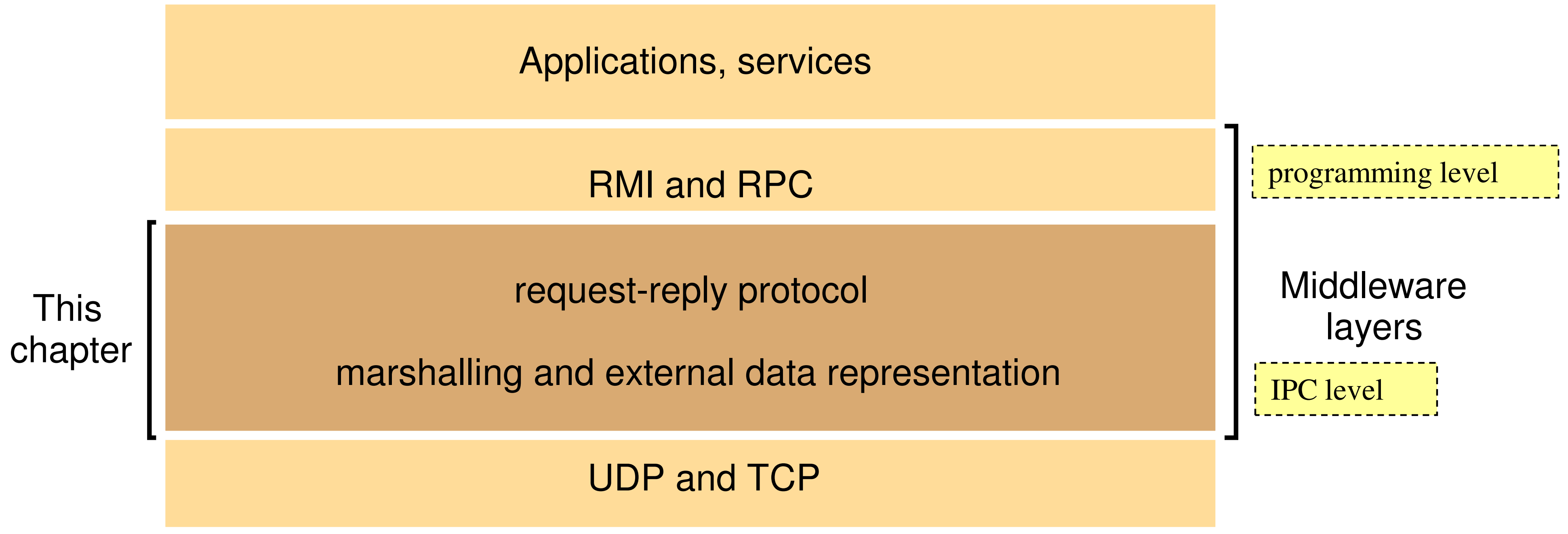
Chapter 4

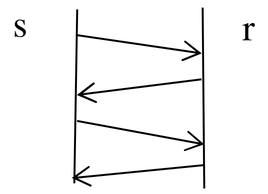
Interprocess Communication

Middleware Layers

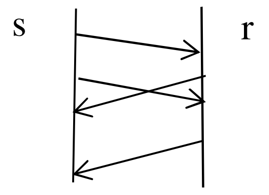


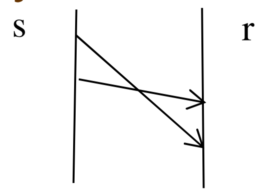
Message Passing

- A message passing is supported by
  - Operation send(destination, message)
  - Operation receive(destination, message)
- Each destination has a queue to receive and fetch messages.
- Synchronous communication
  - After issuing send(), the sending process is blocked until the corresponding receive() is issued.
  - When issuing receive(), the receiving process blocks until a message arrives.



- Asynchronous communication
  - After issuing send(), the sending process can proceed as soon as the message has been copied to a local buffer.
  - receive() can be either blocking or non-blocking.
  - Non-blocking receive() needs polling or interrupt to identify that the buffer has been filled, and may have to manage out-of-flow-of-control messages.





Message Destination

- In Internet protocol, a destination can be defined by (Internet address of the host, local port number of the process).
  - A port has exactly one receiver but can have many senders.
  - A process may receive from multiple ports.

location-dependent

- If a client uses a fixed Internet address to refer to a service, the service must always run on that computer for the address to remain valid.

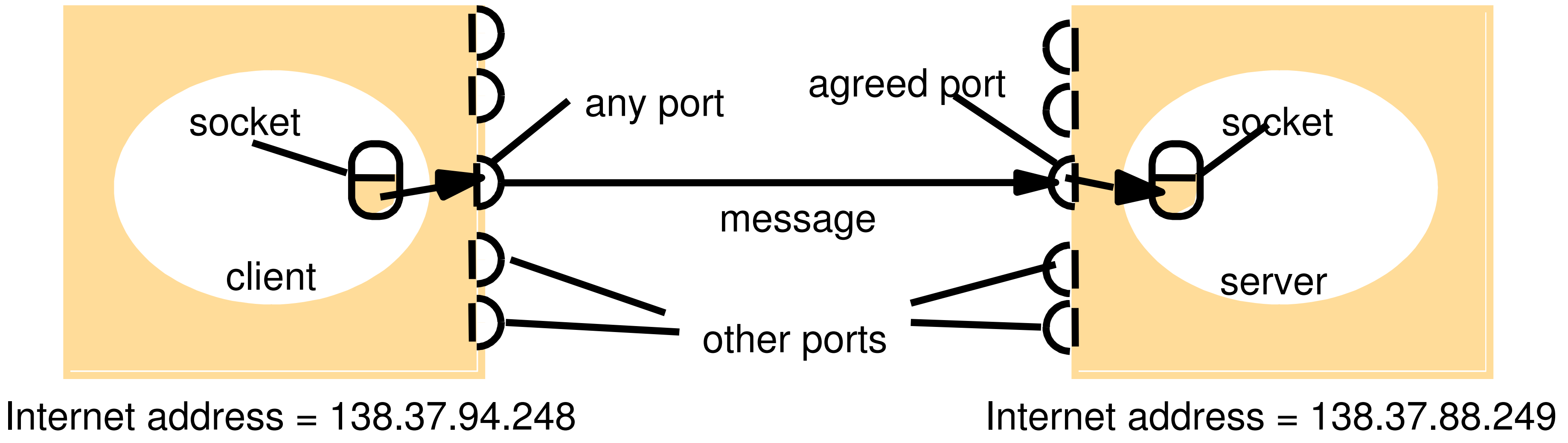
location-independent

- The client may refer to a service by name and use a name service to map the name into server location at run time.
  - Java API (class InetAddress) uses DNS to get Internet address:

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmw.ac.uk");

Socket

- Socket abstraction provides an indirect reference for communication between processes.
- A message is transmitted between a socket in one process and a socket in another process.
- A receiving process must have its socket bound to a local port and the Internet address of its host.
- A process may use the same socket for sending and receiving messages.
- Each socket is associated with a particular transport protocol – UDP or TCP.


socket
any port
agreed port
socket
message
client
server
other ports
Internet address = 138.37.94.248
Internet address = 138.37.88.249

UDP (User Datagram Protocol)

- UDP provides a message passing abstraction.
- One process sends a single message and another receives it.
- A packet containing a message is called datagram.
- UDP is not reliable.
  - There is no acknowledgement or retry; if failure occurs, the message may not arrive.
- The receiving process specifies an array of bytes for the socket that is bound to the destination port.
- UDP uses non-blocking send() and blocking receive().
  - Timeout can be set on the receiving socket.
  - The receiving process may use separate threads to do other work and to wait for messages from other processes.
  - receive() returns Internet address and local port of the sender.

UDP client sends a message and receives a reply

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

UDP server repeatedly receives a request and sends it back

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

Failure Handling in UDP

- UDP uses checksum to check corrupted data.
- Messages will be dropped if they are corrupted or the buffer is full at the source or destination (c.f. omission failure).
- Messages may be delivered out of sender order.
- Applications using UDP have to provide their own check to achieve reliable communication:
  - Use of acknowledgement to the sender to distinguish between successful delivery and omission failure.
- UDP is acceptable for applications that can tolerate occasional omission failure or require just a single client-server request and reply (e.g. DNS lookup).
- It does not suffer from overheads associated with guaranteed message delivery (as in TCP).

TCP (Transport Control Protocol) (1)

- TCP provides an abstraction of a two-way stream of bytes with no message boundaries.
- The application can choose how much data to be read/written from/to the stream.
  - Underlying implementation of a TCP stream decides how much data to collect before sending it as one or more packets.
- TCP controls flow of the stream.
  - When the writer is too fast, it is blocked until the reader has consumed sufficient data.

TCP (2)

- Communicating processes need to establish a connection first (connection-oriented).

To IP address and port number

  - A process (taking a client role) makes a connect request to another process (taking the server role). The server makes an accept request to the client. After that they can be peers.

Write to/read from stream associated with the connection

  - Once a connection is established, reading and writing no longer need Internet addresses and ports.
  - Each socket in client and server connection is associated with an input stream and an output stream.
  - When a server accepts a connection, it uses a separate thread to communicate with each client.

TCP (3)

- The sender and receiver have to agree on the contents of the data stream (e.g. an int followed by a double)
- The receiver may block if the input buffer is empty. The sender may block by the flow control mechanism.
- Services run over TCP connection with reserved port number includes HTTP, FTP, Telnet, SMTP.

TCP client makes connection to server, sends request, and receives reply

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

in and out are associated with socket

TCP server accepts connection from each client, and echoes client’s message (1)

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

// continue on the next slide
```

TCP server accepts connection from each client, and echoes client’s message (2)

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

Failure Handling in TCP (1)

- TCP uses checksum to detect and reject corrupted packets.
- TCP uses sequence numbers to detect and reject duplicate packets, and can reorder packets that arrive out of sender order.
- TCP uses timeout and retry to deal with lost packets.
  - The sender keeps record of the packets sent and the receiver acknowledges all the arrivals.
  - The sender resends the message if acknowledgement is not received within a timeout.
  - If packet loss passes some limit or the network is severely congested, TCP software will declare the connection to be broken.
- When the connection is broken
  - The process cannot distinguish between network failure and failure of the process at the other end.
  - The process cannot tell whether the recent message has been received or not.

Failure Handling in TCP (2)

- Failure handling makes communication more reliable but it incurs overhead:
  - The need to store state information at source and destination
  - Transmission of extra messages
  - Latency for the sender

Complication in Data Transmission

- Information in running programs is represented as data structures or sets of interconnected objects whereas information in messages is sequences of bytes.
- Not all computers store data values in the same format.
  - Different byte ordering for integers (big-endian and little-endian)
  - Different representation for floating-point numbers
  - ASCII character uses 1 byte whereas Unicode uses 2 bytes.

Marshalling and Unmarshalling

- Data structures must be flattened (converted to sequence of bytes) and individual primitive data values must be converted to an agreed format before transmission, and rebuilt on arrival.
- **External data representation** - An agreed standard for the representation of data structures and primitive values.

XDR

- **Marshalling** – The process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- **Unmarshalling** – The process of disassembling them on arrival to produce an equivalent collection of data items at the destination.
- Marshalled data may be in binary form or ASCII text.
- Marshalling and unmarshalling are carried out by middleware.

Example: CDR Message of CORBA Middleware

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

The flattened form represents a *Person* struct with value: {‘Smith’, ‘London’, 1934}

Marshalling and Unmarshalling Operations in CORBA

- Marshalling and unmarshalling operations are generated automatically from the type definitions described in CORBA IDL:

```
marshallPerson(p) =
   marshallString(p.name)+
   marshallString(p.place)+
   marshallLong(p.year)
```

```
unmarshallPerson(marshalledp) =
   unmarshallString(marshalledname)+
   unmarshallString(marshalledplace)+
   unmarshallLong(marshalledyear)
```

```
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

- The generation is by CORBA interface compiler. See next chapter.

Example: Java Object Serialization

- Serialization – Flattening objects into a serial form that is suitable for storing on disk or transmitting in a message.
- Deserialization – Storing the state of objects from their serialized form.
  - There is no need to generate special marshalling or unmarshalling operations for each type of object as for CORBA; the operations are generic.
- All objects that an object to be serialized references to will be serialized as well.
  - References are serialized as handles.

A Serialized Java Object

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

The true serialized form contains additional type markers; h0 and h1 are handles

```
public class Person implements Serializable {
   private String name;
   private String place;
   private int year;
   //followed by methods
}
```

Remote Object Reference

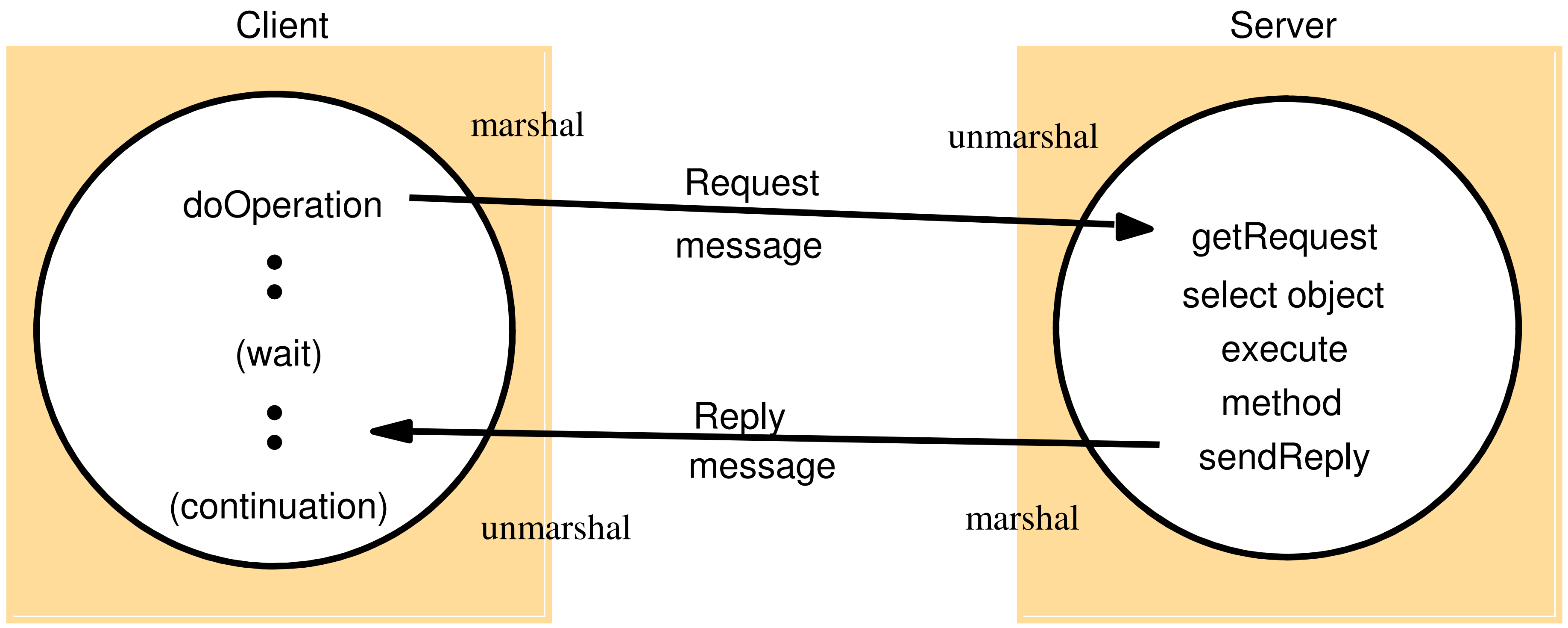
- Remote object reference is an identifier for a remote object to which an invocation is made.
- It is valid and unique throughout a distributed system.

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

- In the simplest form, remote object reference is valid as long as the process that creates the remote object continues to run.

Request-Reply Communication

- Request-reply communication is synchronous.
- It can be reliable as the reply is effectively an acknowledgement to the request.



Operations of the request-reply protocol

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

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

public byte[] getRequest ();

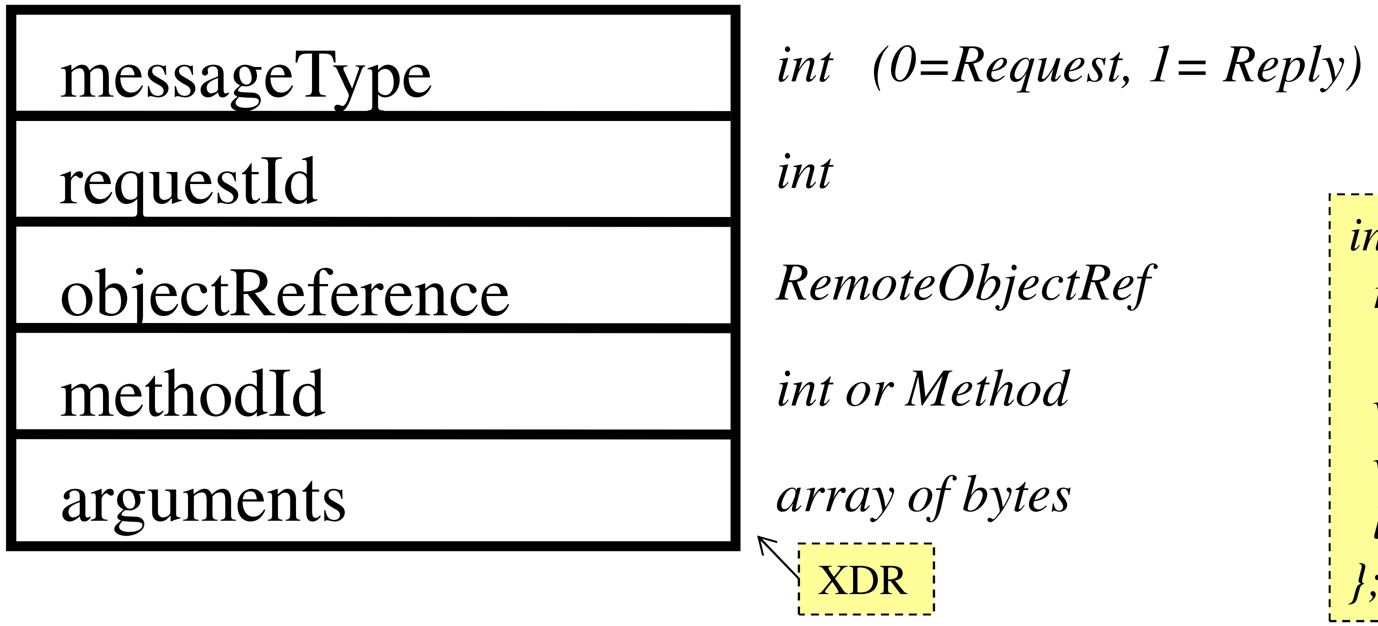
acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.

- The client calling doOperation() marshals the argument into an array of bytes and unmarshals the results from the array of bytes that is returned.
- After sending the request, doOperation() invokes receive() to get a reply.

Request-Reply Message Structure



```
interface PersonList {
  readonly attribute string
     listname;            1
  void addPerson(...) ;      2
  void getPerson(...);      3
  long number();       4
};
```

- requestID is taken from an increasing sequence of integers by the sending process.
- For reliable communication, unique messageID will be used; it consists of requestID and sender's processID (Internet address and port number).
- methodID could be numbered 1,2,3,… according to the interface, or it could be a representation of the method for the language that supports reflection (e.g. an instance of Method in Java).

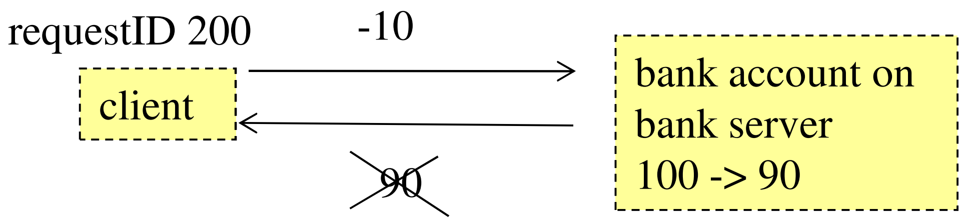
Failure Handling in Request-Reply Protocol (1)

- The protocol suffers from the failures of the transport protocol over which it is implemented (e.g. omission failure over UDP).
- Timeout is used to handle lost request or reply.
  - doOperation() sends request repeatedly until either it gets a reply or it is sure that the delay is due to lack of response from the server rather than to lost messages.
  - doOperation() will raise exception to the client that no result is received.
- The protocol can recognize successive messages of the same messageID (duplicate requests) to avoid the server executing the same request more than once.
  - The server can ignore the duplicate request if it has not sent the reply previously since it will transmit the reply eventually when it has finished executing the operation.

Failure Handling in Request-Reply Protocol (2)

- When receiving duplicate requests, some servers can execute their operations more than once and obtain the same results each time (e.g. add an item to a set).
  - **An idempotent operation** can be performed repeatedly with the same effect as if it has been performed exactly once.
  - Servers with non-idempotent operations need to take special measure to avoid executing operations more than once (e.g. add an item to a sequence).

Non-idempotent: withdraw 10
Idempotent: update 90



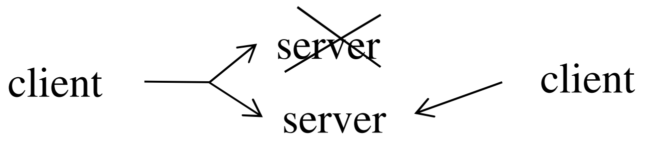
requestID	reply	client
200	90	IP+port

Failure Handling in Request-Reply Protocol (3)

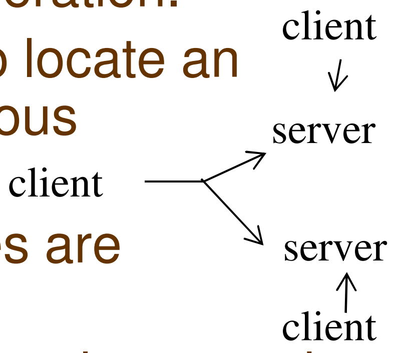
- Servers may retransmit the reply without re-executing duplicate requests if the history of reply messages is kept.
  - A history record maintains the requestID, the reply message, and the client address.
  - The reply history incurs memory cost.
  - The history contains only the last reply message sent to each client, as each request from the client is effectively an acknowledgement of its previous reply.
  - Anyway, there is a large number of clients and clients terminate without acknowledging their last replies. Messages in the history will then be discarded after a period of time.

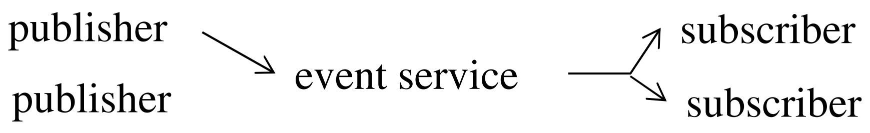
Group Communication

- Multicast – An operation that sends a single message from one process to each member of the group of processes. Usually the membership of the group is transparent to the sender.
- Use of multicast



  - Fault tolerance based on replicated service – Client requests are multicast to the members, each performs an identical operation.
  - Finding an object or a service – Multicast can be used to locate an object or a service (e.g. a discovery service in spontaneous networking).
  - Better performance through replicated data – New values are multicast to update the replicas.
  - Propagation of event notifications – Events are multicast to interested processes (e.g. a new message is posted on a newsgroup).





$$\begin{tikzcd}
& & & \text{recipient1} \\
\text{sender} \arrow[r] & \text{Thailand} \arrow[r] & \text{Finland} \arrow[ur] \arrow[r] \arrow[dr] & \text{recipient2} \\
& & & \text{recipient3}
\end{tikzcd}$$

IP Multicast Implementation

- The sender can transmit a single IP packet to a set of computers that form a multicast group.
- The sender is unaware of the identities of individual recipients and the size of the group.
- Computers can join or leave the groups.
- IP multicast is available via UDP.
- A computer belongs to a multicast group when one or more of its processes has sockets that belong to that group.
- IP packets can be multicast both on a local network and on Internet.
  - Local multicasts use multicast capability of the local network such as Ethernet.
  - Multicasts in Internet use multicast routers which forward datagrams to routers on other networks, where they in turn multicast to local members.
- Multicast addresses may be allocated permanently (224.0.0.1 to 224.0.0.255) or temporarily.

Multicast peer joins a group and sends and receives datagrams (1)

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);


         // continued on the next slide
```

no need to join if send only

Multicast peer joins a group and sends and receives datagrams (2)

```
                // get messages from others in group
                        byte[] buffer = new byte[1000];
                        for(int i=0; i< 3; i++) {
                            DatagramPacket messageIn =
                                    new DatagramPacket(buffer, buffer.length);
                            s.receive(messageIn);
                            System.out.println("Received:" + new String(messageIn.getData()));
                        }
                        s.leaveGroup(group);
                }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
               }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(s != null) s.close();}
    }
}
```

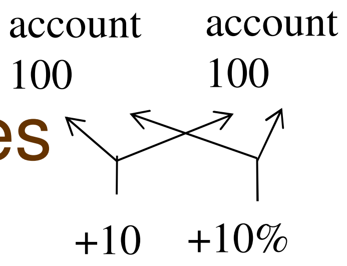
Failures in IP Multicast

- IP multicast suffers from omission failure and out-of-order messages by UDP.
  - A member may drop a message because its buffer is full.
  - A multicast router may fail or the message to it may be lost.
  - Packets from a single sender may not arrive at the members in the same order.
  - Packets from different senders may arrive at the members in different order.

*** Consistency ***

Effects on Failures in IP Multicast (1)

- Some applications require that multicast must be reliable while some are not serious.
- On fault tolerance based on replicated services
  - Either all or none of the replicas are to receive the multicast message in order to maintain consistency between replicas.
  - In most cases, replicas have to receive the messages in the same order as one another (c.f. totally ordered multicast).

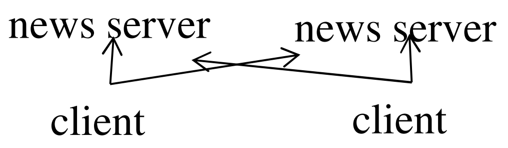


- On finding an object or a service
  - Having members missing a request is not a serious issue when locating a service as long as some member receives it and responds.

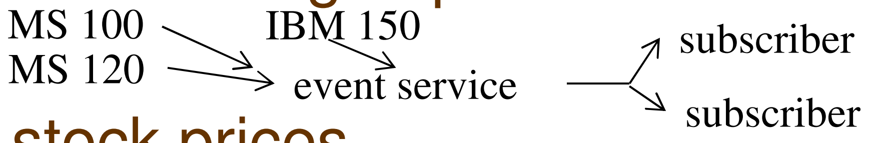
*** Consistency ***

Effects on Failures in IP Multicast (2)

- On better performance through replicated data
  - The effect of lost messages and ordering depends on the importance of all replicas being up-to-date.
  - Replicas of newsgroup are not necessarily consistent at one time and users can cope with messages that appear in different order.



- On propagation of event notifications
  - The effect of lost messages and ordering depends on the applications.



  - It is serious for notification of stock prices.
  - It is not serious for frequent announcement of the discovery service available in spontaneous networking.

Facebook