Solutions-IAT-1

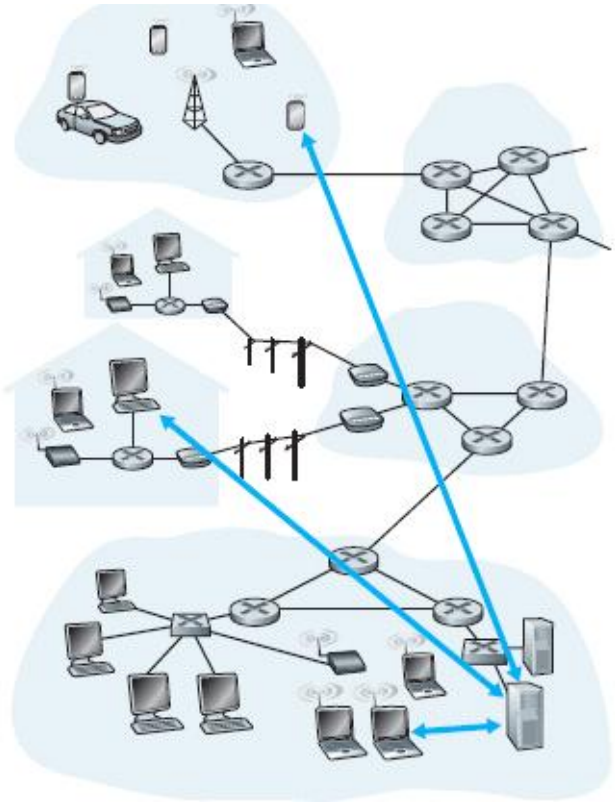
Computer Networks

1. Explain different types of Network A pplication architectures
Answer:

The **application architecture**, dictates how the application is structured over the various end systems. In choosing the application architecture, an application developer will likely draw on one of the two predominant architectural paradigms used in modern network applications

- Client-Server architecture
- Peer-to-peer (P2P) architecture

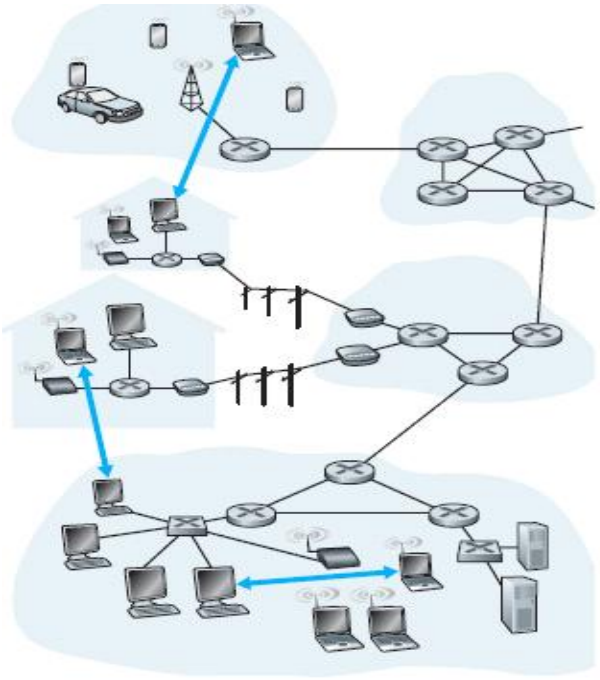
1. Client Server architecture:



In a **client-server architecture**, there is an always-on host, called the *server*, which services requests from many of the hosts, called *clients*. A classic example is the Web application for which an always-on Web server services requests from browsers running on client hosts.

When a Web server receives a request for an object from a client host, it responds by sending the requested object to the client host. Note that with the client-server architecture, clients do not directly communicate with each other; for example, in the Web application, two browsers do not directly communicate. Another characteristic of the client-server architecture is that the server has a fixed, well-known address, called an IP address (which we'll discuss soon). Because the server has a fixed, well-known address, and because the server is always on, a client can always contact the server by sending a packet to the server's IP address. Some of the better-known applications with a client-server architecture include the Web, FTP, Telnet, and e-mail.

2. Peer-to peer Architecture: The figure given below depicts the architecture of peer to peer model.



In a **P2P architecture**, there is minimal (or no) reliance on dedicated servers in data centers. Instead the application exploits direct communication between pairs of intermittently connected hosts, called *peers*.

The peers are not owned by the service provider, but are instead desktops and laptops controlled by users, with most of the peers residing in homes, universities, and offices. Because the peers

communicate without passing through a dedicated server, the **architecture is called peer-to-peer**. Many of today's most popular and traffic-intensive applications are based on P2P architectures. These applications include file sharing (e.g., BitTorrent), peer-assisted download acceleration (e.g., Xunlei), Internet Telephony (e.g., Skype), and IPTV (e.g., Kankan and PPstream).

One of the most compelling features of P2P architectures is their **self-scalability**. For example, in a P2P file-sharing application, although each peer generates workload by requesting files, each peer also adds service capacity to the system by distributing files to other peers. P2P architectures are also cost effective, since

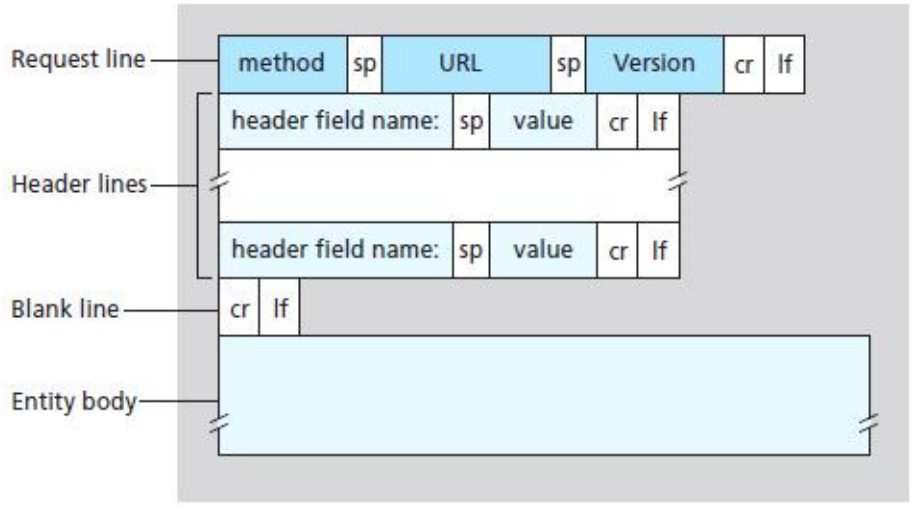
they normally don't require significant server infrastructure and server bandwidth.

2. Explain HTTP request and response message formats with the relevant diagrams

Answer:

The HTTP specifications [RFC 1945; RFC 2616] include the definitions of the HTTP message formats. There are two types of HTTP messages, request messages and response messages, both of which are discussed below.

HTTP Request Message Format:



The first line of an HTTP request message is called the **request line**; the subsequent lines are called the **header lines**. The request line has three fields: the method field, the URL field, and

the HTTP version field. The method field can take on several different values, including GET, POST, HEAD, PUT, and DELETE.

The great majority of HTTP request messages use the GET method. The GET method is used when the browser requests an object, with the requested object identified in the URL field.

Example Message request in HTTP:

GET /somedir/page.html HTTP/1.1

Host: www.someschool.edu

Connection: close

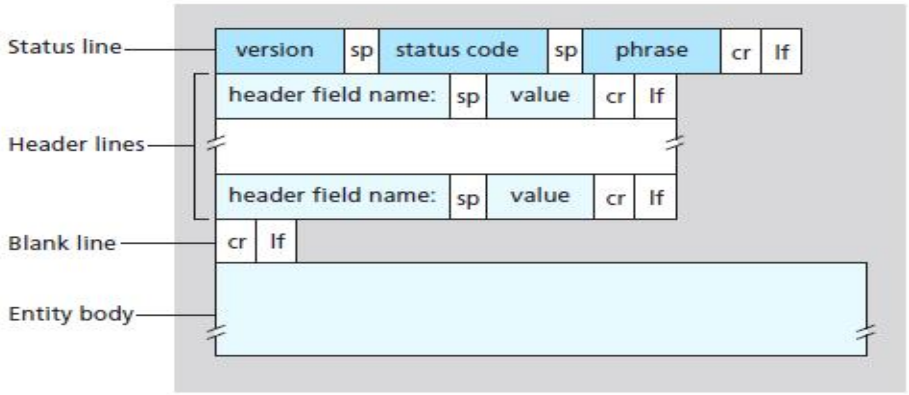
User-agent: Mozilla/5.0

Accept-language: fr

In this **Request line**, the browser is requesting the object /somedir/page.html. The version is self-explanatory; in this example, the browser implements version HTTP/1.1.

Now let's look at the header lines in the example. The header line **Host:** www.someschool.edu specifies the host on which the object resides. By including the **Connection:** close header line, the browser is telling the server that it doesn't want to bother with persistent connections; it wants the server to close the connection after sending the requested object. The **User-agent:** header line specifies the user agent, that is, the browser type that is making the request to the server. Here the user agent is Mozilla/5.0, a Firefox browser. This header line is useful because the server can actually send different versions of the same object to different types of user agents. Finally, the **Accept language:** header indicates that the user prefers to receive a French version of the object, if such an object exists on the server; otherwise, the server should send its default version. The Accept-language: header is just one of many content negotiation headers available in HTTP.

HTTP Response Message Format: Below we provide a typical HTTP response message.



It has three sections: an **initial status line**, six **header lines**, and then the **entity body**. The entity body is the

meat of the message it contains the requested object itself (represented by data data data data data ...). The status line has three fields: the protocol version field, a status code, and a corresponding status message. Entity body contains the actual data.

HTTP/1.1 200 OK

Connection: close

Date: Tue, 09 Aug 2011 15:44:04 GMT

Server: Apache/2.2.3 (CentOS)

Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT

Content-Length: 6821

Content-Type: text/html

(data data data data data ...)

In this example, the status line indicates that the server is using HTTP/1.1 and that everything is OK

(that is, the server has found, and is sending, the requested object). Now let's look at the header lines. The server uses the Connection: close header line to tell the client that it is going to close the TCP connection after sending the message. The Date: header line indicates the time and date when the HTTP response was created and sent by the server. Note that this is not the time when the object was created or last modified; it is the time when the server retrieves the object from its file system, inserts the object into the response message, and sends the response message. The Server: header line indicates that the message was generated by an Apache Web server; it is analogous to the User-agent: header line in the HTTP request message. The Last-Modified:

header line indicates the time and date when the object was created or last modified. The Last-Modified: header, which we will soon cover in more detail, is critical for object caching, both in the local client and in network cache servers (also known as proxy servers). The Content-Length: header line indicates the number of bytes in the object being sent. The Content-Type: header line indicates that the object in the entity body is HTML text. (The object type is officially indicated by the Content-Type: header and not by the file extension.)

3. Write short note on a). Cookies b). Web cache

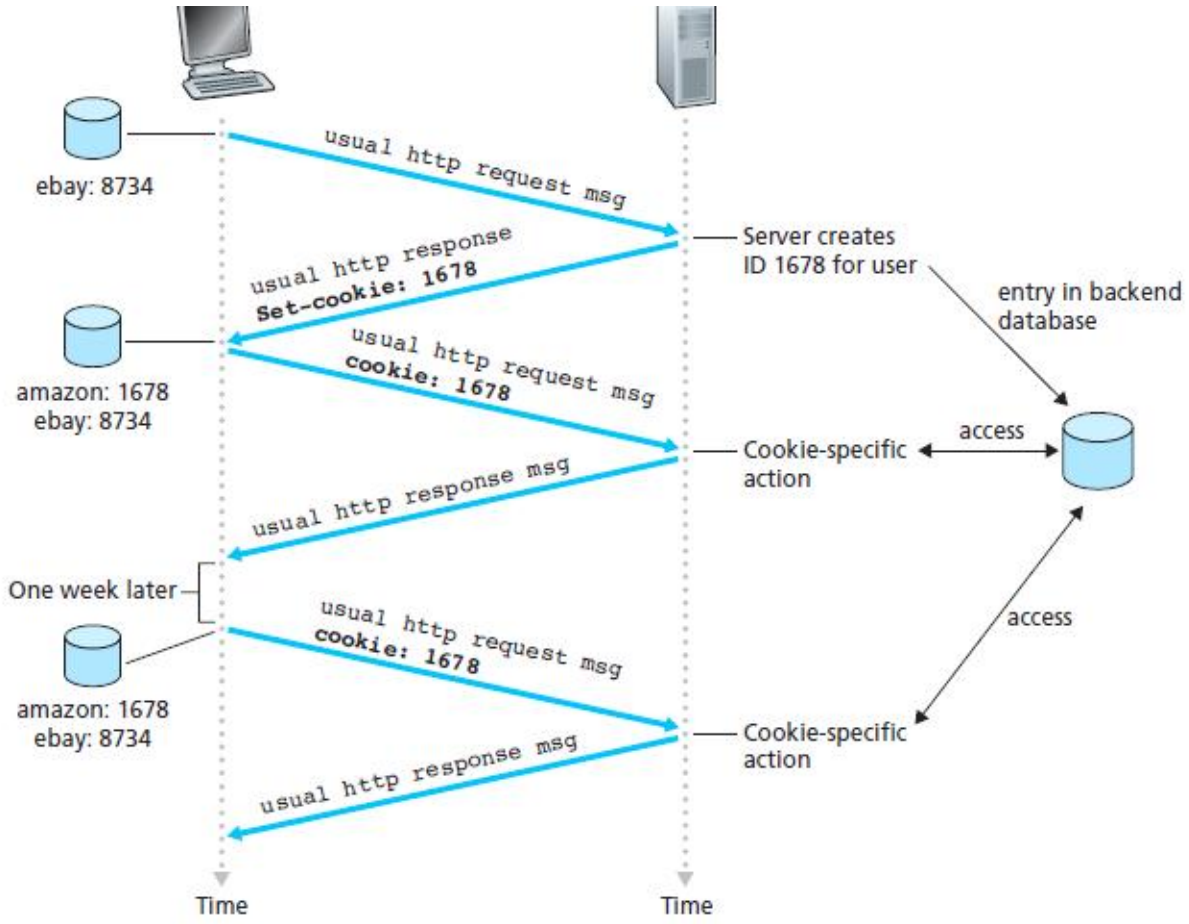
a). Cookies

It is often desirable for a Web site to identify users, either because the server wishes to restrict user access or

because it wants to serve content as a function of the user identity. For these purposes, HTTP uses cookies**.**

Cookie is a small piece of information the website makes the broser to store in the client system for further reference.

The following diagram depicts how the cookie works.



As shown in the above given figure, cookie technology has four components:

(1) a cookie header line in the HTTP response message;

(2) a cookie header line in the HTTP request message;

(3) a cookie file kept on the user's end system and managed by the user's browser; and

(4) a back-end database at the Web site.

Using the figure given above, let's walk through an example of how cookies work. Suppose Susan, who always accesses the Web using Internet Explorer from her home PC, contacts Amazon.com for the first time. Let us suppose that in the past she has already visited the eBay site. When the request comes into the Amazon Web server, the server creates a unique identification number and creates an entry in its back-end database that is indexed by the identification number. The Amazon Web server then responds to Susan's browser, including in the HTTP response a Set-cookie: header, which contains the identification number. For example, the header line might be: Set-cookie: 1678 When Susan's browser receives the HTTP response message, it sees the Setcookie: header. The browser then appends a line to the special cookie file

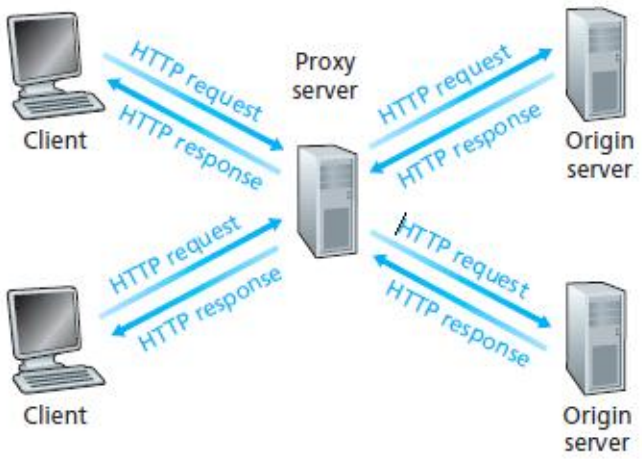
that it manages. This line includes the hostname of the server and the identification number in the Set-cookie: header. Note that the cookie file already has an entry **for eBay, since Susan has visited that site in the past. As Susan continues** to browse the Amazon site, each time she requests a Web page, her browser consults her cookie file, extracts her identification number for this site, and puts a cookie header line that includes the identification number in the HTTP request. Specifically, each of her HTTP requests to the Amazon server includes the header line: Cookie: 1678, In this manner, the Amazon server is able to track Susan's activity at the Amazon site.

The first time a user visits a site, the user can provide a user identification (possibly his or her name). During the subsequent sessions, the browser passes a cookie header to the server, thereby identifying the user to the server. Cookies can thus be used to create a user session layer on top of stateless HTTP. For example, when a user logs in to a Web-based e-mail application (such as Hotmail), the browser sends cookie information to the server, permitting the server to identify the user throughout the user's session with the application.

b). Web cache

A **Web cache** also called a **proxy server** is a network entity that satisfies HTTP requests on the behalf of an origin Web server. The Web cache has its own disk storage and keeps copies of recently requested objects in this storage. A user's browser can be configured so that all of the user's HTTP requests are first directed to the Web cache. Once a browser is configured, each browser request for an object is first directed to the Web cache.

The following figure illustrates the working of a web cache.



As an example, suppose a browser is requesting the object http://www.someschool.edu/campus.gif. Here is what happens:

1. The browser establishes a TCP connection to the Web cache and sends an HTTP request for the object to the Web cache.
2. The Web cache checks to see if it has a copy of the object stored locally. If it does, the Web cache returns the object within an HTTP response message to the client browser.
3. If the Web cache does not have the object, the Web cache opens a TCP connection to the origin server, that is, to www.someschool.edu. The Web cache then sends an HTTP request for the object into the cache-to-server TCP connection. After receiving this request, the origin server sends the object within an HTTP response to the Web cache.
4. When the Web cache receives the object, it stores a copy in its local storage and sends a copy, within an HTTP response message, to the client browser (over the existing TCP connection between the client browser and the Web cache).

Note that a cache is both a server and a client at the same time. When it receives requests from and sends responses to a browser, it is a server. When it sends requests to and receives responses from an origin server, it is a client.

4. a). Explain why FTP is called as 'Out-of-band Protocol' and why SMTP is called 'Push Protocol'?

b). What is DHT? Explain the working of Circular DHT

Solution:

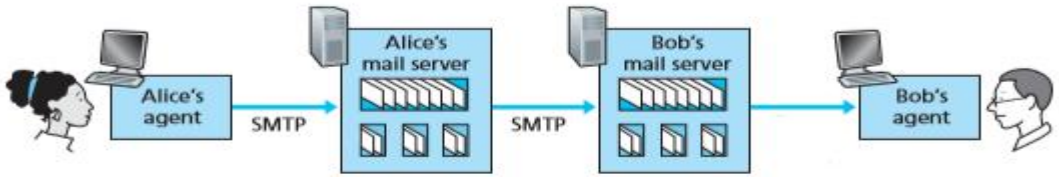
a) FTP is out-of-band protocol:

FTP is called outbound protocol because it uses two parallel TCP connections to transfer a file, a control connection and a data connection. The control connection is used for sending control information between the two hosts—information such as user identification, password, commands to change remote directory, and commands to "put" and "get" files. The data connection is used to actually send a file. Because FTP uses a separate control connection, FTP is said to send its control information out-of-band.

SMTP is called 'Push Protocol':

SMTP or Simple Mail Transfer Protocol is used to transfer mails from user's browser to user's mail server and from sender's mail server to receiver's mail server. So SMTP is used to push the

mails basically, one time from sender to sender's mail server and another time from sender's mail server to receiver's mail server. So SMTP is called push protocol.



.

b) DHT-Circular DHT:

DHT is a distributed Database used in Peer to peer networks. In the P2P system, each peer will only hold a

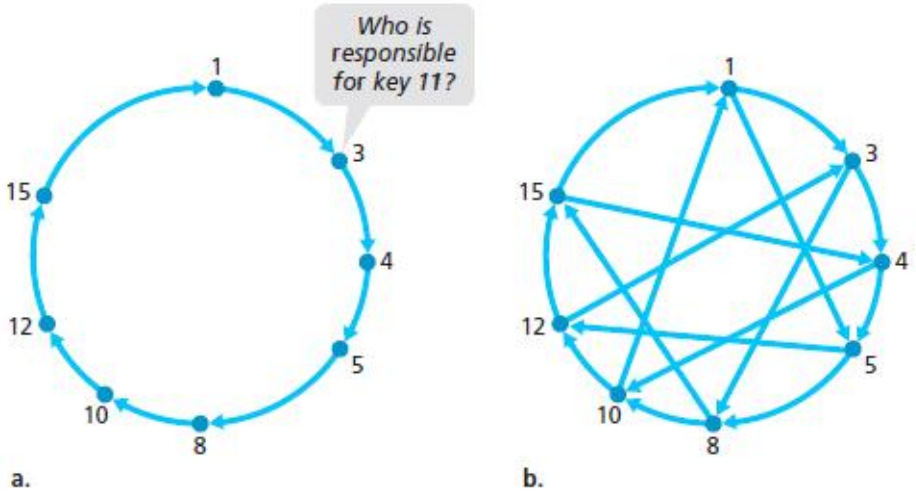
small subset of the totality of the (key, value) pairs. We'll allow any peer to query the distributed database with a particular key. The distributed database will then locate the peers that have the corresponding (key, value) pairs and return the key-value pairs to the querying peer. Any peer will also be allowed to insert new key-value pairs into the database. Such a distributed database is referred to as a **distributed hash table (DHT)**.

Circular DHT

Suppose a peer, Alice, wants to insert a (key, value) pair into the DHT. Conceptually, this is straightforward: She first determines the peer whose identifier is closest to the key; she then sends a message to that peer, instructing it to store the (key, value) pair. But how does Alice determine the peer that is closest to the key? If

Alice were to keep track of all the peers in the system (peer IDs and corresponding IP addresses), she could locally determine the closest peer. But such an approach requires e*ach* peer to keep track of *all* other peers in the DHT which is completely impractical for a large-scale system with millions of peers.

To address this problem of scale, let's now consider organizing the peers into a circle. In this circular arrangement, each peer only keeps track of its immediate successor and immediate predecessor (modulo 2*n*). An example of such a circle is shown in figure given below.



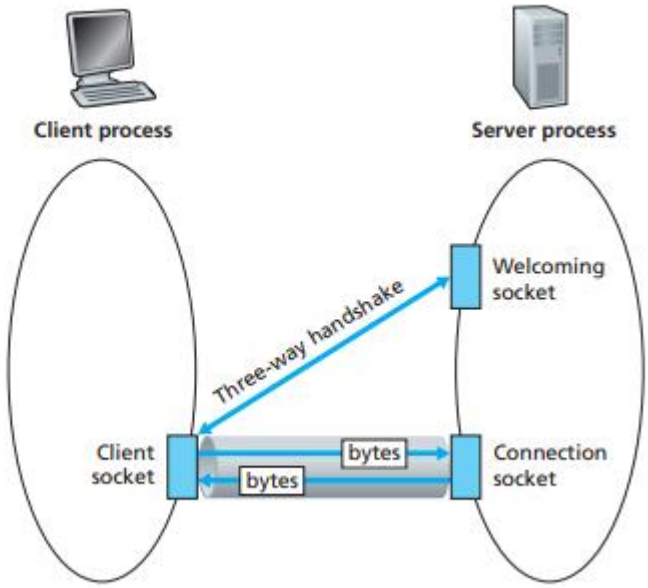
In this example, *n* is again 4 and there are the same eight peers from the previous example. Each peer is only aware of its immediate successor and predecessor; for example, peer 5 knows the IP address and identifier for

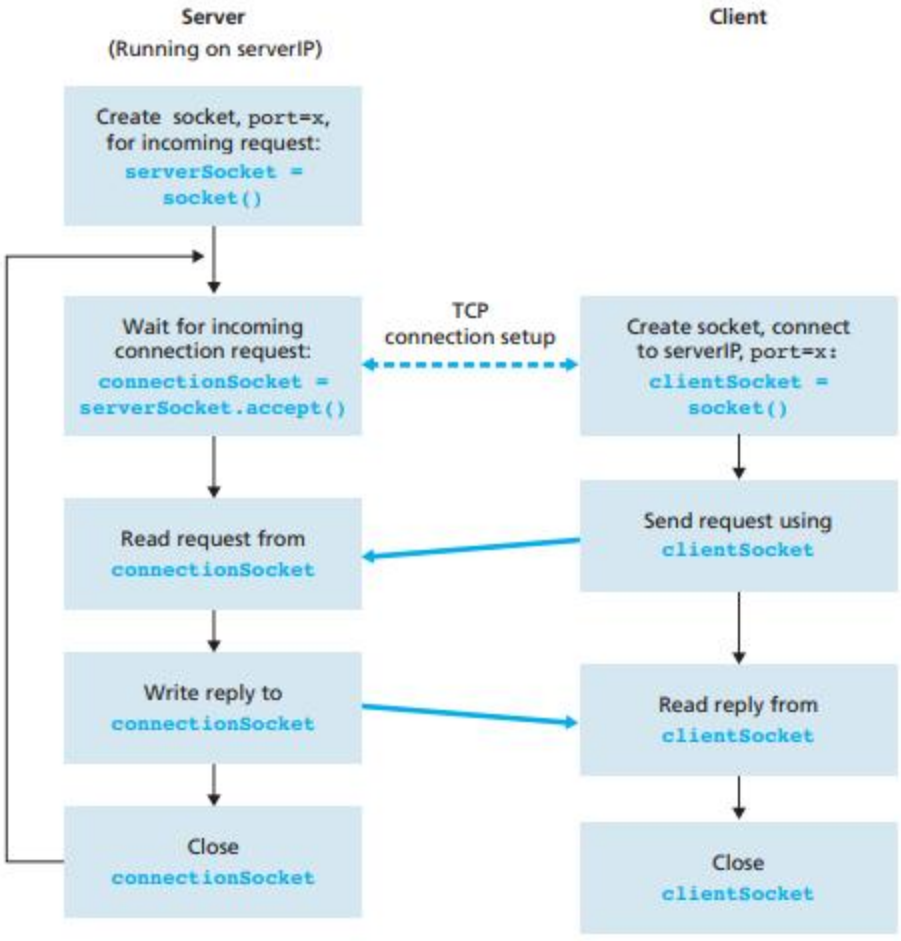
peers 8 and 4 but does not necessarily know anything about any other peers that may be in the DHT. This circular arrangement of the peers is a special case of an **overlay network**. In an overlay network, the peers form an abstract logical network which resides above the "underlay" computer network consisting of physical links, routers, and hosts. The links in an overlay network are not physical links, but are simply virtual

liaisons between pairs of peers.

5. Design and implement a network application for client server communication using sockets over TCP .

Here we are going to design a client server web application model where client makes a request to get a content of the file by providing the name of the file, to the server and server will reply to client be sending the content of the file. Two processes, namely client process and server process will communicate with each other by TCP Transport layer protocol. As TCP will be connection oriented protocol so before exchanging of request and response connection to be establish between client and server. Server will be having two sockets, one is for connection establishment and another one for accepting request and sending the response and client process consists of one socket to send request and receive response.


Client process
Server process
Welcoming socket
Three-way handshake
Client socket
bytes
bytes
Connection socket


Server
(Running on serverIP)
Client
Create socket, port=x, for incoming request: serverSocket = socket()
Wait for incoming connection request: connectionSocket = serverSocket.accept()
TCP connection setup
Create socket, connect to serverIP, port=x: clientSocket = socket()
Read request from connectionSocket
Send request using clientSocket
Write reply to connectionSocket
Read reply from clientSocket
Close connectionSocket
Close clientSocket

Server.java

```
import java.net.*;
import java.io.*;
public class ContentsServer
{
  public static void main(String args[]) throws Exception
  {                            // establishing the connection with the
server
     ServerSocket sersock = new ServerSocket(4000);
     System.out.println("Server ready for connection");
     Socket sock = sersock.accept();            // binding with port:
4000
     System.out.println("Connection is successful and wating for
chatting");

                               // reading the file name from client
     InputStream istream = sock.getInputStream( );
     BufferedReader fileRead =new BufferedReader(new
InputStreamReader(istream));
     String fname = fileRead.readLine( );
                              // reading file contents
     BufferedReader contentRead = new BufferedReader(new
FileReader(fname) );

                           // keeping output stream ready to send the
contents
     OutputStream ostream = sock.getOutputStream( );
     PrintWriter pwrite = new PrintWriter(ostream, true);

     String str;
     while((str = contentRead.readLine()) !=  null) // reading line-
by-line from file
     {
pwrite.println(str);         // sending each line to client
     }

     sock.close();  sersock.close();       // closing network sockets
     pwrite.close();  fileRead.close(); contentRead.close();
  }
}
```

Client.java

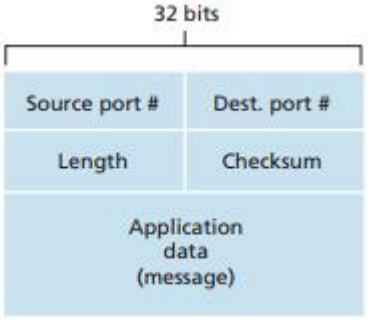
```
public class ContentsClient
{
  public static void main( String args[ ] ) throws Exception
  {
     Socket sock = new Socket( "127.0.0.1", 4000);

                    // reading the file name from keyboard. Uses input
stream
```

```
     System.out.print("Enter the file name");
     BufferedReader keyRead = new BufferedReader(new
InputStreamReader(System.in));
     String fname = keyRead.readLine();

        // sending the file name to server. Uses PrintWriter
     OutputStream  ostream = sock.getOutputStream( );
     PrintWriter pwrite = new PrintWriter(ostream, true);
     pwrite.println(fname);
                    // receiving the contents from server.  Uses
input stream
     InputStream istream = sock.getInputStream();
     BufferedReader socketRead = new BufferedReader(new
InputStreamReader(istream));

     String str;
     while((str = socketRead.readLine())  !=  null) // reading line-
by-line
     {
         System.out.println(str);
     }
     pwrite.close(); socketRead.close(); keyRead.close();
  }
}
```

6 a) With a neat diagram, explain UDP segment structure.

b)With an example, explain how to generate UDP checksum and how the generated checksum is used to detect the transmission errors at the receiver end?(Note: Take any four 16 bit data blocks as input)

a)



The UDP segment structure, shown in Figure above.

The *application data* occupies the data field of the UDP segment. The UDP header has only four fields, each consisting of two bytes

The *port numbers* allow the destination host to pass the application data to the correct process running on the destination end system (that is, to perform the demultiplexing function).

The *length* field specifies the number of bytes in the UDP segment (header plus data). An explicit length value is needed since the size of the data field may differ from one UDP segment to the next. Thelength field specifies the length of the UDP segment, including the header, in bytes

The *checksum* is used by the receiving host to check whether errors have been introduced into the segment.

b) The UDP checksum provides for error detection. That is, the checksum is used to determine whether bits within the UDP segment have been altered (for example, by noise in the links or while stored in a router) as it moved from source to destination. Let's discuss the functionality of UDP checksum with the following example.

Sender side

UDP at the sender side performs the 1s complement of the sum of all the 16-bit
words in the segment, with any overflow encountered during the sum being wrapped around.

Consider 3 16 bits words are application data say W1=0110011001100000, W2=0101010101010101, W3=1000111100001100 , W4=0000100101000000

The sum of first two of these 16-bit words is

0110011001100000

0101010101010101

1011101110110101

Adding the third word to the above sum gives

1011101110110101

1000111100001100

0100101011000010

Note that this last addition had overflow, which was wrapped around .

Adding the fourth word with above sum gives,

0100101011000010

0000100101000000

0101010000000010

Now the 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus the 1s complement of the sum 0101010000000010 is 1010101111111101, which becomes the checksum.

Receiver Side

At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111. If one of the bits is a 0, then we know that errors have been introduced into the packet.

Consider in receiver side we got W1=0110011001100000, W2=0101010101010101, W3=1000111100001100 and W4=0000100101000000 (with no error). Now if we add W1, W2, W3, checksum we will get

0110011001100000

0101010101010101

1011101110110101

Adding the third word to the above sum gives

1011101110110101

1000111100001100

0100101011000010

Adding the fourth word with above sum gives,

0100101011000010

0000100101000000

0101010000000010

Adding Checksum with above result

0101010000000010

1010101111111101

1111111111111111 (which means no error)

If any error has occurred, say in W4 = 1000100101000000 then, we got W1 + W2 + W3= 0100101011000010.

Adding the fourth word W4 with above sum gives,

0100101011000010

1000100101000000

1101010000000010

Adding Checksum with above result

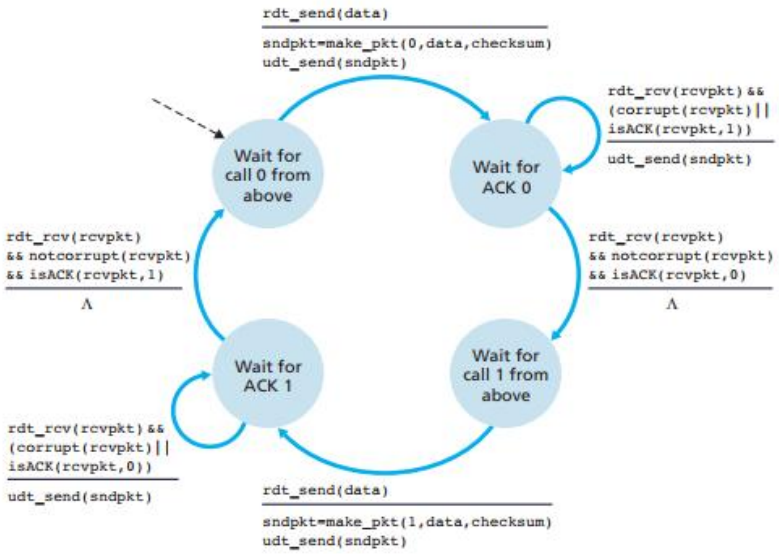
1101010000000010

1010101111111101

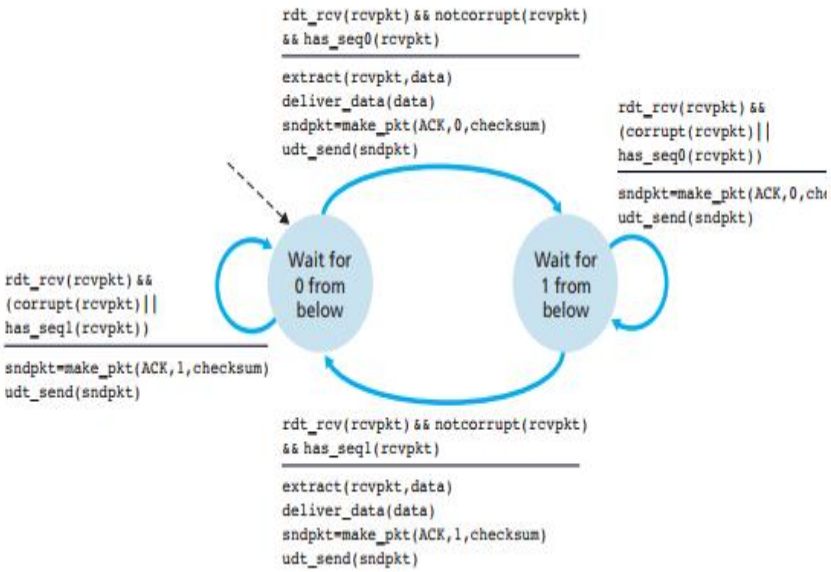
0111111111111111 (which means error)

7.) Explain the working of rdt2.2 and rdt3.0 with the help of FSMs.

rdt 2.2



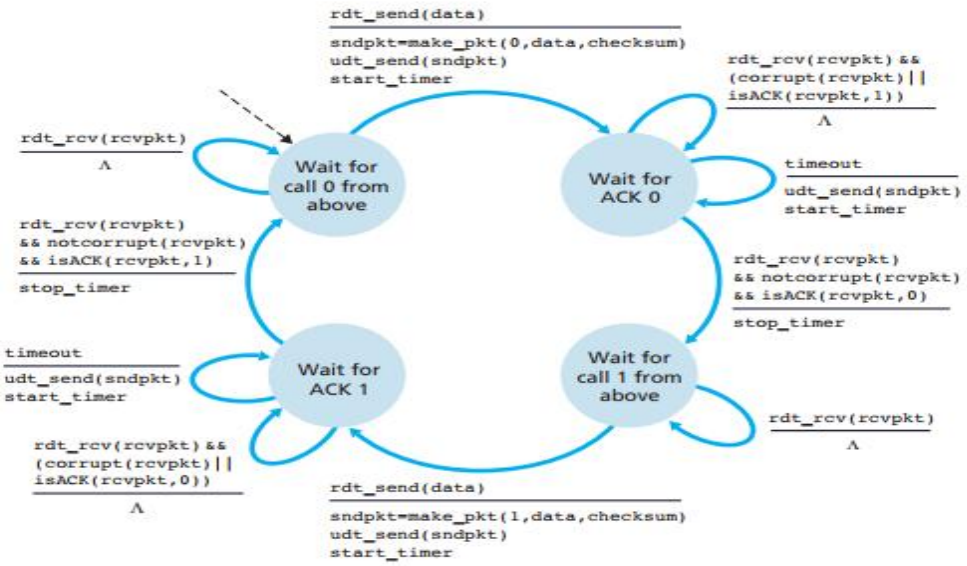
rdt 2.2 sender



rdt 2.2 receiver

Rdt 2.2 send an ACK for the last correctly received data packet, instead of sending NAK. A sender that receives two ACK for the same packet (duplicate ACK) knows that the receiver did not correctly received the packet following the packet that is being acknowledged twice. So the NAK free rdt for channel with bit errors is rdt 2.2. Above are the FSM of rdt 2.2 sender and rdt 2.2 receiver. The causes and actions are mentioned in the fig itself.

rdt 3.0



rdt 3.0 sender

Two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when packet loss occurs. The use of checksumming, sequence numbers, ACK packets, and retransmissions—the techniques already developed in rdt2.2—will allow us to answer the latter concern. Handling the first concern will require adding a new protocol mechanism.

There are many possible approaches toward dealing with packet loss. Here, we'll put the burden of detecting and recovering from lost packets on the sender. Suppose that the sender transmits a data packet and either that packet, or the receiver's ACK of that packet, gets lost. In either case, no reply is forthcoming at the sender from the receiver. If the sender is willing to wait long enough so that it is certain that a packet has been lost, it can simply retransmit the data packet.

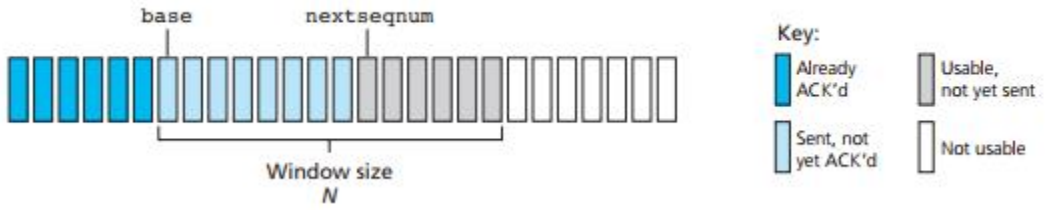
The sender must clearly wait at least as long as a round-trip delay between the sender and receiver plus whatever amount of time is needed to process a packet at the receiver. In many networks, this worst-case maximum delay is very difficult even to estimate, much less know with certainty.

From the sender's viewpoint, retransmission is a panacea. The sender does not know whether a data packet was lost, an ACK was lost, or if the packet or ACK was simply overly delayed. In all cases, the action is the same: retransmit. Implementing a time-based retransmission mechanism requires a countdown timer that can interrupt the sender after a given amount of time has expired. The sender will thus need to be able to (1) start the timer each time a packet (either a first-time packet or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and (3) stop the timer.

8.) **Explain the working of Go-Back-N for pipelined transmission. Use state transition diagram? Why the sliding window size is restricted to 2^k-1 for 'k' bit sequence number in GO-Back-N protocol?**

In a Go-Back-N (GBN) protocol, the sender is allowed to transmit multiple packets

(when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, N, of unacknowledged packets in the pipeline.

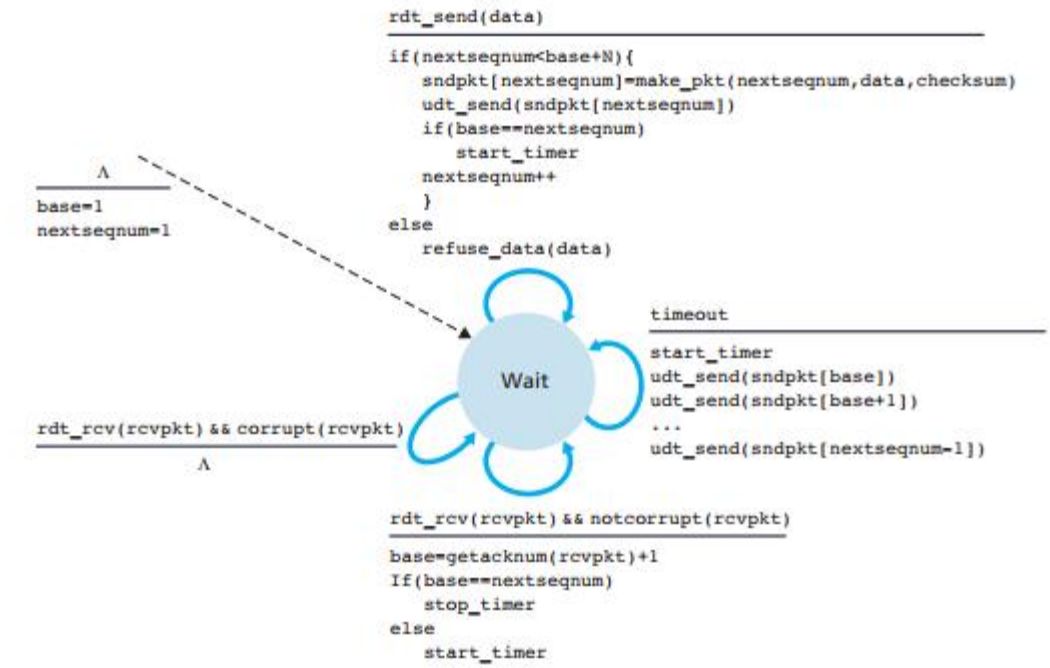


Sender's view of sequence numbers

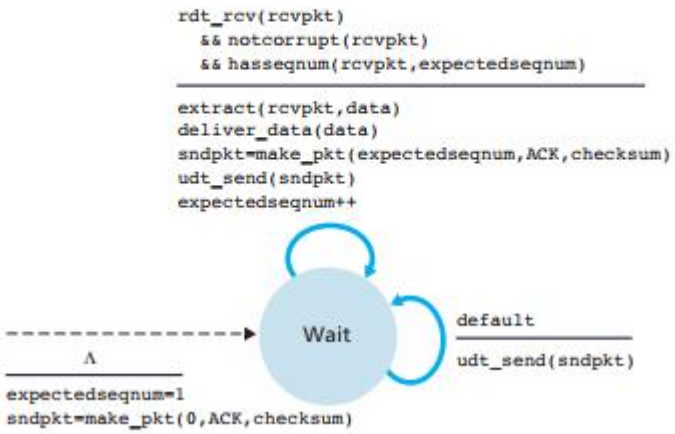
- base = sequence number of the oldest unacknowledged packet
- nextseqnum = the smallest unused sequence number (that is, the sequence number of the next packet to be sent)
- Sequence numbers in the interval [0,base-1] = packets that have already been transmitted and acknowledged.
- The interval [base,nextseqnum-1] = packets that have been sent but not yet acknowledged.
- Sequence numbers in the interval [nextseqnum,base+N-1] = packets that can be sent immediately, should data arrive from the upper layer.
- sequence numbers >= base+N cannot be used until an unacknowledged packet currently in the pipeline (specifically, the packet with sequence number base) has been acknowledged.

The range of permissible sequence numbers for transmitted but not yet acknowledged packets can be viewed as a window of size N over the range of sequence numbers. As the protocol operates, this window slides forward over the sequence number space. For this reason, N is often referred to as the window size and the GBN protocol itself as a sliding-window protocol.

Sequence Number of a packet carried fixed length field in the packet header. The sequence number ranges from 0 to 2^k-1, k= number of bits in the sequence number header of the packet.



GBN sender FSM



GBN receiver FSM

The GBN sender must respond to three types of events:

Invocation from above: When rdt_send() is called from above, the sender first checks to see if the window is full, that is, whether there are N outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer.

Receipt of an ACK: In our GBN protocol, an acknowledgment for a packet with sequence number n will be taken to be a cumulative acknowledgment, indicating that all packets with a sequence number up to and including n have been correctly received at the receiver.

A timeout event : If a timeout occurs, the sender resends all packets that have been previously sent but that have not yet been acknowledged.
The receiver will not keep any out of order packet.

The sliding window size is restricted to 2^k-1 for 'k' bit sequence number in this protocol because it uses the operations on sequence number follows modulo-2^k arithmatic, where the sequence number 2^k-1 followed by sequence number 0.