The UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

CS4/MSc

Distributed Systems

Jane Hillston

jeh@dcs.ed.ac.uk

Room 1604

(Lecture 3: Inter-process Communication, 14th January 2002)

Ports and Sockets (1)

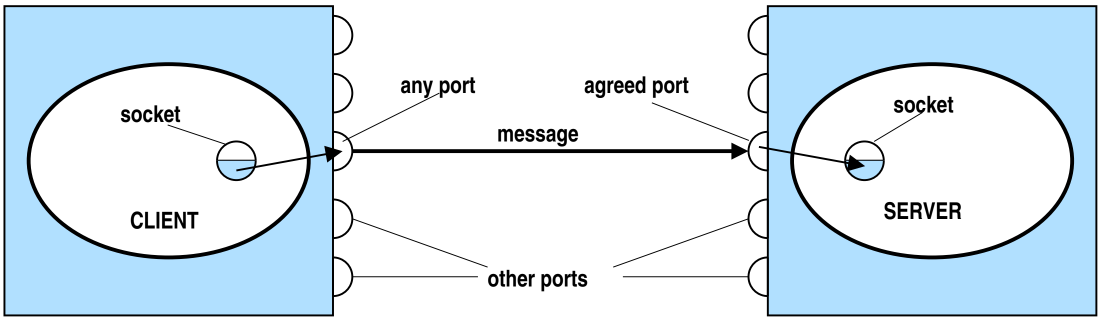
Communication between processes (as opposed to hosts) is made between *ports*.

- A port is represented by an integer $(0 - 2^{16})$: some values/ports have specific use but others are available for general use.
- Each port corresponds to a single receiving process, but each process may have more than one port at which it receives.
- Any number of senders can send messages to a port.

Sockets are software abstractions of ports used within running processes.

- Messages are sent between a pair of sockets, each of which is attached or *bound* to a port. This binding may be automatic (Java) or may need to be done explicitly (BSD UNIX).
- To send a message to a socket the sender must know the IP address of the host and the port number to which the socket is bound.
- Each socket is associated with a particular protocol, i.e. UDP (datagrams) or TCP (streams).

Ports and Sockets (2)



- Different programming languages use different levels of abstraction for sockets.
- Java provides three basic socket classes: `DatagramSocket` for datagram communication, which has a specialised subclass `MulticastSocket` for group communication; and `Socket` and `ServerSocket` for stream communication.
- `DatagramPacket` class is provided to package an array of bytes.

UDP datagram communication

- Datagram communication is unacknowledged and unreliable.
- A datagram is communicated between processes when one process `sends` it and the other `receives` it.
- Sending is non-blocking but receiving is blocking although time-outs can be set.
- Arriving messages are placed in a queue bound to the receiver's port.
- The `receive` method returns the Internet address and port number of the sender in addition to the datagram content.
- The `receive` message does not specify who the message should be received from, it will retrieve any message which has arrived at the port. However it is possible to bind a socket so that it only sends to or receives messages from a particular remote port.
- The receiver must provide an array of bytes into which the message is placed when `received`. Message will be truncated if necessary.

UDP Example – Client

```
import java.net.*; import java.io.*;
public class UDPClient{
  public static void main(String args[]){
    // args give message contents and server hostname
    try {
      Datagram 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()));
      aSocket.close();
      }
      catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
      catch (IOException e){System.out.println("IO: " + e.getMessage());}
  }
}
```

UDP Example – Server

```
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();}
  }
}
```

TCP Stream Communication

- When TCP is to be used the abstraction of a stream of bytes is used to effect communication.
- Connection must first be established and at this stage distinction is made between a requester/client (`Socket`) and a receiver/server (`ServerSocket`). The client sends a `connect` request.
- When the server `accepts` the request a new stream `Socket` is created for communication with this client, whilst the `ServerSocket` keeps listening for new `connect` requests.
- The established pair of sockets then support streams in both directions.
- The sender writes onto its output stream via its socket and the receiver reads from its input stream via its socket.
- When a process `closes` a socket, any remaining is transmitted together with an indicator that the connection is now broken.

TCP Example – Client

```
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) {/*close failed*/}
    }
  }
}
```

TCP Example – Server (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());}
  }
}
```

TCP Example – Server (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*/}}
  }
}
```

Data Marshalling

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. *Unmarshalling* is the complementary process of re-assembling the data structure at the destination. Heterogeneity means that data formats at the two ends of communication may not agree. The problem is overcome by using an external data representation format such as CORBA's CDR or SUN NFS's XDR. The transmission format need not be binary: HTTP uses ASCII text.

Marshalling should be carried out by a middleware layer, as it would be difficult to achieve accuracy and efficiency if programmed by hand. In CORBA the types of the data to be marshalled must be specified in order to generate the marshalling and unmarshalling methods (IDL). In contrast Java makes use of reflection and information within the serialized data itself.

Java Serialization and Reflection

- The Java interface `Serializable` has no methods but is used to mark classes which may be serialized and deserialized: that is converted into a format suitable for transmission or for storing on disk, and back.
- Serialization is achieved by creating an instance of the class `ObjectOutputStream` and invoke its `writeObject` method passing the object concerned as an argument.
- Deserialization is achieved by creating an instance of the class `ObjectInputStream` and using its `readObject` method to reconstruct the object.
- This is usually not done by the application programmer, however (cf. Java RMI) but automatically in middleware.
- The Java proprerty of *reflection* means that it is possible to probe a class to find out about its methods and state variables. This means that prior publication of types to be marshalled is not necessary.

Remote Object References

- When a distributed object system is used, a client needs to be able to uniquely identify the remote object on which it wishes to invoke a method. This is achieved using a *remote object reference*.
- A common approach is to incorporate the IP address name of the host and port number of the hosting process into the reference of the object.
- The time of creation may also be included, since the time and port together will uniquely identify a process.
- An object number may be used to differentiate different instantiations of transient objects.

32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface

Client-server Communication

Client-server communication requires some form of *request-reply* protocol. For example:

The three-message protocol or RRA protocol In this case the client makes a request, the server responds and the client acknowledges the response. Requests are numbered and this allows some history to be stored to increase reliability.

The single-shot protocol or R or RR protocol This is suited to applications where requests are *idempotent* – repeating the requested activity will have no adverse effect. A request is sent; a response is sent if it the operation requires it.

These can be implemented on top of UDP or TCP, according to the needs of the application. HTTP is an example, implemented on TCP.

Group Communication

Multicast is the term used to denote communication to a pre-defined group of processes and is used when it is more appropriate to send a message to a group than to processes individually. It is closely linked to replication which may be used to enhance reliability and/or performance.

- IP multicast is built on top of IP, and is designed for communication between groups of hosts (not ports at this level of the protocol stack).
- A specific class of IP addresses is reserved for multicast groups, on a temporary or a permanent basis.
- IP multicast is only accessible via UDP. Datagrams are sent to a multicast IP address and ordinary port numbers.
- When a multicast message arrives at a host, copies are forwarded to all the local sockets that have joined the specified multicast address and are bound to the specified port number. NB. in this circumstance sockets may share a port.