Distributed Systems: Concepts and Design

Chapter 5 Exercise Solutions

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

5.1 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;}
}
```

5.2 Program each of the three operations of the request-reply protocol in Figure 5.3, using UDP communication, but without adding any fault-tolerance measures. You should use the classes you defined in Exercise 4.13 and Exercise 5.1.

5.2 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);
    }
}
```

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

5.3 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.marshal();
            sendReply(reply, clientHost, clientPort );
        }catch {... }
    }
}
```

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

5.4 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;
    }
```

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

5.5 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.

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

5.6 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.]

5.7 Discuss whether the following operations are *idempotent*:

i) Pressing a lift (elevator) request button;

ii) Writing data to a file;

iii) Appending data to a file.

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

5.7 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.

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

5.8 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 **x** 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.

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

5.9 Ans.

The timeout period for storing a reply message is the maximum time that it is likely for any client to a 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.

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

5.10 Ans.

The time for the exchange of a message = A + B* 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:

client	*server*
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.

5.11 The *Election* interface provides two remote methods:

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

result: with two parameters through which the server supplies the client with the name of a candidate

and the number of votes for that candidate.

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

5.11 Ans.

vote: input parameters: name of candidate, voter's number;
result: output parameters: name of candidate, number of votes

---

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

5.12 Ans.

A process is informed that a connection is broken:

- when one of the processes exits or closes the connection.
- when the network is congested or fails altogether

Therefore a client process cannot distinguish between network failure and failure of the server.

Provided that the connection continues to exist, no messages are lost, therefore, every request will receive a corresponding reply, in which case the client knows that the method was executed exactly once.

However, if the server process crashes, the client will be informed that the connection is broken and the client will know that the method was executed either once (if the server crashed after executing it) or not at all (if the server crashed before executing it).

But, if the network fails the client will also be informed that the connection is broken. This may have happened either during the transmission of the request message or during the transmission of the reply message. As before the method was executed either once or not at all.

Therefore we have at-most-once call semantics.

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

5.13 Ans.

CORBA IDL:

```
interface Election {
    void vote(in string name, in long number);
    void result(out string name, out long votes);
};
```

Java RMI

We need to define a class for the result e.g.

```
class Result {
    String name;
    int votes;
}
```

The interface is:

```
import java.rmi.*;
public interface Election extends Remote{
    void vote(String name, int number) throws RemoteException;
    Result result () throws RemoteException;
};
```

This example shows that the specification of input arguments is similar in CORBA IDL and Java RMI.

This example shows that if a method returns more than one result, Java RMI is less convenient than CORBA IDL because all output arguments must be packed together into an instance of a class.

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

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

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

5.14 Ans.

Maybe call semantics is obviously inadequate for *vote*! Ex 5.1 specifies that the voter's number is used to ensure that the user only votes once. This means that the server keeps a record of who has voted. Therefore at-least-once semantics is alright, because any repeated attempts to vote are foiled by the server.

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

5.15 Ans.

In the first case, the implementor assumes that if the client observes an omission failure it cannot tell whether it is due to loss of the request or reply message, to the server having crashed or having taken longer than usual. Therefore when the request is re-transmitted the client may receive late replies to the original request. The implementation must deal with this.

In the second case, an omission failure observed by the client cannot be due to the server taking too long. Therefore when the request is re-transmitted after time T, it is certain that a late reply will not come from the server. There is no need to deal with late replies

---

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

5.16 Ans.

Suppose that each vote in the form {*String vote*, *int number*} is appended to a data structure such as a Java *Vector*. Before this is done, the voter number in the request message must be checked against every vote recorded in the *Vector*. Note that an array indexed by voter's number is not a practical implementation as the numbers are allocated sparsely.

The operations to access and update a *Vector* are synchronized, making concurrent access safe.

Alternatively use any form of synchronization to ensure that multiple clients' access and update operations do not conflict with one another.

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

5.17 Ans.

The state of the server must be recorded in persistent storage so that it can be recovered when the server is restarted. It is essential that every successful vote is recorded in persistent storage before the client request is acknowledged.

A simple method is to serialize the *Vector* of votes to a file after each vote is cast.

A more efficient method would append the serialized votes incrementally to a file.

Recovery will consist of de-serializing the file and recreating a new vector.

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

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

Hint: an instance variable of the proxy class should hold a remote object reference (see Exercise 4.12).

5.18 Ans.

Use classes *Class* and *Method*. Use type *RemoteObjectRef* as type of instance variable. The class *Class* has method *getMethod* whose arguments give class name and an array of parameter types. The proxy's *vote* method, should have the same parameters as the vote in the remote interface - that is: two parameters of type *String* and *int*. Get the object representing the *vote* method from the class *Election* and pass it as the second argument of *doOperation*. The two arguments of *vote* are converted to an array of byte and passed as the third argument of *doOperation.*

```
import java.lang.reflect;

class VoteProxy {
    RemoteObjectRef ref;
     private static Method voteMethod;
     private static Method resultMethod;
    static {
      try {
        voteMethod = Election.class.getMethod ("vote", new Class[]
            {java.lang.String.class,int.class}));
        resultMethod = Election.class.getMethod ("result", new Class[] {}));
      }catch(NoSuchMethodException){}
    }

public void vote  (String arg1, int arg2) throws RemoteException {
        try {
            byte args [] = // convert arguments arg1 and arg2 to an array of bytes
            byte result = DoOperation(ref, voteMethod, args);
            return ;
        } catch (...) {}
}
```

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

5.19 Ans.

Each proxy method is generated from the signature of the method in the IDL interface,e.g.

void vote(in string name, in long number);

An equivalent stub method in the client language e.g. C++ is produced e.g.

*void vote(const char *vote, int number)*

Each method in the interface is given a number e.g. vote = 1, result = 2.

use *char args[length of string + size of int]* and marshall two arguments into this array and call *doOperation* as follows:

*char * result = DoOperation(ref, 1, args);*

we still assume that *ref* is an instance variable of the proxy class. A marshalling method is generated for each argument type used.

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

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

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

5.20 Ans.

Use the class *Method*. To invoke a method supply the object to be invoked and an array of *Object* containing the arguments. The arguments supplied in an array of bytes must be converted to an array of *Object*.

```
public void dispatch(Object target, Method aMethod, byte[] args)
    throws RemoteException {
    Object[] arguments = // extract arguments from array of bytes
    try{
        aMethod.invoke(target, arguments);
    catch(...){}
}
```

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

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

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

5.21 Ans.

The method *DoOperation* sends the invocation to the target's remote object reference by setting up a TCP connection (as shown in Figures 4.5 and 4.6) to the host and port specified in *ref*. It opens an *ObjectOutputStream* and uses *writeObject* to marshal *ref*, the method, *m* and the arguments by serializing them to an *ObjectOutputStream*. For the results, it opens an *ObjectIntputStream* and uses *readObject* to get the results from the stream.

At the server end, the dispatcher is given a connection to the client and opens an *ObjectIntputStream* and uses *readObject* to get the arguments sent by the client. Its signature will be:

public void dispatch(Object target, Method aMethod)

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

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

(i) if it is single-threaded, and

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

You can ignore context-switching times. Is there a need for asynchronous RPC if client and server processes are threaded?

5.22 Ans.

i) time per call = calc. args + marshal args + OS send time + message transmission +
OS receive time + unmarshall args + execute server procedure
+ marshall results + OS send time + message transmission +
OS receive time + unmarshal args

= 5 + 4*marshal/unmarshal + 4*OS send/receive + 2*message transmission + execute server procedure
= 5+ 4*0.5 + 4*0.5 + +2*3 + 10 ms = 5+2+2+6+10 =25ms.
Time for two calls = 50 ms.

ii) threaded calls:

client does calc. args + marshal args + OS send time (call 1) = 5+.5=.5 = 6
then calc args + marshal args + OS send time (call 2) = 6
= 12 ms then waits for reply from first call

server gets first call after
message transmission + OS receive time + unmarshal args = 6+ 3+.5+.5
= 10 ms, takes 10+1 to execute, marshal, send at 21 ms
server receives 2nd call before this, but works on it after 21 ms taking
10+1, sends it at 32 ms from start
client receives it 3+1 = 4 ms later i.e. at 36 ms
(message transmission + OS receive time + unmarshal args) later
Time for 2 calls = 36 ms.

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

5.23 Ans..

local reference	*remote reference*	*holders*

The table will have three columns containing the local reference and the remote reference of a remote object and the virtual machines that currently have proxies for that remote object. There will be one row in the table for each remote object exported at the site and one row for each proxy held at the site.

To illustrate its use, suppose that there are 3 sites with the following exported remote objects:
S1: A1, A2, A3 S2: B1, B2; S3: C1;

and that proxies for A1 are held at S2 and S3; a proxy for B1 is held at S3.

Then the tables hold the following information:.

at S1			*at S2*			*at S3*		
local	*remote*	*holders*	*local*	*remote*	*holders*	*local*	*remote*	*holders*
a1	*A1*	*S2, S3*	*b1*	*B1*	*S3*	*c1*	*C1*	
a2	*A2*		*b2*	*B2*		*a1*	*A1proxy*	
a3	*A3*		*a1*	*A1proxy*		*b1*	*B1proxy*	

Now suppose that C1(at S3) invokes a method in B1 causing it to return a reference to B2. The table at S2 adds the holder S3 to the entry for B2 and the table at S3 adds a new entry for the proxy of B2.

Suppose that the proxy for A1 at S3 becomes unreachable. S3 sends a message to S1 and the holder S3 is removed from A1. The proxy for A1 is removed from the table at S3.

5.24 A simpler version of the distributed garbage collection algorithm described in Section 5.2.6 just invokes *addRef* at the site where a remote object lives whenever a proxy is created and *removeRef* whenever a proxy is deleted. Outline all the possible effects of communication and process failures on the algorithm. Suggest how to overcome each of these effects, but without using leases.

5.24 Ans.

AddRef message lost - the owning site doesn't know about the client's proxy and may delete the remote object when it is still needed. (The client does not allow for this failure).

RemoveRef message lost - the owning site doesn't know the remote object has one less user. It may continue to keep the remote object when it is no longer needed.

Process holding a proxy crashes - owning site may continue to keep the remote object when it is no longer needed.

Site owning a remote object crashes. Will not affects garbage collection algorithm

Loss of *addRef* is discussed in the Section 5.2.6.

When a *removeRef* fails, the client can repeat the call until either it succeeds or the owner's failure has been detected.

One solution to a proxy holder crashing is for the owning sites to set failure detectors on holding sites and then remove holders after they are known to have failed.