

## BCS502 COMPUTER NETWORKS

### Module-4: Transport Layer

**Ch. 23. Introduction to Transport Layer:** Introduction, Transport-Layer Protocols

**Ch. 24. Transport-Layer Protocols:** Services, Features, Segments, TCP connections, Flow control, Error control, Congestion control.

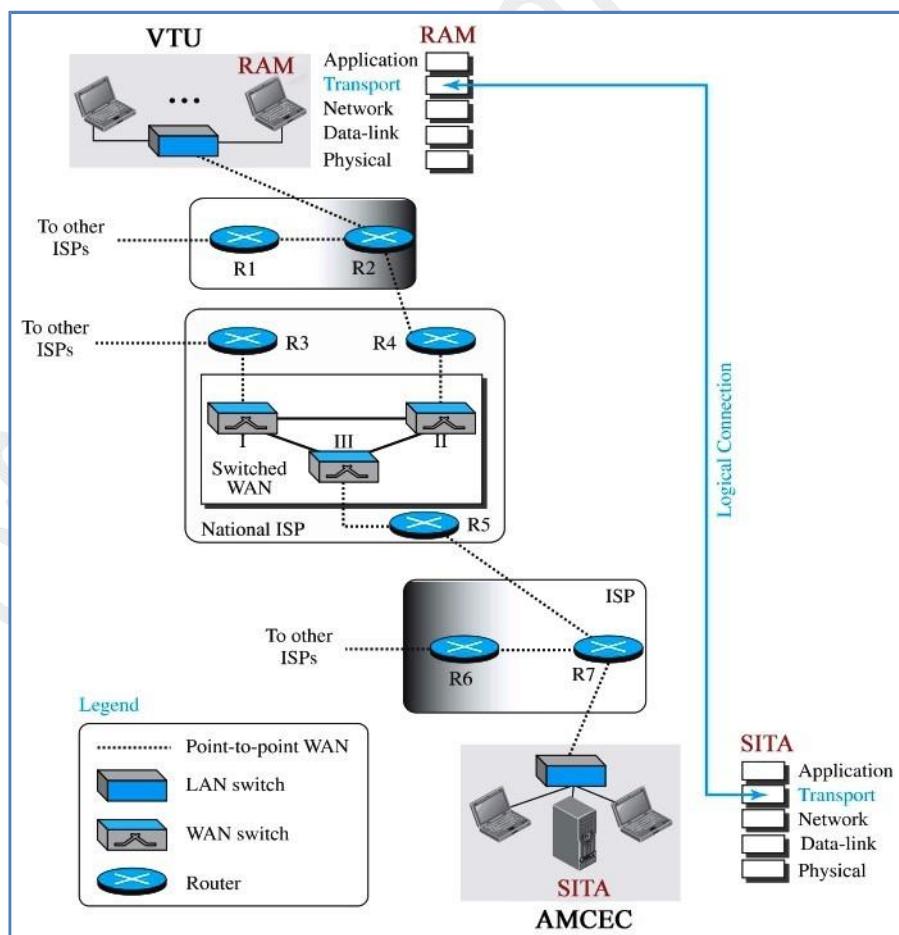
**Textbook:** Ch. 23.1, 23.2, 24.1 – 24.3.4, 24.3.6 - 24.3.9.

### Chapter 23: Introduction to Transport Layer [P-730]

#### Introduction, Transport-Layer Protocols

##### Introduction to Transport-Layer

- The transport layer is located between the application layer and the network layer. It provides **a process-to-process communication between two application layers**, one at the local host and the other at the remote host.
- The transport layer is the heart of the TCP/IP protocol suite; it is the **end-to-end logical vehicle for transferring data from one point to another in the Internet**.
- Communication is provided using a logical connection, which means that the two application layers, which can be located in different parts of the globe, assume that there is an imaginary direct connection through which they can send and receive messages. Figure shows the idea behind this logical connection.

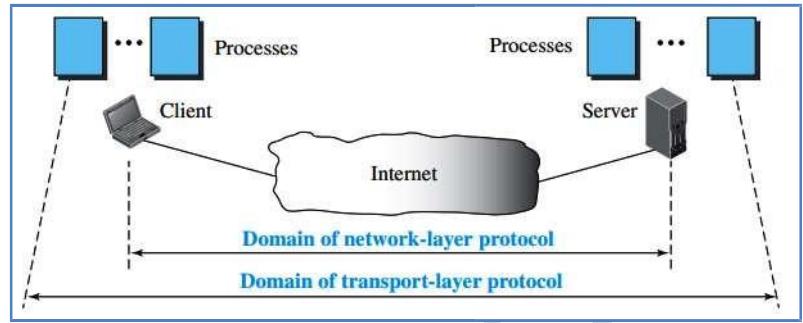


#### Transport-Layer Services

- The transport layer is responsible for providing services to the application layer; it receives services from the network layer.

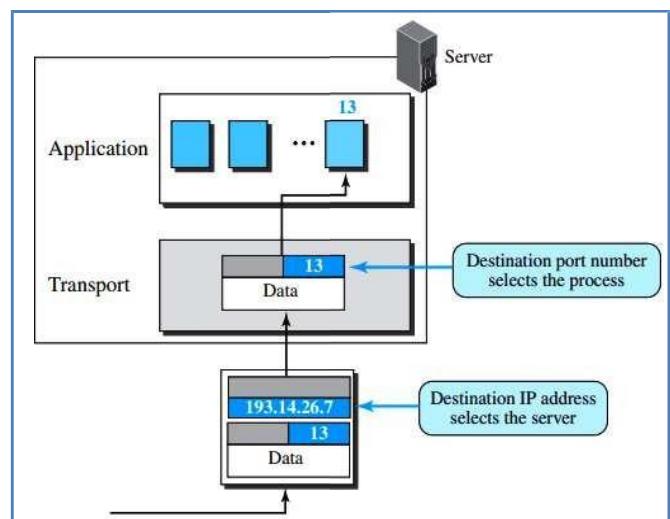
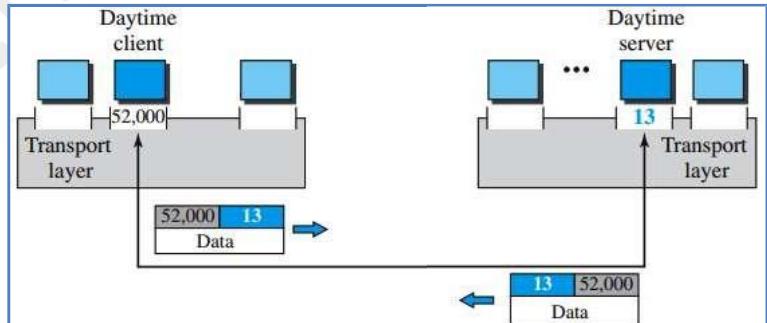
### Process-to-Process Communication

- The first duty of a transport-layer protocol is to provide process-to-process communication.
- A process is an application-layer entity (running program) that uses the services of the transport layer. We need to understand the difference between host-to-host communication and process-to-process communication.
- The network layer is responsible for communication at the computer level (host-to-host communication). A network-layer protocol can deliver the message only to the destination computer. However, this is an incomplete delivery. The message still needs to be handed to the correct process. This is where a transport-layer protocol takes over. A transport-layer protocol is responsible for delivery of the message to the appropriate process.



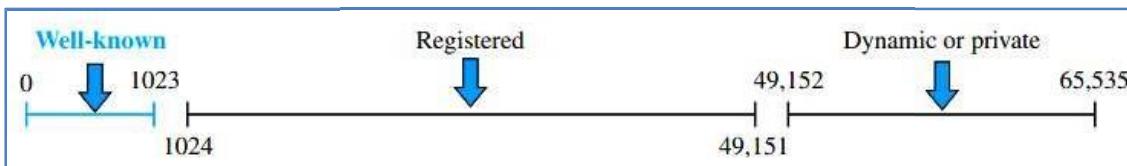
### Addressing: Port Numbers

- Although there are a few ways to achieve process-to-process communication, the most common is through the client-server paradigm. A process on the local host, called a client, needs services from a process usually on the remote host, called a server.
- Operating systems today support both multiuser and multiprogramming environments. A remote computer can run several server programs at the same time, just as several local computers can run one or more client programs at the same time.
- For communication, we must define the local host, local process, remote host, and remote process. The local host and the remote host are defined using IP addresses. To define the processes, we need second identifiers, called port numbers. In the TCP/IP protocol suite, the port numbers are integers between 0 and 65,535 (16 bits).
- The client program defines itself with a port number, called the e ephemeral port number. The word ephemeral means “short-lived” and is used because the life of a client is normally short. An ephemeral port number is recommended to be greater than 1023 for some client/server programs to work properly.



### ICANN Ranges

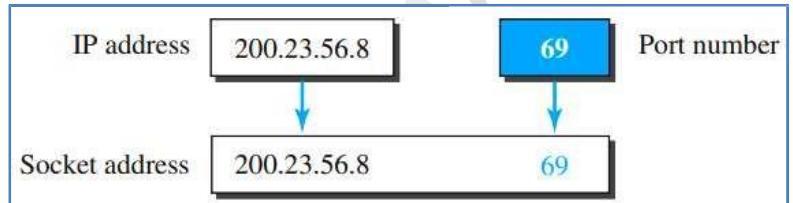
- ICANN has divided the port numbers into three ranges: well-known, registered, and dynamic.



- Well-known ports. The ports ranging from 0 to 1023 are assigned and controlled by ICANN. These are the well-known ports.
- Registered ports. The ports ranging from 1024 to 49,151 are not assigned or controlled by ICANN. They can only be registered with ICANN to prevent duplication.
- Dynamic (or private) ports. The ports ranging from 49,152 to 65,535 are neither controlled nor registered. They can be used as temporary or private port numbers.

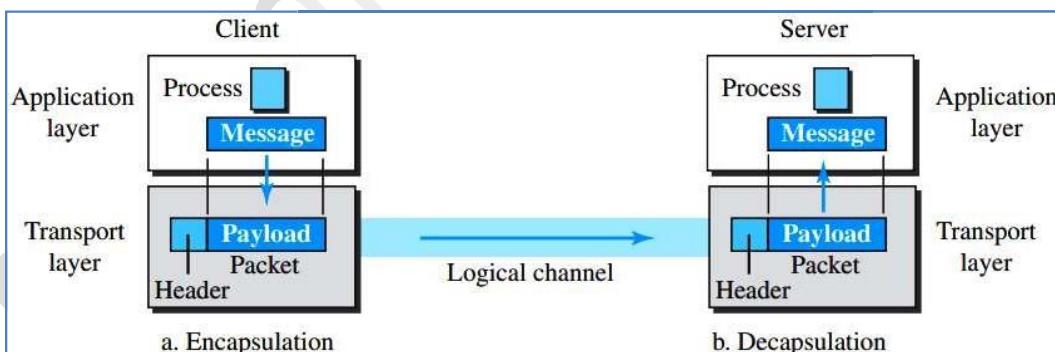
### Socket Addresses

- A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a socket address. The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely (see Figure).
- To use the services of the transport layer in the Internet, we need a pair of socket addresses: the client socket address and the server socket address. These four pieces of information are part of the network-layer packet header and the transport-layer packet header. The first header contains the IP addresses; the second header contains the port numbers.



### Encapsulation and Decapsulation

- To send a message from one process to another, the transport-layer protocol encapsulates and decapsulates messages (Figure).

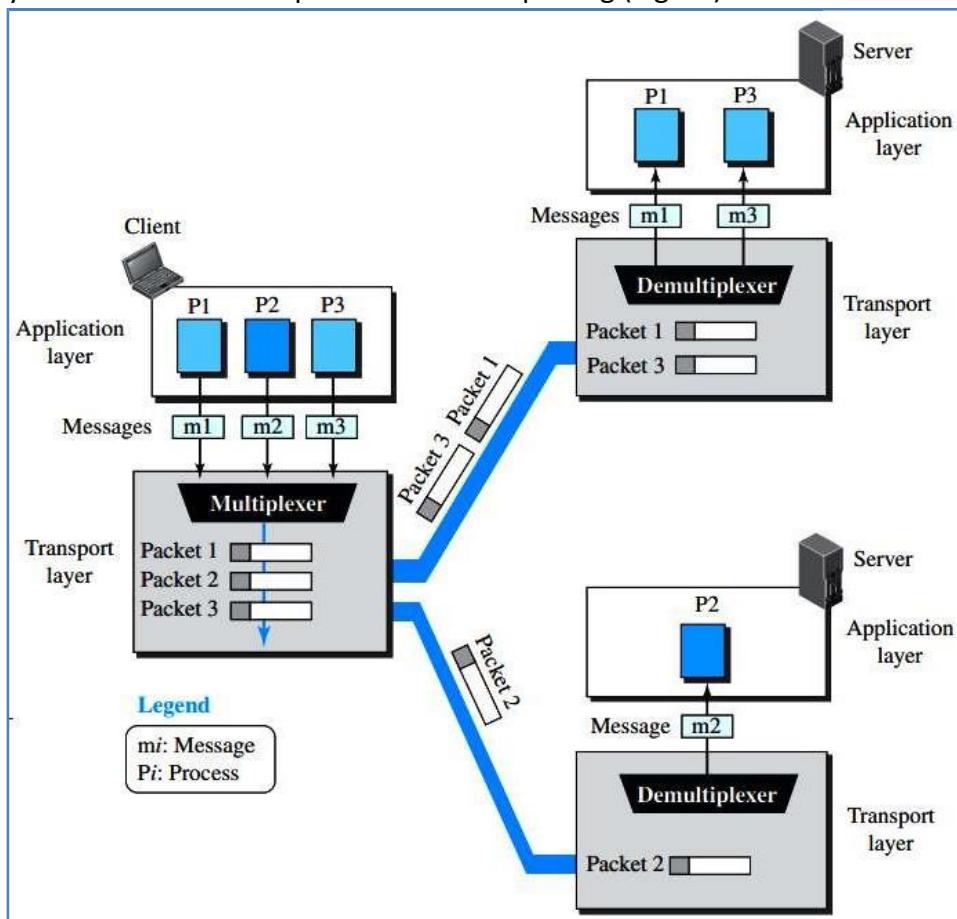


- Encapsulation happens at the sender site. When a process has a message to send, it passes the message to the transport layer along with a pair of socket addresses and some other pieces of information, which depend on the transport-layer protocol.
- The transport layer receives the data and adds the transport-layer header. The packets at the transport layer in the Internet are called user datagrams, segments, or packets, depending on what transport-layer protocol we use. We refer to transport-layer payloads as packets.
- Decapsulation happens at the receiver site. When the message arrives at the destination transport layer, the header is dropped and the transport layer delivers the message to the process running at the application layer.

- The sender socket address is passed to the process in case it needs to respond to the message received.

### Multiplexing and Demultiplexing

- Whenever an entity accepts items from more than one source, this is referred to as multiplexing (many to one); whenever an entity delivers items to more than one source, this is referred to as demultiplexing (one to many). The transport layer at the source performs multiplexing; the transport layer at the destination performs demultiplexing (Figure).

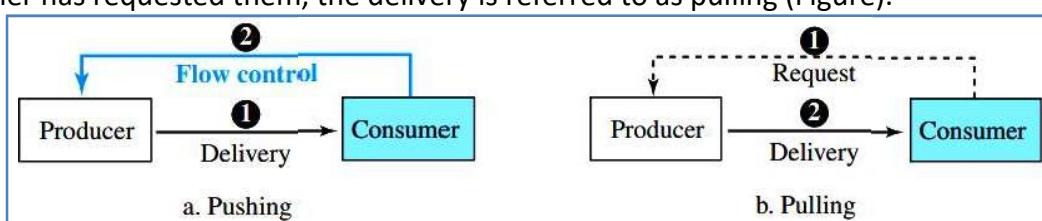


### Flow Control

- Whenever an entity produces items and another entity consumes them, there should be a balance between production and consumption rates. If the items are produced faster than they can be consumed, the consumer can be overwhelmed and may need to discard some items. If the items are produced more slowly than they can be consumed, the consumer must wait, and the system becomes less efficient. **Flow control is related to the first issue. We need to prevent losing the data items at the consumer site.**

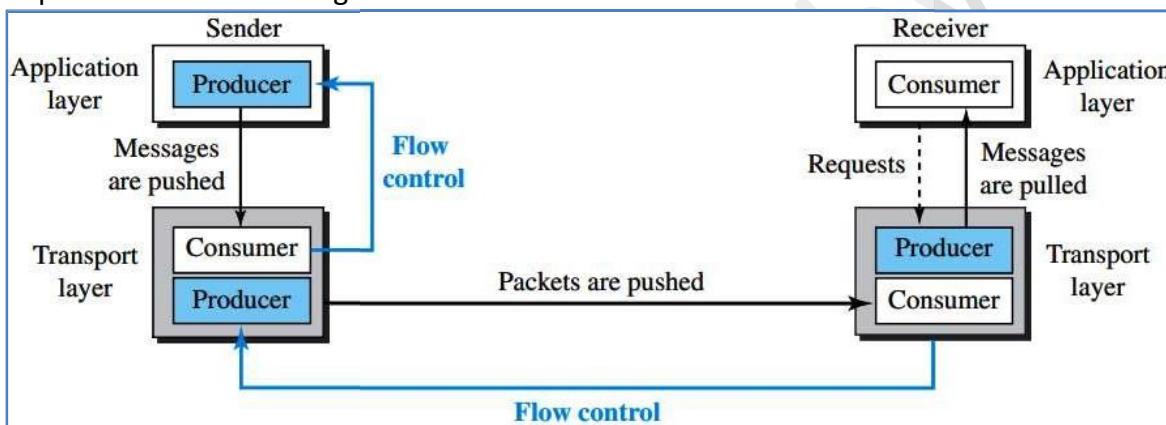
### Pushing or Pulling

- Delivery of items from a producer to a consumer can occur in one of two ways: pushing or pulling. If the sender delivers items whenever they are produced - without a prior request from the consumer - the delivery is referred to as pushing. If the producer delivers the items after the consumer has requested them, the delivery is referred to as pulling (Figure).



### Flow Control at Transport Layer

- In communication at the transport layer, we are dealing with **four entities**:
  1. sender process
  2. sender transport layer
  3. receiver transport layer
  4. receiver process.
- The **sending process** at the application layer is only a producer. It produces message chunks and pushes them to the transport layer.
- The **sending transport layer** has a double role: it is both a consumer and a producer. It consumes the messages pushed by the producer. It encapsulates the messages in packets and pushes them to the receiving transport layer.
- The **receiving transport layer** also has a double role: it is the consumer for the packets received from the sender and the producer that decapsulates the messages and delivers them to the application layer.
- The **receiver process** - is normally a pulling delivery; the transport layer waits until the application-layer process asks for messages.



- Figure shows that we need at least two cases of flow control: from the sending transport layer to the sending application layer and from the receiving transport layer to the sending transport layer.

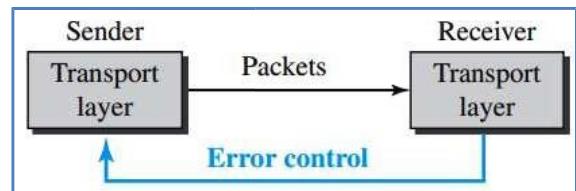
### Buffers

- Although flow control can be implemented in several ways, **one of the solutions is normally to use two buffers: one at the sending transport layer and the other at the receiving transport layer**. A buffer is a set of memory locations that can hold packets at the sender and receiver. The flow control communication can occur by sending signals from the consumer to the producer.
- When the buffer of the sending transport layer is full**, it informs the application layer to stop passing chunks of messages; when there are some vacancies, it informs the application layer that it can pass message chunks again.
- When the buffer of the receiving transport layer is full**, it informs the sending transport layer to stop sending packets. When there are some vacancies, it informs the sending transport layer that it can send packets again.

### Error Control

- In the Internet, since the underlying network layer (IP) is unreliable, we need to make the transport layer reliable if the application requires reliability. Reliability can be achieved to add error control services to the transport layer.
- Error control at the transport layer is responsible for**
  1. Detecting and discarding corrupted packets.
  2. Keeping track of lost and discarded packets and resending them.
  3. Recognizing duplicate packets and discarding them.
  4. Buffering out-of-order packets until the missing packets arrive.

- **Error control**, unlike flow control, **involves only the sending and receiving transport layers**. Figure shows the error control between the sending and receiving transport layers. As with the case of flow control, the receiving transport layer manages error control, most of the time, by informing the sending transport layer about the problems.



### Sequence Numbers

- **Error control requires that the sending transport layer knows which packet is to be resent and the receiving transport layer knows which packet is a duplicate, or which packet has arrived out of order. This can be done if the packets are numbered.** We can add a field to the transport-layer packet to hold the sequence number of the packet. When a packet is corrupted or lost, the receiving transport layer can somehow inform the sending transport layer to resend that packet using the sequence number. The receiving transport layer can also detect duplicate packets if two received packets have the same sequence number. The out-of-order packets can be recognized by observing gaps in the sequence numbers.
- If the header of the packet allows  $m$  bits for the sequence number, the sequence numbers range from **0** to  **$2^m - 1$** . For example, if  $m$  is 4, the only sequence numbers are 0 through 15, inclusive. However, we can wrap around the sequence. So the sequence numbers in this case are: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...**

### Acknowledgment

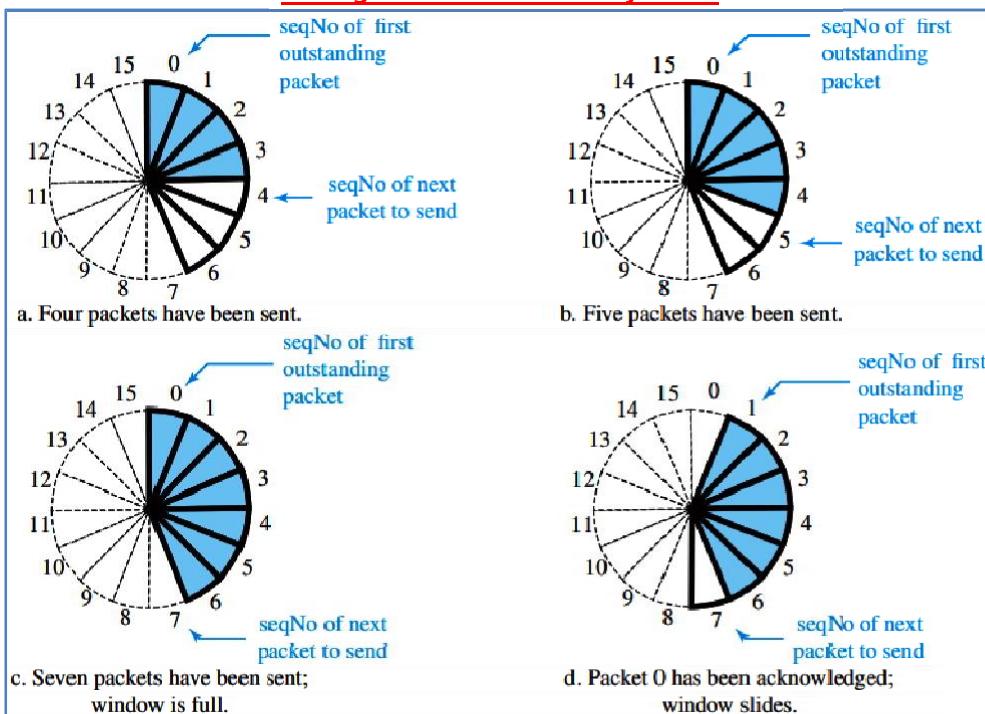
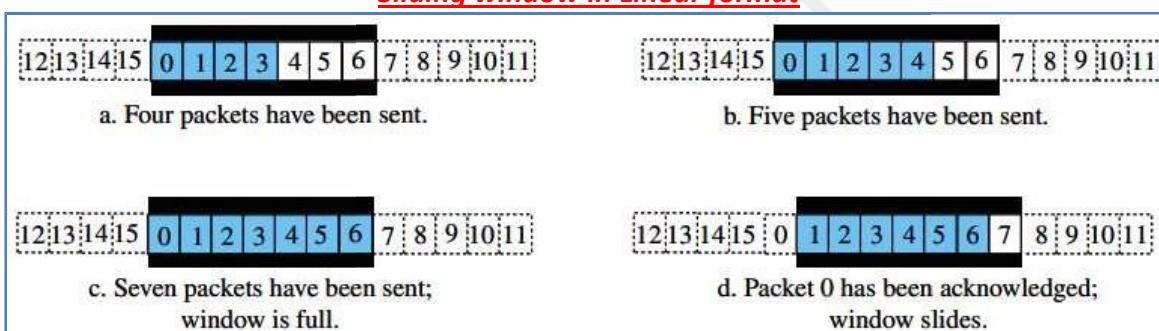
- We can use both positive and negative signals as error control, but we discuss only positive signals, which are more common at the transport layer. The receiver side can send an acknowledgment (ACK) for each of a collection of packets that have arrived safe and sound. The receiver can simply discard the corrupted packets.
- The sender can detect lost packets if it uses a timer. When a packet is sent, the sender starts a timer. If an ACK does not arrive before the timer expires, the sender resends the packet. Duplicate packets can be silently discarded by the receiver. Out-of-order packets can be either discarded (to be treated as lost packets by the sender), or stored until the missing one arrives.

### Combination of Flow and Error Control

- **Flow control** requires the use of two buffers, one at the sender site and the other at the receiver site. **Error control** requires the use of sequence and acknowledgment numbers by both sides.
- These two requirements can be combined if we use two numbered buffers, one at the sender, one at the receiver.
- **At the sender**, when a packet is prepared to be sent, we use the number of the next free location,  $x$ , in the buffer as the sequence number of the packet. When the packet is sent, a copy is stored at memory location  $x$ , awaiting the acknowledgment from the other end. When an acknowledgment related to a sent packet arrives, the packet is purged and the memory location becomes free.
- **At the receiver**, when a packet with sequence number  $y$  arrives, it is stored at the memory location  $y$  until the application layer is ready to receive it. An acknowledgment can be sent to announce the arrival of packet  $y$ .

### Sliding Window

- Since the sequence numbers use modulo  $2^m$ , a circle can represent the sequence numbers from **0** to  **$2^m - 1$**  (Figure). The buffer is represented as a set of slices, called the sliding window, that occupies part of the circle at any time.
- At the sender site, when a packet is sent, the corresponding slice is marked. When all the slices are marked, it means that the buffer is full and no further messages can be accepted from the application layer. **When an acknowledgment arrives, the corresponding slice is unmarked.**

***Sliding window in circular format******Sliding window in Linear format*****Congestion Control**

- An important issue in a packet-switched network, such as the Internet, is congestion.
- Congestion in a network may occur if the load on the network—the number of packets sent to the network—is greater than the capacity of the network—the number of packets a network can handle.
- ***Congestion control refers to the mechanisms and techniques that control the congestion and keep the load below the capacity.***
- Congestion in a network or internetwork occurs because routers and switches have queues—buffers that hold the packets before and after processing. Congestion at the transport layer is actually the result of congestion at the network layer, which manifests itself at the transport layer.

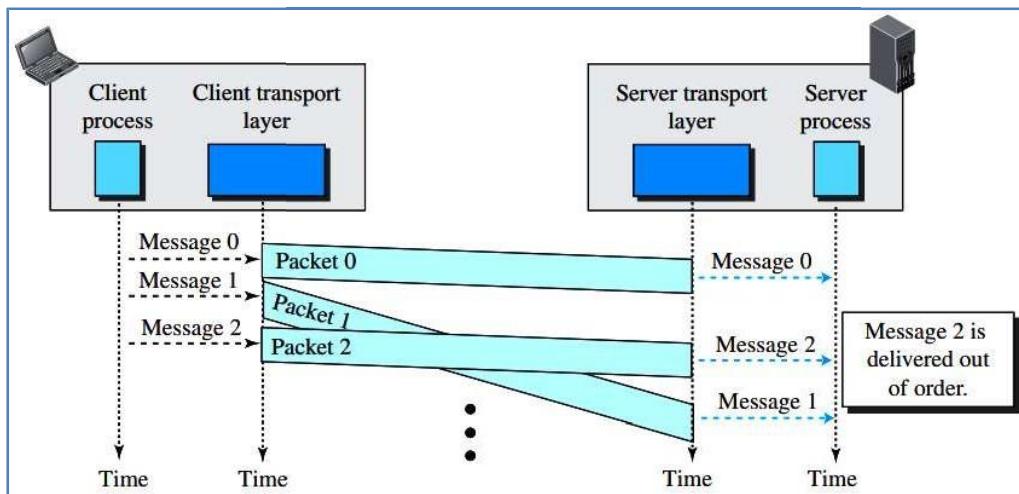
**Connectionless and Connection-Oriented Protocols**

- A transport-layer protocol provide two types of services: connectionless and connection-oriented.
- ***At the transport layer, we are not concerned about the physical paths of packets*** (we assume a logical connection between two transport layers). Connectionless service at the transport layer means ***independency between packets; connection-oriented means dependency.***

**Connectionless Service**

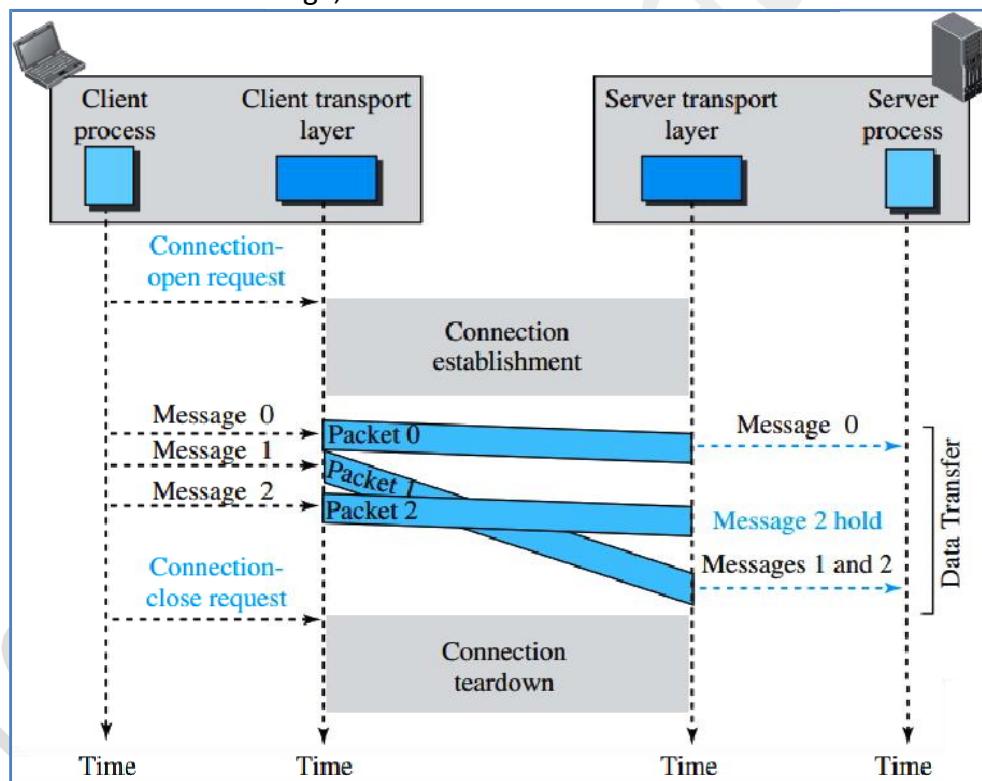
- In a connectionless service, the source process (application program) needs to divide its message into chunks of data of the size acceptable by the transport layer and deliver them to the transport layer one by one.

- 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.



### 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.

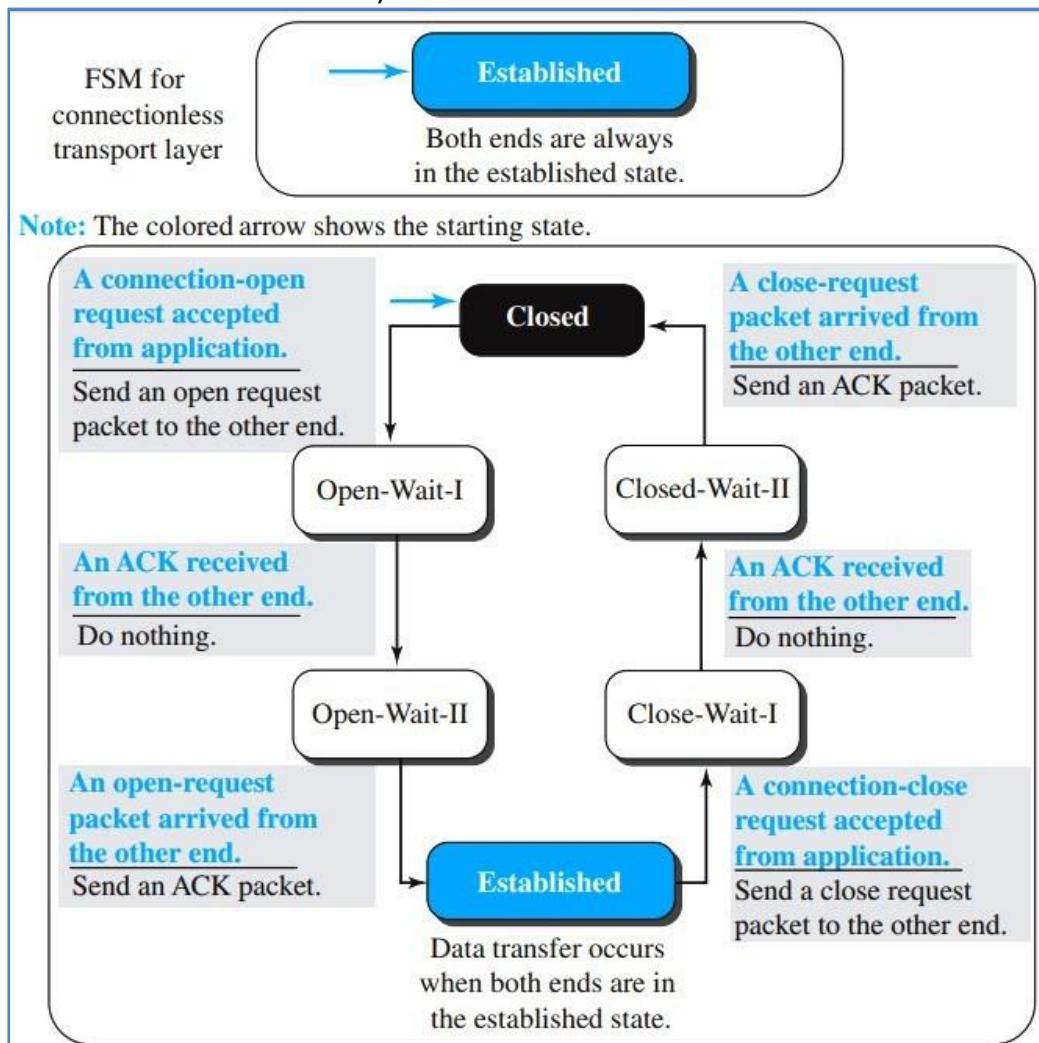


- At the transport layer, connection-oriented service involves only the two hosts; the service is end to end. This means that we should be able to make a connection-oriented protocol at the transport layer over either a connectionless or connection-oriented protocol at the network layer.
- Figure shows the connection establishment, data-transfer, and tear-down phases in a connection-oriented service at the transport layer. We can implement flow control, error control, and congestion control in a connection-oriented protocol.

### 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).

- Figure shows a representation of a transport layer using an FSM. Using this tool, each transport layer (sender or receiver) is taught as a machine with a finite number of states. The machine is always in one of the states until an event occurs. Each event is associated with two reactions: defining the list (possibly empty) of actions to be performed and determining the next state (which can be the same as the current state).



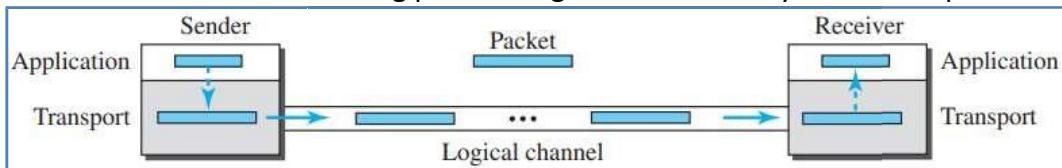
- An FSM in a connection-oriented transport layer, on the other hand, needs to go through **three states before reaching the established state.**
- The machine also needs to go through three states before closing the connection. The machine is in the closed state when there is no connection. It remains in this state until a request for opening the connection arrives from the local process; the machine sends an open request packet to the remote transport layer and moves to the **open-wait-I state.**
- When an **acknowledgment is received** from the other end, the local FSM moves to the **open-wait-II state.** Then a unidirectional connection has been established, but if a bidirectional connection is needed, the machine needs to wait in this state until the other end also requests a connection.
- When **the request is received**, the machine sends an acknowledgment and moves to the established state. Data and data acknowledgment can be exchanged between the two ends when they are both in the established state.
- To **tear down a connection, the application layer sends a close request message to its local transport layer.** The transport layer sends a close-request packet to the other end and moves to close-wait-I state. When an acknowledgment is received from the other end, the machine moves to the close-wait-II state and waits for the close-request packet from the other end. When this packet arrives, the machine sends an acknowledgment and moves to the closed state.

## 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.

### Simple Protocol

- Our first protocol - 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 shows the layout for this protocol.



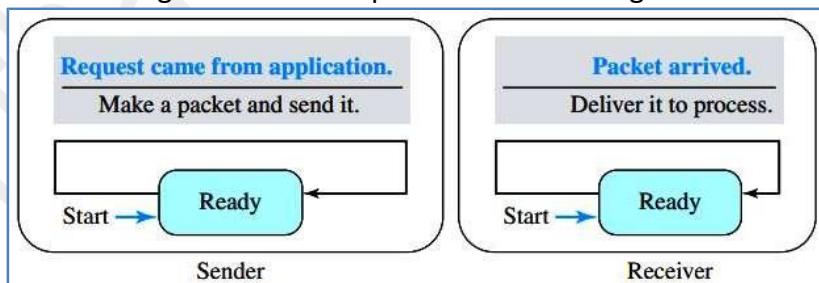
- The **transport layer at the sender** gets a message from its application layer, makes a packet out of it, and sends the packet.
- The **transport layer at the receiver receives a packet** from its network layer, extracts the message from the packet, and delivers the message to its application layer.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

### FMS

- 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 shows the FSMs for the simple protocol. We see later that the UDP protocol is a slight modification of this protocol.

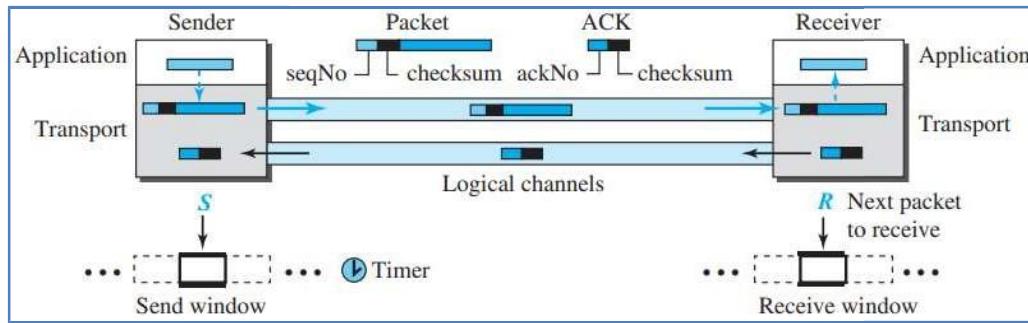
### Stop-and-Wait Protocol

- Our second protocol is a connection-oriented protocol called the Stop-and-Wait protocol, which 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.



### Sequence Numbers

- To prevent duplicate packets, the protocol uses sequence numbers and acknowledgment numbers. A field is added to the packet header to hold the sequence number of that packet.



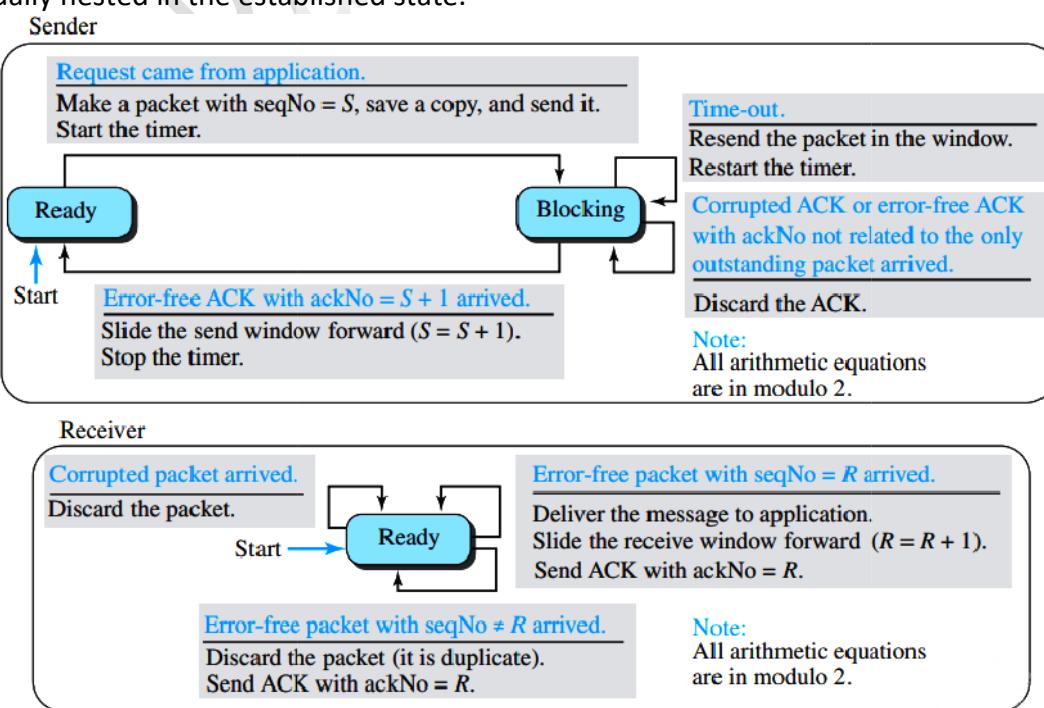
- Let us discuss the range of sequence numbers we need. Assume we have used  $x$  as a sequence number; we only need to use  $x + 1$  after that.
- To show this, assume that the sender has sent the packet with sequence number  $x$ . Three things can happen.
  - The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment.** The acknowledgment arrives at the sender site, causing the sender to send the next packet numbered  $x + 1$ .
  - 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.
  - The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment, but the acknowledgment 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 acknowledgments, we use this convention: The acknowledgment numbers always announce, in modulo-2 arithmetic, the sequence number of the next packet expected by the receiver.
- The sender has a control variable, which we call **S (sender)**, that points to the only slot in the send window. The receiver has a control variable, which we call **R (receiver)**, that points to the only slot in the receive window.

### FSMs

- Figure shows the FSMs for the Stop-and-Wait protocol. Since the protocol is a connection-oriented protocol, both ends should be in the established state before exchanging data packets. The states are actually nested in the established state.



**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 and then moves to the blocking state.**
- **Blocking state.** When the sender is in this state, **three events can occur:**
  1. If an error-free ACK arrives with the ackNo related to the next packet to be sent, which means  $\text{ackNo} = (S + 1) \text{ modulo } 2$ , then the timer is stopped. The window slides,  $S = (S + 1) \text{ modulo } 2$ . Finally, the sender moves to the ready state.
  2. If a corrupted ACK or an error-free ACK with the  $\text{ackNo} \neq (S + 1) \text{ modulo } 2$  arrives, the ACK is discarded.
  3. If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

**Receiver**

- The receiver is always in the ready state. **Three events may occur:**
  1. If an error-free packet with  $\text{seqNo} = R$  arrives, the message in the packet is delivered to the application layer. The window then slides,  $R = (R + 1) \text{ modulo } 2$ . Finally an ACK with  $\text{ackNo} = R$  is sent.
  2. If an error-free packet with  $\text{seqNo} \neq R$  arrives, the packet is discarded, but an ACK with  $\text{ackNo} = R$  is sent.
  3. If a corrupted packet arrives, the packet is discarded.

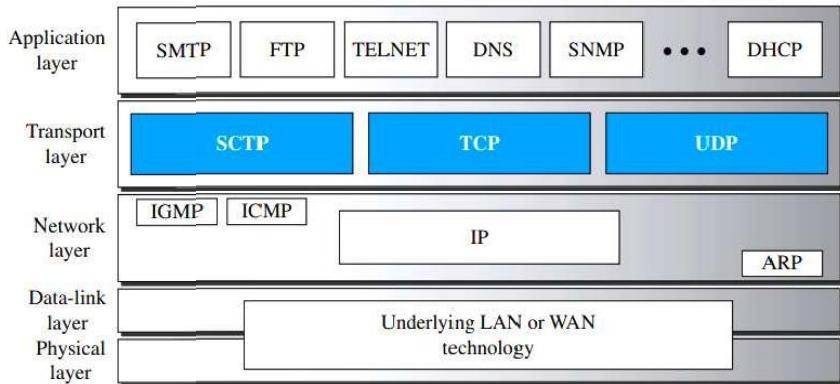
**Efficiency**

- The Stop-and-Wait 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 **bandwidth-delay** product.

**Pipelining**

- In networking and in other areas, a task is often begun before the previous task has ended. This is known as **pipelining**. There is no pipelining in the Stop-and-Wait protocol because a sender must wait for a packet to reach the destination and be acknowledged before the next packet can be sent.
- However, pipelining does apply to our next two protocols because several packets can be sent before a sender receives feedback about the previous packets.
- Pipelining improves the efficiency of the transmission if the number of bits in transition is large with respect to the **bandwidth-delay** product.

- The position of these three protocols in the TCP/IP protocol suite



### Services

- Each protocol provides a different type of service and should be used appropriately.
- 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.

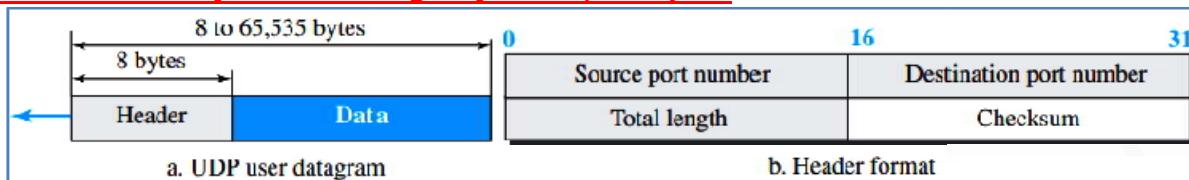
### Port Numbers

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
111	RPC	✓	✓	✓	Remote Procedure Call
123	NTP	✓	✓	✓	Network Time Protocol
161	SNMP-server	✓			Simple Network Management Protocol
162	SNMP-client	✓			Simple Network Management Protocol

### User Datagram Protocol (UDP)

- The User Datagram Protocol (UDP) is a connectionless, unreliable transport protocol - providing process-to-process communication instead of host-to-host communication.
- Advantages.** 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. Sending a small message using UDP takes much less interaction between the sender and receiver than using TCP.
- ***UDP packets***, called ***user datagrams***, have a ***fixed-size header of 8 bytes*** made of four fields, each of 2 bytes (16 bits). Figure shows the format of a user datagram.
- ***The 16 bits can define a total length of 0 to 65,535 bytes.***



### ***UDP Services***

- Process-to-Process Communication
- Connectionless Services
- Flow Control
- Error Control
- Checksum
- Congestion Control
- Encapsulation and Decapsulation
- Queuing
- Multiplexing and Demultiplexing

### **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.
- One of the consequences of being connectionless is that the process that uses UDP cannot send a stream of data to UDP and expect UDP to chop them into different, related user datagrams.

### **Flow Control**

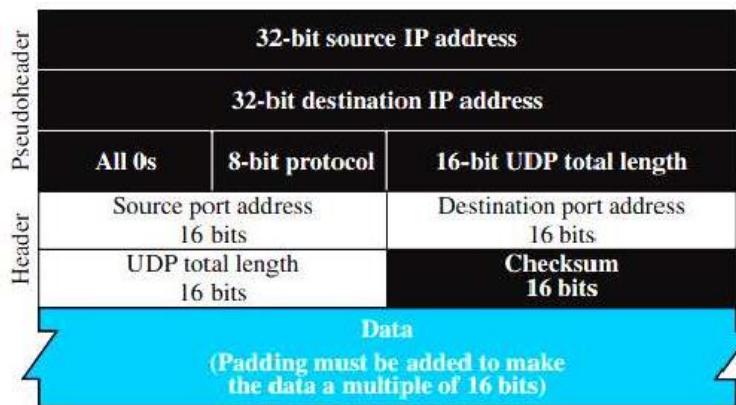
- UDP is a very simple protocol. ***There is no flow control***, and hence no window mechanism. The receiver may overflow with incoming messages. The lack of flow control means that the process using UDP should provide for this service, if needed.

### **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.

### **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 (see Figure).



## 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. This assumption may or may not be true today, when UDP is used for interactive real-time transfer of audio and video.

## Encapsulation and Decapsulation

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

## 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. Other implementations create only an incoming queue associated with each process.

## 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 demultiplexes.

## Typical 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.

## TCP Services (P743 – 785)

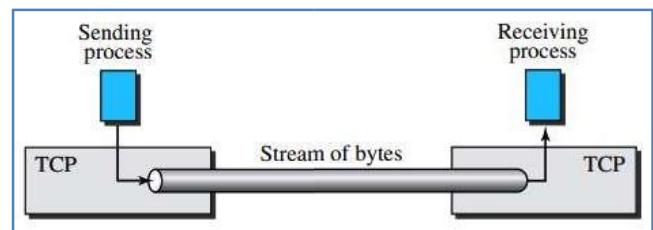
### Process-to-Process Communication

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

### Stream Delivery Service

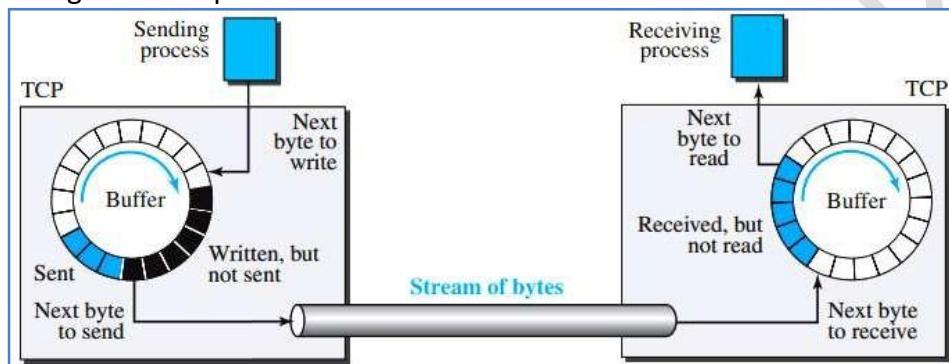
- TCP is a stream-oriented protocol - allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes.

- TCP creates an environment in which the two processes seem to be connected by an imaginary “tube” that carries their bytes across the Internet (Figure). The sending process produces (writes to) the stream and the receiving process consumes (reads from) it.



### Sending and Receiving Buffers

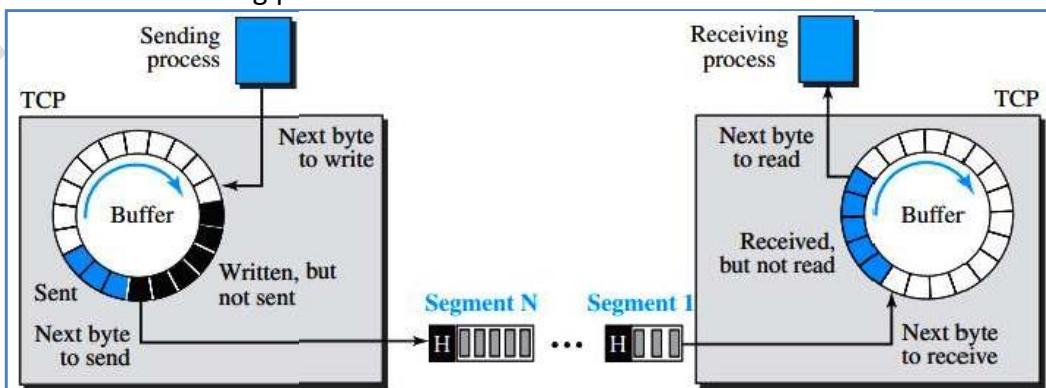
- Because the sending and the receiving processes may not necessarily write or read data at the same rate, TCP needs buffers for storage.
- There are two buffers, the sending buffer and the receiving buffer, one for each direction. One way to implement a buffer is to use a circular array of 1-byte locations as shown in Figure. For simplicity, we have shown two buffers of 20 bytes each; normally the buffers are hundreds or thousands of bytes, depending on the implementation.



- The figure shows the movement of the data in one direction. 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 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

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



## 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. However, since TCP is a connection-oriented protocol, a connection needs to be established for each pair of processes.

## 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.
- **Note that this is a logical connection, not a physical connection.** The TCP segment is encapsulated in an IP datagram and can be sent **out of order, or lost or corrupted, and then resent.**
- **Each may be routed over a different path to reach the destination.** TCP creates a stream-oriented environment in which it accepts the responsibility of delivering the bytes in order to the other site.

## 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.

## Features

### **Numbering System**

- Although the TCP software keeps track of the segments being transmitted or received, there is no field for a segment number value in the segment header. Instead, there are **two fields**, called 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  **$2^{32} - 1$**  for the number of the first byte. For example, if the number happens to be 1057 and the total data to be sent is 6000 bytes, the bytes are numbered from 1057 to 7056. We will see that byte numbering is used for flow and error control.

### **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:
  1. The sequence number of the first segment is the ISN (initial sequence number), which is a random number.
  2. 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. Later, we show that some control segments are thought of as carrying one imaginary byte.

### **Example**

**Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 10001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1000 bytes?**

- **The following shows the sequence number for each segment:**

Segment 1 → Sequence Number: 10001 Range: 10001 to 11000

Segment 2 → Sequence Number: 11001 Range: 11001 to 12000

Segment 3 → Sequence Number: 12001 Range: 12001 to 13000

Segment 4 → Sequence Number: 13001 Range: 13001 to 14000

Segment 5 → Sequence Number: 14001 Range: 14001 to 15000

- When a segment carries a combination of data and control information (piggybacking), it uses a sequence number. If a segment does not carry user data, it does not logically define a sequence number.

### Acknowledgment Number

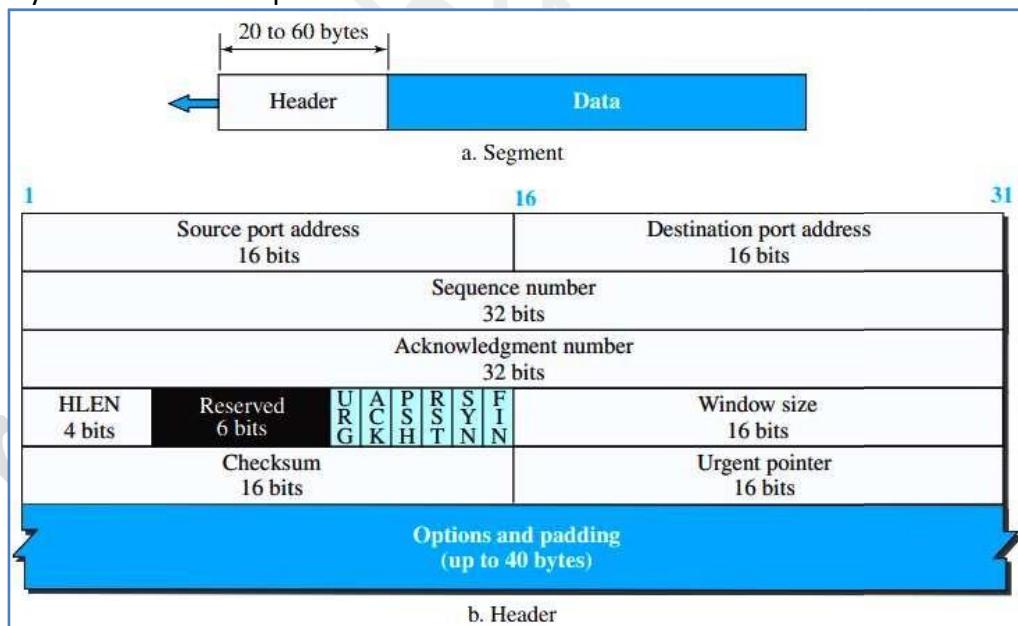
- Communication in TCP is full duplex; when a connection is established, both parties can send and receive data at the same time.
- Each party numbers the bytes, usually with a different starting byte number. The sequence number in each direction shows the number of the first byte carried by the segment. Each party also uses an acknowledgment number to confirm the bytes it has received.
- However, **the acknowledgment number defines the number of the next byte that the party expects to receive**. In addition, 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.
- The term cumulative here means that if a party uses 5643 as an acknowledgment number, it has received all bytes from the beginning up to 5642. Note that this does not mean that the party has received 5642 bytes, because the first byte number does not have to be 0.

### Segments

- A packet in TCP is called a segment.

### Format

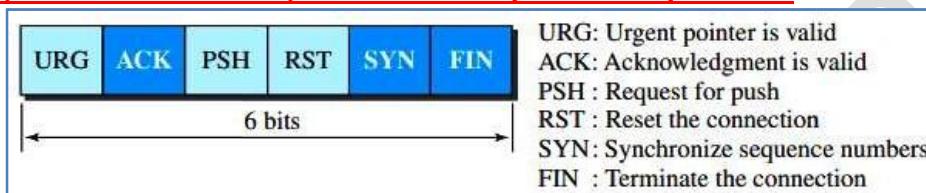
- The format of a segment is shown in Figure. The segment consists of a **header of 20 to 60 bytes, followed by data from the application program**. The header is 20 bytes if there are no options and up to 60 bytes if it contains options.



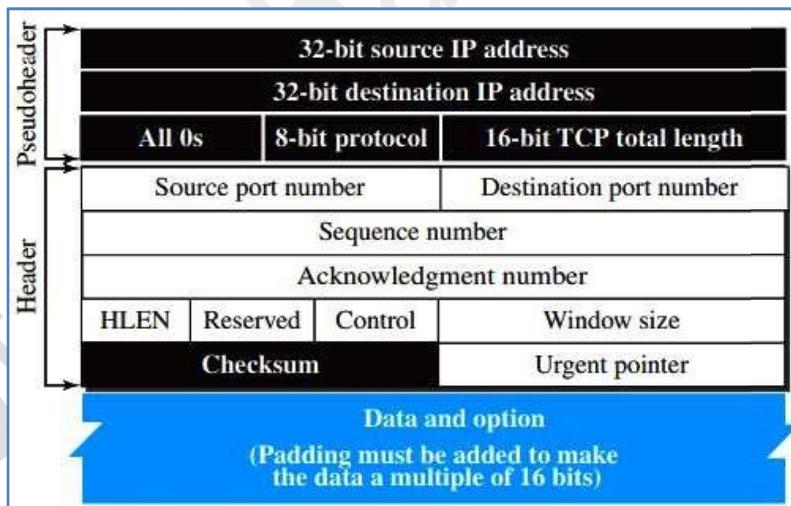
- Source port address**. This is a **16-bit field** that defines the port number of the application program in the host that is sending the segment.
- Destination port address**. This is a **16-bit field** that defines the port number of the application program in the host that is receiving the segment.
- 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** each party uses a random number generator to **create an initial sequence number (ISN)**, which is usually different in each direction.

- **Acknowledgment number**. This **32-bit field** defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number  $x$  from the other party, it returns  $x + 1$  as the acknowledgment number. **Acknowledgment and data can be piggybacked together.**
- **Header length**. This **4-bit field** indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field is always between 5 ( $5 \times 4 = 20$ ) and 15 ( $15 \times 4 = 60$ ).
- **Control**. This field defines **6 different control bits or flags, as shown in Figure**. 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. **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. The use of the checksum in the UDP datagram is optional, whereas the use of the checksum for TCP is mandatory. The same pseudoheader, serving the same purpose, is added to the segment. For the TCP pseudoheader, the value for the protocol field is 6. See Figure.



- **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**.  
**Encapsulation**
- A TCP segment encapsulates the data received from the application layer. The TCP segment is encapsulated in an IP datagram, which in turn is encapsulated in a frame at the data-link layer.

## TCP connections

### Topics covered

- Connection Establishment (Three-Way Handshaking - SYN Flooding Attack)
- Data Transfer (Pushing Data - Urgent Data)
- Connection Termination (Three-Way Handshaking - Half-Close)
- Connection Reset

- TCP is connection-oriented. A connection-oriented transport protocol establishes a logical path between the source and destination. All of the segments belonging to a message are then sent over this logical path.
- Using a single logical pathway for the entire message facilitates the acknowledgment process as well as retransmission of damaged or lost frames. You may wonder how TCP, which uses the services of IP, a connectionless protocol, can be connection-oriented.
- The point is that a **TCP connection is logical, not physical**. TCP operates at a higher level. TCP uses the services of IP to deliver individual segments to the receiver, but it controls the connection itself. If a segment is lost or corrupted, it is retransmitted.
  - Unlike TCP, **IP is unaware of this retransmission**. If a **segment arrives out of order**, **TCP holds it until the missing segments arrive**; IP is unaware of this reordering.
- In TCP, connection-oriented transmission requires **three phases**:
  1. Connection establishment
  2. Data transfer
  3. Connection termination.

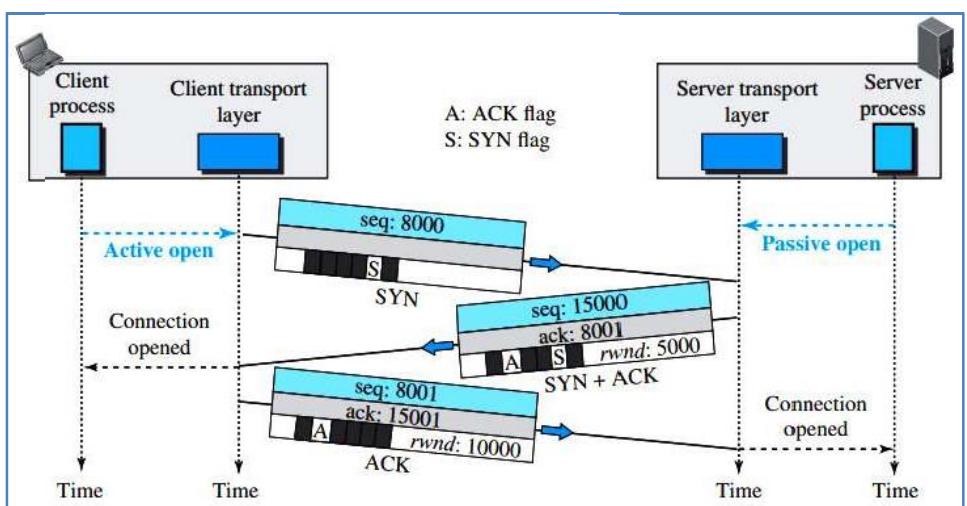
### **1. Connection Establishment (Three-Way Handshaking - SYN Flooding Attack)**

- TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously. This implies that each party must initialize communication and get approval from the other party before any data are transferred.

### Three-Way Handshaking

- The **connection establishment in TCP is called three-way handshaking**. In our example, an application program, called the client, wants to make a connection with another application program, called the server, using TCP as the transport-layer protocol.
- The process starts with the server. **The server program tells its TCP that it is ready to accept a connection. This 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.**



- To show the process we use time lines. Each segment has values for all its header fields and perhaps for some of its option fields too.

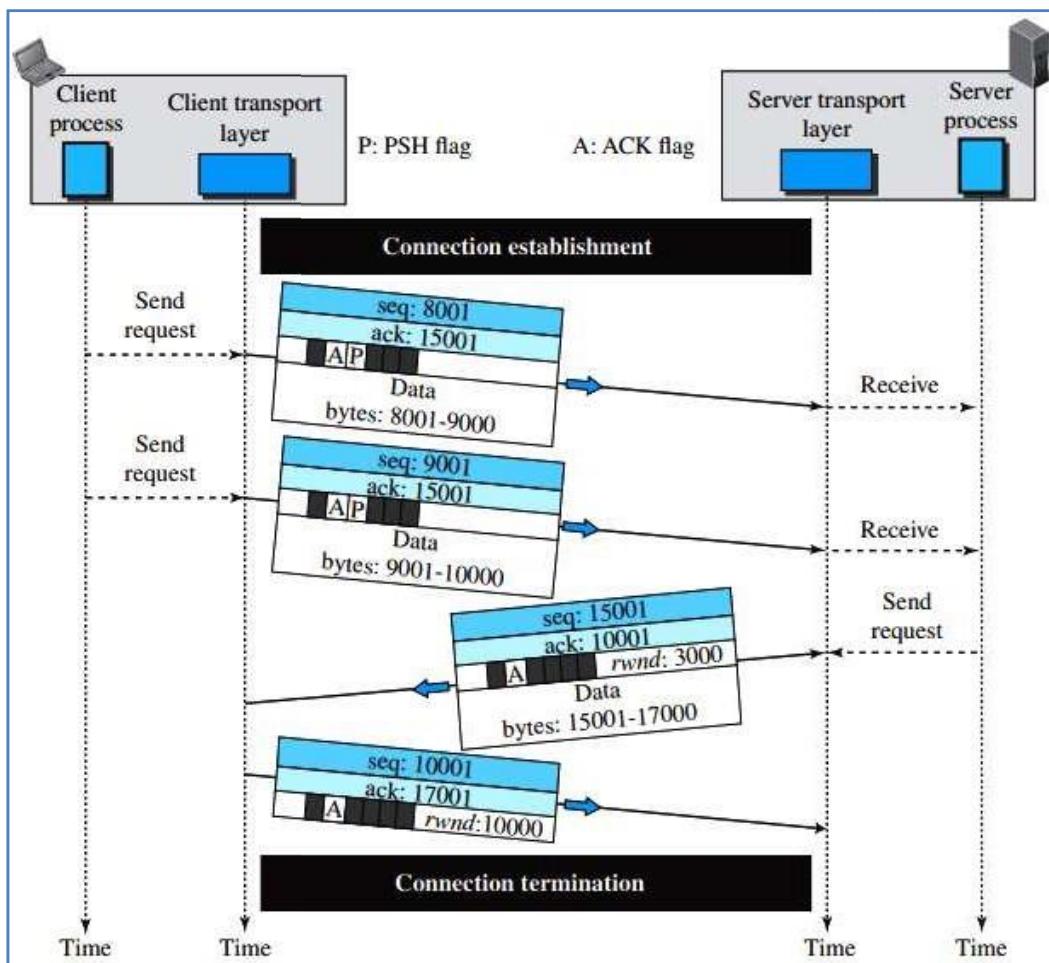
- We show the sequence number, the acknowledgment number, the control flags (only those that are set), and window size if relevant. The three steps in this phase are as follows.
  1. The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization 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).
    - The SYN segment is a control segment and carries no data. However, it consumes one sequence number because it needs to be acknowledged. We can say that the SYN segment carries one imaginary byte.
  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 ( just an ACK segment) It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field. Note that the ACK segment does not consume any sequence numbers if it does not carry data, but some implementations allow this third segment in the connection phase to carry the first chunk of data from the client. In this case, the segment consumes as many sequence numbers as the number of data bytes.

### **SYN Flooding Attack - denial of service attack - cookie**

- 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.
- The 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.
- Some implementations of TCP have strategies to alleviate the effect of a SYN attack. Some have imposed a limit of connection requests during a specified period of time. Others try to filter out datagrams coming from unwanted source addresses. One recent strategy is to postpone resource allocation until the server can verify that the connection request is coming from a valid IP address, by using what is called a cookie. SCTP, the new transport-layer protocol uses this strategy.

### **2. Data Transfer (Pushing Data - Urgent Data)**

- After connection is established, bidirectional data transfer can take place. The client and server can send data and acknowledgments in both directions. We will study the rules of acknowledgment later in the chapter; for the moment, it is enough to know that data traveling in the same direction as an acknowledgment are carried on the same segment. The acknowledgment is piggybacked with the data. Figure shows an example.



- In this example, after a connection is established, the **client sends 2,000 bytes of data in two segments**. The server then sends 2,000 bytes in one segment. The client sends one more segment. The first three segments carry both data and acknowledgment, but the last segment carries only an acknowledgment because there is no more data to be sent.
- Note the **values of the sequence and acknowledgment numbers**. The data segments sent by the client have the **PSH (push) flag** set so that the server TCP knows to deliver data to the server process as soon as they are received. The **segment from the server, on the other hand, does not set the push flag**. Most TCP implementations have the option to set or not to set this flag.

### Pushing Data

- 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. This type of flexibility increases the efficiency of TCP.
- There are occasions in which** the application program has no need for this flexibility. Delayed transmission and delayed delivery of data may not be acceptable by the application program.
- TCP can handle such a situation**. The application program at the sender can request a push operation. This means that the sending TCP must not wait for the window to be filled. It must create a segment and send it immediately. The sending TCP must also set the push bit (PSH) to let the receiving TCP know that the segment includes data that must be delivered to the receiving application program as soon as possible and not to wait for more data to come.

### Urgent Data

- There are **occasions in which an application program needs to send urgent bytes, some bytes that need to be treated in a special way by the application at the other end**. The solution is to send a

segment with the URG bit set. The sending application program tells the sending TCP that the piece of data is urgent. The sending TCP creates a segment and inserts the urgent data at the beginning of the segment. The rest of the segment can contain normal data from the buffer. The urgent pointer field in the header defines the end of the urgent data (the last byte of urgent data).

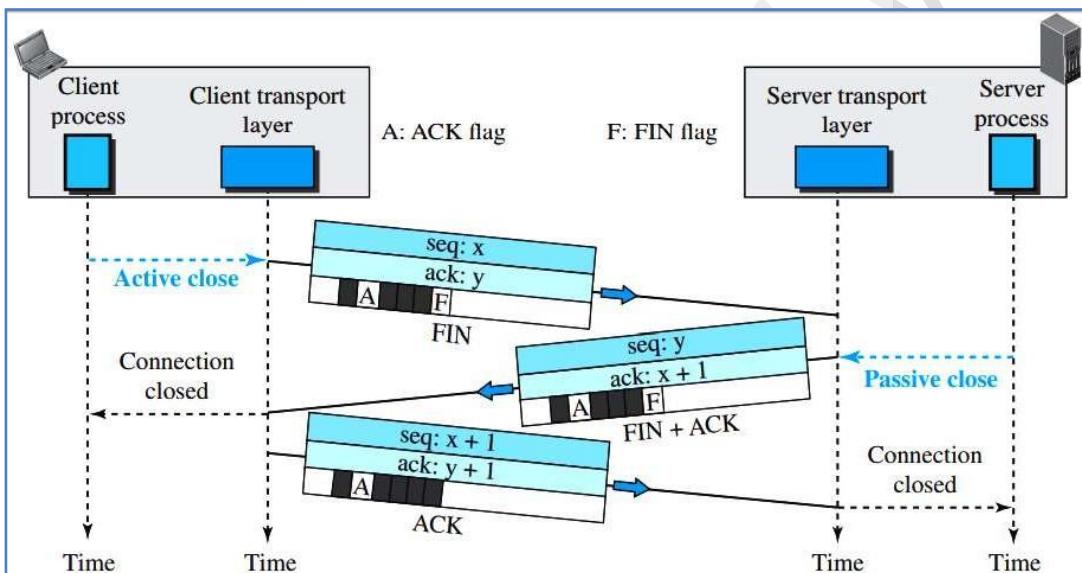
- For example, if the segment sequence number is 15000 and the value of the urgent pointer is 200, the first byte of urgent data is the byte 15000 and the last byte is the byte 15200. The rest of the bytes in the segment (if present) are non-urgent.

### 3. Connection Termination (Three-Way Handshaking - Half-Close)

- Either of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client. Most implementations today allow two options for connection termination: three-way handshaking and four-way handshaking with a half-close option.

#### Three-Way Handshaking

- Most implementations today allow three-way handshaking for connection termination, as shown in Figure.

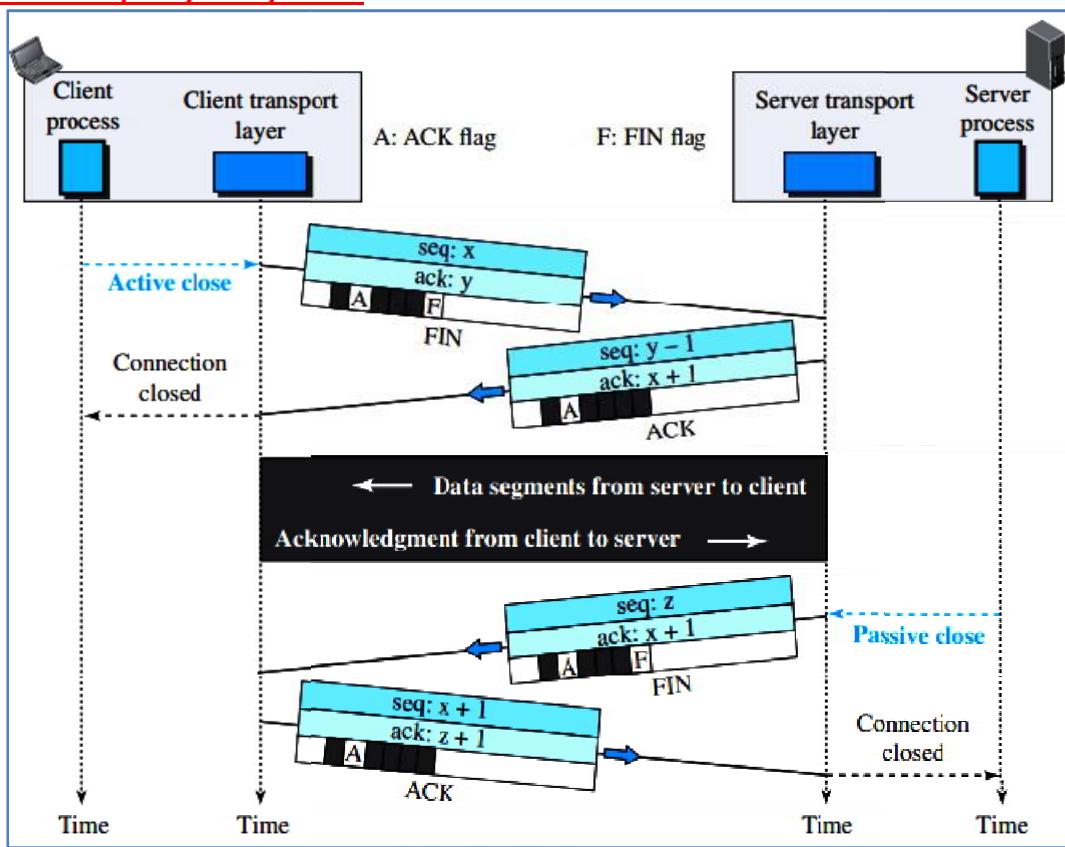


1. In this situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set. Note that a FIN segment can include the last chunk of data sent by the client or it can be just a control segment as shown in the figure. If it is only a control segment, it consumes only one sequence number because it needs to be acknowledged.
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. This segment can also contain the last chunk of data from the server. If it does not carry data, it consumes only one sequence number because it needs to be acknowledged.
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. This segment cannot carry data and consumes no sequence numbers.

#### Half-Close

- In TCP, one end can stop sending data while still receiving data. This is called a halfclose. Either the server or the client can issue a half-close request. It can occur when the server needs all the data before processing can begin. A good example is sorting.

- When the client sends data to the server to be sorted, the server needs to receive all the data before sorting can start.
- The ***server-to-client direction must remain open to return the sorted data***. The server, after receiving the data, still needs time for sorting; its outbound direction must remain open. ***Figure shows an example of a half-close***.



- The ***data transfer from the client to the server stops. The client half-closes the connection by sending a FIN segment***. The server accepts the half-close by sending the ACK segment. The server, however, can still send data. When the server has sent all of the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client.
- After ***half-closing the connection, data can travel from the server to the client and acknowledgments can travel from the client to the server***. The client cannot send any more data to the server.

### Connection Reset

- TCP at one end may deny a connection request, may abort an existing connection, or may **terminate an idle connection**. All of these are done with the **RST (reset) flag**.

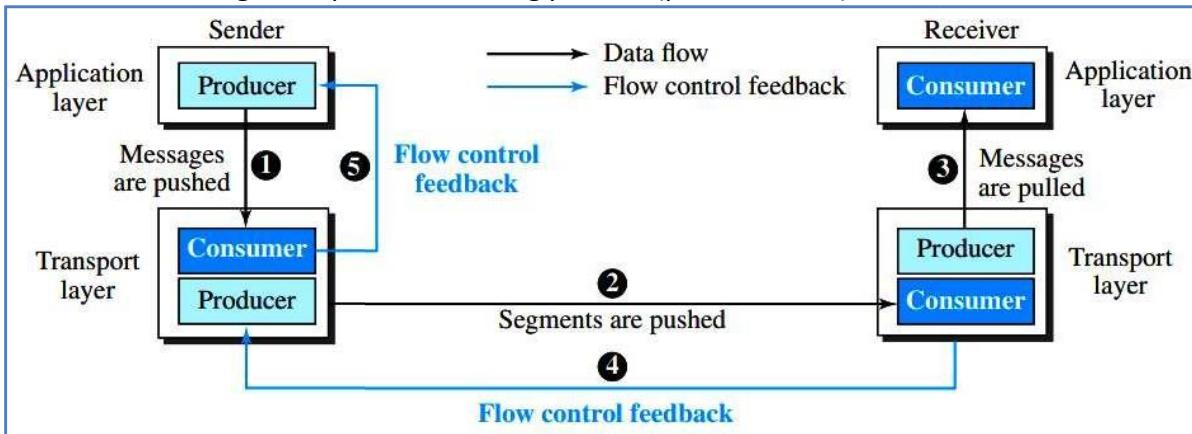
### Flow control

#### Topics covered

- Opening and Closing Windows
- Shrinking of Windows
- Silly Window Syndrome

- 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. We **assume that the logical channel between the sending and receiving TCP is error-free**.

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



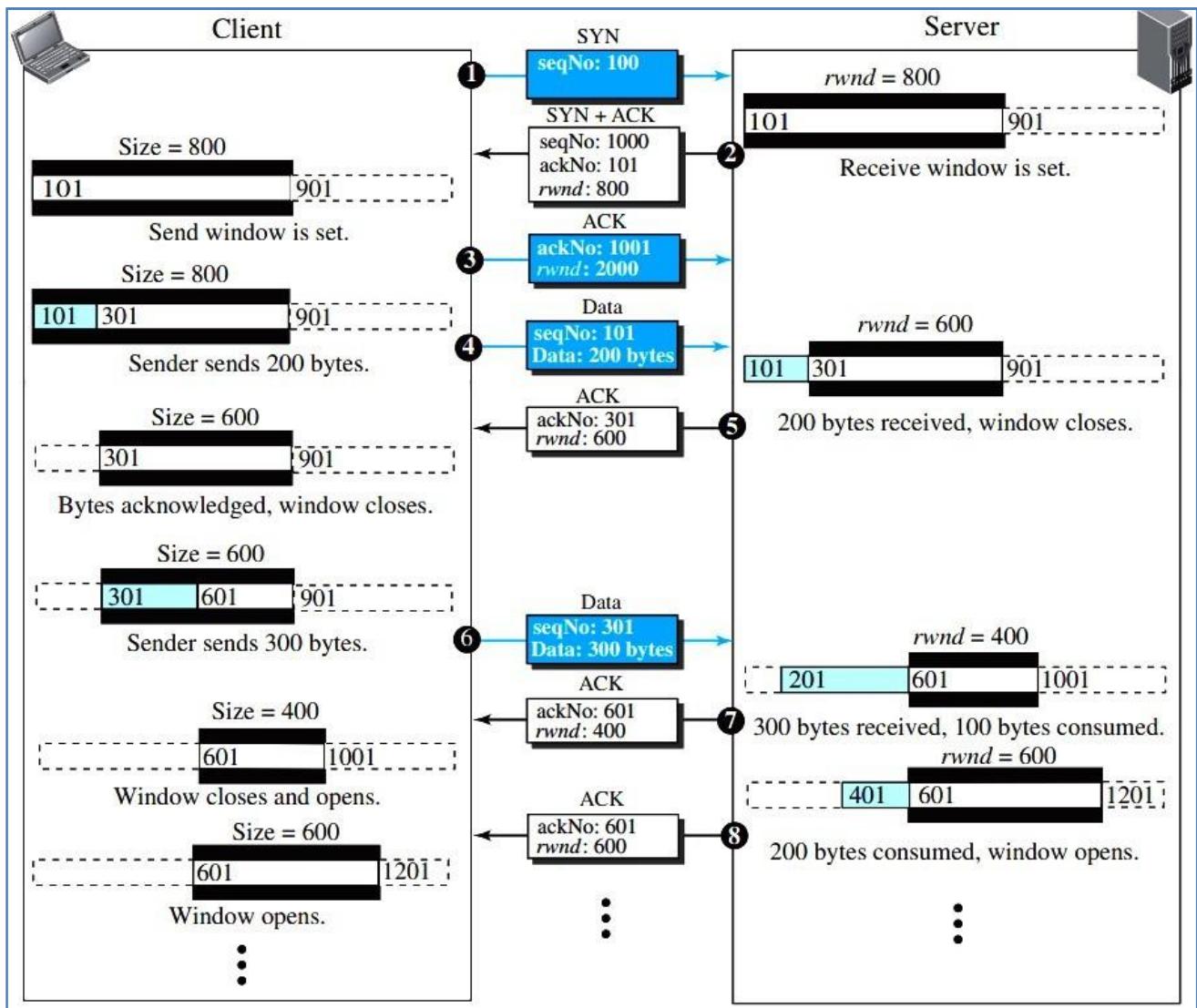
- **Most implementations of TCP do not provide flow control feedback from the receiving process to the receiving TCP**; they let the receiving process pull data from the receiving TCP whenever it is ready to do so.
- The receiving TCP controls the sending TCP; the sending TCP controls the sending process.
- 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. Flow control concentrates on the feedback sent from the receiving TCP to the sending TCP (path 4).

### Opening and Closing Windows

- To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes, although the size of the buffer for both parties is fixed when the connection is established.
- The **receive window closes** when more bytes arrive from the sender; it **opens** when more bytes are pulled by the process. We assume that it does not shrink.
- The **opening**, **closing**, and **shrinking** of the send window is **controlled by the receiver**. The send window closes when a new acknowledgment allows it to do so. The send window opens when the **receive window size (rwnd) advertised by the receiver allows it to do so (new ackNo + new rwnd > last ackNo + last rwnd)**. The send window shrinks in the event this situation does not occur.

### A Scenario

- We show how the send and receive windows are set during the connection establishment phase, and how their situations will change during data transfer. **Figure shows a simple example of unidirectional data transfer (from client to server)**. For the time being, we ignore error control, assuming that no segment is corrupted, lost, duplicated, or has arrived out of order. Note that we have shown only two windows for unidirectional data transfer. Although the client defines server's window size of 2000 in the third segment, we have not shown that window because the communication is only unidirectional.
- **Eight segments are exchanged between the client and server:**
  1. The **first segment is from the client to the server (a SYN segment) to request connection**. The client announces its initial seqNo = 100. When this segment arrives at the server, it allocates a buffer size of 800 (an assumption) and sets its window to cover the whole buffer (rwnd = 800). Note that the number of the next byte to arrive is 101.
  2. The **second segment is from the server to the client**. This is an **ACK + SYN segment**. The segment uses ackNo = 101 to show that it expects to receive bytes starting from 101. It also announces that the client can set a buffer size of 800 bytes.



3. The **third segment is the ACK segment** from the client to the server. Note that the client has defined a rwnd of size 2000, but we do not use this value in our figure because the communication is only in one direction.
4. After the client has set its window with the **size (800) dictated by the server, the process pushes 200 bytes of data**. The TCP client numbers these bytes 101 to 300. It then creates a segment and sends it to the server. The segment shows the starting byte number as 101 and the segment carries 200 bytes. The window of the client is then adjusted to show that 200 bytes of data are sent but waiting for acknowledgment. When this segment is received at the server, the bytes are stored, and the receive window closes to show that the next byte expected is byte 301; the stored bytes occupy 200 bytes of buffer.
5. The **fifth segment is the feedback from the server to the client**. The server acknowledges bytes up to and including 300 (expecting to receive byte 301). The segment also carries the size of the receive window after decrease (600). The client, after receiving this segment, purges the acknowledged bytes from its window and closes its window to show that the next byte to send is byte 301. The window size, however, decreases to 600 bytes. Although the allocated buffer can store 800 bytes, the window cannot open (moving its right wall to the right) because the receiver does not let it.
6. **Segment 6 is sent by the client after its process pushes 300 more bytes**. The segment defines seqNo as 301 and contains 300 bytes. When this segment arrives at the server, the server stores them, but it has to reduce its window size. After its process has pulled 100 bytes of data, the window closes from the left for the amount of 300 bytes, but opens from the right for the

amount of 100 bytes. The result is that the size is only reduced 200 bytes. The receiver window size is now 400 bytes.

7. **In segment 7, the server acknowledges the receipt of data, and announces that its window size is 400.** When this segment arrives at the client, the client has no choice but to reduce its window again and set the window size to the value of rwnd = 400 advertised by the server. The send window closes from the left by 300 bytes, and opens from the right by 100 bytes.
8. **Segment 8 is also from the server after its process has pulled another 200 bytes.** Its window size increases. The new rwnd value is now 600. The segment informs the client that the server still expects byte 601, but the server window size has expanded to 600. Some implementations may **not allow advertisement of the rwnd at this time; the server then needs to receive some data before doing so.** After this segment arrives at the client, the client opens its window by 200 bytes without closing it. The result is that its window size increases to 600 bytes.

### **Shrinking of Windows**

- The receive window cannot shrink. The send window, on the other hand, can shrink if the receiver defines a value for rwnd that results in shrinking the window. However, some implementations do not allow shrinking of the send window.
- The limitation does not allow the right wall of the send window to move to the left. In other words, the receiver needs to keep the following relationship between the last and new acknowledgment and the last and new rwnd values to prevent shrinking of the send window.

$$\text{new ackNo} + \text{new rwnd} \geq \text{last ackNo} + \text{last rwnd}$$

- The left side of the inequality represents the new position of the right wall with respect to the sequence number space; the right side shows the old position of the right wall.
- The relationship shows that the right wall should not move to the left. The inequality is a mandate for the receiver to check its advertisement. However, note that the inequality is valid only if  $S_f < S_n$ ; we need to remember that all calculations are in modulo  $2^{32}$ .

### **Window Shutdown**

- We said that shrinking the send window by moving its right wall to the left is strongly discouraged. However, **there is one exception: 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.
- Even when the window is shut down by an order from the receiver, 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.** 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 traveling through an internet.
- 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? If it waits too long, it may delay the process. If it does not wait long enough, it may end up sending small segments. Nagle found an elegant solution. Nagle's algorithm is simple:
  - The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.
  - 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.
  - 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.
- Two solutions have been proposed to prevent the silly window syndrome created by an application program that consumes data more slowly than they arrive.
  - 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.
  - 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 acknowledgment may result in the sender unnecessarily retransmitting the unacknowledged segments. The protocol balances the advantages and disadvantages - defines that the acknowledgment should not be delayed by more than 500 ms.

## Error control

### **Topics covered**

- Checksum
  - Acknowledgment
  - Cumulative Acknowledgment (ACK)
  - Selective Acknowledgment (SACK)
  - Retransmission
  - Out-of-Order Segments
  - FSMs for Data Transfer in TCP
- 

- TCP provides reliability using error control. Error control includes mechanisms for detecting and resending corrupted segments, resending lost segments, storing out-of-order segments until missing segments arrive, and detecting and discarding duplicated segments. Error control in TCP is achieved through the use of three simple tools: checksum, acknowledgment, and time-out.

### **Checksum**

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

### **Acknowledgment**

- **TCP uses acknowledgments to confirm the receipt of data segments.** Control segments that carry no data, but consume a sequence number, are also acknowledged. ACK segments are never acknowledged.
- **In the past, TCP used only one type of acknowledgment: cumulative acknowledgment.** Today, some TCP implementations also use **selective acknowledgment**.

### **Cumulative Acknowledgment (ACK)**

- TCP was originally designed to acknowledge receipt of segments cumulatively. The receiver advertises the next byte it expects to receive, ignoring all segments received and stored out of order. This is sometimes referred to as **positive cumulative acknowledgment, or ACK**.
- The **word positive indicates that no feedback is provided for discarded, lost, or duplicate segments.** The **32-bit ACK** field in the TCP header is used for cumulative acknowledgments, and its value is valid only when the ACK flag bit is set to 1.

### **Selective Acknowledgment (SACK)**

- More and more implementations are adding another type of acknowledgment called **selective acknowledgment, or SACK**. A **SACK does not replace an ACK, but reports additional information to the sender.** A SACK reports a block of bytes that is out of order, and also a block of bytes that is duplicated, i.e., received more than once. However, since there is no provision in the TCP header for adding this type of information, SACK is implemented as an option at the end of the TCP header.

### **Generating Acknowledgments**

#### **When does a receiver generate acknowledgments?**

- During the evolution of TCP, several rules have been defined and used by several implementations. We give the most common rules here. The order of a rule does not necessarily define its importance.
  1. When **end A sends a data segment to end B, it must include (piggyback) an acknowledgment that gives the next sequence number it expects to receive.** This rule **decreases** the number of segments needed and therefore reduces traffic.

2. When the receiver has no data to send and it receives an in-order segment (with expected sequence number) and the previous segment has already been acknowledged, the receiver delays sending an ACK segment until another segment arrives or until a period of time (normally 500 ms) has passed.
  3. When a segment arrives with a sequence number that is expected by the receiver, and the previous in-order segment has not been acknowledged, the receiver immediately sends an ACK segment. There should not be more than two in-order unacknowledged segments at any time. This prevents the unnecessary retransmission of segments that may create congestion in the network.
  4. When a segment arrives with an out-of-order sequence number that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment. This leads to the fast retransmission of missing segments.
  5. When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected. This informs the receiver that segments reported missing have been received.
  6. If a duplicate segment arrives, the receiver discards the segment, but immediately sends an acknowledgment indicating the next in-order segment expected. This solves some problems when an ACK segment itself is lost.
- 

### Retransmission

- The heart of the error control mechanism is the retransmission of segments. When a segment is sent, it is stored in a queue until it is acknowledged. When the retransmission timer expires or when the sender receives three duplicate ACKs for the first segment in the queue, that segment is retransmitted.
  - Retransmission after RTO - The sending TCP maintains one retransmission time-out (RTO) for each connection. When the timer matures, i.e. times out, TCP resends the segment in the front of the queue (the segment with the smallest sequence number) and restarts the timer. Note that again we assume  $S_f < S_n$ . We will see later that the value of RTO is dynamic in TCP and is updated based on the round-trip time (RTT) of segments. RTT is the time needed for a segment to reach a destination and for an acknowledgment to be received.
  - Retransmission after Three Duplicate ACK Segments - The previous rule about retransmission of a segment is sufficient if the value of RTO is not large. To expedite service throughout the Internet by allowing senders to retransmit without waiting for a time out, most implementations today follow the three duplicate ACKs rule and retransmit the missing segment immediately. This feature is called fast retransmission. In this version, if three duplicate acknowledgments (i.e., an original ACK plus three exactly identical copies) arrive for a segment, the next segment is retransmitted without waiting for the time-out.
- 

### Out-of-Order Segments

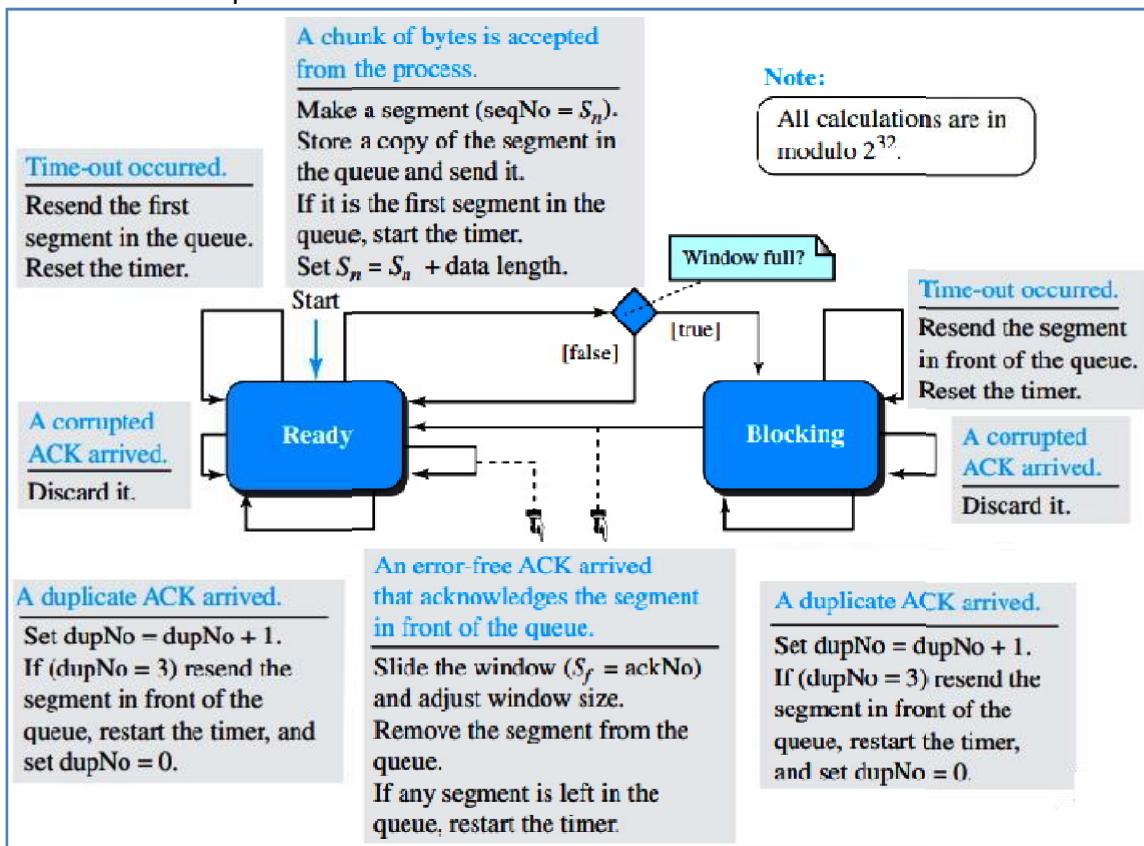
- TCP implementations today do not discard out-of-order segments. They store them temporarily and flag them as out-of-order segments until the missing segments arrive.
  - Data may arrive out of order and be temporarily stored by the receiving TCP, but TCP guarantees that no out-of-order data are delivered to the process.
- 

### FSMs for Data Transfer in TCP

- Data transfer in TCP is close to the Selective-Repeat protocol with a slight similarity to GBN. Since TCP accepts out-of-order segments, TCP can be thought of as behaving more like the SR protocol, but since the original acknowledgments are cumulative, it looks like GBN. However, if the TCP implementation uses SACKs, then TCP is closest to SR.
-

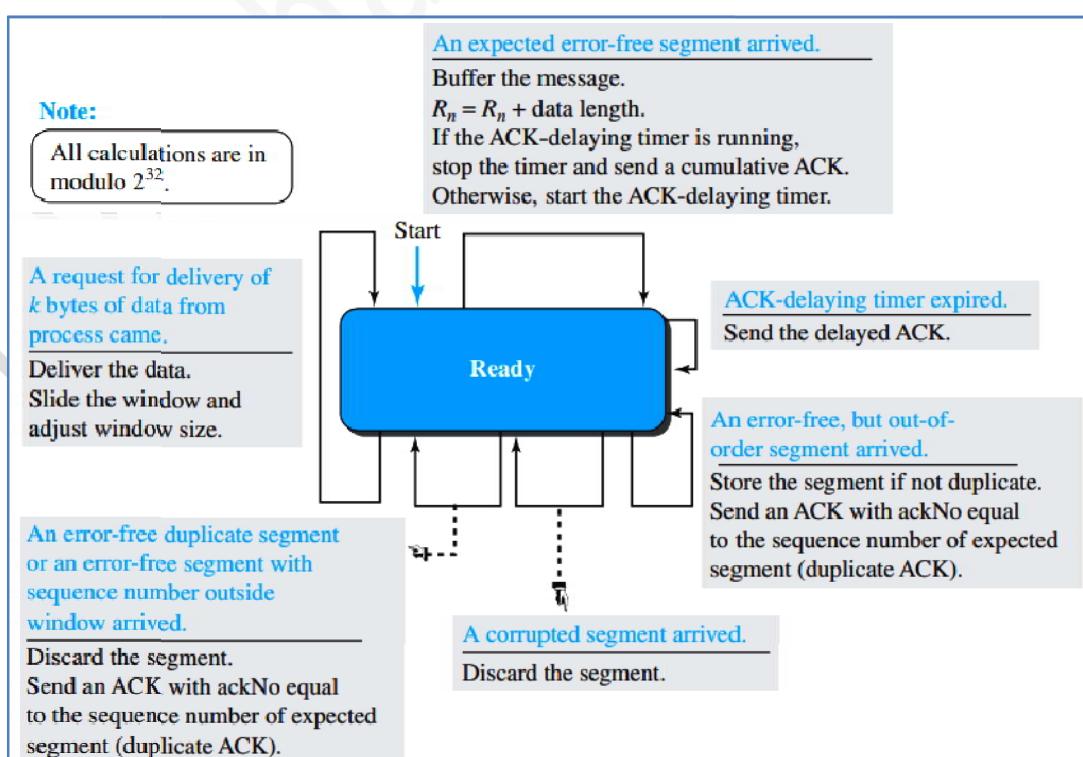
## Sender-Side FSM

- We assume that the communication is unidirectional and the segments are acknowledged using ACK segments. We also ignore selective acknowledgments and congestion control for the moment.
- Figure shows the simplified FSM for the sender site.



## Receiver-Side FSM

- Now let us show a simplified FSM for the receiver-side TCP protocol similar to the one we discuss for the SR protocol, but with some changes specific to TCP. We assume that the communication is unidirectional and the segments are acknowledged using ACK segments.



- We also ignore the selective acknowledgment and congestion control for the moment. **Figure shows the simplified FSM for the receiver.**

### Congestion control

**Topics covered - TCP uses different policies to handle the congestion in the network.**

- Congestion Window
  - Congestion Detection
  - Congestion Policies
  - Fast Recovery
  - Policy Transition
  - Additive Increase, Multiplicative Decrease
  - TCP Throughput
- 

#### **Congestion Window**

- Flow control in TCP, the **size of the send window is controlled by the receiver** using the value of **rwnd**, which is advertised in each segment traveling in the opposite direction. The use of this strategy guarantees that the receive window is never overflowed with the received bytes (no end congestion). This, however, does not mean that the intermediate buffers, buffers in the routers, do not become congested.
- A **router may receive data from more than one sender. No matter how large the buffers of a router may be**, it may be overwhelmed with data, which results in dropping some segments sent by a specific TCP sender.
- TCP **needs to worry about congestion in the middle because many segments** lost may seriously affect the error control. More segment loss means resending the same segments again, resulting in worsening the congestion, and finally the collapse of the communication.
- TCP is an **end-to-end protocol that uses the service of IP**. The congestion in the router is in the IP territory and should be taken care of by IP. IP is a simple protocol with no congestion control. TCP, itself, needs to be responsible for this problem.
- **TCP cannot ignore the congestion in the network; it cannot aggressively send segments to the network**. The result of such aggressiveness would hurt the TCP itself. TCP cannot be very conservative, either, sending a small number of segments in each time interval, because this means not utilizing the available bandwidth of the network. TCP needs to define policies that accelerate the data transmission when there is no congestion and decelerate the transmission when congestion is detected.
- **To control the number of segments to transmit, TCP uses another variable called a congestion window, cwnd**, whose size is controlled by the congestion situation in the network.
- The **cwnd (related to the congestion in the middle - network)** variable and the **rwnd** variable (**related to the congestion at the end**) together define the size of the send window in TCP.

$$\text{Actual window size} = \min(\text{rwnd}, \text{cwnd})$$


---

#### **Congestion Detection**

**How a TCP sender can detect the possible existence of congestion in the network?**

- The TCP sender uses the occurrence of **two events** as signs of congestion in the network:
  1. **Time-out**
  2. **Receiving three duplicate ACKs.**
- **Time-out**. If a TCP sender does not receive an ACK for a segment or a group of segments before the time-out occurs, it assumes that the corresponding segment or segments are lost and the loss is due to congestion.
- **Receiving of three duplicate ACKs (four ACKs with the same acknowledgment number)**. When a TCP receiver sends a duplicate ACK, it is the sign that a segment has been delayed, but sending three duplicate ACKs is the sign of a missing segment, which can be due to congestion in the

network. However, the congestion in the case of three duplicate ACKs can be less severe than in the case of time-out.

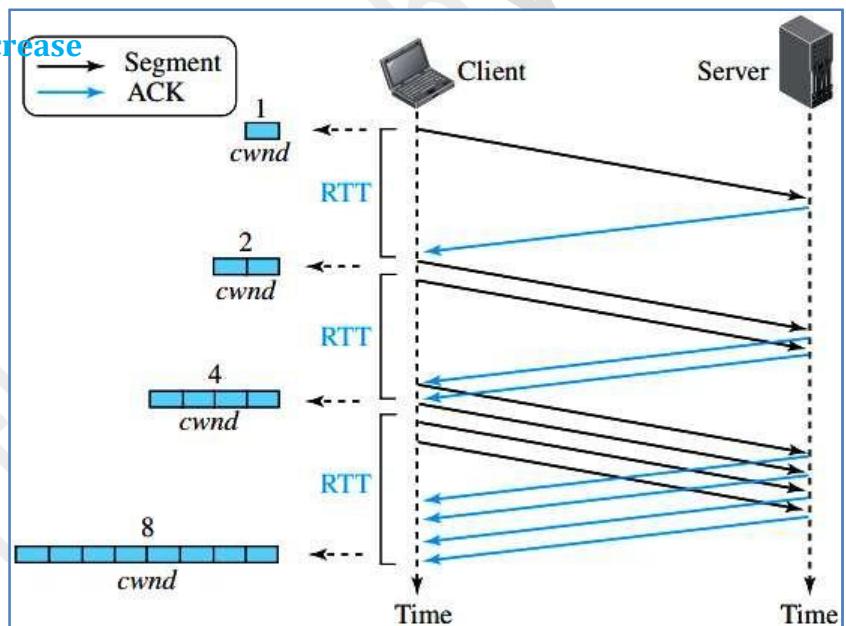
- When **a receiver sends three duplicate ACKs**, it means that one segment is missing, but three segments have been received. The network is either slightly congested or has recovered from the congestion.
- An **earlier version of TCP**, called **Tahoe TCP**, **treated both events** (time-out and three duplicate ACKs) similarly. **The later version of TCP**, called **Reno TCP**, treats these two signs differently.
- A very interesting point in TCP congestion is that the TCP sender uses only one feedback from the other end to detect congestion: **ACKs**. The lack of regular, timely receipt of **ACKs**, which results in a **time-out**, is the **sign of a strong congestion**; the **receiving of three duplicate ACKs is the sign of a weak congestion in the network**.

### Congestion Policies

- TCP's general policy for handling congestion is based on **three algorithms**
  - Slow start algorithm**
  - Congestion avoidance**
  - Fast recovery.**

#### **1. 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. The MSS is a value negotiated during the connection establishment, using an option of the same name.
- The name of this algorithm is misleading; the algorithm starts slowly, but grows exponentially. **To show the idea, let us look at Figure.**



- We assume that **rwnd** is much larger than **cwnd**, so that the sender window size always equals cwnd. We also assume that each segment is of the same size and carries MSS bytes. For simplicity, we also ignore the delayed-ACK policy and assume that each segment is acknowledged individually.
- The sender starts with  $cwnd = 1$ . This means that the sender can send only one segment. After the first ACK arrives, the acknowledged segment is purged from the window, which means there is now one empty segment slot in the window. The size of the congestion window is also increased by 1 because the arrival of the acknowledgment is a good sign that there is no congestion in the network. The size of the window is now 2. After sending two segments and receiving two individual acknowledgments for them, the size of the congestion window now becomes 4, and so on.

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

- If we look at the size of the **cwnd** in terms of **round-trip times (RTTs)**, we find that the **growth rate** is exponential in terms of each round trip time, which is a very aggressive approach:
- A slow start cannot continue indefinitely.

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	$\rightarrow cwnd = cwnd + 4 = 4 + 4 = 8 \rightarrow 2^3$

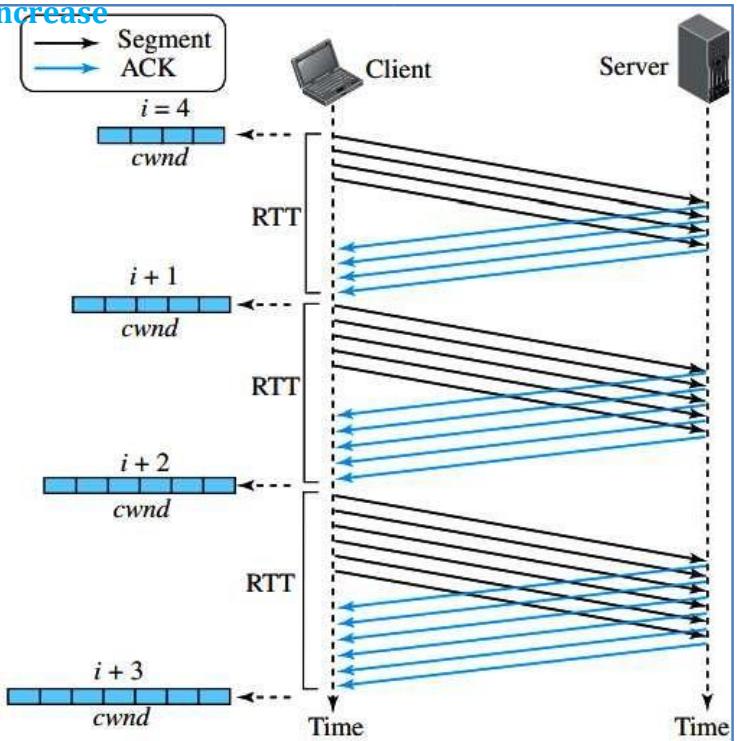
- There must be a threshold to stop this phase. The sender keeps track of a variable named *ssthresh* (slow-start threshold). When the size of the *window in bytes reaches this threshold, slow start stops and the next phase starts.*

- We need to mention that the slow-start strategy is slower in the case of delayed acknowledgments.

### 2. Congestion Avoidance: Additive Increase

- If we continue with the slow-start algorithm, the size of the congestion window increases exponentially. 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 shows the idea.



- The sender starts with *cwnd = 4*. This means that the sender can send only four segments. After four ACKs arrive, the acknowledged segments are purged from the window, which means there is now one extra empty segment slot in the window. The size of the congestion window is also increased by 1.
- The size of window is now 5. After sending five segments and receiving five acknowledgments for them, the size of the congestion window now becomes 6, and so on. The size of the congestion window in this algorithm is also a function of the number of ACKs that have arrived and can be determined as follows: If an ACK arrives,  $cwnd = cwnd + 1 / cwnd$ .
- If we look at the size of the cwnd in terms of round-trip times (RTTs), we find that the growth rate is linear in terms of each round-trip time, which is much more conservative than the slow-start approach.

Start	$\rightarrow cwnd = i$
After 1 RTT	$\rightarrow cwnd = i + 1$
After 2 RTT	$\rightarrow cwnd = i + 2$
After 3 RTT	$\rightarrow cwnd = i + 3$

### Fast Recovery

- The fast-recovery algorithm is optional in TCP. It starts when three duplicate ACKs arrive, which is interpreted as light congestion in the network. This algorithm is also an additive increase, but it increases the size of the congestion window when a duplicate ACK arrives (after the three duplicate ACKs that trigger the use of this algorithm). If a duplicate ACK arrives,  $cwnd = cwnd + 1 / cwnd$ .

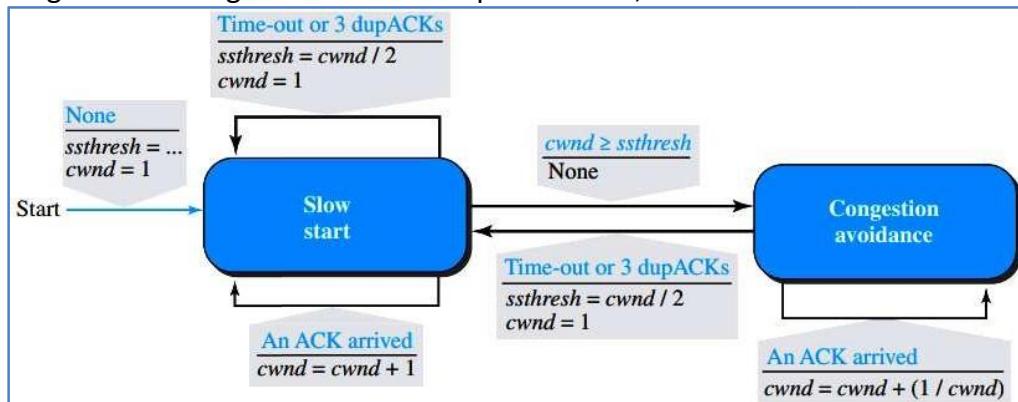
### Policy Transition

- We discussed three congestion policies in TCP. Now the question is when each of these policies is used and when TCP moves from one policy to another. To answer these questions, we need to refer to three versions of TCP:

1. Taho TCP
2. Reno TCP
3. New Reno TCP.

## Taho TCP

- The early TCP, known as Taho TCP, used only two different algorithms in their congestion policy: slow start and congestion avoidance. We use Figure to show the FSM for this version of TCP. However, we need to mention that we have deleted some small trivial actions, such as incrementing and resetting the number of duplicate ACKs, to make the FSM less crowded, simpler.

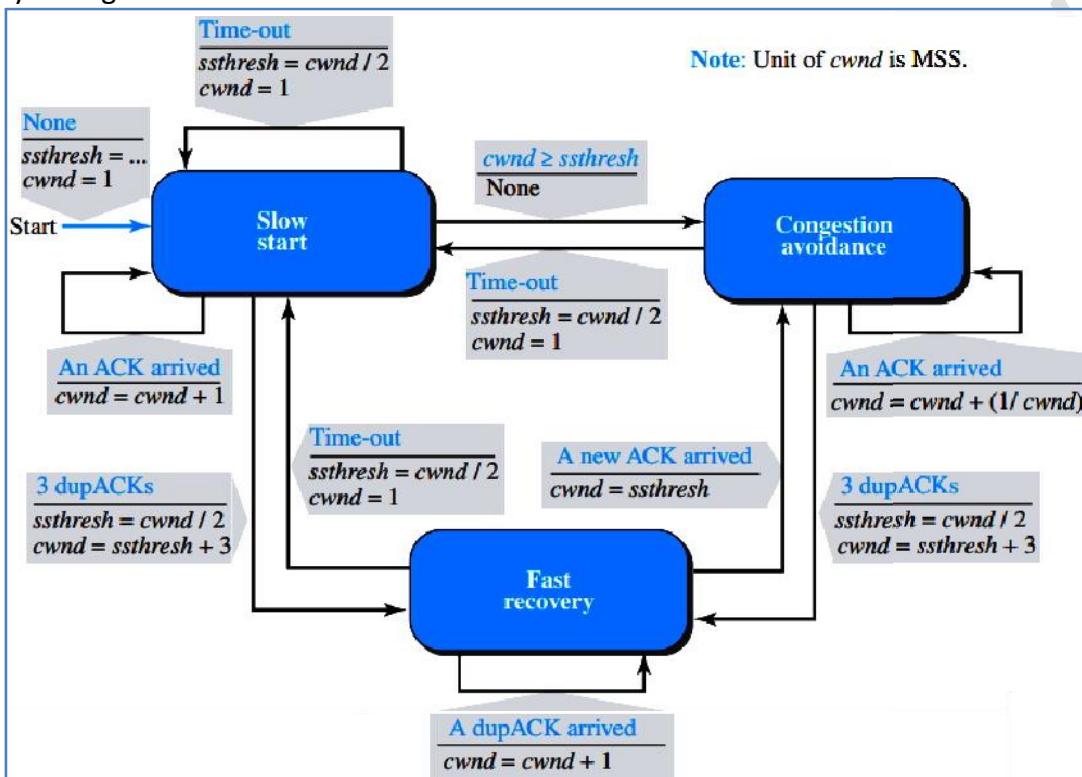


- Taho TCP** treats the two signs used for congestion detection, time-out and three duplicate ACKs, in the same way. In this version, when the connection is established, TCP starts the slow-start algorithm and sets the  $ssthresh$  variable to a pre-agreed value (normally a multiple of MSS) and the  $cwnd$  to 1 MSS. In this state, as we said before, each time an ACK arrives, the size of the congestion window is incremented by 1. We know that this policy is very aggressive and exponentially increases the size of the window, which may result in congestion.
- If congestion is detected (occurrence of time-out or arrival of three duplicate ACKs), TCP immediately interrupts this aggressive growth and restarts a new slow start algorithm by limiting the threshold to half of the current  $cwnd$  and resetting the congestion window to 1.
- If no congestion is detected while reaching the threshold, TCP learns that the ceiling of its ambition is reached; it should not continue at this speed. It moves to the congestion avoidance state and continues in that state.
- In the congestion-avoidance state, the size of the congestion window is increased by 1 each time a number of ACKs equal to the current size of the window has been received. For example, if the window size is now 5 MSS, five more ACKs should be received before the size of the window becomes 6 MSS. If congestion is detected in this state, TCP again resets the value of the  $ssthresh$  to half of the current  $cwnd$  and moves to the slow-start state again.
- Although in this version of TCP the size of  $ssthresh$  is continuously adjusted in each congestion detection. For example, if the original  $ssthresh = 8$  MSS and congestion is detected when TCP is in the congestion avoidance state and the value of the  $cwnd = 20$ , the new value of the  $ssthresh = 10$ , which means it has been increased.

## Reno TCP

- A newer version of TCP, called Reno TCP, added a new state to the congestion-control FSM, called the fast-recovery state. This version treated the two signals of congestion, time-out and the arrival of three duplicate ACKs, differently.
- In this version, if a time-out occurs, TCP moves to the slow-start state (or starts a new round if it is already in this state); on the other hand, if three duplicate ACKs arrive, TCP moves to the fast-recovery state and remains there as long as more duplicate ACKs arrive.
- The fast-recovery state is a state somewhere between the slow-start and the congestion-avoidance states. It behaves like the slow start, in which the  $cwnd$  grows exponentially, but the  $cwnd$  starts with the value of  $ssthresh$  plus 3 MSS (instead of 1).

- When TCP enters the fast-recovery state, three major events may occur.
  - If duplicate ACKs continue to arrive, TCP stays in this state, but the cwnd grows exponentially.
  - If a time-out occurs, TCP assumes that there is real congestion in the network and moves to the slow-start state.
  - If a new (nonduplicate) ACK arrives, TCP moves to the congestion-avoidance state, but deflates the size of the cwnd to the ssthresh value, as though the three duplicate ACKs have not occurred, and transition is from the slow-start state to the congestion-avoidance state.
- Figure shows the simplified FSM for Reno TCP. Again, we have removed some trivial events to simplify the figure and discussion.



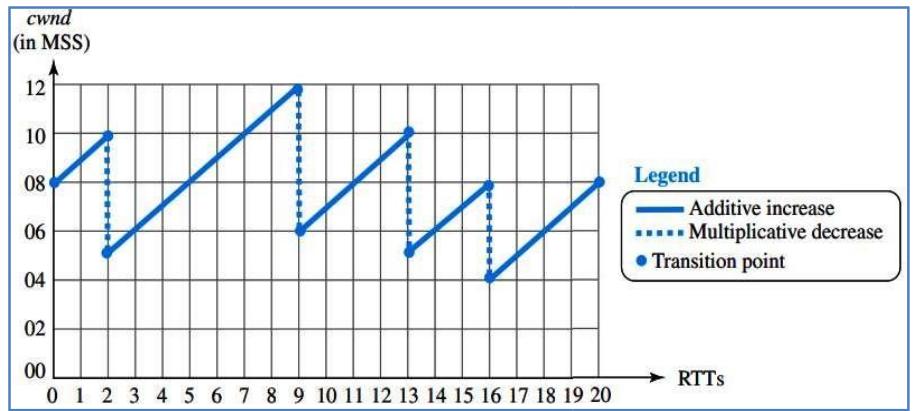
### NewReno TCP

- A later version of TCP, called NewReno TCP, made an extra optimization on the Reno TCP. In this version, TCP checks to see if more than one segment is lost in the current window when three duplicate ACKs arrive. When TCP receives three duplicate ACKs, it retransmits the lost segment until a new ACK (not duplicate) arrives.
- If the new ACK defines the end of the window when the congestion was detected, TCP is certain that only one segment was lost. However, if the ACK number defines a position between the retransmitted segment and the end of the window, it is possible that the segment defined by the ACK is also lost.
- NewReno TCP retransmits this segment to avoid receiving more and more duplicate ACKs for it.

### Additive Increase, Multiplicative Decrease

- Out of the three versions of TCP, the Reno version is most common today. 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 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.



### 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  $\text{throughput} = cwnd / \text{RTT}$ . In this assumption, TCP sends a  $cwnd$  bytes of data and receives acknowledgement for them in RTT time.
- The **behavior of TCP, as shown in Figure above, 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  $\text{throughput} = [(maximum + minimum) / 2] / \text{RTT}$ .
- However, we know that the value of the maximum is twice the value of the minimum because in each congestion detection the value of  $cwnd$  is set to half of its previous value. So the throughput can be better calculated as  $\text{throughput} = (0.75) W_{\max} / \text{RTT}$  in which  $W_{\max}$  is the average of windowsizes when the congestion occurs.

### Example

If  $MSS = 10 \text{ KB}$  (kilobytes) and  $\text{RTT} = 100 \text{ ms}$  in Figure, we can calculate the throughput as

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

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