MODULE -4

TRANSPORT LAYER

VTUSYNC.IN

1. INTRODUCTION

The transport layer is an essential part of the network architecture, located between the application layer and the network layer. Its primary function is to facilitate process-to-process communication between application layers on two different hosts—one local and one remote. This communication is achieved using a logical connection, meaning that even though the application layers may be on opposite sides of the world, they interact as if they are directly connected.

Logical Connection

A logical connection at the transport layer is an abstract concept that represents a direct communication link between two application layers. In reality, the data travels through multiple intermediate devices like routers and switches, but the transport layer manages the connection in such a way that the two communicating applications are unaware of the underlying complexity.

For example, consider a scenario where Alice's computer at Sky Research communicates with Bob's computer at Scientific Books. Even though the data passes through various intermediate devices and networks, the transport layer makes it appear as though there is a direct link between Alice and Bob's applications.

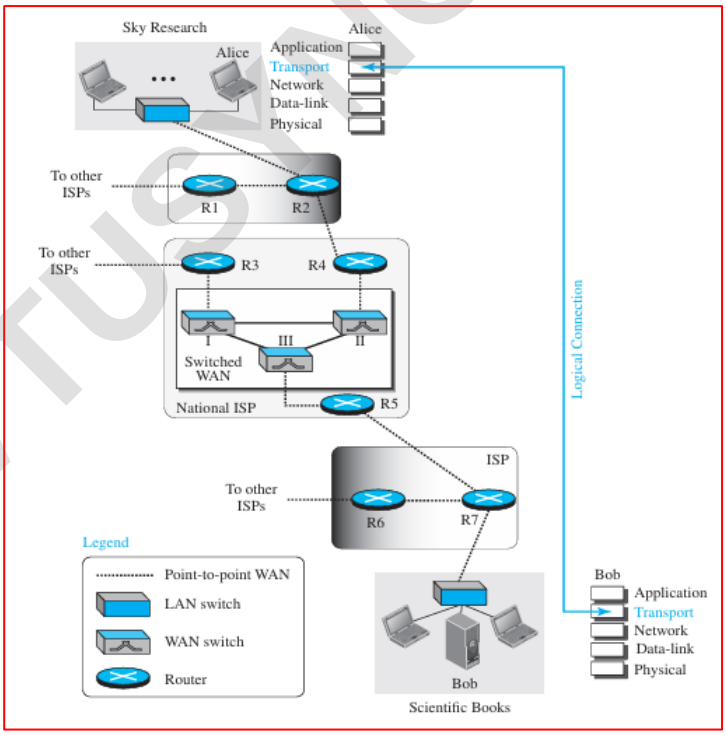


Figure 4.1: Logical connection at the transport layer

1.1 Transport-Layer Services

The transport layer is positioned between the network layer and the application layer in the OSI model. Its primary role is to provide communication services to the application layer while receiving services from the network layer. The transport layer ensures that data is transferred between processes running on different hosts in a reliable and efficient manner.

- Process-to-Process Communication

VTUSYNC.IN

- Addressing: Port Numbers
- Encapsulation and Decapsulation
- Multiplexing and Demultiplexing
- Flow Control
- Error Control
- Combination of Flow and Error Control
- Congestion Control

1. Process-to-Process Communication

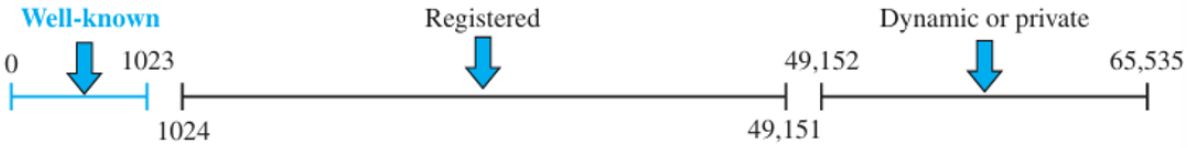
The transport layer's fundamental responsibility is to facilitate process-to-process communication. A process refers to an application-layer program that utilizes the services provided by the transport layer. This communication is distinguished from host-to-host communication, which is handled by the network layer. The network layer ensures that data reaches the correct host, but the transport layer is responsible for delivering the data to the appropriate process within that host.

2. Addressing: Port Numbers

To achieve process-to-process communication, the transport layer uses port numbers. Each process is identified by a port number, allowing multiple processes to run on the same host simultaneously. In a typical client-server model, the client is assigned an ephemeral (temporary) port number, while the server uses a well-known port number.

- **Well-known Ports:** These range from 0 to 1023 and are assigned by ICANN for standard services.
- **Registered Ports:** These range from 1024 to 49,151 and can be registered with ICANN to prevent duplication.
- **Dynamic or Private Ports:** These range from 49,152 to 65,535 and are used temporarily by client processes.

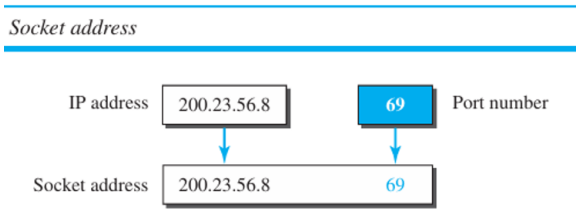
Figure 23.5 *ICANN ranges*



3. Socket Addresses

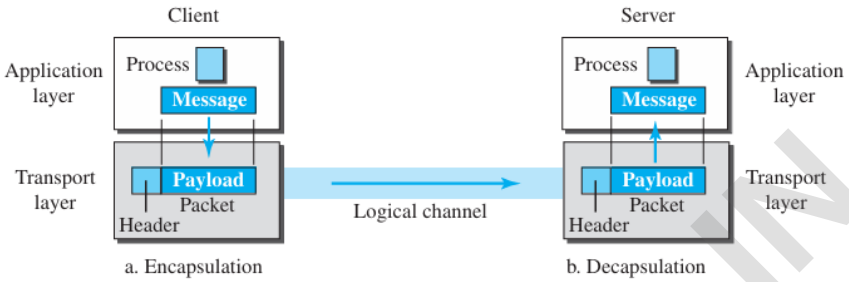
A socket address is a combination of an IP address and a port number, uniquely identifying a process on a networked host. For communication to occur, the transport layer requires the socket addresses of both the client and the server. This allows data to be directed to the correct process on the correct host.

VTUSYNC.IN



4. Encapsulation and Decapsulation

Figure 23.7 *Encapsulation and decapsulation*



- **Encapsulation:** At the sender's end, the transport layer encapsulates the message with a transport-layer header, creating a packet (user datagram, segment, or packet depending on the protocol).
- **Decapsulation:** At the receiver's end, the transport layer removes the header and delivers the message to the appropriate process.
-

5. Multiplexing and Demultiplexing

- **Multiplexing:** The transport layer at the sender's side combines data from multiple processes into a single stream for transmission over the network.
- **Demultiplexing:** At the receiver's side, the transport layer separates the combined data stream back into individual messages and delivers them to the correct processes.

6. Flow Control

Flow control ensures that the rate of data transmission is balanced between the sender and receiver to prevent overwhelming the receiver. Flow control mechanisms can be implemented using buffers at both the sender and receiver's transport layers. The transport layer at the sender's side manages the data flow based on feedback from the receiver's transport layer.

- **Pushing vs. Pulling:** Flow control can be push-based (data is sent as it is produced) or pull-based (data is sent upon request).

7. Error Control

Error control at the transport layer ensures data integrity and reliability, particularly in environments where the underlying network layer (IP) is unreliable. The transport layer handles:

- Detection and discarding of corrupted packets.
- Resending lost or discarded packets.
- Identifying and discarding duplicate packets.
- Buffering out-of-order packets until the correct sequence is restored.

VTUSYNC.IN

Sequence Numbers and Acknowledgments: Sequence numbers in the transport-layer packets help manage these tasks. The sender assigns a sequence number to each packet, and the receiver uses acknowledgments (ACKs) to confirm receipt or request retransmission of missing packets.

8. Sliding Window

The sliding window mechanism is used in flow control and error control to manage the transmission of packets. The window represents a range of sequence numbers for packets that can be sent or have been sent but not yet acknowledged. When an acknowledgment is received, the window "slides," allowing new packets to be sent.

9. Congestion Control

Congestion occurs when the network's load exceeds its capacity, leading to packet delays and losses. Congestion control mechanisms aim to regulate traffic to prevent congestion, ensuring that the network operates efficiently. The transport layer, particularly in protocols like TCP, incorporates congestion control techniques to manage traffic based on network conditions.

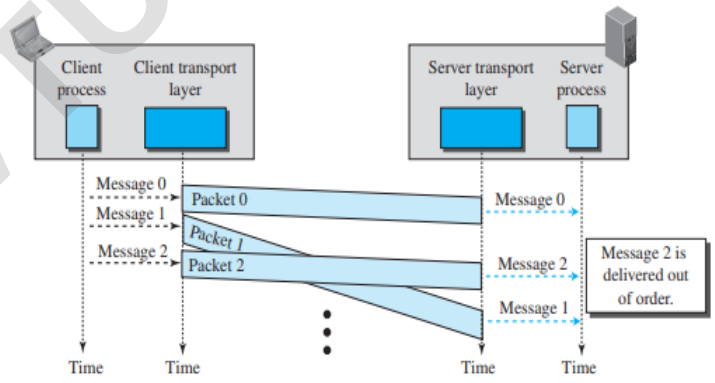
1.2 Connectionless and Connection-Oriented Protocols

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

Connectionless Service In a connectionless service, the source process (application program) needs to divide its message into chunks of data of the size acceptable by the transport layer and deliver them to the transport layer one by one. The transport layer treats each chunk as a single unit without any relation between the chunks. When a chunk arrives from the application layer, the transport layer encapsulates it in a packet and sends it.

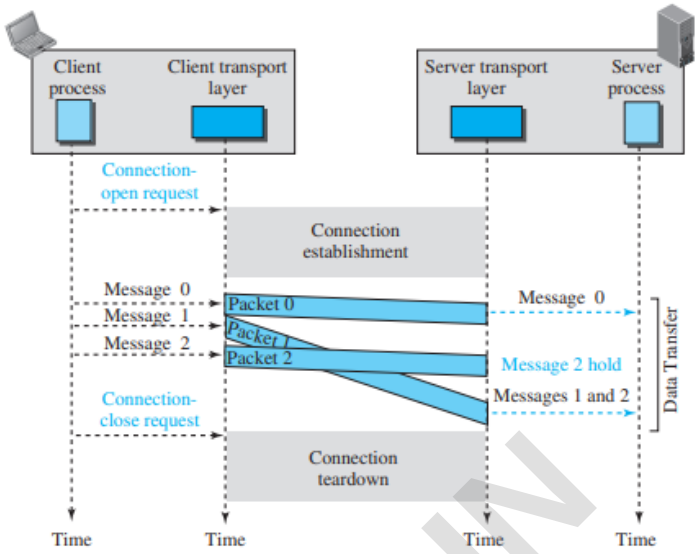
VTUSYNC.IN

Figure 23.14 *Connectionless service*



Connection-Oriented Service In a connection-oriented service, the client and the server first need to establish a logical connection between themselves. The data exchange can only happen after the connection establishment. After data exchange, the connection needs to be torn down

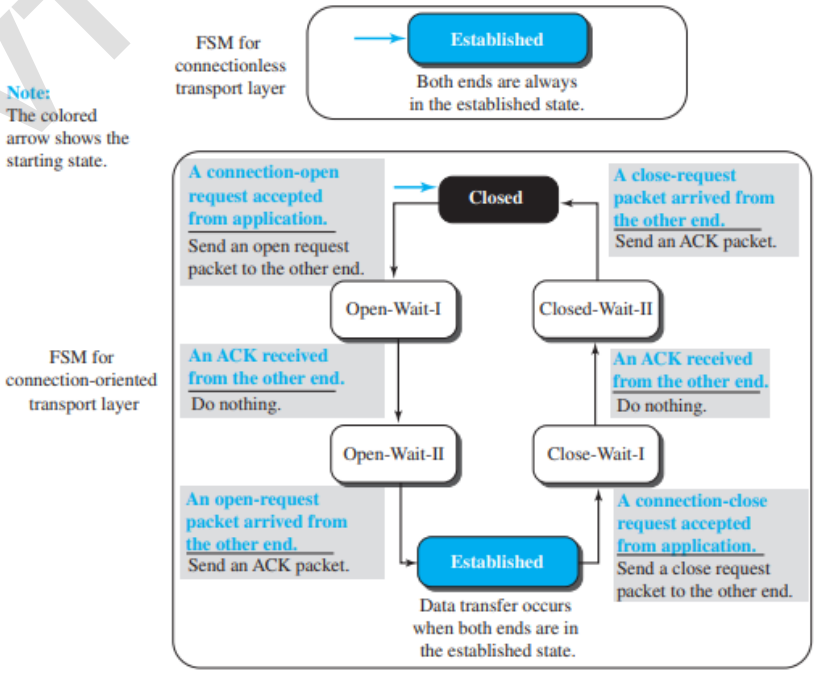
Figure 23.15 *Connection-oriented service*



Finite State Machine

- The behavior of a transport-layer protocol, both when it provides a connectionless and when it provides a connection-oriented protocol, can be better shown as a finite state machine (FSM).
- The machine is always in one of the states until an event occurs.
- A horizontal line is used to separate the event from the actions, although later we replace the horizontal line with a slash. The arrow shows the movement to the next state. The machine is in the closed state when there is no connection.

Figure 23.16 *Connectionless and connection-oriented service represented as FSMs*



VTUSYNC.IN

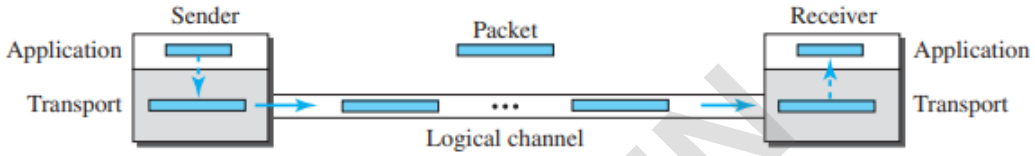
2. TRANSPORT-LAYER PROTOCOLS

The TCP/IP protocol uses a transport-layer protocol that is either a modification or a combination of some of these protocols.

2.1 Simple Protocol

It is a simple connectionless protocol with neither flow nor error control. We assume that the receiver can immediately handle any packet it receives. In other words, the receiver can never be overwhelmed with incoming packets.

Figure 23.17 *Simple protocol*



- The transport layer at the sender gets a message from its application layer, makes a packet out of it, and sends the packet.
- The transport layer at the receiver receives a packet from its network layer, extracts the message from the packet, and delivers the message to its application layer.
- The transport layers of the sender and receiver provide transmission services for their application layers.

FSMs

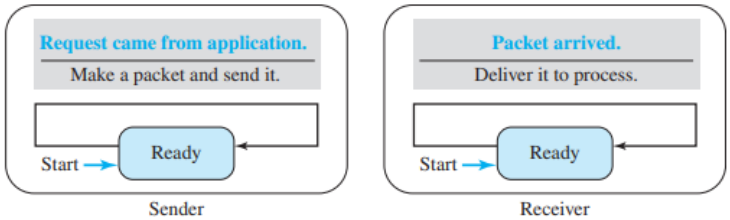
- The sender site should not send a packet until its application layer has a message to send.
- The receiver site cannot deliver a message to its application layer until a packet arrives.

We can show these requirements using two FSMs. Each FSM has only one state, the ready state.

- The sending machine remains in the ready state until a request comes from the process in the application layer. When this event occurs, the sending machine encapsulates the message in a packet and sends it to the receiving machine.
- The receiving machine remains in the ready state until a packet arrives from the sending machine. When this event occurs, the receiving machine decapsulates the message out of the packet and delivers it to the process at the application layer.

Figure 23.18 shows the FSMs for the simple protocol.

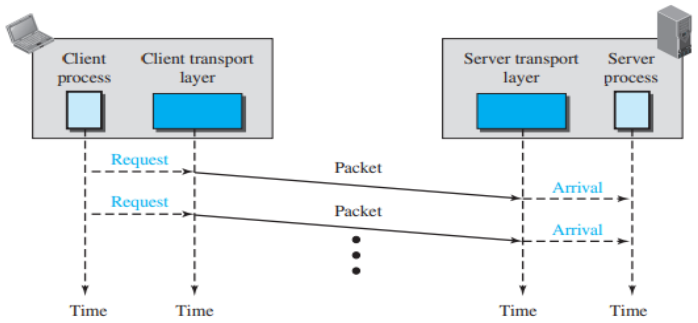
Figure 23.18 *FSMs for the simple protocol*



VTUSYNC.IN

Figure 23.19 shows an example of communication using this protocol. It is very simple. The sender sends packets one after another without even thinking about the receiver.

Figure 23.19 *Flow diagram for Example 23.3*

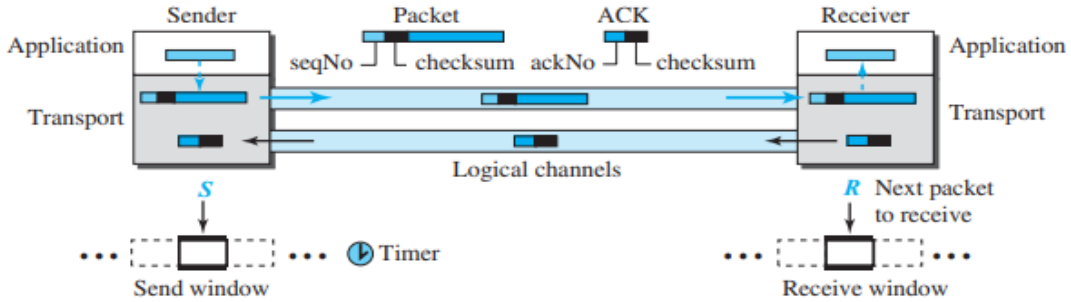


2.2 Stop-and-Wait Protocol

This protocol uses both flow and error control. Both the sender and the receiver use a sliding window of size 1.

- The sender sends one packet at a time and waits for an acknowledgment before sending the next one. To detect corrupted packets, we need to add a checksum to each data packet.
- When a packet arrives at the receiver site, it is checked. If its checksum is incorrect, the packet is corrupted and silently discarded.
- The silence of the receiver is a signal for the sender that a packet was either corrupted or lost. Every time the sender sends a packet, it starts a timer. If an acknowledgment arrives before the timer expires, the timer is stopped and the sender sends the next packet (if it has one to send).
- If the timer expires, the sender resends the previous packet, assuming that the packet was either lost or corrupted. This means that the sender needs to keep a copy of the packet until its acknowledgment arrives.
- Figure 23.20 shows the outline for the Stop-and-Wait protocol. Note that only one packet and one acknowledgment can be in the channels at any time.

Figure 23.20 *Stop-and-Wait protocol*



The Stop-and-Wait protocol is a connection-oriented protocol that provides flow and error control.

Sequence Numbers

VTUSYNC.IN

A field is added to the packet header to hold the sequence number of that packet to prevent the duplicate of packet. Since we want to minimize the packet size, we look for the smallest range that provides unambiguous communication.

Let us discuss the range of sequence numbers we need. Assume we have used x as a sequence number; we only need to use x + 1 after that. There is no need for x + 2.

To show this, assume that the sender has sent the packet with sequence number x. Three things can happen.

1. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment. The acknowledgment arrives at the sender site, causing the sender to send the next packet numbered x + 1.
2. The packet is corrupted or never arrives at the receiver site; the sender resends the packet (numbered x) after the time-out. The receiver returns an acknowledgment.
3. The packet arrives safe and sound at the receiver site; the receiver sends an ack, but the ack is corrupted or lost. The sender resends the packet (numbered x) after the time-out. Note that the packet here is a duplicate. The receiver can recognize this fact because it expects packet x + 1 but packet x was received.

Acknowledgment Numbers

- ❖ Since the sequence numbers must be suitable for both data packets and ack packet
- ❖ The ack numbers always announce the sequence number of the next packet expected by the receiver.
- ✓ For example, if packet 0 has arrived safe and sound, the receiver sends an ACK with acknowledgment 1 (meaning packet 1 is expected next).
- ✓ If packet 1 has arrived safe and sound, the receiver sends an ACK with acknowledgment 0 (meaning packet 0 is expected).

FSMs

The Stop-and-Wait protocol is a connection-oriented protocol; both ends should be in the established state before exchanging data packets.

Sender

The sender is initially in the ready state, but it can move between the ready and blocking state. The variable S is initialized to 0.

❑ **Ready state.** When the sender is in this state, it is only waiting for one event to occur. If a request comes from the application layer, the sender creates a packet with the sequence number set to S. A copy of the packet is stored, and the packet is sent. The sender then starts the only timer. The sender then moves to the blocking state.

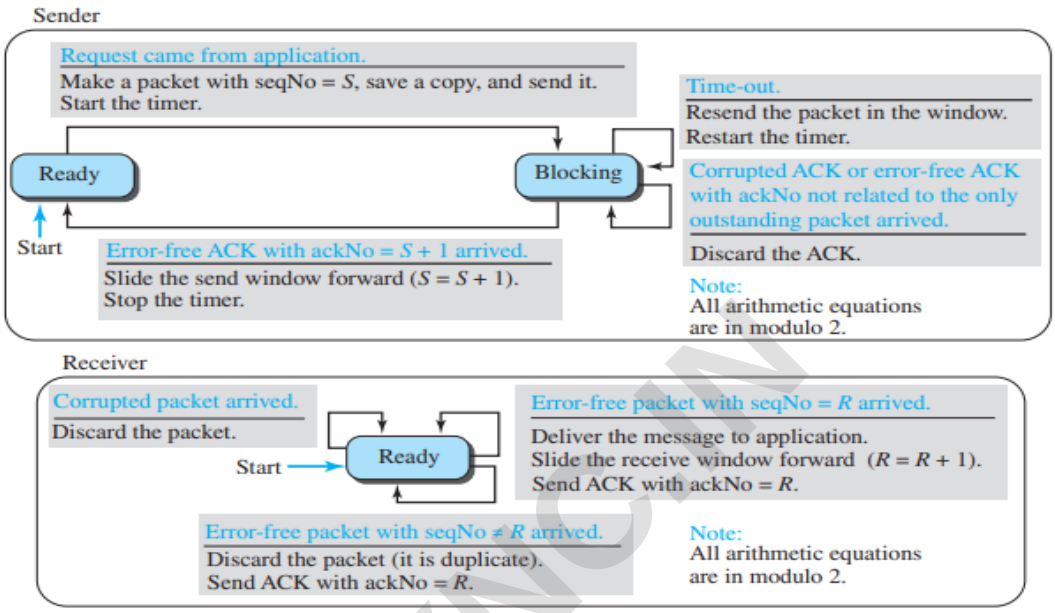
❑ **Blocking state.** When the sender is in this state, three events can occur:

a. If an error-free ACK arrives with the ackNo related to the next packet to be sent, which means ackNo = (S + 1) modulo 2, then the timer is stopped. The window slides, S = (S + 1) modulo 2. Finally, the sender moves to the ready state.

VTUSYNC.IN

b. If a corrupted ACK or an error-free ACK with the ackNo ≠ (S + 1) modulo 2 arrives, the ACK is discarded.
c. If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

Figure 23.21 *FSMs for the Stop-and-Wait protocol*



Receiver

The receiver is always in the ready state. Three events may occur:

a. If an error-free packet with seqNo = R arrives, the message in the packet is delivered to the application layer. The window then slides, R = (R + 1) modulo 2. Finally an ACK with ackNo = R is sent.
b. If an error-free packet with seqNo ≠ R arrives, the packet is discarded, but an ACK with ackNo = R is sent.
c. If a corrupted packet arrives, the packet is discarded.

Efficiency

This protocol is very inefficient if our channel is thick and long. By thick, we mean that our channel has a large bandwidth (high data rate); by long, we mean the round-trip delay is long. The product of these two is called the bandwidthdelay product.

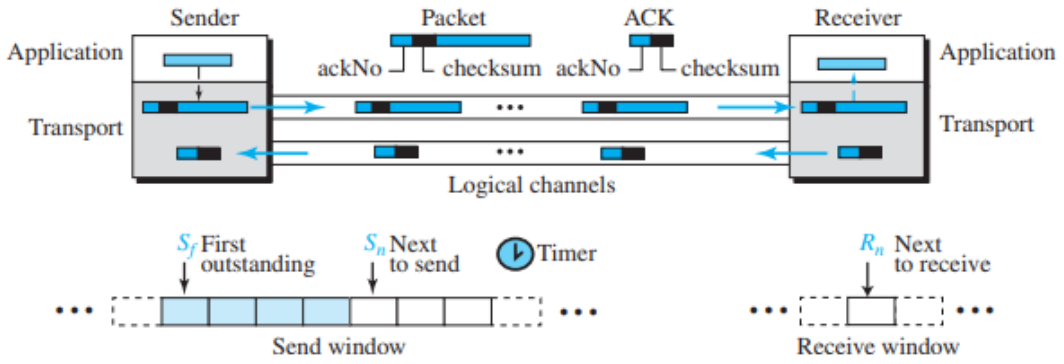
2.3 Go-Back-N Protocol (GBN)

To improve the efficiency of transmission (to fill the pipe), multiple packets must be in transition while the sender is waiting for acknowledgment.

The key to Go-back-N is to send several packets before receiving ack, but the receiver can only buffer one packet. We keep a copy of the sent packets until the ack arrive. Note that several data packets and ack can be in the channel at the same time

VTUSYNCIN

Figure 23.23 *Go-Back-N protocol*



Sequence Numbers

The sequence numbers are modulo 2m, where m is the size of the sequence number field in bits.

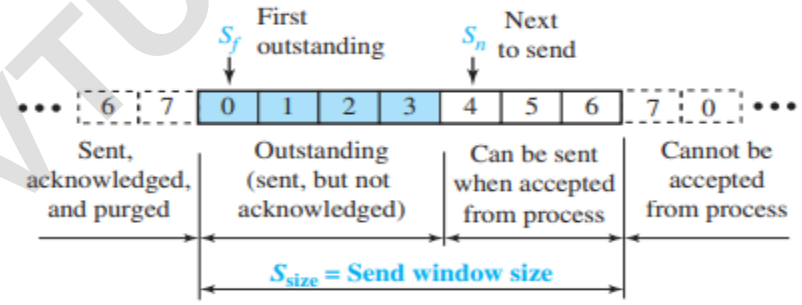
Acknowledgment Numbers

An acknowledgment number is cumulative and defines the sequence number of the next packet expected.

Send Window

The send window is an imaginary box covering the sequence numbers of the data packets that can be in transit or can be sent. In each window position, some of these sequence numbers define the packets that have been sent; others define those that can be sent. The maximum size of the window is $2^m - 1$. Figure 23.24 shows a sliding window of size 7 (m = 3) for the Go-Back-N protocol

Send window for Go-Back-N



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

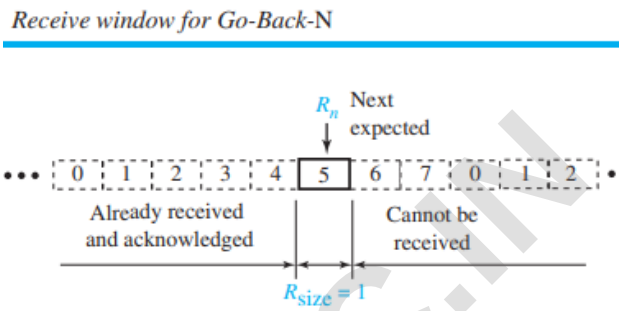
1. The first region, left of the window, defines the sequence numbers belonging to packets that are already acknowledged.
2. The second region, colored, defines the range of sequence numbers belonging to the packets that have been sent, but have an unknown status. The sender needs to wait to find out if these packets have been received or were lost. We call these outstanding packets.
3. The third range, white in the figure, defines the range of sequence numbers for packets that can be sent; however, the corresponding data have not yet been received from the application layer.
4. Finally, the fourth region, right of the window, defines sequence numbers that cannot be used until the window slides.

VTUSYNC.IN

The window is an abstraction of three variables defines its size and location at any time.
We call these variables Sf (send window, the first outstanding packet), Sn (send window, size), and Ssize (send window, size).

Receive Window

It ensures the correct data packets are received and that the correct acknowledgments are sent. In Go-Back-N, the size of the receive window is always 1. The receiver is always looking for the arrival of a specific packet. Any packet arriving out of order is discarded and needs to be resent. Note that we need only one variable, Rn (receive window, next packet expected), to define this abstraction.



Timers

Although there can be a timer for each packet that is sent, in our protocol we use only one. The reason is that the timer for the first outstanding packet always expires first. We resend all outstanding packets when this timer expires.

Resending packets

When the timer expires, the sender resends all outstanding packets.

FSMs

Sender

The sender starts in the ready state, but thereafter it can be in one of the two states: ready or blocking. The two variables are normally initialized to 0 (Sf = Sn = 0).

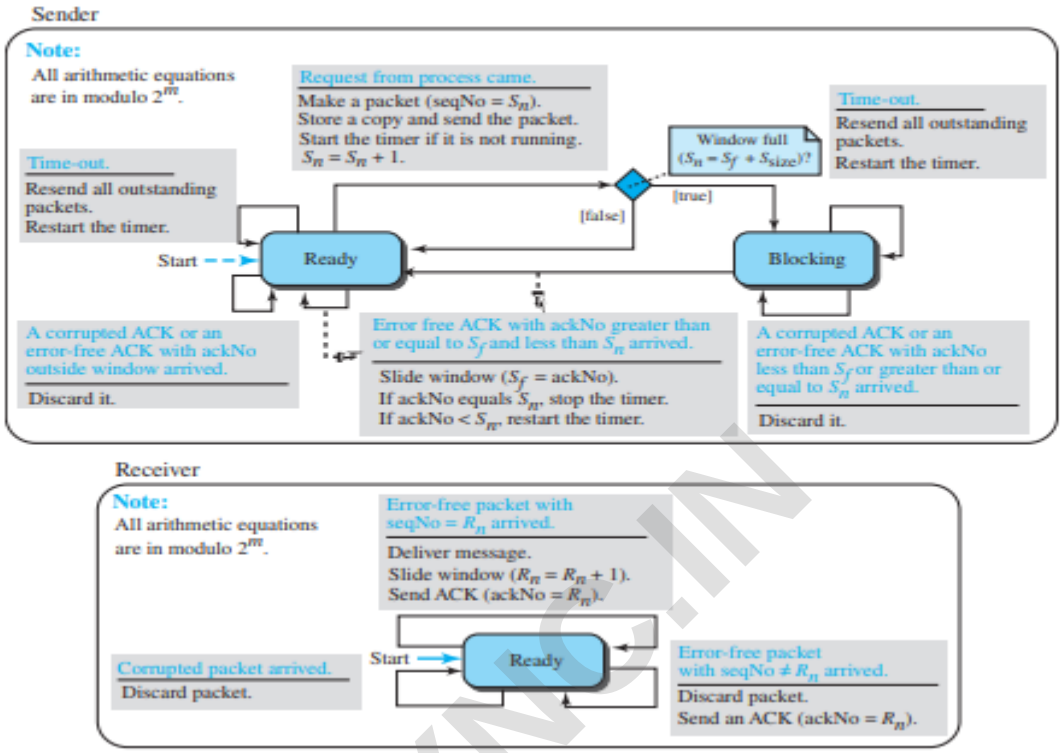
❑**Ready state.** Four events may occur when the sender is in ready state.

a. If a request comes from the application layer, the sender creates a packet with the sequence number set to Sn. A copy of the packet is stored, and the packet is sent. The sender also starts the only timer if it is not running. The value of Sn is now incremented, (Sn = Sn + 1) modulo 2^m. If the window is full, Sn = (Sf + Ssize) modulo 2m, the sender goes to the blocking state.

b. If an error-free ACK arrives with ackNo related to one of the outstanding packets, the sender slides the window (set Sf = ackNo), and if all outstanding packets are acknowledged (ackNo = Sn), then the timer is stopped. If all outstanding packets are not acknowledged, the timer is restarted.

VTUSYNC.IN

Figure 23.27 *FSMs for the Go-Back-N protocol*



c. If a corrupted ACK or an error-free ACK with ackNo not related to the outstanding packet arrives, it is discarded.

d. If a time-out occurs, the sender resends all outstanding packets and restarts the timer.

❑**Blocking state.** Three events may occur in this case:

a. If an error-free ACK arrives with ackNo related to one of the outstanding packets, the sender slides the window (set Sf = ackNo) and if all outstanding packets are acknowledged (ackNo = Sn), then the timer is stopped. If all outstanding packets are not acknowledged, the timer is restarted. The sender then moves to the ready state.

b. If a corrupted ACK or an error-free ACK with the ackNo not related to the outstanding packets arrives, the ACK is discarded.

c. If a time-out occurs, the sender sends all outstanding packets and restarts the timer.

Receiver

The receiver is always in the ready state. The only variable, Rn, is initialized to 0. Three events may occur:

a. If an error-free packet with seqNo = Rn arrives, the message in the packet is delivered to the application layer. The window then slides, Rn = (Rn + 1) modulo 2m. Finally an ACK is sent with ackNo = Rn.

b. If an error-free packet with seqNo outside the window arrives, the packet is discarded, but an ACK with ackNo = Rn is sent.

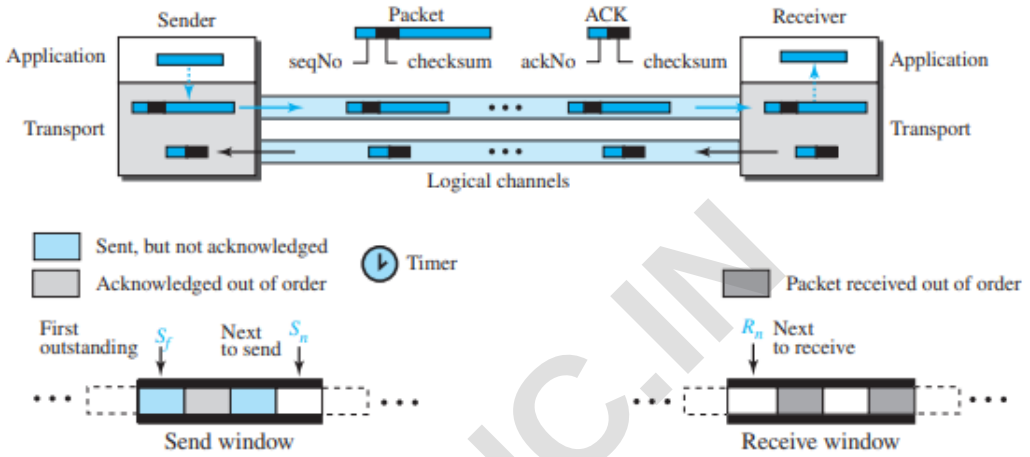
VTUSYNC.IN

c. If a corrupted packet arrives, it is discarded.

2.4 Selective-Repeat Protocol

The Selective-Repeat (SR) protocol, has been devised, which, as the name implies, resends only selective packets, those that are actually lost.

Figure 23.31 *Outline of Selective-Repeat*

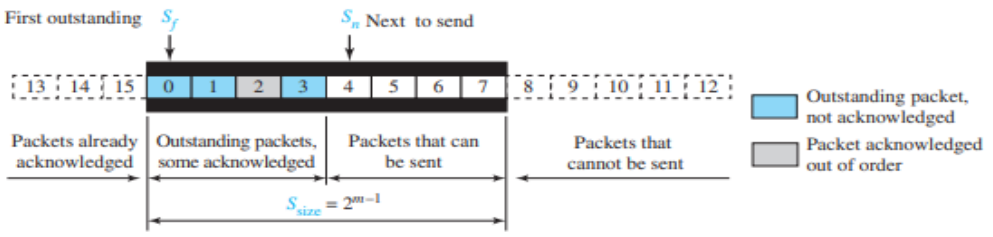


Windows

The Selective-Repeat protocol also uses two windows: a send window and a receive window

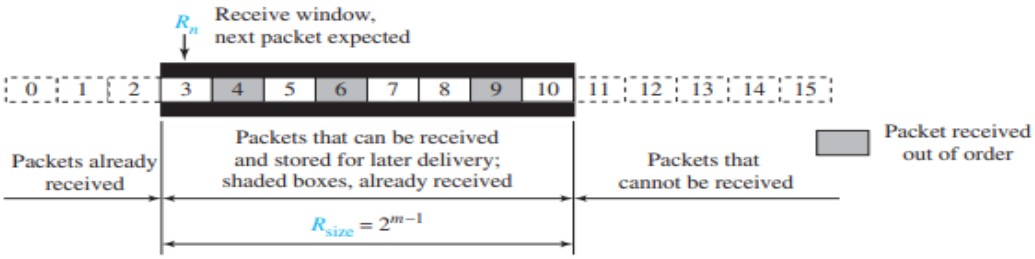
- The send window maximum size can be 2m−1. For example, if m = 4, the sequence numbers go from 0 to 15, but the maximum size of the window is just 8 (it is 15 in the Go-Back-N Protocol).
- The size of the receive window is the same as the size of the send window (maximum 2^m-1).
- The SR protocol allows as many packets as the size of the receive window to arrive out of order and be kept until there is a set of consecutive packets to be delivered to the application layer. Because the sizes of the send window and receive window are the same, all the packets in the send packet can arrive out of order and be stored until they can be delivered

Figure 23.32 *Send window for Selective-Repeat protocol*



VTUSYNC.IN

Figure 23.33 *Receive window for Selective-Repeat protocol*



Timer

Theoretically, Selective-Repeat uses one timer for each outstanding packet. When a timer expires, only the corresponding packet is resent.

Acknowledgments

In SR, an ackNo defines the sequence number of a single packet that is received safe and sound; there is no feedback for any other.

FSMs

Sender

The sender starts in the ready state, but later it can be in one of the two states: ready or blocking. The following shows the events and the corresponding actions in each state.

❑**Ready state.** Four events may occur in this case:

a. If a request comes from the application layer, the sender creates a packet with the sequence number set to Sn. A copy of the packet is stored, and the packet is sent. If the timer is not running, the sender starts the timer. The value of Sn is now incremented, Sn = (Sn + 1) modulo 2m. If the window is full, Sn = (Sf + Ssize) modulo 2m, the sender goes to the blocking state.

b. b. If an error-free ACK arrives with ackNo related to one of the outstanding packets, that packet is marked as acknowledged. If the ackNo = Sf , the window slides to the right until the Sf points to the first unacknowledged packet (all consecutive acknowledged packets are now outside the window). If there are outstanding packets, the timer is restarted; otherwise, the timer is stopped.

c. If a corrupted ACK or an error-free ACK with ackNo not related to an outstanding packet arrives, it is discarded. d. If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.

❑**Blocking state.** Three events may occur in this case:

a. If an error-free ACK arrives with ackNo related to one of the outstanding packets, that packet is marked as acknowledged. In addition, if the ackNo = Sf , the window is slid to the right until the Sf points to the first unacknowledged packet (all consecutive acknowledged packets are now outside the window). If the window has slid, the sender moves to the ready state.

b. If a corrupted ACK or an error-free ACK with the ackNo not related to outstanding packets arrives, the ACK is discarded.

c. If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.

Receiver

The receiver is always in the ready state. Three events may occur:

a. If an error-free packet with seqNo in the window arrives, the packet is stored and an ACK with ackNo = seqNo is sent. In addition, if the seqNo = Rn, then the packet and all previously arrived

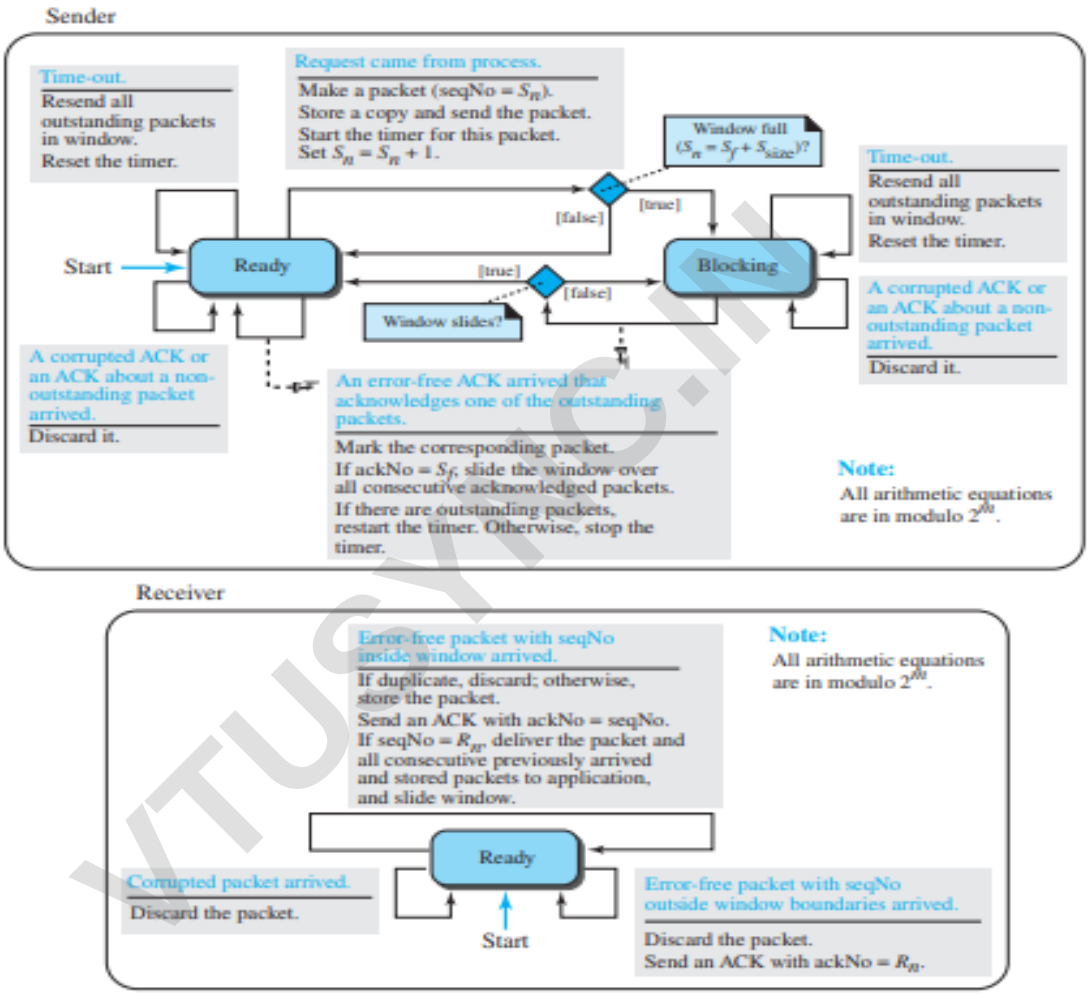
VTUSYNC.IN

consecutive packets are delivered to the application layer and the window slides so that the Rn points to the first empty slot.

b. If an error-free packet with seqNo outside the window arrives, the packet is discarded, but an ACK with ackNo = Rn is returned to the sender. This is needed to let the sender slide its window if some ACKs related to packets with seqNo < Rn were lost.

c. If a corrupted packet arrives, the packet is discarded.

Figure 23.34 *FSMs for SR protocol*

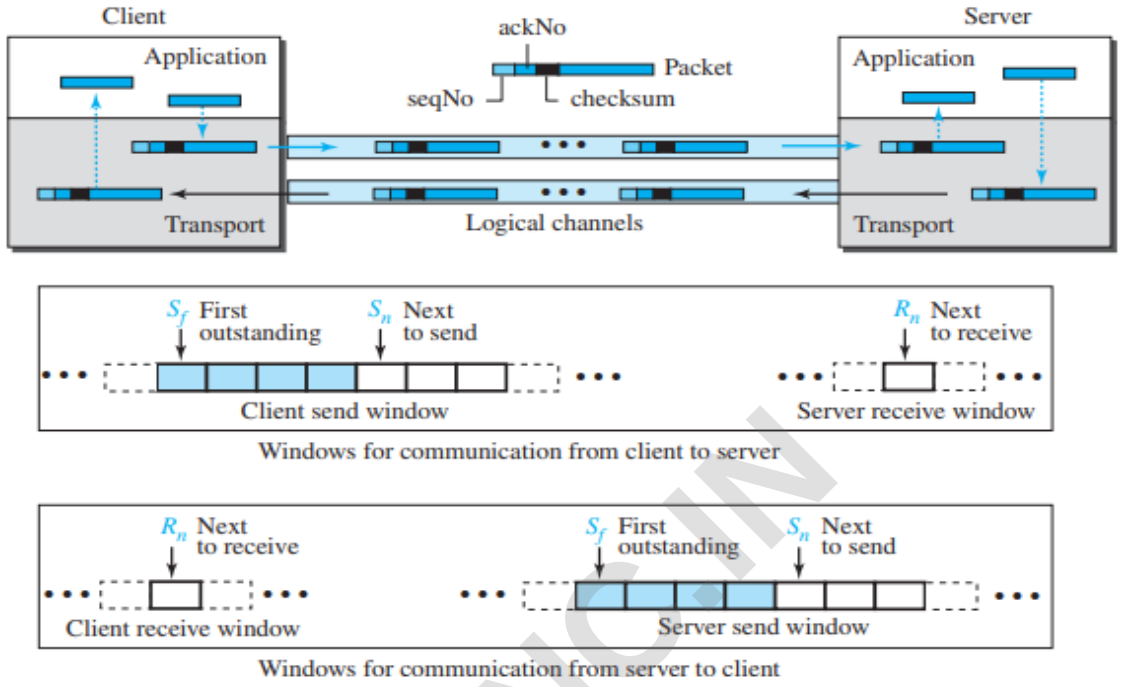


2.5 Bidirectional Protocols: Piggybacking

A technique called piggybacking is used to improve the efficiency of the bidirectional protocols. When a packet is carrying data from A to B, it can also carry acknowledgment feedback about arrived packets from B; when a packet is carrying data from B to A, it can also carry acknowledgment feedback about the arrived packets from A.

VTUSYLLABUS.IN

Figure 23.37 *Design of piggybacking in Go-Back-N*



3.1 Services

- **UDP:** UDP is an unreliable connectionless transport-layer protocol used for its simplicity and efficiency in applications where error control can be provided by the application-layer process.
- **TCP: TCP** is a reliable connection-oriented protocol that can be used in any application where reliability is important.
- **SCTP**: SCTP is a new transport-layer protocol that combines the features of UDP and TCP.

3.2 Port Numbers

Port numbers provide end-to-end addresses at the transport layer and allow multiplexing and demultiplexing at this layer, just as IP addresses do at the network layer.

VTUSYNC.IN

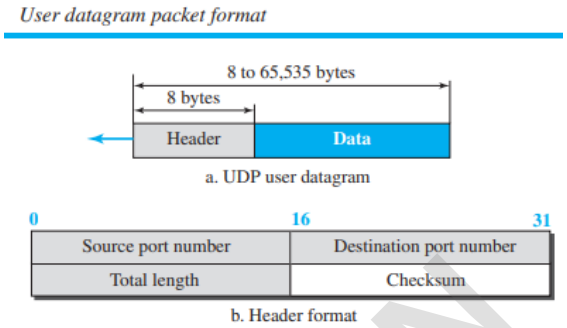
4. USER DATAGRAM PROTOCOL

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

4.1 User Datagram

- UDP packets, called user datagram's, 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 (explained later).



4.2 UDP Services

1. **Process-to-Process Communication:** UDP provides process-to-process communication using socket addresses, a combination of IP addresses and port numbers
2. **Connectionless Services:** As mentioned previously, 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 datagram's even if they are coming from the same source process and going to the same destination program.
3. **Flow Control**: 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.
4. **Checksum**: UDP checksum calculation 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 .
5. **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.
6. **Encapsulation and Decapsulation**: To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages.
7. **Queuing:** In UDP, queues are associated with ports. 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.
8. **Multiplexing and Demultiplexing**: In a host running a TCP/IP protocol suite, there is only one UDP but possibly several processes that may want to use the services of UDP. To handle this situation, UDP multiplexes and demultiplex

VTUSYNC.IN

4.3 UDP Applications

The following shows some typical applications that can benefit more from the services of UDP than from those of TCP.

❑ 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 FTP 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)

❑ UDP is normally used for interactive real-time applications that cannot tolerate uneven delay between sections of a received message

- ***Finer application-level control over what data is sent, and when.***
- ***No connection establishment***
- ***No connection state***
- ***Small packet header overhead:*** **The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of**

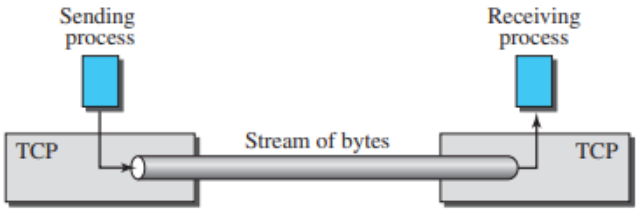
5. TRANSMISSION CONTROL PROTOCOL

- 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.
- To achieve reliability in TCP uses checksum (for error detection), retransmission of lost or corrupted packets, cumulative and selective acknowledgments, and timer

5.1 TCP Services

1. Process-to-Process Communication: TCP provides process-to-process communication using port numbers.

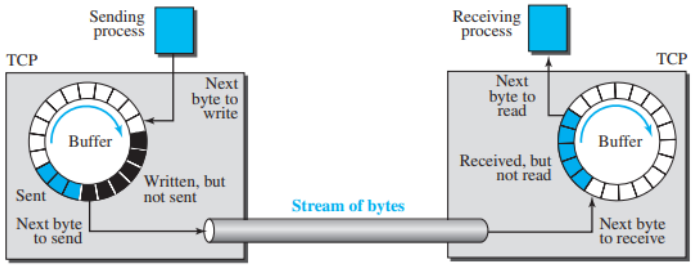
2. Stream Delivery Service: TCP allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes.



VTUSYNC.IN

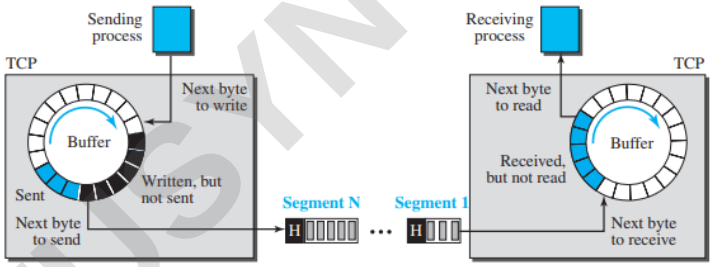
Sending and Receiving Buffers: Because the sending and the receiving processes may not necessarily write or read data at the same rate, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer, one for each direction.

Figure 24.5 *Sending and receiving buffers*



Segments: Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data. 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.

Figure 24.6 *TCP segments*



3. 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.

4. Multiplexing and Demultiplexing: TCP performs multiplexing at the sender and demultiplexing at the receiver. However, since TCP is a connection-oriented protocol, a connection needs to be established for each pair of processes.

5. 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:

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

6. Reliable Service TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in the section on error control.

5.2 TCP Features

VTUSYNC.IN

Numbering System

There are two fields, the sequence number and the acknowledgment number. These two fields refer to a byte number and not a segment number.

Byte Number: TCP numbers all data bytes (octets) that are transmitted in a connection. Numbering is independent in each direction. When TCP receives bytes of data from a process, TCP stores them in the sending buffer and numbers them. The numbering does not necessarily start from 0. Instead, TCP chooses an arbitrary number between 0 and 232 − 1 for the number of the first byte.

Sequence Number:After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent. The sequence number, in each direction, is defined as follows:

- ❖ The sequence number of the first segment is the ISN (initial sequence number), which is a random number.
- ❖ The sequence number of any other segment is the sequence number of the previous segment plus the number of bytes (real or imaginary) carried by the previous segment.

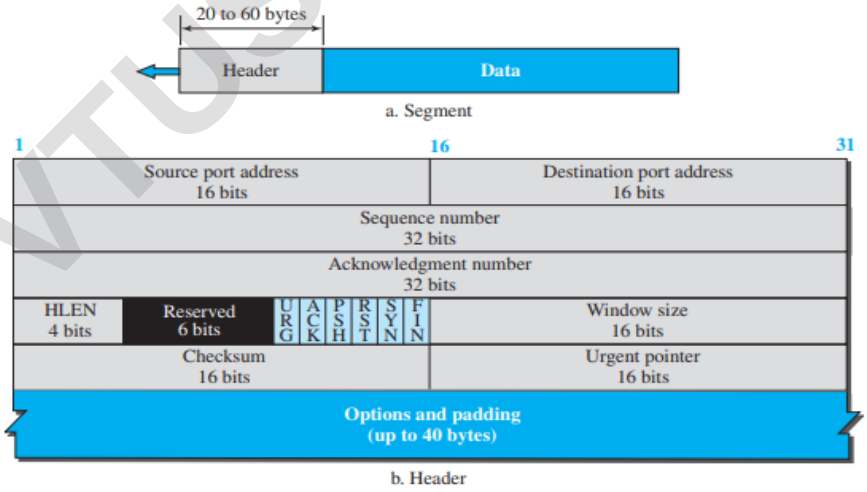
Acknowledgment Number: the acknowledgment number is cumulative, which means that the party takes the number of the last byte that it has received, safe and sound, adds 1 to it, and announces this sum as the acknowledgment number.

5.3 Segment

A packet in TCP is called a segment. The segment consists of a header of 20 to 60 bytes, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

VTUSYNC.IN

Figure 24.7 *TCP segment format*



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

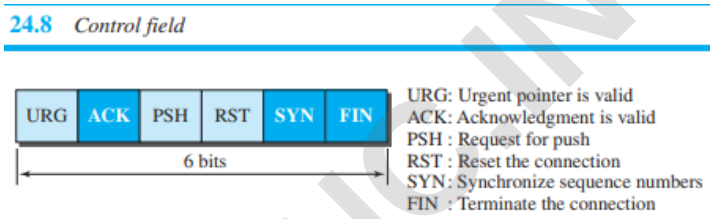
❑**Destination port address:** 16-bit field that defines the port number of the application program in the host that is receiving the segment.

❑**Sequence number:** This 32-bit field defines the number assigned to the first byte of data contained in this segment. 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.

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

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

❑**Control:**This field defines 6 different control bits or flags, as shown in Figure 24.8. One or more of these bits can be set at a time. These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP.



❑**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.

❑**Checksum:** This 16-bit field contains the checksum. The calculation of the checksum for TCP follows the same procedure as the one described for UDP. However, the use of the checksum in the UDP datagram is optional, whereas the use of the checksum for TCP is mandatory.

❑**Urgent pointer:** This 16-bit field, which is valid, only if the urgent flag is set, is used when the segment contains urgent data. It defines a value that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.

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

VTUSYNC.IN

5.4 TCP Connection

In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

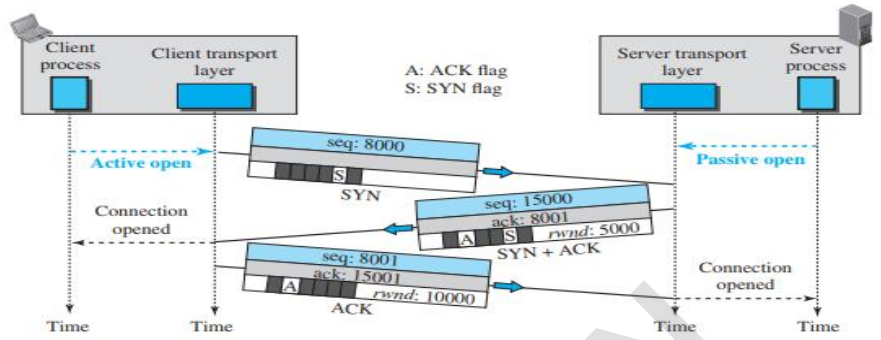
Phase1: Connection Establishment

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 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, as shown in Figure

Figure 24.10 *Connection establishment using three-way handshaking*



The three steps in this phase are as follows.

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).
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. First, it is a SYN segment for communication in the other direction. The server uses this segment to initialize a sequence number for numbering the bytes sent from the server to the client. The server also acknowledges the receipt of the SYN segment from the client by setting the ACK flag and displaying the next sequence number it expects to receive from the client.
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.

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 datagram's.

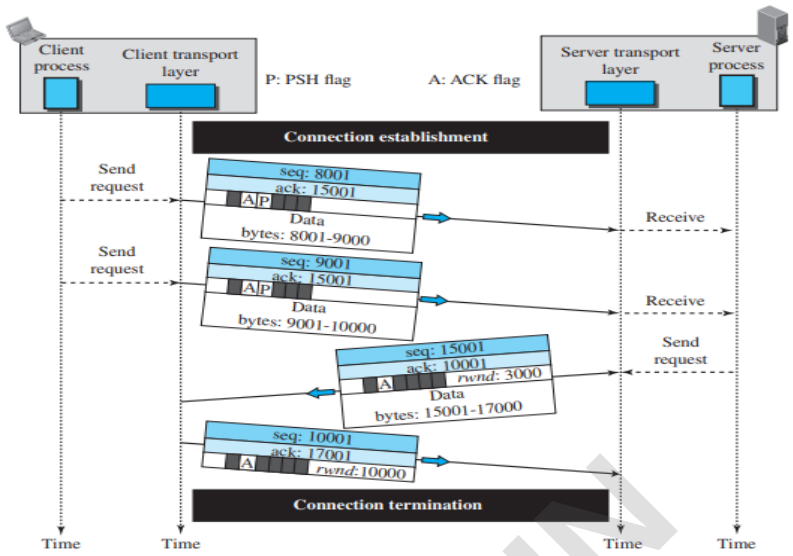
Phase2: Data Transfer

The client and server can send data and acknowledgments in both directions.

Pushing Data We saw that the sending TCP uses a buffer to store the stream of data coming from the sending application program. 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.

VTUSYNC.IN

Figure 24.11 *Data transfer*

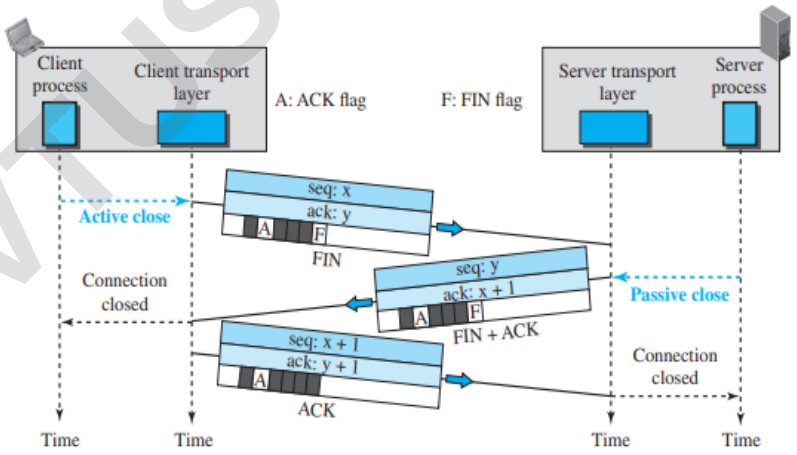


Urgent Data: CP urgent mode is a service by which the application program at the sender side marks some portion of the byte stream as needing special treatment by the application program at the receiver side.

Phase3: Connection Termination

Either of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client.

Figure 24.12 *Connection termination using three-way handshaking*



1. 1. 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.
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 announce the closing of the connection in the other direction.
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 one plus the sequence number received in the FIN segment from the server.

VTUSYNC.IN

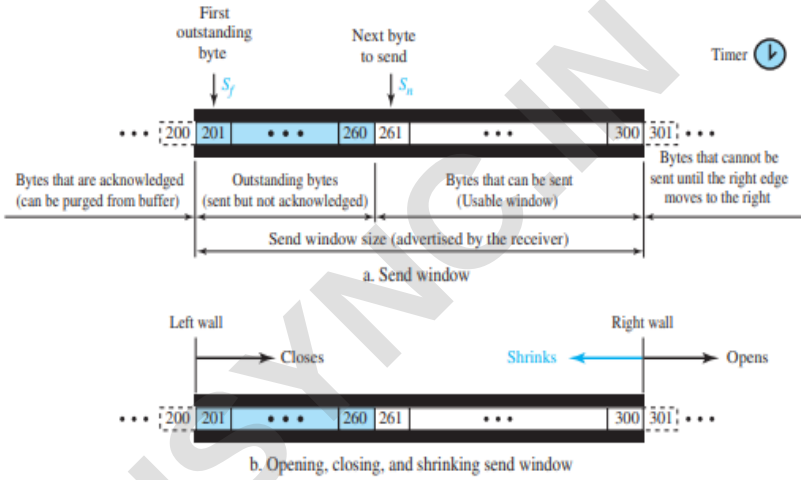
5.5 Windows in TCP

TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication.

Send Window

Figure 24.17 shows an example of a send window. The window size is 100 bytes, but later we see that the send window size is dictated by the receiver (flow control) and the congestion in the underlying network (congestion control). The figure shows how a send window opens, closes, or shrinks.

Figure 24.17 *Send window in TCP*



The send window in TCP is similar to the one used with the Selective-Repeat protocol, but with some differences:

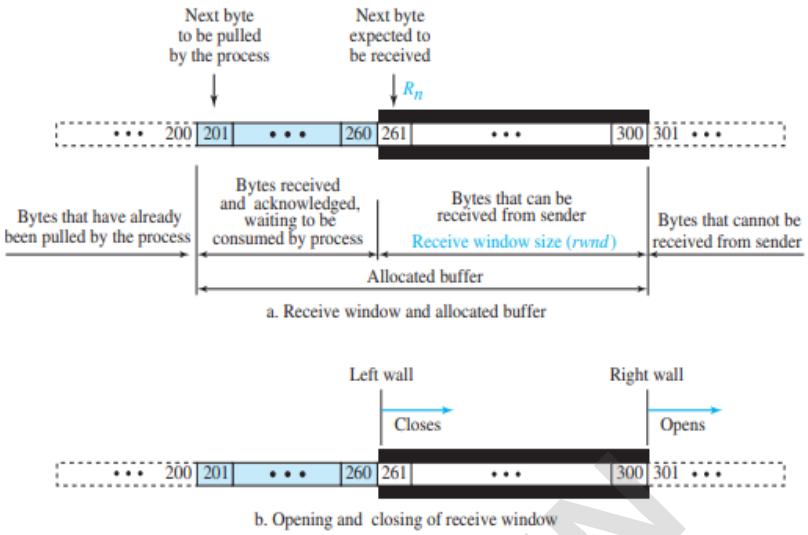
1. One difference is the nature of entities related to the window. The window size in SR is the number of packets, but the window size in TCP is the number of bytes. Although actual transmission in TCP occurs segment by segment, the variables that control the window are expressed in bytes.
2. The second difference is that, in some implementations, TCP can store data received from the process and send them later, but we assume that the sending TCP is capable of sending segments of data as soon as it receives them from its process.
3. Another difference is the number of timers. The theoretical Selective-Repeat protocol may use several timers for each packet sent, but as mentioned before, the TCP protocol uses only one timer.

Receive Window

Figure 24.18 shows an example of a receive window. The window size is 100 bytes. The figure also shows how the receive window opens and closes; in practice, the window should never shrink.

VTUSYNC.IN

Figure 24.18 *Receive window in TCP*



There are two differences between the receive window in TCP and the one we used for SR.

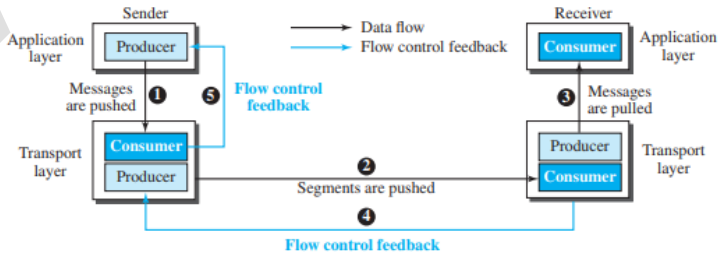
1. The first difference is that TCP allows the receiving process to pull data at its own pace.
2. The second difference is the way acknowledgments are used in the TCP protocol.

5.6 Flow Control

Flow control balances the rate a producer creates data with the rate a consumer can use the data. TCP separates flow control from error control.

Figure 24.19 shows unidirectional data transfer between a sender and a receiver; bidirectional data transfer can be deduced from the unidirectional process.

Figure 24.19 *Data flow and flow control feedbacks in TCP*



- The figure shows that data travel from the sending process down to the sending TCP, from the sending TCP to the receiving TCP, and from the receiving TCP up to the receiving process (paths 1, 2, and 3).
- Flow control feedbacks, however, are traveling from the receiving TCP to the sending TCP and from the sending TCP up to the sending process (paths 4 and 5).
- Most implementations of TCP do not provide flow control feedback from the receiving process to the receiving TCP; they let the receiving process pull data from the receiving TCP whenever

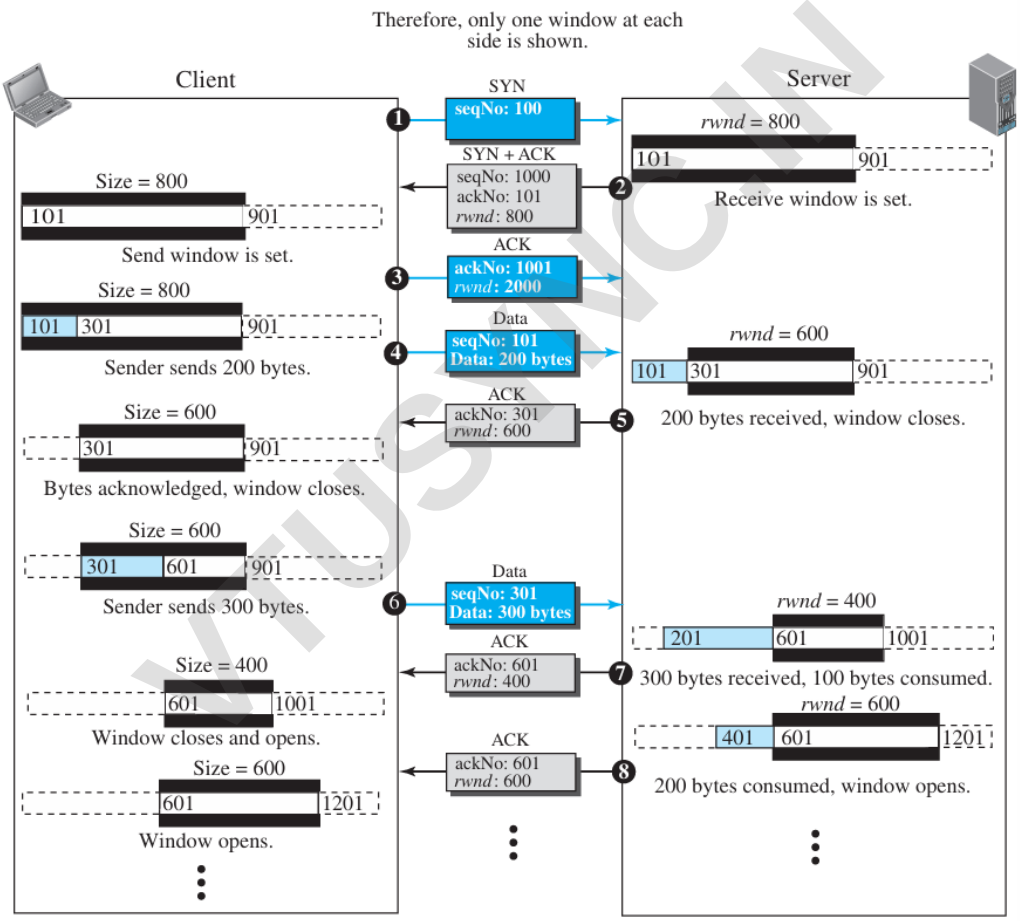
VTUSYNC.IN

it is ready to do so. In other words, the receiving TCP controls the sending TCP; the sending TCP controls the sending process.

- Flow control feedback from the sending TCP to the sending process (path 5) is achieved through simple rejection of data by the sending TCP when its window is full. This means that our discussion of flow control concentrates on the feedback sent from the receiving TCP to the sending TCP (path 4).

Opening, Closing and shrinking Windows

- The opening, closing, and shrinking of the send window is controlled by the receiver.
- new ackNo + new rwnd > last ackNo + last rwnd , send window shrinks



Window Shutdown

- The receiver can temporarily shut down the window by sending a rwnd of 0.
- This can happen if for some reason the receiver does not want to receive any data from the sender for a while.
- In this case, the sender does not actually shrink the size of the window, but stops sending data until a new advertisement has arrived.

- The sender can always send a segment with 1 byte of data. This is called probing and is used to prevent a deadlock.

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.
- 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.**

Nagle's algorithm – Sender side

- The solution is to prevent the sending TCP from sending the data byte by byte. The sending TCP must be forced to wait and collect data to send in a larger block. How long should the sending TCP wait? 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.

Clark's solution - Receiver Side

- 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.
- 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.
- Delayed acknowledgment also has another advantage: it reduces traffic. The receiver does not have to acknowledge each segment. However, there also is a disadvantage in that the delayed

VTUSYNC.IN

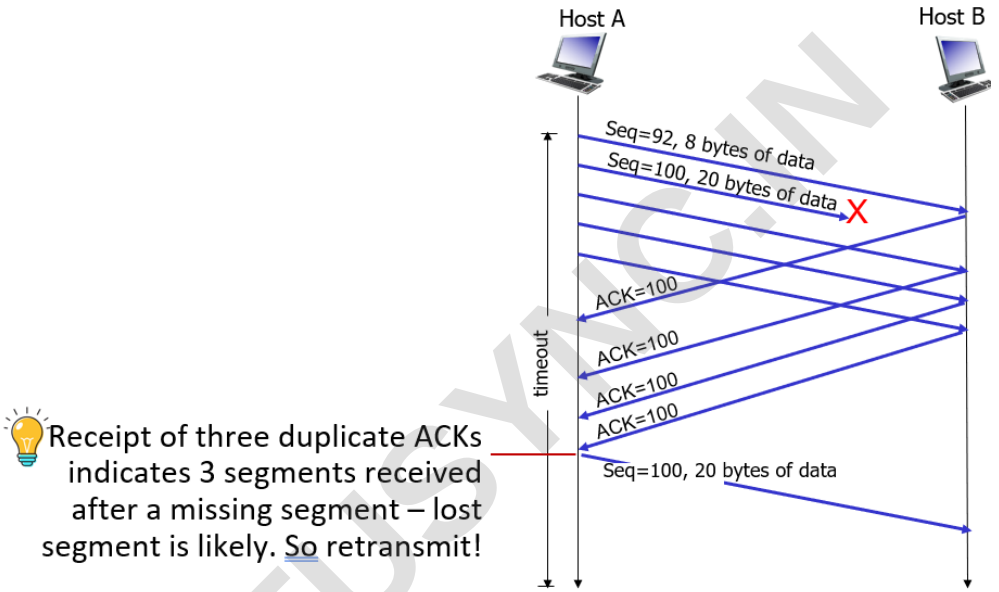
acknowledgment may result in the sender unnecessarily retransmitting the unacknowledged segments.

- The protocol balances the advantages and disadvantages. It now defines that the acknowledgment should not be delayed by more than 500 ms.

TCP fast retransmit

If sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout



5.8 TCP Congestion Control

- TCP use end to end congestion control as IP layer does not provide explicit feedback to the end system regarding network congestion.
- The approach taken by TCP is to have each sender limit the rate at which it sends the traffic into its connection as a function of perceived network congestion.
- If a TCP sender perceives that there is little or no congestion on the path between itself and destination, then the TCP sender increases its sender rate increases its send rate, if there is congestion, then the sender reduces the send rate.
- **Sender keep tracks of additional variable – cwnd (Congestion Window)**
- **Congestion Window (cwnd)** is a TCP state variable that limits the amount of data the TCP can send into the network before receiving an ACK.

VTUSYNC.IN

- The **Receiver Window (rwnd)** is a variable that advertises the amount of data that the destination side can receive.
- Together, the two variables are used to regulate data flow in TCP connections, minimize congestion, and improve network performance. The amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd, that is:

$$LastByteSent - LastByteAcked \leq \min\{cwnd, rwnd\}$$

TCP maintains the sending rate by following the principles namely

1. Lost segment implies congestion and hence the **sender's rate** should be **decreased** when a **segment is lost.**
2. An acknowledged segment indicates that the network is delivering the sender's segment to the receiver and hence the **sender rate can be increased** when an **ACK arrives** for the previously **unacknowledged frame.**
3. Bandwidth Probing: ***sstrsh(slow start threshold)***

Congestion Detection:

Signs of Congestion in TCP:

- **Time-out**: If an acknowledgment (ACK) is not received before the time-out, the sender assumes segment loss due to severe congestion.
- **Three Duplicate ACKs**: Receiving four ACKs with the same acknowledgment number indicates a missing segment, signalling mild or recovering congestion.

Severity of Congestion:

- Time-out indicates strong congestion, as it implies multiple segments may be lost.
- Three duplicate ACKs suggest weak congestion, as most segments are still being delivered.

TCP Versions and Congestion Handling:

- **Tahoe TCP**: Treats both time-out and duplicate ACKs as equally severe.
- **Reno TCP**: Differentiates between time-out (strong congestion) and duplicate ACKs (weak congestion).

TCP Congestion control includes:

1. Slow start.
2. Congestion Avoidance.
3. Fast recovery

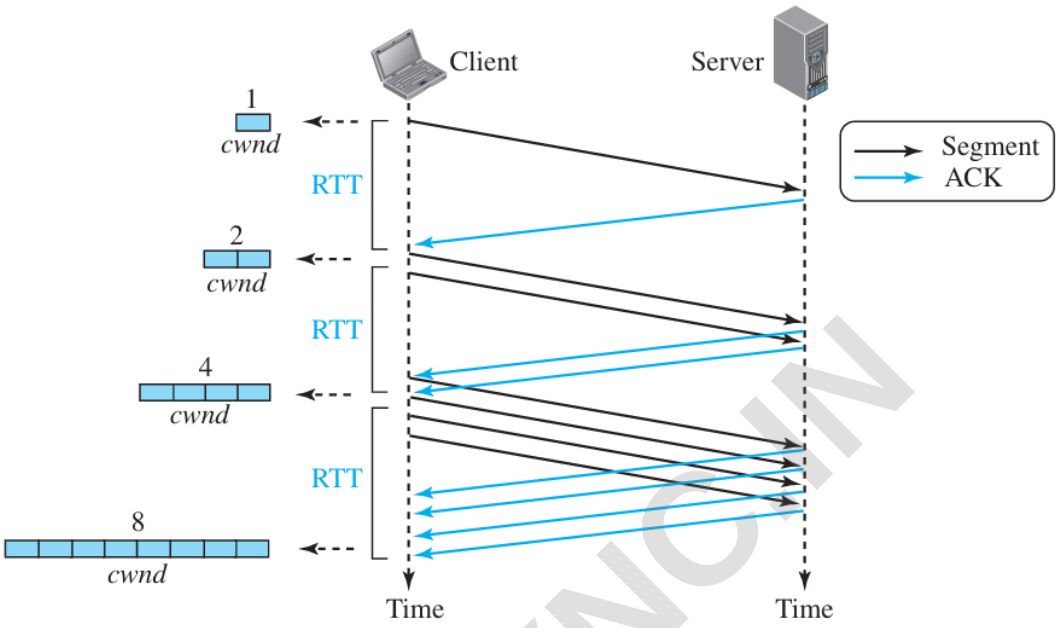
1. TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS

VTUSYNC.IN

- double **cwnd** every RTT
- done by incrementing **cwnd** for every ACK received

Figure 24.29 *Slow start, exponential increase*



the size of the congestion window in this algorithm is a function of the number of ACKs arrived and can be determined as follows.

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

Start	$\rightarrow$	$cwnd = 1 \rightarrow 2^0$
After 1 RTT	$\rightarrow$	$cwnd = cwnd + 1 = 1 + 1 = 2 \rightarrow 2^1$
After 2 RTT	$\rightarrow$	$cwnd = cwnd + 2 = 2 + 2 = 4 \rightarrow 2^2$
After 3 RTT	$\rightarrow$	$cwnd = cwnd + 4 = 4 + 4 = 8 \rightarrow 2^3$

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

When should this exponential growth ends?

1. Lost segment (Time out): Ssthresh = cwnd/2;
2. when **cwnd >=ssthresh**, Then TCP enters Congestion Avoidance mode.
3. When three DUPACK are received:

Enter Fast recovery state: v**Cwnd= sstresh+3MSS**

Sstresh= cwnd/2

VTUSYNOPSIS

2. Congestion Avoidance: Additive Increase

- On entry on congestion state, value of cwnd will be half.
- TCP now adopts a more conservative approach

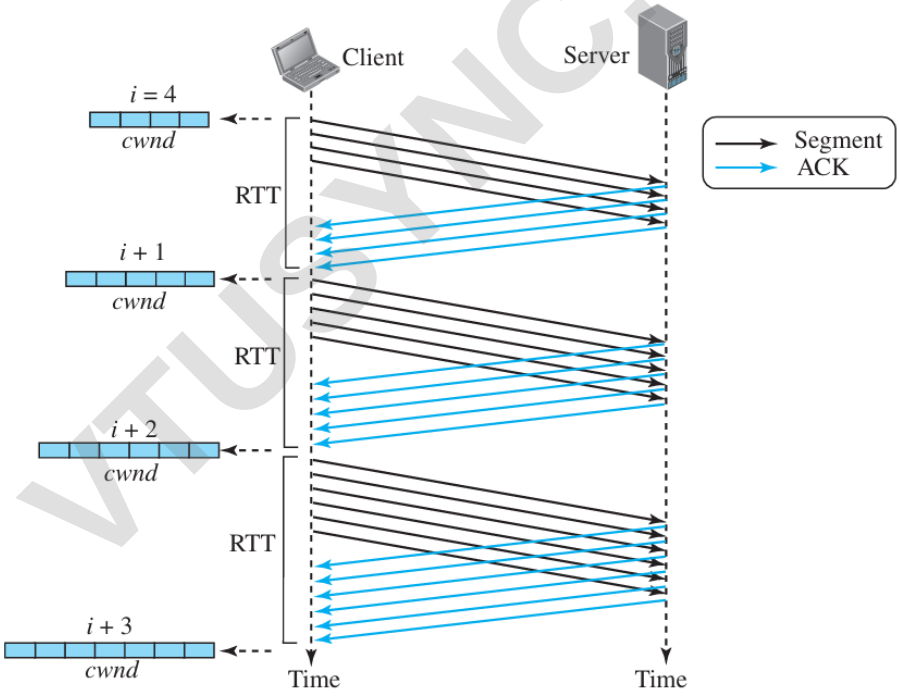
$$Cwnd=cwnd + MSS[MSS/cwnd]$$

This increases cwnd by 1/10 MSS for each iteration

- Congestion avoidance linear increase ends when timeout occurs or if 3 ACK duplicative is received.

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 shows the idea.

Figure 24.30 *Congestion avoidance, additive increase*



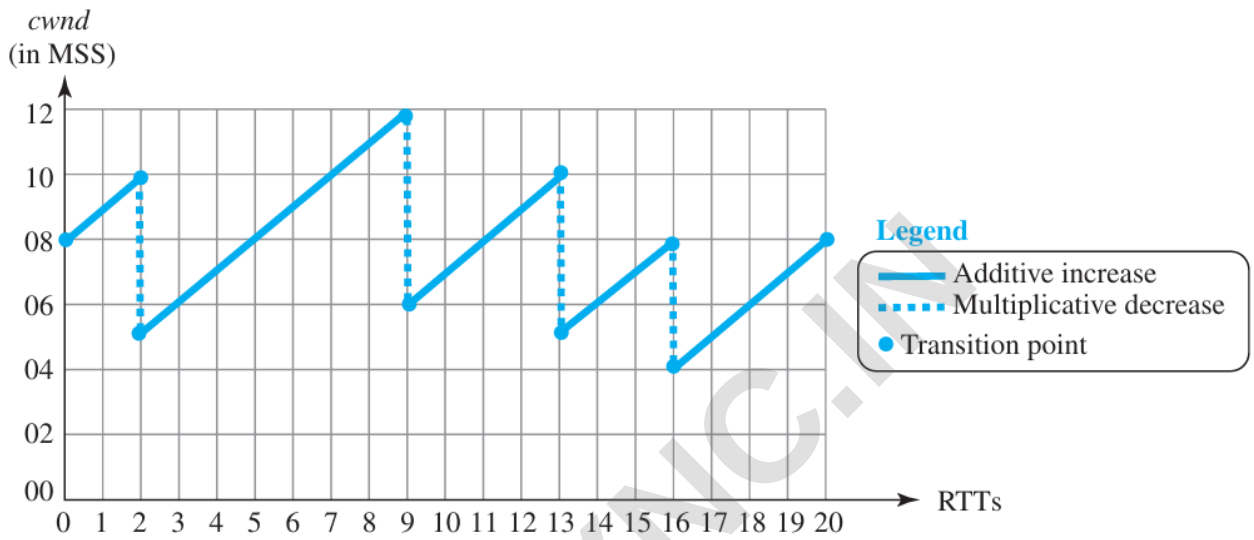
VTUSYNC.IN

Additive Increase, Multiplicative Decrease

It has been observed that, in this version, most of the time the congestion is detected and taken care of by observing the three duplicate ACKs. Even if there are some time-out events, TCP recovers from them by aggressive exponential growth. In other words, in a long TCP connection, if we ignore the slow-start states and short exponential growth during fast recovery, the TCP congestion window is cwnd = cwnd + (1 / cwnd) when an ACK arrives (congestion avoidance), and cwnd = cwnd / 2 when congestion is detected, as though SS does not exist and the length of FR is reduced to zero. The first is called additive increase; the

second is called multiplicative decrease. This means that the congestion window size, after it passes the initial slow-start state, follows a saw tooth pattern called additive increase, multiplicative decrease (AIMD), as shown in Figure 24.35.

Figure 24.35 *Additive increase, multiplicative decrease (AIMD)*



VTUSYNC.IN

3. **Fast recovery**

- In fast recovery , the value of cwnd is increased by 1MSS for every duplicate ACK.
- TCP enters back :
  - **To Congestion Avoidance**: on new ACK
  - **To Slow start**: if Time out occurs.

TCP Throughput

- The throughput for TCP, which is based on the congestion window behavior, can be easily found if the cwnd is a constant (flat line) function of RTT.
- The throughput with this unrealistic assumption is throughput = cwnd / RTT. In this assumption, TCP sends a cwnd bytes of data and receives acknowledgement for them in RTT time.
- The behavior of TCP is not a flat line; it is like saw teeth, with many minimum and maximum values. If each tooth were exactly the same, we could say that the throughput = [(maximum + minimum) / 2] / RTT.

$$\text{throughput} = (0.75)\, W_{max} / RTT$$

in which W_{max} is the average of window sizes when the congestion occurs.

Example 24.11

If MSS = 10 KB (kilobytes) and RTT = 100 ms in Figure 24.35, we can calculate the throughput as shown below.

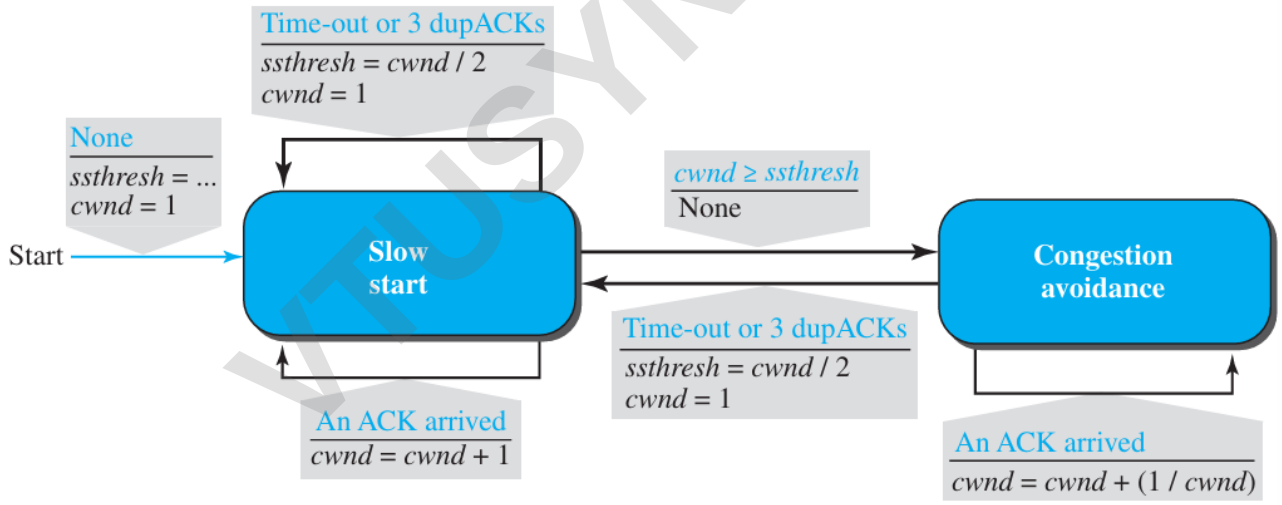
$W_{max} = (10 + 12 + 10 + 8 + 8) / 5 = 9.6$ MSS

Throughput = $(0.75\, W_{max} / RTT) = 0.75 \times 960$ kbps / 100 ms = 7.2 Mbps

Taho TCP

The early TCP, known as Taho TCP, used only two different algorithms in their conges tion policy: slow start and congestion avoidance.

Figure 24.31 *FSM for Taho TCP*



Congestion Detection in Tahoe TCP:

- Treats both **time-out** and **three duplicate ACKs** equally as signs of congestion.

Slow-Start Algorithm:

- TCP starts with cwnd = 1 MSS and increases the congestion window size aggressively (exponentially) with each ACK received until congestion is detected or the threshold (ssthresh) is reached.

Reaction to Congestion:

VTUSYNC.IN

- On detecting congestion (time-out or duplicate ACKs), TCP:
  - Resets cwnd to 1 MSS.
  - Adjusts ssthresh to **half the current cwnd**.
  - Restarts the **slow-start algorithm**.

Congestion Avoidance State:

- When cwnd reaches ssthresh without congestion, TCP switches to a **congestion avoidance state**.
- In this state, cwnd grows additively, increasing by 1 MSS for every full window of ACKs received (e.g., if cwnd = 5 MSS, it requires 5 ACKs to increase cwnd to 6 MSS).

Dynamic Adjustment of Threshold:

- ssthresh is continuously adjusted based on congestion events.
- It can increase or decrease depending on the current state of the congestion window. For instance, if congestion occurs when cwnd = 20 MSS, ssthresh will adjust to 10 MSS (half of cwnd).

Growth Continuity:

- Additive growth of cwnd continues until the end of data transfer unless further congestion is detected, which resets the process.

VTUSYNC.IN

Figure 24.32 *Example 24.9*

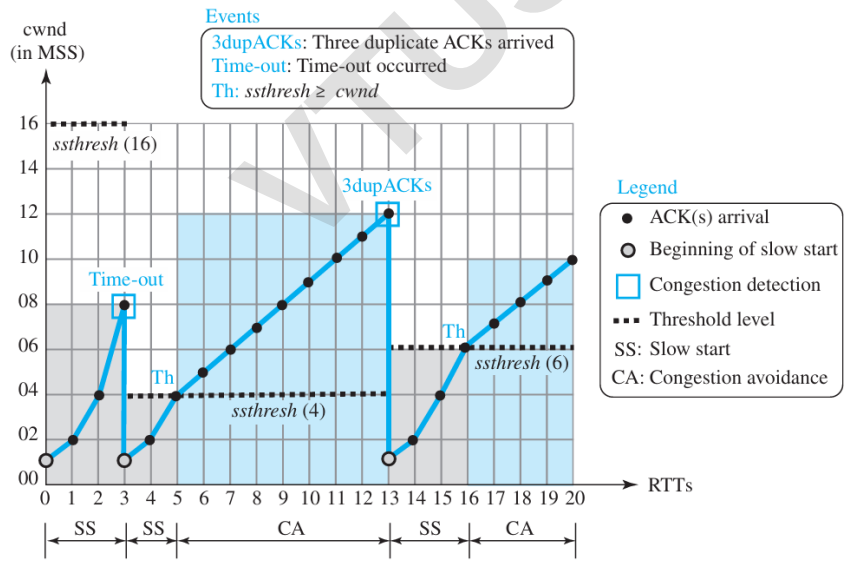
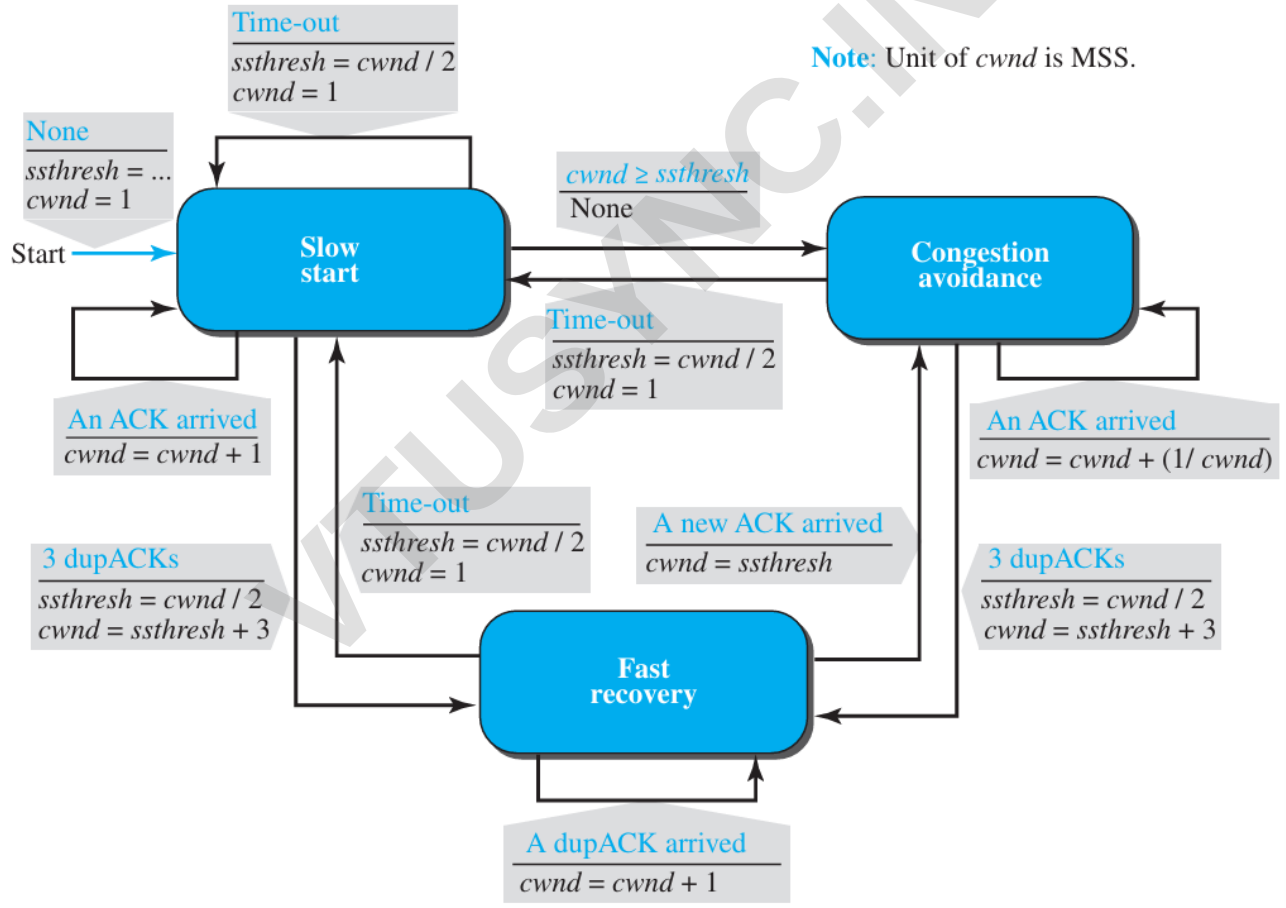


Figure 24.32 shows an example of congestion control in a Taho TCP. TCP starts data transfer and sets the ssthresh variable to an ambitious value of 16 MSS. TCP begins at the slow-start (SS) state with the cwnd = 1. The congestion window grows exponentially, but a time-out occurs after the third RTT (before reaching the threshold). TCP assumes that there is congestion in the net work. It immediately sets the new ssthresh =

4 MSS (half of the current cwnd, which is 8) and begins a new slow-start (SA) state with cwnd = 1 MSS. The congestion window grows exponen tially until it reaches the newly set threshold. TCP now moves to the congestion-avoidance (CA) state and the congestion window grows additively until it reaches cwnd = 12 MSS. At this moment, three duplicate ACKs arrive, another indication of congestion in the network. TCP again halves the value of ssthresh to 6 MSS and begins a new slow-start (SS) state. The expo nential growth of the cwnd continues. After RTT 15, the size of cwnd is 4 MSS. After sending four segments and receiving only two ACKs, the size of the window reaches the ssthresh (6) and TCP moves to the congestion-avoidance state. The data transfer now continues in the congestion avoidance (CA) state until the connection is terminated after RTT 20.

Reno TCP

Figure 24.33 *FSM for Reno TCP*



Reno TCP introduces a **fast-recovery state** to handle congestion more efficiently.

Differentiated Congestion Signals:

- **Time-out**: Treated as a severe congestion signal. TCP resets to the **slow-start state**, restarting with cwnd = 1 MSS.

- **Three Duplicate ACKs**: Treated as a less severe congestion signal. TCP moves to the **fast-recovery state**.

Behaviour in Fast-Recovery:

- **Initial cwnd**: Starts with ssthresh + 3 MSS (not 1 MSS like in slow start).
- **Duplicate ACKs**: TCP remains in this state and grows cwnd exponentially.
- **Time-out**: Indicates serious congestion, transitioning TCP back to **slow-start**.
- **New ACK**: Indicates recovery; TCP transitions to **congestion avoidance** and resets cwnd to ssthresh.

Fast-Recovery Transition:

- Fast-recovery bridges slow start and congestion avoidance, allowing quicker recovery while still addressing mild congestion.
- Reno TCP optimizes throughput by treating mild and severe congestion events differently, reducing unnecessary resets to slow start.

Figure 24.34 *Example 24.10*

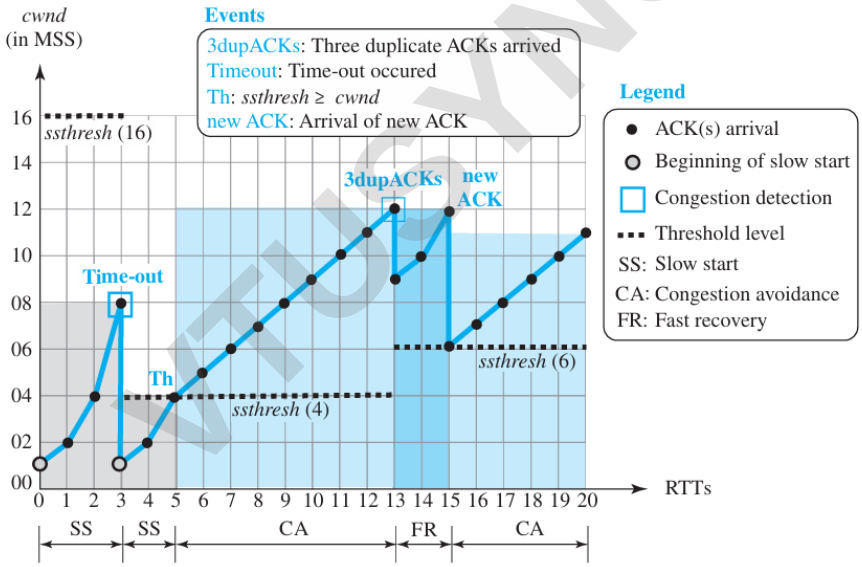


Figure 24.34 shows the same situation as Figure 24.32, but in Reno TCP. The changes in the con gestion window are the same until RTT 13 when three duplicate ACKs arrive. At this moment, Reno TCP drops the ssthresh to 6 MSS (same as Taho TCP), but it sets the cwnd to a much higher value (ssthresh + 3 = 9 MSS) instead of 1 MSS. Reno TCP now moves to the fast recov ery state. We assume that two more duplicate ACKs arrive until RTT 15, where cwnd grows exponentially. In this moment, a new ACK (not duplicate) arrives that announces the receipt of the lost segment. Reno TCP now moves to the congestion-avoidance state, but first deflates the congestion window to 6 MSS (the ssthresh value) as though ignoring the whole fast-recovery state and moving back to the previous track.

VTUSYNC.IN