Transport layer protocols

Lecture 15:
Operating Systems and Networks
Behzad Bordbar
Interprocess communication

- Synchronous and asynchronous comm.
- Message destination
- Reliability
- Ordering

Client

- doOperation
  - (wait)
  - (continuation)

Server

- getRequest
  - select object
  - execute method
  - sendReply

Request message

Reply message
Asynchronous vs. Synchronous

• Synchronize communication: sender and receiver
• synchronize on every message, i.e. *blocking* operations
• Asynchronous
  ❑ send is non-blocking
  ❑ receive can be blocking/non-blocking

Which one is better?
Message destination: Socket + Port

Socket = Internet address + port number. Only one receiver but multiple senders per port.
Sockets

• Characteristics:
  – **endpoint** for inter-process communication
  – message transmission **between sockets**
  – socket associated with **either UDP or TCP**
  – processes **bound** to sockets, can use **multiple** ports

• Implementations
  – originally **BSD Unix**, but available in **Linux, Windows,**...
  – here **Java API** for Internet programming
Operations of Request-Reply

• **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.
Java API for Internet addresses

• Class InetAddress
  – uses DNS (Domain Name System)

```java
InetAddress aC = InetAddress.getByName("gromit.cs.bham.ac.uk");
```

– throws UnknownHostException
– encapsulates detail of IP address (4 bytes for IPv4 and 16 bytes for IPv6)
Remote Object Reference

- An identifier for an object that is valid throughout the distributed system
  - must be **unique**
  - may be passed as argument, hence need **external** representation

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
</tr>
</tbody>
</table>
Reliability

- Reliable communication:
  - messages are guaranteed to be delivered despite a
  - ‘reasonable’ number of packets being dropped or lost

Unreliable communication:
- messages are not guaranteed to be delivered in the face of even a single packet dropped or lost
- >>>> Failure
Failure in point to point comm.

• DSs expected to continue if failure has occurred:
  – message failed to arrive
  – process stopped (and others may detect this)
  – process crashed (and others cannot detect this)

• Types of failures
  – benign
    • omission, stopping, timing/performance
  – arbitrary (called Byzantine)
    • corrupt message, wrong method called, wrong result
# Omission and arbitrary failures

<table>
<thead>
<tr>
<th>Class of failure</th>
<th>Affects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail-stop</td>
<td>process</td>
<td>Process halts and remains halted. Other processes may detect this state.</td>
</tr>
<tr>
<td>Crash</td>
<td>process</td>
<td>Process halts and remains halted. Other processes may not be able to detect this state.</td>
</tr>
<tr>
<td>Omission</td>
<td>channel</td>
<td>A message inserted in an outgoing message buffer never arrives at the other end’s incoming message buffer.</td>
</tr>
<tr>
<td>Send-omission</td>
<td>process</td>
<td>A process completes a send but the message is not put in its outgoing message buffer.</td>
</tr>
<tr>
<td>Receive-omission</td>
<td>process</td>
<td>A message is put in a process’s incoming message buffer, but that process does not receive it.</td>
</tr>
<tr>
<td>Arbitrary (Byzantine)</td>
<td>process or channel</td>
<td>Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step</td>
</tr>
</tbody>
</table>
Timing

- failure can be caused because of timing:
  - No global time
  - Computer clocks
    - may have varying drift rate
    - rely on GPS radio signals (not always reliable), or synchronise via clock synchronisation algorithms
  - Event ordering (message sending, arrival)
    - carry timestamps
    - may arrive in wrong order due to transmission delays (cf email)
Types of interaction

- **Synchronous interaction model:**
  - known upper/lower **bounds** on execution **speeds**, message transmission **delays** and clock **drift** rates
  - more difficult to build, conceptually simpler model

- **Asynchronous interaction model** (more common, cf Internet, more general):
  - arbitrary process execution **speeds**, message transmission **delays** and clock **drift** rates
  - some problems **impossible** to solve (e.g. agreement)
  - if solution valid for asynchronous then also valid for synchronous.
Request-Reply Communication

Client

- doOperation
  - (wait)
  - (continuation)

Server

- getRequest
  - select object
  - execute
  - method
  - sendReply

Request message

Reply message
Java API for Datagram Comms

- Simple send/receive, with messages possibly lost/out of order
- Class *DatagramPacket*

<table>
<thead>
<tr>
<th>message (=array of bytes)</th>
<th>message length</th>
<th>Internet addr</th>
<th>port no</th>
</tr>
</thead>
</table>

- packets may be transmitted between sockets
- packets truncated if too long
- provides `getData`, `getPort`, `getAddress`
Java API for Datagram Comms

• Class `DatagramSocket`
  – `socket constructor` (returns free port if no arg)
  – `send` a ` DatagramPacket`, non-blocking
  – `receive` ` DatagramPacket`, blocking
  – `setSoTimeout` (receive blocks for time T and throws `InterruptedException`)
  – `close` `DatagramSocket`
  – throws `SocketAddress` if port unknown or in use
In the example...

• UDP Client
  – sends a message and gets a reply

• UDP Server
  – repeatedly receives a request and sends it back to the client

See textbook website for Java code
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);
} catch (SocketException e) {System.out.println("Socket: " + e.getMessage());
} catch (IOException e) {System.out.println("IO: " + e.getMessage());
} finally {
    if(aSocket != null)
        aSocket.close();
}
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();}
    }
}
Java API for Data Stream Comms

• Data stream abstraction
  – attempts to match the data between sender/receiver
  – marshaling/unmarshaling

• Class Socket
  – used by processes with a connection
  – connect, request sent from client
  – accept, issued from server; waits for a connect request, blocked if none available

See the API
Java API for Data Stream Comms

• Class *ServerSocket*
  – socket constructor (for listening at a server port)
  – *getInputStream, getOutputStream*
  – *DataInputStream, DataOutputStream*
    (automatic marshaling/unmarshaling)
  – *close* to close a socket
  – raises *UnknownHost, IOException*, etc
In the next example...

• TCP Client
  – makes connection, sends a request and receives a reply

• TCP Server
  – makes a connection for each client and then echoes the client’s request
public class TCPCClient {
    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 Sec 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data);
            s.close();
        }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)….}
    }
}
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());}
    }
}

// this figure continues on the next slide
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)…..}
    }
}
Group Communication:

- Multicast: an operation that sends a single message from one process to each of the members of a group of processes
- Fault tolerance based on replicated services
- Finding the discovery servers in spontaneous networking
- Better performance through replicated data
- Propagation of event notifications
IP multicast

- multicast group is specified by a class D Internet address
- membership is dynamic, to join make a socket
- programs using multicast use UDP and send datagrams to multicast addresses and (ordinary) port
(For example of Java code see book)
Summary

• some of the main issues design of Distributed systems: timing, failure and interaction architecture
• Request-Reply Communication
  – sockets and ports, API 4 Internet address, remote object reference
• TCP and UDP client server programming