Unit 4 Transport Layer

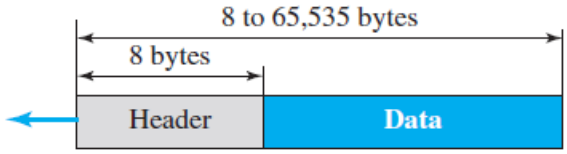
USER DATAGRAM PROTOCOL

The **User Datagram Protocol (UDP)** is a connectionless, unreliable transport protocol. If a process wants to send a small message and does not care much about reliability, it can use UDP.

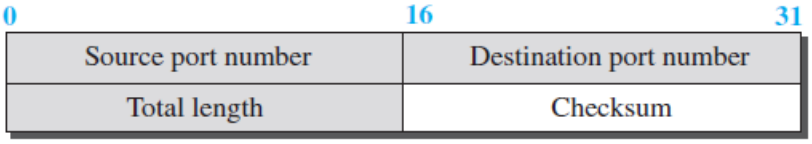
User Datagram

UDP packets, called ***user datagrams,*** have a fixed-size header of 8 bytes made of four fields, each of 2 bytes (16 bits). The first two fields define the source and destination port numbers. The third field 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. However, the total length needs to be less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The last field can carry the optional checksum.

User datagram packet format



a. UDP user datagram



b. Header format

Example 24.1

The following is the content of a UDP header in hexadecimal format.

CB84000D001C001C

a. What is the source port number?
b. What is the destination port number?
c. What is the total length of the user datagram?
d. What is the length of the data?
e. Is the packet directed from a client to a server or vice versa?
f. What is the client process?

Solution

a. The source port number is the first four hexadecimal digits $(CB84)_{16}$, which means that the source port number is 52100.
b. The destination port number is the second four hexadecimal digits $(000D)_{16}$, which means that the destination port number is 13.
c. The third four hexadecimal digits $(001C)_{16}$ define the length of the whole UDP packet as 28 bytes.
d. The length of the data is the length of the whole packet minus the length of the header, or $28 - 8 = 20$ bytes.
e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.
f. The client process is the Daytime (see Table 24.1).

UDP Services:

Process-to-Process Communication
UDP provides process-to-process communication using **socket addresses,** a combination of IP addresses and port numbers.

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. Also, unlike TCP, there is no connection establishment and no connection termination. This means that each user datagram can travel on a different path.

Flow Control
UDP is a very simple protocol. There is no *flow control,* and hence no window mechanism. The receiver may overflow with incoming messages. The lack of flow control means that the process using UDP should provide for this service, if needed.

Error Control
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. The lack of error control means that the process using UDP should provide for this service, if needed.

Congestion Control
Since UDP is a connectionless protocol, it does not provide congestion control. UDP assumes that the packets sent are small and sporadic and cannot create congestion in the network.

Encapsulation and Decapsulation
To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages.

TRANSMISSION CONTROL PROTOCOL

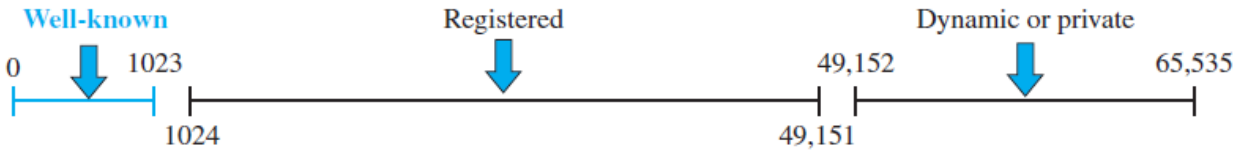
Transmission Control Protocol (TCP) is a connection-oriented, reliable protocol. TCP explicitly defines connection establishment, data transfer, and connection teardown phases to provide a connection-oriented service.

TCP Services:

Process-to-Process Communication
As with UDP, TCP provides process-to-process communication using port numbers.

Port	Protocol	UDP	TCP	SCTP	Description
7	Echo	√	√	√	Echoes back a received datagram
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	FTP-data		√	√	File Transfer Protocol
21	FTP-21		√	√	File Transfer Protocol
23	TELNET		√	√	Terminal Network
25	SMTP		√	√	Simple Mail Transfer Protocol
53	DNS	√	√	√	Domain Name Service
67	DHCP	√	√	√	Dynamic Host Configuration Protocol
69	TFTP	√	√	√	Trivial File Transfer Protocol
80	HTTP		√	√	HyperText Transfer Protocol

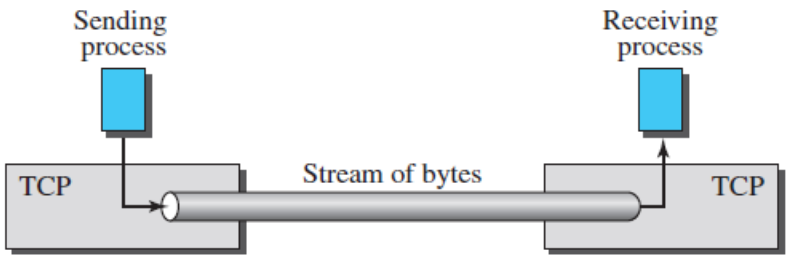
Figure 23.5 *ICANN ranges*



Stream Delivery Service

TCP, unlike UDP, is a stream-oriented protocol. TCP, on the other hand, 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 bytes across the Internet. The sending process produces (writes to) the stream and the receiving process consumes (reads from) it.

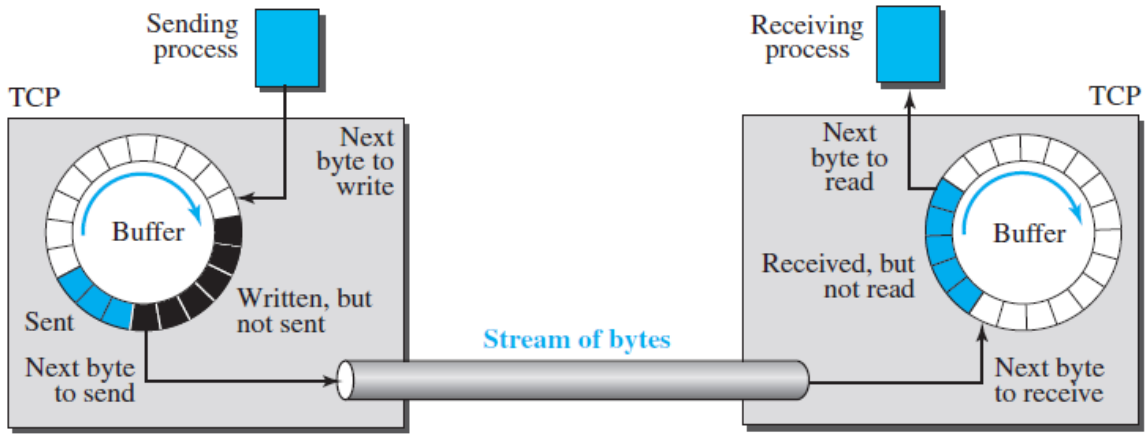
Stream delivery



Sending and Receiving Buffers

Because the sending and the receiving processes may not necessarily write or read data at the same rate, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer, one for each direction. These buffers are also necessary for flow- and error-control mechanisms used by TCP. One way to implement a buffer is to use a circular array of 1-byte locations as shown in
Figure

Figure 24.5 *Sending and receiving buffers*



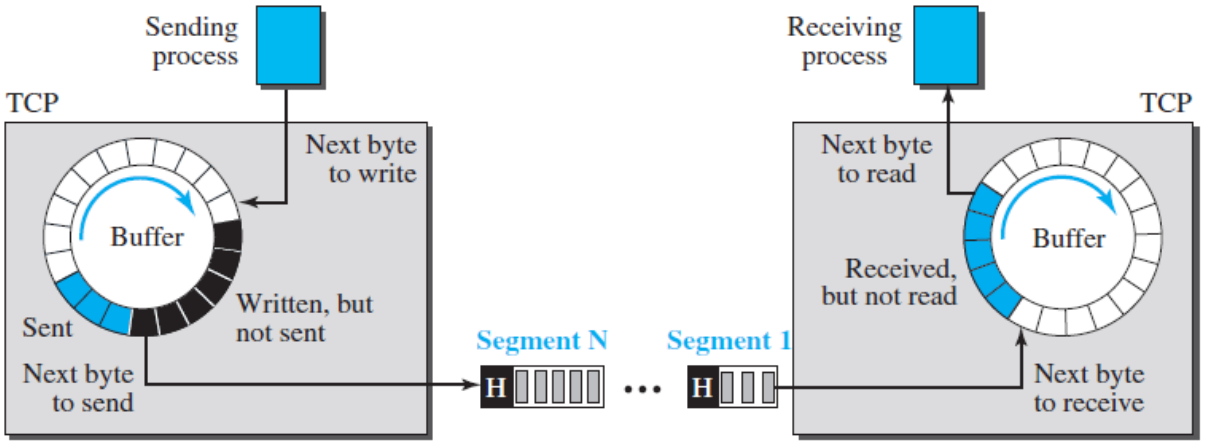
At the sender, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The colored area holds bytes that have been sent but not yet acknowledged. The TCP sender keeps these bytes in the buffer until it receives an acknowledgment. The shaded area contains bytes to be sent by the sending TCP.

The operation of the buffer at the receiver is simpler. 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.

Segments:

The network layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes. At the transport layer, 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 network layer for transmission. The segments are encapsulated in an IP datagram and transmitted. This entire operation is transparent to the receiving process.

Figure 24.6 *TCP segments*



Full-Duplex Communication

TCP offers *full-duplex service,* where data can flow in both directions at the same time Each TCP endpoint then has its own sending and receiving buffer, and segments move in both directions.

Multiplexing and Demultiplexing

Like UDP, TCP performs multiplexing at the sender and demultiplexing at the receiver.

Connection-Oriented Service

TCP, unlike UDP, is a connection-oriented protocol. When a process at site A wants to send to and receive data from another process at site B, the following three phases occur:

1. The two TCP's establish a logical connection between them.
2. Data are exchanged in both directions.
3. The connection is terminated.

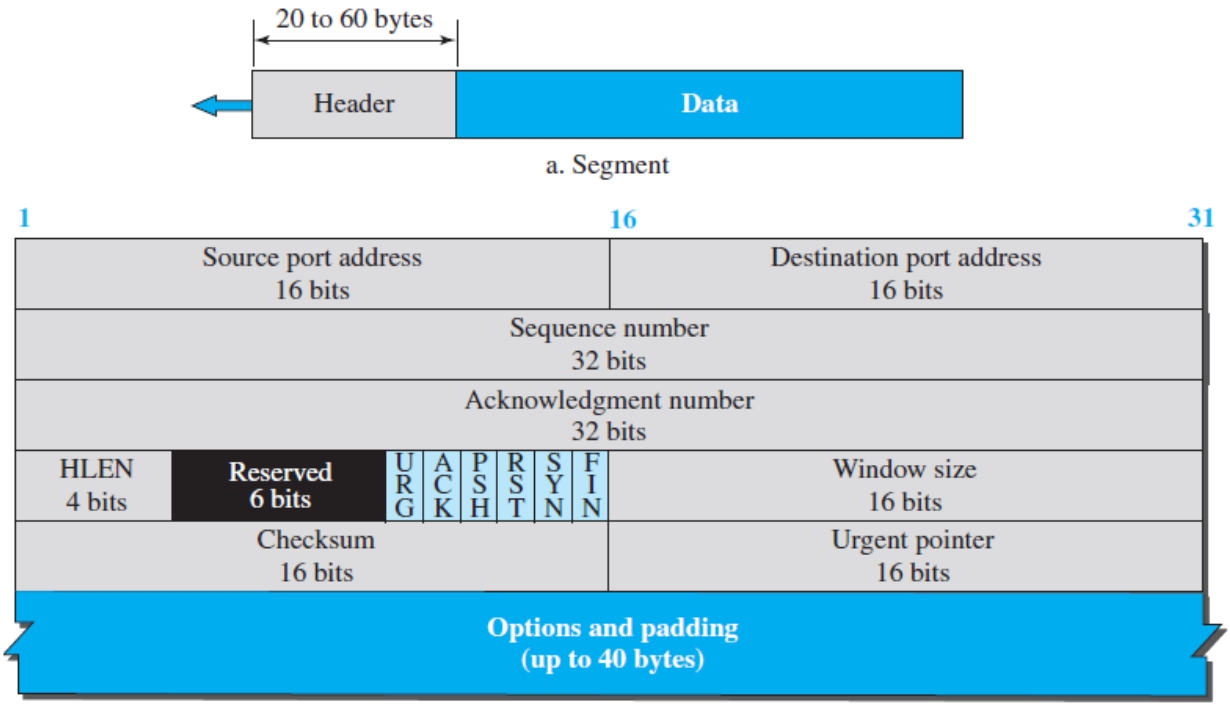
Reliable Service

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

TCP Segment

A packet in TCP is called a *segment.*

Figure 24.7 *TCP segment format*



Source port address. This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.

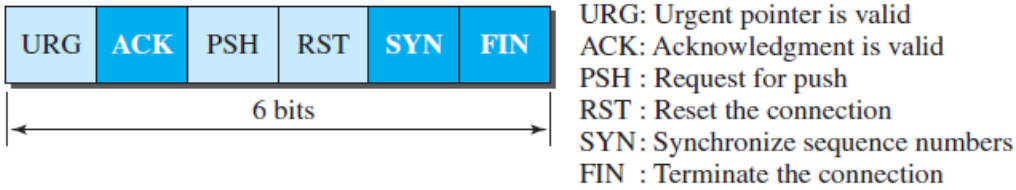
Destination port address. This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.

Sequence number. This 32-bit field defines the number assigned to the first byte of data contained in this segment. TCP is a stream transport protocol. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence is the first byte in the segment. During connection establishment (discussed later) each party uses a random number generator to create an **initial sequence number** (ISN), which is usually different in each direction.

Acknowledgment number. This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number *x* from the other party, it returns *x*+1 as the acknowledgment number. Acknowledgment and data can be piggybacked together.

Header length. This 4-bit field indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field is always between 5 (5x4=20) and 15 (15x4=60).

Control. This field defines 6 different control bits or flags, as shown in Figure 24.8. One or more of these bits can be set at a time.



Window size. This field defines the window size of the sending TCP in bytes. Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes. This value is normally referred to as the receiving window (*rwnd*) and is determined by the receiver. The sender must obey the dictation of the receiver in this case.

Checksum. This 16-bit field contains the checksum. The calculation of the checksum for TCP follows the same procedure as the one described for UDP

Urgent pointer. This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data.

Options. There can be up to 40 bytes of optional information in the TCP header.

Assignment:

The following is part of a TCP header dump (contents) in hexadecimal format.

E293 0017 00000001 00000000 5002 07FF...

a. What is the source port number?
b. What is the destination port number?
c. What is the sequence number?
d. What is the acknowledgment number?
e. What is the length of the header?
f. What is the type of the segment?
g. What is the window size?

Assignment:

The following is part of a TCP header dump (contents) in hexadecimal format.

E293 0017 00000001 00000000 5002 07FF...

a. What is the source port number?
b. What is the destination port number?
c. What is the sequence number?
d. What is the acknowledgment number?
e. What is the length of the header?
f. What is the type of the segment?
g. What is the window size?

Assignment:

The following is a dump of a TCP header in hexadecimal format.

05320017 00000001 00000000 500207FF 00000000

a. What is the source port number?
b. What is the destination port number?
c. What the sequence number?
d. What is the acknowledgment number?
e. What is the length of the header?
f. What is the type of the segment?
g. What is the window size?

Assignment:

The following is a dump of a UDP header in hexadecimal format.

06 32 00 0D 00 1C E2 17

a. What is the source port number?
b. What is the destination port number?
c. What is the total length of the user datagram?
d. What is the length of the data?
e. Is the packet directed from a client to a server or vice versa?
f. What is the client process?

TCP Connection

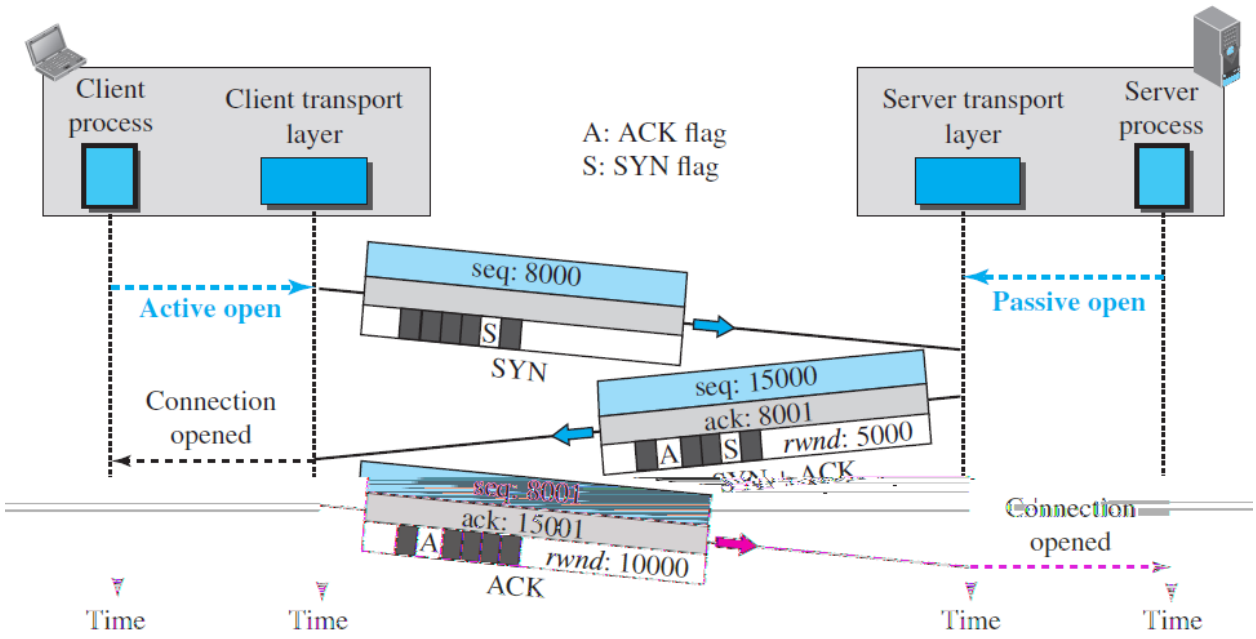
TCP is connection-oriented. a connection-oriented transport protocol establishes a logical path between the source and destination.
In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

Connection Establishment

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.
Three-Way Handshaking: The connection establishment in TCP is called ***three-way handshaking.*** The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This request is called a *passive open*. Although the server TCP is ready to accept a connection from any machine in the world, it cannot make the connection itself.

The client program issues a request for an *active open*. A client that wishes to connect to an open server tells its TCP to connect to a particular server. TCP can now start the three-way handshaking process.

Figure 24.10 *Connection establishment using three-way handshaking*



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. The client in our example chooses a random number as the first sequence number and sends this number to the server. This sequence number is called the *initial sequence number (ISN).*
A SYN segment cannot carry data, but it consumes one sequence number.

2. The server sends the second segment, a SYN + ACK segment with two flag bits set as: SYN and ACK. This segment has a dual purpose.
A SYN + ACK segment cannot carry data, but it 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.
An ACK segment, if carrying no data, consumes no sequence number.

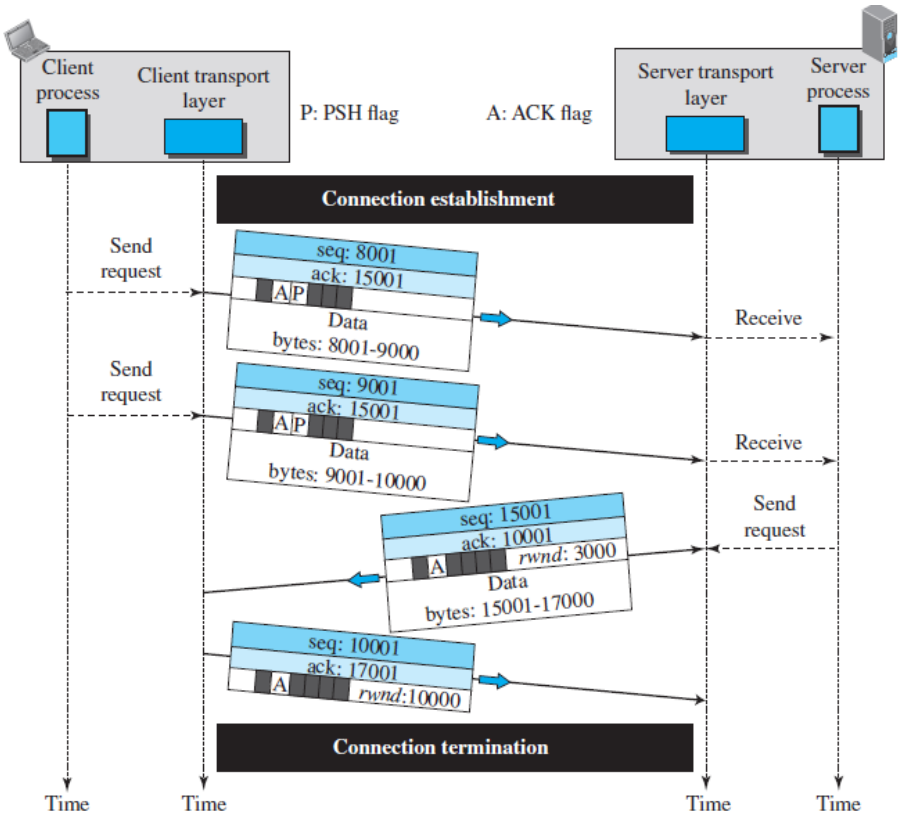
SYN Flooding Attack:

The connection establishment procedure in TCP is susceptible to a serious security problem called ***SYN flooding attack.*** This happens when one or more malicious attackers send a large number of SYN segments to a server pretending that each of them is coming from a different client by faking the source IP addresses in the datagrams. The server, assuming that the clients

are issuing an active open, allocates the necessary resources, such as creating transfer control block (TCB) tables and setting timers.

TCP server then sends the SYN + ACK segments to the fake clients, which are lost. When the server waits for the third leg of the handshaking process, however, resources are allocated without being used. If, during this short period of time, the number of SYN segments is large, the server eventually runs out of resources and may be unable to accept connection requests from valid clients. This SYN flooding attack belongs to a group of security attacks known as a ***denial of service attack,*** in which an attacker monopolizes a system with so many service requests that the system overloads and denies service to valid requests.

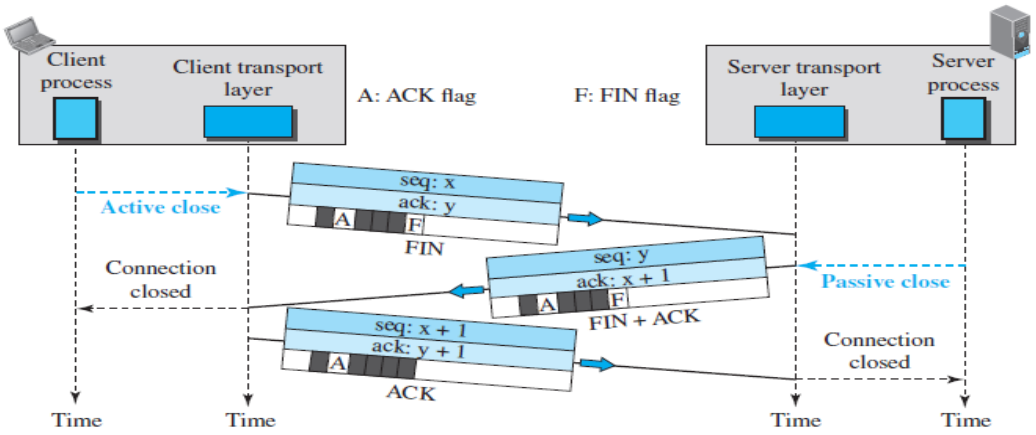
Data Transfer:



Connection Termination:

Using Three-Way Handshaking:

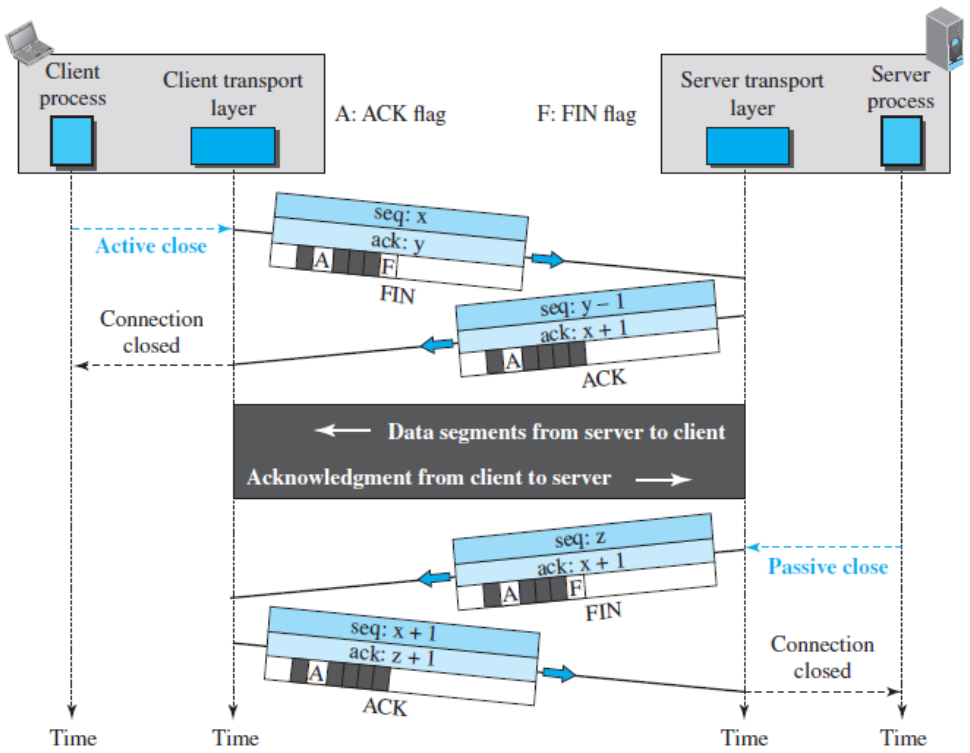
Figure 24.12 *Connection termination using three-way handshaking*



The FIN + ACK segment consumes only one sequence number if it does not carry data.

Using Half Close:

Figure 24.13 *Half-close*



Silly Window Syndrome:

A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both. Any of these situations results in the sending of data in very small segments, which reduces the efficiency of the operation. For example, if TCP sends segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data. Here the overhead is 41/1, which indicates that we are using the capacity of the network very inefficiently. The inefficiency is even worse after accounting for the data-link layer and physical-layer overhead. This problem is called the ***silly window syndrome.***

Syndrome Created by the Sender

The sending TCP may create a silly window syndrome if it is serving an application program that creates data slowly, for example, 1 byte at a time. The application program writes 1 byte at a time into the buffer of the sending TCP. If the sending TCP does not have any specific instructions, it may create segments containing 1 byte of data. The result is a lot of 41-byte segments that are travelling through an internet.

Nagle found an elegant solution. Nagle's algorithm is simple:

1. The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.
2. After sending the first segment, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data have accumulated to fill a maximum-size segment. At this time, the sending TCP can send the segment.
3. Step 2 is repeated for the rest of the transmission. Segment 3 is sent immediately if an acknowledgment is received for segment 2, or if enough data have accumulated to fill a maximum-size segment.

Syndrome Created by the Receiver

The receiving TCP may create a silly window syndrome if it is serving an application program that consumes data slowly, for example, 1 byte at a time. Suppose that the sending application program creates data in blocks of 1 kilobyte, but the receiving application program consumes data 1 byte at a time. Also suppose that the input buffer of the receiving TCP is 4 kilobytes. The sender sends the first 4 kilobytes of data. The receiver stores it in its buffer. Now its buffer is full. It advertises a window size of zero, which means the sender should stop sending data. The receiving application reads the first byte of data from the input buffer of the receiving TCP. Now there is 1 byte of space in the incoming buffer. The receiving TCP announces a window size of 1 byte, which means that the sending TCP, which is eagerly waiting to send data, takes this advertisement as good news and sends a segment carrying only 1 byte of data. The procedure will continue. One byte of data is consumed and a segment carrying 1 byte of data is sent. Again we have an efficiency problem and the silly window syndrome.

1. **Clark's solution** is to send an acknowledgment as soon as the data arrive, but to announce a window size of zero until either there is enough space to accommodate a segment of maximum size or until at least half of the receive buffer is empty.

2. The second solution is to delay sending the acknowledgment. This means that when a segment arrives, it is not acknowledged immediately. The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments. The delayed acknowledgment prevents the sending TCP from sliding its window. After the sending TCP has sent the data in the window, it stops. This kills the syndrome.

TCP Congestion Control:

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)

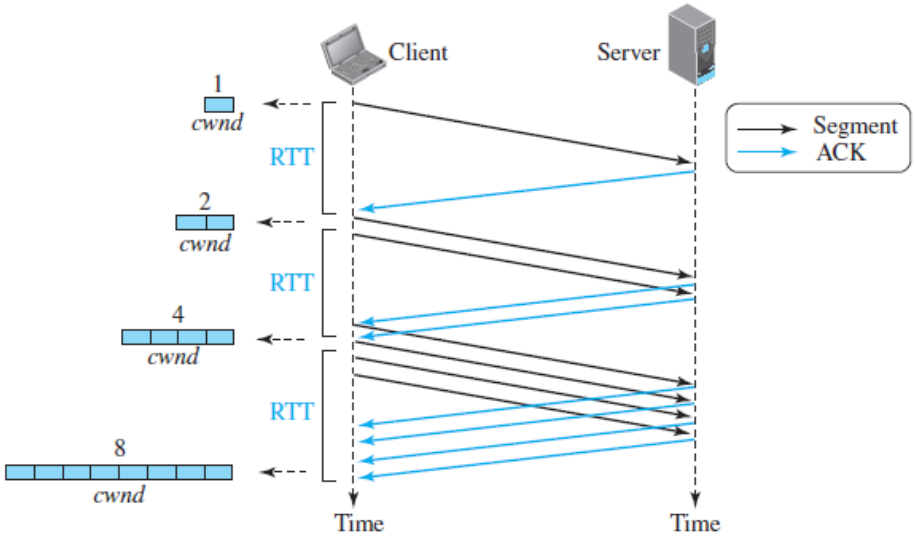
Congestion Policy

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.

Slow Start: Exponential Increase:

The **slow-start algorithm** is based on the idea that the size of the congestion window (*cwnd*) starts with one maximum segment size (MSS), but it increases one MSS each time an acknowledgment arrives.

Figure 24.29 *Slow start, exponential increase*



If an ACK arrives, $cwnd = cwnd + 1$.

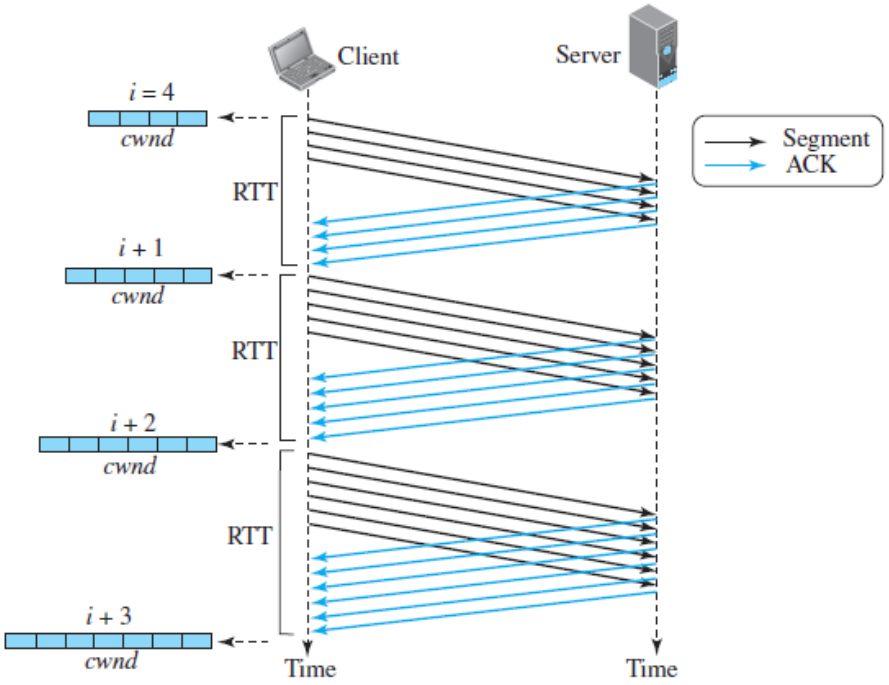
Start	➡	$cwnd = 1$
After round 1	➡	$cwnd = 2^1 = 2$
After round 2	➡	$cwnd = 2^2 = 4$
After round 3	➡	$cwnd = 2^3 = 8$

In the slow-start algorithm, the size of the congestion window increases exponentially until it reaches a threshold.

Congestion Avoidance: Additive Increase

To avoid congestion before it happens, we must slow down this exponential growth. TCP defines another algorithm called ***congestion avoidance,*** which increases the *cwnd* additively instead of exponentially. When the size of the congestion window reaches the slow-start threshold in the case where *cwnd* = *i*, the slow-start phase stops and the additive phase begins. In this algorithm, each time the whole "window" of segments is acknowledged, the size of the congestion window is increased by one. A window is the number of segments transmitted during RTT.

Figure 24.30 *Congestion avoidance, additive increase*



If an ACK arrives, $cwnd = cwnd + (1/cwnd)$.

Start	→	$cwnd = i$
After 1 RTT	→	$cwnd = i + 1$
After 2 RTT	→	$cwnd = i + 2$
After 3 RTT	→	$cwnd = i + 3$

ow — **In the congestion-avoidance algorithm, the size of the congestion windo 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. However, retransmission can occur in one of two cases: when a timer times out or when three Duplicate ACKs are received. In both cases, the size of the threshold is dropped to one-half, a multiplicative decrease.

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.

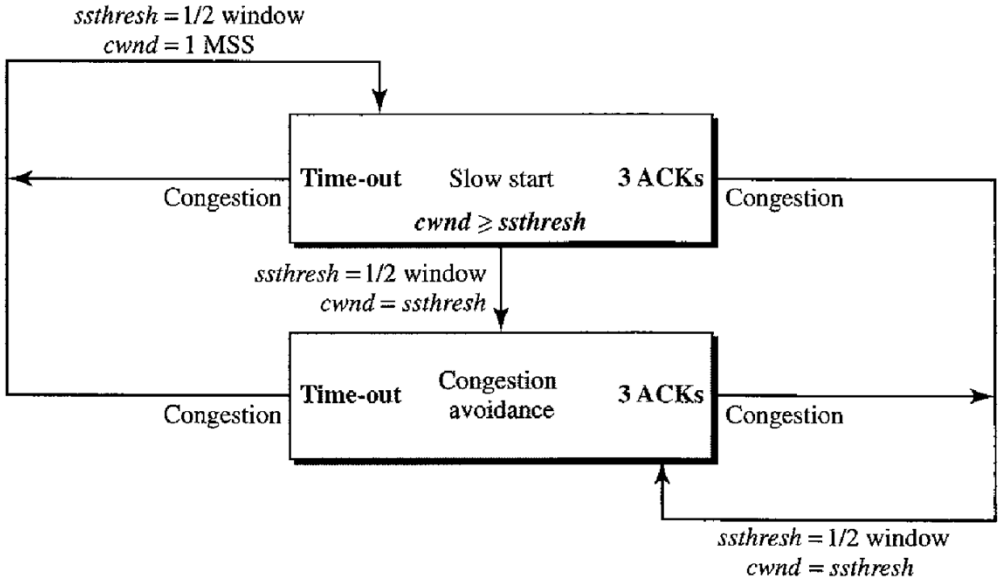
2. If three ACKs are received, there is a weaker possibility of congestion; a segment may have been dropped, but some segments after that may have arrived safely since three ACKs are received. This is called fast transmission and fast recovery.

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.
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.

Figure 24.10 *TCP congestion policy summary*

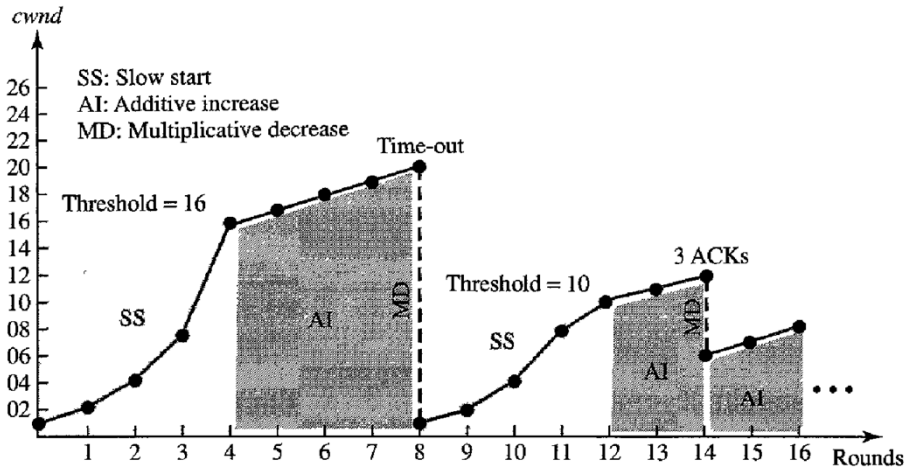


Example: We assume that the maximum window size is 32 segments. The threshold is set to 16 segments (one-half of the maximum window size). In the *slow-start* phase the window size starts from 1 and grows exponentially until it reaches the threshold. After it reaches the threshold, the

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 24.11, 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 duplicate 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 duplicate ACKs happen.

Figure 24.11 *Congestion example*



Sockets:

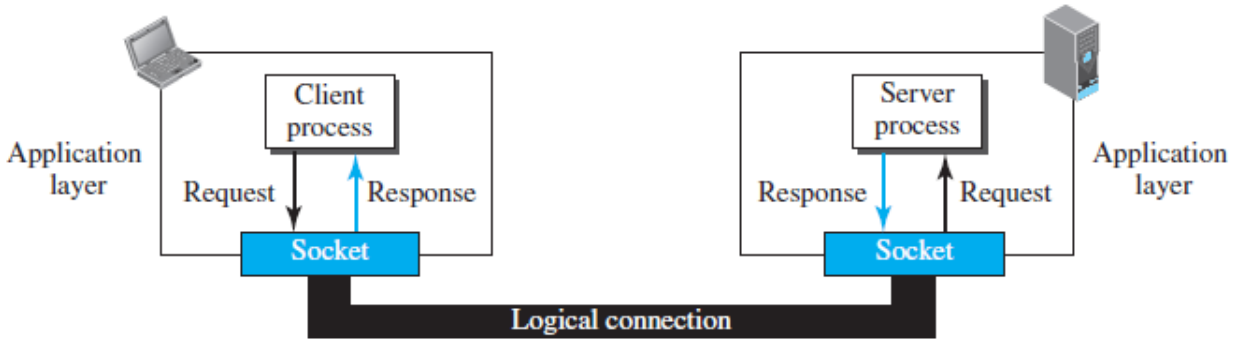
Communication between a client process and a server process is communication between two sockets, created at two ends The client thinks that the socket is the entity that receives the request and gives the response; the server thinks that the socket is the one that has a request and needs the response. If we create two sockets, one at each end, and define the source and destination addresses correctly, we can use the available instructions to send and receive data.

Assignment:

Question: Consider the effect of using slow start on a line with a 10 m-sec RTT and no congestion. The receive window is 30KB and the maximum segment size is 2KB. How long does it take before the first full window can be sent?

Question: Consider the effect of using congestion avoidance on a line with a 10 m-sec RTT and no congestion. The receive window is 30KB and the maximum segment size is 2KB. How long does it take before the first full window can be sent?

Figure 25.6 *Use of sockets in process-to-process communication*



Socket Addresses:

Figure 25.7 *A socket address*