UNIT IV

TRANSPORT LAYER

Overview of Transport layer – UDP – Reliable byte stream (TCP) – Connection management – Flow control – Retransmission – TCP Congestion control – Congestion avoidance (DECbit, RED) – QoS – Application requirements

4.1. Overview of Transport layer

- The transport level of the network architecture supports communication between application programs running in end nodes, is sometimes called the end-to-end protocol.
- Common properties that a transport protocol can be expected to provide:
  - ✓ Guarantees message delivery
  - ✓ Delivers messages in the same order they are sent
  - ✓ Delivers at most one copy of each message
  - ✓ Supports arbitrarily large messages
  - ✓ Supports synchronization between the sender and the receiver
  - ✓ Allows the receiver to apply flow control to the sender
  - ✓ Supports multiple application processes on each host
- The transport protocol operates has certain limitations in the level of service it can provide.
  - ✓ Drop messages
  - ✓ Reorder messages
  - ✓ Deliver duplicate copies of a given message
  - ✓ Limit messages to some finite size
  - ✓ Deliver messages after an arbitrarily long delay
- A network is said to provide a best-effort level of service, demultiplexing service, a reliable byte-stream service, a request/reply service, and a service for real-time applications.

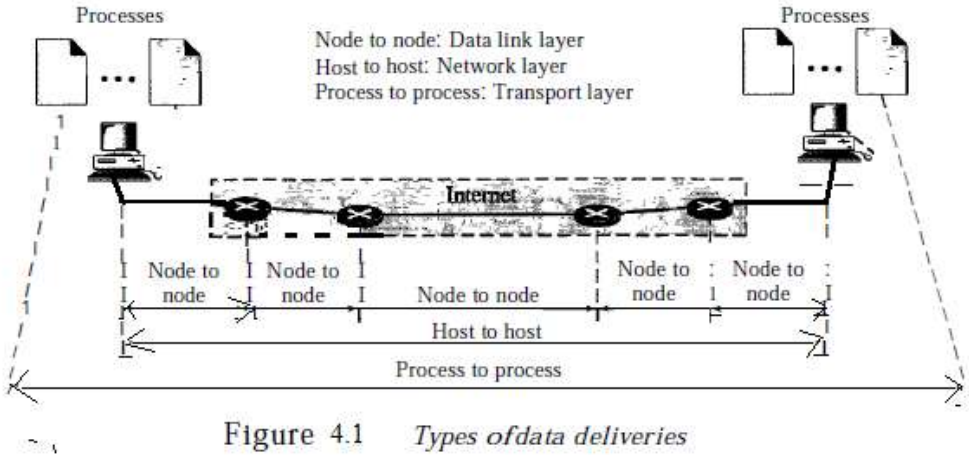


Figure 4.1 *Types ofdata deliveries*

Connectionless Versus Connection-Oriented Service

A transport layer protocol can either be connectionless or connection-oriented.

Connectionless Service

- In a connectionless service, the packets are sent from one party to another with no need for connection establishment or connection release.

The packets are not numbered; they may be delayed or lost or may arrive out of sequence.

- There is no acknowledgment.
- UDP, is connectionless.

ConnectionOriented Service

In a connection-oriented service, a connection is first established between the sender and the receiver. Data are transferred. At the end, the connection is released.

Reliable Versus Unreliable

- The transport layer service can be reliable or unreliable.
- If the application layer program needs reliability, we use a reliable transport layer protocol by implementing flow and error control at the transport layer. This means a slower and more complex service.
- TCP and SCTP are connectionoriented and reliable. These three can respond to the demands of the application layer programs.

4.2. UDP

- The User Datagram Protocol (UDP) is called a connectionless, unreliable transport protocol.
- Advantages:
  - UDP is a very simple protocol using a nrinimum of overhead. If a process wants to send a small message and does not care much about reliability, it can use UDP.
  - Sending a small message by using UDP takes much less interaction between the sender and receiver than using TCP or SCTP.

Well-Known Ports for UDP

Table shows some well-known port numbers used by UDP. Some port numbers can be used by both UDP and TCP.

Well-known ports used with UDP

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service
67	BOOTPs	Server port to download bootstrap information
68	BOOTPc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
III	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

Example In UNIX, the well-known ports are stored in a file called fetcfservices. Each line in this file gives the name of the server and the well-known port number. We can use the *grep* utility to extract the line corresponding to the desired application. The following shows the port for FTP. Note that FTP can use port 21 with either UDP or TCP.

```
$grep    ftp   fetclservices
ftp        21ftcp
fip        211udp

$grep      snmp fetclservices
snmp           161ftcp          #Simple Net  Mgmt Proto
snmp           1611udp          #Simple Net  Mgmt Proto
snmptrap       162/udp          #Traps for SNMP
```

User Datagram format

- UDP packets, called user datagrams, have a fixed-size header of 8 bytes.

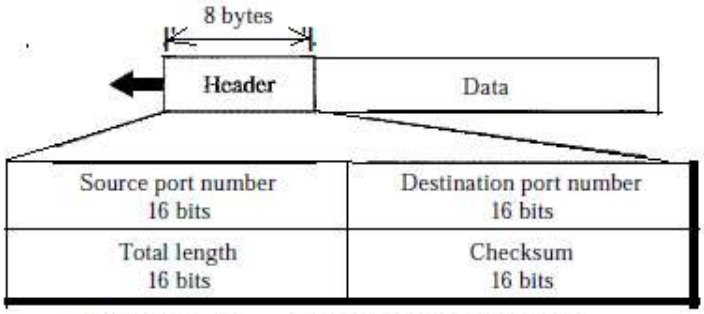


Figure 4.2 *User datagram format*

- Figure 4.2 shows the format of a user datagram.
- The fields are as follows:
  - *Source port number.*
  - This is the port number used by the process running on the source host.
  - It is 16 bits long, which means that the port number can range from 0 to 65,535.
  - If the source host is the client (a client sending a request), the port number, in most cases, is an ephemeral port number requested by the process and chosen by the UDP software running on the source host.
  - If the source host is the server (a server sending a response), the port number, in most cases, is a well-known port number.
  - *Destination port number.*
  - This is the port number used by the process running on the destination host.
  - It is also 16 bits long. If the destination host is the server (a client sending a request), the port number, in most cases, is a well-known port number.
  - If the destination host is the client (a server sending a response), the port number, in most cases, is an ephemeral port number.
  - In this case, the server copies the ephemeral port number it has received in the request packet.
  - *Length*.
  - This is a 16-bit field that defines the total length of the user datagram, header plus data.
  - The 16 bits can define a total length of 0 to 65,535 bytes. The length of a UDP datagram that is encapsulated in an IP datagram.

UDP length = IP length - IP header's length

- *Checksum*

- This field is used to detect errors over the entire user datagram (header plus data).
- The checksum includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer.
- The pseudoheader is the part of the header of the IP packet in which the user datagram is to be encapsulated with some fields filled with 0s (see Figure 4.3).

- If the *IP header* is corrupted, it may be delivered to the wrong host.
- The *protocol field* is added to ensure that the packet belongs to UDP, and not to other transport-layer protocols.
- The value of the protocol field for UDP is 17. If this value is changed during transmission, the checksum calculation at the receiver will detect it and UDP drops the packet. It is not delivered to the wrong protocol.
- The similarities between the pseudoheader fields and the last 12 bytes of the IP header.

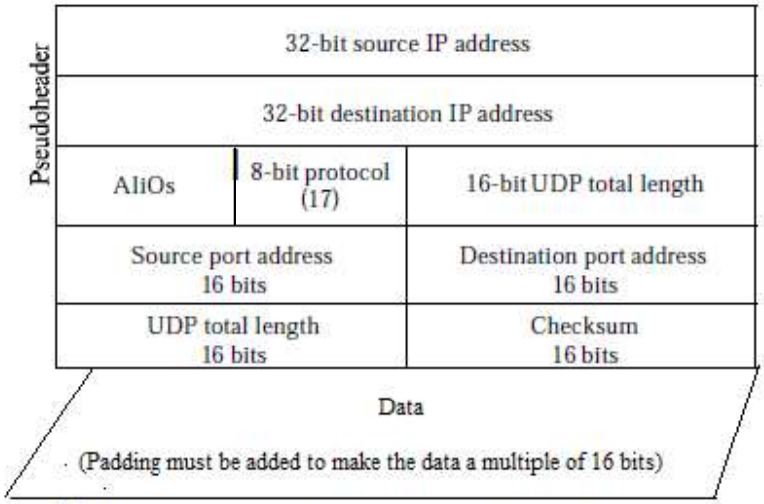


Figure 4.3 *Pseudoheader for checksum calculation*

Example

Figure 4.4 shows the checksum calculation for a very small user datagram with only 7 bytes of data. Because the number of bytes of data is odd, padding is added for checksum calculation.

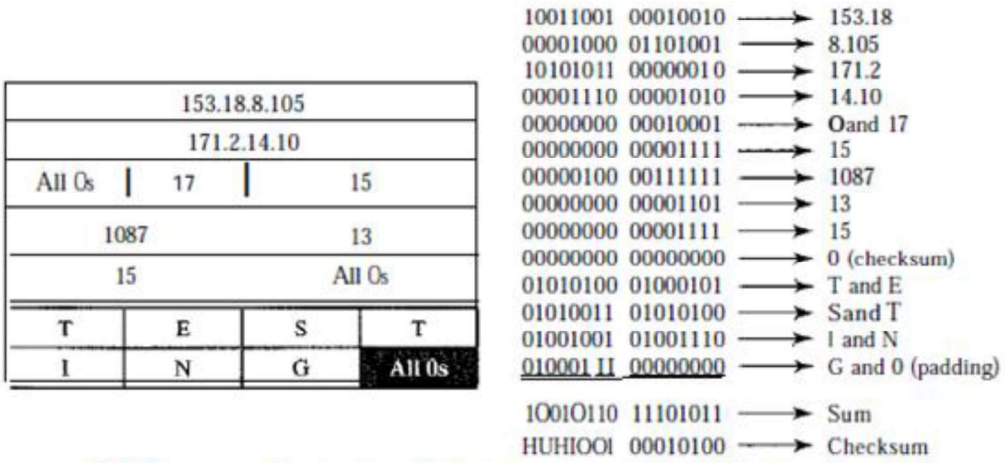


Figure 4.4 *Checksum calculation of a simple UDP user datagram*

The pseudoheader as well as the padding will be dropped when the user datagram is delivered to IP.

Optional Use of the Checksum

The calculation of the checksum and its inclusion in a user datagram are optional. If the checksum is not calculated, the field is filled with 1s.

UDP Operation

Connectionless Services

- UDP provides a connectionless service. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program.
- The user datagrams are not numbered.
- There is no connection establishment and no connection termination, as is the case for TCP. This means that each user datagram can travel on a different path.

Flow and Error Control

- UDP is a very simple, unreliable transport protocol.
- There is no flow control and hence no window mechanism. The receiver may overflow with incoming messages.
- There is no error control mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through the checksum, the user datagram is silently discarded.

Encapsulation and Decapsulation

- To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages in an IP datagram.

Queuing

- At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue associated with each process.

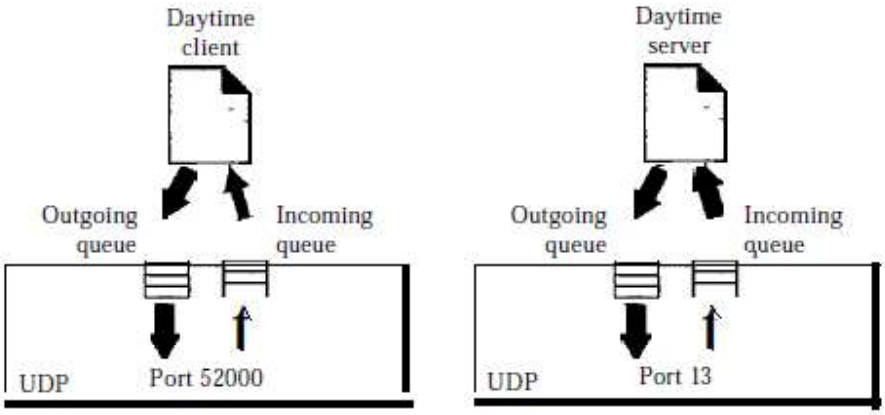


Figure 4.5 *Queues in UDP*

- The queues opened by the client are, in most cases, identified by ephemeral port numbers. The queues function as long as the process is running.
- When the process terminates, the queues are destroyed.

- The client process can send messages to the outgoing queue by using the source port number specified in the request.
- UDP removes the messages one by one and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system can ask the client process to wait before sending any more messages.
- When a message arrives for a client, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a *port unreachable* message to the server.
- All the incoming messages for one particular client program, whether coming from the same or a different server, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the server.
- At the server site, the mechanism of creating queues is different.
- A server asks for incoming and outgoing queues, using its well-known port, when it starts running. The queues remain open as long as the server is running.
- When a message arrives for a server, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram.
- If there a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a port unreachable message to the client.
- All the incoming messages for one particular server, whether coming from the same or a different client, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the client.
- Whe a server wants to respond to a client, it sends messages to the outgoing queue, using the source port number specified in the request. UDP removes the messages one by one and, after adding the UDP header, delivers them to IP.
- An outgoing queue can overflow. If this happens, the operating system asks the server to wait before sending any more messages.

Use of UDP

The following lists some uses of the UDP protocol:

- UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FrP that needs to send bulk data.
- UDP is suitable for a process with internal flow and error control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) process includes flow and error control. It can easily use UDP.
- UDP is a suitable transport protocol for multicasting. Multicasting capability is embedded in the UDP software but not in the TCP software.
- UDP is used for management processes such as SNMP.
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP)

4.3. Reliable byte stream (TCP)

- TCP is a connection oriented protocol; it creates a virtual connection between two TCPs to send data.

- TCP uses flow and error control mechanisms at the transport level that is *reliable* transport protocol.

TCP Services

Process-to-Process Communication

TCP provides process-to-process communication using port numbers. Table lists some well-known port numbers used by TCP.

Table *Well-known ports used by TCP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20	FIP, Data	File Transfer Protocol (data connection)
21	FIP, Control	File Transfer Protocol (control connection)
23	TELNET	Tenninal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol
111	RPC	Remote Procedure Call

Stream Delivery Service

- TCP, is a stream-oriented protocol.
- TCP allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their data across the Internet. This imaginary environment is depicted in Figure 4.6.
- The sending process produces (writes to) the stream of bytes, and the receiving process consumes (reads from) them.

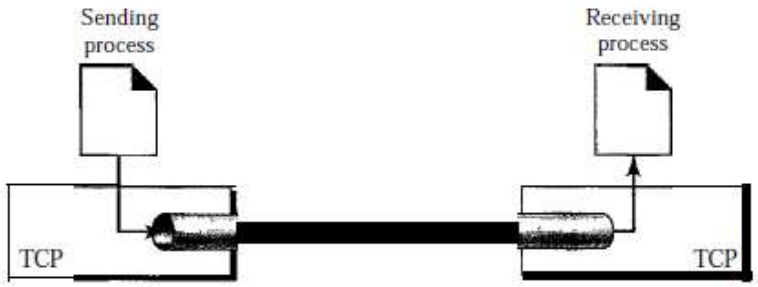


Figure 4.6 *Stream delivery*

- Sending and Receiving Buffers Because the sending and the receiving processes may not write or read data at the same speed, TCP needs buffers for storage.
- There are two buffers, the sending buffer and the receiving buffer, one for each direction.

- Figure 4.6. shows two buffers of 20 bytes each normally the buffers are hundreds or thousands of bytes, depending on the implementation.
- Figure 4.6 shows the movement of the data in one direction. At the sending site, the buffer has three types of chambers.
- The white section contains empty chambers that can be filled by the sending process (producer). The gray area holds bytes that have been sent but not yet acknowledged. TCP keeps these bytes in the buffer until it receives an acknowledgment.
- The colored area contains bytes to be sent by the sending TCP.

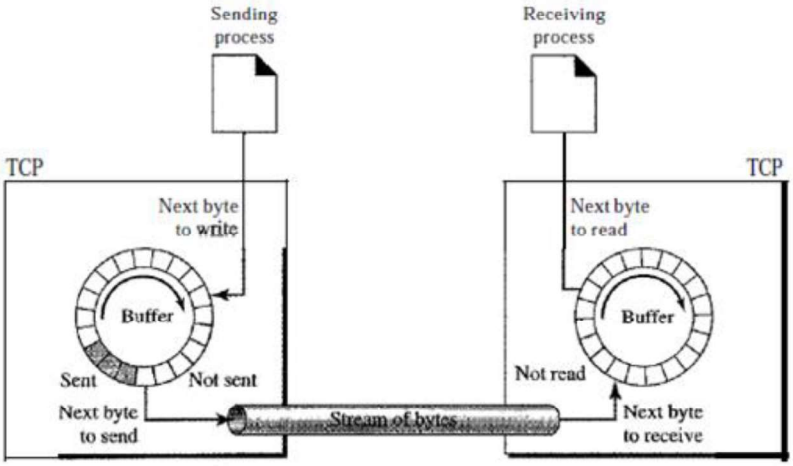


Figure 4.7 *Sending and receiving buffers*

- Also note that after the bytes in the gray chambers are acknowledged, the chambers are recycled and available for use by the sending process.
- The circular buffer is divided into two areas (shown as white and colored). The white area contains empty chambers to be filled by bytes received from the network. The colored sections contain received bytes that can be read by the receiving process. When a byte is read by the receiving process, the chamber is recycled and added to the pool of empty chambers.
- TCP groups a number of bytes together into a packet called a segment. TCP adds a header to each segment (for control purposes) and delivers the segment to the IP layer for transmission. The segments are encapsulated in IP datagrams and transmitted. This entire operation is transparent to the receiving process.
- Figure 4.4 shows how segments are created from the bytes in the buffers. In Figure 4.8, for simplicity, show one segment carrying 3 bytes and the other carrying 5 bytes.
- At the sending side, three pointers are maintained into the send buffer, each with an obvious meaning: LastByteAcked, LastByteSent, and LastByteWritten.

  $$\texttt{LastByteAcked} \leq \texttt{LastByteSent}$$

  since the receiver cannot have acknowledged a byte that has not yet been sent, and

  $$\texttt{LastByteSent} \leq \texttt{LastByte}$$

  since TCP cannot send a byte that the application process has not yet written.

- set of pointers (sequence numbers) are maintained on the receiving side: LastByteRead, NextByteExpected, and LastByteRcvd. Theinequalities are a little less intuitive, however, because of the problem of out-of-order delivery. The first relationship

  `LastByteRead < NextByteExpected`

  is true because a byte cannot be read by the application until it is received *and* all preceding bytes have also been received.
- NextByteExpected points to the byte immediately after the latest byte to meet this criterion. Second,

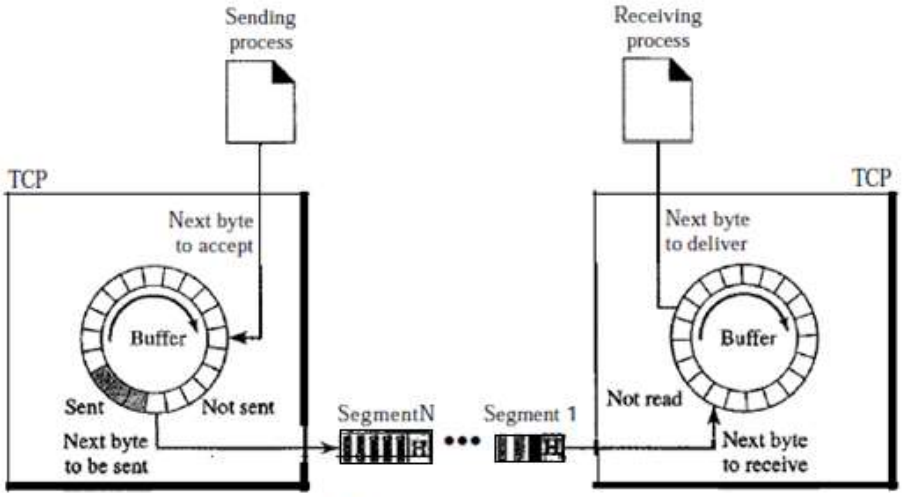


Figure 4.8 *TCP segments*

`NextByteExpected ≤ LastByteRcvd+1`

since, if data has arrived in order, NextByteExpected points to the byte after LastByteRcvd, whereas if data has arrived out of order, then NextByteExpected points to the start of the first gap in the data.

Full-Duplex Communication

- TCP offers full-duplex service, in which data can flow in both directions at the same time. Each TCP then has a sending and receiving buffer, and segments move in both directions.

Connection-Oriented Service

- TCP is a connection-oriented protocol. When a process at site A wants to send and receive data from another process at site B, the following occurs:
  1. The two TCPs establish a connection between them.
  2. Data are exchanged in both directions.
  3. The connection is terminated.

Reliable Service

- TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data.

TCP Features

Numbering System

- Two fields called the sequence number and the acknowledgment number. These two fields refer to the byte number and not the segment number.

- Byte Number TCP numbers all data bytes that are transmitted in a connection.
- Numbering is independent in each direction. When TCP receives bytes of data from a process, it stores them in the sending buffer and numbers them. The numbering does not necessarily start from 0.
- TCP generates a random number between 0 and $^{232-1}$ for the number of the first byte. For example, if the random number happens to be 1057 and the total data to be sent are 6000 bytes, the bytes are numbered from 1057 to 7056.
- The bytes of data being transferred in each connection are numbered by TCP. The numbering starts with a randomly generated number.
- The sequence number for each segment is the number of the first byte carried in that segment.

Example

Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1000 bytes?

Solution

The following shows the sequence number for each segment:

Segment 1	Sequence Number: 10,001 (range: 10,001 to 11,(00)
Segment 2	Sequence Number: 11,001 (range: 11,001 to 12,000)
Segment 3	Sequence Number: 12,001 (range: 12,001 to 13,000)
Segment 4	Sequence Number: 13,001 (range: 13,001 to 14,000)
Segment 5	Sequence Number: 14,001 (range: 14,001 to 15,000)

- When carrying only control information, need a sequence number to allow an acknowledgment from the receiver. These segments are used for connection establishment, termination, or abortion.
- Each of these segments consumes one *sequence number* as though it carried 1 byte, but there are no actual data. If the randomly generated sequence number is *x,* the first data byte is numbered $x + 1$.
- *Acknowledgment Number* - communication in TCP is full duplex; when a connection is established, both parties can send and receive data at the same time. Each party numbers the bytes, usually with a different starting byte number. The sequence number in each direction shows the number of the first byte carried by the segment. Each party also uses an acknowledgment number to confirm the bytes it has received.
- The acknowledgment number is cumulative, which means that the party takes the number of the last byte that it has received, safe and sound, adds I to it, and announces this sum as the acknowledgment number. The term *cumulative* here means that if a party uses 5643 as an acknowledgment number, it has received all bytes from the beginning up to 5642.

Flow Control

TCP provides *flow control.* The receiver of the data controls the amount of data that are to be sent by the sender. This is done to prevent the receiver from being overwhelmed with data. The numbering system allows TCP to use a byte-oriented flow control.

Error Control

To provide reliable service, TCP implements an error control mechanism. Error control considers a segment as the unit of data for error detection (loss or corrupted segments), error control is byte-oriented.

Congestion Control

TCP takes into account congestion in the network. The amount of data sent by a sender is not only controlled by the receiver (flow control), but is also determined by the level of congestion in the network.

Segment

A packet in TCP is called a segment.

Format

The format of a segment is shown in Figure 4.9.

- The segment consists of a 20- to 60-byte header, 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.
- *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. This serves the same purpose as the source port address in the UDP header.
- *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. This serves the same purpose as the destination port address in the UDP header.
- *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 comprises the first byte in the segment.

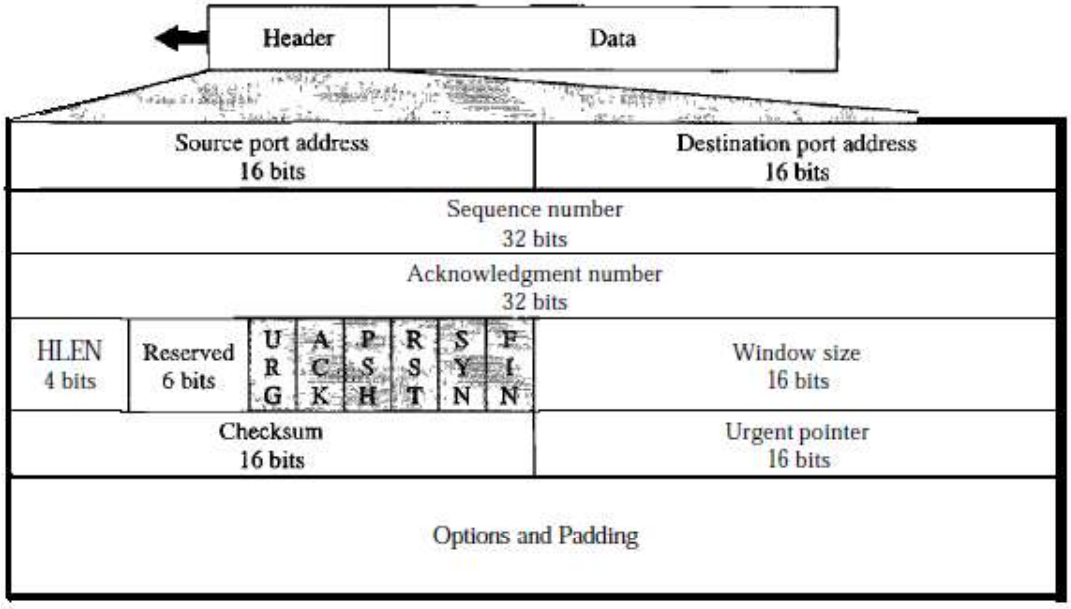


Figure 4.9 *TCP segment format*

- During connection establishment, 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 defines *x* + I 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 can be between 5 (5 x 4 =20) and 15 (15 x 4 =60).
- *Reserved.* This is a 6-bit field reserved for future use.

- *Control.* This field defines 6 different control bits or flags as shown in Figure 4.10. One or more of these bits can be set at a time.

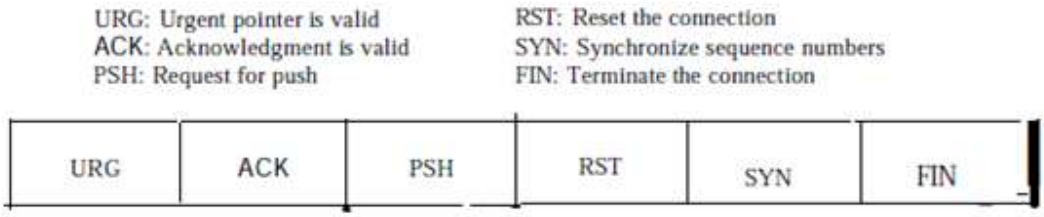


Figure 4.10 *Control field*

- *Window size.* This field defines the size of the window, in bytes, that the other party must maintain. 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 inclusion of the checksum in the UDP datagram is optional, whereas the inclusion 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.

Table *Description offlags in the controlfield*

Flag	*Description*
URG	The value of the urgent pointer field is valid.
ACK	The value of the acknowledgment field is valid.
PSH	Push the data.
RST	Reset the connection.
SYN	Synchronize sequence numbers during connection.
FIN	Terminate the connection.

- *Urgent pointer.* This l6-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data. It defines the number that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.
- *options.* There can be up to 40 bytes of optional information in the TCP header.

4.3.1. Connection management

- TCP is connection-oriented. A connection-oriented transport protocol establishes a virtual path between the source and destination. All the segments belonging to a message are then sent over this virtual path.
- TCP operates at a higher level. TCP uses the services of IP to deliver individual segments to the receiver, but it controls the connection itself.
- If a segment is lost or corrupted, it is retransmitted. Unlike TCP, IP is unaware of this retransmission. If a segment arrives out of order, TCP holds it until the missing segments arrive; IP is unaware of this reordering.
- Connection-oriented transmission requires three phases:
    1. Connection establishment phase
    2. Data transfer phase

3. Connection termination phase

1. Connection Establishment phase

- TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously. This implies that each party must initialize communication and get approval from the other party before any data are transferred.
- *Three-Way Handshaking* The connection establishment in TCP is called three way handshaking.
  In our example, an application program, called the client, wants to make a connection with another application program, called the server, using TCP as the transport layer protocol.
- The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This is called a request for a *passive open.*
- The client program issues a request for an *active open.* A client that wishes to connect to an open server tells its TCP that it needs to be connected to that particular server. TCP can now start the three-way handshaking process as shown in Figure 4.11.
- Fig. show the sequence number, the acknowledgment number, the control flags (only those that are set), and the window size, if not empty. The three steps in this phase are as follows.

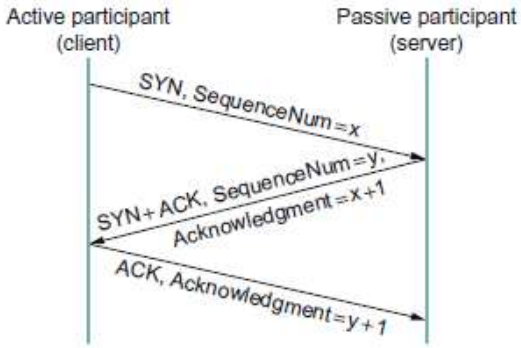


Figure 4.11 Timeline for three-way handshake algorithm

  1. The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization of sequence numbers. It consumes one sequence number. When the data transfer starts, the sequence number is incremented by 1. The SYN segment carries no real data, but we can think of it as containing 1 imaginary byte. A SYN segment cannot carry data, but it consumes one sequence number.
  2. The server sends the second segment, a SYN +ACK segment, with 2 flag bits set: SYN and ACK. This segment has a dual purpose. It is a SYN segment for communication in the other direction and serves as the acknowledgment for the SYN segment. It consumes one sequence number. A SYN +ACK segment cannot carry data, but does consume one sequence number.
  3. The client sends the third segment. This is just an ACK segment. It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field.

2. Data Transfer phase

- After connection is established, bidirectional data transfer can take place. The client and server can both send data and acknowledgments.
- The acknowledgment is piggybacked with the data. Figure 4.12 shows an example.
- In this example, after connection is established (not shown in the figure), the client sends 2000 bytes of data in two segments. The server then sends 2000 bytes in one segment. The client sends one more segment. The first three segments carry both data and acknowledgment, but the last segment carries only an acknowledgment because there are no more data to be sent.

- The data segments sent by the client have the PSH (push) flag set so that the server TCP knows to deliver data to the server process as soon as they are received.
- The sending TCP can select the segment size. The receiving TCP also buffers the data when they arrive and delivers them to the application program when the application program is ready or when it is convenient for the receiving TCP. This type of flexibility increases the efficiency of TCP.
- Delayed transmission and delayed delivery of data may not be acceptable by the application program.TCP can handle such a situation. The application program at the sending site can request a *push* operation. This means that the sending TCP must not wait for the window to be filled. It must create a segment and send it immediately. The sending TCP must also set the push bit (PSH) to let the receiving TCP know that the segment includes data that must be delivered to the receiving application program as soon as possible and not to wait for more data to come.

Urgent Data

- An application program needs to send *urgent* bytes. This means that the sending application program wants a piece of data to be read out of order by the receiving application program.

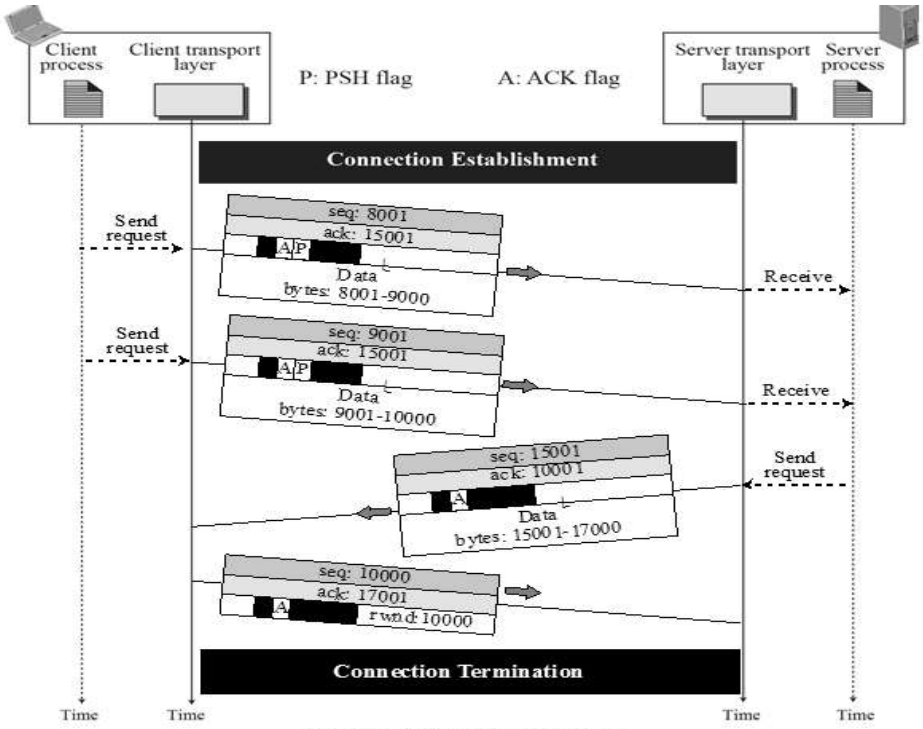


Figure 4.12 Data Transfer Phase

- As an example, suppose that the sending application program is sending data to be processed by the receiving application program.
- When the result of processing comes back, the sending application program finds that everything is wrong. It wants to abort the process, but it has already sent a huge amount of data. If it issues an abort command (control C), these two characters will be stored at the end of the receiving TCP buffer. It will be delivered to the receiving application program after all the data have been processed. The solution is to send a segment with the URG bit set.
- The sending application program tells the sending TCP that the piece of data is urgent. The sending TCP creates a segment and inserts the urgent data at the beginning of the segment.

- The rest of the segment can contain normal data from the buffer. The urgent pointer field in the header defines the end of the urgent data and the start of normal data.
- When the receiving TCP receives a segment with the URG bit set, it extracts the urgent data from the segment, using the value of the urgent pointer, and delivers them, out of order, to the receiving application program.

Connection Termination

- This is also called as three-way handshaking and four-way handshaking with a half-close option.
- *Three-way handshaking* for connection termination as shown in Figure 4.13.

1. In a normal situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set. Note that a FIN segment can include the last chunk of data sent by the client, or it can be just a control segment as shown in Figure 4.12. If it is only a control segment, it consumes only one sequence number. The FIN segment consumes one sequence number ifit does not carry data.
2. The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN +ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to

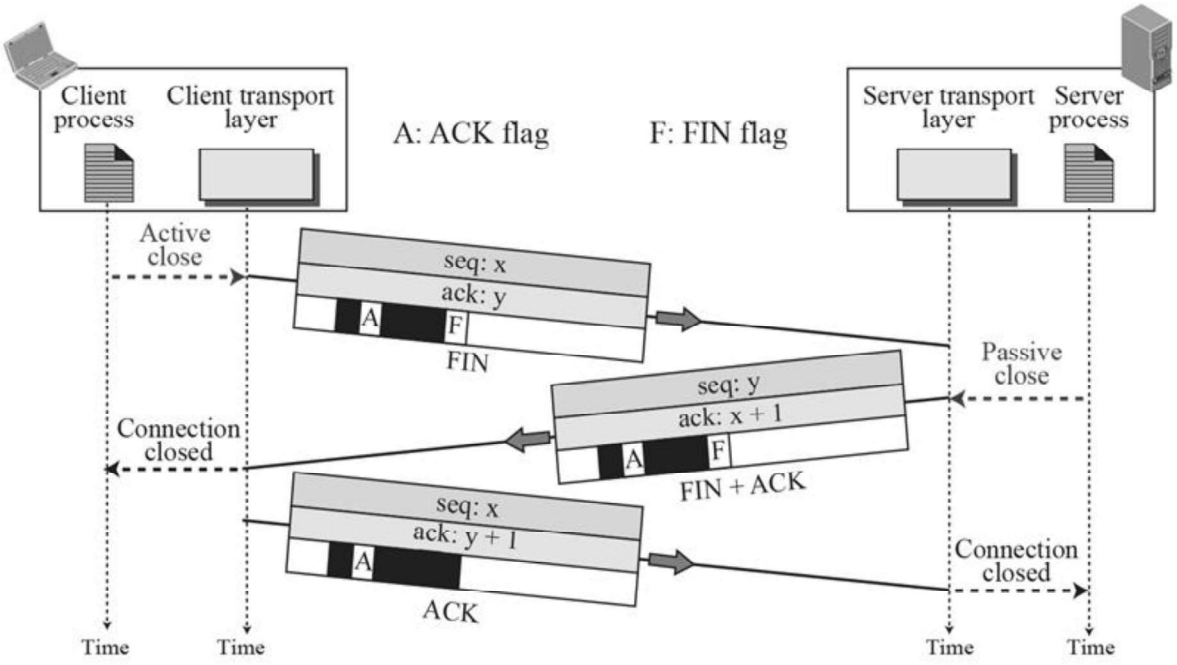


Figure 4.13 Connection Termination - 3WH

announce the closing of the connection in the other direction. This segment can also contain the last chunk of data from the server. If it does not carry data, it consumes only one sequence number. The FIN +ACK segment consumes one sequence number if it does not carry data.

3. The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server. This segment contains the acknowledgment number, which is 1 plus the sequence number received in the FIN segment from the server. This segment cannot carry data and consumes no sequence numbers.

Half-Close

- One end can stop sending data while still receiving data. This is called a half-close.

- A good example is sorting. When the client sends data to the server to be sorted, the server needs to receive all the data before sorting can start. This means the client, after sending all the data, can close the connection in the outbound direction.
- The server, after receiving the data, still needs time for sorting; its outbound direction must remain open.
- Figure 4.14 shows an example of a half-close. The client half-closes the connection by sending a FIN segment. The server accepts the half-close by sending the ACK segment. The data transfer from the client to the server stops. The server, however, can still send data. When the server has sent all the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client.
- After half-closing of the connection, data can travel from the server to the client and acknowledgments can travel from the client to the server. The client cannot send any more data to the server. Note the sequence numbers we have used. The second segment (ACK) consumes no sequence number. Although the client has received sequence number *y* - 1 and is expecting *y,* the server sequence number is still *y* - 1.
- When the connection finally closes, the sequence number of the last ACK segment is still because no sequence numbers are consumed during data transfer in that direction.

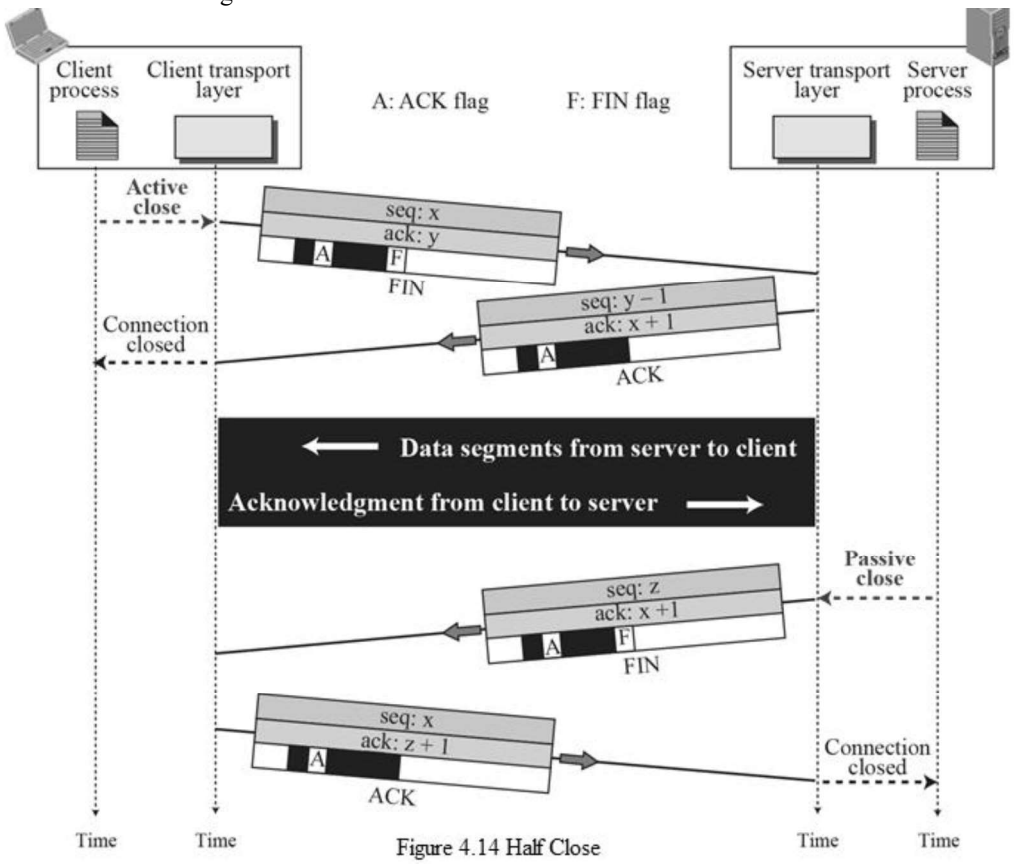


Figure 4.14 Half Close

State-Transition Diagram

- Figure 4.15. shows state transition diagram of TCP
- This diagram shows only the states involved in opening a connection (everything above ESTABLISHED) and in closing a connection (everything below ESTABLISHED).

- Everything that goes on while a connection is open—that is, the operation of the sliding window algorithm—is hidden in the ESTABLISHED state.
- Each circle denotes a state that one end of a TCP connection can find itself in. All connections start in the CLOSED state.
- The connection progresses, the connection moves from state to state according to the arcs. Each arc is labeled with a tag of the form *event/action*.
  - if a connection is in the LISTEN state and a SYN segment arrives (i.e., a segment with the SYN flag set), the connection makes a transition to the SYN RCVD state and takes the action of replying with an ACK+SYN segment.
- Two kinds of events trigger a state transition: (1) *a segment arrives from the peer* (e.g., the event on the arc from LISTEN to SYN RCVD), or (2) *the local application process invokes an operation on TCP* (e.g., the *active open* event on the arc from CLOSED to SYN SENT).
- TCP's state-transition diagram effectively defines the *semantics* of both its peer-to-peer interface and its service interface
- The *syntax* of these two interfaces is given by the segment format and by some application programming interface respectively.
- When opening a connection, the server first invokes a passive open operation on TCP, which causes TCP to move to the LISTEN state. At some later time, the client does an active open, which causes its end of the connection to send a SYN segment to the server and to move to the SYN SENT state.
- When the SYN segment arrives at the server, it moves to the SYN RCVD state and responds with a SYN+ACK segment. The arrival of this segment causes the client to move to the ESTABLISHED state and to send an ACK back to the server.
- When this ACK arrives, the server finally moves to the ESTABLISHED state.
- There are three things to notice about the connection establishment half of the state-transition diagram.
  - First, if the client's ACK to the server is lost, corresponding to the third leg of the three-way handshake, then the connection still functions correctly. This is because the client side is already in the ESTABLISHED state, so the local application process can start sending data to the other end. Each of these data segments will have the ACK flag set, and the correct value in the Acknowledgment field, so the server will move to the ESTABLISHED state when the first data segment arrives. This is actually an important point about TCP—every segment reports what sequence number the sender is expecting to see next, even if this repeats the same sequence number contained in one or more previous segments.
  - The second thing to notice about the state-transition diagram is that there is a funny transition out of the LISTEN state whenever the local process invokes a *send* operation on TCP. That is, it is possible for a passive participant to identify both ends of the connection (i.e., itself and the remote participant that it is willing to have connect to it), and then for it to change its mind about waiting for the other side and instead actively establish the connection. To the best of our knowledge, this is a feature of TCP that no application process actually takes advantage of.
  - The final thing to notice about the diagram is the arcs that are not shown. Specifically, most of the states that involve sending a segment to the other side also schedule a timeout that eventually causes the segment to be

present if the expected response does not happen. These retransmissions are not depicted in the state-transition diagram. If after several tries the expected response does not arrive, TCP gives up and returns to the CLOSED state.

- If only one side closes the connection, then this means it has no more data to send, but it is still available to receive data from the other side.
- Any one side there are three combinations of transitions that get a connection from the ESTABLISHED state to the CLOSED state:

i. This side closes first: ESTABLISHED→FIN WAIT 1→FIN WAIT 2→TIME WAIT→CLOSED.

ii. The other side closes first: ESTABLISHED→CLOSE WAIT→LAST ACK→CLOSED.

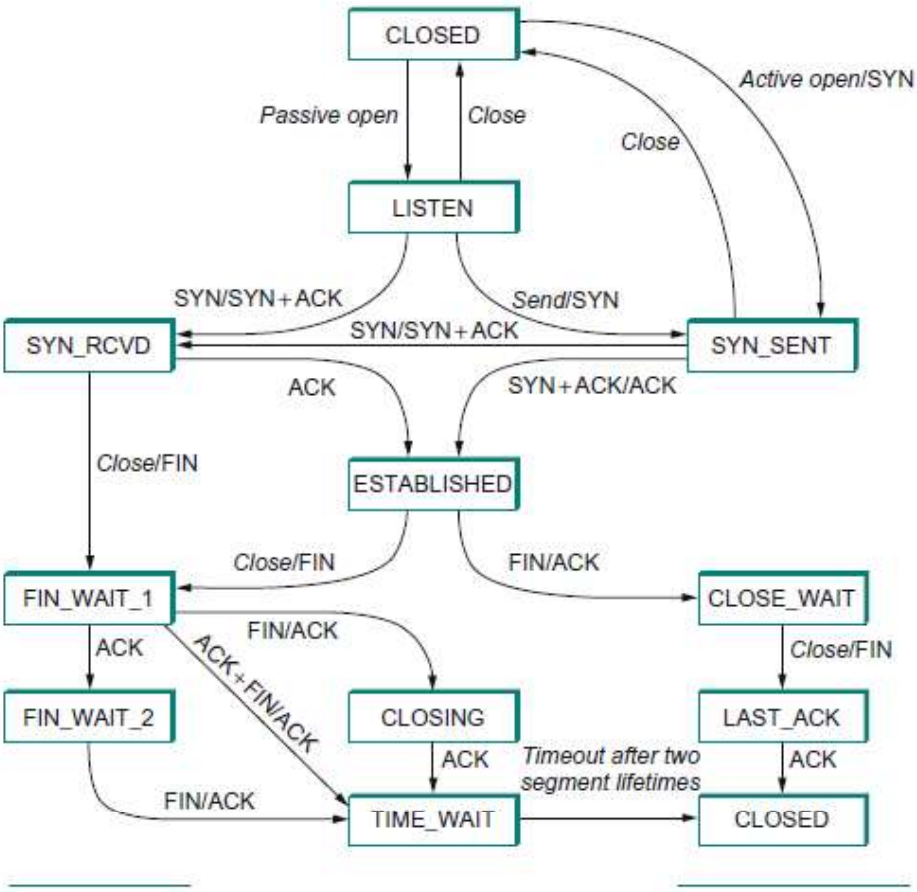


Figure 4.15 State Transition Diagram

iii. Both sides close at the same time: ESTABLISHED→FIN WAIT 1→CLOSING→TIME WAIT→CLOSED.

- The CLOSED state; it follows the arc from FIN WAIT 1 to TIME WAIT.
- *Connection Teardown*
  - A connection in the TIME WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the Internet (i.e., 120 seconds).
  - The local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered.

- Retransmit its FIN segment, and this second FIN segment might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection (i.e., use the same pair of port numbers), and the delayed FIN segment from the earlier incarnation of the connection would immediately initiate the termination of the later incarnation of that connection.

4.3.2. Remote Procedure Call

✓ RPC is a powerful technique for constructing distributed, client/server based applications, also called *message transaction*: A client sends a request message to a server, and the server responds with a reply message, with the client blocking (suspending execution) to wait for the reply.

✓ Figure 4.3.2 (a) illustrates the basic interaction between the client and server in such a message transaction.

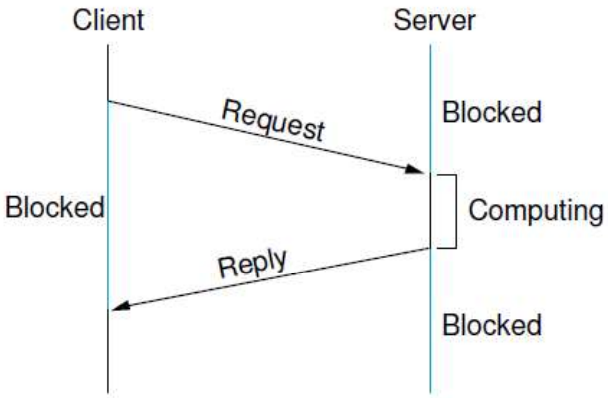


Fig 4.3.2(a) RPC Timeline

Advantages of RPC

- UDP does not guarantee any reliability
- TCP incurs high overhead (e.g., setting up and tearing down the connection) simply to delivery a pair of request/reply messages

Components of RPC

- Two major components:

1. A protocol that manages the messages sent between the client and the server processes and that deals with the potentially undesirable properties of the underlying network
2. Programming language and compiler support to package the arguments into a request message on the client machine and then to translate this message back into the arguments on the server machine, and likewise with the return value (this piece of the RPC mechanism is usually called a *stub compiler*)

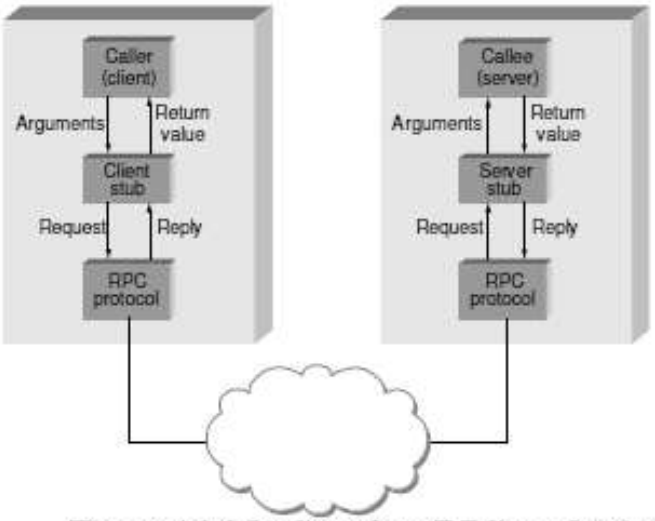


Figure 4.3.2(b) Complete RPC mechanism

Process of RPC

Figure 4.3.2(b) schematically depicts what happens when a client invokes a remote procedure.

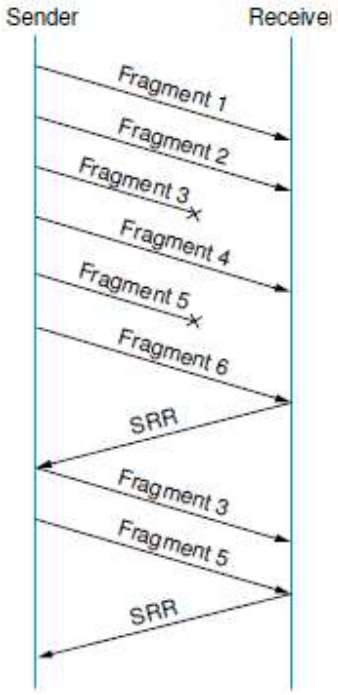
✓ First, the client calls a local stub for the procedure, passing it the arguments required by the procedure. This stub hides the fact that the procedure is remote by translating the arguments into a request message and then invoking an RPC protocol to send the request message to the server machine.

- ✓ At the server, the RPC protocol delivers the request message to the server stub (sometimes called a *skeleton*), which translates it into the arguments to the procedure and then calls the local procedure. After the server procedure completes, it returns the answer to the server stub, which packages this return value in a reply message that it hands off to the RPC protocol for transmission back to the client.
- ✓ The RPC protocol on the client passes this message up to the client stub, which translates it into a return value that it returns to the client program.

RPC Protocol (stack)

- ✓ fragments and reassembles large messages (by BLAST)
- ✓ synchronizes request and reply messages (by CHAN)
- ✓ dispatches request to the correct process/procedure (by SELECT)

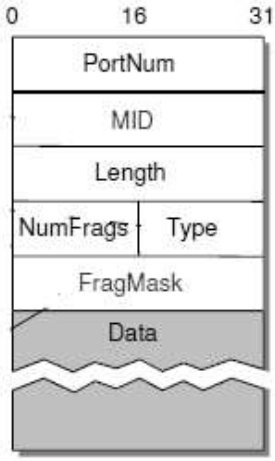
Bulk Transfer (BLAST)



- Fragmentation & reassembly as in ATM-AAL and IP
- Unlike AAL and IP, tries to recover from lost fragments
  - So as not to retransmit the whole large packet (for higher efficiency)
  - Strategy: selective retransmission via negative acknowledgment
- But does not go so far to guarantee 100% reliable delivery
  - Does not wait for any of the fragment to be acked before sending the next (hence the name Blast)
- ***Sender***
  - temporarily keeps a fragment for potential retransmission
    - after sending all fragments, set timer DONE
    - if receive Selective Retransmission Request (SRR), send missing fragments and reset DONE
      - if timer DONE expires, free fragments; -Give up if there is lost fragments
  - SRR acts as negative acknowledgment
  - Interprets lost negative-ack as “fragments have been received” - Thus, does not guarantee reliable fragment delivery
- ***Receiver***
  - in the presence of fragment loss, sends limited number of retransmission requests
    - When the first fragment arrives, set timer LAST_FRAG _ LAST_FRAG is reset whenever receiving a new fragment
    - When all fragments are present, reassemble and pass up
    - Four exceptional conditions:

```
if the last fragment arrives, but message not complete
     send SRR and set timer RETRY
  if timer LAST_FRAG expires
     send SRR and set timer RETRY
 if timer RETRY expires for first or second time
```

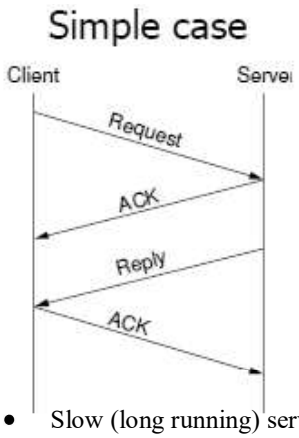
```
send SRR and set timer RETRY
if timer RETRY expires a third time
give up and free partial message
```

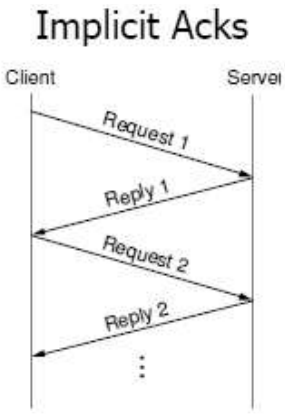


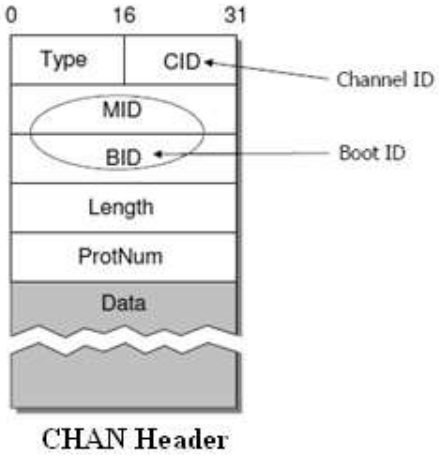
- **BLAST Header Format**
  - ✓ MID (message ID): must protect against wrap around
  - ✓ TYPE = DATA or SRR
  - ✓ NumFrags: indicates total number of fragments in the message
  - ✓ FragMask distinguishes among fragments
    - o if Type=DATA, identifies fragments carried in this packet
    - o if Type=SRR, identifies missing fragments
- **Summary of BLAST**
  - ✓ For fragmentation and reassembly, but tries to recover lost fragments (to improve efficiency)
  - ✓ No sliding-window flow control (unlike sliding window for reliable transmission)
    - o Only need to deliver a single "large" message each time; no need for flow and congestion control
  - ✓ No guarantee on reliable message delivery, instead efficiencyoriented fragment retransmission
    - o Reliability is guaranteed by the protocol next layer up, i.e., CHAN

CHAN Request/Reply

- Guarantees message delivery
- Synchronizes client with server
  - ✓ Implements a logical request/reply channel between client and server (thus the name CHAN)
  - ✓ At most one message is active on a given channel at any time
- Supports at-most-once semantics
- Lost message (request, reply, or ACK)
  - ✓ set RETRANSMIT timer
  - ✓ use message id (MID) field to distinguish
  - ✓ Machines crash and reboot
    - o use boot id (BID) field to distinguish







CHAN Header

- Slow (long running) server
  - ✓ client periodically sends "are you alive" probe, or
  - ✓ server periodically sends "I'm alive" notice

- Want to support multiple outstanding calls
  - ✓ use channel id (CID) field to distinguish

Dispatcher (SELECT)

- Dispatch to appropriate procedure
- Synchronous counterpart to UDP
- Implement concurrency (open multiple CHANs)
- Address Space for Procedures
  - ✓ *flat:* unique id for each possible procedure
  - ✓ *hierarchical:* program + procedure number

4.3.3. RTP

- Real-time traffic: digitized voice, video, etc.
- Experiments with real-time traffic since 1981
- **Advantages**
  - ✓ UDP: best effort, no guarantee on delay and delay jitter
  - ✓ TCP: long delay and large delay jitter due to retransmission
  - ✓ RPC: designed for interactive exchange of (mostly short) messages
- **Requirements for real-time traffic transport**
  - ✓ To be generic and to support different applications(e.g., w/ diff. encoding schemes)
  - ✓ To identify timing relationship among received data;
  - ✓ To synchronize related media streams (e.g., audio & video data streams)
  - 5. To detect and report packet loss (even though no need for 100% reliability)
- **Features of RTP: Real-time Transport Protocol**
  - ✓ Runs over UDP
  - ✓ Application-Level Framing : leave application specific details to applications through "profile" and "formats"
  - ✓ Profile: specifies how to interpret the RTP header information
  - ✓ Format: specifies how to interpret the data following the RTP header
  1. Data packets: specified by RTP
     - Timestamp: for timing and synchronization
       - At application-specific granularity (app defines "tick")
       - Sequence number: for detecting lost or misordered packets
  2. Periodic control packets: specified by RTCP (Real-time Transport Control Protocol)
       - loss rate (fraction of packets received since last report)
       - delay jitter

4.3.4. Flow control

❖ Both buffers are of some finite size, denoted MaxSendBuffer and MaxRcvBuffer

❖ A sliding window protocol, the size of the window setsthe amount of data that can be sent without waiting for acknowledgment from the receiver. Thus, the receiver throttles the sender by advertising a window that is no larger than the amount of data that it can buffer.

- Observe that TCP on the receive side must keep to avoid overflowing its buffer.

```
LastByteRcvd-LastByteRead ≤ MaxRcvBuffer
```

- It therefore advertises a window size of

```
AdvertisedWindow = MaxRcvBuffer-((NextByteExpected-1)-LastByteRead)
```

  - which represents the amount of free space remaining in its buffer.
  - LastByteRcvd moves to the right (is incremented), meaning that the advertised window potentially shrinks.
  - TCP on the send side must then adhere to the advertised windowit gets from the receiver. This means that at any given time, it must ensure that

```
LastByteSent-LastByteAcked ≤ AdvertisedWindow
```

- The sender computes an *effective* window that limits how much data it can send:

```
EffectiveWindow = AdvertisedWindow-(LastByteSent-LastByteAcked)
```

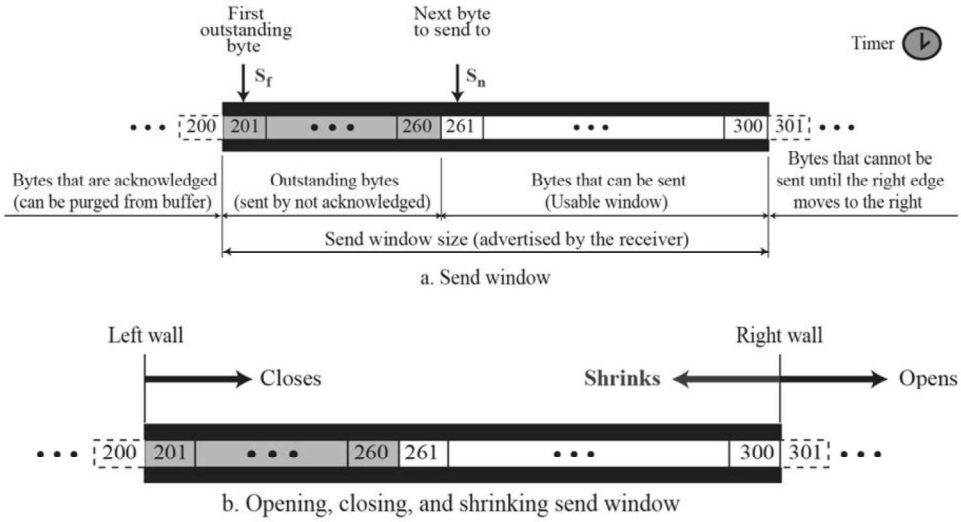


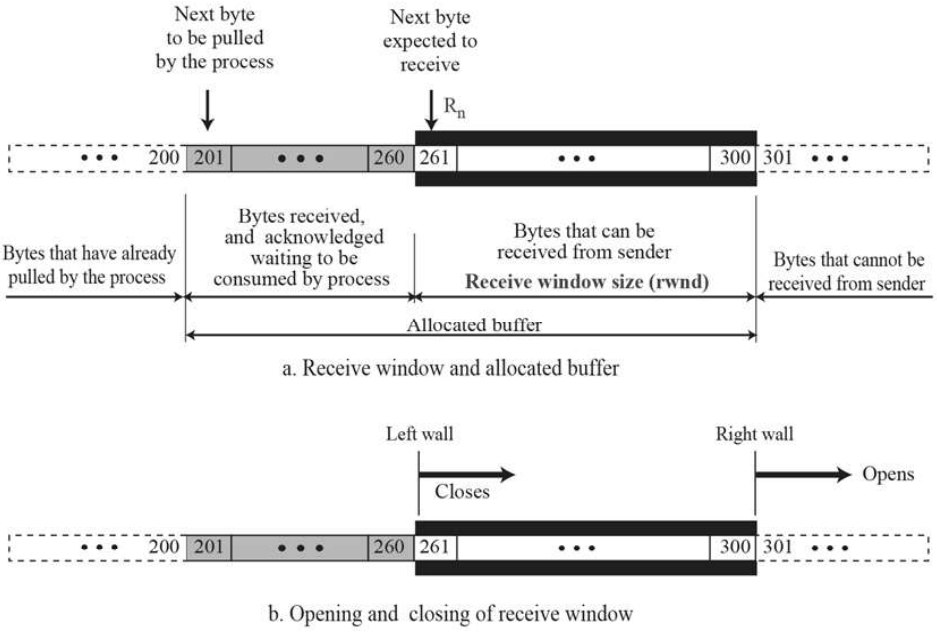
Figure 4.16 Send Sliding Window

  - EffectiveWindow must be greater than 0 before the source can send more data. It is possible, therefore, that a segment arrives acknowledging x bytes, thereby allowing the sender to increment LastByteAcked by x, but because the receiving process was not reading any data, the advertised window is now x bytes smaller than the time before.

- The send side must also make sure that the local application process does not overflow the send buffer—that is,

```
LastByteWritten-LastByteAcked ≤ MaxSendBuffer
```

- If the sending process tries to write y bytes to TCP, but (LastByteWritten−LastByteAcked)+y > MaxSendBuffer then TCP blocks the sending process and does not allow it to generate more data.
- Figure 4.16 shows *Send window in TCP*
- Figure 4.17 shows *Receive Sliding Window*



Figue 4.17 Receive Sliding Window

Silly Window Syndrome

- The silly window syndrome is only a problem when either the sender transmits a small segment or the receiver opens the window a small amount. If neither of these happens, then the small container is never introduced into the stream. It's not possible to outlaw sending small segments.
- For example, the application might do a *push* after sending a single byte. It is possible, however, to keep the receiver from introducing a small container (i.e., a small open window). The rule is that after advertising a zero windowthe receiver must wait for space equal to an MSS before it advertises an open window.
- Figure 4.18 helps visualize what happens.

Figue 4.18 Silly window syndrome. (a) As long as the sender sends MSS-sized segments and the receiver ACKs one MSS at a time, the system works smoothly. (b) As soon as the sender sends less than one MSS, or the receiver ACKs less than one MSS, a small "container" enters the system and continues to circulate.

- ✓ If you think of a TCP stream as a conveyer belt with "full" containers (data segments) going in one direction and empty containers (ACKs) going in the reverse direction, then MSS-sized segments correspond to large containers and 1-byte segments correspond to very small containers.
- ✓ As long as the sender is sending MSS-sized segments and the receiver ACKs at least one MSS of data at a time, everything is good (Figure 4.18(a)).
- ✓ But, what if the receiver has to reduce the window, so that at some time the sender can't send a full MSS of data? If the sender aggressively fills a smaller-than-MSS empty container as soon as it arrives, then the receiver will ACK that smaller number of bytes, and hence the small container introduced into the system remains in the system indefinitely. That is, it is immediately filled and emptied at each end and is never coalesced with adjacent containers to create larger containers, as in Figure 4.18(b).

Nagle's Algorithm

- ❖ if there is data to send but the window is open less than MSS, then we may want to wait some amount of time before sending the available data, but the question is how long? If we wait too long, then we hurt interactive applications like Telnet. If we don't wait long enough, then we risk sending a bunch of tiny packets and falling into the silly window syndrome.
  - ✓ The answer is to introduce a timer and to transmit when the timer expires.
  - ✓ While we could use a clock-based timer—for example, one that fires every 100 ms—Nagle introduced an elegant *self-clocking* solution.
  - ✓ The sender will eventually receive an ACK. This ACK can be treated like a timer firing, triggering the transmission of more data.
- ❖ Nagle's algorithm provides a simple, unified rule for deciding when to transmit:

```
When the application produces data to send
        if both the available data and the window ≥ MSS
                send a full segment
        else
                if there is unACKed data in flight
                        buffer the new data until an ACK arrives
                else
                        send all the new data now
```

- ❖ Some segments will contain a single byte, while others will contain as many bytes as the user was able to type in one round-trip time.
- ❖ The socket interface allows the application to turn off Nagel's algorithm by setting the TCP NODELAY option.
  - Setting this option means that data is transmitted as soon as possible.

i. Retransmission

- ❖ TCP guarantees the reliable delivery of data, it retransmits each segment if an ACK is not received in a certain period of time.
- ❖ TCP sets this timeout as a function of the RTT it expects between the two ends of the connection. Unfortunately, given the range of possible RTTs between any pair of hosts in the Internet, as well as the variation in RTT between the same two hosts over time, choosing an appropriate timeout value is not that easy.

❖ ***Original Algorithm***

- ✓ For computing a timeout value between a pair of hosts.
- ✓ The idea is to keep a running average of the RTT and then to compute the timeout as a function of this RTT. Specifically, every time TCP sends a data segment, it records the time.
- ✓ When an ACK for that segment arrives, TCP reads the time again, and then takes the difference between these two times as a SampleRTT.
- ✓ TCP then computes an EstimatedRTT as a weighted average between the previous estimate and this new sample.

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1-\alpha) \times \text{SampleRTT}$$

- The parameter α is selected to *smooth* the EstimatedRTT.
- A small α tracks changes in the RTT but is perhaps too heavily influenced by temporary fluctuations.

✓ EstimatedRTT to compute the timeout in a rather conservative way:

$$\text{TimeOut} = 2 \times \text{EstimatedRTT}$$

Karn/Partridge Algorithm

❖ Whenever a segment is retransmitted and then an ACK arrives at the sender, it is impossible to determine if this ACK should be associated with the first or the second transmission of the segment for the purpose of measuring the sample RTT.

❖ As illustrated in Figure 4.19, if you assume that the ACK is for the original transmission but it was really for the second, then the SampleRTT is too large (a); if you assume that the ACK is for the second transmission but it was actually for the first, then the SampleRTT is too small (b).

❖ The solution, which was proposed in 1987, is surprisingly simple.

❖ Whenever TCP retransmits a segment, it stops taking samples of the RTT; it only measures SampleRTT for segments that have been sent only once.

❖ Each time TCP retransmits, it sets the next timeout to be twice the last timeout, rather than basing it on the last EstimatedRTT.

❖ Karn and Partridge proposed that TCP use exponential backoff, similar to what the Ethernet does.

❖ The motivation for using exponential backoff is simple: Congestion is the most likely cause of lost segments, meaning that the TCP source should not react too aggressively to a timeout.

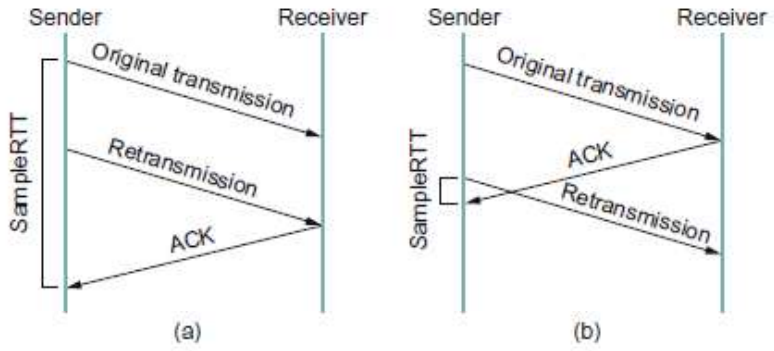


Figure 4.19 Associating the ACK with (a) original transmission versus (b) retransmission

Jacobson/Karels Algorithm

- The Karn/Partridge algorithm was introduced at a time when the Internetwas suffering from high levels of network congestion.
- Their approach was designed to fix some of the causes of that congestion, but, although it was an improvement, the congestion was not eliminated.
- The main problem with the original computation is that it does not take the variance of the sample RTTs into account. Intuitively, if the variation among samples is small, then the EstimatedRTT can be better trusted and there is no reason for multiplying this estimate by 2 to compute the timeout.
- On the other hand, a large variance in the samples suggests that the timeout value should not be too tightly coupled to the EstimatedRTT.
- In the new approach, the sender measures a new SampleRTT as before.
- It then folds this new sample into the timeout calculation as follows:

```
Difference = SampleRTT−EstimatedRTT
EstimatedRTT = EstimatedRTT+(δ×Difference)
Deviation = Deviation+δ(|Difference|−Deviation)
```

  - where δ is a fraction between 0 and 1. That is, we calculate both the mean RTT and the variation in that mean.
- TCP then computes the timeout value as a function of both Estimated-RTT and Deviation as follows:

```
TimeOut = μ×EstimatedRTT+ϕ×Deviation
```

  - where based on experience, μ is typically set to 1 and ϕ is set to 4.
  - when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the Deviation termto dominate the calculation.

Implementation

- There are two items of note regarding the implementation of timeouts in TCP.
  i. The first is that it is possible to implement the calculation for EstimatedRTT and Deviation without using floating-point arithmetic.
     - The whole calculation is scaled by 2n, with ϕ selected to be 1/2n. This allows us to do integer arithmetic, implementing multiplication and division using shifts, thereby achieving higher performance.
     - The resulting calculation is given by the following code fragment, where n = 3 (i.e., _ = 1/8).

```
{
        SampleRTT -= (EstimatedRTT >> 3);
        EstimatedRTT += SampleRTT;
        if (SampleRTT < 0)
        SampleRTT = -SampleRTT;
        SampleRTT -= (Deviation >> 3);
        Deviation += SampleRTT;
        TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);
}
```

ii. The second point of note is that the Jacobson/Karels algorithm is only as good as the clock used to read the current time.

4.3.5. Error Control

- TCP is a reliable transport layer protocol. This means that an application program that delivers a stream of data to TCP relies on TCP to deliver the entire stream to the application program on the other end in order, without error, and without any part lost or duplicated.
- TCP provides reliability using error control. Error control includes mechanisms for detecting corrupted segments, lost segments, out-of-order segments, and duplicated segments. Error control also includes a mechanism for correcting errors after they are detected.
- Error detection and correction in TCP is achieved through the use of three simple tools:

ii. *Checksum*
- Each segment includes a checksum field which is used to check for a corrupted segment.
- If the segment is corrupted, it is discarded by the destination TCP and is considered as lost.
- TCP uses a 16-bit checksum that is mandatory in every segment.

iii. *Acknowledgment*
- TCP uses acknowledgments to confirm the receipt of data segments.
- Control segments that carry no data but consume a sequence number are also acknowledged.
- ACK segments are never acknowledged.
- ACK segments do not consume sequence numbers and are not acknowledged.

iv. *Retransmission*
- The heart of the error control mechanism is the retransmission of segments.
- When a segment is corrupted, lost, or delayed, it is retransmitted.
- In modern implementations, a segment is retransmitted on two occasions:
  - when a retransmission timer expires
  - when the sender receives three duplicate ACKs.
- RTI is the time needed for a segment to reach a destination and for an acknowledgment to be received.
- It uses a back-off strategy
- *Retransmission After RTO*
  - When the timer matures, the earliest outstanding segment is retransmitted even though lack of a received ACK can be due to a delayed segment, a delayed ACK, or a lost acknowledgment.
  - No time-out timer is set for a segment that carries only an acknowledgment, which means that no such segment is resent.
  - The value of RTO is dynamic in TCP and is updated based on the round-trip time (RTT) of segments.
  - An RTI is the time needed for a segment to reach a destination and for an acknowledgment to be received. It uses a back-off strategy.
- *Retransmission After Three Duplicate ACK Segments*
  - The previous rule about retransmission of a segment is sufficient if the value of RTO is not very large.

- Sometimes, one segment is lost and the receiver receives so many out-of-order segments that they cannot be saved (limited buffer size). To alleviate this situation, the three-duplicate-ACKs rule and retransmit the missing segment immediately.
- This feature is referred to as fast retransmission

- *Out-of--Order Segments*
  - When a segment is delayed, lost, or discarded, the segments following that segment arrive out of order.
  - TCP was designed to discard all out-of-order segments, resulting in the retransmission of the missing segment and the following segments.

4.3.5.1. Some Scenarios

Normal Operation

The first scenario shows bidirectional data transfer between two systems, as in Figure 4.30. The client TCP sends one segment; the server TCP sends three. The figure shows which rule applies to each acknowledgment.

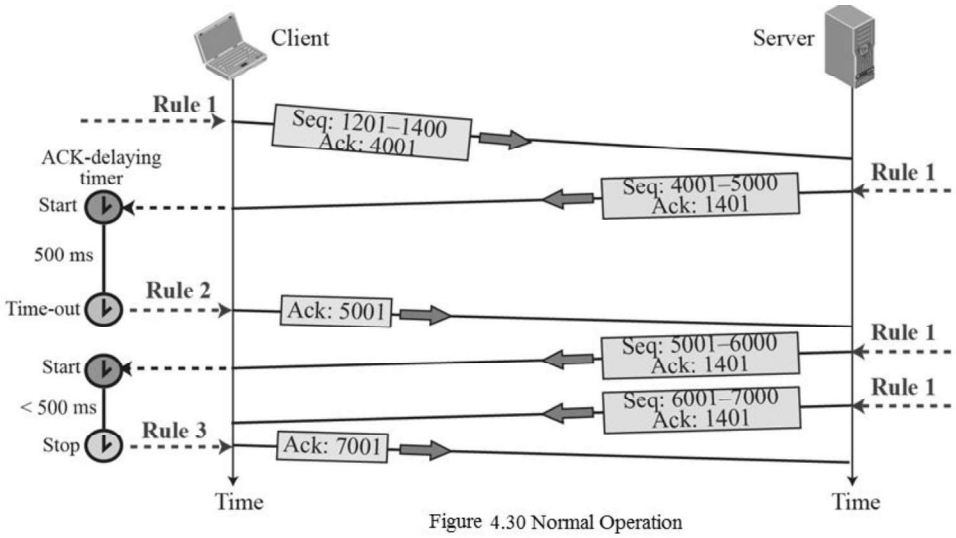


Figure 4.30 Normal Operation

There are data to be sent, so the segment displays the next byte expected. When the client receives the first segment from the server, it does not have any more data to send; it sends only an ACK segment.

However, the acknowledgment needs to be delayed for 500 ms to see if any more segments arrive. When the timer matures, it triggers an acknowledgment. This is so because the client has no knowledge if other segments are coming; it cannot delay the acknowledgment forever. When the next segment arrives, another acknowledgment timer is set. However, before it matures, the third segment arrives. The arrival of the third segment triggers another acknowledgment.

Lost Segment

In this scenario, we show what happens when a segment is lost or corrupted. A lost segment and a corrupted segment are treated the same way by the receiver. A lost segment is discarded somewhere in the network; a corrupted segment is discarded by the receiver itself. Both are considered lost. Figure 4.31 shows a situation in which a segment is lost and discarded by some router in the network, perhaps due to congestion.

Here data transfer is unidirectional: one site is sending, the other is receiving. In our scenario, the sender sends segments 1 and 2, which are acknowledged immediately by an ACK. Segment 3, however, is lost. The receiver receives segment 4, which is out of order. The receiver stores the data in the segment in its buffer but leaves a gap to indicate that there is no continuity in the data. The receiver immediately sends an acknowledgment to the sender,

displaying the next byte it expects. Note that the receiver stores bytes 801 to 900, but never delivers these bytes to the application until the gap is filled.

The receiver TCP delivers only ordered data to the process.

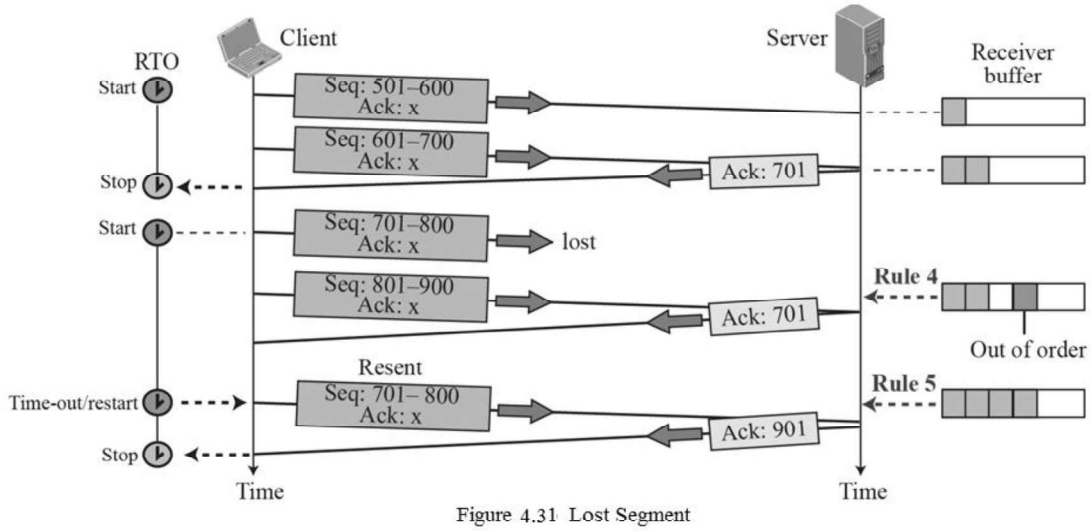


Figure 4.31 Lost Segment

Fast Retransmission

- Our scenario is the same as the second except that the RTO has a higher value
- When the receiver receives the fourth, fifth, and sixth segments, it triggers an acknowledgment. The sender receives four acknowledgments with the same value (three duplicates). Although the timer for segment 3 has not matured yet, the fast transmission requires that segment 3, the segment that is expected by all these acknowledgments, be resent immediately.
- only one segment is retransmitted although four segments are not acknowledged. When the sender receives the

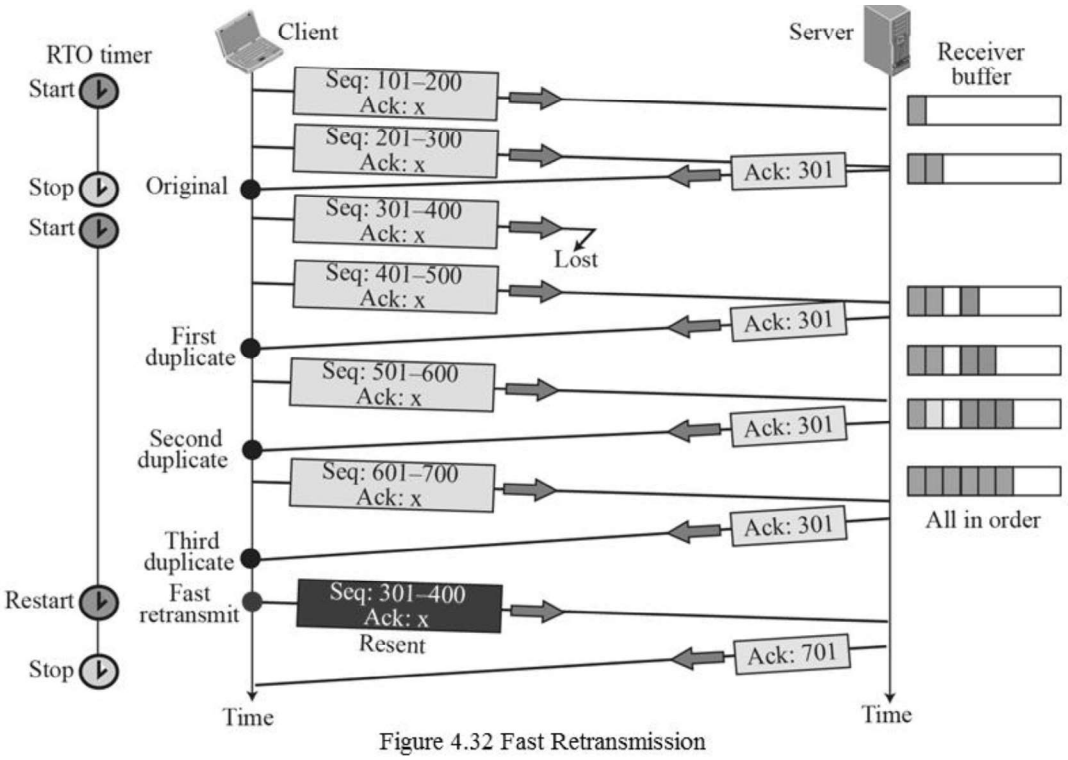


Figure 4.32 Fast Retransmission

retransmitted ACK, it knows that the four segments are safe and sound because acknowledgment is cumulative.

4.3.6. TCP Congestion control

DATA TRAFFIC

- The main focus of congestion control and quality of service is data traffic.
- In congestion control we try to avoid traffic congestion. In quality of service, we try to create an appropriate environment for the traffic. So, before talking about congestion control and quality of service,

Traffic Descriptor

Traffic descriptors are qualitative values that represent a data flow. Figure 4.33 shows a traffic flow with some of these values.

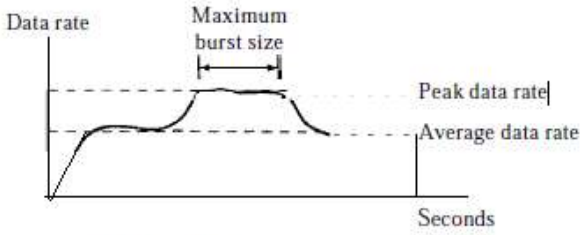


Figure 4.33 *Traffic descriptors*

Average Data Rate The average data rate is the number of bits sent during a period of time, divided by the number of seconds in that period.

Average data rate =amount of data/time

The average data rate is a very useful characteristic of traffic because it indicates the average bandwidth needed by the traffic.

Peak Data Rate The peak data rate defines the maximum data rate of the traffic. In Figure 4.33 it is the maximum *y* axis value. The peak data rate is a very important measurement because it indicates the peak bandwidth that the network needs for traffic to pass through without changing its data flow.

Maximum Burst Size Although the peak data rate is a critical value for the network, it can usually be ignored if the duration of the peak value is very short. For example, if data are flowing steadily at the rate of 1 Mbps with a sudden peak data rate of 2 Mbps for just 1 ms, the network probably can handle the situation. However, if the peak data rate lasts 60 ms, there may

be a problem for the network. The maximum burst size normally refers to the maximum length of time the traffic is generated at the peak rate.

Effective Bandwidth The effective bandwidth is the bandwidth that the network needs to allocate for the flow of traffic. The effective bandwidth is a function of three values: average data rate, peak data rate, and maximum burst size. The calculation of this value is very complex.

Traffic Profiles

For our purposes, a data flow can have one of the following traffic profiles: constant bit rate, variable bit rate, or bursty as shown in Figure 4.34.

Constant Bit Rate A constant-bit-rate (CBR), or a fixed-rate, traffic model has a data rate that does not change. In this type of flow, the average data rate and the peak data rate are the same.

The maximum burst size is not applicable. This type of traffic is very easy for a network to handle since it is predictable. The network knows in advance how much bandwidth to allocate for this type of flow.

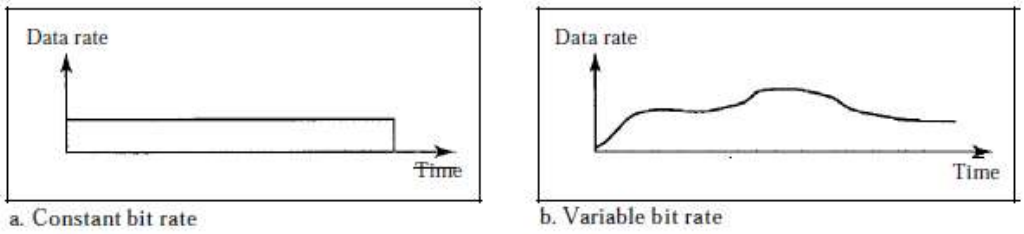
Variable Bit Rate In the variable-bit-rate (VBR) category, the rate of the data flow changes in time, with the changes smooth instead of sudden and sharp. In this type of flow, the average data

Bursty In the **bursty data** category, the data rate changes suddenly in a very short time. It may jump from zero, for example, to 1 Mbps in a few microseconds and vice versa. It may also remain at this value for a while.

The average bit rate and the peak bit rate are very different values in this type of flow.

The maximum burst size is significant.

Bursty traffic is one of the main causes of congestion in a network.



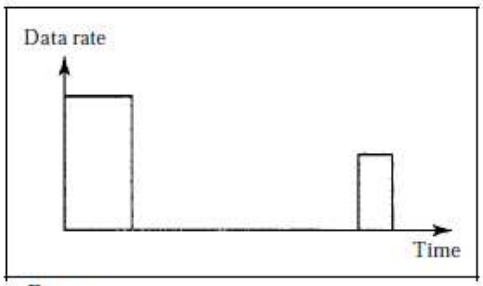


Figure 4.34 *Three traffic profiles*

CONGESTION

- An important issue in a packet-switched network is **congestion.**
- Congestion in a network may occur if the **load** on the network-the number of packets sent to the network is greater than the *capacity* of the network-the number of packets a network can handle.
- ***Congestion control*** refers to the mechanisms and techniques to control the congestion and keep the load below the capacity.
- Congestion happens in any system that involves waiting. For example, congestion happens on a freeway because any abnormality in the flow, such as an accident during rush hour, creates blockage.
- Congestion in a network or internetwork occurs because routers and switches have queues-buffers that hold the packets before and after processing.
- A router, for example, has an input queue and an output queue for each interface. When a packet arrives at theincoming interface, it undergoes three steps before departing, as shown in Figure 4.35.

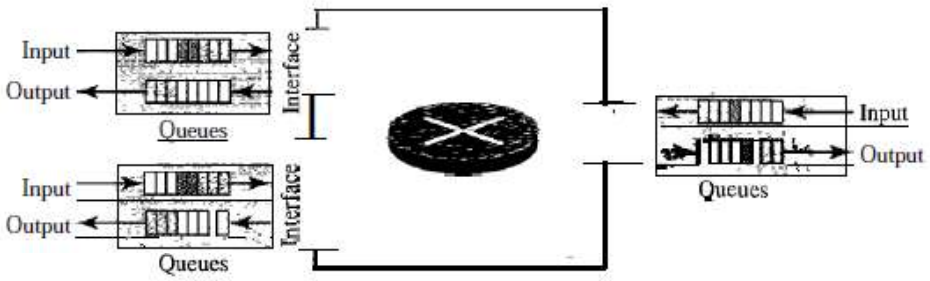
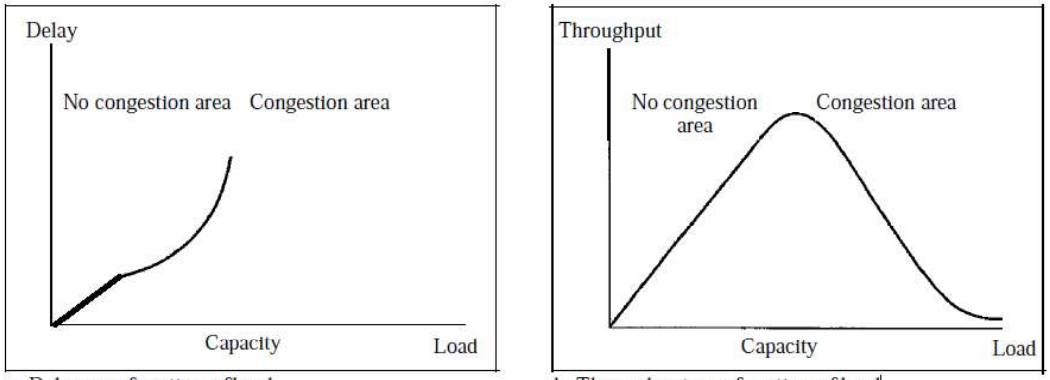


Figure 4.35 *Queues in a router*

1. The packet is put at the end of the input queue while waiting to be checked.
2. The processing module of the router removes the packet from the input queue once it reaches the front of the queue and uses its routing table and the destination address to find the route.
3. The packet is put in the appropriate output queue and waits its turn to be sent.

Network Performance

Congestion control involves two factors that measure the performance of a network: *delay* and *throughput.* Figure 4.36 shows these two performance measures as function of load.



a. Delay as a function of load

b. Throughput as a function of load

Figure 4.36 *Packet delay and throughput as functions of load*

Delay Versus Load

- when the load is much less than the capacity of the network, the delay is at a minimum.
- When the load reaches the network capacity, the delay increases sharply because we now need to add the waiting time in the queues (for all routers in the path) to the total delay.
- The delay becomes infinite when the load is greater than the capacity. If this is not obvious, consider the size of the queues when almost no packet reaches the destination, or reaches the destination with infinite delay; the queues become longer and longer.
- When a packet is delayed, the source, not receiving the acknowledgment, retransmits the packet, which makes the delay, and the congestion, worse.

Throughput Versus Load

- Define throughput in a network as the number of packets passing through the network in a unit of time. Notice that when the load is below the capacity of the network, the throughput increases proportionally with the *load.*
- When the load exceeds the capacity, the queues become full and the routers have to discard some packets. Discarding packet does not reduce the number of packets in the network because the sources retransmit the packets, using time-out mechanisms, when the packets do not reach the destinations.

CONGESTION CONTROL

- Congestion control refers to techniques and mechanisms that can either prevent congestion, before it happens, or remove

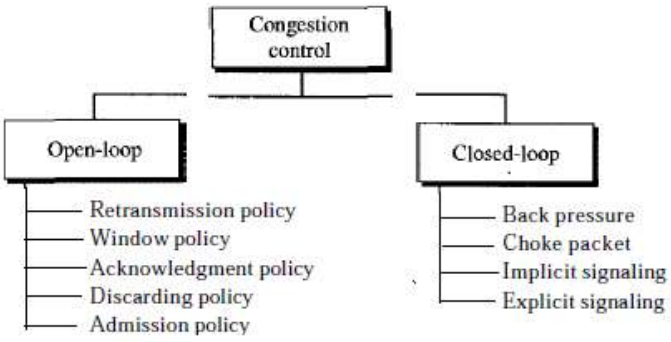


Figure 4.37 *Congestion control categories*

congestion, after it has happened.

- Divide congestion control mechanisms into two broad categories: open-loop congestion control (prevention) and closed-loop congestion control (removal) as shown in Figure 4.37.

Open-Loop Congestion Control

In open-loop congestion control, policies are applied to prevent congestion before it happens. In these mechanisms, congestion control is handled by either the source or the destination.

i. Retransmission Policy

- If the sender feels that a sent packet is lost or corrupted, the packet needs to be retransmitted.
- Retransmission in general may increase congestion in the network.
- The retransmission policy and the retransmission timers must be designed to optimize efficiency and at the same time prevent congestion.
- For example, the retransmission policy used by TCP (explained later) is designed to prevent or alleviate congestion.

ii. Window Policy

- The type of window at the sender may also affect congestion.
- The Selective Repeat window is better than the Go-Back-N window for congestion control.
- In the *Go-Back-N* window, when the timer for a packet times out, several packets may be resent, although some may have arrived safe and sound at the receiver.
- The Selective Repeat window, on the other hand, tries to send the specific packets that have been lost or corrupted.

iii. Acknowledgment Policy

- The acknowledgment policy imposed by the receiver may also affect congestion.
- If the receiver does not acknowledge every packet it receives, it may slow down the sender and help prevent congestion.
- A receiver may send an acknowledgment only if it has a packet to be sent or a special timer expires.
- A receiver may decide to acknowledge only *N* packets at a time.
- The acknowledgments are also part of the load in a network. Sending fewer acknowledgments means imposing fewer loads on the network.

iv. Discarding Policy

- A good discarding policy by the routers may prevent congestion and at the same time may not harm the integrity of the transmission.
- For example, in audio transmission, if the policy is to discard less sensitive packets when congestion is likely to happen, the quality of sound is still preserved and congestion is prevented or alleviated.

v. Admission Policy

- An admission policy, which is a quality-of-service mechanism, can also prevent congestion in virtual-circuit networks.
- Switches in a flow first check the resource requirement of a flow before admitting it to the network.
- A router can deny establishing a virtual circuit connection if there is congestion in the network or if there is a possibility of future congestion.

Closed-Loop Congestion Control

Closed-loop congestion control mechanisms try to alleviate congestion after it happens.

i. Backpressure

- The technique of *backpressure* refers to a congestion control mechanism in which a congested node stops receiving data from the immediate upstream node or nodes. This may cause the upstream node or nodes to become congested, and they, in turn, reject data from their upstream nodes or nodes. And so on.
- Backpressure is a node-to-node congestion control that starts with a node and propagates, in the opposite direction of data flow, to the source.
- The backpressure technique can be applied only to virtual circuit networks, in which each node knows the upstream node from which a flow of data is corning. Figure 4.38 shows the idea of backpressure.
- Node III in the figure has more input data than it can handle. It drops some packets in its input buffer and informs node II to slow down. Node II, in turn, may be congested because it is slowing down the output flow of data.
- If node II is congested, it informs node I to slow down, which in turn may create congestion. If so, node I inform the source of data to slow down. This, in time, alleviates the congestion.

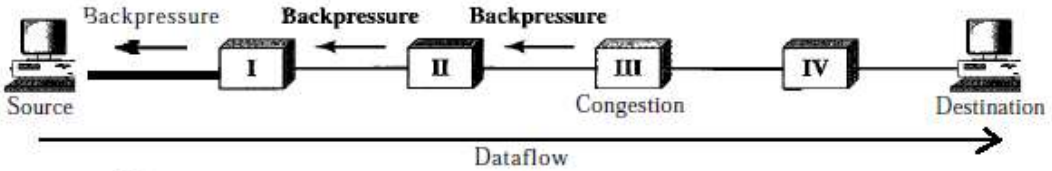


Figure 4.38 *Backpressure method for alleviating congestion*

- The *pressure* on node III is moved backward to the source to remove the congestion.

ii. Choke Packet

- A choke packet is a packet sent by a node to the source to inform it of congestion.
- Note the difference between the backpressure and choke packet methods. In backpressure, the warning is from one node to its upstream node, although the warning may eventually reach the source station. In the choke packet method, the warning is from the router, which has encountered congestion, to the source station directly.
- The intermediate nodes through which the packet has traveled are not warned.
- An example of this type of control in ICMP.
- When a router in the Internet is overhead with IP datagrams, it may discard some of them; but it informs the source host, using a source quench ICMP message. The warning message goes directly to the source station; the intermediate routers, and does not take any action. Figure 4.39 shows the idea of a choke packet.

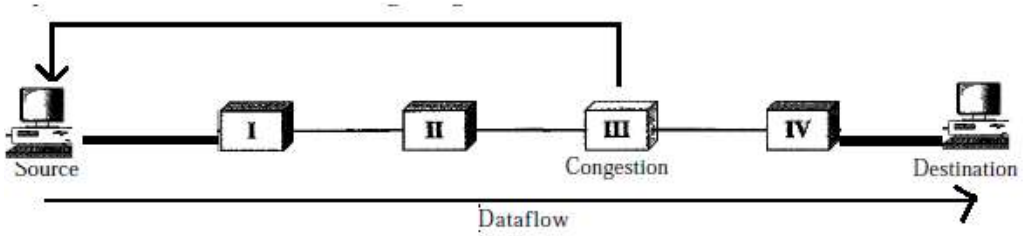


Figure 4.39 *Choke packet*

iii. Implicit Signaling

- In implicit signaling, there is no communication between the congested node or nodes and the source.

- The source guesses that there is a congestion somewhere in the network from other symptoms. For example, when a source sends several packets and there is no acknowledgment for a while, one assumption is that the network is congested.
- The delay in receiving an acknowledgment is interpreted as congestion in the network; the source should slow down.

iv. Explicit Signaling

- The node that experiences congestion can explicitly send a signal to the source or destination.
- The explicit signaling method, however, is different from the choke packet method. In the choke packet method, a separate packet is used for this purpose; in the explicit signaling method, the signal is included in the packets that carry data.
- Example Frame Relay congestion control, can occur in either the forward or the backward direction.
- *Backward Signaling* A bit can be set in a packet moving in the direction opposite to the congestion. This bit can warn the source that there is congestion and that it needs to slow down to avoid the discarding of packets.
- *Forward Signaling* A bit can be set in a packet moving in the direction of the congestion. This bit can warn the destination that there is congestion. The receiver in this case can use policies, such as slowing down the acknowledgments, to alleviate the congestion.

TWO EXAMPLES

1. Congestion Control in TCP

It show how TCP uses congestion control to avoid congestion or alleviate congestion in the network.

Congestion Window

- The sender window size is determined by the available buffer space in the receiver *(rwnd).*
- If the network cannot deliver the data as fast as they are created by the sender, it must tell the sender to slow down.
- The sender's window size is determined not only by the receiver but also by congestion in the network.
- The sender has two pieces of information: the receiver-advertised window size and the congestion window size. The actual size of the window is the minimum of these two.

```
Actual window size= minimum (rwnd, cwnd)
```

Congestion Policy

- TCP's general policy for handling congestion is based on three phases: slow start, congestion avoidance, and congestion detection.
- In the slow-start phase, the sender starts with a very slow rate of transmission, but increases the rate rapidly to reach a threshold.
- When the threshold is reached, the data rate is reduced to avoid congestion. Finally if congestion is detected, the sender goes back to the slow-start or congestion avoidance phase based on how the congestion is detected.
- Figure 4.40, summarize the congestion policy of TCP and the relationships between the three phases.

Slow Start:

- Exponential Increase One of the algorithms used in TCP congestion control is called slow start. This algorithm is based on the idea that the size of the congestion window *(cwnd)* starts with one maximum segment size (MSS).
- The MSS is determined during connection establishment by using an option of the same name. The size of the window increases one MSS each time an acknowledgment is received.

- As the name implies, the window starts slowly, but grows exponentially. To show the idea, let us look at Figure 4.41.
- Assumed that *rwnd* is much higher than *cwnd,* so that the sender window size always equals *cwnd.* We have assumed that each segment is acknowledged individually.

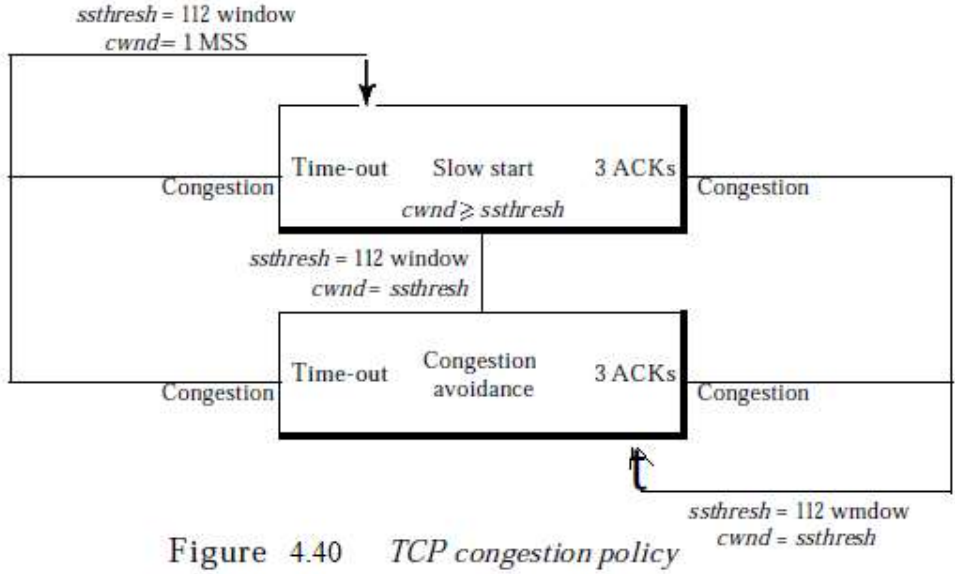


Figure 4.40 *TCP congestion policy*

- The sender starts with *cwnd* =1 MSS. This means that the sender can send only one segment. After receipt of the acknowledgment for segment 1, the size of the congestion window is increased by 1, which means that *cwnd* is now 2. Now two more segments can be sent. When each acknowledgment is received, the size of the window is increased by 1 MSS. When all seven segments are acknowledged, *cwnd* = 8.
- If we look at the size of *cwnd* in terms of rounds (acknowledgment of the wholewindow of segments), we find that the rate is exponential as shown below:

Start	➡	$cwnd=1$
After round 1	➡	$cwnd=2^1=2$
After round 2	➡	$cwnd=2^2=4$
After round 3	➡	$cwnd=2^3=8$

- We need to mention that if there is delayed ACKs, the increase in the size of the window is less than power of 2.
- Slow start cannot continue indefinitely. There must be a threshold to stop this phase. The sender keeps track of a variable named *ssthresh* (slow-start threshold).
- When the size of window in bytes reaches this threshold, slow start stops and the next phase starts. In most implementations the value of *ssthresh* is 65,535 bytes.
- In the slow-start algorithm, the size of the congestion window increases exponentially until it reaches a threshold.
- *Slow start algorithm is as follows*

```
If   cwnd <= ssthresh then
          Each time an Ack is received:
          cwnd = cwnd + MSS
else      /*  cwnd >  ssthresh  */
```

Each time an Ack is received :

cwnd = cwnd + MSS. MSS / cwnd

endif

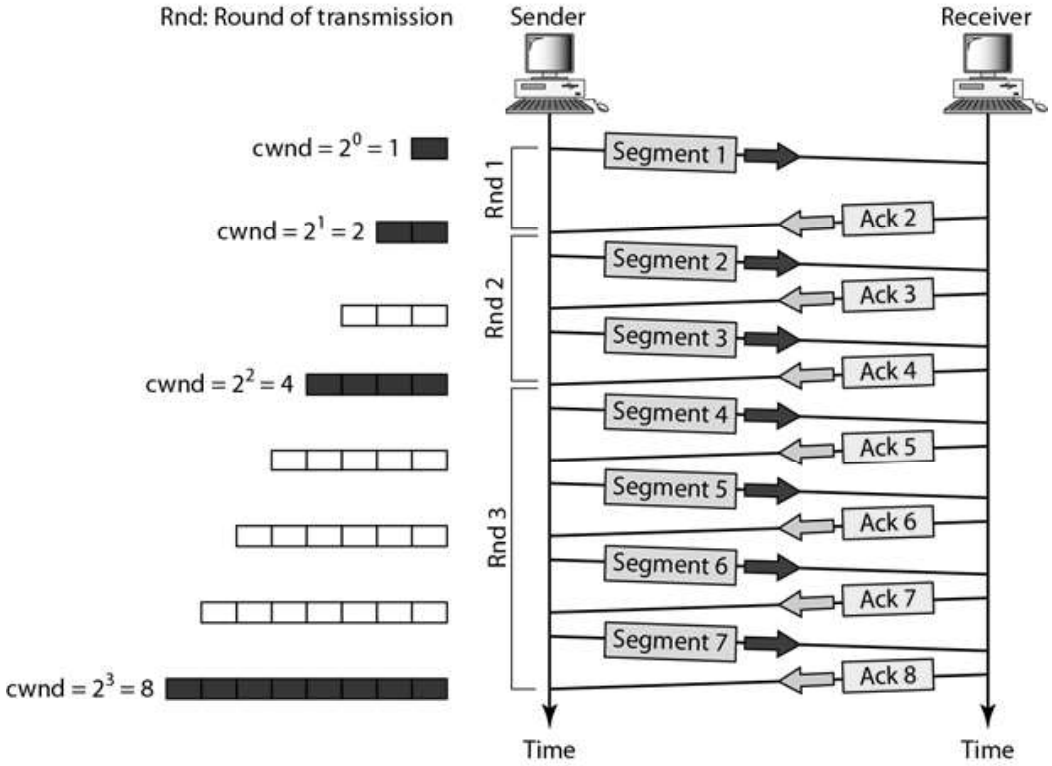


Figure 4.41 Slow Start, Exponentialy Increase

Congestion Avoidance

- Additive Increase If we start with the slow-start algorithm, the size of the congestion window increases exponentially.
- To avoid congestion before it happens, one must slow down this exponential growth. TCP defines another algorithm called congestion avoidance, which undergoes an additive increase instead of an exponential one.
- When the size of the congestion window reaches the slow-start threshold, the slow-start phase stops and the additive phase begins. In this algorithm, each time the whole window of segments is acknowledged (one round), the size of the congestion window is increased by 1.
- To show the idea, we apply this algorithm to the same scenario as slow start, although we will see that the congestion avoidance algorithm usually starts when the size of the window is much greater than 1.
- Figure 4.35 shows the idea.
- In this case, after the sender has received acknowledgments for a complete window size of segments, the size of the window is increased by one segment.
- If we look at the size of *cwnd* in terms of rounds, we find that the rate is additive as shown below:

Start	➡	*cwnd=1*
After round 1	➡	*cwnd*= 1 + 1 = 2
After round 2	➡	*cwnd=2*+ 1 = 3
After round 3	➡	*cwnd=3*+ 1 = 4

- In the congestion avoidance algorithm, the size of the congestion window increases additively until congestion is detected.

Congestion Detection

- Multiplicative Decrease If congestion occurs, the congestion window size must be decreased. The only way the sender can guess that congestion has occurred is by the need to retransmit a segment.
- Retransmission can occur in one of two cases: when a timer times out or when three ACKs are received. In both cases, the size of the threshold is dropped to one-half, a multiplicative decrease.

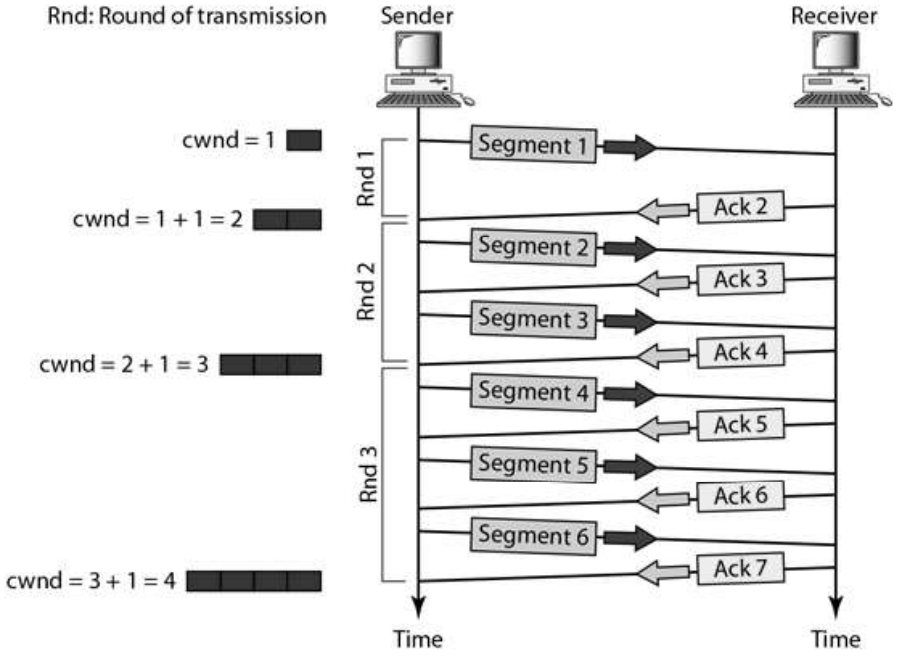


Figure 4.42 Congestion avoidance, additive increase

TCP implementations have two reactions:

1. If a time-out occurs, there is a stronger possibility of congestion; a segment has probably been dropped in the network, and there is no news about the sent segments.In this case TCP reacts strongly:
   a) It sets the value of the threshold to one-half of the current window size.
   b) It sets *cwnd* to the size of one segment.
   c) It starts the slow-start phase again.
2. If three ACKs are received, there is a weaker possibility of congestion; a segment may have been dropped, but some segments after that may have arrived safely since three ACKs are received. This is called fast transmission and fast recovery. In this case, TCP has a weaker reaction:
   a) It sets the value of the threshold to one-half of the current window size.
   b) It sets *cwnd* to the value of the threshold (some implementations add three segment sizes to the threshold).
   c) It starts the congestion avoidance phase.

- An implementations reacts to congestion detection in one of the following ways:
  - If detection is by time-out, a new *slow-start* phase starts.
  - If detection is by three ACKs, a new *congestion avoidance* phase starts.

- We give an example in Figure 4.42. We assume that the maximum window size is 32 segments. The threshold is set to 16 segments (one-half of the maximum window size).
- In the *slow-start* phase the window size starts from 1 and grows exponentially until it reaches the threshold. After it reaches the threshold, the *congestion avoidance (additive increase)* procedure allows the window size to increase linearly until a timeout occurs or the maximum window size is reached.
- In Figure 4.37, the time-out occurs when the window size is 20. At this moment, the *multiplicative decrease* procedure takes over and reduces the threshold to one-half of the previous window size.
- The previous window size was 20 when the time-out happened so the new threshold is now 10.
- TCP moves to slow start again and starts with a window size of 1, and TCP moves to additive increase when the new threshold is reached.
- When the window size is 12, a three-ACKs event happens. The multiplicative decrease procedure takes over again. The threshold is set to 6 and TCP goes to the additive increase phase this time. It remains in this phase until another time-out or another three ACKs happen.

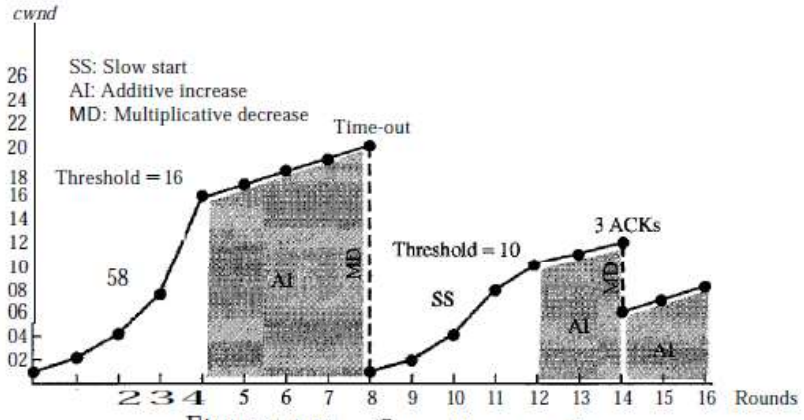


Figure 4.43 *Congestion example*

2. Congestion Control in Frame Relay

- Congestion in a Frame Relay network decreases throughput and increases delay. A high throughput and low delay are the main goals of the Frame Relay protocol.
- Frame Relay does not have flow control. In addition, Frame Relay allows the user to transmit bursty data. This means that a Frame Relay network has really congested with traffic, thus requiring congestion control.

Congestion Avoidance

- For congestion avoidance, the Frame Relay protocol uses 2 bits in the frame to explicitly warn the source and the destination of the presence of congestion.

BECN The backward explicit congestion notification (BECN)

- Bit warns the sender of congestion in the network. There are two methods: The switch can use response frames

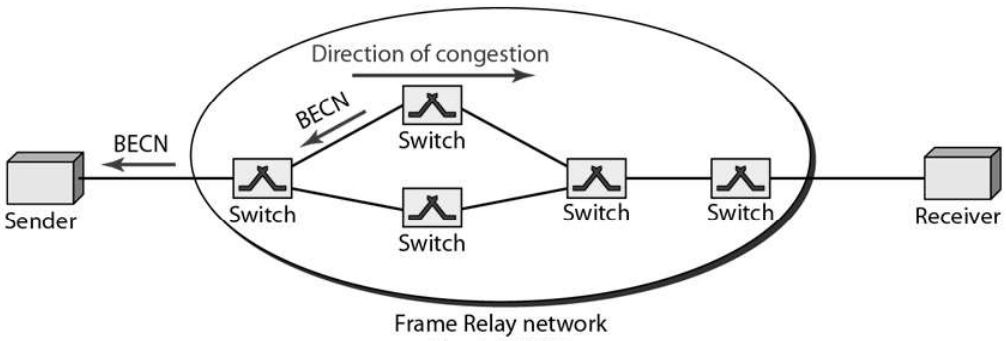


Figure 4.44 BECN

from the receiver (full-duplex mode), or else the switch can use a predefined connection (DLCI =1023) to send special frames for this specific purpose.

- The sender can respond to this warning by simply reducing the data rate. Figure 4.44 shows the use of BECN.

FECN The forward explicit congestion notification (FECN)

- Bit is used to warn the receiver of congestion in the network. It might appear that the receiver cannot do anything to relieve the congestion.
- The Frame Relay protocol assumes that the sender and receiver are communicating with each other and are using some type of flow control at a higher level.
- For example, if there is an acknowledgment mechanism at this higher level, the receiver can delay the acknowledgment, thus forcing the sender to slow down. Figure 4.45 shows the use of FECN.
- When two endpoints are communicating using a Frame Relay network, four situations may occur with regard to congestion. Figure 4.46 shows these four situations and the values of FECN and BECN.

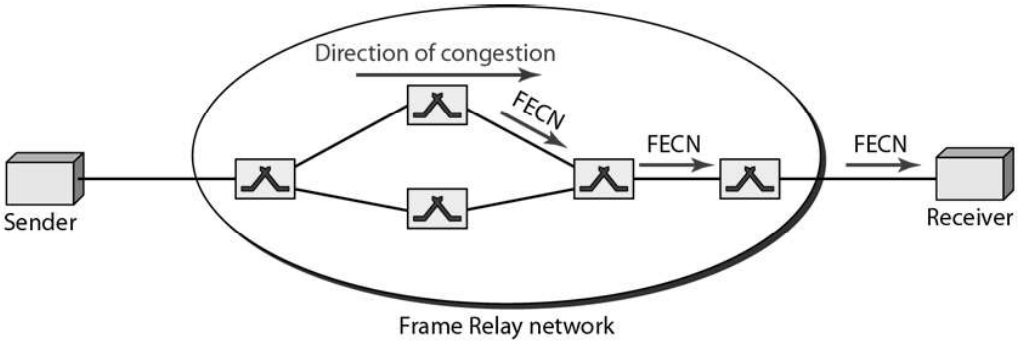


Figure 4.45 FECN

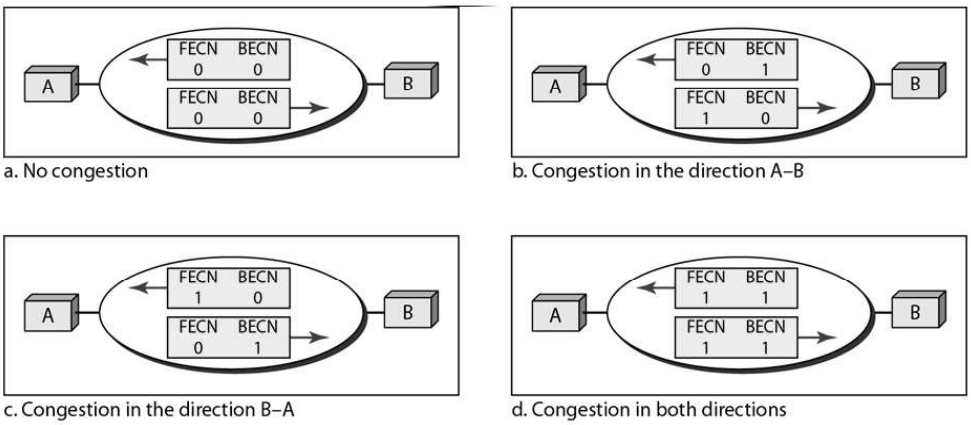


Figure 4.46 Four cases of congestion

4.3.7. Congestion avoidance (DECbit, RED)

4.3.7.1. DECbit

- The idea here is to more evenly split the responsibility for congestion control between the routers and the end nodes.
- Each router monitors the load it is experiencing and explicitly notifies the end nodes when congestion is about to occur.
- This notification is implemented by setting a binary congestion bit in the packets that flow through the router, hence the name *DECbit*.

- The destination host then copies this congestion bit into the ACK it sends back to the source. Finally, the source adjusts its sending rate so as to avoid congestion.
- The following discussion describes the algorithm in more detail, starting with what happens in the router.
  - ✓ A single congestion bit is added to the packet header.
  - ✓ A router sets this bit in a packet if its average queue length is greater than or equal to 1 at the time the packet arrives.
  - ✓ This average queue length is measured over a time interval that spans the last busy+idle cycle, plus the current busy cycle. (The router is *busy* when it is transmitting and *idle* when it is not.)
  - ✓ Figure 4.47 shows the queue length at a router as a function of time.
  - ✓ The router calculates the area under the curve and divides this value by the time interval to compute the average queue length.
  - ✓ Using a queue length of 1 as the trigger for setting the congestion bit is a trade-off between significant queuing (and hence higher throughput) and increased idle time (and hence lower delay). In other words, a queue length of 1 seems to optimize the power function.
  - ✓ The source maintains a congestion window, just as in TCP, and watches to see what fraction of the last window's worth of packets resulted in the bit being set.

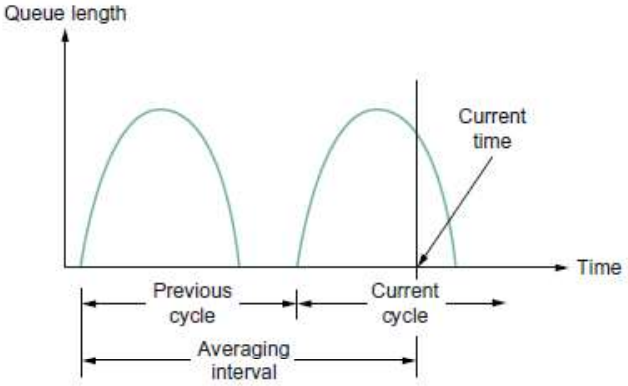


Figure 4.47 Computing average queue length at a router.

  - ✓ If less than 50% of the packets had the bit set, then the source increases its congestion window by one packet. If 50% or more of the last window's worth of packets had the congestion bit set, then the source decreases its congestion window to 0.875 times the previous value.
  - ✓ The value 50% was chosen as the threshold based on analysis that showed it to correspond to the peak of the power curve. The "increase by 1, decrease by 0.875" rule was selected because additive increase/multiplicative decrease makes the mechanism stable.

4.3.7.2. Random Early Detection (RED)

- Each router is programmed to monitor its own queue length and, when it detects that congestion is imminent, to notify the source to adjust its congestion window.
- RED, invented by Sally Floyd and Van Jacobson in the early 1990s, differs from the DECbit scheme in two major ways.
  - ✓ The first is that rather than explicitly sending a congestion notification message to the source, RED is most commonly implemented such that it *implicitly* notifies the source of congestion by dropping one of its packets.
    - o The source is, therefore, effectively notified by the subsequent timeout or duplicate ACK. As the "early" part of the RED acronym suggests, the gateway drops the packet earlier than it would have to, so as to notify thes source that it should decrease its congestion window sooner than it would normally have.

- In other words, the router drops a few packets before it has exhausted its buffer space completely, so as to cause the source to slow down, with the hope that this will mean it does not have to drop lots of packets later on.

✓ The second difference between RED and DECbit is in the details ofhow RED decides when to drop a packet and what packet it decides to drop.

- To understand the basic idea, consider a simple FIFO queue.
- Rather than wait for the queue to become completely full and then be forced to drop each arriving packet we could decide to drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*. This idea is called *early random drop*.

- The RED algorithm defines the details of how to monitor the queue length and when to drop a packet.
- Implementations are close to the algorithm that follows.
  - First, RED computes an average queue length using a weighted running average similar to the one used in the original TCP timeout computation. That is, AvgLen is computed as

```
AvgLen = (1−Weight)×AvgLen+Weight×SampleLen
```

- where 0 < Weight < 1 and SampleLen is the length of the queue
  - The queue length is measured every time a new packet arrives at the gateway.
  - In hardware, it might be calculated at some fixed sampling interval. The reason for using an average queue length rather than an instantaneous one is that it more accurately captures the notion of congestion.
  - If a queue is spending most of its time empty, then it's probably not appropriate to conclude that the router is congested and to tell the hosts to slow down.
  - Thus, the weighted running average calculation tries to detect long-lived congestion, as indicated in the right-hand portion of Figure 4.48, by filtering out short-term changes in the queue length.

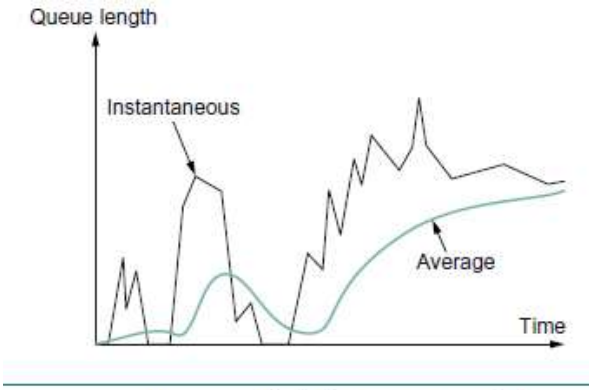


Figure 4.48 Weighted running average queue length.

  - Second, RED has two queue length thresholds that trigger certain activity: MinThreshold and MaxThreshold. When a packet arrives at the gateway, RED compares the current AvgLen with these two thresholds, according to the following rules:

```
if AvgLen < MinThreshold
      !queue the packet
if MinThreshold < AvgLen < MaxThreshold
      !calculate probability P
      !drop the arriving packet with probability P
if MaxThreshold ≤ AvgLen
      !drop the arriving packet
```

- If the average queue length is smaller than the lower threshold, no action is taken, and if the average queue length is larger than the upper threshold, then the packet is always dropped.
- If the average queue length is between the two thresholds, then the newly arriving packet is dropped with some probability P.
- This situation is depicted in Figure 4.49.
- The approximate relationship between P and AvgLen is shown in Figure 4.50.

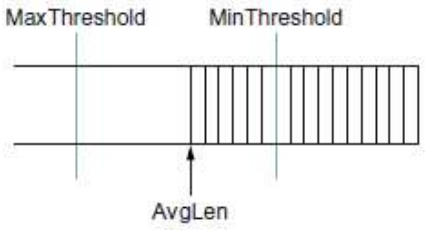


Figure 4.49 RED thresholds on a FIFO queue.

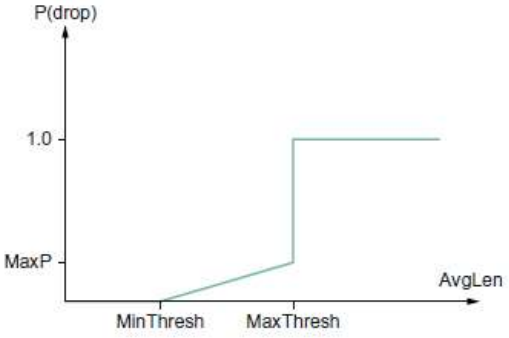


Figure 4.50 Drop probability function for RED.

- The probability of drop increases slowly when AvgLen is between the two thresholds, reaching MaxP at the upper threshold, at which point it jumps to unity.
- The rationale behind this is that, if AvgLen reaches the upper threshold, then the gentle approach (dropping a few packets) is not working and drastic measures are called for: dropping all arriving packets.
- Although Figure 6.50 shows the probability of drop as a function only of AvgLen, the situation is actually a little more complicated.

- P is a function of both AvgLen and how long it has been since the last packet was dropped. Specifically, it is computed as follows:

```
TempP                                                    =
MaxP×(AvgLen−MinThreshold)/(MaxThreshold−MinThreshold)
P = TempP/(1−count×TempP)
```

- TempP is the variable that is plotted on the y-axis in Figure 4.50,
- countkeeps track of how many newly arriving packets have been queued (not dropped)
- AvgLen has been between the two thresholds.
- P increases slowly as count increases, thereby making a drop increasingly likely as the time since the last drop increases.

- As an example, suppose that we set MaxP to 0.02 and count is initialized to zero. If the average queue length were halfway between the two thresholds, then TempP, and the initial value of P, would be half ofMaxP, or 0.01. An arriving packet, of course, has a 99 in 100 chance of getting into the queue at this point. With each successive packet that is not dropped, P slowly increases, and by the time 50 packets have arrived without a drop, P would have doubled to 0.02. In the unlikely event that 99 packets arrived without loss, P reaches 1, guaranteeing that the next packet is dropped.
- If RED drops a small percentage of packets when AvgLen exceeds MinThreshold, this will cause a few TCP connections to reduce their window sizes, which in turn will reduce the rate at which packets arrive at the router.

- All going well, AvgLen will then decrease and congestion is avoided.
- The queue length can be kept short, while throughput remains high since few packets are dropped.
- One of the goals of RED is to prevent tail drop behavior if possible.
- Consider the setting of the two thresholds, MinThreshold and Max-Threshold. If the traffic is fairly bursty, then MinThreshold should be sufficiently large to allowthe link utilization to be maintained at an acceptably high level. Also, the difference between the two thresholds should be larger than the typical increase in the calculated average queue length in one RTT.
- Setting MaxThreshold to twice MinThreshold seems to be a reasonable rule of thumb given the traffic mix on today's Internet.

4.4. QoS

Flow Characteristics

Four types of characteristics are attributed to a flow: reliability, delay, jitter, and bandwidth, as shown in Figure 4.51.

Reliability

- Reliability is a characteristic that a flow needs. Lack of reliability means losing a packet or acknowledgment, which entails retransmission.
- For example, it is more important that electronic mail, file transfer, and Internet access have reliable transmissions than telephony or audio conferencing.

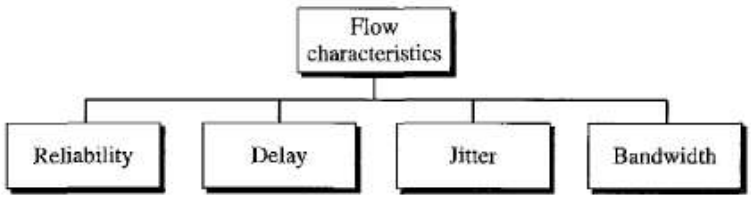


Figure 4.51 *Flow characteristics*

Delay

- Source-to-destination delay is another flow characteristic. Again applications can tolerate delay in different degrees.
- Applications: Telephony, audio conferencing, video conferencing, and remote log-in need minimum delay, while delay in file transfer or e-mail is less important.

Jitter

- Jitter is the variation in delay for packets belonging to the same flow. For example, if four packets depart at times 0, 1, 2, 3 and arrive at 20, 21, 22, 23, all have the same delay, 20 units of time. On the other hand, if the above four packets arrive at 21, 23, 21, and 28, they will have different delays: 21,22, 19, and 24.
- For applications such as audio and video, the first case is completely acceptable; the second case is not. For these applications, it does not matter if the packets arrive with a short or long delay as long as the delay is the same for all packets. For this application, the second case is not acceptable.
- High jitter means the difference between delays is large; low jitter means the variation is small.

Bandwidth

- Different applications need different bandwidths.
- In video conferencing we need to send millions of bits per second to refresh a color screen while the total number of bits in an e-mail may not reach even a million.

Flow Classes

- Based on the flow characteristics, we can classify flows into groups, with each group having similar levels of characteristics. This categorization is not formal or universal; some protocols such as ATM have defined classes.

TECHNIQUES TO IMPROVE QOS

four common methods to improve the QOS:

1. Scheduling
2. Traffic shaping
3. Admission control
4. Resource reservation.

Scheduling

- Packets from different flows arrive at a switch or router for processing.
- A good scheduling technique treats the different flows in a fair and appropriate manner. Several scheduling techniques are designed to improve the quality of service.
  1. FIFO queuing
  2. Priority queuing
  3. Weighted fair queuing.

FIFO Queuing

- In first-in, first-out (FIFO) queuing, packets wait in a buffer (queue) until the node (router or switch) is ready to process them.
- If the average arrival rate is higher than the average processing rate, the queue will fill up and new packets will be discarded.
- A FIFO queue is familiar to those who have had to wait for a bus at a bus stop.
- Figure 4.52 shows a conceptual view of a FIFO queue.

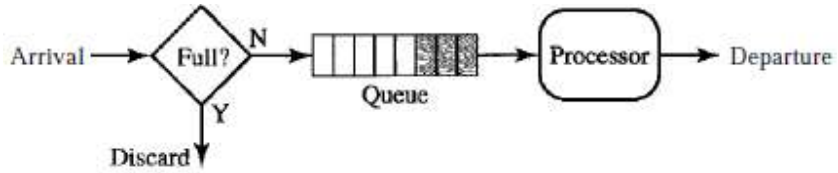


Figure 4.52 *FIFO queue*

Priority Queuing

- In priority queuing, packets are first assigned to a priority class. Each priority class has its own queue. The packets in the highest-priority queue are processed first. Packets in the lowest-priority queue are processed last.
- Figure 4.53 shows priority queuing with two priority levels

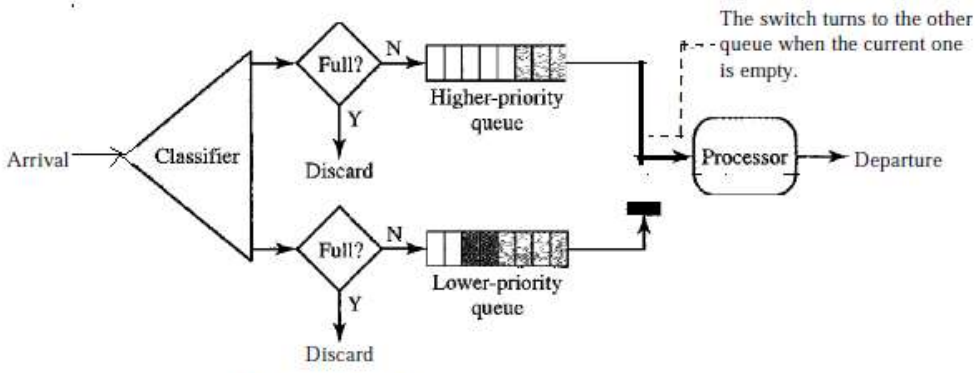


Figure 4.53 *Priority queuing*

- A priority queue can provide better QoS than the FIFO queue because higher priority traffic, such as multimedia, can reach the destination with less delay.
- If there is a continuous flow in a high-priority queue, the packets in the lower-priority queues will never have a chance to be processed. This is a condition called *starvation.*

Weighted Fair Queuing

- In this technique, the packets are still assigned to different classes and admitted to different queues.
- The queues are weighted based on the priority of the queues; higher priority means a higher weight.
- The system processes packets in each queue in a round-robin fashion with the number of packets selected from each queue based on the corresponding weight.
- For example, if the weights are 3, 2, and 1, three packets are processed from the first queue, two from the second queue, and one from the third queue. If the system does not impose priority on the classes, all weights can be equal.
- In this way, we have fair queuing with priority. Figure 4.54 shows the technique with three classes.

Traffic Shaping

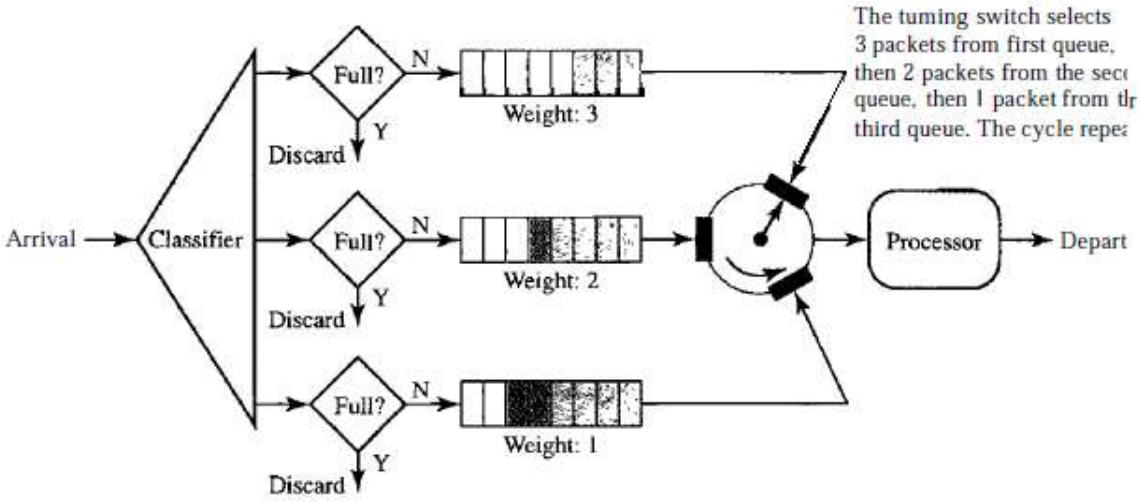


Figure 4.54 *Weighted fair queuing*

- Traffic shaping is a mechanism to control the amount and the rate of the traffic sent to the network. Two techniques can shape traffic: *leaky bucket and token bucket*.

Leaky Bucket

- If a bucket has a small hole at the bottom, the water leaks from the bucket at a constant rate as long as there is water in the bucket. The rate at which the water leaks does not depend on the rate at which the water is input to the bucket unless the bucket is empty.

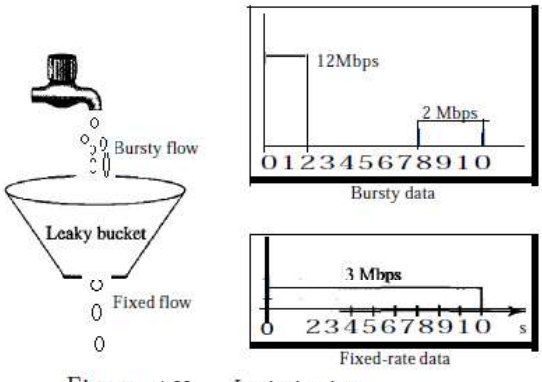


Figure 4.55 *Leaky bucket*

- The input rate can vary, but the output rate remains constant. Similarly, in networking, a technique called leaky bucket can smooth out bursty traffic.
- Bursty chunks are stored in the bucket and sent out at an average rate. Figure 4.55shows a leaky bucket and its effects.

Fixed-rate data

- In the figure, we assume that the network has committed a bandwidth of 3 Mbps for a host. The use of the leaky bucket shapes the input traffic to make it conform to this commitment.
- In Figure 4.56 the host sends a burst of data at a rate of 12 Mbps for 2 s, for a total of 24 Mbits of data. The host is silent for 5 s and then sends data at a rate of 2 Mbps for 3 s, for a total of 6 Mbits of data. In all, the host has sent 30 Mbits of data in 10s.

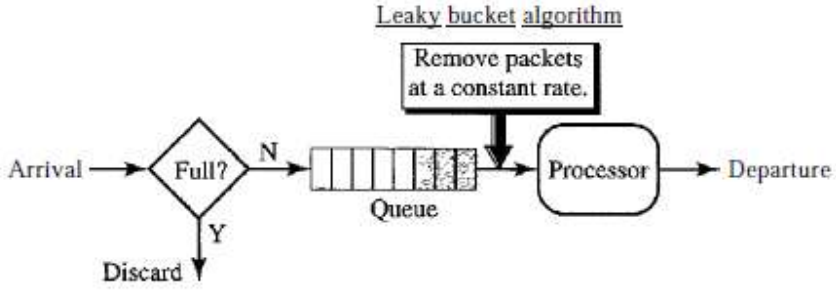


Figure 4.56 *Leaky bucket implementation*

- The leaky bucket smooths the traffic by sending out data at a rate of 3 Mbps during the same 10 s.
- Without the leaky bucket, the beginning burst may have hurt the network by consuming more bandwidth than is set aside for this host.
- A simple leaky bucket implementation is shown in Figure 4.47. A FIFO queue holds the packets. If the traffic consists of fixed-size packets (e.g., cells in ATM networks), the process removes a fixed number of packets from the queue at each tick of the clock.
- If the traffic consists of variable-length packets, the fixed output rate must be based on the number of bytes or bits.
- The following is an algorithm for variable-length packets:
  1. Initialize a counter to *n* at the tick of the clock.
  2. If *n* is greater than the size of the packet, send the packet and decrement the counter by the packet size. Repeat this step until *n* is smaller than the packet size.
  3. Reset the counter and go to step 1.
- A leaky bucket algorithm shapes bursty traffic into fixed-rate traffic by averaging the data rate. It may drop the packets if the bucket is full.

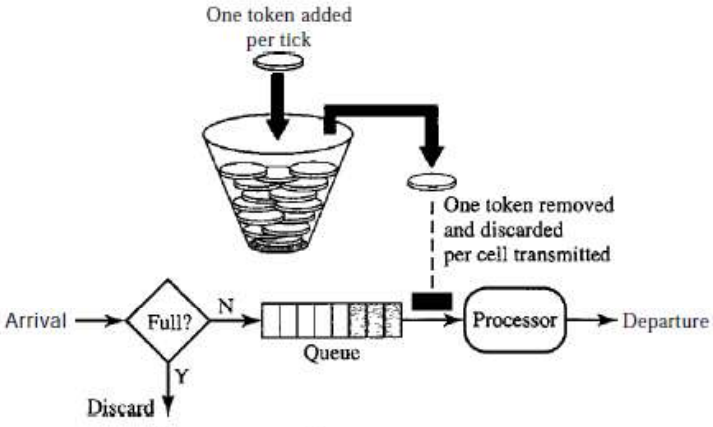


Figure 4.57 *Token bucket*

Token Bucket

- The leaky bucket is very restrictive. It does not credit an idle host. For example, if a host is not sending for a while, its bucket becomes empty.
- If the host has bursty data, the leaky bucket allows only an average rate. The time when the host was idle is not taken into account. On the other hand, the token bucket algorithm allows idle hosts to accumulate credit for the future in the form of tokens. For each tick of the clock, the system sends *n* tokens to the bucket.
- The system removes one token for every cell (or byte) of data sent. For example, if *n* is 100 and the host is idle for 100 ticks, the bucket collects 10,000 tokens. Now the host can consume all these tokens in one tick with 10,000 cells, or the host takes 1000 ticks with 10 cells per tick. In other words, the host can send bursty data as long as the bucket is not empty. Figure 4.57 shows the idea.
- The token bucket can easily be implemented with a counter. The token is initialized to zero. Each time a token is added, the counter is incremented by 1.
- Each time a unit of data is sent, the counter is decremented by 1. When the counter is zero, the host cannot send data.
- The token bucket allows bursty traffic at a regulated maximum rate.

Combining Token Bucket and Leaky Bucket

- The leaky bucket is applied after the token bucket; the rate of the leaky bucket needs to be higher than the rate of tokens dropped in the bucket.

Resource Reservation

- A flow of data needs resources such as a buffer, bandwidth, CPU time, and so on. The quality of service is improved if these resources are reserved beforehand.
- One QoS model called Integrated Services, which depends heavily on resource reservation to improve the quality of service.

Admission Control

- Admission control refers to the mechanism used by a router, or a switch, to accept or reject a flow based on predefined parameters called flow specifications.
- Before a router accepts a flow for processing, it checks the flow specifications to see if its capacity (in terms of bandwidth, buffer size, CPU speed, etc.) and its previous commitments to other flows can handle the new flow.

4.5. Application requirements

- There are two types of applications
  i. Real-time applications
  ii. Non-real-time applications. (*elastic applications*)
- These applications can work without guarantees of timely delivery of data.

Real-Time Audio Example

- As a concrete example of a real-time application, consider an audio application similar to the one illustrated in Figure 4.58.

Figure 4.58 An Audio Application

- Data is generated by collecting samples from a microphone and digitizing them using an analog-to-digital (A->D) converter.
- The digital samples are placed in packets, which are transmitted across the network and received at the other end.
- At the receiving host, the data must be *played back* at some appropriate rate.
- For example, if the voice sampleswere collected at a rate of one per 125 μs, they should be played back at the same rate.
- Each sample as having a particular *playback time*: the point in time at which it is needed in the receiving host.
- In the voice example, each sample has a playback time that is 125 μs later than the preceding sample.
- If data arrives after its appropriate playback time, either because it was delayed in the network or because it was dropped and subsequently retransmitted, it is essentially useless. It is the complete worthlessness of late data that characterizes real-time applications.
- In elastic applications, it might be nice if data turns up on time, but we can still use it when it does not.
- One way to make our voice application work would be to make sure that all samples take exactly the same amount of time to traverse the network. Then, since samples are injected at a rate of one per 125 μs, they will appear at the receiver at the same rate, ready to be played back.
- Packets encounter queues in switches or routers, and the lengths of these queues vary with time, meaning that the delays tend to vary with time and, as a consequence, are potentially different for each packet in the audio stream.
- The way to deal with this at the receiver end is to buffer up some amount of data in reserve, thereby always providing a store of packets waiting to be played back at the right time.
- If a packet is delayed a short time, it goes in the buffer until its playback time arrives. If it gets delayed a long time, then it will not need to be stored for very long in the receiver's buffer before being played back. Thus, we have effectively added a constant offset to the playback time of all packets as a form of insurance. This offset the *playback point.*
- The only time we run into trouble is if packets get delayed in the network for such a long time that they arrive after their playback time, causing the playback buffer to be drained.
- The operation of a playback buffer is illustrated in Figure 4.59
  - The left hand diagonal line shows packets being generated at a steady rate.
  - The wavy line shows when the packets arrive, some variable amount of time after theywere sent, depending on what they encountered

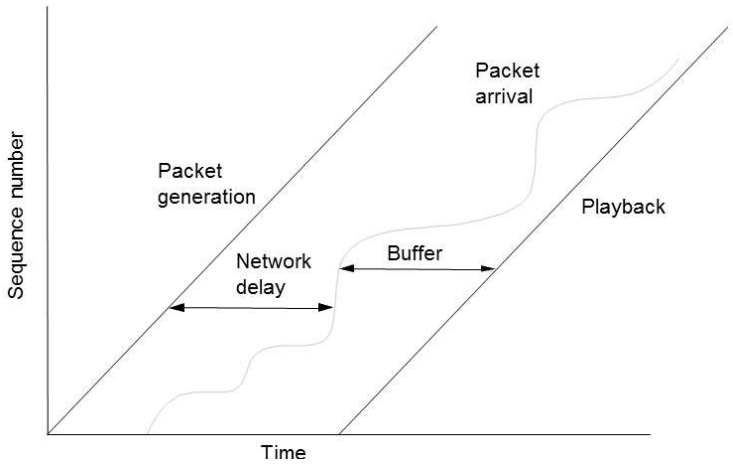


Figure 4.59 Playback Buffer

in the network.

- The right-hand diagonal line shows the packets being played back at a steady rate, after sitting in the playback buffer for some period of time.
- The playback line is far enough to the right in time, the variation in network delay is never noticed by the application.
- If we move the playback line a little to the left, then some packets will begin to arrive too late to be useful.

- if the time between when you speak and when your listener hears you is more than 300 ms.
- If data arrives early, we buffer it untilits correct playback time.
- If it arrives late, we have no use for it and must discard it.
- Figure 4.60 shows the one-way delay measured over a certain path across the Internet over the course of one particular day.
  - While the exact numbers would vary depending on the path and the date, the key factor here is the *variability* of the delay, which is consistently found on almost any path at any time.
  - As denoted by the cumulative percentages given across the top of the graph, 97% of the packets in this case had a latency of 100 ms or less. This means that if our example audio application were to set the playback point at 100 ms, then, on average, 3 out of every 100 packets would arrive too late to be of any use.

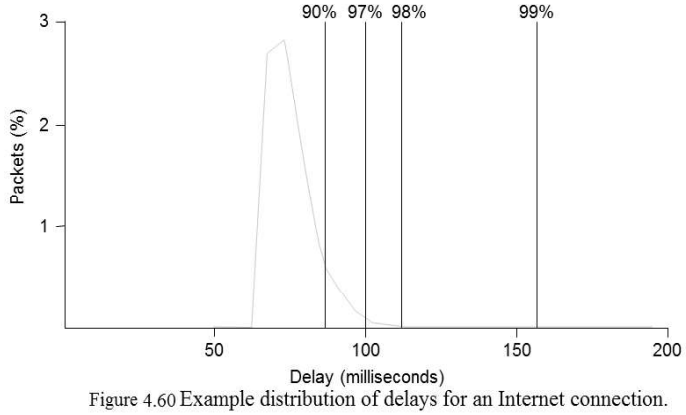


Figure 4.60 Example distribution of delays for an Internet connection.

Taxonomy of Real-Time Applications

- The following taxonomy owes much to the work of Clark, Braden, Shenker, and Zhang, whose papers on this subject
- The taxonomy of applications is summarized in Figure 4.61.
  - ✓ The first characteristic by which we can categorize applications is their tolerance of loss of data, where "loss" might occur because a packet arrived too late to be played back as well as arising from the usual causes in the network.
  - On the one hand, one lost audio sample can be interpolated from the surrounding samples with relatively little effect on the perceived audio quality. It is only as more and more samples are lost that quality declines to the point that the speech becomes incomprehensible.
  - On the other hand, a robot control program is likely to be an example of a real-time application that cannot tolerate loss—losing the packet that contains the command instructing the robot arm to stop is unacceptable.

- Thus, we can categorize real-time applications as *tolerant* or *intolerant* depending on whether they can tolerate occasional loss.

✓ A second way to characterize real-time applications is by their adaptability.

- For example, an audio application might be able to adapt to the amount of delay that packets experience as they traverse the network.
- If we notice that packets are almost always arriving within 300 ms of being sent, then we can set our playback point accordingly, buffering any packets that arrive in less than 300 ms.
- Suppose that we subsequently observe that all packets are arriving within 100 ms of being sent.
- If we moved up our playback point to 100 ms, then the users of the application would probably perceive an improvement.
- The process of shifting the playback point would actually require us to play out samples at an increased rate for some period of time.
- With a voice application, this can be done in a way that is barely perceptible, simply by shortening the silences between words. Thus, playback point adjustment is fairly easy in this case, and it has been effectively implemented for several voice applications such as the audio teleconferencing program known as vat.

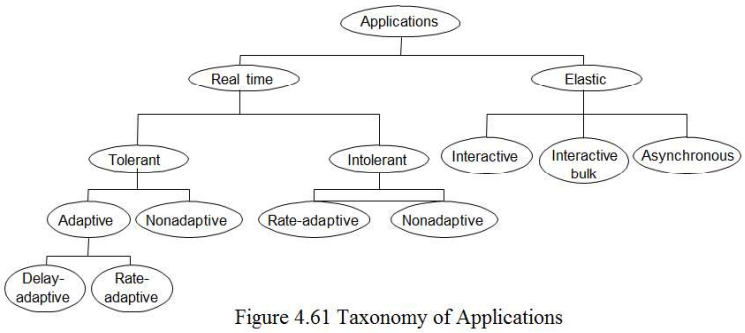


Figure 4.61 Taxonomy of Applications

- Observe that if we set our playback point on the assumption that all packets will arrive within 100 ms and then find that some packets are arriving slightly late, we will have to drop them, whereas we would not have had to drop them if we had left the playback point at 300 ms.
- We call applications that can adjust their playback point *delay-adaptive* applications.
- Another cl**a**ss of adaptive applications is *rate adap-tive*.
- For example, many video coding algorithms can trade off bit rate versus quality. Thus, if we find that the network can support a certain bandwidth, we can set our coding parameters accordingly.
- If more bandwidth becomes available later, we can change parameters to increase the quality.

Approaches to QoS Support

- Thesecan be divided into two broad categories:

1. *Fine-grained* approaches, which provide QoS to individual applications or flows.
   - Find *Integrated Services*, a QoS architecture developed in the IETF and often associated with the Resource Reservation Protocol (RSVP); ATM's approach to QoS was also in this category.
2. *Coarse-grained* approaches, which provide QoS to large classes of data or aggregated traffic
   - *Differentiated Services*, which is probably the most widely deployed QoS mechanism at the time of writing.