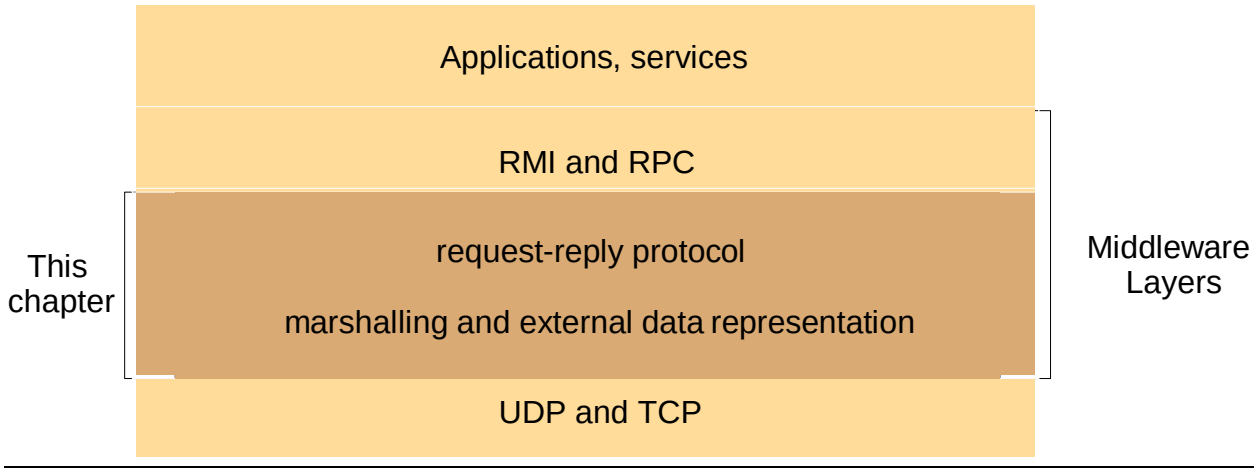
UNIT – II INTERPROCESS COMMUNICATION



2.1 API for the internet protocol:

- Internet protocol as a example, explaining how programmers can use them, either by meaning of UDP message or through TCP streams.
- The application program interface **(API)** to **UDP** provides a message passing abstraction.
- The application program interface **(API)** to **TCP** provides the abstraction of a two-way stream between pairs of processes.
- The Java UDP and TCP APIs are discussed in the second section.

2.1.1 Characteristics of Inter process Communication:

- Message passing between a pair of processes can be supported by two message communication operations: ***send*** and ***receive***.
- In the **synchronous** form of communication, both **send** and **receive** are ***blocking*** operations.
- In the **asynchronous** form of communication, the ***send*** operation is **non-blocking** and the ***receive*** Operation can be **blocking and non-blocking.**
- Non-blocking receive is more efficient but more complex. So current systems do not generally provide the non-blocking form of receive.
- In the Internet protocols, messages are sent to (*Internet address*, *local port*) pairs. Any process that knows the number of a port can send a message to it.
- Using a fixed Internet address can be avoided by one of the following approaches to provide location transparency:
- Client program refer to services by name. This allows services to be relocated but not to migrate (be moved while running).
- The operating system, for example, Mach, provides location-independent identifiers for message destinations.

- Messages can be addressed to processes in the V system. However, ports provide several alternative points of entry to a receiving process. Chorus provides the ability to send messages to groups of destination.
  - **Ordering**: Some applications required that messages be delivered in send order-i.e; the order in which they were transmitted by sender. The delivery of messages out of sender order is regarded as failure by such applications.

Inter process communication in the Internet provides both datagram and stream communication. The Java APIs for these are presented, together with a discussion of their failure models. They provide alternative building blocks for communication protocols. This is complemented by a study of protocols for the representation of collections of data objects in messages and of references to remote objects.

Multicast is an important requirement for distributed applications and must be provided even if underlying support for IP multicast is not available. This is typically provided by an overlay network constructed on top of the underlying TCP/IP network. Overlay networks can also provide support for file sharing, enhanced reliability and content distribution.

Message destinations

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

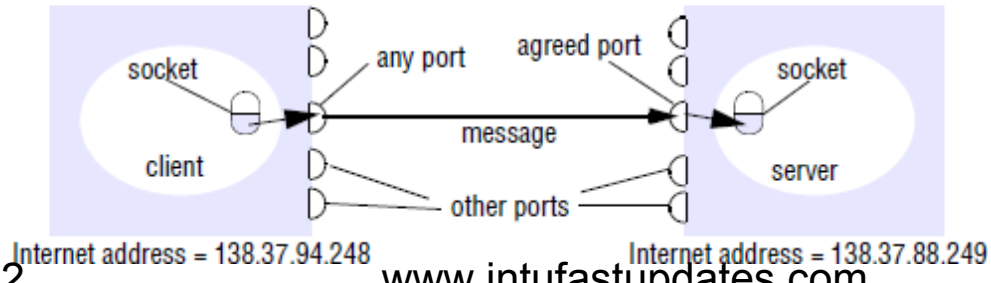
Reliability

- A reliable communication is defined in terms of **validity** and **integrity**.
- **Validity** is defined, a point-to-point message service is described as reliable if messages are guaranteed to be delivered without dropped or lost.
- For **integrity**, messages must arrive uncorrupted and without duplication.
- Some applications require that messages be delivered in ***send order***

2.2. Sockets

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an **endpoint for communication between processes**. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Inter process communication consists of transmitting a message between a socket in one process and a socket in another process, as illustrated in Figure below.

- A socket provides endpoints for communication between processes as shown in Figure
- A socket must be bound to a local port.
- A socket pair (local IP address, local port, foreign IP address, foreign port) uniquely identifies a coSockets and ports



2.2.1. Java API for Internet Address

As the IP packets underlying UDP and TCP are sent to internet addresses. Java provides a class, Inet Address that represents Internet addresses. Users of this class refer to computers by DNS hostnames.

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk") ;

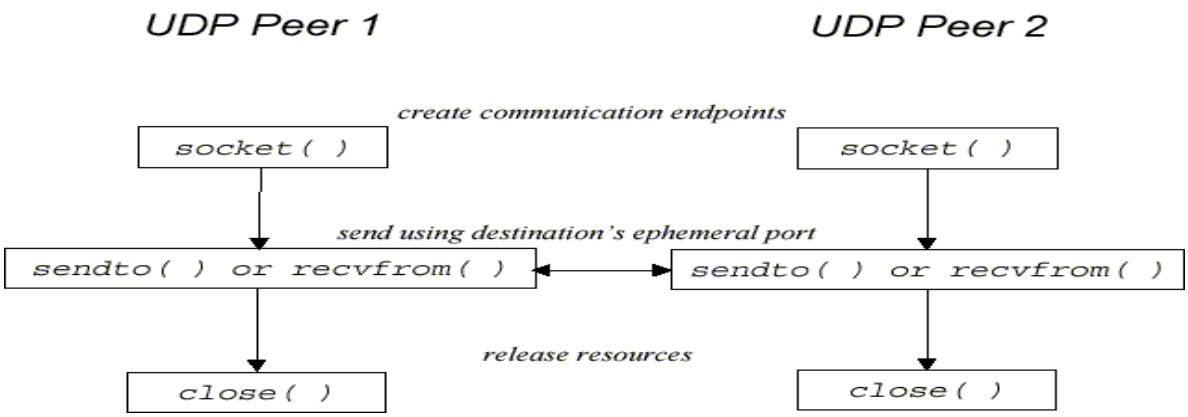
This method throws an UnknownHostexception.

2.2.2. UDP datagram communication

Datagram sent by UDP is transmitted from a **sending process** to a **receiving process without acknowledgement or retries**. If a failure occurs, the message may not arrive. 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.

The receiving process needs to specify an array of bytes of a particular size in which to receive a message. The most size restriction is **8 kilobytes**.

Sockets normally provide **non-blocking** sends and **blocking receives** for datagram communication.



The following are some issues relating to datagram communication:

- ***Message size***: The receiving process needs to specify an array of bytes of a particular size in which to receive a message. The most size restriction is 8 kilobytes.
- ***Timeouts*:** The method receive blocks until a datagram is received, unless a timeout has been set on the socket
- **Receive from any:** The receive method does not specify an origin for messages. Instead ,an invocation of receive gets a message addressed to its socket from any origin.The receive method returns the Internetaddress and local port of the sender, allowing the recipient to check where the message came from.
- **Blocking** : Sockets normally provide non-blocking sends(Sender will perform all operations along with sending operation) and blocking receives(receiver stop the all operation except receiver) for datagram communication.

  Sockets normally provide blocking sends (Sender will perform all operations along with sending operation) and blocking receiver (receiver stop the all operation except receiver) for TCP communication.

- **Failure model / UDP datagram's suffer for following failure**:

Reliable communication in terms of two properties: integrity and validity. The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP data grams suffer from the following failures:

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. To simplify the discussion, we regard send-omission and receive-omission failures as omission failures in the communication channel.

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

> Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements.

Use of UDP • For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over 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.

Common Internet Applications that use UDP

- **Trace route** - print the route packets take to network host. Try /usr/sbin/traceroute www.microsoft.com.
- **RIP** (Routing Information Protocol) is a widely-used protocol for managing router information
- **BOOTP** (Bootstrap Protocol) is a protocol that lets a network user be automatically configured (receive an IP address) and have an operating system boot or initiated without user involvement. BOOTP is the basis for a more advanced network manager protocol, the Dynamic Host Configuration Protocol (DHCP).
- **Dynamic Host Configuration Protocol (DHCP)** is a communications protocol that lets network administrators manage centrally and automate the assignment of Internet Protocol (IP) addresses in an organization's network.)
- **Network Time Protocol (NTP)** is a protocol that is used to synchronize computer clock times in a network of computers.
- **Trivial File Transfer Protocol (TFTP)** is an Internet software utility for transferring files where user authentication and directory visibility are not required.
- **Simple Network Management Protocol (SNMP)** is the protocol governing network management and the monitoring of network devices and their functions.
- The **domain name system (DNS)** is the way that Internet domain names are located and translated into Internet Protocol addresses.
- The **Network File System (NFS)** is a client/server application that lets a computer user view and optionally store and update file on a remote computer as though they were on the user's own computer. UDP is used with the early version of NFS.
- **Remote Procedure Call (RPC)** is a protocol that one program can use to request a service from a

program located in another computer in a network without having to understand network details.

- There are several RPC models and implementations. A popular model and implementation is the Open Software Foundation's **Distributed Computing Environment (DCE).**
- **Open Network Computing (ONC)** RPC, sometimes referred to as Sun RPC, was one of the first commercial implementations of RPC.

The Java API provides datagram communication by means of two classes:

DatagramPacket - Datagram packets are used to implement a connectionless packet delivery service. DatagramSocket - A datagram socket is the sending or receiving point for a packet delivery service.

DatagramPacket contain the following methods

getData - Returns the data buffer.

getPort - Returns the port number on the remote host.

getAddress - Returns the IP address.

DatagramSocket contain the following methods:

send - Sends a datagram packet from this socket.
receive - Receives a datagram packet from this socket.

setSoTimeout - Enable/disable the specified timeout, in milliseconds.
connect - Connects the socket to a remote address for this socket.

Datagram packet:

arrayofbytescontainingmessage	lengthofmessage	Internetaddress	portnumber

UDP CLIENT

```
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, 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()));
        }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

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

2.2.3. TCP stream communication

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. The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handed to the application as requested. Applications can, if necessary, force data to be sent immediately.

***Lost messages*:**

The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges

all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required.

Flow control:

The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.

***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*:**

Once the connection between a pair of processes is established, the processes simply read from and write to the stream

TCP Abstractions

- *The data is the abstraction of a stream of bytes.*
- *A connection is established before messages are sent.*
- *It assumes one process is the client and one is the server in establishing a connection.*
- *The client create a socket bound to any local port and then makes a **connect** request for connecting the server at its server port.*
- *The server creates a listening socket bound to a server port and wait for (**accept**) the incoming requests.*
- *Messages are sent using handles (sockets) rather than source-destination addresses.*
- *An application **closes** a socket once it is done with the transmission.*
- ***The issues related to stream communication are:***
    - ***Matching of data items**: Two communicating processes need to agree on the contents of the transmitted data.*
    - ***Blocking**: When a process tries to read data from an input channel (stream), it will get the data or block until the data is available.*
    - ***Thread**: When a sever accepts a connection, it generally creates a new thread for the new client.*

TCP Failure Model

- ***TCP uses three mechanism to create a reliable communication**:*
    - ***Checksums and sequence numbers for integrity**: TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets.*
    - ***Timeouts** and retransmission for validity: TCP streams use timeouts and retransmissions to deal with lost packets.*
- *The problem is if the **network becomes congested**, no acknowledge is received and then the connection is broken. Thus TCP does not provide reliable communication.*

- ***When a connection is broken, it will has the following effects:***
  - *The processes using the connection cannot distinguish between network failure and failure of the process.*
  - *The communication process cannot tell whether their recent messages have been received or not.*
- *The programmers need to deal with this situation in the program.*

Common Internet applications that use 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.BGP (Border Gateway Protocol) is a protocol for exchanging routing information between gateway hosts (each with its own router) in a network of autonomous systems.

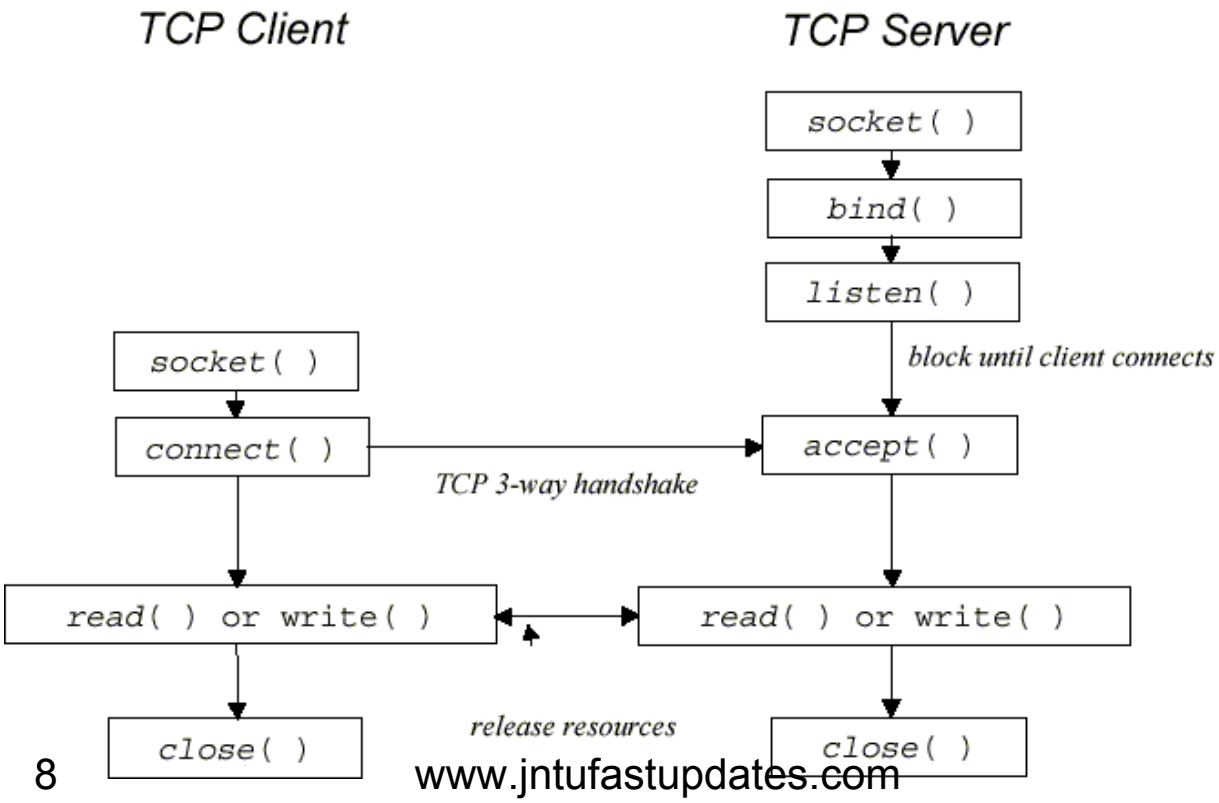
The domain name system (DNS) is the way that Internet domain names are located and translated into Internet Protocol addresses.

NNTP (Network News Transfer Protocol) is the predominant protocol used by computer clients and servers for managing the notes posted on Usenet newsgroups.

The Network File System (NFS) is a client/server application that lets a computer user view and optionally store and update file on a remote computer as though they were on the user's own computer.

Open Network Computing (ONC) RPC, sometimes referred to as Sun RPC, was one of the first commercial implementations of RPC.

The Open Software Foundation's Distributed Computing Environment (DCE) is a popular model and implementation of RPC.



TCP 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 ) {System.out.println ("close:"
+e.getMessage()); }}}
```

TCP Server *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());}
```

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

2.3 External data representation and marshalling

The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival. The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures. There are two variants for the ordering of integers: the so-called ***big-endian*** order, in which the most significant byte comes first; and ***little-endian*** order, in which it comes last. Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use **ASCII character** Coding, taking one byte per character, whereas the **Unicode standard** allows for the representation of texts in many different languages and takes two bytes per character. One of the following methods can be used to enable any two computers to Exchange binary data values:

• The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.

• The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. **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. Thus marshalling consists of the translation of structured data items andPrimitive values into an external data representation. Similarly, Unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

2.3.1. Three alternative approaches to external data representation and marshalling are discussed

• **CORBA's** common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages

• **Java's object serialization**, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

• **XML (Extensible Markup Language),** which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

In the first two cases, the marshalling and un-marshalling activities are intended to be carriedoutbyamiddlewarelayerwithoutanyinvolvementonthepartoftheapplication programmer. In the first two approaches ,the primitive data types are marshaled into a binary form .In the third approach(XML),the primitive data types are represented textually.

2.3.1.1 CORBA's CommonDataRepresentation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0 (OMG).

CDR can represent all the data types that can be used as arguments and return values in remote invocations in CORBA.

CORBA CDR data types:

Primitive types:

1. char, boolean, 2. octet (1 byte), 3. short, 4. unsigned short,5. wchar (2 byte), 5. long, 6. unsigned long, 7. float (4 byte), 8. long long, 9. unsigned long long,10. Double 11.long double byte), any

Constructed types**:**

1. sequence,2. string, 3. array, 4. struct, 5. enumerated , 6. Union

Sun XDR (eXternal Data Representation) is another example of data exchange standard. It is developed by Sun for use in messages exchanged between clients and servers in Sun NFS.

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}

Marshalling in CORBA

Marshalling in CORBA is generated automatically from the specification of the data to be transmitted. This standard data item can be specified in CORBA IDL (Interface Definition Language) as follows:

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

The CORBA interface compiler generates appropriate marshalling and umarshalling operations for the arguments and results of remote method.

2.3.1.2 Java object serialization

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations. An object is an instance of a Java class.The 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 n, String pl, int yr) {
        name = n;
        place = pl;
        year = yr;
} // followed by methods for accessing the instance variables. }
```

Java's object serialization

In Java, the term serialization refers to the activity of flattening an object or a connected set of objects into a serial form for storing or transmission.

Java's object de-serialization

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

- The information about a class consists of the name of the class and a version number, which is changed when major changes are made to the class.
- Java objects can contain references to other objects. These referred objects are serialized with the referring object.
- References are serialized as handles. The serialization must ensure 1-1 correspondence between object references and handles.
- To serialize an object is done as follows:
    - Its class name and version are written out, followed by its instance variables.
    - If the instance variables belong to new classes, their class information are also written out.
    - This recursive procedure continues until all necessary classes have been written out.
- The contents of primitive types are written in a portable binary format using methods of the ObjectOutputStream class.
- Strings and characters are written by its writeUTF method using Universal Transfer Format (UTF):
    - It enables ASCII characters to be represented unchanged (in one byte).
    - Unicode characters are represented by multiple bytes.
    - Strings are preceded by the number of bytes they occupy in the stream.
- The serialized form of Person p = new Person("Smith", "London", 1934) is illustrated in Figure .
- To make use of Java serialization, for example to serialize the Person object, create an instance of the class ObjectOutputStream and invoke its writeObject method, passing the Person object as argument.
- To deserialize an object from a stream of data, open an ObjectInputStream on the stream and use its readObject method to rebuild the original object.

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.

2.3.1.3. Extensible Markup Lan1guage (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. Both XML and HTML were derived from SGML (Standardized Generalized Markup Language) [ISO 8879], a very complex markup language.

- XML data items are tagged with 'markup' strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures.
- XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services.
- XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them.

XML elements and attributes • The following Figure shows the XML definition of the *Person* structure that was used to illustrate marshalling in CORBA CDR and Java.

Figure XML definition of the Person structure

<person id="123456789">

<name>Smith</name>

<place>London</place>

<year>1984</year>

<!-- a comment -->

</person >

It shows that XML consists of tags and character data. The character data, for example *Smith* or *1984*,is the actual data. As in HTML, the structure of an XML document is defined by pairs of tags enclosed in angle brackets. In above Figure, <*name*> and <*place*> are both tags.

Elements: An element in XML consists of a portion of character data surrounded by matching start and end tags. For example, one of the elements in Figure consists of the data *Smith* contained within the <*name*> ...

</*name*> tag pair. Note that the element with the <*name*> tag is enclosed in the element with the <*person id="123456789"*> ...</*person* > tag pair.

Attributes: A start tag may optionally include pairs of associated attribute names and values such as *id="123456789"*, as shown above as attributes. An element is generally a container for data, whereas an attribute issued for labeling that data. In our example, *123456789* might be an identifier used by the application, whereas *name, place* and *year* might be displayed.

Names: The names of tags and attributes in XML generally start with a letter, but can also start with an underline or a colon. The names continue with letters, digits, hyphens, underscores, colons or full stops. Letters are case-sensitive. Names that start with *xml*are reserved.

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

XML namespaces • Traditionally, namespaces provide a means for scoping names. An XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL. Any other

XML document can use an XML namespace by referring to its URL.

Any element that makes use of an XML namespace can specify that namespace asan attribute called *xmlns*, whose value is a URL referring to the file containing the namespace definitions. For example: *xmlns:pers = http://www.cdk5.net/person*

The name after *xmlns,* in this case *pers*can be used as a prefix to refer to the elements

in a particular namespace, as shown in following Figure. The *pers*prefix is bound to*http://www.cdk4.net/person* for the *person* element.

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>

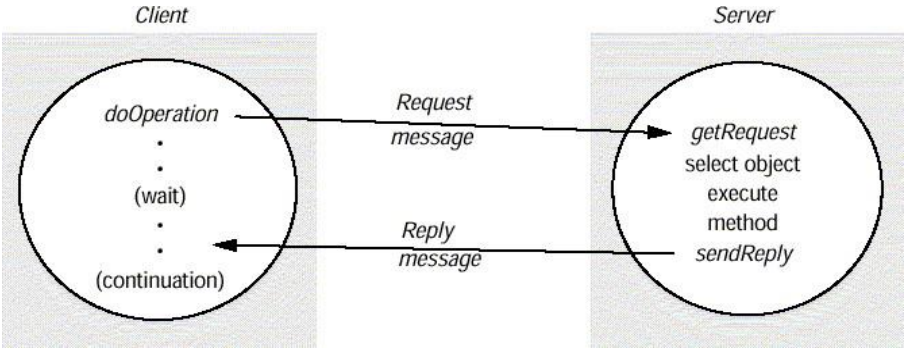
</person>

2.4 Client-Server Communication

The client-server communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server. Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.

1.The request-reply protocol

The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in following Figure.



The designed request-reply protocol matches requests to replies. If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.

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.

The following are the operations of Request-reply protocol

The information to be transmitted in a request message or a reply message is shown in following Figure.

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*// array of bytes*

The Request-reply protocol message structure contains the following.

- The first field indicates whether the message is a request or a reply message.
- The second field request id contains a message identifier.
- The third field is a remote object reference.
- The fourth field is an identifier for the method to be invoked.

Message identifier

A message identifier consists of two parts:

- A requestId, which is taken from an increasing sequence of integers by the sending process
- An identifier for the sender process, for example its port and Internet address.

Failure model of the request-reply protocol

If the three primitive *dooperation, getRequest,* and*sendReply* are implemented over UDP datagram, they have some communication failures.

- They suffer from omission failure
- Messages are not guaranteed to be delivered in sender order.

- Timeouts: There are various options as to what doOperation can do after a timeout. The simplest option is to return immediately from doOperation with an indication to the client that the doOperation has failed.The timeout may have been due to the request or reply messages getting lost and in the latter case, the operation will have been performed.
- Discarding duplicate request messages: In cases when the request message is retransmitted, the server may receive it more than once. To avoid this the protocol is designed to recognize successive messages with the same request identifier and to filter out duplicates.
- Lost reply messages: If the server has already sent the reply when it receives a duplicate request it will

need to execute the operation again to obtain the result, unless it has stored the result of the original execution. Some servers can execute their operations more than once and obtain the same results each time.

RPC exchange protocols

Three protocols are used for implementing various types of RPC.

- The request (R) protocol.
- The request-reply (RR) protocol.

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

- The request-reply-acknowledge (RRA) protocol.

In the R protocol, a single request message is sent by the client to the server.

The R protocol may be used when there is no value to be returned from the remote method.

The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol. Special acknowledgement messages are not required, because a server reply message is considered as an acknowledgement of the client's request message.

RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply. The acknowledgement reply message contains the requested from the reply message being acknowledged. This will enable the server to discard entries from its history.

HTTP: an example of a request-reply protocol

HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.

- Content Negotiation: Clients request can include information as to what data representation they can accept, enabling the server to choose the representation that is most appropriate for the user.
- Authentication: Credentials and challenges are used to support password style authentication on the first attempt to access a password protected area, the server reply contains a challenge applicable to the resource.

HTTP protocol steps are:

- Connection establishment between client and server at the default server port or at a port specified in the URL
- client sends a request message to the server
- server sends a reply message to the client
- Connection is closed

method	*URL*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure: HTTP Request message

HTTP methods

1.GET

- Requests the resource, identified by URL as argument.
- If the URL refers to data, then the web server replies by returning the data
- If the URL refers to a program, then the web server runs the program and returns the output to the client.

2.HEAD

- This method is similar to GET, but only meta data on resource is returned (like date of last modification, type, and size)

3. POST

- Specifies the URL of a resource (for instance, a server program) that can deal with the data supplied with the request.
- This method is designed to deal with:
- Providing a block of data to a data-handling process
- Posting a message to a bulletin board, mailing list or news group.
- Extending a dataset with an append operation

4. PUT

- Supplied data to be stored in the given URL as its identifier.

5. DELETE

- The server deletes an identified resource by the given URL on the server.

6. OPTIONS

- A server supplies the client with a list of methods.
- It allows to be applied to the given URL

7. TRACE

- The server sends back the request message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

HTTP reply message is shown below.

Above reply message specifies

2.5. Group Communication

The pair wise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary, for example, when a service is implemented as a number of different processes in different computers, perhaps to provide fault tolerance or to enhance availability. A

multicast operation is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired

Behavior of a multicast. The simplest multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1 .*Fault tolerance based on replicated services*: A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.

2. Finding the *Discovering services in spontaneous networking*: Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.

3. *Better performance through replicated data*: Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes,the new value is multicast to the processes managing the replicas.

4. *Propagation of event notifications*: Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers.

2.5.1. IP multicast – An implementation of multicast communication

IP multicast • *IP multicast* is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers – ports belong to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. A *multicast group* is specified by a Class D Internet address.

Being a member of a multicast group allows a computer to receive IP packets sent to the group. The membership of multicast **groups is dynamic**, allowing computers to join or leave at any time and to join an arbitrary number of groups. It is possible to send datagram's to a multicast group without being a member.

When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

***Multicast routers*:** IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single datagram's to routers on other networks, where they are again multicast to local members. To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass – called the *time to live*, or **TTL** for short.

***Multicast address allocation*:**

Class D addresses in IPV4 (that is,addresses in the range (**224.0.0.0 to 239.255.255.255)** are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA) and address start with first 4 bit 1110 .

Failure model for multicast datagrams

Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. That is, some but not all of the members of the group may receive it. This can be called *unreliable* multicast, because it does not guarantee that a message will be delivered to any member of a group.

Java API to IP multicast

The Java API provides a datagram interface to IP multicastthrough the class *MulticastSocket*, which is a subclass of *DatagramSocket* with theadditional capability of being able to join multicast groups. The class *MulticastSocket*provides two alternative constructors, allowing sockets to be created to use either aspecified local port (6789, in following figure) or any free local port. A process can join amulticast group with a given multicast address by invoking the *joinGroup()*method of itsmulticast socket. Effectively, the socket joins a multicast group at a given port and it willreceive datagrams sent by processes on other computers to that group at that port. Aprocess can leave a specified group by invoking the *leaveGroup()*method of its multicastsocket.

Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;

public class MulticastPeer{

public static void main(String args[]){

// args give message contents & destination multicast group (e.g. "228.5.6.7")
MulticastSocket s =null;

try {

InetAddress group = InetAddress.getByName(args[1]);
s = new MulticastSocket(6789);

s.joinGroup(group);

byte [] m = args[0].getBytes();

DatagramPacket messageOut =

new DatagramPacket(m, m.length, group, 6789);
s.send(messageOut);

byte[] buffer = new byte[1000];

for(int i=0; i< 3; i++) { // get messages from others in group
DatagramPacket messageIn =

new DatagramPacket(buffer, buffer.length);
s.receive(messageIn);
```

```
System.out.println("Received:" +new String(messageIn.getData()));
}
s.leaveGroup(group);
} catch (SocketException e){System.out.println("Socket: " + e.getMessage());
} catch (IOException e){System.out.println("IO: " + e.getMessage());
} finally { if(s != null) s.close();}
}}
```

2.5.1.1 Reliability and ordering of multicast

A datagram sent from one multicast router to another may be lost, thus preventing all recipients beyond that router from receiving the message. Also, when a multicast on a local area network uses the multicasting capabilities of the network to allow a single datagram to arrive at multiple recipients, any one of those recipients may drop the message because its buffer is full.

Another factor is that any process may fail. If a multicast router fails, the group members beyond that router will not receive the multicast message, although local members may do so. Ordering is another issue. IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent, with the possible effect that some group members receive datagrams from a single sender in a different order from other group members. In addition, messages sent by two different processes will not necessarily arrive in the same order at all the members of the group.

Some examples of the effects of reliability and ordering • We now consider the effect of the failure semantics of IP multicast as follows

1. ***Fault tolerance based on replicated services*:** Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others. In most cases, this service would require that all members receive request messages in the same order as one another.
2. ***Discovering services in spontaneous networking*:** One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services.
3. ***Better performance through replicated data*:** Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.
4. ***Propagation of event notifications*:** The particular application determines the qualities required of multicast.

Some applications require a multicast protocol that is more reliable than IP multicast. In particular, there is a need for *reliable multicast*, in which any message transmitted is either received by all members of a group or by none of them. The examples also suggest that some applications have strong requirements for ordering, the strictest of which is called *totally ordered multicast*, in which all of the messages transmitted to a group reach all of the members in the same order.