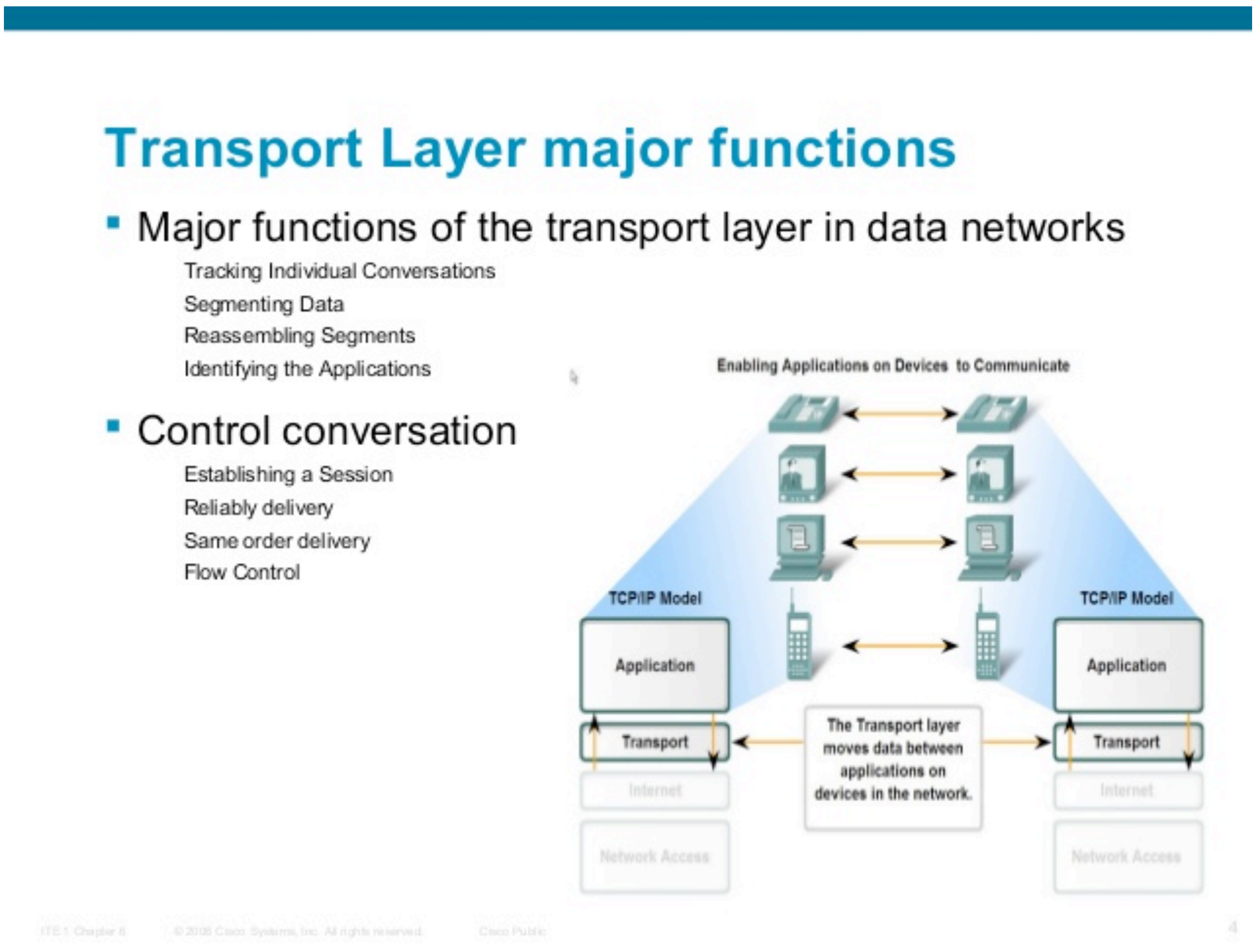
UNIT IV TRANSPORT LAYER

Overview of Transport layer - UDP - Reliable byte stream (TCP) - Connection management - Flow control - Retransmission – TCP Congestion control - Congestion avoidance (DECbit, RED) – QoS – Application requirements

In computer networking, a **transport layer** provides end-to-end or host-to-host communication services for applications within a layered architecture of network components and protocols.[1] The transport layer provides services such as connection-oriented data stream support, reliability, flow control, and multiplexing.

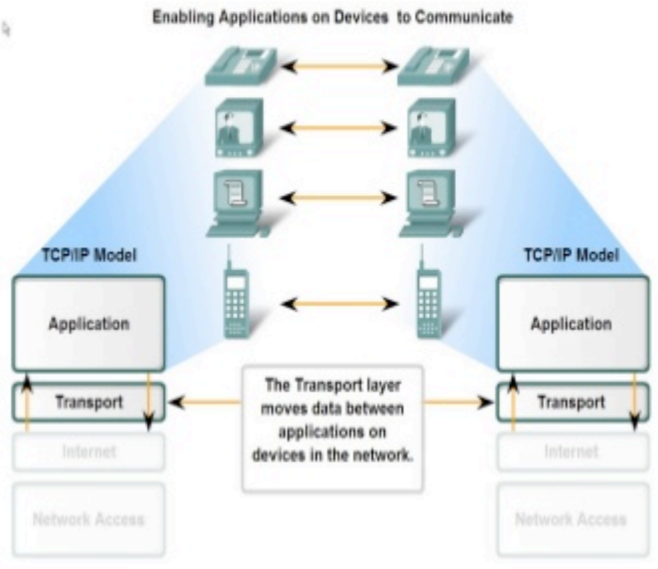
Transport layer implementations are contained in both the TCP/IP model which is the foundation of the Internet, and the Open Systems Interconnection (OSI) model of general networking, however, the definitions of details of the transport layer are different in these models. In the Open Systems Interconnection model the transport layer is most often referred to as **Layer 4** or**L4**.

The best-known transport protocol is the Transmission Control Protocol (TCP). It lent its name to the title of the entire Internet Protocol Suite, *TCP/IP*. It is used for connection-oriented transmissions, whereas the connectionless User Datagram Protocol (UDP) is used for simpler messaging transmissions. TCP is the more complex protocol, due to its stateful design incorporating reliable transmission and data stream services. Other prominent protocols in this group are the Datagram Congestion Control Protocol (DCCP) and the Stream Control Transmission Protocol (SCTP).



Transport Layer major functions

- Major functions of the transport layer in data networks
  - Tracking Individual Conversations
  - Segmenting Data
  - Reassembling Segments
  - Identifying the Applications
- Control conversation
  - Establishing a Session
  - Reliably delivery
  - Same order delivery
  - Flow Control


Enabling Applications on Devices to Communicate
TCP/IP Model
Application
Transport
Internet
Network Access
TCP/IP Model
Application
Transport
Internet
Network Access
The Transport layer moves data between applications on devices in the network.

Transport layer services are conveyed to an application via a programming interface to the transport layer protocols. The services may include the following features:

- Connection-oriented communication: It is normally easier for an application to interpret a connection as a data stream rather than having to deal with the underlying connection-less models, such as the datagram model of the User Datagram Protocol (UDP) and of the Internet Protocol (IP).
- Same order delivery: The network layer doesn't generally guarantee that packets of data will arrive in the same order that they were sent, but often this is a desirable feature. This is usually done through the use of segment numbering, with the receiver passing them to the application in order. This can cause head-of-line blocking.
- Reliability: Packets may be lost during transport due to network congestion and errors. By means of an error detection code, such as a checksum, the transport protocol may check

that the data is not corrupted, and verify correct receipt by sending an ACK or NACK message to the sender. Automatic repeat request schemes may be used to retransmit lost or corrupted data.

- Flow control: The rate of data transmission between two nodes must sometimes be managed to prevent a fast sender from transmitting more data than can be supported by the receiving data buffer, causing a buffer overrun. This can also be used to improve efficiency by reducing buffer underrun.
- Congestion avoidance: Congestion control can control traffic entry into a telecommunications network, so as to avoid congestive collapse by attempting to avoid oversubscription of any of the processing or link capabilities of the intermediate nodes and networks and taking resource reducing steps, such as reducing the rate of sendingpackets. For example, automatic repeat requests may keep the network in a congested state; this situation can be avoided by adding congestion avoidance to the flow control, including slow-start. This keeps the bandwidth consumption at a low level in the beginning of the transmission, or after packet retransmission.
- Multiplexing: Ports can provide multiple endpoints on a single node. For example, the name on a postal address is a kind of multiplexing, and distinguishes between different recipients of the same location. Computer applications will each listen for information on their own ports, which enables the use of more than one network service at the same time. It is part of the transport layer in the TCP/IP model, but of the session layer in the OSI model.

Comparison of UDP and TCP

Transmission Control Protocol is a connection-oriented protocol, which means that it requires handshaking to set up end-to-end communications. Once a connection is set up, user data may be sent bi-directionally over the connection.

- *Reliable* – TCP manages message acknowledgment, retransmission and timeout. Multiple attempts to deliver the message are made. If it gets lost along the way, the server will re-request the lost part. In TCP, there's either no missing data, or, in case of multiple timeouts, the connection is dropped.
- *Ordered* – If two messages are sent over a connection in sequence, the first message will reach the receiving application first. When data segments arrive in the wrong order, TCP buffers delay the out-of-order data until all data can be properly re-ordered and delivered to the application.
- *Heavyweight* – TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.

- *Streaming* – Data is read as a byte stream, no distinguishing indications are transmitted to signal message (segment) boundaries.

User Datagram Protocol is a simpler message-based connectionless protocol. Connectionless protocols do not set up a dedicated end-to-end connection. Communication is achieved by transmitting information in one direction from source to destination without verifying the readiness or state of the receiver.

- *Unreliable* – When a UDP message is sent, it cannot be known if it will reach its destination; it could get lost along the way. There is no concept of acknowledgment, retransmission, or timeout.
- *Not ordered* – If two messages are sent to the same recipient, the order in which they arrive cannot be predicted.
- *Lightweight* – There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP.
- *Datagrams* – Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, meaning a read operation at the receiver socket will yield an entire message as it was originally sent.
- *No congestion control* – UDP itself does not avoid congestion, unless they implement congestion control measures at the application level.
- *Broadcasts* - being connectionless, UDP can broadcast - sent packets can be addressed to be receivable by all devices on the subnet.

UDP:

The User Datagram Protocol (UDP) is simplest Transport Layer communication protocol available of the TCP/IP protocol suite. It involves minimum amount of communication mechanism. UDP is said to be an unreliable transport protocol but it uses IP services which provides best effort delivery mechanism.

In UDP, the receiver does not generate an acknowledgement of packet received and in turn, the sender does not wait for any acknowledgement of packet sent. This shortcoming makes this protocol unreliable as well as easier on processing.

Requirement of UDP

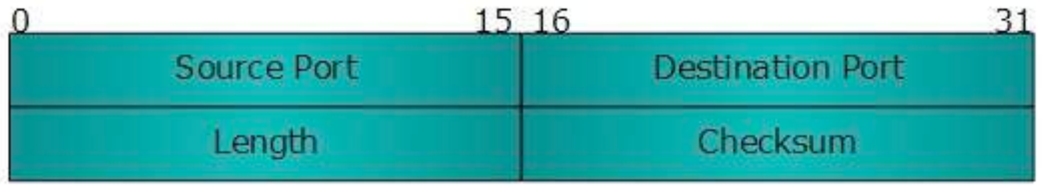
A question may arise, why do we need an unreliable protocol to transport the data? We deploy UDP where the acknowledgement packets share significant amount of bandwidth along with the actual data. For example, in case of video streaming, thousands of packets are forwarded towards its users. Acknowledging all the packets is troublesome and may contain huge amount of bandwidth wastage. The best delivery mechanism of underlying IP protocol ensures best efforts to deliver its packets, but even if some packets in video streaming get lost, the impact is not calamitous and can be ignored easily. Loss of few packets in video and voice traffic sometimes goes unnoticed.

Features

- UDP is used when acknowledgement of data does not hold any significance.
- UDP is good protocol for data flowing in one direction.
- UDP is simple and suitable for query based communications.
- UDP is not connection oriented.
- UDP does not provide congestion control mechanism.
- UDP does not guarantee ordered delivery of data.
- UDP is stateless.
- UDP is suitable protocol for streaming applications such as VoIP, multimedia streaming.

UDP Header

UDP header is as simple as its function.



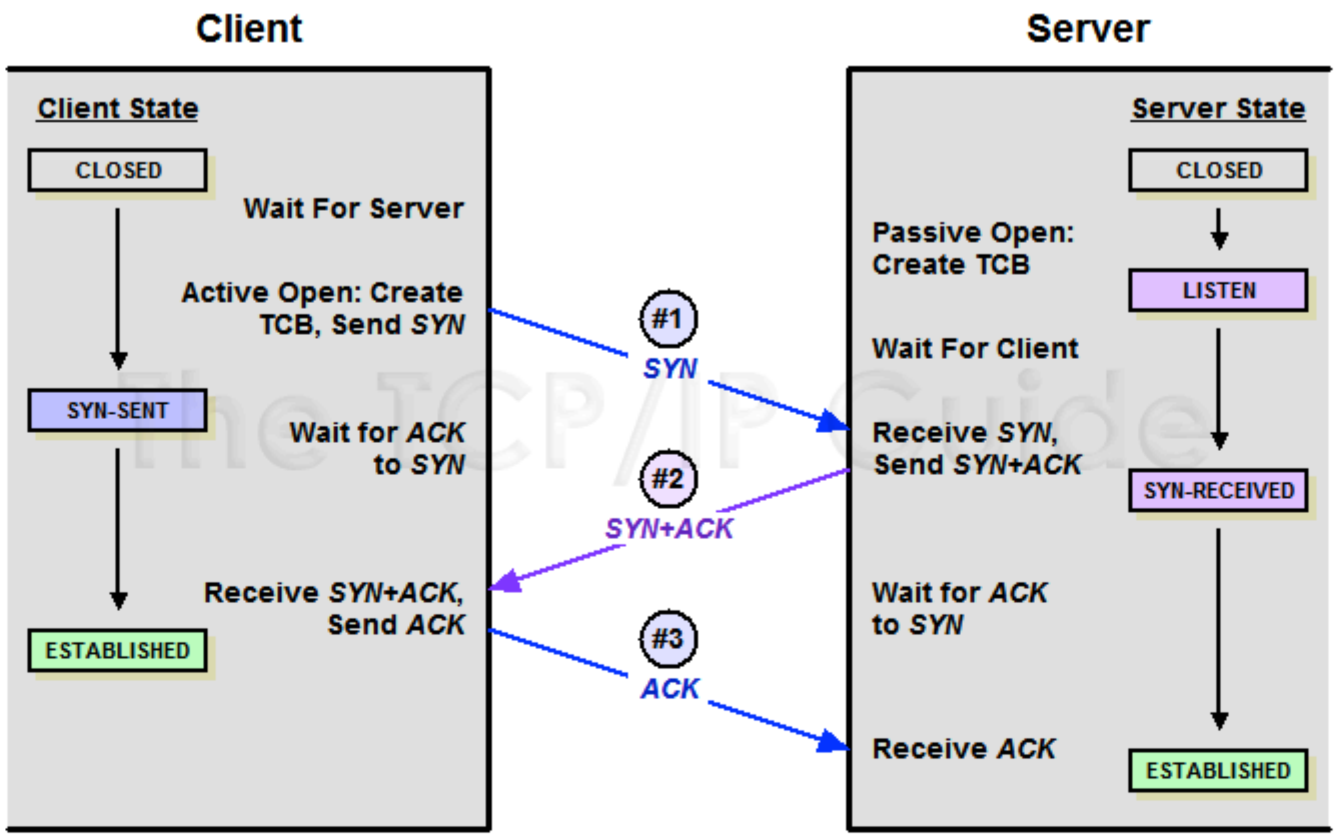
UDP header contains four main parameters:

- **Source Port** - This 16 bits information is used to identify the source port of the packet.
- **Destination Port** - This 16 bits information, is used identify application level service on destination machine.
- **Length** - Length field specifies the entire length of UDP packet (including header). It is 16-bits field and minimum value is 8-byte, i.e. the size of UDP header itself.

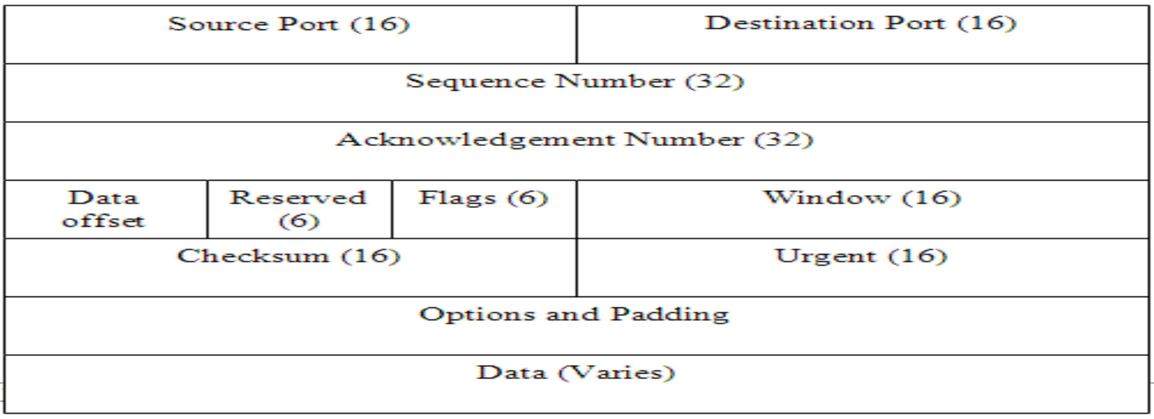
- **Checksum** - This field stores the checksum value generated by the sender before sending. IPv4 has this field as optional so when checksum field does not contain any value it is made 0 and all its bits are set to zero.

TCP:

TCP Connection Establishment Process: The "Three-Way Handshake" :



To establish a connection, each device must send a *SYN* and receive an *ACK* for it from the other device. Thus, conceptually, we need to have four control messages pass between the devices. However, it's inefficient to send a *SYN* and an *ACK* in separate messages when one could communicate both simultaneously. Thus, in the normal sequence of events in connection establishment, one of the *SYNs* and one of the *ACKs* is sent together by setting both of the relevant bits (a message sometimes called a *SYN+ACK*). This makes a total of three messages, and for this reason the connection procedure is called a *three-way handshake*.



Source Port (16)			Destination Port (16)
Sequence Number (32)			
Acknowledgement Number (32)			
Data offset	Reserved (6)	Flags (6)	Window (16)
Checksum (16)			Urgent (16)
Options and Padding			
Data (Varies)			

Each TCP segment contains the header.

- The **SrcPort**and **DstPort** fields identify the source and destination ports, resp. like UDP. These two fields, plus the source and destination IP addresses, combine to uniquely identify each TCP connection. TCP's demux key is given by the 4-tuple - SrcPort, SrcIPAddr, DstPort, DstIPAddr
- The Acknowledgment, SequenceNum, and AdvertisedWindow fields are all involved in TCP's sliding window algorithm.
- TCP is a byte-oriented protocol. So each byte of data has a **sequence number**; SequenceNum field contains the sequence number for the first byte of data carried in that segment.
- The **Acknowledgment** and **Advertised Window** fields carry information about the flow of data going in the other direction.
- The 6-bit Flags field is used to relay control information between TCP peers.
- The flags include SYN, FIN, RESET, PUSH, URG, and ACK. The SYN and FIN flags are used when establishing and terminating a TCP connection.

Code Bits:

URG =1 : Activates URGENT PTR field.
ACK =1 : Activates the acknowledgement field.
PSH =1 : pushes the data even before buffer fills.
RST = 1: Reset the connection.
SYN =1: Synchronize the sequence number.
FIN = 1 : Sender has reached end of its data.

The **ACK** flag is set any time the Acknowledgment field is valid, implying that the receiver should pay attention to it.

The **URG** flag 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 **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 of this fact.

The **RESET** flag signifies that the receiver has become confused—for example, because it received a segment it did not expect to receive—and so wants to abort the connection.

The **Checksum** covers the TCP segment - The TCP header and the TCP data. This is a mandatory field that must be calculated by the sender and then verified by the receiver.
A**HdrLen** field gives the length of the header in 32-bit words. This field is also known as the Offset field, since it measures the offset from the start of the packet to the start of the data.

ii) Silly window syndrome:

- If the sender or the receiver application program processes slowly and can send only 1 byte of data at a time, then the overhead is high. This is because to send one byte of data, 20 bytes of TCP header and 20 bytes of IP header are sent. This is called as silly window syndrome.
- The silly window syndrome occurs when either the sender transmits a small segment or the receiver opens the window to a small amount only. Both involve inefficient use of BW.
- If neither of these two happens, then small sized segments are never introduced into the stream. But if sending appln specifically goes for invoking PUSH, then small sized segments can be introduced.
- Receiver could be stopped from advertising small window and asked to wait until a space equal to *Maximum Segment Size(*MSS) is available.
- This is just a partial soln, as the receiver, in no way knows how long to delay acknowledgements.
- The answer is to introduce a timer and to transmit when the timer expires. Nagle introduced an elegant self-clocking solution.
- The idea is that as long as TCP has any data in flight, the sender will eventually receive an ACK. This ACK can be treated like a timer firing, triggering the transmission of more data.

Nagle's algorithm provides a simple, unified rule for deciding when to transmit:
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
```

Nagle's algorithm can also be turned off by setting the TCP NODELAY option
Suppose receiver buffer is full. It advertises window is zero. Effective window becomes a negative value. Sender will not transmit any data to receiver, finally sender buffer will fill. This will stop its own application program from writing in buffer.

As soon as receiver process starts to read again, its advertiser window will become > 0 that allows sender to transmit data out of its buffer. This allow sender application program to restart its writing. The sender knows this from advertised a window of size. Then the interactive application reads one character from the TCP stream. This action makes the receiving TCP happy, so it sends a window update to the sender saying that it is all right to send 1 byte. The sender obliges and sends 1 byte.

The buffer is now full, so the receiver acknowledges the 1 –byte segments but sets the window to 0. This behavior can go on forever. In applications each byte is sent as TCP segment:
1byte data + 20 byte IP header + 20 byte TCP header=41 byte => known as 'TINYGRAM' overhead is more. (- for one byte data over head is 40 byte).

TCP Congestion control:

TCP maintains a new state variable for each connection, called **CongestionWindow**, which is used by the source to limit how much data it is allowed to have in transit at a given time.
TCP is modified such that the maximum number of bytes of unacknowledged data allowed is now the minimum of the congestion window and the advertised window.
Thus, TCP's effective window is as follows:
MaxWindow = MIN (CongestionWindow, AdvertisedWindow)
EffectiveWindow = MaxWindow − (LastByteSent − LastByteAcked).
Thus, a TCP source should not send faster than what the n/w or destination host can accommodate.

1) Additive Increase/Multiplicative Decrease (AIMD):

TCP source sets the congestion window based on the level of congestion it perceives to exist in the n/w. This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down. Taken together, the mechanism is commonly called **AIMD.**
If packets are not delivered, a timeout results, congestion is present in them. TCP interprets timeouts as a sign of congestion and reduces the rate at which it is transmitting.
Whenever timeout occurs, the source sets congestion window to half of its previous value each time – **multiplicative decrease**. Suppose now congestion window is 16 packets. If a loss is detected, congestion window is set to 8. Additional losses cause congestion window to be 4, then to 2 finally to 1.
Now how congestion window takes the advantage of newly available capacity in the network. Every time the source successfully sends a congestion window, it adds 1 packet to the congestion window.-**additive increase.** This pattern of continually increasing and decreasing congestion window continues throughout life time of the connection. If we draw congestion window as a function of time, the curve is saw tooth form.
TCP interprets timeouts as a sign of congestion and reduces the rate at which it is transmitting. Specifically, each time a timeout occurs, the source sets CongestionWindow to half of its previous value. This halving of the CongestionWindow for each timeout corresponds to the "multiplicative decrease" part of AIMD.
Suppose the CongestionWindow is currently set to 16 packets. If a loss is detected, CongestionWindow is set to 8. An additional loss cause CongestionWindow to be reduced to 4, then 2, and finally to 1 packet.CongestionWindow is not allowed to fall below the size of a single packet, or the *Maximum Segment Size (MSS).*

Every time the source successfully sends a Congestion Window's worth of packets and gets acknowledgement for the same, it adds the equivalent of one packet to CongestionWindow.
This is the ***additive increase part*** of AIMD. In practice, TCP does not wait for an entire window's worth of ACKs to add 1 packet's worth to congestion window, but instead increments congestion window as given below:

Increment = MSS × (MSS/CongestionWindow)
CongestionWindow + = Increment

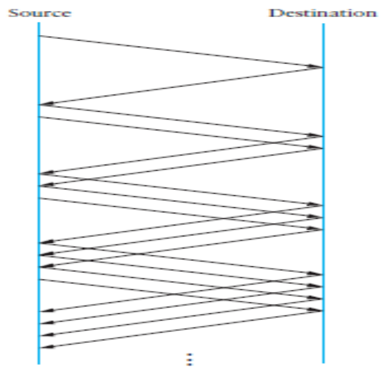


Figure 6.8 Packets in transit during additive increase, with one packet being added each RTT.

i.e., rather than incrementing congestion window by an entire MSS bytes for each RTT, we increment it by a fraction of MSS every time an ACK is received. Assuming that each ACK acknowledges the receipt of MSS bytes, then that fraction is MSS / Congestion Window. The important concept of AIMD is that the source is willing to reduce its congestion window at a much faster rate than it is willing to increase its congestion window.
The additive increase mechanism just described is the right approach to use when the source is operating close to the available capacity of the network. For a source, which has just begun from scratch, it is not suitable.

1. Slow start:

It is a congestion control technique. The additive increase mechanism is the right approach to use when the source is operating close to the available capacity of the network, but it takes too long to ramp up a connection when it is starting from scratch. TCP therefore 472 6 Congestion Control and Resource Allocation & Source Destination provides a second mechanism, ironically called *slow start*, that is used to increase the congestion window rapidly from a cold start. Slow start effectively increases the congestion window exponentially, rather than linearly. Specifically, the source starts out by setting CongestionWindow to one packet. When the ACK for this packet arrives, TCP adds 1 to CongestionWindow and then sends two packets. Upon receiving the corresponding two ACKs, TCP increments CongestionWindow by 2—one for each ACK—and next sends four packets. The end result is that TCP effectively doubles the number of packets it has in transit every RTT.
The source starts out by setting congestion window to one packet. When ACK for this packet arrives, TCP adds 1 to congestion window and then sends 2 packets. Upon receiving 2 ACK, TCP increments congestion window to be 4.
Consider the case when timeout occurs. By that time source will not transmit any more packets. After sometime, source will receive a single cumulative ACK that re opens a entire advertised window.
Now source uses slow start rather than using effective windows (i.e.) window size is 1. It uses slow start (i.e.) multiplicative increase until window size is half value of congestion window size

because of what loss occurs just now. This target congestion window size is also known as threshold value. Slow start is used to rapidly increase the sending rate up to the value. Then additional increase is used beyond d this point.

Two different situations in which slow start runs:

- The first is at the very beginning of a connection, at which time the source has no idea how many packets it is going to be able to have in transit at a given time.
- The second situation, it is used is when the connection at source goes dead while waiting for a timeout to occur. This might happen when a packet is lost, source reaches a point where it has sent its entire data as specified by advertised window and it blocks for ACK to arrive, which will not arrive. Eventually a time out happens; source receives a single cumulative ACK that reopens the entire advertised window, making the source to begin slow start to restart flow of data.

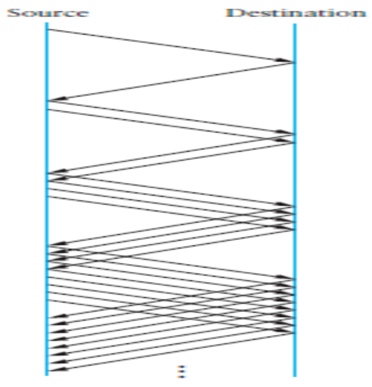


Figure 6.10 Packets in transit during slow start.

Although the source is using slow start again, it now knows more information than it did at the beginning of a connection. The source now knows the congestion window value (target congestion window value) that existed before packet loss. Due to packet loss, multiplicative decrease happened and the congestion window has now reduced to one half of what it was before. The new reduced value of congestion window is stored in a temporary variable Congestion Threshold. Source starts slow start phase, increasing exponentially till it reaches the Congestion Threshold value. After reaching, it begins Additive Increase phase begins where the congestion window size is reset to 1 packet. Now the congestion window grows linearly till it reaches the target congestion window value or till next time out.

1) Fast retransmit and fast recovery

The mechanisms described so far were part of the original proposal to add congestion control to TCP. A new mechanism called **fast re-transmit** was added to TCP. Fast retransmit is a heuristic that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism. Every time a data packet arrives at the receiving side, the receiver responds with an acknowledgment, even if this sequence number has already been acknowledged.

Thus, when a packet arrives out of order, TCP resends the same acknowledgment it sent the last time. This second transmission of the same acknowledgment is called a **duplicateACK**. Whenthe sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that an earlier packet might have been lost. The sender waits until it sees some number of duplicate ACKs and then retransmits the missing packet. TCP waits until it has seen three duplicate ACKs before retransmitting the packet.

When the fast retransmit mechanism signals congestion, rather than drop the congestion window to 1 packet and start slow start, it is possible to use the ACKs that are still in the pipe to clock the sending of packets. This mechanism, which is called **fast recovery**, effectively removes the slow start phase that happens between when fast retransmit detects a lost packet and additive increase begins.

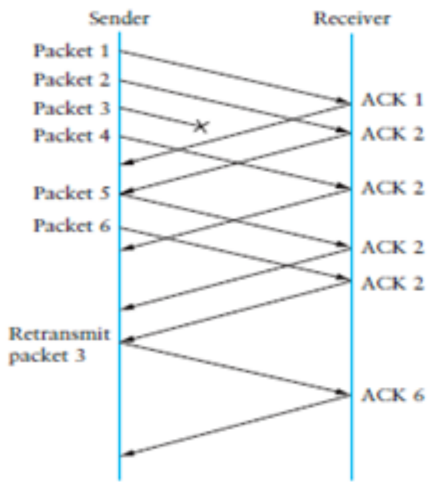


Figure 6.12 Fast retransmit based on duplicate ACKs.

TCP Congestion Avoidance methods:

It is a congestion avoidance technique. Each router monitors the load and explicitly notifies the end nodes when congestion is about to occur.

- If avg. queue length is >= 1, then notify.
- Notification is implemented by setting a binary congestion bit in the packets that flow thru the router.

The destination host then copies the bit into ACK, it sends to source. The source records how many of its packets resulted in some router setting the congestion bit. Source maintains a congestion window and checks what fraction of last window's packets resulted in the bit being set.Source adjusts its sending rate to avoid congestion

- If less than 50% packets/ACKs have the bit set, increase congestion window by 1
- If 50% or more packets have congestion bit set, decrease congestion window to 0.875 of previous value.

b) **Random early detection (RED)**

This method***implicitly***notifies the source of congestion by dropping one of the packets.

The gateway drops packets earlier (before its queue fills up), notifying sender to decrease its congestion window or a router drops few packets before its buffer space gets exhausted, so as to cause the source to slow down, so that it need not drop lots of packets later on.

Rather than wait for the queue to become completely full and then be forced to drop each arriving packet, we could decide to drop each arriving packet with some **drop**

probability whenever the queue length exceeds some *drop level*. This idea is called **early random drop**.

RED alg defines how to monitor the queue length and when to drop a pkt.

RED computes an avg. queue length using a weighted running average.

AvgLen= (1-weight) AvgLen + weight X SampleLen

Where 0 < weight < 1 and SampleLen is length of queue when a sample measurement is made.

Two queue length thresholds, MinThreshold&MaxThreshold which trigger certain activity are defined. When a pkt arrives at gateway, RED compares the current AvgLen with these 2 thresholds as below:

i) ifAvgLen<= MinThreshold, queue the pkt.

ii) ifMinThreshold<AvgLen<MaxThreshold, calculate probability P and drop arriving pkt with probability P.

iii) ifMaxThreshold<= AvgLen, drop arriving pkt.

In the following graph, if AvgLen is between min & max thresholds, packets are dropped gently. When maxP is reached, all arriving pkts are dropped.

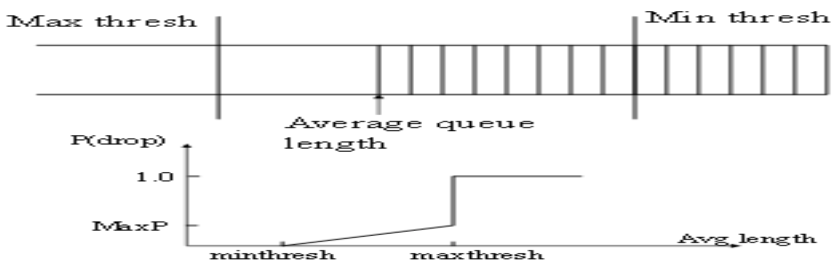
Dropping probability is a function of both AvgLen and how long it has been since the last pkt was dropped.

TempP = MaxP(AvgLen – MinThreshold) / (MaxThreshold – MinThreshold)

Just marking based on TempPcan lead to clustered dropping. (can cause multiple drop in a single connection, which is unfair, similar to tail drop). Dropping should be evenly distributed over timeline.

Better to bias TempP by history of undropped packets ***P = TempP / (1 - count*TempP)***, where count equals the number of packets not dropped since the last drop.

RED operation



c)Source Based Congestion Avoidance

The general idea of these techniques is to watch for some sign from the network that some router's queue is building up and that congestion will happen soon if nothing is done about it.

Increasing RTT:

- We have seen that Timeout is considered as indication of congested state.
- Increment in RTT is an indicator of increased load in bottleneck router, which will soon reach 'congested state'.
- Sending packets in loaded state will probably only increase queue length in the bottleneck router.

TCP Vegas controls its window size based on achieved sending rate .

Step: 1

- Define ***BaseRTT*** for a given flow, to be the minimum of all RTTs when flow is not congested. That is when all Acks returns within a round trip time.

- Initially it is set to the RTT of first packet.
- If connection is not overflowing, then

Expected Rate = CongestionWindow / BaseRTT, where CongestionWindow gives the no. of bytes in transit.

Step: 2

Next we see if this rate is achieved in next RTT period.

Compute current sending rate (Actual Rate) for a distinguished packet and measure the RTT of this packet.

Record sending time of a distinguished packet and then count how many bytes have been sent to network by TCP till the acknowledgement of this distinguished packet returns. Then compute SampleRTT for distinguished packet when itsAck returned and dividing no. of bytes transmitted by SampleRTT. This calculation is done once per RTT.

Step: 3

Compute ***Diff = ExpectedRate - Actual Rate*****,** the RTT should be changed if Diff < 0.

- If Actual Rate >ExpectedRate, BaseRTT changes to NewRTT.

If Actual Rate <ExpectedRate, (Diff >=0) Change Congestion Window as follows:

Define α <β , corresponding to too little or too much extra data in the network

- If Diff < α increase cwnd linearly in next RTT
- If Diff > β decrease cwnd linearly in next RTT
- If α <Diff <β , leave cwnd unchanged.
- Therefore, if actual thruput gets farther away from expected thruput, more congestion is there in n/w, which implies that the sending rate should be reduced. Beta threshold triggers this decrease.
- If actual thruput gets too close to expected thruput, the connection is in danger of not utilizing the available BW. Alpha threshold triggers this increase.
- Overall goal is to keep BW between alpha and beta bytes in the n/w.
- This method decreases congestion window linearly in conflict with the rule that multiplicative decrease is needed to ensure stability. Multiplicative decrease is used only when time out occurs.
- Linear decrease described here is an early decrease in congestion window which hopefully happens before congestion occurs and pkts start being dropped.