UNIT – IV

THE TRANSPORT LAYER

Flow Control

- Flow control defines the amount of data a source can send before receiving an acknowledgement from receiver
- The flow control protocol must make sure that receiver does not get overwhelmed with data (can't let sender send all of its data without worrying about acknowledgements)
- TCP uses a sliding window protocol to accomplish flow control
- For each TCP connection (always duplex), the sending and receiving TCP peer use this window to control the flow.
- TCP's variant of the sliding window algorithm, which 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.

TCP Sliding Window for flow control

Buffer at Sender

- Maintains data sent but not ACKed
- Data written by application but not sent.

Three pointers are maintained at Sender
LastByteAcked, LastByteSent, LastByteWritten.

Sender maintains
LastByteAcked ≤ LastByteSent
LastByteSent ≤ LastByteWritten

Buffer at Receiver

- Data that arrives out of order

- Data that is in correct order but not yet read by application.

Three pointers are maintained at Receiver

LastByteRead, NextByteExpected ,LastByteRcvd

Receiver maintains

LastByteRead <NextByteExpected

NextByteExpected ≤ LastByteRcvd + 1

How is Flow Control done?

- Receiver "advertises" a window size to the sender based on the buffer size allocated for the connection through "Advertised Window" field in the TCP header.
- Sender cannot have more than "Advertised Window" bytes of unacknowledged data.
- Buffers are of finite size - i.e., there is a MaxRcvBuffer and MaxSendBuffer.

Setting the Advertised Window

On the TCP receive sider, clearly,

LastByteRcvd -LastByteRead ≤ MaxRcvBuffer

Thus, it advertises the space left in the buffer i.e.,

Advertised Window = MaxRcvBuffer - (LastByteRcvd -LastByteRead)

Sender Side Response

At the sender side, the TCP sender should ensure that:

LastByteSent - LastByteAcked ≤ Advertised Window.

The "Effective Window" which limits the amount of data that TCP can send :

Effective Window = Advertised Window - (LastByteSent - LastByteAcked)

In order to prevent the overflow of the Sender Side buffer:

LastByteWritten - LastByteAcked ≤ MaxSendBuffer

If application tries to write more, TCP blocks writing into the buffer.

3.8 Adaptive Retransmission

Because TCP guarantees the reliable delivery of data, it retransmits each segment if an ACK is not received in a certain period of time. TCP sets this timeout as a function of the RTT it expects between the two ends of the connection. Choosing an appropriate timeout value is not that easy. To address this problem, TCP uses an adaptive retransmission mechanism.

3.8.1 Original Algorithm – calculation of RTT

- Every time TCP sends a data segment, it records the time.
- When an ACK for that segment arrives, TCP reads the time again and
- Then takes the difference between these two times as a SampleRTT. TCP then computes an EstimatedRTT as a weighted average between the previous estimate and this new sample. That is,

  EstimatedRTT = α × EstimatedRTT + (1 − α) × SampleRTT
- The parameter α is selected to *smooth* the EstimatedRTT.
- The original TCP specification recommended a setting of α between 0.8 and 0.9.
- TCP then uses EstimatedRTT to compute the timeout in a rather conservative way:

  TimeOut = 2 × EstimatedRTT

3.8.2 Karn/Partridge Algorithm

As illustrated in Figure 5.10, if you assume that the ACK is for the original transmission but it was really for the second, then the SampleRTT is too large (a), while if you assume that the ACK is for the second transmission but it was actually for the first, then the SampleRTT is too small (b).

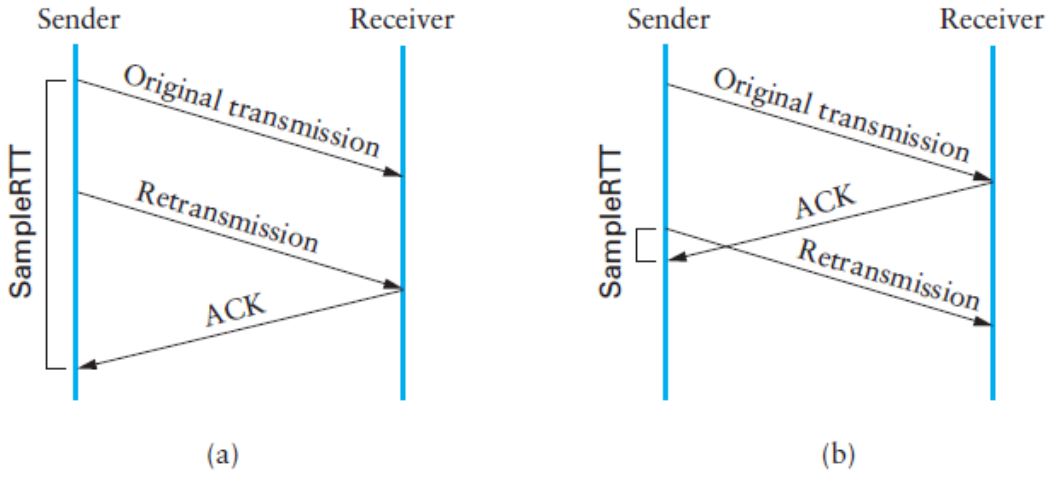


Figure5.10 Associating the ACK with (a) original transmission versus (b) retransmission.

Solution:

- Whenever TCP retransmits a segment, it stops taking samples of the RTT; it only measures SampleRTT for segments that have been sent only once.
- Each time TCP retransmits, it sets the next timeout to be twice the last timeout, rather than basing it on the last EstimatedRTT. That is, Karn and Partridge proposed that TCP use exponential backoff.

3.8.3 Jacobson/Karels Algorithm

- Jacobson and Karels— proposed a more drastic change to TCP to battle congestion.
- The main problem with the original computation is that it does not take the variance of the sample RTTs into account.
- The sender measures a new SampleRTT as before. It then folds this new sample into the timeout calculation as follows:

  Difference = SampleRTT − EstimatedRTT

  EstimatedRTT = EstimatedRTT + (δ × Difference)

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

  whereδ is a fraction between 0 and 1
- TCP then computes the timeout value as a function of both EstimatedRTT and Deviation as follows:

  TimeOut = μ × EstimatedRTT + φ × Deviation

where based on experience, μ is typically set to 1 and φ is set to 4.

3.9 Silly Window Syndrome

- MSS (Maximum Segment Size) is the largest chunk of data TCP will send to the other side
- MSS can be announced in the options field of the TCP.
- Serious problems can arise in the sliding window operationwhen:
  - Sending application creates data slowly, or
  - Receiving applications consumes data slowly(or both)

- Suppose a MSS worth of data is collected and advertised window is MSS/2. What should the sender do ? -- transmit half full segments or wait to send a full MSS when window opens ?Early implementations were aggressive -- transmit MSS/2. Aggressively doing this, would consistently result in small segment sizes -- called the Silly Window Syndrome.

Causes of Silly Window Syndrome

- Poor use of network bandwidth
- Unnecessary computational overhead

Solution:

- Use heuristics at sender to avoid transmitting a small amount of data in each segment
- Use heuristics at receiver to avoid sending small window advisements

Receive-side silly window avoidance

- Monitor receive window size
- Delay advertising an increase until a "significant" increase is possible
- "Significant" = min(half the window, maximum segment size)

Send-Side Silly Window Avoidance

- Avoid sending small segments. TCP must delay sending a segment until it contains a reasonable amount of data.
- How long should TCP wait before transmitting data?. This is given by Nagle's algorithm.

Nagle's Algorithm

If both available data and Window ≥ MSS, send full segment.
Else, if there is unACKed data in flight, buffer new data until ACK returns.
Else, send new data now.

3.10 Queueing discipline

Queueing discipline governs how packets are buffered while waiting to be transmitted. The queuing algorithm can bethought of as allocating both bandwidth (which packets get transmitted) and bufferspace (which packets get discarded). This sectionintroduces three common queuing algorithms—first-in-first-out (FIFO) , Priority queuing and fair queuing (FQ).

3.10.1 FIFO

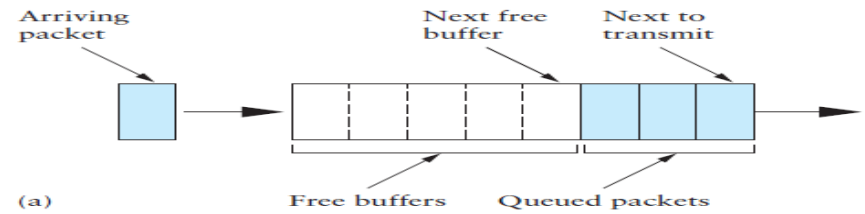
The idea of FIFO queuing, also called first-come-first-served (FCFS) queuing, is simple:The first packet that arrives at a router is the first packet to be transmitted. This isillustrated in Figure 6.5(a), which shows a FIFO with "slots" to hold up to eightpackets. Given that the amount of buffer space at each router is finite, if a packetarrives and the queue (buffer space) is full, then the router discards that packet, asshown in Figure 6.5(b).

Disadvantage : - Packets are dropped (when buffer is full) in the tail end without regard to which flow the packet belongsto or how important the packet is. This is sometimes called *tail drop*, since packetsthat arrive at the tail end of the FIFO are dropped.

Note that tail drop and FIFO are two separable ideas.

FIFO is a *scheduling discipline*—it determines the order in which packets are transmitted.

Tail drop is a *drop policy*—it determines which packets get dropped.



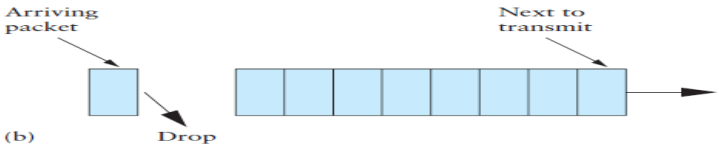


Figure 6.5 (a) FIFO queuing; (b) tail drop at a FIFO queue.

3.10.2 Priority queuing

- A simple variation on basic FIFO queuing is priority queuing.
- The idea is to mark each packet with a priority; the mark could be carried, for example, in the IP Type of Service (TOS) field.
- The routers then implement multiple FIFO queues, one for each priority class.
- The router always transmits packets out of the highest-priority queue if that queue is nonempty before moving on to the next priority queue. Within each priority, packets are still managed in a FIFO manner.

Disadvantage:The high-priority queue canstarve out all the other queues. That is, as long as there is at least one high-prioritypacket in the high-priority queue, lower-priority queues do not get served.

3.10.3 Fair Queuing

- The main problem with FIFO queuing is that it does not discriminate between different traffic sources, or it does not separate packets according to the flow to which they belong.
- Fair queuing (FQ) is an algorithm that has been proposed to address this problem. The idea of FQ is to maintain a separate queue for each flow currently being handled by the router. The router then services these queues in a sort of round-robin.

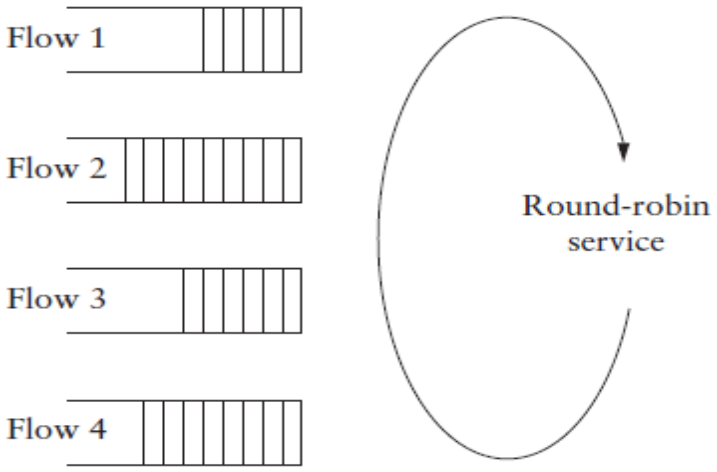


Figure 6.6 Fair queuing at a router.

- The main complication with Fair Queuing is that the packets being processed at a router are not necessarily the same length.

- To truly allocate the bandwidth of the outgoing link in a fair manner, it is necessary to take packet length into consideration.
    - For example, if a router is managing two flows, one with 1000-byte packets and the other with 500-byte packets (perhaps because of fragmentation upstream from this router), then a simple round-robin servicing of packets from each flow's queue will give the first flow two thirds of the link's bandwidth and the second flow only one-third of its bandwidth.
- The FQ mechanism first determines when a given packet would finish being transmitted if it were being sent using bit-by-bit round-robin, and then using this finishing time to sequence the packets for transmission.
- To understand the algorithm for approximating bit-by-bit round robin, consider the behavior of a single flow
    - For this flow, let

      P_i : denote the length of packet i

      S_i: time when the router starts to transmit packet i

      F_i: time when router finishes transmitting packet i

      Clearly, $F_i = S_i + P_i$

When do we start transmitting packet i?Depends on whether packet i arrived before or after the router finishes transmitting packet i-1 for the flow

- Let A_i denote the time that packet i arrives at the routerThen

  $S_i = \max(F_{i-1}, A_i)F_i = \max(F_{i-1}, A_i) + P_i$

- Now for every flow, we calculate F_i for each packet that arrives using our formula. We then treat all the F_i as timestamps
- Next packet to transmit is always the packet that has the lowest timestamp. i.e the packet that should finish transmission before all others

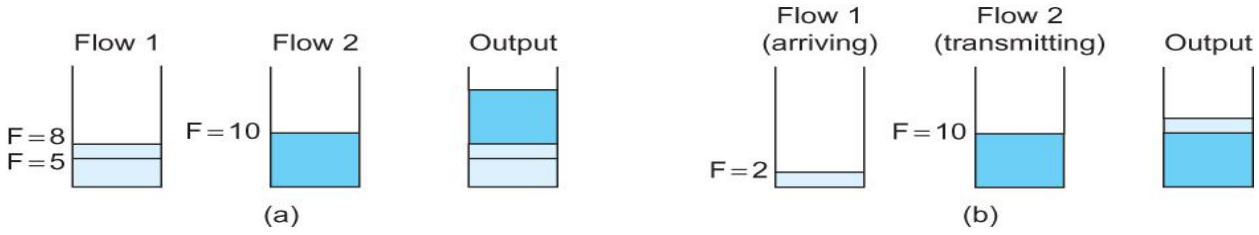


Fig Example of fair queuing in action: (a) packets with earlier finishing times are sent first; (b) sending of a packet already in progress is completed

It is possible to implement a variation of FQ, called *weighted fair queuing* (WFQ),that allows a weight to be assigned to each flow (queue). This weight logically specifieshow many bits to transmit each time the router services that queue, which effectivelycontrols the percentage of the link's bandwidth that flow will get. Simple FQ giveseach queue a weight of 1, which means that logically only 1 bit is transmitted fromeach queue each time around. This results in each flow getting 1/*n*th of the bandwidthwhen there are *n* flows.

3.11 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.

- Once a given source has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network, and that it is therefore safe to insert a new packet into the network without adding to the level of congestion.
- By using ACKs to pace the transmission of packets, TCP is said to be *self-clocking.*

3.11.1 Additive Increase Multiplicative Decrease

- 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.
- The congestion window is congestion control's counterpart to flow control's advertised window.
- 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
- TCP's effective window is revised as follows:

  MaxWindow = MIN(CongestionWindow, AdvertisedWindow)

  EffectiveWindow = MaxWindow − (LastByteSent − LastByteAcked).

- Thus, a TCP source is allowed to send no faster than the slowest component—the network or the destination host—can accommodate.
- The TCP source sets the CongestionWindow based on the level of congestion it perceives to exist in the network.
- 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 *additive increase/multiplicative decrease (AIMD)*
- 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.
- CongestionWindow is not allowed to fall below the size of a single packet, or in TCP terminology, the *maximum segment size (MSS).*
- We also need to be able to increase the congestion window to take advantage of newly available capacity in the network. This is the "additive increase" part of AIMD, and it works as follows.

  Every time the source successfully sends a CongestionWindow's worth of packets—that is, each packet sent out during the last RTT has been ACKed—it adds the equivalent of 1 packet to CongestionWindow.

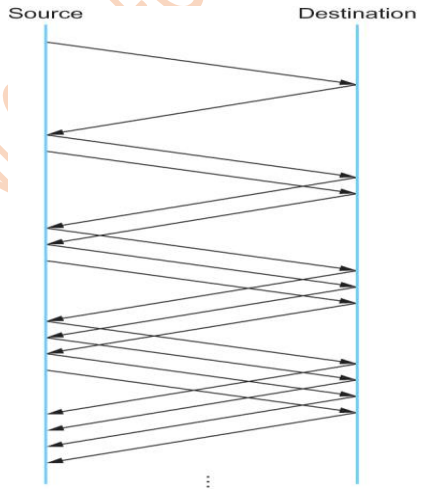


Fig Packets in transit during additive increase, with one packet being added each RTT.

- Specifically, the congestion window is incremented as follows each time an ACK arrives:

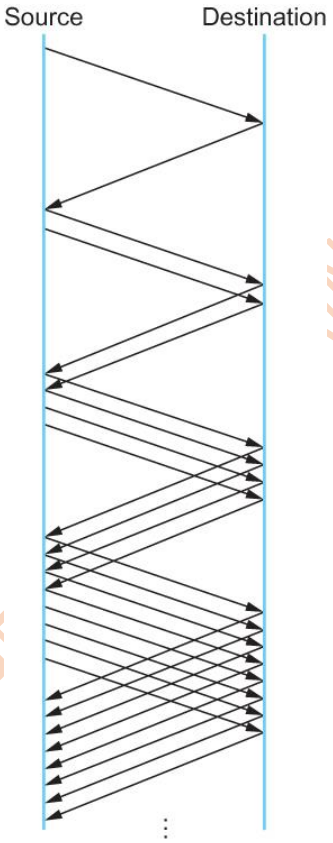
  Increment = MSS × (MSS/CongestionWindow)

  CongestionWindow+= Increment

3.11.2 Slow Start

- 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.
- TCP therefore 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 windowexponentially, rather than linearly.
- 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.
- There are actually 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.
        - In this situation, slow start continues to double CongestionWindow each RTT until there is a loss, at which time a timeout causes multiplicative decrease to divide CongestionWindow by 2.
    - The second situation occurs when the connection goes dead while waiting for a timeout to occur.
        - The source has a current (and useful) value of CongestionWindow; this is the value of CongestionWindow that existed prior to the last packet loss, divided by 2 as a result of the loss. We can think of this as the “target” congestion window.
        - Slow start is used to rapidly increase the sending rate up to this value, and then additive increase is used beyond this point.

- TCP introduces a temporary variable to store the target window, typically called CongestionThreshold, that is set equal to the CongestionWindow value that results from multiplicative decrease.
- The variable CongestionWindow is then reset to one packet, and it is incremented by one packet for every ACK that is received until it reaches.
- CongestionThreshold, at which point it is incremented by one packet per RTT.

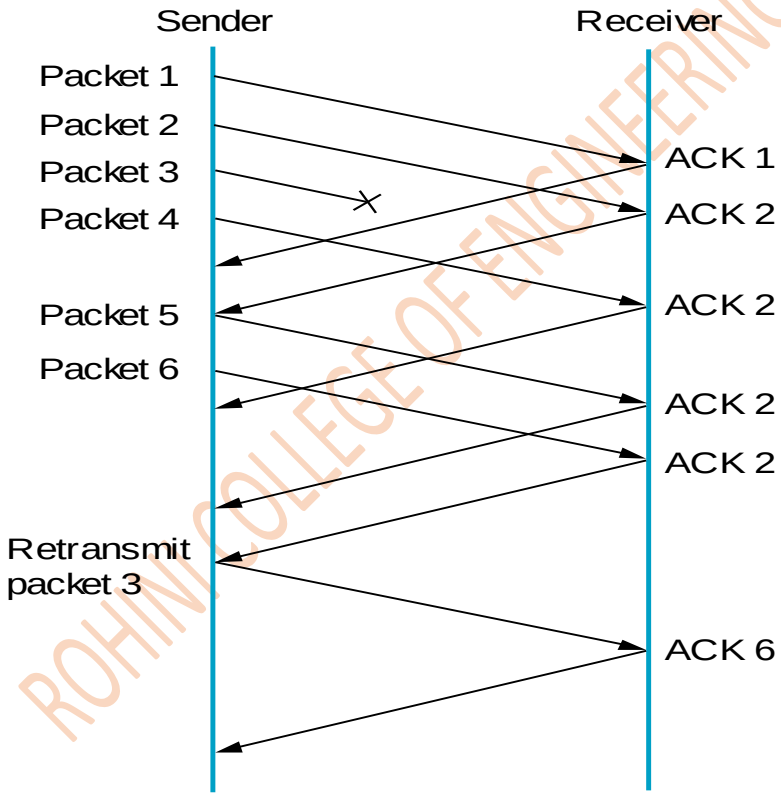


Packets in transit during slow start

3.11.3 Fast Retransmit and Fast Recovery

- 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 *duplicate ACK*.

- When the 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.
- Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it sees some number of duplicate ACKs and then retransmits the missing packet. In practice, 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 all the way back to one packet and run 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.



3.11 Congestion avoidance

When congestion is about to happen and then to reduce the rate at which hosts send data just before packets start being discarded. We call such a strategy*congestion avoidance*.

This section describes three different congestion-avoidance mechanisms.

The first two take a similar approach: They put a small amount of additional functionality into the router to assist the end node in the anticipation of congestion.

The third mechanism is very different from the first two: It attempts to avoid congestion purely from the end nodes.

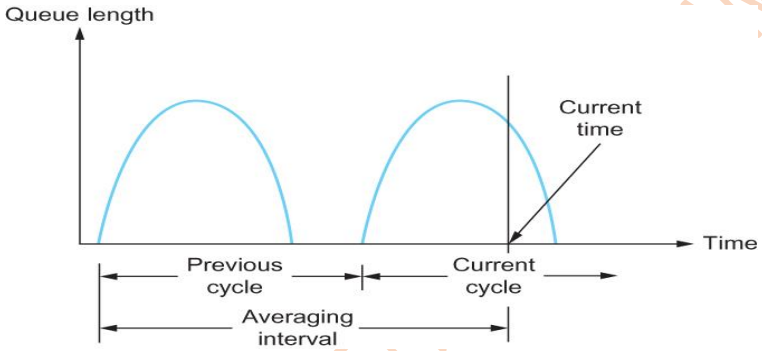
3.11.1 DECbit

- This mechanism was developed for use on the Digital Network Architecture (DNA), a connectionless network with a connection-oriented transport protocol. This mechanism could, therefore, also be applied to TCP and IP.
- The idea is to more evenly split the responsibility for congestion control between the routers and the end nodes.
- Each router monitors the load it is experiencing and explicitly notifies the end nodes when congestion is about to occur.
- This notification is implemented by setting a binary congestion bit in the packets that flow through the router; hence the name DECbit.
- The destination host then copies this congestion bit into the ACK it sends back to the source. Finally, the source adjusts its sending rate so as to avoid congestion.

Description of the 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 greater than or equal to 1 at the time the packet arrives.
- This average queue length is measured over a time interval that spans the last busy+idle cycle, plus the current busy cycle.
- Essentially, the router calculates the area under the curve and divides this value by the time interval to compute the average queue length.
- Using a queue length of 1 as the trigger for setting the congestion bit is a trade-off between significant queuing (and hence higher throughput) and increased idle time (and hence lower delay).

- In other words, a queue length of 1 seems to optimize the power function.
- The source records how many of its packets resulted in some router setting the congestion bit.
- In particular, the source maintains a congestion window, just as in TCP, and watches to see what fraction of the last window's worth of packets resulted in the bit being set.
- If less than 50% of the packets had the bit set, then the source increases its congestion window by one packet.
- If 50% or more of the last window's worth of packets had the congestion bit set, then the source decreases its congestion window to 0.875 times the previous value.
- The value 50% was chosen as the threshold based on analysis that showed it to correspond to the peak of the power curve. The "increase by 1, decrease by 0.875" rule was selected because additive increase/multiplicative decrease makes the mechanism stable.



Computing average queue length at a router

3.11.2 Random Early Detection (RED)

A second mechanism, called *random early detection* (RED), is similar to the DECbit scheme in that each router is programmed to monitor its own queue length, and when it detects that congestion is imminent, to notify the source to adjust its congestion window. RED, invented by Sally Floyd and Van Jacobson in the early 1990s.

RED Differs from DEC in two ways.

1) Rather than explicitly sending a congestion notification message to the source, RED is most commonly implemented such that it *implicitly notifies* the source of congestion by dropping one of its packets.

- The source is, therefore, effectively notified by the subsequent timeout or duplicate ACK.
- RED is designed to be used in conjunction with TCP, which currently detects congestion by means of timeouts (or some other means of detecting packet loss such as duplicate ACKs).
- As the "early" part of the RED acronym suggests,the router drops a few packets before it has exhausted its buffer space completely, so as to cause the source to slow down, with the hope that this will mean it does not have to drop lots of packets later on.

2) The second difference between RED and DECbit is in the details of how RED decides when to drop a packet and what packet it decides to drop.

- To understand the basic idea, consider a simple FIFO queue. 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 ideais *called early random drop. The RED algorithm defines the details of how to monitor the queue length and* when to drop a packet.

Description

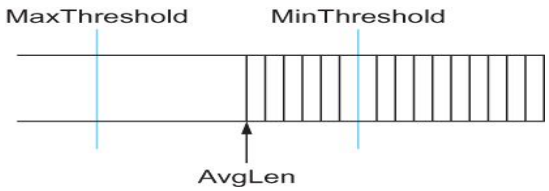
1) RED computes an average queue length. AvgLen is computed as
   AvgLen = (1 − Weight) × AvgLen + Weight × SampleLen
   where 0 < Weight < 1 and SampleLen is the length of the queue when a sample measurement is made.
2) RED has two queue length thresholds that trigger certain activity: MinThreshold and MaxThreshold.
3) When a packet arrives at the gateway, RED compares the current AvgLen with these two thresholds, according to the following rules:

```
if AvgLen ≤ MinThreshold  queue the packet
if MinThreshold < AvgLen < MaxThreshold
        calculate probability P
        drop the arriving packet with probability P
if MaxThreshold ≤ AvgLen
        drop the arriving packet
```

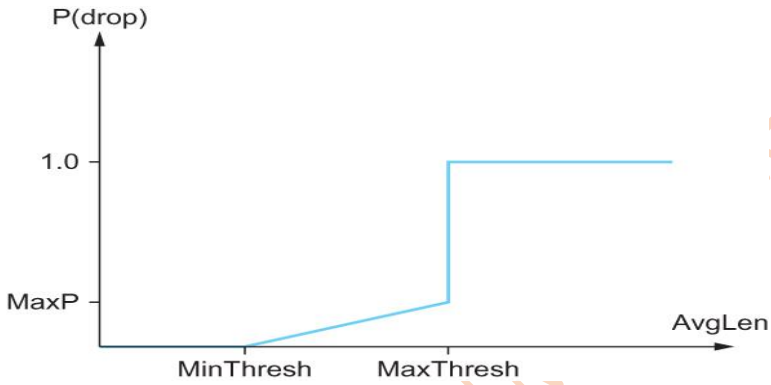
P is a function of both AvgLen and how long it has been since the last packet was dropped. Specifically, it is computed as follows:

$$TempP = MaxP \times (AvgLen - MinThreshold)/(MaxThreshold - MinThreshold)$$

$$P = TempP/(1 - count \times TempP)$$



RED thresholds on a FIFO queue



Drop probability function for RED

3.11.3 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.

First Algorithm exploits this observation as follows: The congestion window normally increases as in TCP, but every two round-trip delays the algorithm checks to see if the current RTT is greater than the average of the minimum and maximum RTTs seen so far. If it is, then the algorithm decreases the congestion window by one-eighth.

A second algorithm does something similar. The decision as to whether or not to change the current window size is based on changes to both the RTT and the window size. The window is adjusted once every two round-trip delays based on the product

(CurrentWindow − OldWindow) ×(CurrentRTT − OldRTT)

If the result is positive, the source decreases the window size by one-eighth; if the result is negative or zero, the source increases the window by one maximum packet size. Note that the window changes during every adjustment; that is, it oscillates around its optimal point.

A third scheme takes advantage of this fact. Every RTT, it increases the window size by one packet and compares the throughput achieved to the throughput when the window was one packet smaller. If the difference is less than one-half the throughput achieved when only one packet was in transit—as was the case at the beginning of the connection—the algorithm decreases the window by one packet.

This scheme calculates the throughput by dividing the number of bytes outstanding in the network by the RTT.

A fourth mechanism, is similar to this last algorithm in that it looks at changes in the throughput rate, or more specifically, changes in the sending rate. However, it differs from the third algorithm in the way it calculates throughput, and instead of looking for a change in the slope of the throughput, it compares the measured throughput rate with an expected throughput rate. The algorithm, TCP VEGAS is given below

- Set BaseRTT to the minimum of all measured round-trip times
- Calculate expected throughput

  ExpectedRate = CongestionWindow / BaseRTT

  Where CongestionWindow is the TCPcongestion window, which is to equal to the number of bytes in transit.
- Calculate ActualRate, given by Divide the number of bytes transmitted by the sample RTT.
- Compare ActualRate to ExpectedRate and adjuststhe window accordingly.

  Let Diff =ExpectedRate – ActualRate

  Two thresholds, $\alpha < \beta$, roughly cor-responding to having too little and toomuch extra data in the network, respec tively
- When Diff $< \alpha$, TCP Vegas in-creases the congestion window linearly
- Wwhen Diff $> \beta$, TCP Vegas decreases the congestionwindow linearly
- TCP Vegas leaves the congestion windowunchanged when $\alpha <$ Diff $< \beta$.