1.

Class RequestMessage{  
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;}  
}

2.

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);  
}  
}

3. 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 {... }  
}  
}

4. 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. n=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.

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

6.

Operating System provide different interface to communication protocols. Interface of request- reply protocol hides these Interface. - In spite of the fact that internet protocol is widely available some computer networks may provide other protocols. - Request-response protocol can easily combined with other protocols.

7.

(i) Pressing the lift request button is an idempotent operation. (ii) The operation to write data to a file can be defined as in Unix where each write is applied at the read-write pointer, in which case the operation is not idempotent; or 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. (iii) 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.

8. 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.

9.

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.

10. 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: 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.

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

11.

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

12.   
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.

13.

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.

14. Maybe call semantics is obviously inadequate for vote! That the voter’s number can be 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.

15. 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

16.