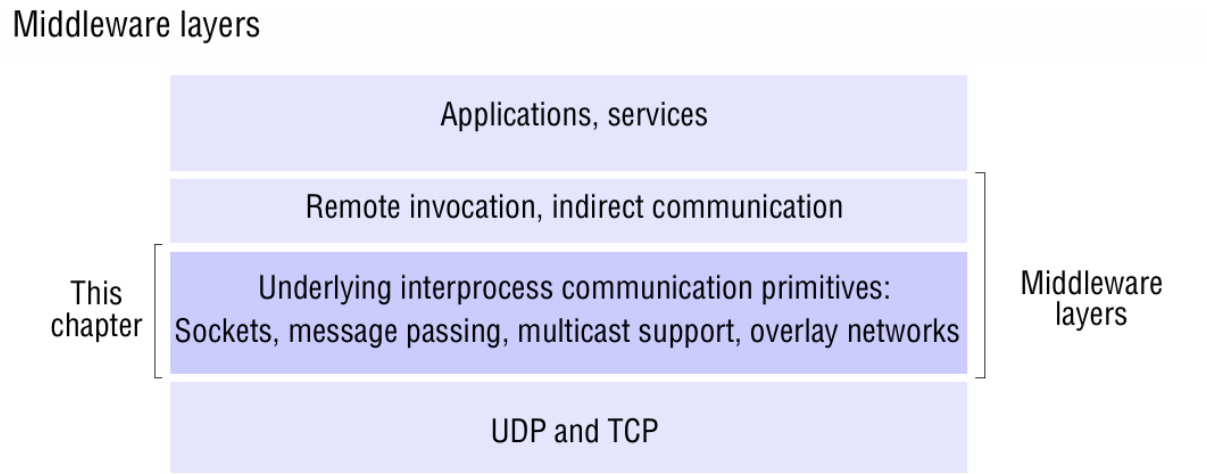
UNIT – 3

The characteristics of protocols for communication between processes in a distributed system – that is, interprocess communication.

Interprocess communication in the Internet provides both datagram and stream communications.




**API for Internet Protocols**

The characteristics of interprocess communication –

Message passing between a pair of processes can be supported by two message communication operations, send and receive, defined in terms of destinations and messages.

To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message.

- involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes

**Synchronous and asynchronous communication**

A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues.

Communication between the sending and receiving processes may be either synchronous or asynchronous.

In the synchronous form of communication, the sending and receiving processes synchronize at every message. In this case, both send and receive are blocking operations. Whenever a send is issued the sending process (or thread) is blocked until the corresponding receive is issued. Whenever a receive is issued by a process (or thread), it blocks until a message arrives.

In the asynchronous form of communication, the use of the send operation is non blocking in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The receive operation can have blocking and non-blocking variants.

**Message destinations**

in the Internet protocols, messages are sent to (Internet address, local port) pairs.

A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but can have many senders.

Servers generally publicize their port numbers for use by clients.

Location Transparency

Client servers refer to services by name and use a name server or binder.

The os provides location independent identifiers for message destinations.

**Reliability**

reliable communication in terms of validity and integrity.

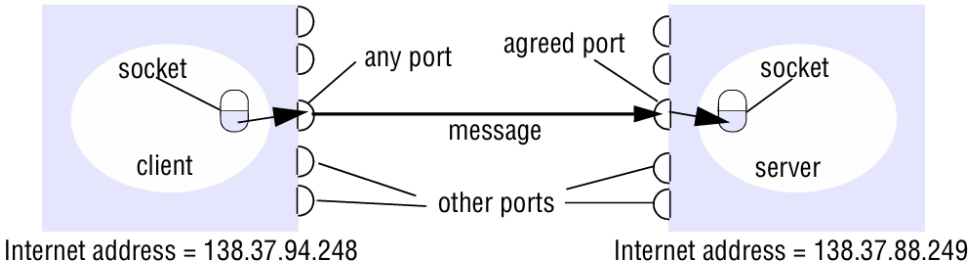
a point-to-point message service if messages are guaranteed to be delivered despite a 'reasonable' number of packets being dropped or lost

Ordering •

applications require that messages be delivered in sender order – that is, the order in which they were transmitted by the sender. The delivery of messages out of sender order is regarded as a failure by such applications.

Sockets

Sockets and ports




Both forms of communication (UDP and TCP) use the socket abstraction, which provides an endpoint for communication between processes.

Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process

For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs

Processes may use the same socket for sending and receiving messages.

Each computer has a large number (2^16) of possible port numbers for use by local processes for receiving messages.

any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

Java API for Internet addresses •

Java provides a class, InetAddress, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk"); This method can throw an UnknownHostException.

the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet addresses – 4 bytes in IPv4 and 16 bytes in IPv6.

**UDP datagram communication**

A datagram is transmitted between processes when one process sends it and another receives it.

To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port.

A server will bind its socket to a server port – one that it makes known to clients so that they can send messages to it.

A client binds its socket to any free local port.

The receive method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

some issues relating to datagram communication are –

**Message size –**

The receiving process needs to specify an array of bytes of a particular size in which to receive a message.

This IP protocol allows packet lengths of up to 2^16 bytes, which includes the headers as well as the message.

environments impose a size restriction of 8 kilobytes

**Blocking -** non-blocking sends and blocking receives for datagram communication

**Timeouts:** a process that invoked a receive operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, timeouts can be set on sockets.

**Receive from any:** The Inet Address and Local port of sender need to be specified.

**Failure model for UDP datagrams**

reliable communication in terms of two properties: integrity and validity.

Omission failures: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination.

Ordering: Messages can sometimes be delivered out of sender order.

**Use of UDP**

Voice over IP (VOIP) also runs over UDP.

UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery.

There are three main sources of overhead:

• the need to store state information at the source and destination;

• the transmission of extra messages;

• latency for the sender.

**Java API for UDP datagrams**

• The Java API provides datagram communication by means of two classes: DatagramPacket and DatagramSocket.

An instance of DatagramPacket may be transmitted between processes when one process sends it and another receives it.

*Datagram packet*

| | | | |
|---|---|---|---|
| array of bytes containing message | length of message | Internet address | port number |

Methods

getData.

Message can be retrieved.

The methods getPort and getAddress access the port and Internet address

DatagramSocket: This class supports sockets for sending and receiving UDP datagrams.

Methods

send and receive: send is an instance of DatagramPacket containing a message and its destination.

receive is an empty DatagramPacket in which to put the message, its length and its origin. The methods send and receive can throw IOExceptions.

setSoTimeout: allows a timeout to be set. throw an InterruptedIOException

connect: connecting to a particular remote port and Internet address,

**Figure 4.3** UDP client sends a message to the server and gets 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,  m.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();}
   }
}
```

**Figure 4.4** UDP server repeatedly receives a request and sends it back to the client

```
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

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:

Message sizes: The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data.

Lost messages: The TCP protocol uses an acknowledgement scheme.

Flow control: The TCP protocol attempts to match the speeds of the processes that read from and write to a stream.

Message duplication and ordering: Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.

Message destinations: A pair of communicating processes establish a connection before they can communicate over a stream

Establishing a connection involves a connect request from client to server followed by an accept request from server to client before any communication can take place.

issues related to stream communication:

Matching of data items: Two communicating processes need to agree as to the contents of the data transmitted over a stream.

Blocking: The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available

Threads: When a server accepts a connection, it generally creates a new thread in which to communicate with the new client.

Failure model • To satisfy the integrity property of reliable communication, TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets.

For the sake of the validity property, TCP streams use timeouts and retransmissions to deal with lost packets.

Use of TCP

HTTP: The Hypertext Transfer Protocol is used for communication between web browsers and web servers

FTP: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.

Telnet: Telnet provides access by means of a terminal session to a remote computer.

SMTP: The Simple Mail Transfer Protocol is used to send mail between computers.

Java API for TCP streams • The Java interface to TCP streams is provided in the classes ServerSocket and Socket:

Socket– a socket to use for communicating with the client

ServerSocket : use by a server to create a socket at a server port for listening for connect requests from clients. Its accept method gets a connect request from the queue or, if the queue is empty, blocks until one arrives.

**Figure 4.5** 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 Sec 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*/}}
    }
}
```

**Figure 4.6** TCP server makes a connection for each client and then echoes the client's request

```
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());}
    }
}
class Connection extends Thread {
```

```
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*/}}
    }
}
```

External Data Representation and Marshalling.

An agreed standard for the representation of data structures and primitive values is called an external data representation.

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 process of disassembling them on arrival to produce an equivalent collection of data items at the destination.

Three alternative approaches to external data representation and marshalling are

CORBA'

Java's object serialization,

XML (Extensible Markup Language),

CORBA's Common Data Representation (CDR)

CDR is the external data representation

CDR can represent all of the data types. These consist of 15 primitive types, which include short (16-bit), long (32-bit), unsigned short, unsigned long, float (32-bit), double (64-bit), char, boolean (TRUE, FALSE), octet (8-bit), and any (which can represent any basic or constructed type);

Primitive types: CDR defines a representation for both big-endian and little-endian orderings.

Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order,

## CORBA CDR for constructed types

| *Type* | *Representation* |
|---|---|
| *sequence* | length (unsigned long) followed by elements in order |
| *string* | length (unsigned long) followed by characters in order (can also have wide characters) |
| *array* | array elements in order (no length specified because it is fixed) |
| *struct* | in the order of declaration of the components |
| *enumerated* | unsigned long (the values are specified by the order declared) |
| *union* | type tag followed by the selected member |

Marshalling in CORBA

CORBA IDL to describe the data structure in the message

```
struct Person{
        string name;
        string place;
        unsigned long year;
};
```

CORBA CDR message

| *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 | 1984 | *unsigned long* |

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

Java object serialization

Java class equivalent to the Person struct defined in CORBA IDL might be:

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
    // followed by methods for accessing the instance variables
}
```

serialization refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message,

Deserialization consists of restoring the state of an object or a set of objects from their serialized form.

As an example, consider the serialization of the following object: Person p = new Person("Smith", "London", 1984);

**Figure 4.9** Indication of Java serialized form

| *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* |
| 1984 | 5 Smith | 6 London | h1 | *values of instance variables* |

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

Extensible Markup Language (XML)

XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web.

In general, the term markup language refers to a textual encoding that represents both a text and details as to its structure or its appearance.

XML is extensible in the sense that users can define their own tags, in contrast to HTML,

XML definition of the *Person* structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

XML elements and attributes

Elements: An element in XML consists of a portion of character data surrounded by matching start and end tags

Ex : <name> </name> tag pair

Attributes: A start tag may optionally include pairs of associated attribute names and values such as id="123456789"

Names: The names of tags and attributes in XML generally start with a letter, but can also start with an underline or a colon.

Binary data: All of the information in XML elements must be expressed as character data.

Parsing and well-formed documents

A basic rule is that every start tag has a matching end tag. When a parser reads an XML document that is not well formed, it will report a fatal error.

CDATA: XML parsers normally parse the contents of elements because they may contain further nested structures

```
<place> King&apos Cross </place >
<place> <![CDATA [King's Cross]]></place >
```

XML prolog: Every XML document must have a prolog as its first line. The prolog must at least specify the version of XML in use (which is currently 1.0). For example:

```
<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>
```

XML namespaces

XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL.

For example: xmlns:pers = http://www.cdk5.net/person

Illustration of the use of a namespace in the *Person* structure

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1984 </pers:year>
</person>
```

XML schemas

• An XML schema defines the elements and attributes that can appear in a document, how the elements are nested and the order and number of elements,

An XML schema for the *Person* structure

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions >
    <xsd:element name= "person" type ="personType" />
        <xsd:complexType name="personType">
            <xsd:sequence>
                <xsd:element name = "name"  type="xs:string"/>
                <xsd:element name = "place"  type="xs:string"/>
                <xsd:element name = "year"  type="xs:positiveInteger"/>
            </xsd:sequence>
            <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
        </xsd:complexType>
</xsd:schema>
```

APIs for accessing XML • XML parsers and generators are available for most commonly used programming languages. For example, there is Java software for writing out Java objects as XML (marshalling) and for creating Java objects from such structures (unmarshalling). Similar software is available in Python for Python data types and objects.

Remote object references

A remote object reference is an identifier for a remote object that is valid throughout a distributed system.

Representation of a remote object reference

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