request-reply protocol has to deal with (cf. RPC protocols in Section 4 of this lecture)

b. Many factors can affect transparency support in RPC:

(a) Comm. failures, which cause omission failures (loss of request and result messages).

(b) Crash failures of server hosts.

(c) Delay of message delivery.

Supporting high degree of transparency is challenging and is impossible in most cases.

(7) Concurrency

a. Blocking calls and their effects

(a) By definition, a client is blocked after a RPC request until a reply message is received.

(b) Problems: degree of parallelism is reduced and hence system performance may degrade. A process may be both a client and server. Blocking a client may prevent it from accepting incoming service requests.

b. *Threads*: a collection of processes created by a single parent that share the same address space as the parent. Benefits of using threads:

(a) Unlike conventional processes, threads have little context switching overhead.

(b) A client may create a collection of threads. Some them are responsible to accept incoming requests and others are responsible for making outgoing requests to other servers.

(c) Similarly, a server may also create multiple threads to serve requests concurrently.

6. Distributed objects and RMI (remote method invocation)

(1) The object model

a. **Objects**. An object contains a collection of of data items and a set of exported (also called public) methods for accessing and manipulating the data items.

(a) Data items in an object can be internal (private) or external (public). Private data items can only be accessed through exported methods. Therefore private data items and ways to access them are *protected*.

(b) Similarly methods can be private or public and public objects can be invoked by applications.

(c) Arguments and return values of each exported method is clearly defined.

(d) The name, argument types and their positions, and type of return value of a method together is termed as *signature* of that method. *Method overloading* allows the same method name to be associated with several different signatures to provide maximum convenience.

(e) The selection of data structures for data items and implementation of methods are also encapsulated. They can be changed if necessary without affecting object users as long as declarations of exported methods haven't been changed.

b. **Object references**. Objects are accessed through object references.

(a) Informally speaking, an object reference (which is given as the name of a class instance variable in C++ and Java) is a pointer that points to a storage location that contains specification of the object, values of each data item in the object, and code for each method (private or public).

(b) An object reference is internally represented an *object number* within each process (like file descriptors).

(c) To invoke a method in a given object, an application supplies an object reference to the object and the method name (plus appropriate arguments as specified in the method declaration). An object whose method is being invoked is called the *target* or *receiver*.

(d) Object references are *first-class* values – they may be assigned to variables, passed as arguments and as return values.

c. **Interfaces**.

(a) An interface contains definitions of signatures of a set of methods and possibly some data types that are used by arguments and return values of some methods . An interface does not contain implementation code for each method in it.

(b) The information in an interface is adequate for programmers to write applications that will invoke methods contained in the interface.

(c) Some OO programming languages like Java allow a class to have multiple interfaces, one for each specific type of applications.

(d) Interfaces do not contain constructors.

d. **Actions**. An action is an invocation of a method of an object by another object.

(a) Besides object reference, an invocation contains all the information in the signature of the invoked method.

(b) Effects of an action.

i. The target will execute the method and send return values back to the invoker.

ii. The execution of the specified method may cause changes in the state of the storage associated with the invoked object. Other objects may be invoked in the course of execution.

e. **Exceptions**. Exceptions are unexpected conditions that are raised during object invoking.

(a) The concept of exceptions is similar to the concept of *signals* in UNIX systems. For each exception, an implementation has its default ways to handle it. Applications can specify their own methods, which override the default ways, to handle exceptions.

(b) Use of exceptions enhances debugging and free programmers from details of dealing with each possible error.

f. **Garbage collection**. When an object is not referenced by any process, it becomes garbage. Garbage should be collected by reclaiming all physical resources (storage) allocated to it. Some OO languages like Java can automatically detect and collect garbages, while others like C++ do not have this support. In the latter case, applications have to free resources explicitly.

(2) Distributed objects and remote interfaces

a. Besides all the characteristics and issues of normal objects as discussed above (object interfaces and object references), distributed objects have its own unique properties and issues.

(a) Distributed objects are physically distributed on different computers and managed by different processes. This further enhances encapsulation.

(b) An invocation of a method of a distributed object involves an IPC message exchange.

b. Distributed processes can access distributed objects through client-server models using request-reply protocols.

(3) The distributed object model

a. The model (Fig.5.3, p.184)

(a) A DS consists of a collection processes, each of which manages a collection of objects. Some of these objects, termed as *remote objects*, are remotely invokable, while the rest can only receive local invocations.

(b) Method invocations between objects in different processes are called *remote method invocations* (RMI).

(c) Two fundamental concepts of distributed object model: *remote object reference* and *remote interface*.

b. Remote object references: Fig.5.4,p.185

(a) A remote object reference is an extension of local object reference. It is an identifier that can be used throughout a DS to refer and identify a particular unique remote object.

(b) Fig.4.13 illustrates elements of a remote object reference.

i. The *IP address* determines the remote host.

ii. The *port number* and *time* components uniquely identify a process on the remote host.

iii. A process will assign an *object number* (unique within the process) for every object created in the process.

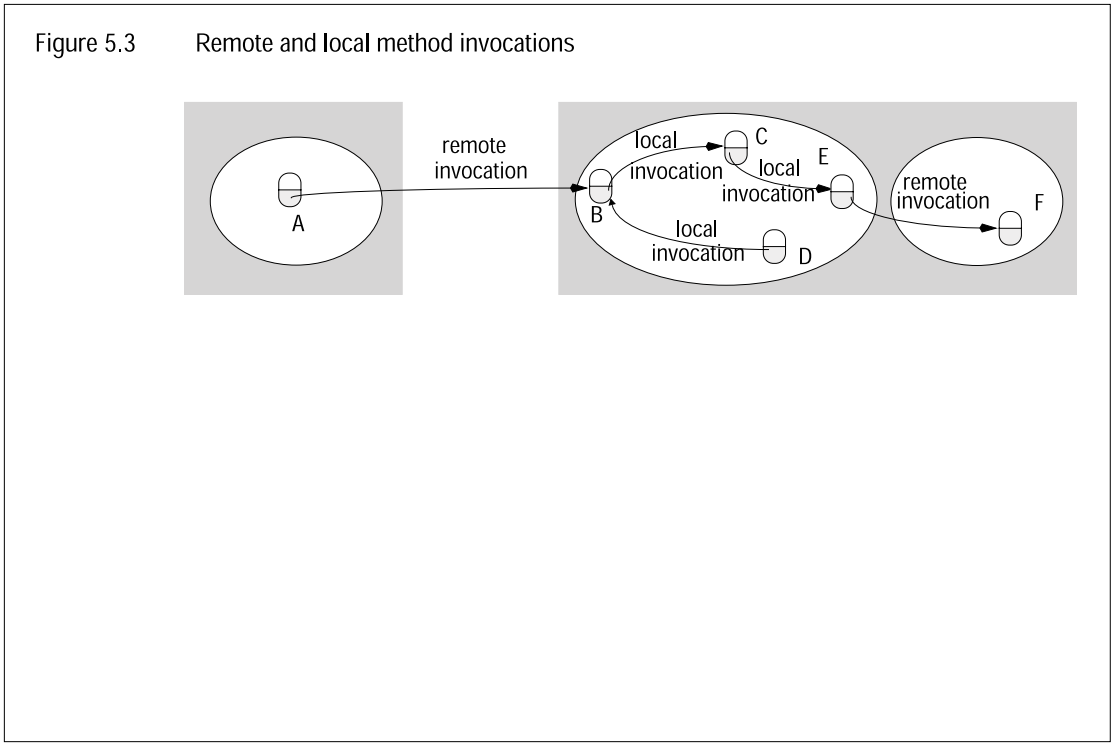

Figure 29: Remote and local method invocations (Fig.5.3, p.184)

iv. The *interface of remote object* is provided so that any process, which receives a remote object reference as an argument or result of a remote invocation, can use the specified remote object interface to find out the methods exported by the remote object.

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Figure 4.13 Representation of a remote object reference

c. Remote interfaces

(a) The concept of remote interfaces extends that of normal interfaces.

(b) Each remote object supports two groups of methods: remote invocable methods and those that can only be invoked locally. A remote interface specify all those remote invocable methods.

(c) In Java RMI, a remote interface is declared the same as a normal interface except that it must *Extends* the interface java.rmi.Remote to *acquire* remote invocable ability for all methods listed in the interface.

(d) CORBA IDL, Java IDL, and DCE IDL all support interface definitions. All objects (or procedures in DCE) can be invoked remotely.

d. Actions in distributed model may cause cascading remote object invocations by different processes hosted on different computers.

(4) Design issues of RMI

a. RMI invocation semantics. RMI can be viewed as a special form of client-server communications. Therefore the *at-least-once, at-most-once, maybe* and *exactly* semantics of request-reply protocols are also applicable to RMI (Cf. Section 4 of this lecture). Fig.5.6, p.187 summarizes RMI invocation semantics.

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

Figure 5.6 Invocation semantics

(a) *Maybe invocation semantics*. The method in an RMI request may or may have not been executed. The invoker cannot be sure either way.

i. This is the least reliable semantics. An RMI design that does not take any fault tolerance mechanisms (timeout, retransmissions, histories) will lead to this semantics. The server does not keep track if a RMI has been successfully completed and the client has no way to know one way or another.

ii. It suffers: (i) omission failures if the invocation or result message is lost; (ii) crash failures when the server hosting the remote object fails. A crash failure may occur either before or after the method is executed.

iii. Although this semantics is the least expensive to implement, it also has the least applicability. An example application using this semantics is sending a fun email message to a group of friends.

(b) *At-least-once invocation semantics*.

i. With this semantics, after an RMI the invoker will either receive a result, or it will receive an exception (such as a timeout) informing it that no result is received. In the former case, the invoker knows that the method has been executed at least once (possibly more than once!), and in the latter case, it knows that the method has been either executed once or not at all. In the latter case, retransmission will be used until a result is received.

ii. This semantics is suitable for *idempotent* RMI methods – methods whose multiple executions have the same effects as a single execution

iii. It suffers: (ii) crash failures when the server hosting the remote object fails; (ii) arbitrary fails – multiple executions of non-idempotent RMI methods will cause inconsistency.

(c) *At-most-once invocation semantics*. With this semantics, after an RMI the invoker will either receive a result, or it will receive an exception (such as a timeout) informing it that no result is received. In the former case, the invoker knows that the method has been executed *exactly once*, and in the latter case, it knows that the method has been either executed once or not at all. It is up to the application to decide what to do next in the second case.

b. Transparency. This issue is very similar to that in RPC (cf. Section 5 of this lecture)

(a) A RMI design supporting IDL can allow programmers to have a choice of calling semantics (at-least-once and etc.) so that a suitable semantics can be selected according to requirements of applications.

(b) RMI designs normally allow applications to place different exceptions handlers to handle different failures.

(c) The consensus is that an RMI should provide transparent syntax, but should specify the remote nature in interfaces. Java RMI uses the Remote interface to achieve this.

(5) Implementation issues of RMI

a. The main objects and modules in a RMI implementations are shown in Fig.5.7, p.190.

b. **Communication module**. This is the same module as in RPC (Fig.5.8). This module is responsible for actual receiving and sending of RMI request and reply messages, whose structure is shown in Fig.4.16.

(a) The comm. module only interprets the first three fields in an RMI message. The rest are left for RMI software.

(b) The comm. modules collectively implement a specified invocation semantics such as at-most-once.

(c) The comm. module in a server selects the dispatcher for the class of object to be invoked.

Figure 5.6 The role of proxy and skeleton in remote method invocation

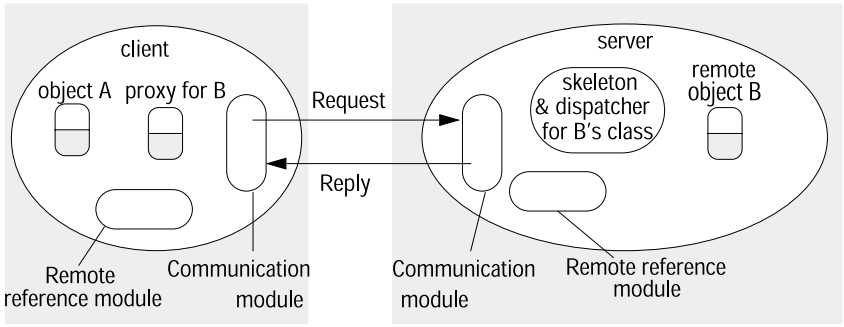

Figure 30: The role of proxy and skeleton in remote method invocation (Fig.5.7, p.190)

c. **Remote reference module** (RRM). Responsible for: a) translating between local and remote object references; b) creating remote object references.

(b) The RRM in each process has a *remote object table* that records the correspondence between local object references in that process and remote object references (system wide). That is, the table maps between local object references to remote object references. The table contains:

i. An entry for all the remote objects held by the process. That is one entry for each remote object managed by the process.

ii. An entry for each local proxy.

(b) This module is called components of RMI software when they marshall or unmarshall remote object references. When a request message arrives, the table is consulted to find out which local object is to be invoked. When a request message is to be sent out, the table will record the remote object reference and its local object reference.

(c) The actions of the RRM:

i. When a remote object is to be passed as argument or result for the first time, the RRM is asked to create a remote object reference that will be added to the table.

ii. 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. If 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.

d. **The RMI software**. A layer of software between the application objects and the communication and remote reference modules. It plays a role similar to stub procedures in RPC. It consists a *proxy*, a *dispatcher*, and a *skeleton*.

(a) *Proxy*. Proxies are *client stubs* (in terms of RPC terminologies) and hence are only in client processes. There is one proxy for each remote object for which a process holds a remote object reference.
A proxy for a remote object acts as a client's local representative or proxy (hence the name) for the remote object. The caller invokes a method on the local proxy which is responsible for carrying out the method call on the remote object. In RMI, a proxy for a remote object implements the same set of remote interfaces that a remote object implements.
When a proxy's method is invoked, it does the following:

· Initiates a connection with the remote JVM containing the remote object;

· Marshals (converts, writes and transmits) the parameters to the remote JVM;

· Waits for the result of the method invocation;

· Unmarshall (reads and converts) the return value or exception returned, and returns the value to the caller.

(b) *Dispatcher*. A dispatcher and a skeleton collectively function as a *server stub*, hence they only reside on a server process for a remote object. A server has one dispatcher and one skeleton for each class representing a remote object.
A dispatcher receives *request* messages from the communication module. For each request message, it uses *methodId* to select appropriate method in the skeleton, passing on the *request* message to the skeleton.

(c) *Skeleton*. A skeleton *implements* the methods in the remote interface. However, this implementation is not actual implementation of remote methods themselves. Rather, a skeleton implements how remote methods in the remote interface will be invoked on the server.
When a skeleton receives an incoming method invocation it does the following:

· Unmarshall (reads and converts) the parameters for the remote method;

· Invokes the method on the actual remote object implementation; and

· Marshals (converts, writes and transmits) the result (return value or exception) to the caller.

Comment: Java 1.2 no longer requires a skeleton class. Only a stub class is required and generated by rmic.

e. **Generation of the classes for proxies, dispatchers, and skeletons**. The classes for the RMI software are generated automatically by an interface compiler. In Java RMI, the set of methods available in a remote object is defined as a Java interface that is implemented within the class of the remote object. The Java RMI compiler (rmic.exe in Windows platform) generates the code for the proxy, dispatcher, and skeleton classes from the class of the remote object.

f. **Server and client programs**.

   (a) The server program contains the classes for the dispatcher and skeleton, together with the codes for classes (called *servant classes*) of all the remote objects it supports.

   (b) Because remote interfaces cannot include constructors, remote objects are created in either an *initialization section*, or in methods in a remote interface designed for that purpose. Such methods are called *factory methods*. Any remote object that intends to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose.

   (c) An *initialization section* is responsible for creating and initializing at least one of the remote objects to be hosted by the server. Additional remote objects may be created in response to requests from clients. The initialization section may also register some of its remote objects with a *binder*. Normally it will register just one of the remote objects, which can be used to access the rest.

   (d) The client program consists of classes for proxies for all of the remote objects that it will invoke. A binder can be used to look up remote object references.

g. **The binder**. Before invoking any remote method, a client has to obtain a reference to a remote object that hosts the remote method.

   (a) A *binder* in a DS is a separate service that maintains a table containing mappings from textual names of remote objects to remote object references.

   (b) A server can use a binder to register its remote objects by name. A client can use a binder to obtain remote object references.

   (c) Java RMI supports a binder RMIregistry. DCE RPC supports DCE name services. CORBA has its own Naming Service.

h. **Object location**.

   (a) In a distributed object system, if remote objects are static, i.e. they stay under management of the same process for the rest of their life after their creation, the remote object references as defined in Section 4.4 of the 1st textbook are adequate for clients to find remote object servers.

(b) If remote objects can migrate from process to process and from computer to computer, then the previously presented representation of remote object references is not sufficient for clients to locate remote object servers.

- In this case a so called *location service* is needed. Such a service will map remote object references to their probable current locations.
- A client will then have to present both a remote object reference and an a current location of the remote object for each RMI.

i. **Server threads**.

(a) An RMI executes a remote object in a server. During the execution, other local and remote objects may be invoked. Hence an RMI may take an arbitrary amount of time to return. In a single thread server, the server will be blocked before the return.

(b) A server can allocate a separate thread to executing each remote invocation. This demands the designers of a remote object to deal with potential problems with concurrent threads. (cf. the *Thread* class in Section 2 about Java TCP API of this lecture).

j. **Activation of remote objects**.

(a) Most servers that provide RMI services are not running as physical processes until explicitly invoked by a remote invocation. This is similar to an FTP or TELNET server, which is not actually *forked* as a separate server process (by the so called *Internet super daemon* process *inetd*) until a client request an FTP or TELNET session.

(b) *Active* and *passive* remote objects.

i. A remote object is termed as *active* when it is available for invocation within a running process. It is called *passive* if it is not currently active but can be made active.

ii. A passive object contains two ports: (i) the implementation of the methods; and (ii) its state in the marshalled form.

(c) *Activators*. A process that starts server processes (more precisely it activates passive processes) to host remote objects are called a activator. 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.

In Java RMI each host must run an activator as a system daemon process. The activator is responsible for activating objects on that host.

k. **Persistent object stores**.

(a) *Persistent objects*. An object that lives between activations is called persistent object (like *static* variables can survive between procedure calls).

(b) Persistent objects are generally managed by persistent object stores in a marshalled form on disk. They will be activated when methods in them are invoked by other object.

(c) The *Activatable* class Java and *Entity beans* in JavaBeans are examples of persistent objects.

7. Group communications

(1) Multicasting: sending or receiving messages to or from a targeted group of processes. Major issues in group communications:

a. Group membership managements.

b. Message delivery mechanisms.

c. Reliability and validity issues.

d. APIs.

(2) Example applications of multicast protocols. Group communication is another important way of message exchanges in DSs.

a. *Fault tolerance based on replicated services*: A service may be replicated and provided by multiple servers in a DS. A client may send a request (such as modify a data file) to all servers. Reply from any of them will satisfy the client. As long as one of the servers is still available, the service will be available to all clients.

b. *Better performance through replicated data*: for a set of replicated data, multicast is used to send a update request to all processes that manage a copy of the replicated data.

c. *Finding the discovery servers in spontaneous networking*: a server of a RPC service or a client may locate available discovery services in order to register its interface or look up interfaces of certain services in a spontaneous networking environment.

d. *Propagation of event notifications*: certain events such as unavailability or reavailability of a comm. link can be sent to a group of routers through multicasting.

(3) IP multicast

a. IP multicast is based on the support of IP.

(a) IP normally supports *unicast* communications (from one computer to another computer). However, in IPv4 an application can specify a *class D* IPv4 address (an address with prefix 1110) in an IP packet to indicate that the packet should be delivered to a group of computers.

(b) Each class D IPv4 address can be used to represent a *multicast group*.

b. Multicast API.

(a) The API to programmers is UDP. A process can perform the following steps to let the hosting computer become a member of a multicast group (Note: multicast membership is with respect to computers, not individual processes on a computer):

i. Obtain a UDP *socket* descriptor and *bind* the descriptor to a *port number*.

ii. Uses *setsockopt* function to set the *IP_ADD_MEMBERSHIP* option with the value of a class D IPv4 address.

If successful, after above operations the hosting computer becomes a member of the multicast group represented by the class D IPv4 address. The computer will be able to send multicast packets to and receive multicast packets from other members in the group.

(b) A process that initiated the membership of a multicast group can ask the hosting computer to leave a multicast group by calling *setsockopt* function to set the *IP_DROP_MEMBERSHIP* option.

(c) Multiple processes on a computer can initiate membership joining operations. The host will remain a member of a specific multicast group as long as at least one process who initiated a membership joining operation before hasn't initiated the corresponding membership dropping operation.

(d) Each multicast packet sent by a multicast process will also be copied to the hosting computer itself as an multicast packet input, although this can be explicitly disabled.

(e) A process can send a multicast packet to a multicast group even if it hasn't initiated an appropriate membership joining operation. However, such a process is not able to receive multicast packets to that multicast group.

c. Multicast on LAN and WAN. Members of a multicast group can be anywhere on a WAN.

(a) Management of multicast groups is done through *multicast routing protocols*.

(b) When a computer joins a multicast group, the computer sends a special control message (a IGMP message) to some attached multicast routers, which in turn will exchange this info with other multicast routers about the change.

(c) Similar actions will be taken when a computer leaves a multicast group.

(d) Informally, the multicast routers for a specific multicast group form a tree. This tree can grow when new members join and be pruned when some member leaves.

(e) On the datalink level (such as Ethernet), when a computer joins a multicast group, the datalink interface of that computer will be instructed to receive datalink packets destined to a special datalink address. For Ethernet, that address is 01:00:5e:00:01:01, which is the Ethernet multicast address. Ethernet packets with this address will be picked up by the interface. The IP layer will then examine the multicast IP address inside.

d. Failure model of IP multicasting.

(a) As IP multicasting uses UDP and is based on IP, it is *unreliable*.

i. Any multicast packet may not be guaranteed to be delivered to all members of the multicast group.

ii. Multicast packets may be delivered not in FOFI order.

(b) IP multicast is a basic multicast. Reliable multicast can be implemented by enhancing a basic multicast (Ch.12).

(4) Consequences of IP multicast failure model on different applications.

a. *Fault tolerance based on replicated services* and *Better performance through replicated data*: The applicability of IP multicast depends on the nature of the replicated data and replicated services. Newsgroups data files and services do not have to be consistent all the time. A user may succeed in posting an article to one news server while failing to post the same article to another news server. But that is acceptable in most cases. On the other hand, a banking account file and service have to be consistent.

b. *Finding the discovery servers in spontaneous networking*: IP multicast is still usable here.

c. *Propagation of event notifications*: This is again application dependent. Failing to notify certain routers about the current condition a comm. link is acceptable, while failing to notify a consensus to some processes in a distributed consensus algorithm (Ch.11) is not acceptable.

(5) Example application of Java IP multicast API (Fig. 4.20, p.166)

a. After compilation, the program can be run by the command:

java MulticastPeer char_string multicast_IP_number

where *char_string* is a char string to be multicast to group members. Note: Java program takes the first command-line parameter as *args[0]*, the second command-line parmeter as *args[1]*, and so on. Different from C/C++.

b. The constructor of the *MulticastSocket* class creates a multicast socket (a UDP socket) and binds the socket to port 6789.

c. The program joins the IPv4 multicast group with group IPv4 number 228.5.6.7 through the *joinGroup* method of object *s* of class *MulticastSocket*.

d. The After joining the group, it multicasts a message (a char string, which is the first command-line argument) and then iterates in a *for* loop three times to receive three multicast messages.

e The process then leaves the group by invoking the *leaveGroup* method of object *s*.

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
   public static void main(String args[]){
   // args give message contents and destination multicast
   //               group (e.g. "228.5.6.7")
   MulticastSocket s = null;
   try {
      InetAddress group = InetAddress.getByName(args[1]);
      s = new MulticastSocket(6789);
      s.joinGroup(group);
      byte [] m = args[0].getBytes();
      DatagramPacket messageOut = new DatagramPacket(m,m.length,group,6789);
      s.send(messageOut);
      // get messages from others in group
      byte[] buffer = new byte[1000];
      for(int i=0; i< 3;i++) {
         DatagramPacket messageIn = new DatagramPacket(buffer,buffer.length);
         s.receive(messageIn);
         System.out.println("Received:"+ new String(messageIn.getData()));
      }
      s.leaveGroup(group);
   }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
   }catch (IOException e){System.out.println("IO: " + e.getMessage());}
   }finally { if (s!=null) s.close();}
}
```