Inter-process communication

1

Topics for this lecture

- **Inter-process communication**
  - Characteristics of IPC
  - Synchronous and asynchronous communication
  - External data representation and marshalling
    - CORBA's Common Data Representation
    - Java Object serialisation
  - **Client-Server Communication**
    - Client-Server Communication
    - Communication within services provided by a group of servers:
      - Group Communication
      - IP multicast

2

Significance of this lecture

- DIS depend on networks
  – Networks depend on messaging
  – Messaging needs to be:
    - Efficient
    - Reliable
    - Robust
    - And ideally transparent
  – How is this achieved?

3

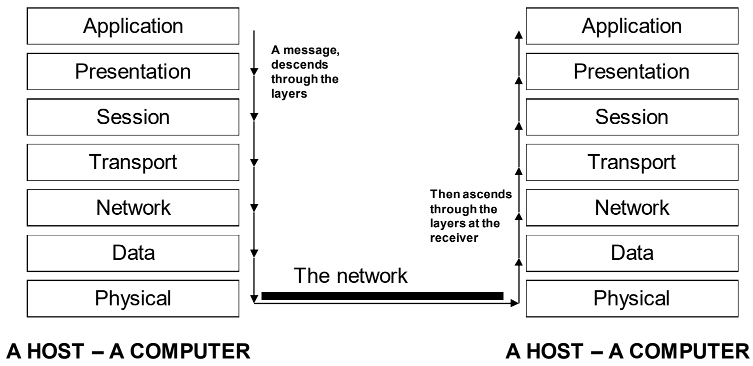
Significance of this lecture

- We have looked at networking
- So now we turn our focus to higher level software layers
- These layers are designed to facilitate easy operation
- And to embody the ‘good’ design characteristics of our DIS

4

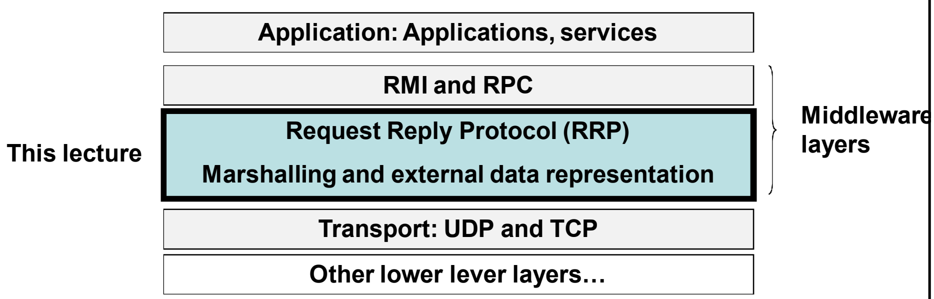
Recall: The software layered approach

- The OSI model



5

The focus of this lecture



6

Messaging basics

- API (application programming interface)
  - In the context of this lecture this refers to an interface for application programmers to use either UDP or TCP
- Message passing has two fundamental operations
  - Send and Receive (Request and Reply)
  - i.e. one process sends the other receives
  - Simply requires both destinations and messages
  - May involve synchronisation
- BUT…
  - How does this occur?

7

Synchronisation

- Synchronisation
  - Synchronous (blocking)
    - Sender is 'blocked', i.e. is frozen during send until the receive message is returned
    - Sender and receiver synchronise
  - Asynchronous (non-blocking)
    - Sender can carry on processing once the message is sent (i.e. copied to some local buffer)
  - Current systems tend to be synchronous
  - Non-blocking adds complexity into software code

8

Synchronisation (cont.)

- Examples
  - Blocking:
    - Request money from an ATM
    - The ATM blocks until it receive authorisation from the bank.
  - Non-blocking
    - In a DIS a non-blocking message allows sending processes to carry on working while waiting for a response
    - Email is a good real world example of non-blocking message
    - You send a mail and carry on working while waiting for a reply
    - Update traffic details on an in car sat/nav device
    - The sat/nav device carries on working while waiting for new data

API & IP: process communication

- **Message destinations**
  - Addressing
    - The address uniquely identifies the hardware, perhaps a computer (host on a node)
    - A local port a destination on a computer
      - One or more messages can be send to port(s)
      - Processes can use one or more ports for receiving messages

156.254.12.35:875

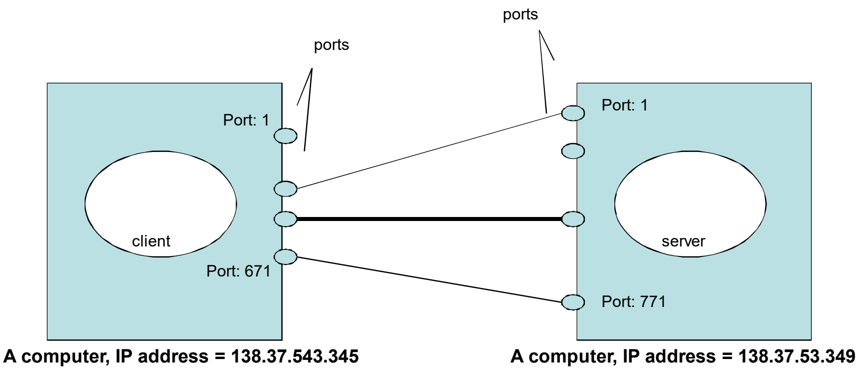
Address + port

API & IP: process communication

- **Other issues to consider during inter-process communication**
  - **Reliability**
    - Reliable systems should not corrupt messages even if packets are lost or dropped
  - **Ordering**
    - Messages be delivered in the order sent

11

Process communication



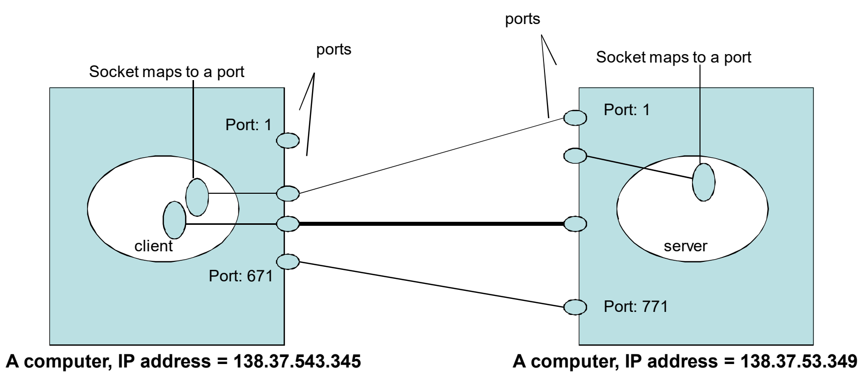
12

Process communication & Sockets

- **Sockets**
  - An abstraction whereby process bind to a socket which relates to a port
  - Assigns a specific local port to a process
  - Process talk to the socket – which relates to a port
  - Processes can only access message from ports linked to their socket
  - Requests must be made to specific ports but could originate for any or in some instances specific ports

13

Process communication



14

UDP communication

- UDP and the Datagram method
  - In a nutshell
    - Datagram sent from one process to a receiving process
    - First the sender (the *client*) and receiver (the *server*) must bind to a socket
      - » Client could bind to any port
      - » Server binds to its specified broadcast port for receiving messages
    - The client then sends its message to the server address including in the message the senders address (required for the response)
    - The server receives and processes
    - The server then sends a response to the client address and port

15

UDP communication

- More details
  - The *send* method is non-blocking (asynchronous)
    - thus they are free once send has taken place
  - The *receive* method uses blocking (synchronous), although other threads can be used to perform other work.
  - Receive blocking can use time-outs to limit the length of the block.
    - although defining 'good' values for timeouts is hard
  - Received messages are stored in the bound socket's queue.
  - Receive inspects the bound socket for messages
  - Received messages could come from anywhere

16

UDP communication

- Uses of UDP and the Datagram method
  - Useful where omission failures are tolerable
    - i.e. naming services
  - Useful in reducing communication overheads that exist in guaranteed delivery methods

UDP communication

- UDP and the Datagram method
  - Method example:
      - aSocket.send(*request*)
      - aSocket.recieve(*reply*)
    - where both *request* and *reply* are DatagramPackets
  - Other methods:
      - setSoTimeout
      - connect

API & IP: UDP communication

- In Java
  - A **DatagramPacket** class contains:

The message	Length of message	Internet address	Port

  - i.e.

3432 543 4531	13	145.25.123.871	589

In Java a DatagramPacket is constructed:

```
myPacket = new DatagramPacket(m,args[0].length(), aHost, serverPort);
```

Note: the DatagramPacket contains the host address (aHost) and the host port (serverPort)

19

API & IP: UDP communication

- In Java

In Java a DatagramPacket is sent and received thus:

```
aSocket.send(myPacket);
aSocket.recieve(myPacket);
```

Note: aSocket is an instantiated instance of the Java DatagramSocket class.

Interested readers are referred to the course text (pages 138 and 139) for more detailed implementations of UDP client and servers

20

API & IP: TCP communication

- API to TCP uses a stream abstraction
  - A stream of bytes
- The stream abstraction hides the following:
  - Message sizes
  - Lost messages
  - Flow control
  - Message duplication and ordering
  - Destinations

21

API & IP: TCP streaming

- How does it work? In a nutshell
  - Client requests a stream
  - Client creates stream socket to any port
  - Client sends a *connect* request to a server at the server specific port
  - The server is bound to a designated server port and listens for *connect* messages from clients
  - Once a connect message is accepted the server uses a new socket for the stream
  - The client is informed of the socket
  - The client then connects to the dedicated socket

22

API & IP: TCP streaming

- How does it work? More detailed
  - The server is bound to a designated server port and listens for *connect* messages from clients
  - A client requests a stream and creates a socket to any port to send its request from
  - The client then sends a *connect* request to a server at the server specific port
  - Once a connect message is accepted the server creates a another socket for the stream
  - The client is informed of the new socket
  - The server send information via the new socket
  - The client and server communicate using input and output streams through their sockets.

23

API & IP: TCP streaming

- Dealing with failures
  - TCP does not provide reliable communication as it can fail
    - The sender stops sending if too many dropped or fail checksums or the network fails
    - NOTE: The cause of the failure cannot be known by each process (i.e. network failure or process failure)

24

API & IP: TCP streaming

- Example uses of TCP
  - HTTP
  - FTP
  - Telnet
  - SMTP
- Frequently using reserved port numbers

25

API & IP: UDP communication

- In Java
  - Both the server and client have two classes:
    - **DataInputStream** and a **DataOutputStream**
  - A **Socket** class is used to retrieve each stream
    - **aClientSocket.getInputStream()**
    - **sClientSocket.getOutputStream()**
  - The streams can be read or write to using
    - **outputStream.writeUTF(data)**
    - **inputStream.readUTF(data)**

Each stream is a string of bits representing in binary primitive data types.

Other classes include:
ServerSocket (used for the server to listen for client stream request)

26

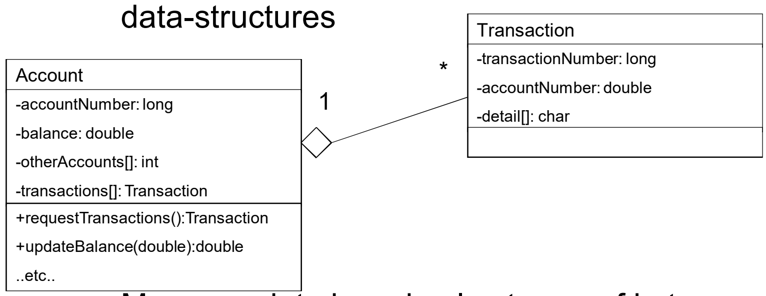
External data & Marshalling

– Process and applications use primitive data types,
  - Integers, doubles, etc.
  - Not all primitive data types are stored the same in each system
  - Heterogeneity
    – big-endian / little-endian with integers,
    – floating points are represented differently too,
    – 1 byte, 2 bytes used for strings.
    – Many more too!!

27

External data & Marshalling

- Process data is frequently stored as objects and data-structures



- Message data is a simple stream of bytes

0101010101010101010101010010111101010100101001010 1

28

External data & Marshalling

- A sender must then convert its data into a stream of bytes
- And bytes have to fit an agreed standard

- Converting data is called **marshalling**
- The agreed standard is called **external data representation**

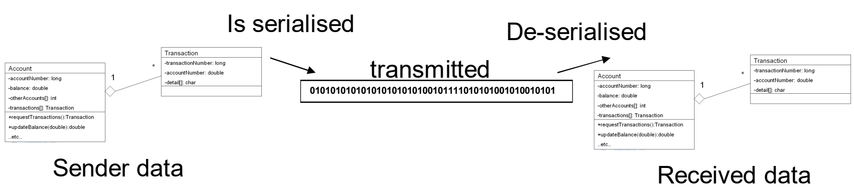
- Both a sender and receiver need to marshal or un-marshal data from the agreed external format

29

API & IP: External data & Marshalling

– All objects and data need to be serialised for sending

– Data can then be reconstructed once received



But of course… The receiver could format the data anyway they want!

30

API & IP: External data & Marshalling

- Essentially there are two methods
  - 1. Agree an external format
    - Sender converts to external format
    - Receiver converts to their local format
  - 2. The data is sent using the format with an indication of the format that is used.

31

API & IP: External data & Marshalling

- We shall take a brief look at:
  - CORBA and the common data representation
  - Java Object serialisation
  - XML

  - CORBA and JAVA use a binary external data representation
  - XML uses a textual representation
  - Either is fine – so long as both the send and receiver know which is being used,

32

CORBA

- Marshalling in CORBA can be automatically accomplished
- We simply need to define an IDL (Interface Definition Language) for the data we want to transmit in messages
- A CORBA IDL contains both:
  - The methods that are to be used remotely
  - The data types and attributes that they will used

33

CORBA - CDR

- Data representation: uses Common Data Representation (CDR)
  - CDR supports 15 primitive types + some composite types
    - Primitives such as *long*, *short*, *boolean*, etc. (see page 146)
    - Complex types are *sequence*, *array*, *union*, *string*, *enumerated*, *struct*
- Marshalling is automatic providing an IDL (Interface Definition Language) is available for the data:

```
struct Person{
   string name;
   string place;
   long DOB;
};
```

This tells CORBA that we are going to use a Person data structure that contains a name and place both stored as strings and a DOB stored as a long

34

CORBA - CDR

- Lets consider the following data structure
  person{“Simon”, “London”, 1976}
- This given the IDL CORBA could convert the data into a a stream of bytes according to CORBA’s rules

0-3	**5**
4-7	**Simo**
8-11	**n_ _ _**
12-15	**6**
16-19	**Lond**
20-23	**on_ _**
24-27	**1976**

CDR sends strings by first sending the string length then the data

Another string, again in the CDR format

Integer sent as is

35

JAVA serialisation

- Java object serialisation
  - Java RMI can serialise objects + primitive types
  - Classes that are to be serialised MUST implement the *Serialisable* Java interface
    - i.e. **public class Person implements Serialisable**
  - Serialised classes can be sent as messages and stored on disk
  - In Java:
    - Serialisation refers to the flattening of the classes and data
    - De-serialisation refers to the un-flatting (reconstruction) of the data and classes

36

JAVA serialisation

What is written?

1. Class information
   name of the class, version and methods
2. The data
3. Any referenced classes
   3.1 This involves writing their class information
   3.2. And.. their data too
   3.3 And.. any referenced classes from the referenced class
      3.31. This involves writing their class information
      3.3.2. And.. Their data too
      3.3.3. Any referenced classes in the referenced class
      And so on as a recursive procedure

37

JAVA serialisation

- So what gets serialised…

```
public class Person implements Serilizable{
  private String name;
  private String place;   private long DOB;
  public Person(String inName, String inPlace,
                long inDOB){
    name=inName;
    place = inPlace;
    DOB = inDOB;
  }
}
```

The class information

Attribute	Data
name	Simon
place	London
DOB	1976

And the data

Class + its data

38

JAVA serialisation

- Example of a serialisation
  – Person p = **new** Person("Simon", "London", 1976);

  – Written to an *ObjectOutputStream* using *writeObject*
  – Read from an *ObjectOutputStream* using *readObject*

Person	8-bype version number		h0
3	Int year	java.lang.String name:	java.lang.String place:
1976	5 Simon	6 London	h1

- Serialisation and de-serialisation is handled by the middleware, the programmers does not need to worry about this process except to understand how to define it (serialisable) and then how to read and write objects.

39

Java serialisation

- Java Reflection
  – The ability to enquire about the properties of a class, names, types, methods and instance variables
  – Using reflection objects can easily be serialised and de-serialised as they can inspect themselves to determine what they are made of.
  – Reflection simplifies the marshalling process significantly as we have seen.

40

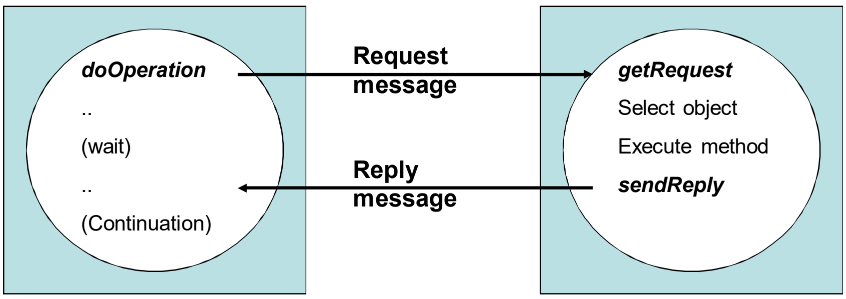
Client / Server communication

- Client-server message exchanges commonly use the Request Reply (RR) protocol.
  - They also tend to be synchronous
    - clients block until a response is returned
  - It is reliable as a response constitutes an acknowledgement
    - RR is reliable for the client
    - Request Reply Acknowledge (RRA) makes the communication reliable for both the server and client

- Communication itself could use either UDP or TCP

41

Client Server Communication



Client calls the server method using **doOperation**

Server receives and reads the request using **getRequest**

The server then matches the object of the request and executes its method

The server then sends the reply (**sendReply**)

42

Client Server Communication

doOperation(RemoteObjectReference o, int methodId, byte[] arguments)

arguments is the marhsaled data

methiodId the method being remotely invoked

o a reference to the remote object

getRequest();

collects a request from a server port

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

sends the request back to the cleint

reply is the marshalled data

43

Object References

- Client invoking a remote object
  - Sends message to server
  - Specifies the object that contains the method
- This requires a reference
- Solution = use an object reference
  - Object references are passed with the message request
  - Object references need to be unique
    - Unique can be guaranteed using the following as a Object Reference

32 bits	32 bits	32 bits	32 bits	
Internet Address	Port	Time	Object Number	Interface of remote object

44

Client Server Communication

A typical message would contain:

- messageType
- requestID
- objectReference
- methodId
- arguments

All in serial form

45

Client Server Communication

- Message labelling / identification
  - Reliability requires a robust scheme for identification of messages
  - This comprises of two parts
    - An ID should be unique local
      - A simple number would suffice
    - An ID should be unique for the DIS
      - Suffix the sender address and port

46

Client Server Communication

- Failures
  - The RR protocol needs to deal with failures
    - If the datagram technique is used
      - Omission failures (timeouts helps solve)
      - Message order failure (labelling helps to solve)
  - And possible process failures too
  - Many techniques can be used to reduce problems

47

Client Server Communication

- Timeouts
  - Clients may have timeouts for blocking for a response before re-submitting the request
- Ignore duplicate requests
  - If the server is still process
- Dealing with lost replies
  - If the server completes processing and then recieves the same request
  - Re-run process… but.. does repeating the process produce the same result?
  - Could store history of process results
  - Different types of operations
    - (idempotent operation = an operation that always has the same affect)
- histories
  - Client block implies we need only store the most recent process for a client, this can be stored in a history file
  - Could have time limit on historical data

48

Client Server Communication

- TCP streams for C/S messaging
  - Can be useful for client server message exchanges
  - This method allows large amounts of data streams to be exchanged without the need for sending and managing numerous datagram packets
    - i.e. serialised objects are typically large
  - Eliminates the need for histories, duplicate checking etc. as the transmission control protocol provides these facilities

49

Group communication

- Multicast
  - Pair wise communication is not always the most efficient or desired
  - Multiple servers might provide similar services
  - It might then be more efficient to request the service from either server
- Benefits of multicast
  - Fault tolerance
  - Discovery services
  - Better performance through replication
  - Useful for event notifications

50

Group communication

- IP multicast
  - Multicast address are specified in the D class
    - 224.0.0.0 to 239.255.255.255
  - Clients send their message to a group IP addresses (see above)
  - Clients join IP groups and when they do receive all messages sent to that group
  - Multicast routers can be used to forward multicasts to other routers and or hosts
  - Note: is available via UDP for obvious reasons

51

Summary

- **Inter-process communication**
  - Characteristics of IPC
  - Synchronous and asynchronous communication
- **External data representation and marshalling**
  - CORBA's Common Data Representation
  - Java Object serialisation
- **Client-Server Communication**
  - Client-Server Communication
  - Communication within services provided by a group of servers:
  - Group Communication
    - IP multicast

52