**MODULE IV**

End to End Protocol, Simple de-multiplexer, Reliable Byte stream, TCP-Issues, segment format, connection establishment and termination, sliding window revisited, triggering transmission, adaptive retransmission, RPC-fundamentals ,TCP Congestion control –additive increase, slow start, fast retransmit and fast recovery, congestion avoidance mechanism, DEC bit, Random Early Detection bit, Source Based Congestion avoidance

**INTRODUCTION**

The role of the transport layer is to turn the host-to-host packet delivery service into a process-to-process communication channel. Some of the common properties that a transport protocol can be expected to provide:

* Guarantees message delivery.
* Delivers messages in the same order they are sent.
* Delivers at most one copy of each message.
* Supports arbitrarily large messages.
* Supports synchronization between the sender and the receiver.
* Allows the receiver to apply flow control to the sender.
* Supports multiple application processes on each host.

Some of the typical limitations of the network upon which the transport protocol operates are that it may

* Drop messages.
* Reorder messages.
* Deliver duplicate copies of a given message.
* Limit messages to some finite size.
* Deliver messages after an arbitrarily long delay

The challenge is to develop algorithms that turn the less-than-desirable properties of the underlying network into the high level of service required by application programs.

**SIMPLE DE-MULTIPLEXER (UDP)**

* User Datagram Protocol
* Extends the host-to-host delivery service of the underlying network into a process-to-process communication service.
* Adds a level of demultiplexing, thereby allowing multiple application processes on each host to share the network
* UDP adds multiplexing by using a port number to identify endpoints within each host. Ports can be dedicated, such as ftp and telnet, or assigned during the application setup processes. Even though the port number gives the appearance of a connection, UDP offers no other enhancements to the simple datagram services provided by IP.

**UDP Header Format**



**SrcPort (Source port number):** This is the port number used by the process running on the source host. It is 16 bits long, which means that the port number can range from 0 to 65,535. If the source host is the client (a client sending a request), the port number, in most cases, is an ephemeral port number requested by the process and chosen by the UDP software running on the source host. If the source host is the server (a server sending a response), the port number, in most cases, is a well-known port number.

**DstPort(Destination port number):**  This is the port number used by the process running on the destination host. It is also 16 bits long. If the destination host is the server (a client sending a request), the port number, in most cases, is a well-known port number. If the destination host is the client (a server sending a response), the port number, in most cases, is an ephemeral port number. In this case, the server copies the ephemeral port number it has received in the request packet

**Length:** This is a 16-bit field that defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be much less because a UDP user datagram is stored in an IP datagram with a total length of 65,535 bytes.

**Checksum:** This field is used to detect errors over the entire user datagram (header plus data).

**RELIABLE BYTE STREAM (TCP)**

TCP offers the following services:

* Reliable
* Connection Oriented
* Byte Stream
* TCP guarantees the reliable, in-order delivery of a stream of bytes
* It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction.
* It includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit at a given time.
* TCP supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers.
* TCP implements a highly-tuned congestion-control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from overrunning the receiver, but to keep the sender from overloading the network.

**TCP SEGMENT FORMAT**

* TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection.
* TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host. TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer
* The packets exchanged between TCP peers in are called *segments*, since each one carries a segment of the byte stream



Fig: How TCP manages byte stream

**TCP Header Format**



Fig: TCP Header Format

**SrcPort:** This is a 16-bit field that defines the port number of the application program in the host that is sending the segment. This serves the same purpose as the source port address in the UDP header.

**DstPort:** This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment. This serves the same purpose as the destination port address in the UDP header.

**SequenceNum:** This 32-bit field defines the number assigned to the first byte of data contained in this segment. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence comprises 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:** This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number *x* from the other party, it defines *x* + 1 as the acknowledgment number. Acknowledgment and data can be piggybacked together.

**HdrLen:** 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.

**Flags:** This field defines 6 different control bits or flags. One or more of these bits can be set at a time.



Fig: Description of Flags

**Advertised Window:** This field defines the size of the window, in bytes, that the other party must maintain. 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.

**Checksum:**

**UrgPtr**: This l6-bit field, which is valid, only if the urgent flag is set, is used when the segment contains urgent data. It defines the number that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.

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

**CONNECTION ESTABLISHMENT AND TERMINATION**

TCP uses an algorithm called the **Three-way handshake** during setup and tear down of connections. In the simplest form the originator of the connection (client) sends a segment to the destination (server) with flag SYN and the planned initial sequence number for the connection. The server then responds with a segment that both acknowledges the client’s sequence number and states the servers planned initial sequence number (SYN and ACK flags). The client then acknowledges the server’s sequence number (ACK flag). The reason for starting at a specific sequence number, actually selected at random, is that each connection will be easily distinguished from others that may have timed out during the setup procedure but used the same port and ip information. If 0 were used at the start it would be possible for the client to give up and attempt another connection and the server to belatedly acknowledge the first. In addition to the sequence number, the minimum information necessary to establish a connection, other optional parameters can be passed back and forth at this time to further customize the connections**.**



Fig: Timeline for three-way handshake algorithm

**State Transition Algorithm**



Fig: TCP state transition diagram

This state diagram focuses on the setup (above the ESTABLISHED state) and teardown (below the ESTABLISHED state). During the ESTABLISHED state there is end-to-end data transfer. All connections start in CLOSED. This diagram is for one end of a connection. There are corresponding states for the other physical end of the connection. For example a server would use a passive open when ready to accept connections and would enter the LISTEN state. If while in LISTEN, a SYN segment arrives the state goes to SYN\_RCVD. An ACK+SYN is sent to the client. If in the SYN\_RCVD state an ACK is received from the client (Third hand shake) the connection advances to ESTABLISHED.

The right side of the diagram is for clients. A client in closed issues a SYN to the server and enters SYN\_SENT. While in SYN\_SENT an ACK+SYN received will cause an ACK to be sent to the server and the connection will enter the ESTABLISHED state.

The left lower side of the diagram is for tear-down originating locally. The right side is for tear-downs initiated from the remote end of the connection. This diagram is busy enough but in fact the timeouts that would resolve lost segments during setup or tear-down are not shown. Each state change is accompanied by a timer being set. If the timer times out there is a re-transmission interval and an abandonment interval allowing an orderly recovery from network problems.

**TCP’s SLIDING WINDOW**

TCP’s variant of the sliding window algorithm, serves several purposes:

(1) it guarantees the reliable delivery of data,

(2) it ensures that data is delivered in order, and

(3) it enforces flow control between the sender and the receiver.



Fig: Relationship between (a) TCP send buffer (b) TCP receive buffer

* TCP on the sending side maintains a *send* buffer that is divided into 3 segments namely acknowledged data, unacknowledged data and data to be transmitted
* Similarly TCP on the receiving side maintains a *receive* buffer to hold data even if it arrives of order.
* The send buffer maintains three variables namely LastByteAcked, LastByteSent, and LastByteWritten as shown above. The relation between them is obvious

LastByteAcked≤ LastByteSent and LastByteSent ≤ LastByteWritten

* The bytes to the left of LastByteAcked are not kept as it had been acknowledged.
* The receive buffer maintains three variables namely LastByteRead, NextByteExpected, and LastByteRcvd. The relation between them is

LastByteRead < NextByteExpected and NextByteExpected ≤LastByteRcvd + 1

* If data are received in order, NextByteExpected is the next byte after LastByteRcvd
* Bytes to the left of LastByteRead is not buffered as it has been read by the application

**Flow Control**

* The capacity of *send* and *receive* buffer is MaxSendBuffer and MaxRcvBuffer respectively.
* The sending TCP prevents overflowing of its buffer by maintaining

LastByteWritten- LastByteAcked ≤ MaxSendBuffer

* The receiving TCP avoids overflowing its receive buffer by maintaining

LastByteRcvd− LastByteRead ≤ MaxRcvBuffer

* The receiver throttles the sender by advertising a window that is no larger than the amount of *free* space that it can buffer as

AdvertisedWindow = MaxRcvBuffer−*((*NextByteExpected− 1*)* −LastByteRead*)*

* When data arrives, the receiver acknowledges it as long as preceding bytes have arrived.
* LastByteRcvd moves to its right (incremented), and the advertised window shrinks
* The advertised window expands when the data is read by the application
* If data is read as fast as it arrives then AdvertisedWindow = MaxRcvBuffer
* If it is read slow, it eventually leads to a AdvertisedWindow of size 0.
* The sending TCP adheres to the advertised window by computing *effective* window that limits how much data it should send as

EffectiveWindow = AdvertisedWindow−*(*LastByteSent−LastByteAcked*)*

* When a acknowledgement arrives for *x* bytes, LastByteAcked is incremented by *x* and

the buffer space is freed accordingly

**Fast Sender vs. Slow Receiver**

* A slow receiver prevents being swamped with data from a fast receiver by using AdvertisedWindow field
* Initially the fast sender transmits at a higher rate.
* The receiver's buffer gets filled up. Hence, AdvertisedWindow shrinks, eventually to 0.
* When the receiver advertises window of size 0, sender cannot transmit any further data. Therefore, the TCP at the sender blocks the sending process.
* When the receiving process reads some data, those bytes are acknowledged. Thus the AdvertisedWindow expands.
* The LastByteAcked is incremented and buffer space is freed to that extent
* The sending process becomes unblocked and is allowed to fill up the free space.

**Checking AdvertisedWindow status**

* TCP always sends a segment in response that contains the latest values for the Acknowledge and AdvertisedWindow fields, even if these values have not changed.
* Thus the sender can come to know the status of AdvertisedWindow even after the receiver advertises a window of size 0.

**AdvertisedWindow**

* The TCP's AdvertisedWindow field is 16 bits long, half the size of SequenceNum
* The length of 16-bits ensures that it does not wrap around
* The length of AdvertisedWindow is designed such that it allows the sender to keep the pipe full.
* The 16-bit length also accounts for product of delay × bandwidth.

**TRIGGERING TRANSMISSION**

TCP has three mechanisms to trigger the transmission of a segment.

* 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. That is, MSS is set to the MTU (Maximum Transmission Unit) of the directly connected network, minus the size of the TCP and IP headers.
* The second thing that 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.
* 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.

***What is silly window syndrome? When should TCP transmit a segment?***

* When an ACK arrives, the window enlarges for transmission.
* Even if window size is less than one MSS, TCP decides to go ahead and transmit a half-full segment.
* The strategy of aggressively taking advantage of any available window leads to a situation now known as the silly window syndrome.
* If the sender aggressively fills, then any small segments introduced into the system remains in the system indefinitely as it does not combine with adjacent segments to create larger ones as shown.



**Nagle’s Algorithm**

* Nagle's suggests a solution as to what the sending TCP should do when there is data to send and window size is less than one MSS. The algorithm is listed below:

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

* It’s always OK to send a full segment if the window allows.
* It’s also OK to immediately send a small amount of data if there are currently no segments in transit, but if there is anything in flight, the sender must wait for an AC before transmitting the next segment.

**ADAPTIVE RETRANSMISSION**

*What is adaptive retransmission? Explain the algorithms used?*

* TCP guarantees reliability through retransmission.
* Retransmission due to timeout before ACK.
* Timeout is a function of RTT(Round Trip Time)
* RTT is highly variable between any two hosts on the internet.
* Appropriate timeout is chosen using adaptive retransmission.

**Original Algorithm**

* TCP estimates SampleRTT by computing the duration between sending of a packet and arrival of its ACK.
* TCP then computes EstimatedRTT as a weighted average between the previous and current estimate as

EstimatedRTT = *α* ×EstimatedRTT+*(*1−*α)*×SampleRTT where *α* is the smoothening factor and its value is in the range 0.8–0.9

* Timeout is twice the EstimatedRTT

TimeOut = 2 × EstimatedRTT

**Karn/Partridge Algorithm**

* The flaw discovered in original algorithm after years of use is
  + whether ACK should be associated with the original or retransmission segment
  + If ACK is associated with original one, then SampleRTT becomes too large
  + If ACK is associated with retransmission, then SampleRTT becomes too small



* Karn/Partridge proposed a solution to the above by making changes to the timeout mechanism.
* Each time TCP retransmits, it sets the next timeout to be twice the last timeout.
  + Loss of segments is mostly due to congestion and hence TCP source does not react aggressively to a timeout

**Jacobson/Karels Algorithm**

* The main problem with original algorithm is that variance of the sample RTTs is not

taken into account.

* + if variation among samples is small, then EstimatedRTT can be trusted
  + otherwise timeout should not be tightly coupled with the EstimatedRTT
* In this new approach, the sender measures a new SampleRTT as before.
* The Deviation amongst RTTs is computed as follows:

Difference = SampleRTT−EstimatedRTT

EstimatedRTT = EstimatedRTT+*(δ* ×Difference*)*

Deviation = Deviation+*δ(*|Difference| − Deviation

where *δ* is a fraction between 0 and 1.

* TCP now computes TimeOut as a function of both EstimatedRTT and Deviation as listed:

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

where *μ* is typically set to 1 and *φ* is set to 4.

* When variance is small, difference between TimeOut and EstimatedRTT is negligible.
* When variance is larger, Deviation plays a greater role in deciding TimeOut.

**REMOTE PROCEDURE CALL**

**TCP CONGESTION CONTROL**

**Explain TCP congestion control techniques in detail.**

* In TCP congestion control, each source has to determine the available capacity in the network, so that it can send packets without loss.
* By using ACKs to pace transmission of packets, TCP is said to be *self-clocking*.
* TCP maintains a state variable CongestionWindow for each connection. Therefore

MaxWindow = MIN(CongestionWindow, AdvertisedWindow)

EffectiveWindow = MaxWindow−*(*LastByteSent− LastByteAcked*)*

* Thus, a TCP source is allowed to send no faster than *network* or *destination* host
* The problem is that available bandwidth changes over time.
* Slow The three congestion control mechanism are:

1. Additive Increase/Multiplicative Decrease
2. Slow Start
3. Fast Retransmit and Fast Recovery

**ADDITIVE INCREASE/MULTIPLICATIVE DECREASE (AIMD)**

* TCP source sets the CongestionWindow based on the level of congestion it perceives to exist in the network.
* The additive increase/multiplicative decrease (AIMD) mechanism works as follows:
  + The source increases CongestionWindow when level of congestion goes down and decreases CongestionWindow when level of congestion goes up.
* TCP interprets timeouts as a sign of congestion and reduces the rate at which it is transmitting.
  + Each time a timeout occurs, the source sets CongestionWindow to half of its previous value. This is known as *multiplicative decrease*.
  + For example, if CongestionWindow is set to 16 packets, after a packet loss, it is set to 8.
  + The CongestionWindow is not allowed to fall below one packet size or MSS irrespective of the level of congestion.
* Every time, the source successfully sends one packet, CongestionWindow is increased by a fraction (*additive increase*).
  + An ACK acknowledges receipt of MSS bytes, the increment is computed as

Increment = MSS × (MSS/CongestionWindow)

CongestionWindow += Increment



* This pattern of continually increasing and decreasing the congestion window continues throughout the lifetime of the connection



* When the current value of CongestionWindow as a function of time, it results as a saw-tooth pattern.
* AIMD decreases its CongestionWindow aggressively but increases conservatively
  + Having small CongestionWindow only results in less probability of packets being dropped.
  + Thus congestion control mechanism becomes stable.
* Since timeout is an indication of congestion that triggers multiplicative decrease, TCP needs the most accurate timeout mechanism.
* AIMD is appropriate only when source is operating close to network capacity.

**SLOW START**

* Slow start increases the congestion window exponentially, rather than linearly. It is usually used from cold start.
* The source starts by setting CongestionWindow to one packet.
  + When ACK arrives, TCP adds 1 to CongestionWindow and sends two packets.
  + Upon receiving two ACKs, TCP increments CongestionWindow by 2 and sends four packets.
  + Thus TCP doubles the number of packets every RTT as shown



* Slow start provides exponential growth and is designed to avoid bursty nature of TCP.
* Initially TCP has no idea about congestion, henceforth it increases CongestionWindow rapidly until there is a packet loss.
* When a packet is lost:
  + TCP immediately decreases CongestionWindow by half (*multiplicative decrease*).
  + It stores the current value of CongestionWindow as CongestionThreshold and resets to CongestionWindow one packet
  + The CongestionWindow is incremented one packet for each ACK arrived until it reaches CongestionThreshold and thereafter one packet per RTT.
* In initial stages, TCP loses more packets because it attempts to learn the available bandwidth quickly through exponential increase



* In example, initial slow start causes increase in CongestionWindow up to 34KB.
* Congestion occurs at 2secs and loss of packets results.
  + CongestionThreshold is set to 17KB and CongestionWindow to 1 packet.
  + Thereafter additive increase is followed

**FAST RETRANSMIT AND FAST RECOVERY**

* Fast retransmit is a heuristic that triggers the retransmission of a dropped packet sooner than the regular timeout mechanism. It does not replace regular timeouts.
* When a packet arrives out of order, the receiving TCP resends the same acknowledgment (*duplicate ACK*) it sent the last time.
* The sending TCP waits for three duplicate ACK, to confirm that the packet is lost before retransmitting the lost packet.
* This is known as *fast retransmit* and it signals congestion.
* Instead of setting CongestionWindow to one packet, this method uses the ACKs that are still in pipe to clock the sending of packets. This is called *fast recovery*
* The fast recovery mechanism removes slow start phase and follows additive increase.
* The fast retransmit/recovery results increase in throughput by 20%.



* In example the third packet gets lost. The sender on receiving three duplicate ACKs (ACK 2) retransmits the third packet.



* In graph shown, fast recovery avoids slow start from 3.8 to 4 sec. Therefore congestion window is reduced by half from 22 KB to 11 KB.
* Slow start is only used at the beginning of a connection and after regular timeout.
* At other times, the congestion window follows a pure additive increase/multiplicative decrease pattern
* TCP's fast retransmit can detect up to three dropped packets per window.

**TCP CONGESTION AVOIDANCE**

Explain in detail about TCP congestion avoidance algorithms.

* Congestion avoidance mechanisms prevent congestion before it actually occurs.
* When congestion is likely to occur, TCP decreases load on the network.
* TCP creates loss of packets in order to determine bandwidth of the connection
* The three congestion-avoidance mechanisms are:

1. DEC bit

2. Random Early Detection (RED)

3. Source-based congestion avoidance

**DEC bit**

* Was developed for use on Digital Network Architecture
* In DEC bit, each router monitors the load it is experiencing and explicitly notifies the end node when congestion is about to occur by setting a binary congestion bit called DECbit in packets that flow through it.
* The destination host copies the DECbit onto the ACK and sends back to the source.
* Eventually the source reduces its transmission rate and congestion is avoided.

**Algorithm**

* A single congestion bit is added to the packet header.
* A router sets this bit in a packet if its average queue length is ≥ 1.
* The average queue length is measured over a time interval that spans the last busy +

last idle cycle + current busy cycle.

* Router calculates average queue length by dividing the curve area by time interval



* The source computes how many ACK has DEC bit set for the previous window packets it has sent.

1. If it is less than 50% then source increases its congestion window by 1 packet.

2. Otherwise, source decrease the congestion window by 87.5%.

**RANDOM EARLY DETECTION (RED)**

* Proposed by Floyd and Jackson
* In RED, router implicitly notifies the source that congestion is likely to occur by dropping one of its packets.
* The source is notified by timeout or duplicate ACK.
* The router drops a few packets earlier before it runs out of space, so that it need not drop more packets later.
* Each incoming packet is dropped with a probability known as *drop probability* when the queue length exceeds *drop level*.

**Algorithm**

* RED computes average queue length using a weighted running average as follows

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

* where 0 < Weight < 1 and SampleLen is length of the queue when a sample measurement is made.
* The weighted running average detects long-lived congestion.
* RED has two queue length thresholds MinThreshold and MaxThreshold. When a packet arrives at the gateway, RED compares the current AvgLen with these