USN

Internal Assessment Test 2 – November 2025

Sub:	COMPUTER NETWORKS & PROTOCOLS				Sub Code:	BEC702	Branch:	ECE
Date:	02-12-2025	Duration:	90 minutes	Max Marks: 50	Sem/Sec:	7th (A,B,C,D)		OBE

ANSWER ANY 5 FULL QUESTIONS

		MARKS	CO	R
1	**Explain the working of DHCP with suitable figures.**	10	CO3	L
2	**Explain distance vector routing algorithm using Bellman Ford Equations.**	10	CO3	L
3	**Describe connectionless and connection-oriented services provided by the transport layer with suitable figures.**	10	CO4	L
4	**Explain why sender window size must be less than 2^m for Go-Back N protocol with figures.**	10	CO4	L
5	**Explain TCP segment format with suitable figure.**	10	CO4	L
6	**Explain the architecture and format of electronic mail with suitable figures.**	10	CO4	L
7	**Explain DNS recursive and iterative resolutions with suitable figures.**	10	CO4	L

M.S. Aravth Kumar
Course Instructor

Chief Course Instructor

HOD

Q1) Solution:

We have seen that a large organization or an ISP can receive a block of addresses directly from ICANN and a small organization can receive a block of addresses from an ISP.

After a block of addresses are assigned to an organization, the network administration can manually assign addresses to the individual hosts or routers.

However, address assignment in an organization can be done automatically using the Dynamic Host Configuration Protocol (DHCP).

DHCP is an application-layer program, using the client-server paradigm, that actually helps TCP/IP at the network layer.

DHCP has found such widespread use in the Internet that it is often called a plug and-play protocol.

It can be used in many situations.

A network manager can configure DHCP to assign permanent IP addresses to the host and routers.

DHCP can also be configured to provide temporary, on demand, IP addresses to hosts.

The second capability can provide a temporary IP address to a traveller to connect her laptop to the Internet while she is staying in the hotel.

It also allows an ISP with 1000 granted addresses to provide services to 4000 households, assuming not more than one-forth of customers use the Internet at the same time.

In addition to its IP address, a computer also needs to know the network prefix (or address mask).

Most computers also need two other pieces of information, such as the address of a default router to be able to communicate with other networks and the address of a name server to be able to use names instead of addresses, as we will see in Chapter 26.

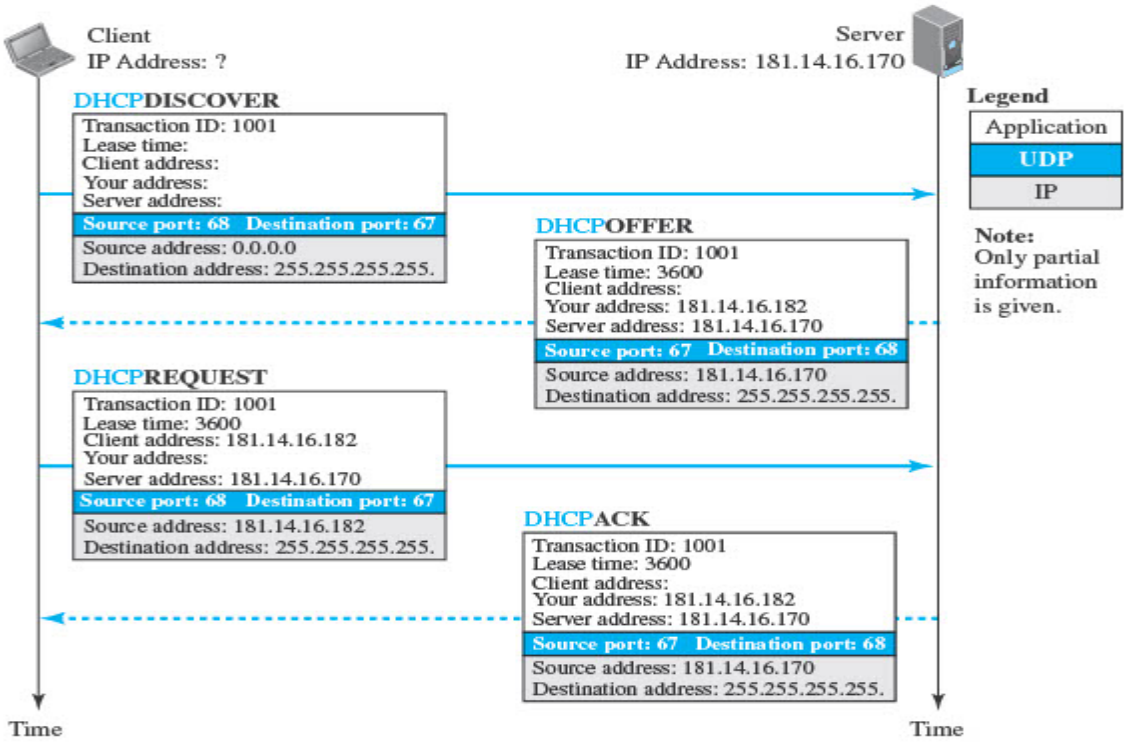
In other words, four pieces of information are normally needed: the computer address, the prefix, the address of a router, and the IP address of a name server.

DHCP can be used to provide these pieces of information to the host.

DHCP Operation

Figure 18.27 shows a simple scenario.

Figure 18.27 *Operation of DHCP*



1. The joining host creates a DHCPDISCOVER message in which only the transaction-ID field is set to a random number.

No other field can be set because the host has no knowledge with which to do so.

This message is encapsulated in a UDP user datagram with the source port set to 68 and the destination port set to 67.

We will discuss the reason for using two well-known port numbers later.

The user datagram is encapsulated in an IP datagram with the source address set to 0.0.0.0 (“this host”) and the destination address set to 255.255.255.255 (broadcast address).

The reason is that the joining host knows neither its own address nor the server address.

2. The DHCP server or servers (if more than one) responds with a DHCPOFFER message in which the your address field defines the offered IP address for the joining host and the server address field includes the IP address of the server.

The message also includes the lease time for which the host can keep the IP address.

This message is encapsulated in a user datagram with the same port numbers, but in the reverse order.

The user datagram in turn is encapsulated in a datagram with the server address as the source IP address, but the destination address is a broadcast address, in which the server allows other DHCP servers to receive the offer and give a better offer if they can.

3. The joining host receives one or more offers and selects the best of them.

The joining host then sends a DHCPREQUEST message to the server that has given the best offer.

The fields with known value are set.

The message is encapsulated in a user datagram with port numbers as the first message.

The user datagram is encapsulated in an IP datagram with the source address set to the new client address, but the destination address still is set to the broadcast address to let the other servers know that their offer was not accepted.

4. Finally, the selected server responds with a DHCPACK message to the client if the offered IP address is valid.

If the server cannot keep its offer (for example, if the address is offered to another host in between), the server sends a DHCPNACK message and the client needs to repeat the process.

This message is also broadcast to let other servers know that the request is accepted or rejected.

Q2) Solution :

The distance-vector (DV) routing uses the goal we discussed in the introduction, to find the best route.

In distance-vector routing, the first thing each node creates is its own least-cost tree with the rudimentary information it has about its immediate neighbors.

The incomplete trees are exchanged between immediate neighbors to make the trees more and more complete and to represent the whole internet.

We can say that in distance-vector routing, a router continuously tells all of its neighbors what it knows about the whole internet (although the knowledge can be incomplete).

Before we show how incomplete least-cost trees can be combined to make complete ones, we need to discuss two important topics: the Bellman-Ford equation and the concept of distance vectors, which we cover next.

Bellman-Ford Equation

The heart of distance-vector routing is the famous Bellman-Ford equation.

This equation is used to find the least cost (shortest distance) between a source node, x, and a destination node, y, through some intermediary nodes (a, b, c, . . .) when the costs between the source and the intermediary nodes and the least costs between the intermediary nodes and the destination are given.

The following shows the general case in which Dij is the shortest distance and cij is the cost between nodes i and j.

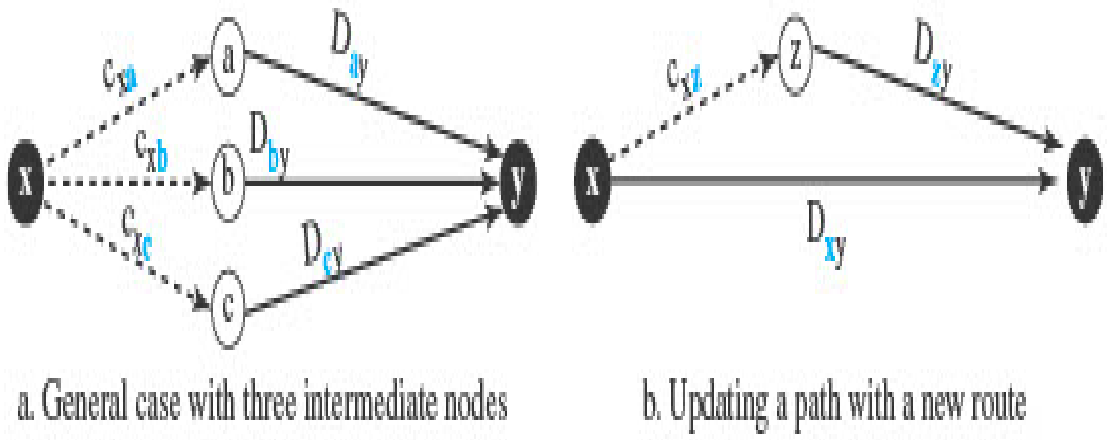
$$D_{xy} = \min\left\{(c_{xa} + D_{ay}), (c_{xb} + D_{by}), (c_{xc} + D_{cy}), \ldots\right\}$$

In distance-vector routing, normally we want to update an existing least cost with a least cost through an intermediary node, such as z, if the latter is shorter. In this case, the equation becomes simpler, as shown below:

$$D_{xy} = \min\left\{D_{xy}, (c_{xz} + D_{zy})\right\}$$

Figure 20.3 shows the idea graphically for both cases.

Figure 20.3 *Graphical idea behind Bellman-Ford equation*



We can say that the Bellman-Ford equation enables us to build a new least-cost path from previously established least-cost paths.

In Figure 20.3, we can think of (a→y), (b→y), and (c→y) as previously established least-cost paths and (x→y) as the new least-cost path.

We can even think of this equation as the builder of a new least-cost tree from previously established least-cost trees if we use the equation repeatedly.

In other words, the use of this equation in distance-vector routing is a witness that this method also uses least-cost trees, but this use may be in the background.

We will shortly show how we use the Bellman-Ford equation and the concept of distance vectors to build least-cost paths for each node in distance-vector routing, but first we need to discuss the concept of a distance vector.

The concept of a distance vector is the rationale for the name distance-vector routing.

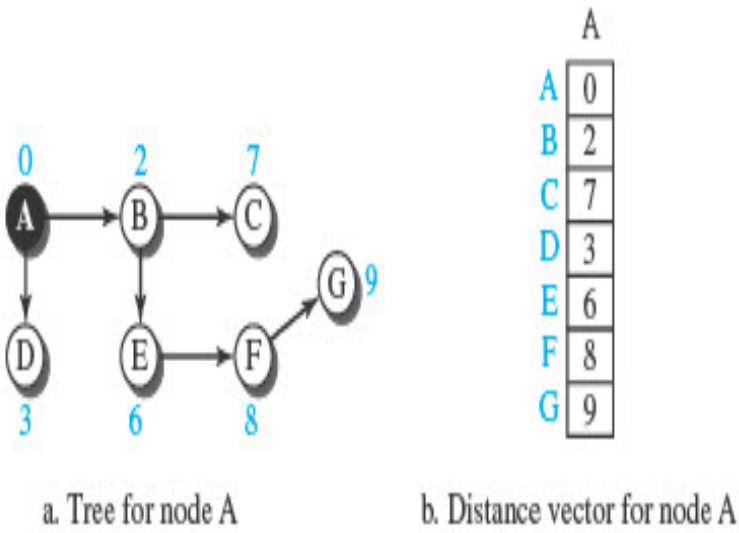
A least-cost tree is a combination of least-cost paths from the root of the tree to all destinations.

These paths are graphically glued together to form the tree.

Distance-vector routing unglues these paths and creates a distance vector, a one-dimensional array to represent the tree.

Figure 20.4 shows the tree for node A in the internet in Figure 20.1 and the corresponding distance vector.

Figure 20.4 *The distance vector corresponding to a tree*



a. Tree for node A b. Distance vector for node A

Note that the name of the distance vector defines the root, the indexes define the destinations, and the value of each cell defines the least cost from the root to the destination.

A distance vector does not give the path to the destinations as the least-cost tree does; it gives only the least costs to the destinations.

Later we show how we can change a distance vector to a forwarding table, but we first need to find all distance vectors for an internet.

We know that a distance vector can represent least-cost paths in a least-cost tree, but the question is how each node in an internet originally creates the corresponding vector.

Each node in an internet, when it is booted, creates a very rudimentary distance vector with the minimum information the node can obtain from its neighborhood.

The node sends some greeting messages out of its interfaces and discovers the identity of the immediate neighbors and the distance between itself and each neighbor.

It then makes a simple distance vector by inserting the discovered distances in the corresponding cells and leaves the value of other cells as infinity.

Do these distance vectors represent least-cost paths?

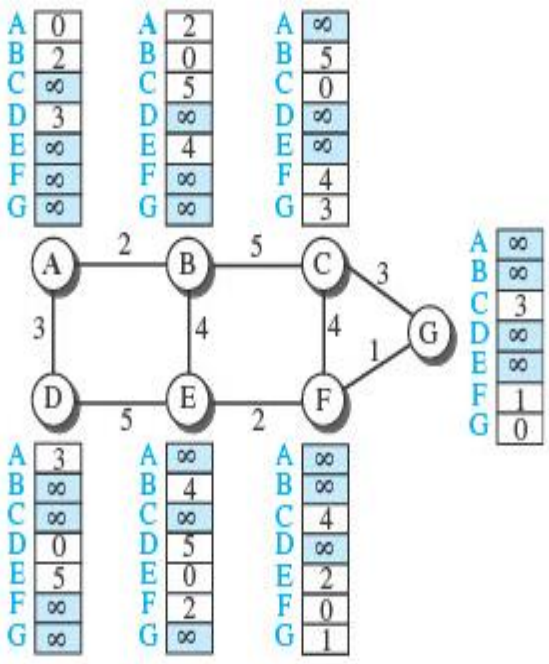
They do, considering the limited information a node has.

When we know only one distance between two nodes, it is the least cost.

Figure 20.5 shows all distance vectors for our internet.

However, we need to mention that these vectors are made asynchronously, when the corresponding node has been booted; the existence of all of them in a figure does not mean synchronous creation of them.

Figure 20.5 *The first distance vector for an internet*



These rudimentary vectors cannot help the internet to effectively forward a packet.

For example, node A thinks that it is not connected to node G because the corresponding cell shows the least cost of infinity.

To improve these vectors, the nodes in the internet need to help each other by exchanging information.

After each node has created its vector, it sends a copy of the vector to all its immediate neighbors.

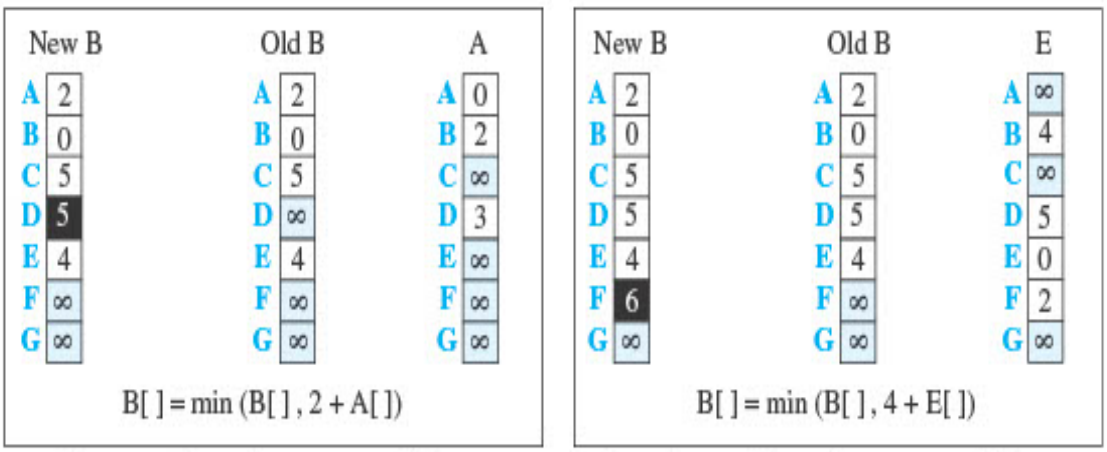
After a node receives a distance vector from a neighbor, it updates its distance vector using the Bellman-Ford equation (second case).

However, we need to understand that we need to update, not only one least cost, but N of them in which N is the number of the nodes in the internet.

If we are using a program, we can do this using a loop; if we are showing the concept on paper, we can show the whole vector instead of the N separate equations.

We show the whole vector instead of seven equations for each update in Figure 20.6.

Figure 20.6 *Updating distance vectors*



a. First event: B receives a copy of A's vector.

b. Second event: B receives a copy of E's vector.

Note:
X[ ]: the whole vector

The figure shows two asynchronous events, happening one after another with some time in between.

In the first event, node A has sent its vector to node B.

Node B updates its vector using the cost cBA = 2.

In the second event, node E has sent its vector to node B.

Node B updates its vector using the cost cEA = 4.

After the first event, node B has one improvement in its vector: its least cost to node D has changed from infinity to 5 (via node A).

After the second event, node B has one more improvement in its vector; its least cost to node F has changed from infinity to 6 (via node E).

We hope that we have convinced the reader that exchanging vectors eventually stabilizes the system and allows all nodes to find the ultimate least cost between themselves and any other node.

We need to remember that after updating a node, it immediately sends its updated vector to all neighbors.

Even if its neighbors have received the previous vector, the updated one may help more.

Distance-Vector Routing Algorithm

Now we can give a simplified pseudocode for the distance-vector routing algorithm, as shown in Table 20.1.

The algorithm is run by its node independently and asynchronously.

Table 20.1 *Distance-Vector Routing Algorithm for a Node*

```
1  Distance_Vector_Routing ( )
2  {
3      // Initialize (create initial vectors for the node)
4      D[myself] = 0
```

Table 20.1 *Distance-Vector Routing Algorithm for a Node (continued)*

```
5     for (y = 1 to N)
6     {
7         if (y is a neighbor)
8             D[y] = c[myself][y]
9         else
10            D[y] = ∞
11    }
12    send vector {D[1], D[2], ..., D[N]} to all neighbors
13    // Update (improve the vector with the vector received from a neighbor)
14    repeat (forever)
15    {
16        wait (for a vector Dw from a neighbor w or any change in the link)
17        for (y = 1 to N)
18        {
19            D[y] = min [D[y], (c[myself][w] + Dw[y])]      // Bellman-Ford equation
20        }
21        if (any change in the vector)
22            send vector {D[1], D[2], ..., D[N]} to all neighbors
23    }
24  } // End of Distance Vector
```

Lines 4 to 11 initialize the vector for the node.

Lines 14 to 23 show how the vector can be updated after receiving a vector from the immediate neighbor.

The for loop in lines 17 to 20 allows all entries (cells) in the vector to be updated after receiving a new vector.

Note that the node sends its vector in line 12, after being initialized, and in line 22, after it is updated.

Q3) Solution :

A transport-layer protocol, like a network-layer protocol, can provide two types of services: connectionless and connection-oriented.

The nature of these services at the transport layer, however, is different from the ones at the network layer.

At the network layer, a connectionless service may mean different paths for different datagrams belonging to the same message.

At the transport layer, we are not concerned about the physical paths of packets (we assume a logical connection between two transport layers).

Connectionless service at the transport layer means independency between packets; connection-oriented means dependency.

Let us elaborate on these two services.

Connectionless Service

In a connectionless service, the source process (application program) needs to divide its message into chunks of data of the size acceptable by the transport layer and deliver them to the transport layer one by one.

The transport layer treats each chunk as a single unit without any relation between the chunks.

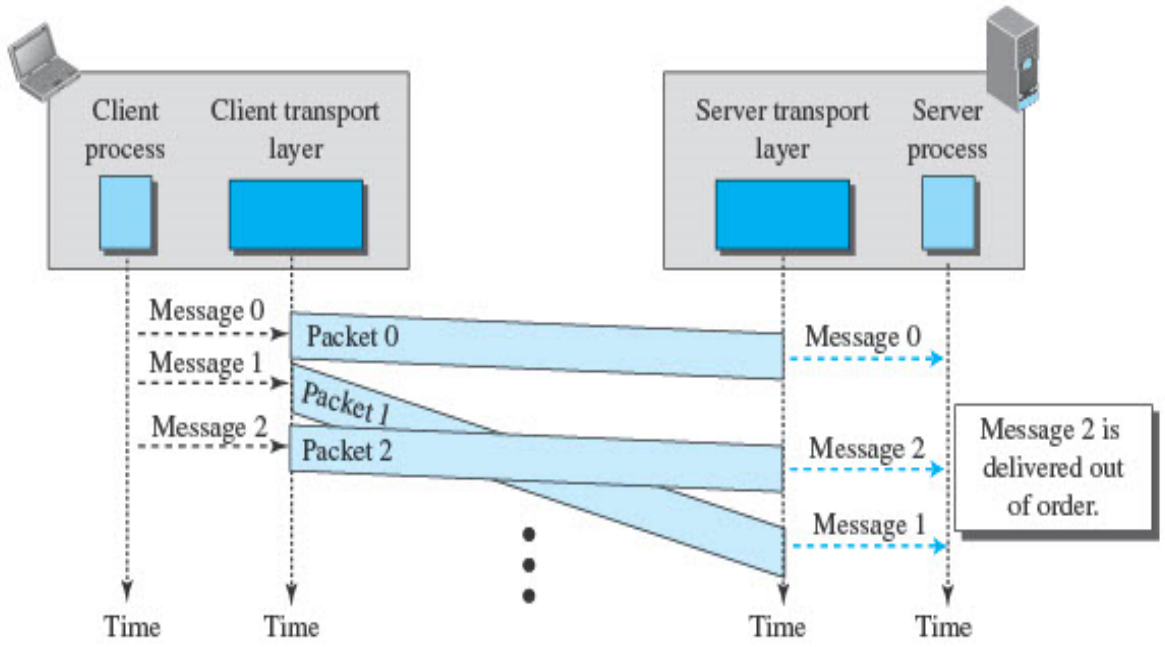
When a chunk arrives from the application layer, the transport layer encapsulates it in a packet and sends it.

To show the independency of packets, assume that a client process has three chunks of messages to send to a server process.

The chunks are handed over to the connectionless transport protocol in order.

However, since there is no dependency between the packets at the transport layer, the packets may arrive out of order at the destination and will be delivered out of order to the server process (Figure 23.14).

Figure 23.14 *Connectionless service*



In Figure 23.14, we have shown the movement of packets using a time line, but we have assumed that the delivery of the process to the transport layer and vice versa are instantaneous.

The figure shows that at the client site, the three chunks of messages are delivered to the client transport layer in order (0, 1, and 2).

Because of the extra delay in transportation of the second packet, the delivery of messages at the server is not in order (0, 2, 1).

If these three chunks of data belong to the same message, the server process may have received a strange message.

The situation would be worse if one of the packets were lost.

Since there is no numbering on the packets, the receiving transport layer has no idea that one of the messages has been lost.

It just delivers two chunks of data to the server process.

The above two problems arise from the fact that the two transport layers do not coordinate with each other.

The receiving transport layer does not know when the first packet will come nor when all of the packets have arrived.

We can say that no flow control, error control, or congestion control can be effectively implemented in a connectionless service.

Connection-Oriented Service

In a connection-oriented service, the client and the server first need to establish a logical connection between themselves.

The data exchange can only happen after the connection establishment.

After data exchange, the connection needs to be torn down (Figure 23.15).

As we mentioned before, the connection-oriented service at the transport layer is different from the same service at the network layer.

In the network layer, connection-oriented service means a coordination between the two end hosts and all the routers in between.

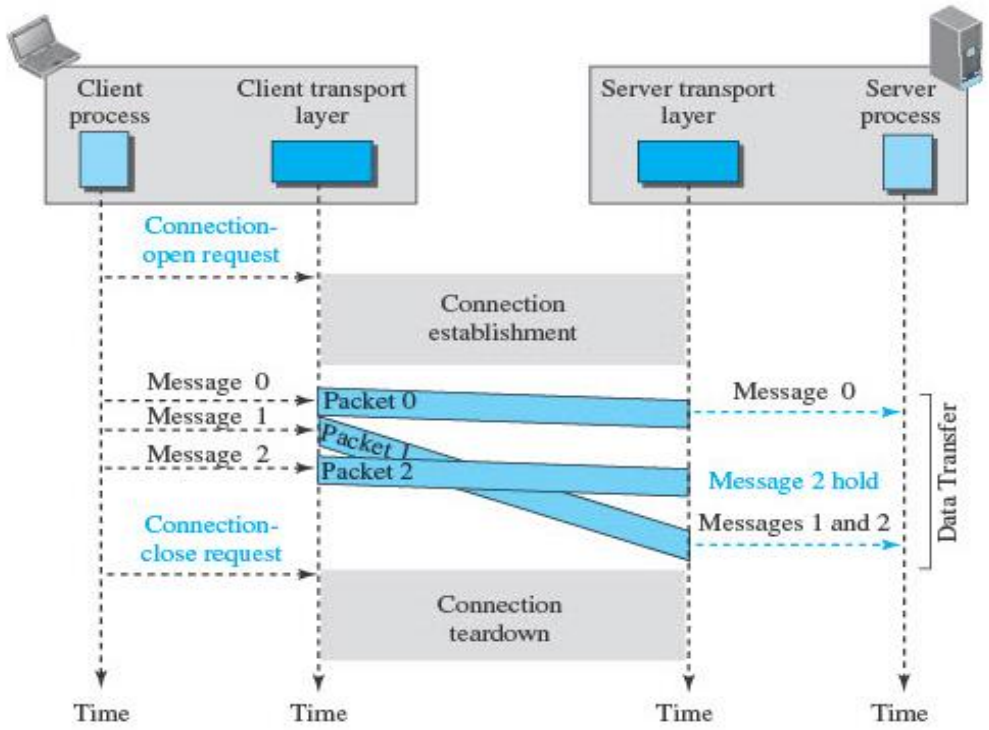
At the transport layer, connection-oriented service involves only the two hosts; the service is end to end.

This means that we should be able to make a connection-oriented protocol at the transport layer over either a connectionless or connection-oriented protocol at the network layer.

Figure 23.15 shows the connection-establishment, data-transfer, and tear-down phases in a connection-oriented service at the transport layer.

We can implement flow control, error control, and congestion control in a connection-oriented protocol.

Figure 23.15 *Connection-oriented service*



Q4) Solution :

The send window is an imaginary box covering the sequence numbers of the data packets that can be in transit or can be sent.

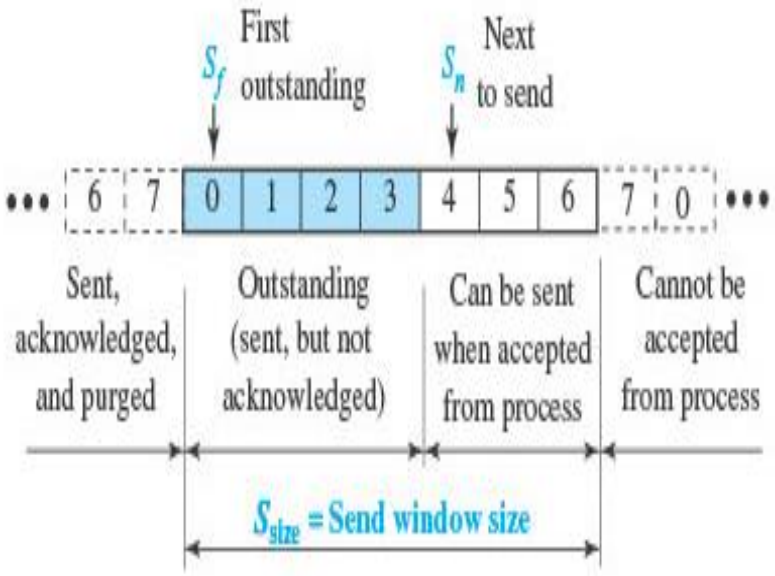
In each window position, some of these sequence numbers define the packets that have been sent; others define those that can be sent.

The maximum size of the window is 2m − 1, for reasons that we discuss later.

In this chapter, we let the size be fixed and set to the maximum value, but we will see later that some protocols may have a variable window size.

Figure 23.24 shows a sliding window of size 7 (m = 3) for the Go-Back-N protocol.

Figure 23.24 *Send window for Go-Back-N*



The send window at any time divides the possible sequence numbers into four regions.

The first region, left of the window, defines the sequence numbers belonging to packets that are already acknowledged.

The sender does not worry about these packets and keeps no copies of them.

The second region, colored, defines the range of sequence numbers belonging to the packets that have been sent, but have an unknown status.

The sender needs to wait to find out if these packets have been received or were lost.

We call these outstanding packets.

The third range, white in the figure, defines the range of sequence numbers for packets that can be sent; however, the corresponding data have not yet been received from the application layer.

Finally, the fourth region, right of the window, defines sequence numbers that cannot be used until the window slides.

The window itself is an abstraction; three variables define its size and location at any time.

We call these variables Sf (send window, the first outstanding packet), Sn (send window, the next packet to be sent), and Ssize (send window, size).

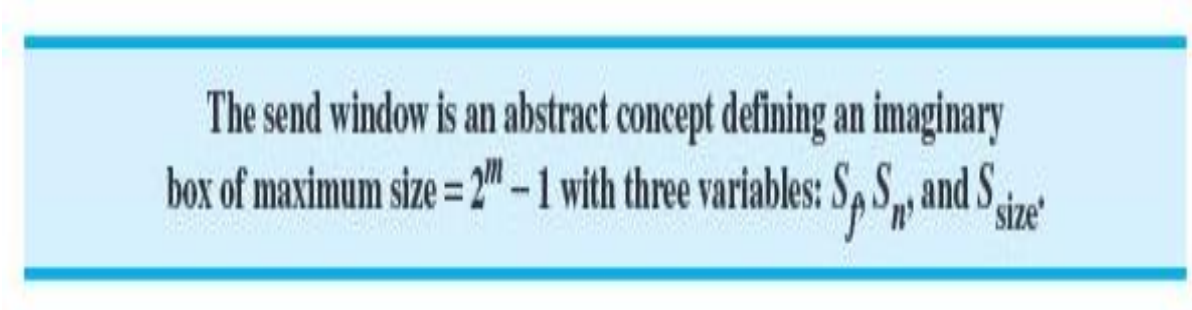
The variable Sf defines the sequence number of the first (oldest) outstanding packet.

The variable Sn holds the sequence number that will be assigned to the next packet to be sent.

Finally, the variable Ssize defines the size of the window, which is fixed in our protocol.

Figure 23.25 shows how a send window can slide one or more slots to the right when an acknowledgment arrives from the other end.

In the figure, an acknowledgment with ackNo = 6 has arrived. This means that the receiver is waiting for packets with sequence number 6.



We can now show why the size of the send window must be less than 2m. As an example, we choose m = 2, which means the size of the window can be 2m − 1, or 3.

Figure 23.28 compares a window size of 3 against a window size of 4. If the size of the window is 3 (less than 2m) and all three acknowledgments are lost, the only timer expires and all three packets are resent.

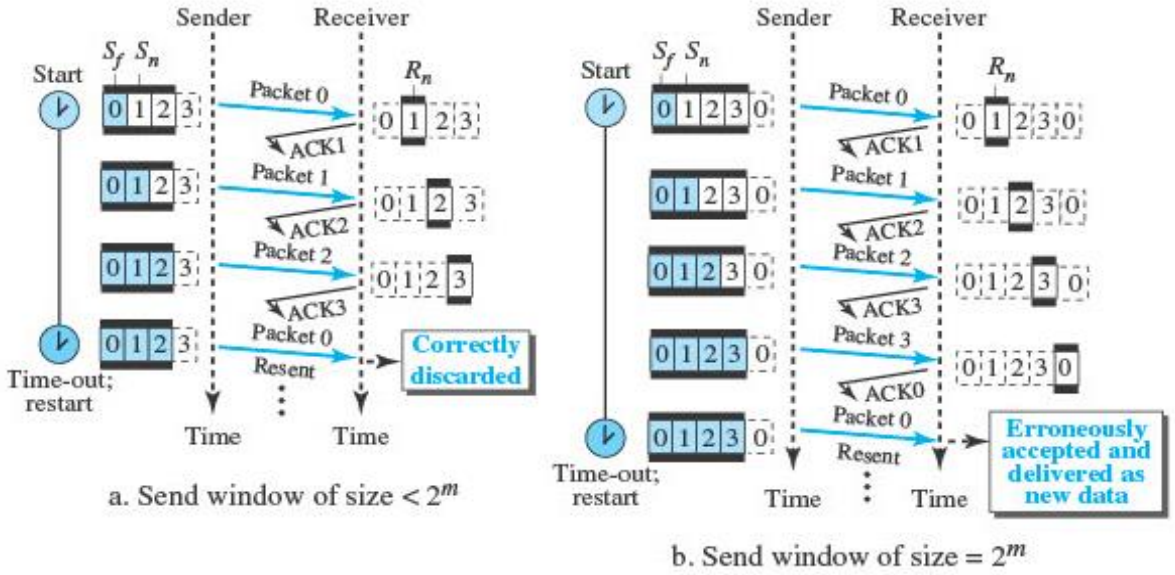
The receiver is now expecting packet 3, not packet 0, so the duplicate packet is correctly discarded.

On the other hand, if the size of the window is 4 (equal to 22) and all acknowledgments are lost, the sender will send a duplicate of packet 0.

However, this time the window of the receiver expects to receive packet 0 (in the next cycle), so it accepts packet 0, not as a duplicate, but as the first packet in the next cycle.

This is an error. This shows that the size of the send window must be less than 2^m.

Figure 23.28 *Send window size for Go-Back-N*



In the Go-Back-*N* protocol, the size of the send window must be less than 2^m; the size of the receive window is always 1.

Q5) Solution :

Before discussing TCP in more detail, let us discuss the TCP packets themselves.

A packet in TCP is called a segment.

The format of a segment is shown in Figure 24.7.

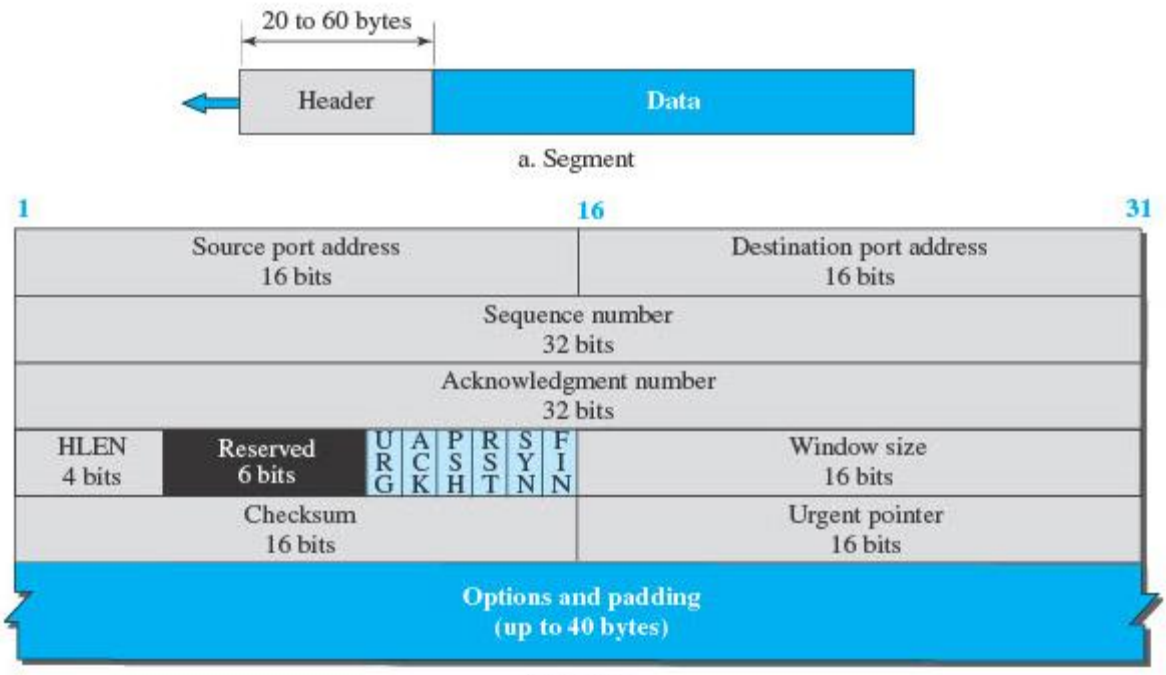
The segment consists of a header of 20 to 60 bytes, followed by data from the application program.

The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

We will discuss some of the header fields in this section.

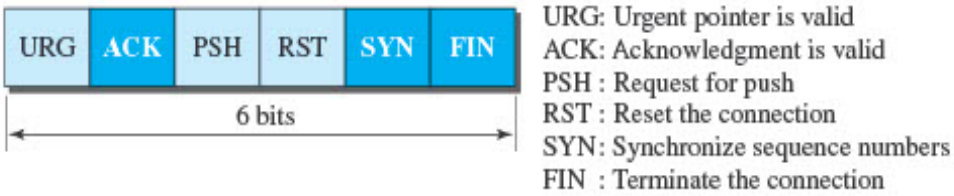
The meaning and purpose of these will become clearer as we proceed through the section.

Figure 24.7 *TCP segment format*



- ***Source port address.*** This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.
- ***Destination port address.*** This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.
- ***Sequence number.*** This 32-bit field defines the number assigned to the first byte of data contained in this segment. As we said before, TCP is a stream transport protocol. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence is the first byte in the segment. During connection establishment (discussed later) each party uses a random number generator to create an **initial sequence number** (ISN), which is usually different in each direction.
- ***Acknowledgment number.*** This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number x from the other party, it returns $x + 1$ as the acknowledgment number. Acknowledgment and data can be piggybacked together.
- ***Header length.*** This 4-bit field indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field is always between 5 ($5 \times 4 = 20$) and 15 ($15 \times 4 = 60$).
- ***Control.*** This field defines 6 different control bits or flags, as shown in Figure 24.8. One or more of these bits can be set at a time. These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP. A brief description of each bit is shown in the figure. We will discuss them further when we study the detailed operation of TCP later in the chapter.

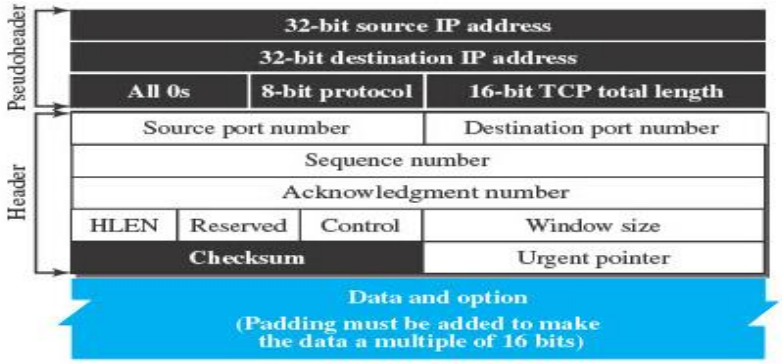
Figure 24.8 *Control field*



- ***Window size.*** This field defines the window size of the sending TCP in bytes. Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes. This value is normally referred to as the receiving window (*rwnd*) and is determined by the receiver. The sender must obey the dictation of the receiver in this case.
- ***Checksum.*** This 16-bit field contains the checksum. The calculation of the checksum for TCP follows the same procedure as the one described for UDP. However, the use of the checksum in the UDP datagram is optional, whereas the use of the checksum for TCP is mandatory. The same pseudoheader, serving the same

purpose, is added to the segment. For the TCP pseudoheader, the value for the protocol field is 6. See Figure 24.9.

Figure 24.9 *Pseudoheader added to the TCP datagram*



The use of the checksum in TCP is mandatory.

- ***Urgent pointer.*** This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data. It defines a value that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment. This will be discussed later in this chapter.
- ***Options.*** There can be up to 40 bytes of optional information in the TCP header. We will discuss some of the options used in the TCP header later in the section.

Q6) Solution :

Electronic mail (or e-mail) allows users to exchange messages.

The nature of this application, however, is different from other applications discussed so far.

In an application such as HTTP or FTP, the server program is running all the time, waiting for a request from a client.

When the request arrives, the server provides the service.

There is a request and there is a response.

In the case of electronic mail, the situation is different.

First, e-mail is considered a one-way transaction.

When Alice sends an email to Bob, she may expect a response, but this is not a mandate.

Bob may or may not respond.

If he does respond, it is another one-way transaction.

Second, it is neither feasible nor logical for Bob to run a server program and wait until someone sends an e-mail to him.

Bob may turn off his computer when he is not using it.

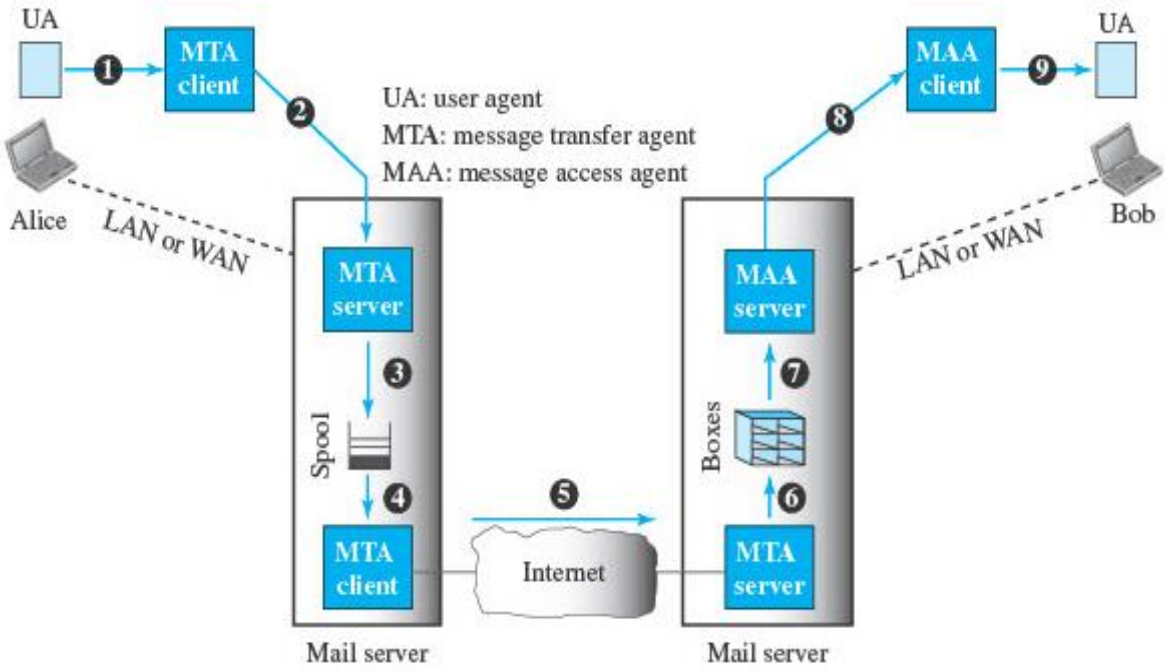
This means that the idea of client/server programming should be implemented in another way: using some intermediate computers (servers).

The users run only client programs when they want and the intermediate servers apply the client/server paradigm, as we discuss in the next section.

To explain the architecture of e-mail, we give a common scenario, as shown in Figure 26.12.

Another possibility is the case in which Alice or Bob is directly connected to the corresponding mail server, in which LAN or WAN connection is not required, but this variation in the scenario does not affect our discussion.

Figure 26.12 *Common scenario*



In the common scenario, the sender and the receiver of the e-mail, Alice and Bob respectively, are connected via a LAN or a WAN to two mail servers.

The administrator has created one mailbox for each user where the received messages are stored.

A mailbox is part of a server hard drive, a special file with permission restrictions.

Only the owner of the mailbox has access to it.

The administrator has also created a queue (spool) to store messages waiting to be sent.

A simple e-mail from Alice to Bob takes nine different steps, as shown in the figure.

Alice and Bob use three different agents: a user agent (UA), a message transfer agent (MTA), and a message access agent (MAA).

When Alice needs to send a message to Bob, she runs a UA program to prepare the message and send it to her mail server.

The mail server at her site uses a queue (spool) to store messages waiting to be sent.

The message, however, needs to be sent through the Internet from Alice's site to Bob's site using an MTA.

Here two message transfer agents are needed: one client and one server.

Like most client-server programs on the Internet, the server needs to run all the time because it does not know when a client will ask for a connection.

The client, on the other hand, can be triggered by the system when there is a message in the queue to be sent.

The user agent at the Bob site allows Bob to read the received message.

Bob later uses an MAA client to retrieve the message from an MAA server running on the second server.

There are two important points we need to emphasize here.

First, Bob cannot bypass the mail server and use the MTA server directly.

To use the MTA server directly, Bob would need to run the MTA server all the time because he does not know when a message will arrive.

This implies that Bob must keep his computer on all the time if he is connected to his system through a LAN.

If he is connected through a WAN, he must keep the connection up all the time.

Neither of these situations is feasible today.

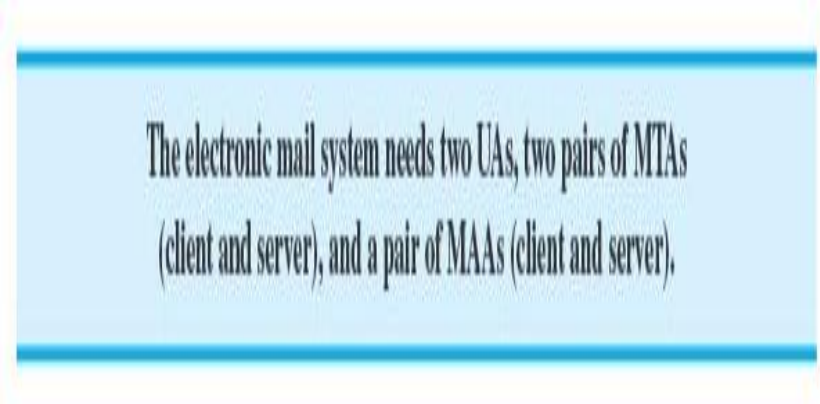
Second, note that Bob needs another pair of client-server programs: message access programs.

This is because an MTA client-server program is a push program: the client pushes the message to the server.

Bob needs a pull program.

The client needs to pull the message from the server.

We discuss more about MAAs shortly.



The first component of an electronic mail system is the user agent (UA).

It provides service to the user to make the process of sending and receiving a message easier.

A user agent is a software package (program) that composes, reads, replies to, and forwards messages.

It also handles local mailboxes on the user computers.

There are two types of user agents: command-driven and GUI-based.

Command-driven user agents belong to the early days of electronic mail.

They are still present as the underlying user agents.

A command-driven user agent normally accepts a one-character command from the keyboard to perform its task.

For example, a user can type the character r, at the command prompt, to reply to the sender of the message, or type the character R to reply to the sender and all recipients.

Some examples of command-driven user agents are mail, pine, and elm.

Modern user agents are GUI-based.

They contain graphical user interface (GUI) components that allow the user to interact with the software by using both the keyboard and the mouse.

They have graphical components such as icons, menu bars, and windows that make the services easy to access.

Some examples of GUI-based user agents are Eudora and Outlook.

To send mail, the user, through the UA, creates mail that looks very similar to postal mail.

It has an envelope and a message (see Figure 26.13).

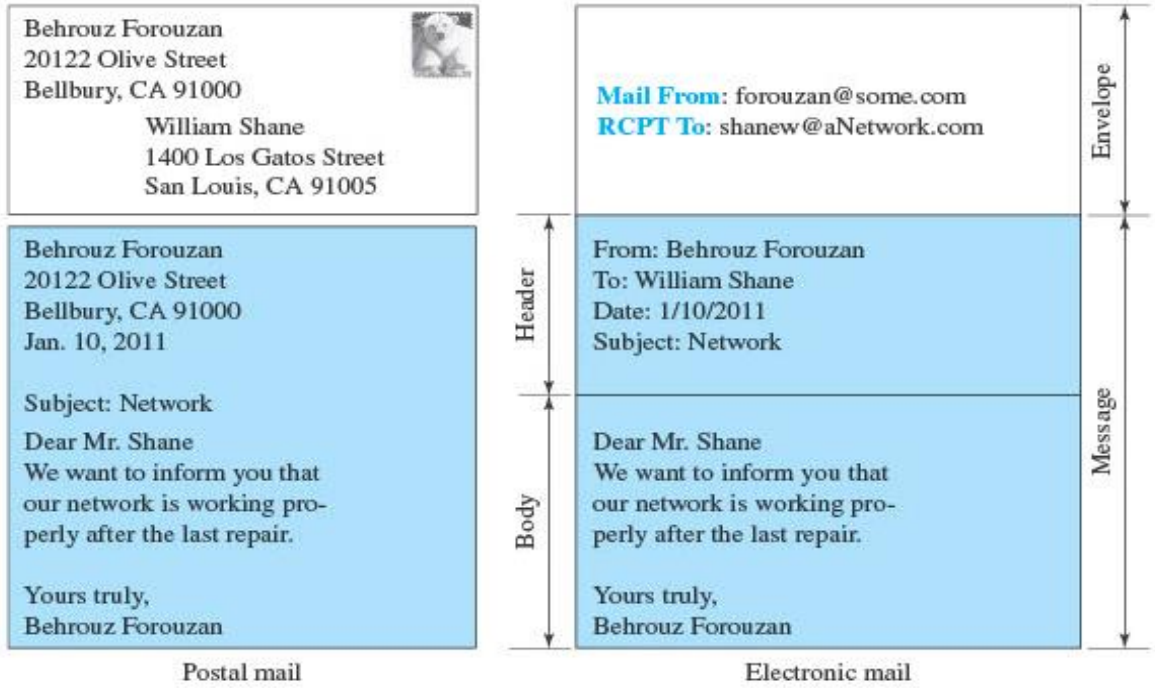
The envelope usually contains the sender address, the receiver address, and other information.

The message contains the header and the body.

The header of the message defines the sender, the receiver, the subject of the message, and some other information.

The body of the message contains the actual information to be read by the recipient.

Figure 26.13 *Format of an e-mail*



The user agent is triggered by the user (or a timer).

If a user has mail, the UA informs the user with a notice.

If the user is ready to read the mail, a list is displayed in which each line contains a summary of the information about a particular message in the mailbox.

The summary usually includes the sender mail address, the subject, and the time the mail was sent or received.

The user can select any of the messages and display its contents on the screen.

Q7) Solution:

The last client-server application program we discuss has been designed to help other application programs.

To identify an entity, TCP/IP protocols use the IP address, which uniquely identifies the connection of a host to the Internet.

However, people prefer to use names instead of numeric addresses.

Therefore, the Internet needs to have a directory system that can map a name to an address.

This is analogous to the telephone network.

A telephone network is designed to use telephone numbers, not names.

People can either keep a private file to map a name to the corresponding telephone number or can call the telephone directory to do so.

We discuss how this directory system in the Internet can map names to IP addresses.

Since the Internet is so huge today, a central directory system cannot hold all the mapping.

In addition, if the central computer fails, the whole communication network will collapse.

A better solution is to distribute the information among many computers in the world.

In this method, the host that needs mapping can contact the closest computer holding the needed information.

This method is used by the Domain Name System (DNS).

We first discuss the concepts and ideas behind the DNS. We then describe the DNS protocol itself.

Mapping a name to an address is called name-address resolution.

DNS is designed as a client-server application.

A host that needs to map an address to a name or a name to an address calls a DNS client called a resolver.

The resolver accesses the closest DNS server with a mapping request.

If the server has the information, it satisfies the resolver; otherwise, it either refers the resolver to other servers or asks other servers to provide the information.

After the resolver receives the mapping, it interprets the response to see if it is a real resolution or an error, and finally delivers the result to the process that requested it.

A resolution can be either recursive or iterative.

Recursive Resolution

Figure 26.36 shows a simple example of a recursive resolution.

We assume that an application program running on a host named some.anet.com needs to find the IP address of another host named engineering.mcgraw-hill.com to send a message to.

The source host is connected to the Anet ISP; the destination host is connected to the McGraw-Hill network.

The application program on the source host calls the DNS resolver (client) to find the IP address of the destination host.

The resolver, which does not know this address, sends the query to the local DNS server (for example, dns.anet.com) running at the Anet ISP site (event 1).

We assume that this server does not know the IP address of the destination host either.

It sends the query to a root DNS server, whose IP address is supposed to be known to this local DNS server (event 2).

Root servers do not normally keep the mapping between names and IP addresses, but a root server should at least know about one server at each top level domain (in this case, a server responsible for com domain).

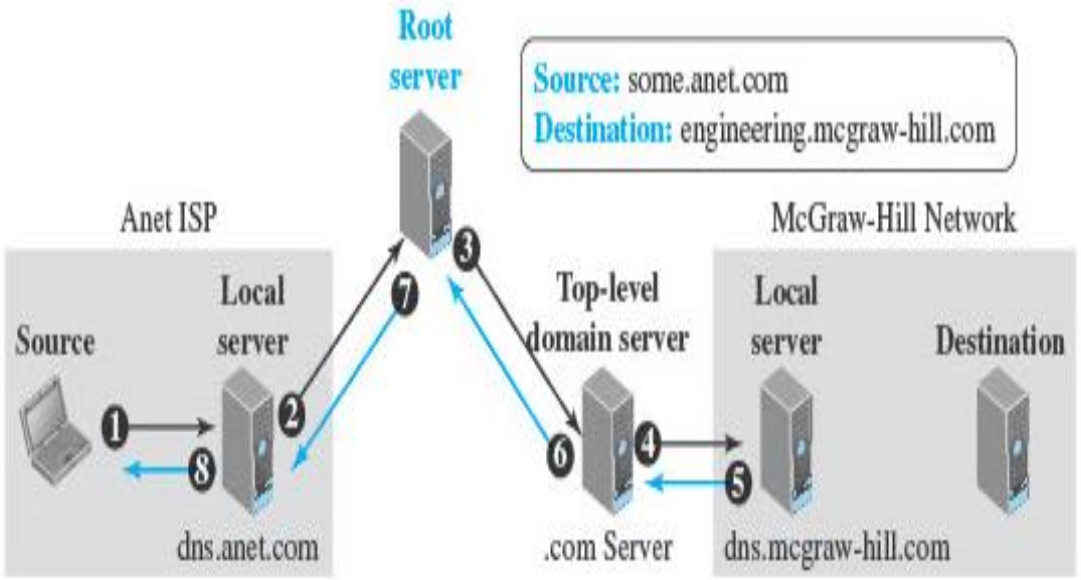
The query is sent to this top-level-domain server (event 3).

We assume that this server does not know the name-address mapping of this specific destination, but it knows the IP address of the local DNS server in the McGraw-Hill company (for example, dns.mcgraw-hill.com).

The query is sent to this server (event 4), which knows the IP address of the destination host.

The IP address is now sent back to the top-level DNS server (event 5), then back to the root server (event 6), then back to the ISP DNS server, which may cache it for the future queries (event 7), and finally back to the source host (event 8).

Figure 26.36 *Recursive resolution*



Iterative Resolution

In iterative resolution, each server that does not know the mapping sends the IP address of the next server back to the one that requested it.

Figure 26.37 shows the flow of information in an iterative resolution in the same scenario as the one depicted in Figure 26.36.

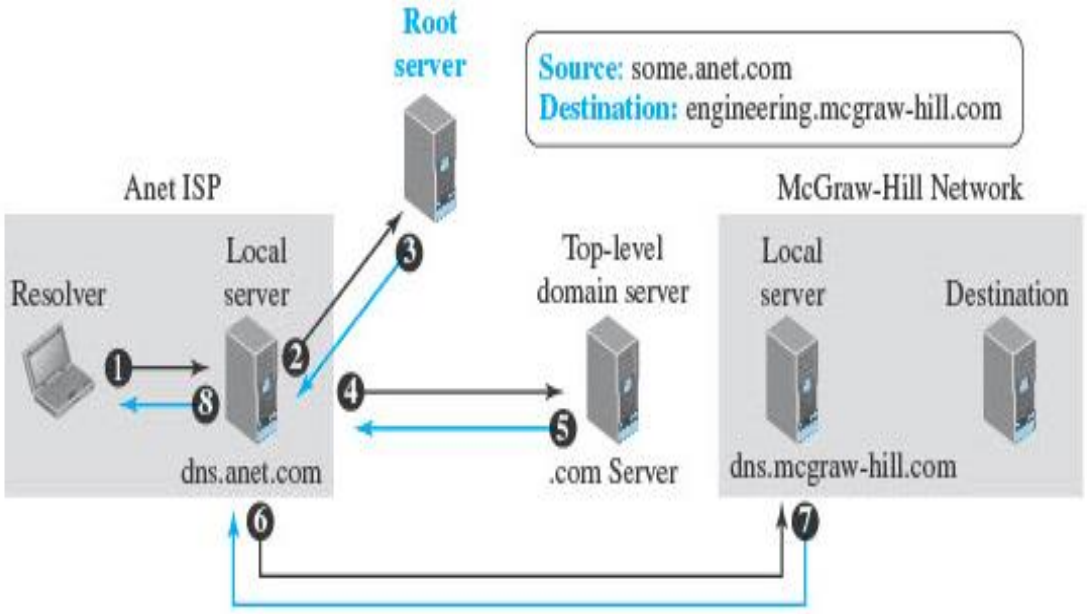
Normally the iterative resolution takes place between two local servers; the original resolver gets the final answer from the local server.

Note that the messages shown by events 2, 4, and 6 contain the same query.

However, the message shown by event 3 contains the IP address of the top-level domain server, the message shown by event 5 contains the IP address of the McGraw-Hill local DNS server, and the message shown by event 7 contains the IP address of the destination.

When the Anet local DNS server receives the IP address of the destination, it sends it to the resolver (event 8).

Figure 26.37 *Iterative resolution*



**********END**********