



# Distributed Systems: Concepts and Design

*Edition 3*

By George Coulouris, Jean Dollimore and Tim Kindberg  
Addison-Wesley, ©Pearson Education 2001.

## Chapter 4 Exercise Solutions

---

4.1 Is it conceivably useful for a port to have several receivers?

*4.1 Ans.*

If several processes share a port, then it must be possible for all of the messages that arrive on that port to be received and processed independently by those processes.

Processes do not usually share data, but sharing a port would require access to common data representing the messages in the queue at the port. In addition, the queue structure would be complicated by the fact that each process has its own idea of the front of the queue and when the queue is empty.

Note that a port group may be used to allow several processes to receive the same message.

---

4.2 A server creates a port which it uses to receive requests from clients. Discuss the design issues concerning the relationship between the name of this port and the names used by clients.

*4.2 Ans.*

The main design issues for locating server ports are:

(i) How does a client know what port and IP address to use to reach a service?

The options are:

- use a name server/binder to map the textual name of each service to its port;
- each service uses well-known location-independent port id, which avoids a lookup at a name server.

The operating system still has to look up the whereabouts of the server, but the answer may be cached locally.

(ii) How can different servers offer the service at different times?

Location-independent port identifiers allow the service to have the same port at different locations. If a binder is used, the client needs to reconsult the binder to find the new location.

(iii) Efficiency of access to ports and local identifiers.

Sometimes operating systems allow processes to use efficient local names to refer to ports. This becomes an issue when a server creates a non-public port for a particular client to send messages to, because the local name is meaningless to the client and must be translated to a global identifier for use by the client.

---

4.3 The programs in Figure 4.3 and Figure 4.4 are available on [cdk3.net/ipc](http://cdk3.net/ipc). Use them to make a test kit to determine the conditions in which datagrams are sometimes dropped. Hint: the client program should be able to vary the number of messages sent and their size; the server should detect when a message from a particular client is missed.

*4.3 Ans.*

For a test of this type, one process sends and another receives. Modify the program in Figure 4.3 so that the program arguments specify i) the server's hostname ii) the server port, iii) the number,  $n$  of messages to be sent and iv) the length,  $l$  of the messages. If the arguments are not suitable, the program should exit immediately. The program should open a datagram socket and then send  $n$  UDP datagram messages to the

server. Message  $i$  should contain the integer  $i$  in the first four bytes and the character '\*' in the remaining 1-4 bytes. It does not attempt to receive any messages.

Take a copy of the program in Figure 4.4 and modify it so that the program argument specifies the server port. The program should open a socket on the given port and then repeatedly receive a datagram message. It should check the number in each message and report whenever there is a gap in the sequence of numbers in the messages received from a particular client.

Run these two programs on a pair of computers and try to find out the conditions in which datagrams are dropped, e.g. size of message, number of clients.

- 
- 4.4 Use the program in Figure 4.3 to make a client program that repeatedly reads a line of input from the user, sends it to the server in a UDP datagram message, then receives a message from the server. The client sets a timeout on its socket so that it can inform the user when the server does not reply. Test this client program with the server in Figure 4.4.

*4.4 Ans.*

The program is as Figure 4.4 with the following amendments:

```
DatagramSocket aSocket = new DatagramSocket();
aSocket.setSoTimeout(3000); // in milliseconds
while (// not eof) {
try{
// get user's input and put in request
.....
aSocket.send(request);
.....
aSocket.receive(reply);
}catch (InterruptedException e){System.out.println("server not responding");}
```

- 
- 4.5 The programs in Figure 4.5 and Figure 4.6 are available at [cdk3.net/ipc](http://cdk3.net/ipc). Modify them so that the client repeatedly takes a line of user's input and writes it to the stream and the server reads repeatedly from the stream, printing out the result of each read. Make a comparison between sending data in UDP datagram messages and over a stream.

*4.5 Ans.*

The changes to the two programs are straightforward. But students should notice that not all sends go immediately and that receives must match the data sent.

For the comparison. In both cases, a sequence of bytes is transmitted from a sender to a receiver. In the case of a message the sender first constructs the sequence of bytes and then transmits it to the receiver which receives it as a whole. In the case of a stream, the sender transmits the bytes whenever they are ready and the receiver collects the bytes from the stream as they arrive.

- 
- 4.6 Use the programs developed in Exercise 4.5 to test the effect on the sender when the receiver crashes and vice-versa.

*4.6 Ans.*

Run them both for a while and then kill first one and then the other. When the reader process crashes, the writer gets IOException - broken pipe. When writer process crashes, the reader gets EOF exception.

- 
- 4.7 Sun XDR marshals data by converting it into a standard big-endian form before transmission. Discuss the advantages and disadvantages of this method when compared with CORBA's CDR.

*4.7 Ans.*

The XDR method which uses a standard form is inefficient when communication takes place between pairs of similar computers whose byte orderings differ from the standard. It is efficient in networks in which the byte-

ordering used by the majority of the computers is the same as the standard form. The conversion by senders and recipients that use the standard form is in effect a null operation.

In CORBA CDR senders include an identifier in each message and recipients to convert the bytes to their own ordering if necessary. This method eliminates all unnecessary data conversions, but adds complexity in that all computers need to deal with both variants.

- 
- 4.8 Sun XDR aligns each primitive value on a four byte boundary, whereas CORBA CDR aligns a primitive value of size  $n$  on an  $n$ -byte boundary. Discuss the trade-offs in choosing the sizes occupied by primitive values.

*4.8 Ans.*

Marshalling is simpler when the data matches the alignment boundaries of the computers involved. Four bytes is large enough to support most architectures efficiently, but some space is wasted by smaller primitive values. The hybrid method of CDR is more complex to implement, but saves some space in the marshalled form. Although the example in Figure 4.8 shows that space is wasted at the end of each string because the following long is aligned on a 4- byte boundary.

- 
- 4.9 Why is there no explicit data-typing in CORBA CDR?

*4.9 Ans.*

The use of data-typing produces costs in space and time. The space costs are due to the extra type information in the marshalled form (see for example the Java serialized form). The performance cost is due to the need to interpret the type information and take appropriate action.

The RMI protocol for which CDR is designed is used in a situation in which the target and the invoker know what type to expect in the messages carrying its arguments and results. Therefore type information is redundant. It is of course possible to build type descriptors on top of CDR, for example by using simple strings.

- 
- 4.10 Write an algorithm in pseudocode to describe the serialization procedure described in Section 4.3.2. The algorithm should show when handles are defined or substituted for classes and instances. Describe the serialized form that your algorithm would produce when serializing an instance of the following class *Couple*.

```
class Couple implements Serializable{
    private Person one;
    private Person two;
    public Couple(Person a, Person b) {
        one = a;
        two = b;
    }
}
```

*4.10 Ans.*

The algorithm must describe serialization of an object as writing its class information followed by the names and types of the instance variables. Then serialize each instance variable recursively.

```

serialize(Object o) {
    c = class(o);
    class_handle = get_handle(c);
    if (class_handle==null) // write class information and define class_handle;
    write class_handle
    write number (n), name and class of each instance variable

    object_handle = get_handle(o);
    if (object_handle==null) {
        define object_handle;
        for (iv = 0 to n-1)
            if (primitive(iv) ) write iv
            else serialize( iv)
    }
    write object_handle
}

```

To describe the serialized form that your algorithm would produce when serializing an instance of the class *Couple*.

For example declare an instance of *Couple* as

```

Couple t1 = new Couple(new Person("Smith", "London", 1934),
new Person("Jones", "Paris", 1945));

```

The output will be:

<i>Serialized values</i>				<i>Explanation</i>
<i>Couple</i>	8 byte version number		h0	<i>class name, version number, handle</i>
2	Person one	Person two		<i>number, type and name of instance variables</i>
Person	8 byte version number		h1	<i>serialize instance variable one of Couple</i>
3	int year	java.lang.String name	java.lang.String place	
1934	5 Smith	6 London	h2	<i>serialize instance variable two of Couple values of instance variables</i>
h1				
1945	5 Jones	5 Paris	h3	

- 
- 4.11 Write an algorithm in pseudocode to describe deserialization of the serialized form produced by the algorithm defined in Exercise 4.10. Hint: use reflection to create a class from its name, to create a constructor from its parameter types and to create a new instance of an object from the constructor and the argument values.

*4.11 Ans.*

Whenever a handle definition is read, i.e. a class\_info, handle correspondence or an object, handle correspondence, store the pair by method map. When a handle is read look it up to find the corresponding class or object.

```

Object deserialize(byte [] stream) {
    Constructor aConstructor;
    read class_name and class_handle;
    if (class_information == null) aConstructor = lookup(class_handle);
    else {
        Class cl = Class.forName(class_name);
        read number (n) of instance variables
        Class parameterTypes[] = new Class[n];
        for (int i=0 to n-1) {
            read name and class_name of instance variable i
            parameterTypes[i] = Class.forName(class_name);
        }
        aConstructor = cl.getConstructor(parameterTypes);
        map(aConstructor, class_handle);
    }
    if (next item in stream is object_handle) o = lookup(object_handle);
    else {
        Object args[] = new Object[n];
        for (int i=0 to n-1) {
            if (next item in stream is primitive) args[i] = read value
            else args[i] = deserialize(rest of stream)
        }
        Object o = cnew.newInstance(args);
        read object_handle from stream
        map(object, object_handle)
        return o;
    }
}
}

```

- 4.12 Define a class whose instances represent remote object references. It should contain information similar to that shown in Figure 4.10 and should provide access methods needed by the request-reply protocol. Explain how each of the access methods will be used by that protocol. Give a justification for the type chosen for the instance variable containing information about the interface of the remote object.

*4.12 Ans.*

```

class RemoteObjectReference{
    private InetAddress ipAddress;
    private int port;
    private int time;
    private int objectNumber;
    private Class interface;
    public InetAddress getIPaddress() { return ipAddress;}
    public int getPort() { return port;};
}

```

The server looks up the client port and IP address before sending a reply.

The variable interface is used to recognize the class of a remote object when the reference is passed as an argument or result. Chapter 5 explains that proxies are created for communication with remote objects. A proxy needs to implement the remote interface. If the proxy name is constructed by adding a standard suffix to the interface name and all we need to do is to construct a proxy from a class already available, then its string name is sufficient. However, if we want to use reflection to construct a proxy, an instance of Class would be needed. CORBA uses a third alternative described in Chapter 17.

- 4.13 Define a class whose instances represent request and reply messages as illustrated in Figure 4.13. The class should provide a pair of constructors, one for request messages and the other for reply messages, showing how the request identifier is assigned. It should also provide a method to marshal itself into an array of bytes and to unmarshal an array of bytes into an instance.

4.13 Ans.

```

private static int next = 0;
private int type
private int requestId;
private RemoteObjectRef o;
private int methodId;
private byte arguments[];
public RequestMessage( RemoteObjectRef aRef,
    int aMethod, byte[] args){
    type=0; ... etc.
    requestId = next++; // assume it will not run long enough to overflow
}
public RequestMessage(int rId, byte[] result){
    type=1; ... etc.
    requestId = rid;
    arguments = result;
}

public byte[] marshall() {
    // converts itself into an array of bytes and returns it
}
public RequestMessage unmarshall(byte [] message) {
    // converts array of bytes into an instance of this class and returns it
}
public int length() { // returns length of marshalled state/
public int getID(){ return requestId;}
public byte[] getArgs(){ return arguments;}
}

```

- 
- 4.14 Program each of the three operations of the request-reply protocol in Figure 4.123, using UDP communication, but without adding any fault-tolerance measures. You should use the classes you defined in Exercise 4.12 and Exercise 4.13.

4.14 Ans.

```

class Client{
    DatagramSocket aSocket ;
    public static messageLength = 1000;
    Client(){
        aSocket = new DatagramSocket();
    }
    public byte [] doOperation(RemoteObjectRef o, int methodId,
        byte [] arguments){
        InetAddress serverIp = o.getIpAddress();
        int serverPort = o.getPort();
        RequestMessage rm = new RequestMessage(0, o, methodId, arguments );
        byte [] message = rm.marshall();
        DatagramPacket request =
            new DatagramPacket(message,message.length(0,serverIp, serverPort);
        try{
            aSocket.send(request);
            byte buffer = new byte[messageLength];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            return reply;
        }catch (SocketException e){...}
    }
}
Class Server{

```

```

private int serverPort = 8888;
public static int messageLength = 1000;
DatagramSocket mySocket;
public Server(){
    mySocket = new DatagramSocket(serverPort);
    // repeatedly call GetRequest, execute method and call SendReply
}
public byte [] getRequest(){
    byte buffer = new byte[messageLength];
    DatagramPacket request = new DatagramPacket(buffer, buffer.length);
    mySocket.receive(request);
    clientHost = request.getHost();
    clientPort = request.getPort();
    return request.getData();
}
public void sendReply(byte[]reply, InetAddress clientHost, int clientPort){
    byte buffer = rm.marshall();
    DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
    mySocket.send(reply);
}
}

```

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

4.15 Ans.

```

class Server{
    private int serverPort = 8888;
    public static int messageLength = 1000;
    DatagramSocket mySocket;

    public Server(){
        mySocket = new DatagramSocket(serverPort);
        while(true){
            byte [] request = getRequest();
            Worker w = new Worker(request);
        }
    }
    public byte [] getRequest(){
        //as above}
    public void sendReply(byte[]reply, InetAddress clientHost, int clientPort){
        // as above}
}
class Worker extends Thread {
    InetAddress clientHost;
    int clientPort;
    int requestId;
    byte [] request;
    public Worker(request){
        // extract fields of message into instance variables
    }
    public void run(){
        try{
            req = request.unmarshal();
            byte [] args = req.getArgs();
            //unmarshall args, execute operation,
            // get results marshalled as array of bytes in result

```

```

        RequestMessage rm = new RequestMessage( requestId, result);
        reply = rm.marshall();
        sendReply(reply, clientHost, clientPort );
    }catch {... }
}
}

```

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

*4.16 Ans.*

With a timeout set on a socket, a receive operation will block for the given amount of time and then an *InterruptedIOException* will be raised.

In the constructor of *Client*, set a timeout of say, 3 seconds

```

Client(){
    aSocket = new DatagramSocket();
    aSocket.setSoTimeout(3000);// in milliseconds
}

```

In *doOperation*, catch *InterruptedIOException*. Repeatedly send the Request message and try to receive a reply, e.g. 3 times. If there is no reply, return a special value to indicate a failure.

```

public byte [] doOperation(RemoteObjectRef o, int methodId,
    byte [] arguments){
    InetAddress serverIp = o.getIpAddress();
    int serverPort = o.getPort();
    RequestMessage rm = new RequestMessage(0, o, methodId, arguments );
    byte [] message = rm.marshall();
    DatagramPacket request =
        new DatagramPacket(message,message.length(0, serverIp, serverPort);
    for(int i=0; i<3;i++){
        try{
            aSocket.send(request);
            byte buffer = new byte[messageLength];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            return reply;
        }catch (SocketException e){}
        }catch (InterruptedIOException e){}
    }
    return null;
}

```

- 4.17 Describe a scenario in which a client could receive a reply from an earlier call.

*4.17 Ans.*

Client sends request message, times out and then retransmits the request message, expecting only one reply. The server which is operating under a heavy load, eventually receives both request messages and sends two replies.

When the client sends a subsequent request it will receive the reply from the earlier call as a result. If request identifiers are copied from request to reply messages, the client can reject the reply to the earlier message.

- 4.18 Describe the ways in which the request-reply protocol masks the heterogeneity of operating systems and of computer networks.

4.18 Ans.

(i) Different operating systems may provide a variety of different interfaces to the communication protocols. These interfaces are concealed by the interfaces of the request-reply protocol.

(ii) Although the Internet protocols are widely available, some computer networks may provide other protocols. The request-reply protocol may equally be implemented over other protocols.

[In addition it may be implemented over either TCP or UDP.]

---

4.19 Discuss whether the following operations are *idempotent*:

- Pressing a lift (elevator) request button;
- Writing data to a file;
- Appending data to a file.

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

4.19 Ans.

The operation to write data to a file can be defined (i) as in Unix where each write is applied at the read-write pointer, in which case the operation is not idempotent; or (ii) as in several file servers where the write operation is applied to a specified sequence of locations, in which case, the operation is idempotent because it can be repeated any number of times with the same effect. The operation to append data to a file is not idempotent, because the file is extended each time this operation is performed.

The question of the relationship between idempotence and server state requires some careful clarification. It is a necessary condition of idempotence that the effect of an operation is independent of previous operations. Effects can be conveyed from one operation to the next by means of a server state such as a read-write pointer or a bank balance. Therefore it is a necessary condition of idempotence that the effects of an operation should not depend on server state. Note however, that the idempotent file write operation does change the state of a file.

---

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

4.20 Ans.

To enable reply messages to be re-transmitted without re-executing operations, a server must retain the last reply to each client. When RR is used, it is assumed that a request message is an acknowledgement of the last reply message. Therefore a reply message must be held until a subsequent request message arrives from the same client. The use of storage can be reduced by applying a timeout to the period during which a reply is stored. The storage requirement for RR = average message size  $\times$  number of clients that have made requests since timeout period. When RRA is used, a reply message is held only until an acknowledgement arrives. When an acknowledgment is lost, the reply message will be held as for the RR protocol.

---

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

4.21 Ans.

The timeout period for storing a reply message is the maximum time that it is likely for any client to re-transmit a request message. There is no definite value for this, and there is a trade-off between safety and buffer space. In the case of RRA, reply messages are generally discarded before the timeout period has expired because an acknowledgement is received. Suppose that a server using RRA re-transmits the reply message after a delay and consider the case where the client has sent an acknowledgement which was late or lost. This requires (i) the client to recognise duplicate reply messages and send corresponding extra acknowledgements and (ii) the server to handle delayed acknowledgments after it has re-transmitted reply messages. This possible improvement gives little reduction in storage requirements (corresponding to the occasional lost acknowledgement message) and is not convenient for the single threaded client which may be otherwise occupied and not be in a position to send further acknowledgements.

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

4.22 Ans.

The time for the exchange of a message =  $A + B * \text{length}$ , where A is the fixed processing overhead and B is the rate of transmission. A is large because it represents significant processing at both sender and receiver; the sending of data involves a system call; and the arrival of a message is announced by an interrupt which must be handled and the receiving process is scheduled. Protocols that involve several rounds of messages tend to be expensive because of paying the A cost for every message.

The new version of RRA has:

<i>client</i>	<i>server</i>
cancel any outstanding Acknowledgement on a timer send Request	
	receive Request send Reply
receive Reply set timer to send Acknowledgement after delay T	
	receive Acknowledgement

The client always sends an acknowledgement, but it is piggy-backed on the next request if one arises in the next T seconds. It sends a separate acknowledgement if no request arises. Each time the server receives a request or an acknowledgement message from a client, it discards any reply message saved for that client.

4.23 IP multicast provides a service that suffers from omission failures. Make a test kit, possibly based on the program in Figure 4.17, to discover the conditions under which a multicast message is sometimes dropped by one of the members of the multicast group. The test kit should be designed to allow for multiple sending processes.

4.23 Ans.

The program in Figure 4.17 should be altered so that it can run as a sender or just a receiver. A program argument could specify its role. As in Exercise 4.3 the number of messages and their size should be variable and a sequence number should be sent with each one. Each recipient records the last sequence number from each sender (sender IP address can be retrieved from datagrams) and prints out any missing sequence numbers. Test with several senders and receivers and message sizes to discover the load required to cause dropped messages. The test kit should be designed to allow for multiple sending processes.

4.24 Outline the design of a scheme that uses message retransmissions with IP multicast to overcome the problem of dropped messages. Your scheme should take the following points into account:

- i) there may be multiple senders;
- ii) generally only a small proportion of messages are dropped;
- iii) unlike the request-reply protocol, recipients may not necessarily send a message within any particular time limit.

Assume that messages that are not dropped arrive in sender ordering.

4.24 Ans.

To allow for point (i) senders must attach a sequence number to each message. Recipients record last sequence number from each sender and check sequence numbers on each message received.

For point (ii) a negative acknowledgement scheme is preferred (recipient requests missing messages, rather than acknowledging all messages). When they notice a missing message, they send a message to the sender to ask for it. To make this work, the sender must store all recently sent messages for retransmission. The sender re-transmits the messages as a unicast datagram.

Point (iii) - refers to the fact that we can't rely on a reply as an acknowledgement. Without acknowledgements, the sender will be left holding all sent messages in its store indefinitely. Possible solutions: a) senders discards stored messages after a time limit b) occasional acknowledgements from recipients which may be piggy backed on messages that are sent.

Note requests for missing messages and acknowledgments are simple - they just contain the sequence numbers of a range of lost messages.

---

4.25 Your solution to Exercise 4.24 should have overcome the problem of dropped messages in IP multicast. In what sense does your solution differ from the definition of reliable multicast?

*4.25 Ans.*

Reliable multicast requires that any message transmitted is received by all members of a group or none of them. If the sender fails before it has sent a message to all of the members (e.g. if it has to retransmit a message) or if a gateway fails, then some members will receive the message when others do not.

---

4.26 Devise a scenario in which multicasts sent by different clients are delivered in different orders at two group members. Assume that some form of message retransmissions are in use, but that messages that are not dropped arrive in sender ordering. Suggest how recipients might remedy this situation.

*4.26 Ans.*

Sender1 sends request r1 to members m1 and m2 but the message to m2 is dropped

Sender2 sends request r2 to members m1 and m2 (both arrive safely)

Sender1 re-transmits request r1 to member m2 (it arrives safely).

Member m1 receives the messages in the order r1;r2. However m2 receives them in the order r2;r1.

To remedy the situation. Each recipient delivers messages to its application in sender order. When it receives a message that is ahead of the next one expected, it hold it back until it has received and delivered the earlier re-transmitted messages.

---

4.27 Define the semantics for and design a protocol for a group form of request-reply interaction, for example using IP multicast.

*4.27 Ans.*

The group request-reply protocol should not use a request-reply to each member. Instead, the request message is multicast to all the members of the group. But there is a question as to how many replies should be transmitted. As examples: a request to a replicated or a partitioned service requires only one reply. In the first case from any member and in the second case from the server with the information. In contrast a request for a vote requires a majority of replies that agree; and a request for a reading on a sensor requires all replies. The semantics should allow the client to specify the number of replies required.

In a group request-reply protocol, the request message is multicast to all the members of the group, using a retransmission scheme like that of the request-reply protocol to deal with lost messages. This requires that each member returns either an acknowledgement or a reply. The client's communication software collates and filters these replies, returning the desired number of replies to the client.

The protocol must deal with the case when there are less replies than the number specified, either by selective re-transmission of the request messages or by repeating the multicast request.