Congestion Control

TCP Congestion Control

- Goal of TCP is to determine the available network capacity and to prevent network overload.
- Depends on other connections that share the link
- Originally TCP assumed FIFO queuing.

Why Prevent Congestion ?

- Congestion is bad for the overall performance in the network.
  - Excessive delays can be caused.
  - Retransmissions may result due to dropped packets
    - Waste of capacity and resources.
  - In some cases (UDP) packet losses are not recovered.
- Note: Main reason for lost packets in the Internet is due to congestion -- errors are rare.

Congestion Window

- CongestionWindow (cwnd) is a variable held by the TCP source for each connection.

$$MaxWindow = min\ (CongestionWindow\ ,\ AdvertisedWindow)$$

$$EffectiveWindow = MaxWindow - (LastByteSent - LastByteAcked)$$

- cwnd is set based on the perceived level of congestion. The Host receives *implicit* (packet drop) or *explicit* (packet mark) indications of internal congestion.

Managing the Congestion Window

- Decrease Congestion window when TCP perceives high congestion.
- Increase Congestion window when TCP knows that there is not much congestion.
- How much Increase / Decrease ?
  - Since increased congestion is more catastrophic, reduce it more aggressively.
  - Increase is additive, decrease is multiplicative - called the Additive Increase/Multiplicative Decrease (AIMD) behavior of TCP.

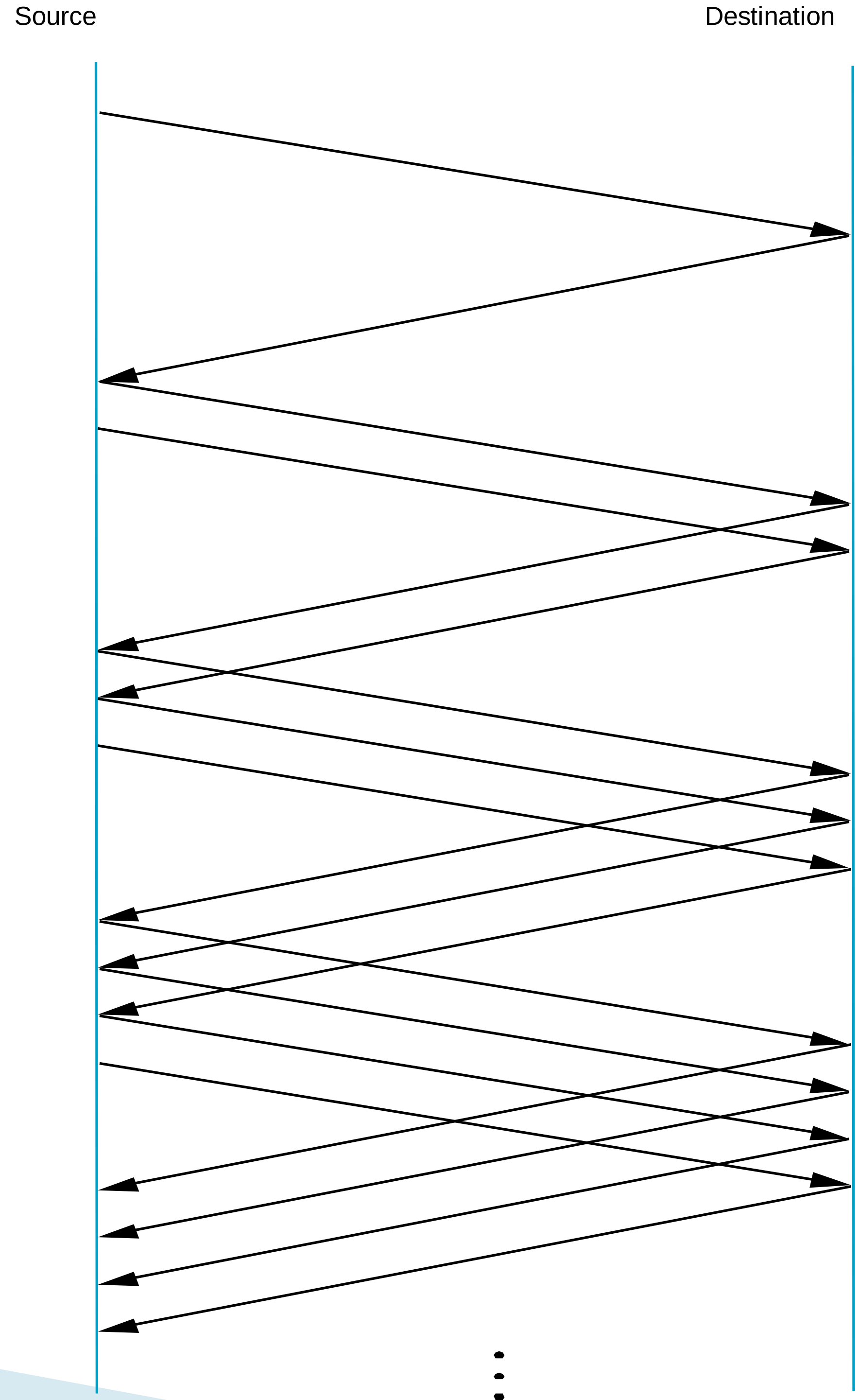
Additive Increase

- Additive Increase is a reaction to perceived available capacity.
- For each cwnd's worth of packets sent, increase cwnd by 1 packet.
- In practice, cwnd is incremented fractionally for each arriving ACK.

$$increment = MSS \times (MSS \, / \, cwnd)$$

$$cwnd = cwnd + increment$$

Additive Increase



Multiplicative Decrease

- The key assumption is that a dropped packet and the resultant timeout (no ack) are due to congestion at a router or a switch.
- TCP reacts to a timeout by *halving cwnd*.
- Although cwnd is defined in bytes, the literature often discusses congestion control in terms of packets (or more formally in MSS == Maximum Segment Size).
- cwnd is not allowed below the size of a single packet.

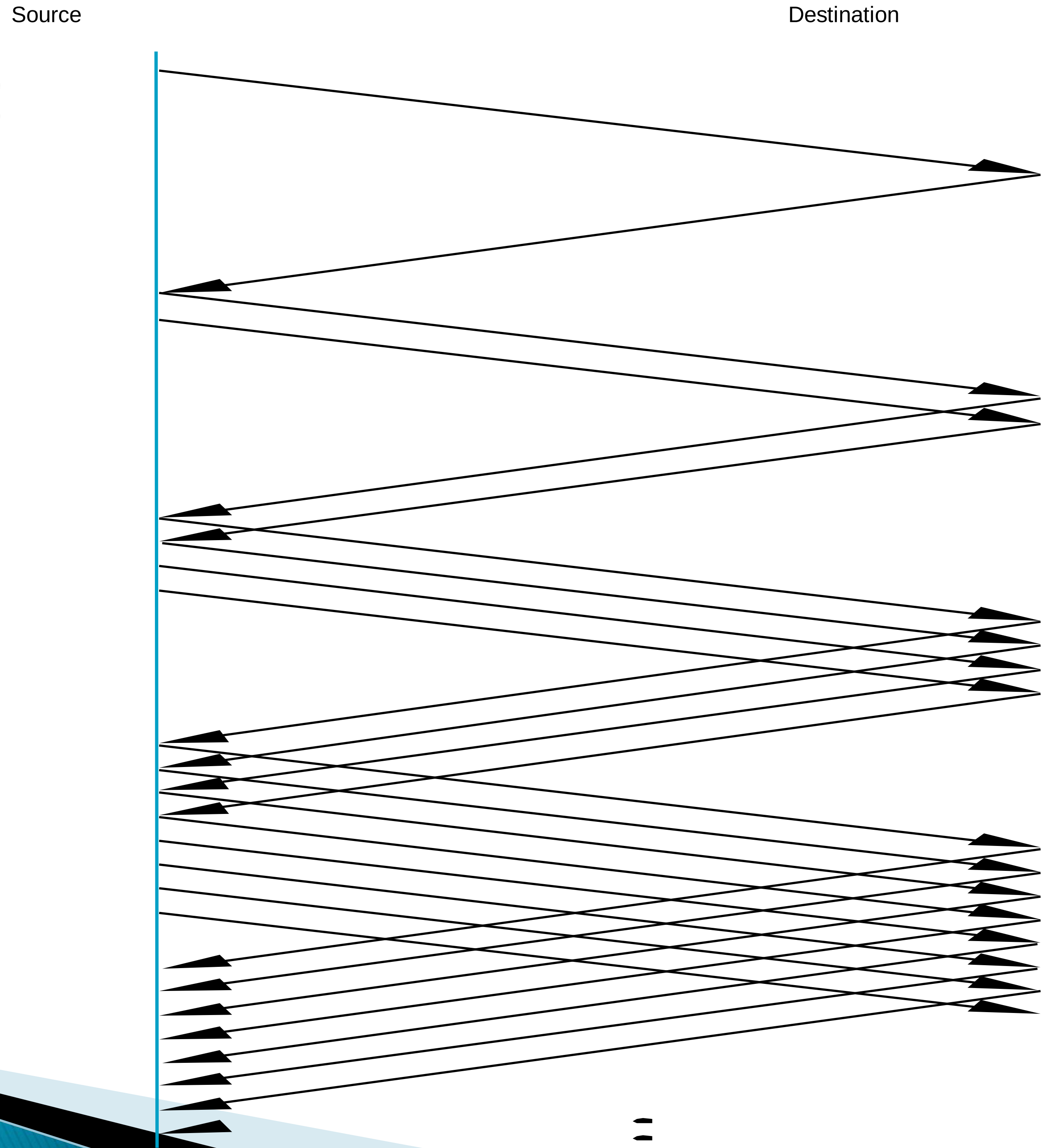
Slow Start

- Linear additive increase takes too long to ramp up a new TCP connection from cold start.
- Beginning with TCP, the slow start mechanism was added to provide an initial exponential increase in the size of cwnd.

Slow Start

- The source starts with cwnd = 1.
- Every time an ACK arrives, cwnd is incremented.
- cwnd is effectively doubled per RTT "epoch".
- Two slow start situations:
  - At the very beginning of a connection **{cold start}**.
  - When the connection goes dead waiting for a timeout to occur (i.e, the advertized window goes to zero!)

Slow Start



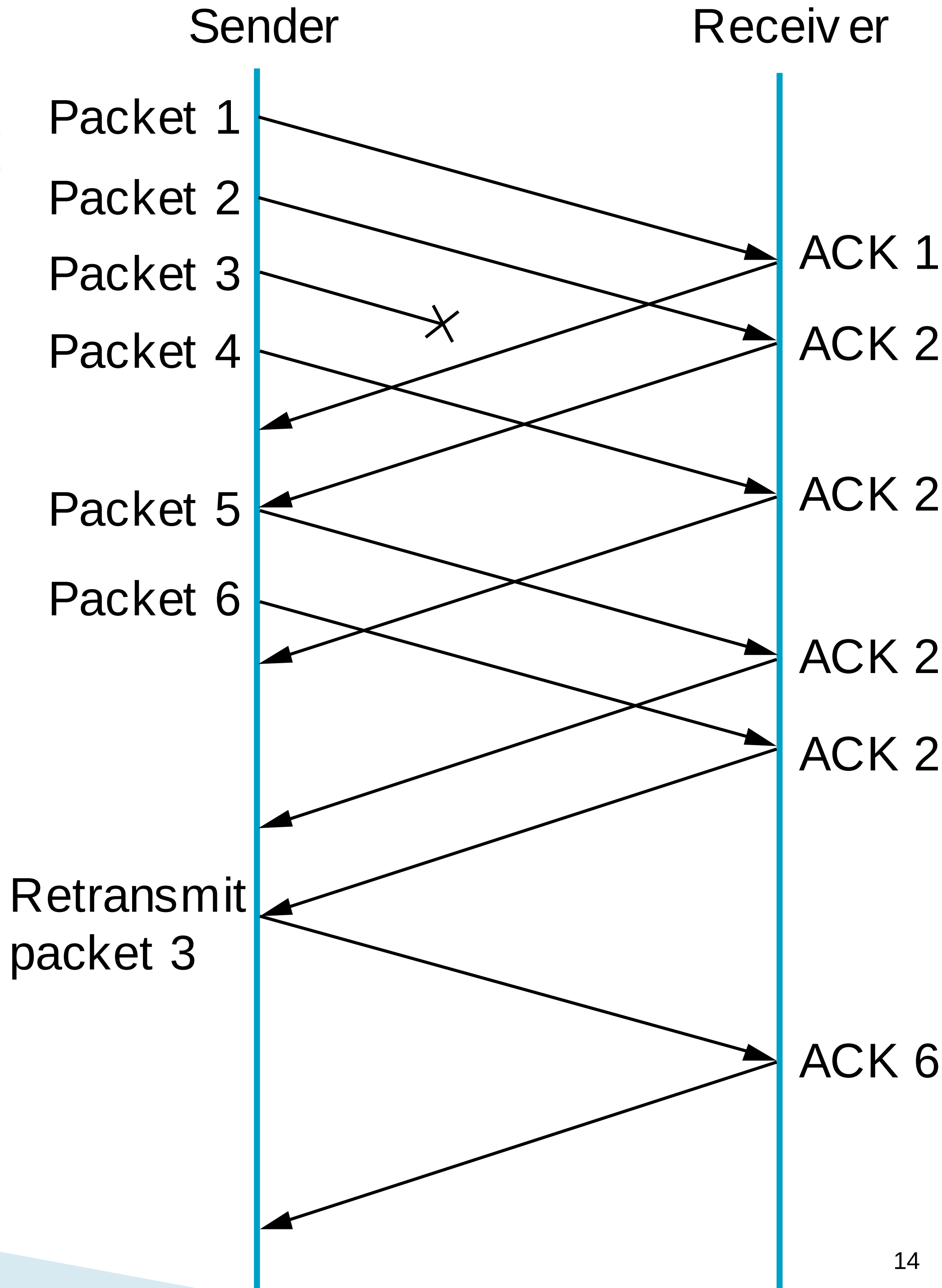
Fast Retransmit

- Coarse timeouts remained a problem, and Fast retransmit was added with TCP.
- Since the receiver responds every time a packet arrives, this implies the sender will see duplicate ACKs.
- Use duplicate ACKs to signal lost packet.
- Upon receipt of *three duplicate ACKs,* the TCP Sender retransmits the lost packet.

Fast Retransmit

- Generally, fast retransmit eliminates about half the coarse-grain timeouts.
- This yields roughly a 20% improvement in throughput.
- Note – Fast Retransmit does not eliminate all the timeouts due to small window sizes at the source.

Fast Retransmit



Fast Recovery

- Fast recovery was added with TCP.
- When fast retransmit detects three duplicate ACKs, start the recovery process from congestion
- After Fast Retransmit, half the cwnd and commence *recovery from this point using linear additive increase*.

Congestion Avoidance

Introduction

- TCP repeatedly increases the load on the network in an effort to find the point at which congestion occurs, and then it backs off from this point.
- TCP *needs* to create losses to find the available bandwidth of the connection
- Predict when congestion is about to happen and then to reduce the rate at which hosts send data just before packets start being discarded

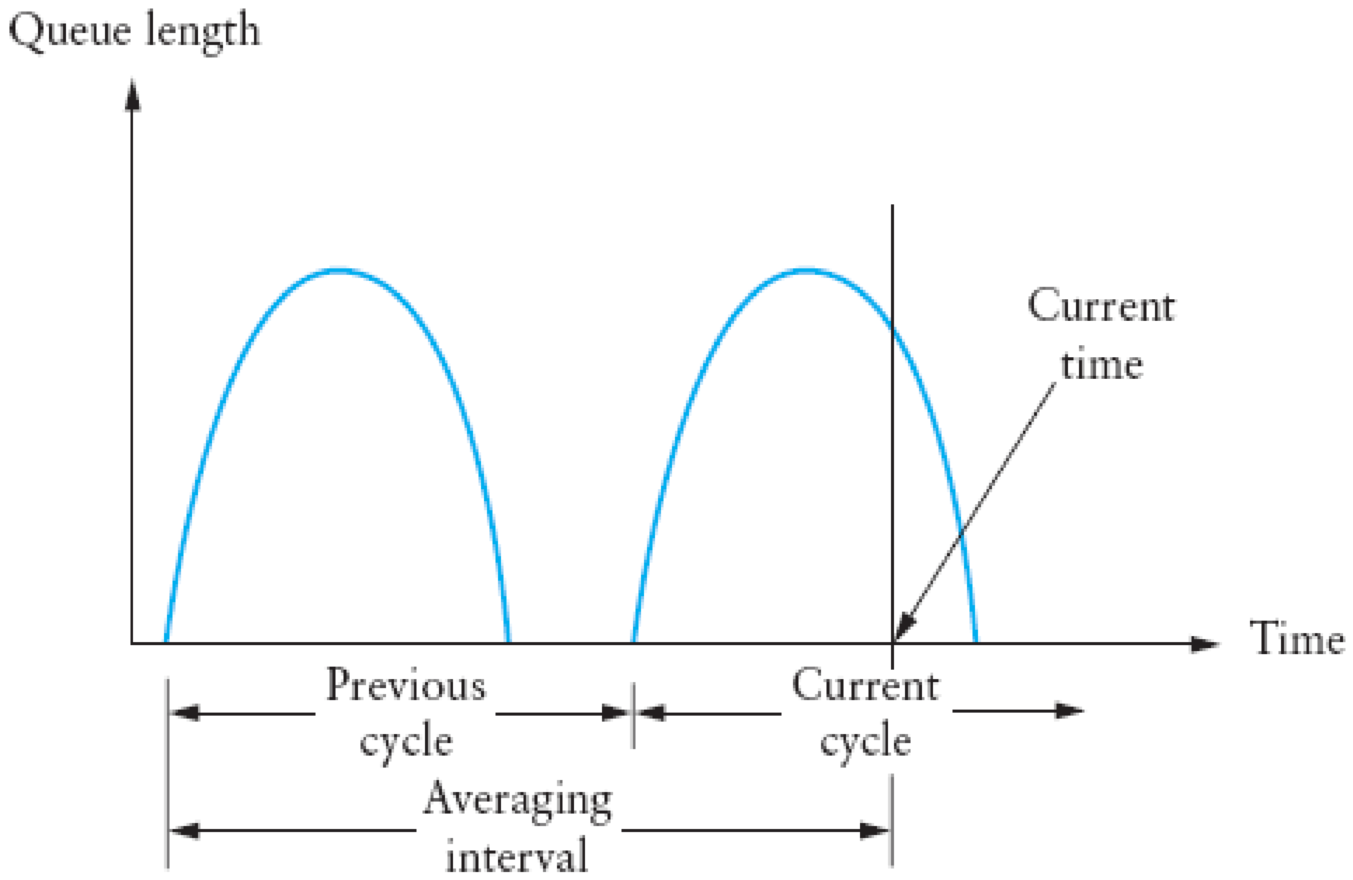
DECbit

- The first mechanism was developed for use on the Digital Network Architecture (DNA)
- 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
- Notification is implemented by setting a binary congestion bit in the packets that flow through the router

DECbit

- The destination host then copies this congestion bit into the ACK it sends back to the source
- The source adjusts its sending rate so as to avoid congestion.
- 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
- *Average Queue Length = Last busy cycle +idle cycle + the current busy cycle*

DECbit



Queue length
Current
time
Time
Previous
cycle
Current
cycle
Averaging
interval

Random Early Detection (RED)

- Invented by Sally Floyd and Van Jacobson
- Similar to the DECbit
- 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
- Differs from the DECbit scheme in two major ways

RED (First Difference)

- Rather than explicitly sending a congestion notification message to the source, router *implicitly* notifies the source about congestion by dropping one of its packets.
- The source is, therefore, effectively notified by the subsequent timeout or duplicate ACK

RED (Second Difference)

- The details of how RED decides when to drop a packet and what packet it decides to drop.
- Consider a simple FIFO queue
- Rather than wait for the queue to become completely full and then be forced to drop each arriving packet, decide to drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*.

Average Queue Length

- $AvgLen = (1 - Weight) \times AvgLen + Weight \times SampleLen$
- 0 <Weight < 1
- SampleLen is the length of the queue when a sample measurement is made

Queue Length Thresholds

- RED has two queue length thresholds MinThreshold and MaxThreshold
- if $\mathit{AvgLen} \leq \mathit{MinThreshold}$

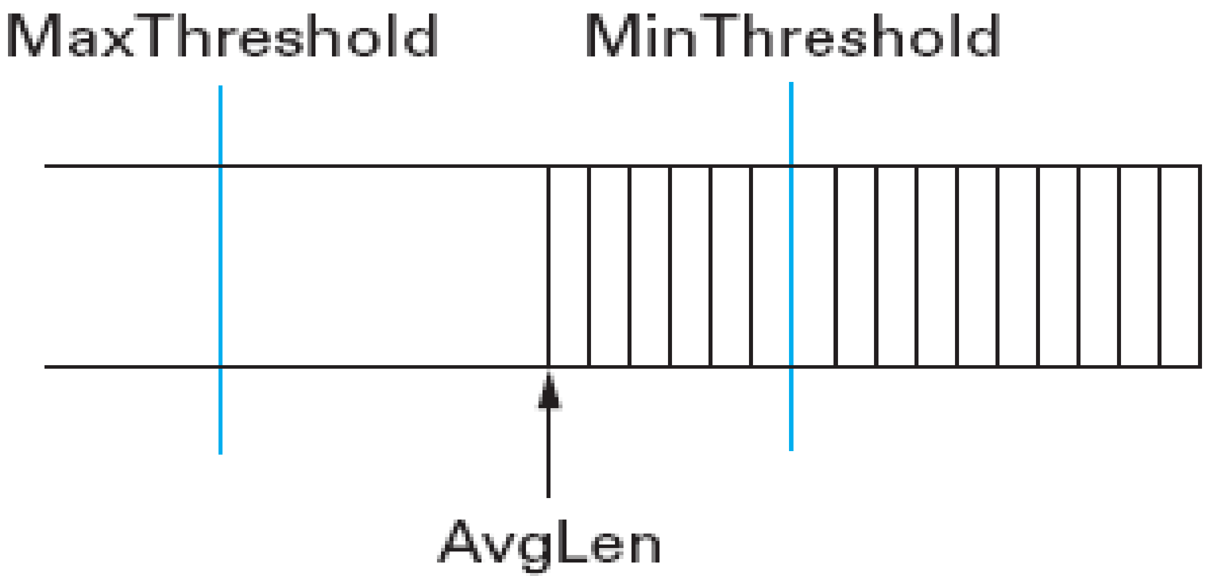
  $\rightarrow$ queue the packet
- if $\mathit{MinThreshold} < \mathit{AvgLen} < \mathit{MaxThreshold}$

  $\rightarrow$ calculate probability P

  $\rightarrow$ drop the arriving packet with probability P
- if $\mathit{MaxThreshold} \leq \mathit{AvgLen}$

  $\rightarrow$ drop the arriving packet

RED Thresholds on a FIFO Queue



Source Based Congestion Avoidance

- Having routers participate in congestion control requires changes to core routers is difficult.
- It is better to do this end-to-end.
- However, we want to still have source based control -- now, it would be source based congestion avoidance.
- We need a TCP that watches out for signs of congestion

First Algorithm

- How much does the RTT increase with each packet sent ?
  - Note that with each additional packet, we are adding load.
- One way is to compute for every two round trip delays (with an increase in a segment) to see if
  - Observed RTT > avg of min and maximum RTT.
- If yes, reduce congestion window one eighth.

Second Algorithm

- 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

  $$(CurrentWindow - OldWindow) \times (CurrentRTT - OldRTT)$$

- 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