



Distributed Systems: Concepts and Design

Edition 3

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

Chapter 5 Exercise Solutions

5.1 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?

5.1 Ans.

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

result: output parameters: name of candidate, number of votes

5.2 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.

5.2 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.

5.3 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.

5.3 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.

- 5.4 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?

5.4 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.

- 5.5 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?

5.5 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

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

5.6 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.

-
- 5.7 The *Election* service 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 5.6.

5.7 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.

- 5.8 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).

5.8 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 (...) {}
    }
}
```

-
- 5.9 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 5.3. Give the details of the implementation of one of the methods in this class, which should call the method *doOperation* defined in Figure 4.12.

5.9 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 marshallng method is generated for each argument type used.

-
- 5.10 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.

5.10 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(...){}
}
```

-
- 5.11 Exercise 5.8 required the client to convert *Object* arguments into an array of bytes before invoking *doOperation* and Exercise 5.10 required the dispatcher to convert an array of bytes into an array of *Objects* 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?

5.11 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 *ObjectInputStream* 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 *ObjectInputStream* and uses *readObject* to get the arguments sent by the client. Its signature will be:

- 5.12 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?

5.12 Ans.

- i) time per call = calc. args + marshal args + OS send time + message transmission + OS receive time + unmarshal args + execute server procedure + marshal results + OS send time + message transmission + OS receive time + unmarshal args
= $5 + 4 * \text{marshal/unmarshal} + 4 * \text{OS send/receive} + 2 * \text{message transmission} + \text{execute server procedure}$
= $5 + 4 * 0.5 + 4 * 0.5 + 2 * 3 + 10 \text{ ms} = 5 + 2 + 2 + 6 + 10 = 25 \text{ ms.}$
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.

-
- 5.13 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.

5.13 Ans..

<i>local reference</i>	<i>remote reference</i>	<i>holders</i>

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.

- 5.14 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.

5.14 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 affect 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.

- 5.15 Discuss how to use events and notifications as described in the Jini distributed event specification in the context of the shared whiteboard application. The *RemoteEvent* class is defined as follows in Arnold *et al.* [1999].

```
public class RemoteEvent extends java.util.EventObject {
    public RemoteEvent(Object source, long eventID,
        long seqNum, MarshallableObject handback)
    public Object getSource () {...}
    public long getID() {...}
    public long getSequenceNumber() {...}
    public MarshallableObject getRegistrationObject() {...}
}
```

The first argument of the constructor is a remote object. Notifications inform listeners that an event has occurred but the listeners are responsible for obtaining further details.

5.15 Ans.

Event identifier, *evIDs*. Decided by the *EventGenerator*. Simplest solution is just to have one type of event - the addition of a new *GraphicalObject*. Other event types could for example refer to deletion of a *GraphicalObject*.

Clients need to be notified of the remote object reference of each new *GraphicalObject* that is added to the server. Suppose that an object in the server is the *EventGenerator*. It could implement the *EventGenerator* interface and provide the *register* operation, or it could be done more simply

```
e.g. addListener(RemoteEventListener listener, long evID)
    // add this listener to a vector of listeners
```

This would be better if *Leases* were used to avoid dealing with lost clients.

The *newShape* method of *shapeListServant* (Figure 5.14) could be the event generator. It will notify all of the *EventListeners* that have registered with it, each time a new *GraphicalObject* is added. e.g.

```
RemoteEvent event = new RemoteEvent(this, ADD_EVENT, version, null)
for all listeners in the vector
    listener.notify(event)
```

Each client creates an *RemoteEventListener* for receiving notifications of events and then registers interest in events with the server, passing the *EventListener* as argument.

```
class MyListener implements RemoteEventListener {
    public MyListener() throws RemoteException{
    }

    public void notify(RemoteEvent event) throws UnknownEventException,
        RemoteException {
        Object source = getSource();
        long id = event.getID();
        long version = event.getSequenceNumber();
        // get the newly created GraphicalObject from the server
    }
}
```

Then to become a listener (add the following to the client program shown in Figure 5.15):

```
sList.addListener(new MyListener(), ADD_EVENT);
```

The client getting the *newGraphicalObject* needs to be able to get it directly from the version number, rather than by getting the list of *Shapes* and then getting the *GraphicalObject*. The interface to *ShapeList* could be amended to allow this.

-
- 5.16 Suggest a design for a notification mailbox service which is intended to store notifications on behalf of multiple subscribers, allowing subscribers to specify when they require notifications to be delivered. Explain how subscribers that are not always active can make use of the service you describe. How will the service deal with subscribers that crash while they have delivery turned on?

5.16 Ans.

The Mailbox service will provide an interface allowing a client to register interest in another object. The client will need to know the *RemoteEventListener* provided by the Mailbox service so that notifications may be passed from event generators to the *RemoteEventListener* and then on to the client. The client will also need a means of interacting with the Mailbox service so as to turn delivery on and off. Therefore define register as follows:

```
Registration register() ...
```

The result is a reference to a remote object whose methods enable the client to get a reference to a *RemoteEventListener* and to turn delivery on and off.

To use the Mailbox service, the client registers with it and receives a *Registration* object, which it saves in a file. It registers the *RemoteEventListener* provided by the Mailbox service with all of the *EventGenerators* whose events it wants to have notification of. If the client crashes, it can restore the *Registration* object when it restarts. Whenever it wants to receive events it turns delivery on and when it does not want them it turns delivery off.

The design should make it possible to specify a lease for each subscriber.

-
- 5.17 Explain how a forwarding observer may be used to enhance the reliability and performance of objects of interest in an event service.

5.17 Ans.

Reliability:

The forwarding observer can retry notifications that fail at intervals of time.

If the forwarding observer is on the same computer as the object of interest, then the two could not fail independently.

Performance:

The forwarding observer can optimize multicast protocols to subscribers.

In Jini it could deal with renewing leases.

-
- 5.18 Suggest ways in which observers can be used to improve the reliability or performance of your solution to Exercise 5.13.

5.18 Ans.

The server can be relieved of saving information about all of the clients' interests by creating a forwarding agent on the same computer. the forwarding agent could use a multicast protocol to send notifications to the clients. IP multicast would do since it is not crucial that every notification be received. A missed version number can be rectified as soon as another one is received.