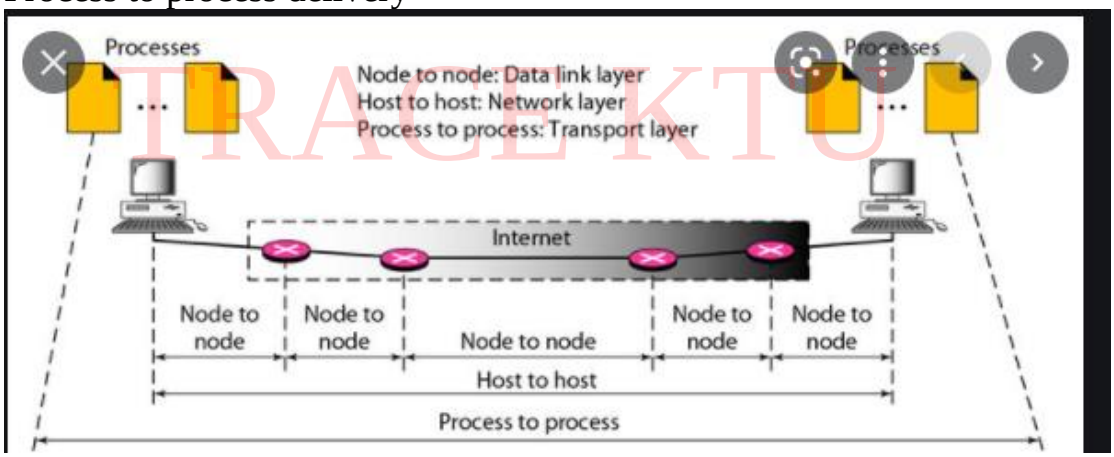
# MODULE 5 (part 1)

**Module – 5 (Transport Layer and Application Layer)**

Transport service – Services provided to the upper layers, Transport service primitives. User Datagram Protocol (UDP). Transmission Control Protocol (TCP) – Overview of TCP, TCP segment header, Connection establishment &release, Connection management modeling, TCP retransmission policy, TCP congestion control.

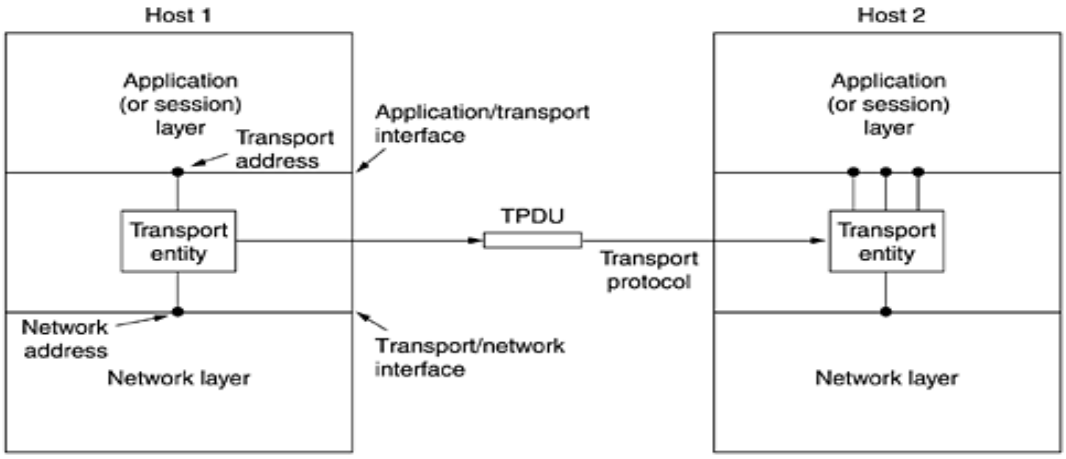
Application Layer –File Transfer Protocol (FTP), Domain Name System (DNS), Electronic mail, Multipurpose Internet Mail Extension (MIME), Simple Network Management Protocol (SNMP), World Wide Web(WWW) – Architectural overview.

## TRANSPORT LAYER

- Heart of the whole protocol hierarchy
- Ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective service to its users, normally processes in the application layer.
- Process to process delivery
- 


## Services provided to the upper layer

- The hardware and/or software within the transport layer that does the work is called the transport entity.
- Features
- Two types of transport service.
- connection-oriented transport service:The connection-oriented transport service has three phases: establishment, data transfer, and release.
- connection-less transport service
- Addressing and flow control




**Transport Service Primitives**

- To allow users to access the transport service, the transport layer must provide some operations to application programs, that is, a transport service interface.
- Each transport service has its own interface.

- purpose of the transport layer to provide a reliable service on top of an unreliable network
- consider an application with a server and a number of remote clients. To start with, the server executes a LISTEN primitive, typically by calling a library procedure that makes a system call to block the server until a client turns up.
- When a client wants to talk to the server, it executes a CONNECT primitive.
- The transport entity carries out this primitive by blocking the caller and sending a packet to the server.
- Encapsulated in the payload of this packet is a transport layer message for the server's transport entity.
- **TPDU** (**Transport Protocol Data Unit**) for messages sent from transport entity to transport entity.
- the client's CONNECT call causes a CONNECTION
- REQUEST TPDU to be sent to the server. When it arrives, the transport entity checks to see that the server is blocked on a LISTEN (i.e., is interested in handling requests). It then unblocks the server and sends a CONNECTION ACCEPTED TPDU back to the client. When this TPDU arrives, the client is unblocked and the connection is established.

- Data can now be exchanged using the SEND and RECEIVE primitives.
- When a connection is no longer needed, it must be released to free up table space within the two transport entities. Disconnection has two variants: asymmetric and symmetric.
- In the asymmetric variant, either transport user can issue a DISCONNECT primitive, which results in a DISCONNECT TPDU being sent to the remote transport entity. Upon arrival, the connection is released.
- In the symmetric variant, each direction is closed separately, independently of the other one. When one side does a DISCONNECT, that means it has no more data to send but it is still willing to accept data from its partner. In this model, a connection is released when both sides have done a DISCONNECT.

| Primitive | Packet sent | Meaning |
|---|---|---|
| LISTEN | (none) | Block until some process tries to connect |
| CONNECT | CONNECTION REQ. | Actively attempt to establish a connection |
| SEND | DATA | Send information |
| RECEIVE | (none) | Block until a DATA packet arrives |
| DISCONNECT | DISCONNECTION REQ. | This side wants to release the connection |



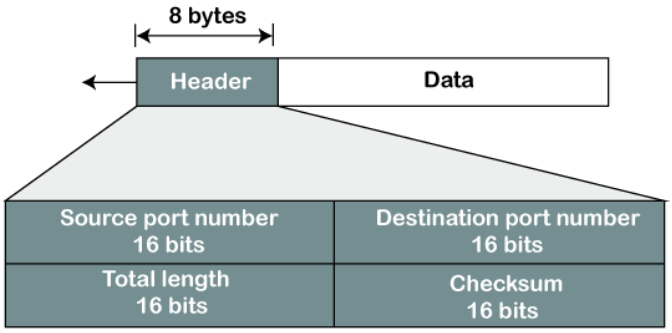
**Internet Transport Protocols:**

**UDP (User Datagram Protocol)**

- **connectionless** transport protocol
- UDP is basically just IP with a short header added
- provides a way for applications to send encapsulated IP datagrams and send them without having to establish a connection
- UDP transmits segments consisting of an 8-byte header followed by the payload
- **Ordered delivery of data is not guaranteed:** In the case of UDP, the datagrams are sent in some order will be received in the same order is not guaranteed as the datagrams are not numbered.
- If a process wants to send a small message and does not care much about reliability, it can use UDP.
- UDP uses special address to identify the target process – PORT numbers.

- **Ports:**The UDP protocol uses different port numbers so that the data can be sent to the correct destination. The port numbers are defined between 0 and 1023.
- **Stateless:** It is a stateless protocol that means that the sender does not get the acknowledgement for the packet which has been sent.
- **unreliable protocol**

## UDP HEADER FORMAT



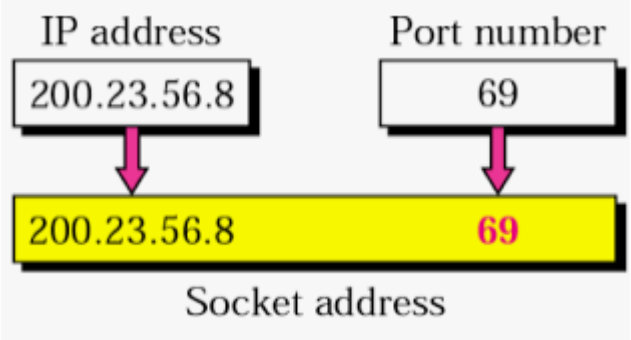

The size of the data that the UDP packet can carry would be 65,535 minus 28 as 8 bytes for the header of the UDP packet and 20 bytes for IP header.



**The UDP header contains four fields:**

- **Source port number:** It is 16-bit information that identifies which port is going to send the packet.
- **Destination port number:** It identifies which port is going to accept the information. It is 16-bit information which is used to identify application-level service on the destination machine.
- If the source host is the client (a client sending a request), the port number, in most cases, is an **ephemeral port number** requested by the process and chosen by the UDP software running on the source host. An ephemeral port number is greater than 1023. If the source host is the server (a server sending a response), the port number, in most cases, is a **well-known port number**. Universal port numbers are assigned for servers. The range of well known port number is from 0 to 1023.
- The destination IP address defines the host among the different host. After the host has been selected, the port number defines one of the processes on this particular host.

- The combination of IP address and port number is called **socket address.**




- **Length:** It is 16-bit field that specifies the entire length of the UDP packet that includes the header also. The minimum value would be 8-byte as the size of the header is 8 bytes.
- **Checksum:** It is a 16-bits field, and it is an optional field. This checksum field checks whether the information is accurate or not as there is the possibility that the information can be corrupted while transmission. It is an optional field, which means that it depends upon the application, whether it wants to write the checksum or not. If it does not want to write the checksum, then all the 16 bits are zero; otherwise, it writes the checksum. In UDP, the checksum field is applied to the entire packet, i.e., header as well as data part whereas, in IP, the checksum field is applied to only the header field.



**Applications:**

- client-server :client sends a short request to the server and expects a short reply back
- Eg: DNS, RPC, RTP
- **DNS**
  - program that needs to look up the IP address of some host name can send a UDP packet containing the host name to a DNS server.
  - The server replies with a UDP packet containing the host's IP address.
  - No setup is needed in advance and no release is needed afterward.
- **Remote Procedure Call (RPC)**
  - When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2.
  - Information can be transported from the caller to the callee in the parameters and can come back in the procedure result.

- No message passing is visible to the programmer.
- This technique is known as RPC (Remote Procedure Call)

- UDP is widely used in another area called real-time multimedia applications
- Egs:Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand
- For generic real-time transport protocol for multiple applications called RTP (Real-time Transport Protocol) is usd
- basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets

## TCP (Transmission control protocol)

- Transmission control protocol is one that offers a reliable, connection-oriented, byte-stream service

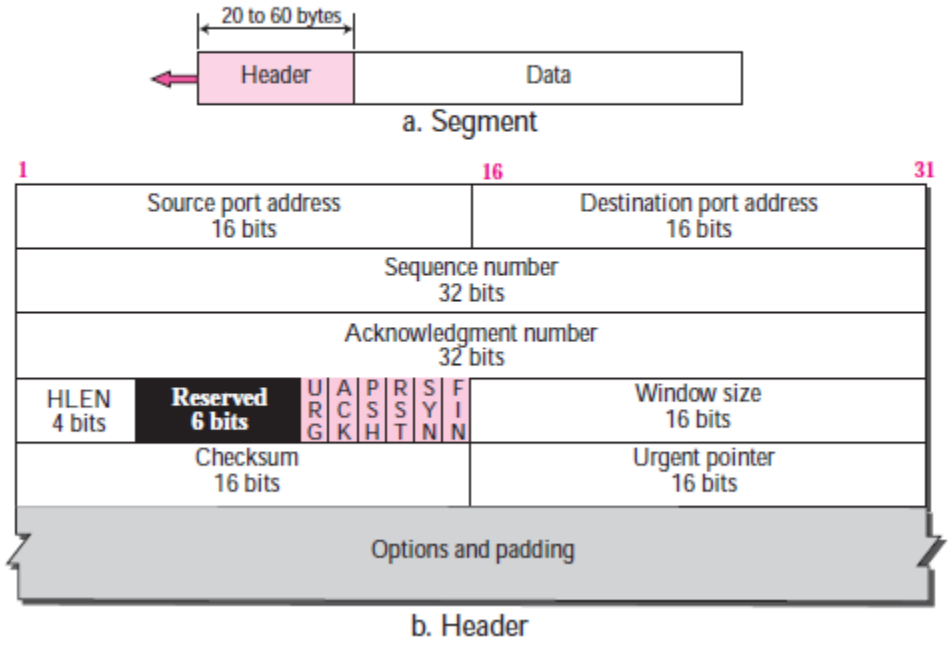
**TCP services**

- **Process-to-Process Communication**: TCP provides process-to-process communication using port numbers
- **TCP is a stream-oriented protocol:** TCP allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes.
- The sending process produces (writes to) the stream of bytes and the receiving process consumes (reads from) them.
- TCP groups a number of bytes together into a packet called a ***segment.*** TCP adds a header to each segment (for control purposes) and delivers the segment to the IP layer for transmission
- TCP offers **full-duplex service**, where data can flow in both directions at the same time.
- TCP performs multiplexing at the sender and demultiplexing at the receiver
- **Connection oriented** service :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 TCPs establish a virtual connection between them.
  **2.** Data are exchanged in both directions.
  **3.** The connection is terminated.

  Note that this is a virtual connection, not a physical connection
- **Reliable Service:** TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data

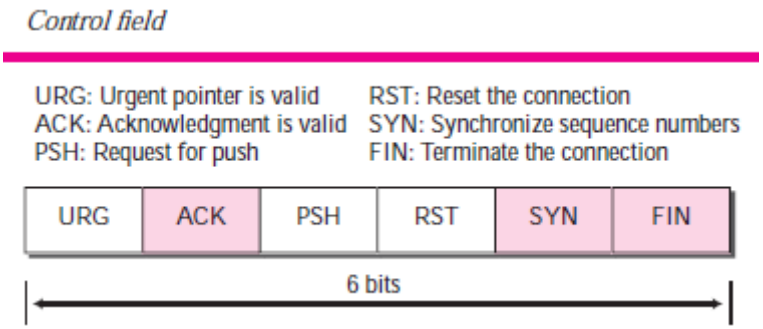
## Segment Format

Figure 15.5 *TCP segment format*






- **Source port address.** This is a 16-bit field that defines the port number of the application program in the host that is sending the segment
- **Destination port address.** This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment
- **Sequence number.** This 32-bit field defines the number assigned to the first byte of data contained in this segment
- **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.
- **Reserved**. This is a 6-bit field reserved for future use.
- **Control.** This field defines 6 different control bits or flags. 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.

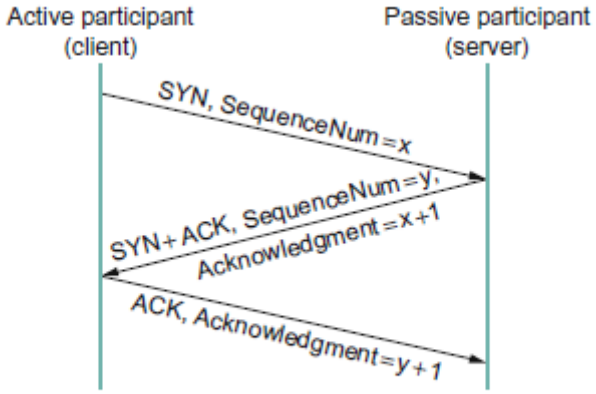



- The SYN and FIN flags are used when establishing and terminating a TCP connection, respectively.
- **SYN bit**
- used to establish connections.
- The CONNECTION REQUEST has SYN = 1 and ACK = 0 to indicate that the piggyback acknowledgement field is not in use.
- The CONNECTION ACCEPTED reply does bear an acknowledgement, so it has SYN = 1 and ACK = 1.
- **FIN bit**
- used to release a connection.
- It specifies that the sender has no more data to transmit.
- TheURGflag signifies that this segment contains urgent data. When this flag is set, the UrgPtr field indicates where the non urgent data contained in this segment begins. The urgent data is contained at the front of the segment body, up to and including a value of UrgPtr bytes into the segment.
- **URG bi**t set to 1 if the Urgent pointer is in use.
- The Urgent pointer is used to indicate a byte offset from the current sequence number at which urgent data are to be found.
- **ACK bit**
- set to 1 to indicate that the Acknowledgement number is valid.
- If ACK is 0, the segment does not contain an acknowledgement so the Acknowledgement number field is ignored.
- 
- The **PUSH flag** signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process that The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received
- **RST bit**
- used to reset a connection that has become confused due to a host crash or some other reason.
- also used to reject an invalid segment or refuse an attempt to open a connection.

- **Window size.** This field defines the window size of the sending TCP in bytes
- **Checksum.** This 16-bit field contains the checksum
- **Urgent pointer**. This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data
- **Options.** There can be up to 40 bytes of optional information in the TCP header
- most important option is the one that allows each host to specify the maximum TCP payload it is willing to accept.
- Another option proposed and now widely implemented is the use of the selective repeat instead of go back n protocol.

**Connection Establishment and Termination**

- A TCP connection begins with a client (caller) doing an active open to a server (callee). Assuming that the server had earlier done a passive open, the two sides engage in an exchange of messages to establish the connection.
- Only after this connection establishment phase is over do the two sides begin sending data
- Connection setup is an asymmetric activity (one side does a passive open and the other side does an active open), connection teardown is symmetric (each side has to close the connection independently).
- **Three way handshake**
- The algorithm used by TCP to establish and terminate a connection is called a three-way handshake**.**
- 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*
- 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.
- **Connection establishment**



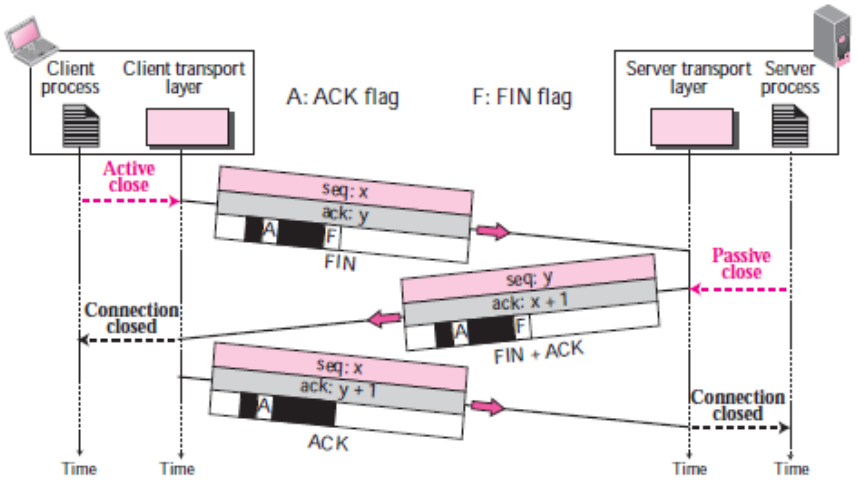

6 Timeline for three-way handshake algorithm.

- First, the client (the active participant) sends a segment to the server (the passive participant) stating the initial sequence number it plans to use (Flags = SYN, SequenceNum = x).
- The server then responds with a single segment that both acknowledges the client's sequence number (Flags = ACK, Ack = x+1) and states its own beginning sequence number (Flags = SYN, SequenceNum = y). both the SYN and ACK bits are set in the Flags field of this second message.
- Finally, the client responds with a third segment that acknowledges the server's sequence number (Flags = ACK, Ack = y +1).



- **Connection termination**

Figure 15.11 *Connection termination using three-way handshaking*




- In a common 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.
- 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.

- 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.

## TCP CONNECTION MANAGEMENT MODELLING

### State-Transition Diagram

- This diagram shows only the states involved in opening a connection (everything above ESTABLISHED) and in closing a connection (everything below ESTABLISHED).
- All the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine

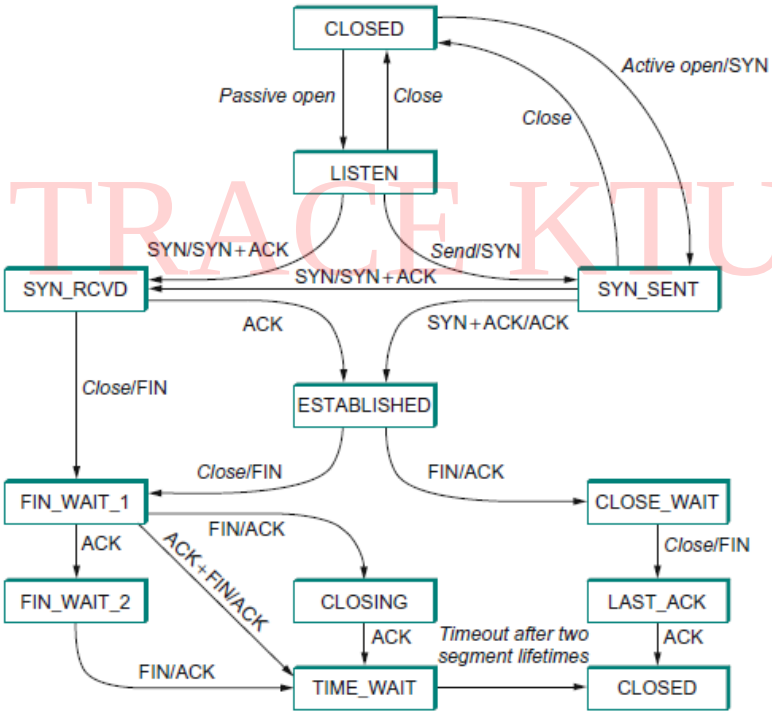



FIGURE 5.7 TCP state-transition diagram.

- Each circle denotes a state that one end of a TCP connection can find itself in.
- All connections start in the CLOSED state. As the connection progresses, the connection moves from state to state according to the arcs. Each arc is labelled with a tag of the form *event/action*.
- When opening a connection, the server first invokes a passive open operation on TCP, which causes TCP to move to the LISTEN state

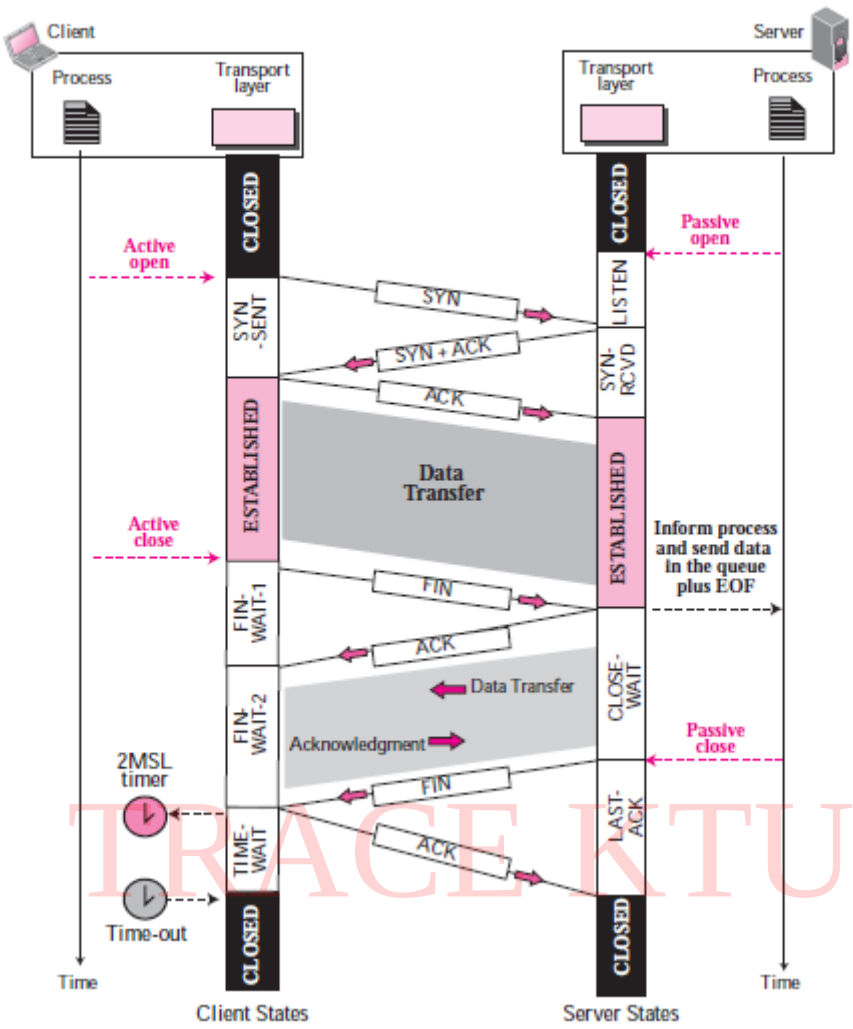
- If a connection is in the LISTEN state and a SYN segment arrives (i.e., a segment with the SYN flag set), the connection makes a transition to the SYN RCVD state and take the action of replying with an ACK+SYN segment.

**Table 15.2** *States for TCP*

| *State* | *Description* |
|---|---|
| CLOSED | No connection exists |
| LISTEN | Passive open received; waiting for SYN |
| SYN-SENT | SYN sent; waiting for ACK |
| SYN-RCVD | SYN+ACK sent; waiting for ACK |
| ESTABLISHED | Connection established; data transfer in progress |
| FIN-WAIT-1 | First FIN sent; waiting for ACK |
| FIN-WAIT-2 | ACK to first FIN received; waiting for second FIN |
| CLOSE-WAIT | First FIN received, ACK sent; waiting for application to close |
| TIME-WAIT | Second FIN received, ACK sent; waiting for 2MSL time-out |
| LAST-ACK | Second FIN sent; waiting for ACK |
| CLOSING | Both sides decided to close simultaneously |

- The client does an active open, which causes its end of the connection to send a SYN segment to the server and to move to the SYN SENT state.
- When the SYN segment arrives at the server, it moves to the SYN RCVD state and responds with a SYN+ACK segment. The arrival of this segment causes the client to move to the ESTABLISHED state and to send an ACK back to the server. When this ACK arrives, the server finally moves to the ESTABLISHED state.
- The application process on both sides of the connection must independently close its half of the connection. If only one side closes the connection, then this means it has no more data to send, but it is still available to receive data from the other side.
- This side closes first: ESTABLISHED→FIN WAIT 1→FIN WAIT 2 →TIME WAIT→CLOSED.
- The other side closes first: ESTABLISHED→CLOSE WAIT→ LAST ACK→CLOSED.
- Both sides close at the same time: ESTABLISHED→FIN WAIT 1→ CLOSING→TIME WAIT→CLOSED.(simultaneous close)
- TIME WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the Internet (i.e., 120 seconds

**Figure 15.15** *Time-line diagram for connection establishment and half-close termination*






## TCP Transmission Policy

### Triggering Transmission

- how TCP decides to transmit a segment
- TCP has three mechanisms to trigger the transmission of a segment

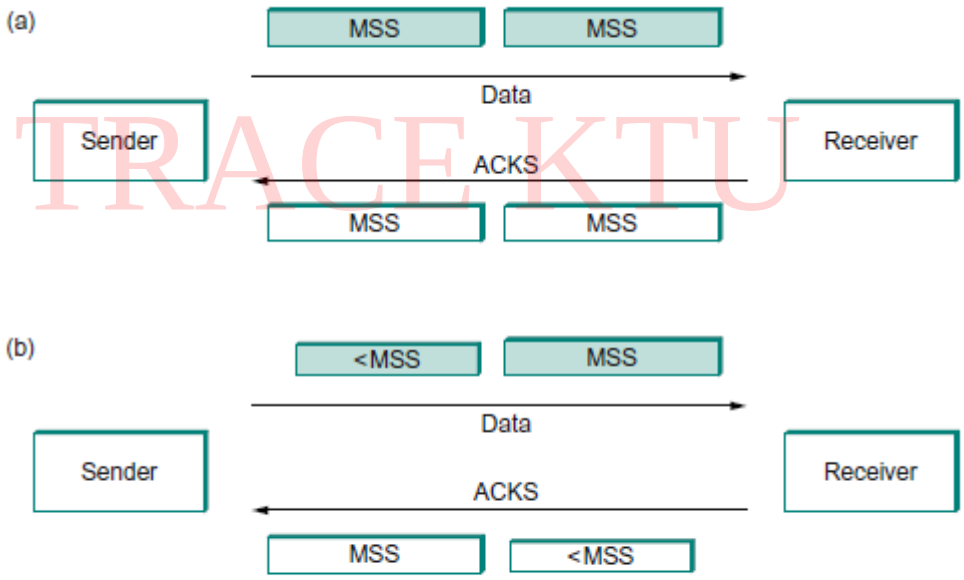
1. TCP maintains a variable, typically called the *maximum segment size* (MSS), and it sends a segment as soon as it has collected MSS bytes from the sending process.
   - MSS is usually set to the size of the largest segment TCP can send without causing the local IP to fragment.
   - MSS is set to the maximum transmission unit (MTU) of the directly connected network, minus the size of the TCP and IP headers
2. Secondly, Triggers TCP to transmit a segment is that the sending process has explicitly asked it to do so. Specifically, TCP supports

a *push* operation, and the sending process invokes this operation to effectively flush the buffer of unsent bytes.

3. The final trigger for transmitting a segment is that a timer fires; the resulting segment contains as many bytes as are currently buffered for transmission

- **Silly window syndrome**

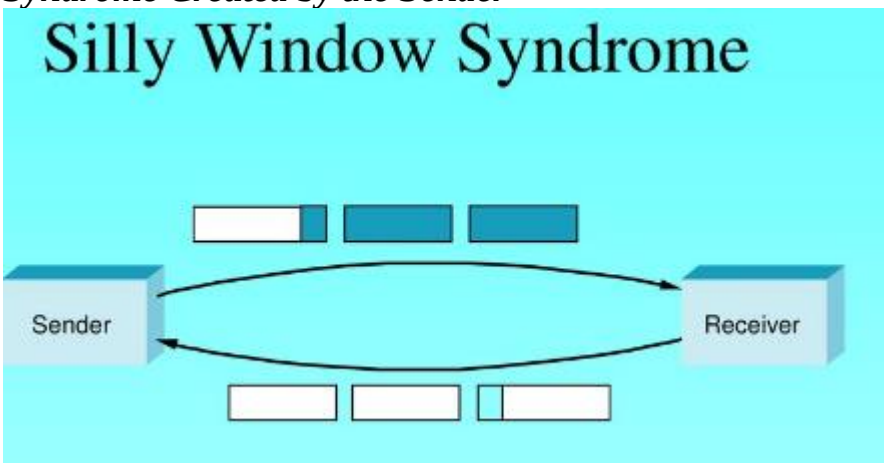
- When either the sending application creates data slowly or receiving application consumes data slowly silly window syndrome occurs.
- For example, if TCP sends segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data. Here the overhead is 41/1, which indicates that we are using the capacity of the network very inefficiently
- MSS-sized segments correspond to large segments and 1-byte segments correspond to very small segments






■ **FIGURE 5.9** Silly window syndrome. (a) As long as the sender sends MSS-sized segments and the receiver ACKs one MSS at a time, the system works smoothly. (b) As soon as the sender sends less than one MSS, or the receiver ACKs less than one MSS, a small "container" enters the system and continues to circulate.

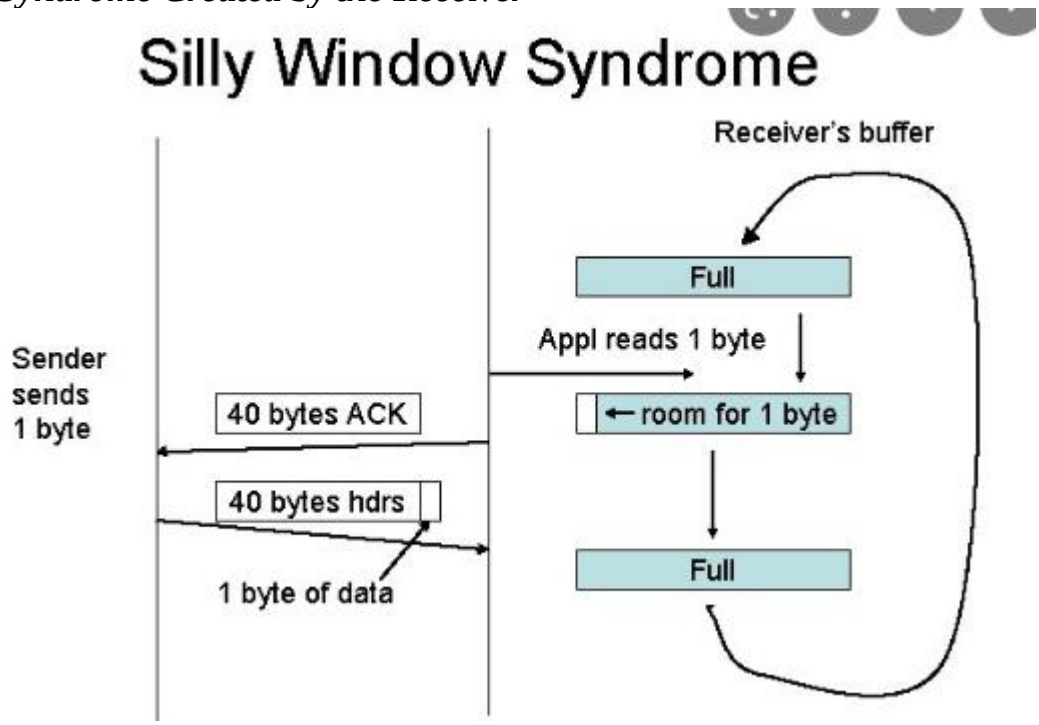
- ***Syndrome Created by the Sender***



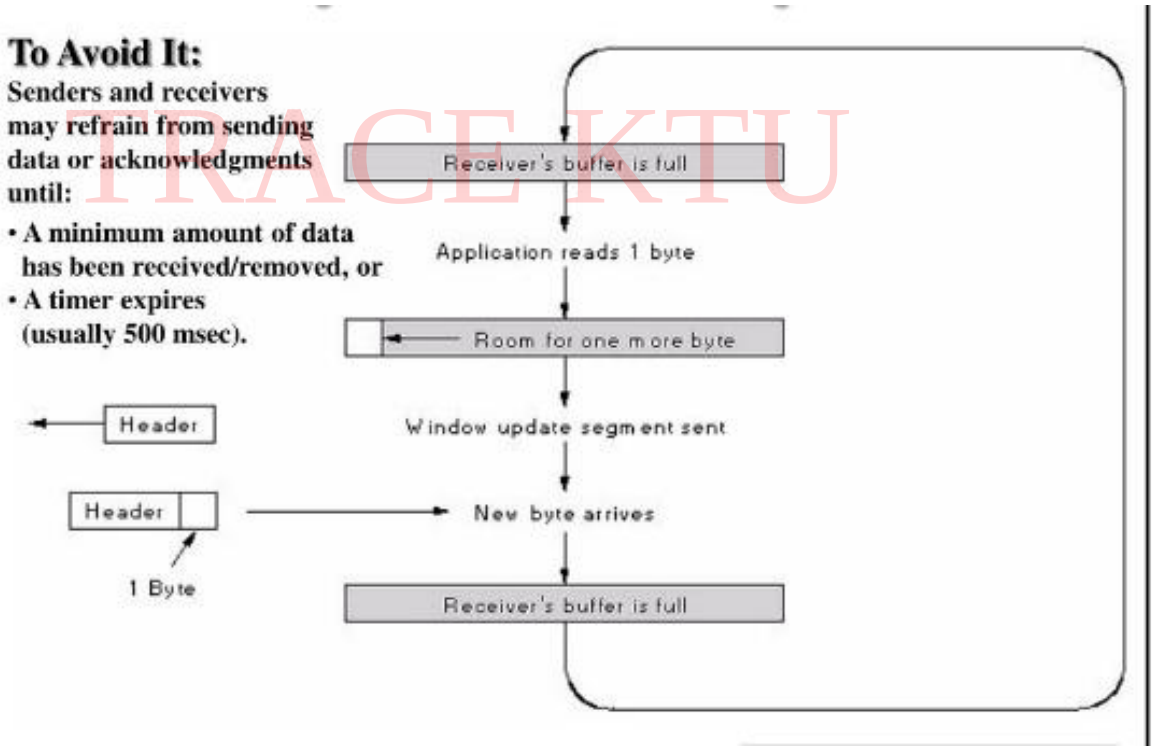

- The sending TCP may create a silly window syndrome if it is serving an application program that creates data slowly, for example, 1 byte at a time. The application program writes 1 byte at a time into the buffer of the sending TCP
- The solution is to prevent the sending TCP from sending the data byte by byte. The sending TCP must be forced to wait and collect data to send in a larger block.
- **Nagle's algorithm**
- Nagle introduced an elegant *self-clocking* solution
- 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 has 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

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

- ***Syndrome Created by the Receiver***




- **or**






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

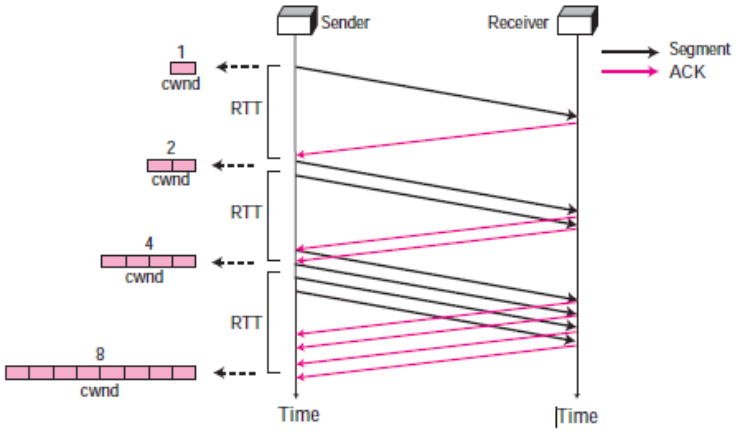


## TCP Congestion Control

- The idea of TCP congestion control is for each source to determine how much capacity is available in the network, so that it knows how many packets it can safely have in transit
- Congestion occurs in a network if the load (no of packet sent to the network) on the network greater than the capacity.
- TCP maintains a new state variable for each connection, called Congestion Window, (CWND) which is used by the source to limit how much data it is allowed to have in transit at a given time.
- TCP congestion control is implemented by 3 phases
- **Slow start: Exponential increase**

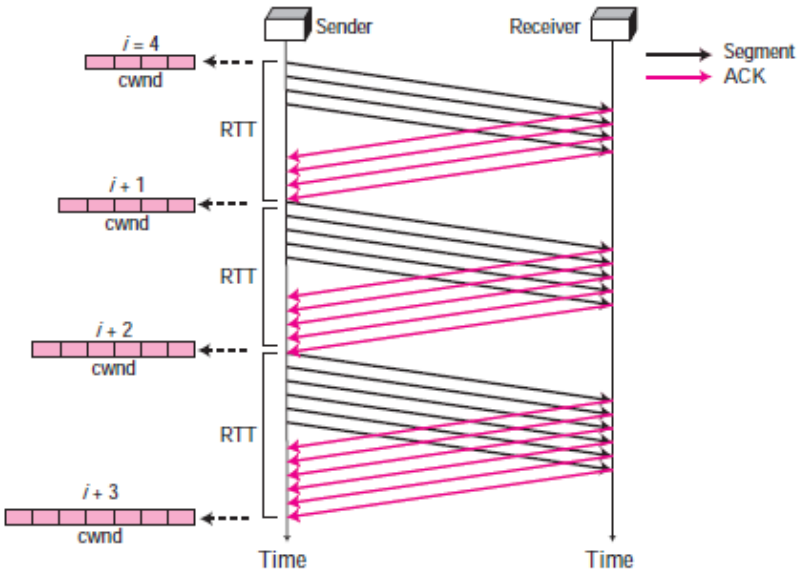
- The slow start algorithm is based on the idea that the size of the congestion window (cwnd) starts with one maximum segment size (MSS).
- The MSS is determined during connection establishment
- The size of the window increases one MSS each time one acknowledgement arrives. As the name implies, the algorithm starts slowly, but grows exponentially.

**Figure 15.34** *Slow start, exponential increase*




- The sender starts with cwnd = 1 MSS. This means that the sender can send only one segment. After the first ACK arrives, the size of the congestion window is increased by 1, which means that cwnd is now 2. Now two more segments can be sent. When two more ACKs arrive, the size of the window is increased by 1 MSS for each ACK, which means cwnd is now 4. Now four more segments can be sent. When four ACKs arrive, the size of the window increases by 4, which means that cwnd is now 8. So on
- Slow start cannot continue indefinitely. There must be a threshold to stop this phase.
- The sender keeps track of a variable named *ssthresh* (slow start threshold) or congestion threshold. When the size of window in bytes reaches this threshold, slow start stops and the next phase starts.
- **Congestion avoidance: Additive increase**
- If we start with the slow start algorithm, the size of the congestion window increases exponentially. To avoid congestion before it happens, one must slow down this exponential growth. TCP defines another algorithm called congestion avoidance, which increases the cwnd additively.

**Figure 15.35** *Congestion avoidance, additive increase*




- 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.
- That is, rather than incrementing CongestionWindow by an entire MSS bytes each RTT, we increment it by a fraction of MSS every time an ACK is received. Assuming that each ACK acknowledges the receipt ofMSSbytes, then that fraction is MSS/CongestionWindow

  Increment = MSS×(MSS/CongestionWindow)
  CongestionWindow+ = Increment
- ***Congestion Detection: Multiplicative Decrease***
- If congestion occurs, the congestion window size must be decreased.
- The only way a sender can guess that congestion has occurred is the need to retransmit a segment. This is a major assumption made by TCP retransmission can occur in one of two cases: when the RTO timer times out or when three duplicate ACKs are received.

- In both cases, the size of the threshold is dropped to half (**multiplicative decrease**). **1.** If a time-out occurs, there is a stronger possibility of congestion;
  - A segment has probably been dropped in the network and there is no news about the following sent segments. In this case TCP reacts strongly:
  - Steps

**a.** It sets the value of the threshold to half of the current window size.
**b.** It reduces cwnd back to one segment.
**c.** It starts the slow start phase again.