Syllabus:Interprocess Communication: Introduction, The API for the Internet Protocols- The Characteristics of Interprocess communication, Sockets, UDP Datagram Communication, TCP Stream Communication; External Data Representation and Marshalling; Client Server Communication; Group Communication- IP Multicast- an implementation of group communication, Reliability and Ordering of Multicast.

INTERPROCESS COMMUNICATION

Interposes 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.

The API for the Internet protocols

The characteristics of intercrosses communication:

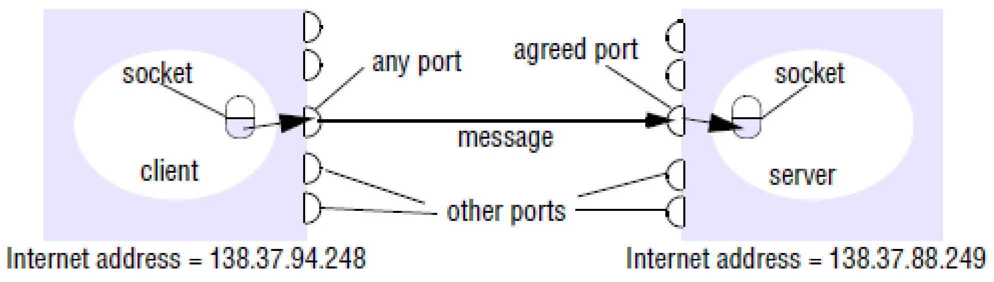
Message passing between a pair of processes can be supported by two message communication operations, *send* and *receive*, defined in terms of destinations and messages. To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity 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 very 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 *nonblocking* 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. In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

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.
→A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost.
→For integrity, messages must arrive uncorrupted and without duplication.
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. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process, as illustrated in Figure below.

Sockets and ports


socket
client
any port
agreed port
message
other ports
socket
server
Internet address = 138.37.94.248
Internet address = 138.37.88.249

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. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number (216) of possible port numbers for use by local processes forreceiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. (Processes using IP multicast are an exception in that they do share ports However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.
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 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. If the message is too big for the array ,it is truncated on arrival. The underlying IP protocol allows packet lengths of upto2^{16} bytes, which includes the headers as well as the message .However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment the min to chunks of that size.
- ***Blocking*:**
  Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication The *send* operation returns when it has handed the message to the underlying UDP and IP protocols ,which are responsible for transmitting it to its destination .On arrival ,the message is placed in a queue for the socket that is bound to the destination port. The method *receive* blocks until a datagram is received, unless a timeout has been set on the socket.If the process that invokes the *receive* method has other work to do while waiting for the message ,it should arrange to use a separate thread.
- ***Timeouts*:**
  The *receive* that blocks for ever is suitable for use by a server that is waiting to receiver requests from its clients. But in some programs, it is not appropriate that a process that has 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. Choosing an appropriate timeout interval is difficult, but it should be fairly large incomparison with the time required to transmit a message.

- ***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.

Failure model for UDP datagrams •

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 datagrams 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.
UDP clientsendsamessagetotheserverandgetsareply

```
importjava.net.*;
importjava.io.*;
publicclassUDPClient{
        publicstaticvoidmain(Stringargs[]){
    // argsgivemessagecontentsandserverhostname
    DatagramSocketaSocket= null;
    try{
        aSocket= newDatagramSocket();
        byte[]m=args[0].getBytes();
        InetAddressaHost= InetAddress.getByName(args[1]);
        intserverPort= 6789;
        DatagramPacketrequest=
                  newDatagramPacket(m,m.length(),aHost,serverPort);
        aSocket.send(request);
        byte[]buffer=newbyte[1000];
        DatagramPacketreply=newDatagramPacket(buffer,buffer.length);
        aSocket.receive(reply);
        System.out.println("Reply:" + newString(reply.getData()));
     }catch(SocketExceptione){System.out.println("Socket:"+e.getMessage());
     }catch(IOExceptione){System.out.println("IO:"+e.getMessage());
     }finally{if(aSocket!=null)aSocket.close();}
  }
}
```

***DatagramSocket*:**This class supportssockets for sending and receiving UDP datagrams.Itprovidesaconstructorthattakesaportnumberasitsargument,foruse byprocessesthatneedtouseaparticular port.Italsoprovidesano-argument constructorthatallowsthesystemtochooseafreelocalport.

Theclass*DatagramSocket*providesmethodsthatincludethefollowing:

***send*and*receive*:**Thesemethodsarefortransmittingdatagramsbetweenapair ofsockets.Theargumentof*send*isaninstanceof*DatagramPacket*containing amessageanditsdestination. Theargumentof*receive*isanempty *DatagramPacket* inwhichtoputthemessage,itslengthanditsorigin.The methods*send*and*receive*canthrow*IOExceptions*.

***setSoTimeout*:**Thismethodallowsatimeouttobeset.Withatimeoutset,the*receive*methodwillblockforthetimes pecifiedandthenthrowan*InterruptedIOException*.

***connect*:**Thismethodisusedforconnecting toaparticularremoteportand Internetaddress,inwhichcasethesocketisonlyabletosendmessagestoand receivemessagesfromthataddress.

UDP serverrepeatedlyreceivesarequestandsendsitbacktotheclient

```
importjava.net.*;
importjava.io.*;
publicclassUDPServer{
   publicstaticvoidmain(Stringargs[]){
     DatagramSocketaSocket= null;
     try{
          aSocket= newDatagramSocket(6789);
          byte[]buffer=newbyte[1000]; while(true){
            DatagramPacketrequest=newDatagramPacket(buffer,buffer.length);
             aSocket.receive(request);
             DatagramPacketreply=newDatagramPacket(request.getData(),
                  request.getLength(),request.getAddress(),request.getPort());
             aSocket.send(reply);
          }
     }catch(SocketExceptione){System.out.println("Socket:"+e.getMessage());
     }catch(IOExceptione){System.out.println("IO:"+e.getMessage());
     } finally{if(aSocket!=null)aSocket.close();}
   }
}
```

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*:**

A pair of communicating processes establish a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. 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.

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. Therefore, messages are guaranteed to be delivered even when some of the underlying packets are lost. But if the packet loss over a connection passes some limit or the network connecting a pair of communicating processes is severed or becomes severely congested, the TCP software responsible for sending messages will receive no acknowledgements and after a time will declare the connection to be broken. Thus TCP does not provide reliable communication, because it does not guarantee to deliver messages in the face of all possible difficulties.

Use of TCP • Many frequently used services run over TCP connections, with reserved port numbers. These include the following:

***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.

JavaAPIforTCPstreams•Th**e**JavainterfacetoTCPstreamsisprovidedintheclasses*ServerSocket*and*Socket*:

***ServerSocket*:**Thisclassisintendedforusebyaservertocreateasocketataserver portforlisteningfor*connect*requestsfromclients.Its *accept*methodgetsa*connect* requestfromthe queueor, ifthe queueis empty,blocksuntilonearrives.Theresult ofexecuting*accept*isaninstanceof*Socket*–asockettouseforcommunicatingwith theclient.

TCPclientmakesconnection toserver,sendsrequestandreceivesreply

```
importjava.net.*;
importjava.io.*;
publicclassTCPClient{
        publicstaticvoidmain(Stringargs[]){
    //argumentssupplymessageandhostnameofdestination
    Sockets=null;
    try{
      intserverPort=7896;
      s=newSocket(args[1],serverPort);
      DataInputStreamin=newDataInputStream(s.getInputStream());
      DataOutputStreamout=
              newDataOutputStream(s.getOutputStream());
      out.writeUTF(args[0]);     //UTFis astringencoding;seeSec4.3
      Stringdata= in.readUTF();
      System.out.println("Received:"+data);
    }catch(UnknownHostExceptione){
      System.out.println("Sock:"+e.getMessage());
    } catch(EOFExceptione){System.out.println("EOF:"+e.getMessage());
    } catch(IOExceptione){System.out.println("IO:"+e.getMessage());
    }finally{if(s!=null)try{s.close();}catch(IOExceptione){/*closefailed*/}}
  }
}
```

The***Socket***classprovidesthemethods*getInputStream*and*getOutputStream*foraccessingthetwostreamsassociated withasocket.Thereturntypesofthese methodsare*InputStream*and*OutputStream*, respectively–abstractclassesthat definemethods forreading andwritingbytes.

TCPservermakesaconnection foreachclientandthenechoestheclient'srequest

```
importjava.net.*;
importjava.io.*;
publicclassTCPServer{
   publicstaticvoidmain(Stringargs[]){
           try{
      intserverPort= 7896;
      ServerSocketlistenSocket= newServerSocket(serverPort);
      while(true){
        SocketclientSocket=listenSocket.accept(); Connectionc
        = newConnection(clientSocket);
      }
    } catch(IOExceptione){System.out.println("Listen:"+e.getMessage());}
  }
 }
```

```
classConnectionextendsThread{
    DataInputStreamin;
    DataOutputStreamout;
    SocketclientSocket;
    publicConnection(SocketaClientSocket){
      try{
        clientSocket=aClientSocket;
        in=newDataInputStream(clientSocket.getInputStream());
        out=newDataOutputStream(clientSocket.getOutputStream());
        this.start();
      }catch(IOExceptione){System.out.println("Connection:"+e.getMessage());}
    }
    publicvoidrun(){
      try{                    // anechoserver
        Stringdata= in.readUTF();
        out.writeUTF(data);
      } catch(EOFExceptione){System.out.println("EOF:"+e.getMessage());
      } catch(IOExceptione){System.out.println("IO:"+e.getMessage());
      } finally{try{clientSocket.close();}catch(IOExceptione){/*closefailed*/}}
    }
}
```

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.

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.

Inthefirsttwocases,themarshallingandunmarshalling activitiesareintendedtobe carriedoutbyamiddlewarelayerwithoutanyinvolvementonthepartoftheapplication programmer.

Inthefirsttwoapproaches,theprimitivedatatypesaremarshalledintoabinary form.Inthethirdapproach(XML),theprimitivedatatypesarerepresented textually.

CORBA'sCommonDataRepresentation (CDR)

CORBACDRistheexternaldatarepresentationdefinedwithCORBA2.0.

Theseconsistof15primitivetypes,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*.

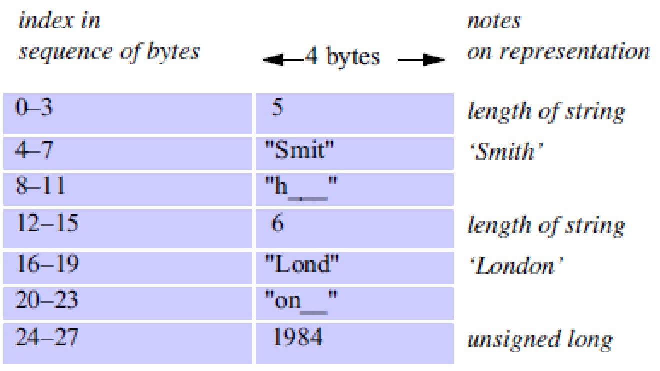
***Primitivetypes*:**CDRdefinesarepresentationforbothbig-endianandlittle-endian orderings.Valuesaretransmittedinthesender'sordering,whichisspecifiedineach message.Therecipienttranslatesifitrequiresadifferentordering.Forexample,a16-bit*short*occupiestwobytesinthemessage,andforbig-endianordering,themost significantbits occupythefirst byte and theleast significantbits occupythesecond byte.

***Constructedtypes*:**Theprimitivevaluesthatcompriseeachconstructedtypeare addedtoa sequenceofbytesinaparticularorder,asshownin following Figure.

CORBACDRforconstructedtypes

Type	*Representation*
sequence	length(unsignedlong)followedbyelementsinorder
string	length(unsignedlong)followedbycharactersinorder(canalso havewidecharacters)
array	arrayelementsinorder(nolengthspecifiedbecauseitisfixed)
struct	intheorderofdeclarationofthecomponents
enumerated	unsignedlong(thevaluesarespecifiedbytheorderdeclared)
union	typetagfollowedbytheselectedmember

The following figure showsamessageinCORBA CDRthatcontains thethreefieldsofa*struct* whoserespective typesare*string*,*string*and*unsignedlong*.



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}

MarshallinginCORBA• Marshallingoperationscanbegeneratedautomaticallyfrom the specificationof the typesof dataitemstobetransmittedina message.

Forexample, wemightuseCORBA IDL(Interface Definition language)todescribethe datastructureinthemessageof above Figureasfollows:

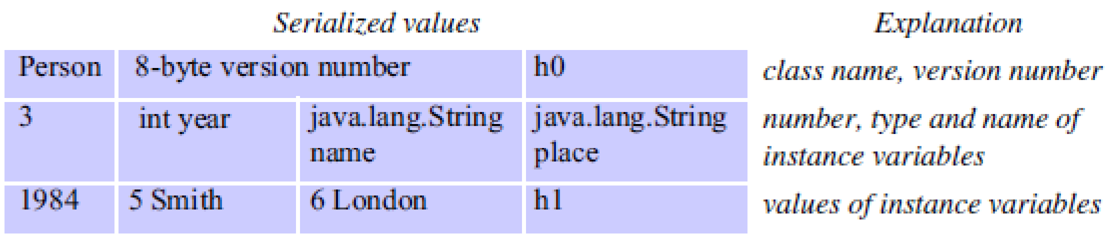
```
structPerson{ stringname;
      stringplace;
      unsignedlongyear;
};
```

Javaobjectserialization

InJavaRMI,bothobjectsandprimitivedatavaluesmaybepassedasarguments and resultsofmethodinvocations.AnobjectisaninstanceofaJavaclass.Forexample,the Javaclassequivalenttothe*Personstruct*definedinCORBAIDLmightbe:

```
publicclassPersonimplementsSerializable{
    privateStringname;
    privateStringplace;
    privateintyear;
    publicPerson(StringaName,StringaPlace,intaYear){
        name=aName;
        place=aPlace; year=
        aYear;
    }
    // followedbymethodsforaccessingtheinstancevariables
}
```

- ✓ The above class states that it implements serializable interface.
- ✓ In java, serialization means flattening an object or a set of objects into a serial form suitable for storing on a disk or transmitting in a message.
- ✓ Deserialization is the restoring of object from serialized form.
- ✓ Information about class(like name, version etc.) are included in serializable form so that it is helpful in deserialization process.
- ✓ Version numbers is intended to change when major changes are made to the class.(usually set by programmer).
- ✓ To serialize an object, its class information is written out followed by the types and names of its instance variables.
- ✓ Each class is given a handle(reference to an object within serialized form).
- ✓ Example, consider serialization of following object
- ✓ Person p=new Person("Smith","London",1934);



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 atextual encoding that represents both a text and details as to its structure or itsappearance. Both XML and HTML were derived from SGML (StandardizedGeneralized Markup Language) [ISO 8879], a very complex markup language.

→XML data items are tagged with 'markup' strings. The tags are used to describethe logical structure of the data and to associate attribute-value pairs with logicalstructures.

→XML is used to enable clients to communicate with web services and for definingthe interfaces and other properties of web services.

→XML is *extensible* in the sense that users can define their own tags, in contrast toHTML, which uses a fixed set of tags. However, if an XML document is intended to beused by more than one application, then the names of the tags must be agreed betweenthem.

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 thatXML 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 pairsof 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 bymatching start and end tags. For example, one of the elements in Figure consists ofthe data *Smith* contained within the <*name*> ... </*name*> tag pair. Note that the elementwith 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 andvalues such as *id="123456789"*, as shown above as attributes. An element is generally a container for data, whereas an attribute isused for labelling that data. In our example, *123456789* might be an identifier used bythe application, whereas *name*, *place* and *year* might be displayed.

Names: The names of tags and attributes in XML generally start with a letter, but canalso 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 characterdata.

XML namespaces • Traditionally, namespaces provide a means for scoping names. AnXML namespace is a set of names for a collection of element types and attributes that isreferenced by a URL. Any other XML document can use an XML namespace byreferring 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 thenamespace 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

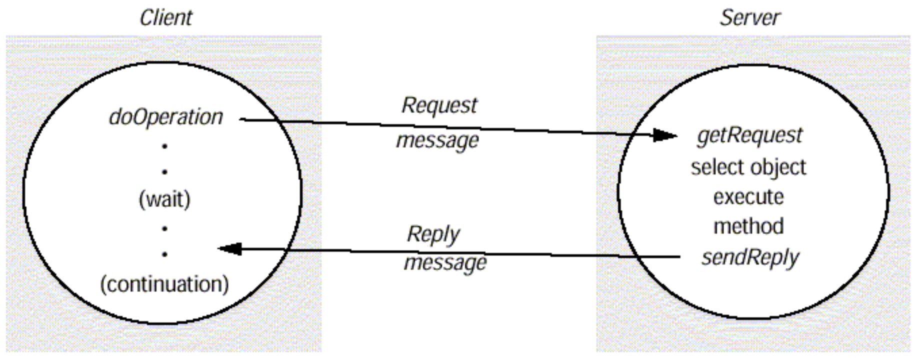
```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
<pers:name> Smith </pers:name>
<pers:place> London </pers:place>
<pers:year> 1984 </pers:year>
    </person>
```

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.

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.

The following Figureoutlines the three communication primitives.

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 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*

TheRequest-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. Such as,

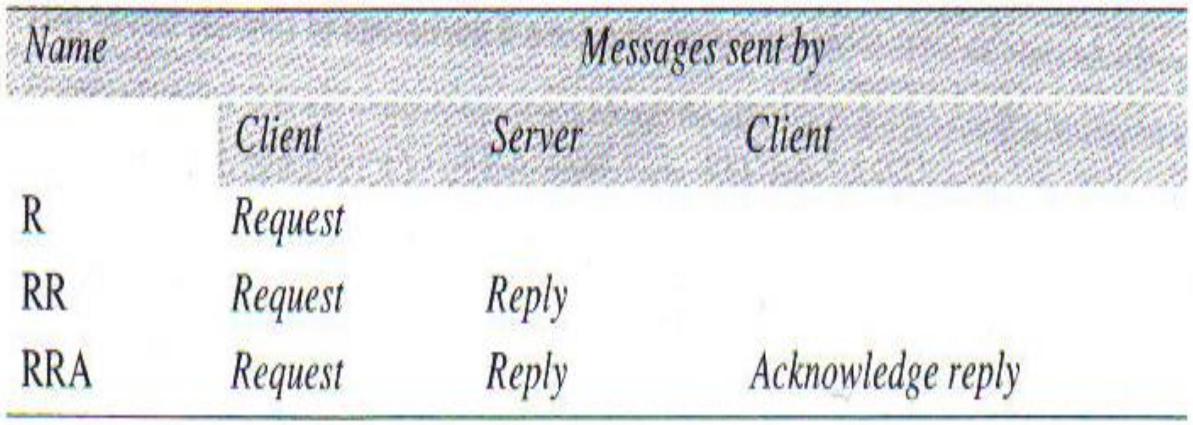
→omission failure

→Messages are not guaranteed to be delivered in sender order.

RPC exchange protocols

Three protocols are used for implementing various types of RPC.

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



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

→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.

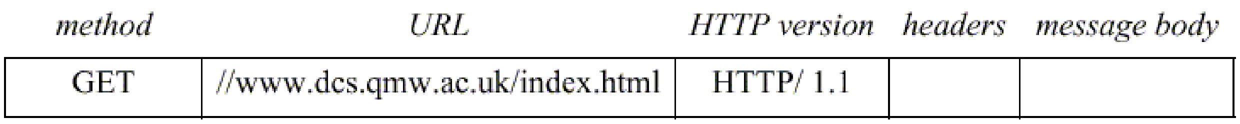
HTTP: an example of a request-reply protocol

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

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

HTTP request message is shown below.

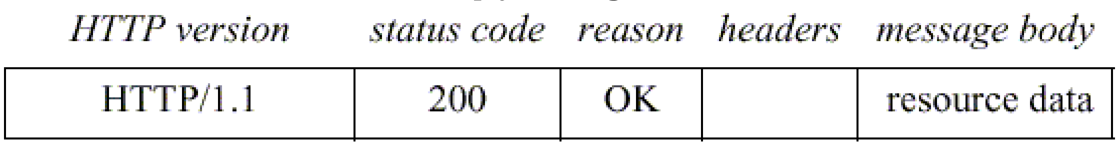


method	URL	HTTP version	headers	message body
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure :. HTTP request message

- **HTTP methods**
  - **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.
  - **HEAD**
    - This method is similar to GET, but only meta data on resource is returned (like date of last modification, type, and size)
  - **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
  - **PUT**
    - Supplied data to be stored in the given URL as its identifier.
  - **DELETE**
    - The server deletes an identified resource by the given URL on the server.
  - **OPTIONS**
    - A server supplies the client with a list of methods.
    - It allows to be applied to the given URL
  - **TRACE**
    - The server sends back the request message

HTTP reply message is shown below.



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

Above reply message specifies

- The protocol version
- A status code
- Reason
- Some headers
- An optional message body

Group Communication

The pairwise 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 Behaviour 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.*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.

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 datagrams 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 datagrams 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 – calledThe*time to live*, or TTL for short.

Multicast address allocation**:** Class D addresses (that is,addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast trafficand managed globally by the Internet Assigned Numbers Authority (IANA).

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.

Figure 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();}
}
}
```

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.