Lecture 4:
Inter-Process Communication
Overview

• Message passing
  – send, receive, group communication
  – synchronous versus asynchronous
  – types of failure, consequences
  – socket abstraction

• Java API for sockets
  – connectionless communication (UDP)
  – connection-oriented communication (TCP)
API for Internet programming...

- Applications, services
- RMI and RPC
- Request-reply protocol
- Marshalling and external data representation
- UDP and TCP

This lecture

Middleware layers
Inter-process communication

- Distributed systems
  - consist of components (processes, objects) which communicate in order to co-operate and synchronise
  - rely on message passing since no shared memory

- Middleware provides programming language support, hence
  - does not support low-level untyped data primitives (this is the function of operating system)
  - implements higher-level language primitives + typed data
Inter-process communication

Possibly several processes on each host (use ports).
Send and receive primitives.
Communication service types

- **Connectionless**: UDP
  - ‘send and pray’ unreliable delivery
  - efficient and easy to implement

- **Connection-oriented**: TCP
  - with basic reliability guarantees
  - less efficient, memory and time overhead for error correction
Connectionless service

UDP (User Datagram Protocol)

- messages possibly lost, duplicated, delivered out of order, without telling the user
- maintains no state information, so cannot detect lost, duplicate or out-of-order messages
- each message contains source and destination address
- may discard corrupted messages due to no error correction (simple checksum) or congestion

Used e.g. for DNS (Domain Name System) or RIP.
Connection-oriented service

TCP (Transmission Control Protocol)

- establishes **data stream** connection to ensure **reliable**, in-sequence delivery
- error checking and reporting to both ends
- attempts to **match speeds** (timeouts, buffering)
- **sliding window**: state information includes
  - unacknowledged messages
  - message sequence numbers
  - flow control information (matching the speeds)

**Used e.g. for HTTP, FTP, SMTP on Internet.**
Timing issues in DSs

• **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)
Failure issues in DSs

• 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 <em>send</em>, 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>
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.
Send and receive

• **Send**
  – send a message to a socket bound to a process
  – can be blocking or non-blocking

• **Receive**
  – receive a message on a socket
  – can be blocking or non-blocking

• **Broadcast/multicast**
  – send to all processes/all processes in a group
Receive

• **Blocked** receive
  – destination process **blocked** until message arrives
  – most commonly used

• **Variations**
  – **conditional receive** (continue until receiving indication that message arrived or polling)
  – **timeout**
  – **selective receive** (wait for message from one of a number of ports)
Asynchronous Send

• Characteristics:
  – unblocked (process continues after the message sent out)
  – buffering needed (at receive end)
  – mostly used with blocking receive
  – usable for multicast
  – efficient implementation

• Problems
  – buffer overflow
  – error reporting (difficult to match error with message)

Maps closely onto connectionless service.
Synchronous Send

• Characteristics:
  – **blocked** (sender suspended until message received)
  – synchronisation point for both sender & receiver
  – easier to reason about

• Problems
  – failure and indefinite delay causes **indefinite blocking** (use timeout)
  – multicasting/broadcasting **not** supported
  – implementation more complex

Maps closely onto connection-oriented service.
Sockets and ports

Socket = Internet address + port number.
Only one receiver but multiple senders per port.
Disadvantages: location dependence (but see Mach study, chap 18)
Advantages: several points of entry to process.
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
  – no port sharing unless IP multicast

• Implementations
  – originally BSD Unix, but available in Linux, Windows,…
  – here Java API for Internet programming
Client-Server Interaction

• Request-reply:
  – port must be known to client processes (usually published on a server)
  – client has a private port to receive replies

• Other schemes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Messages sent by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Client</td>
</tr>
<tr>
<td>R</td>
<td>Request</td>
</tr>
<tr>
<td>RR</td>
<td>Request</td>
</tr>
<tr>
<td>RRA</td>
<td>Request</td>
</tr>
</tbody>
</table>
Request-Reply Communication

Client

doOperation

(wait)

(continuation)

Server

getRequest

select object

execute method

sendReply

Request message

Reply message
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)

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>
<tr>
<td>interface of remote object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 `InterruptedIOException`)
  - *close* `DatagramSocket`
    - throws `SocketException` if port unknown or in use

See textbook site cdk3.net/ipc for complete code.
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();}
}
UDP server example

```java
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();}
    }

10 January, 2002
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 textbook site cdk3.net/ipc for complete code.
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 `UnknownHostException`, `IOException`, etc
Data Marshaling/Unmarshaling

- **Marshaling** (=conversion of data into machine-independent format)
  - necessary due to heterogeneity & varying formats of internal data representation

- **Approaches**
  - CORBA CDR (Common Data Representation)
  - Java object serialisation, cf DataInputStream, DataOutputStream on previous and next slides
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 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 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)….}
TCP server example

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(Exception 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)…..}
}
Summary

• IPC provides high-level language + typed data primitives:
  – socket abstraction, send/receive
  – synchronous/asynchronous communication
  – Java classes (different from operating system primitives)
  – automatic marshaling of data into machine-independent format

• For Java API to IP Multicast:
  – Class MultiSocket (subclass of DatagramSocket)
  – see textbook website for sample code