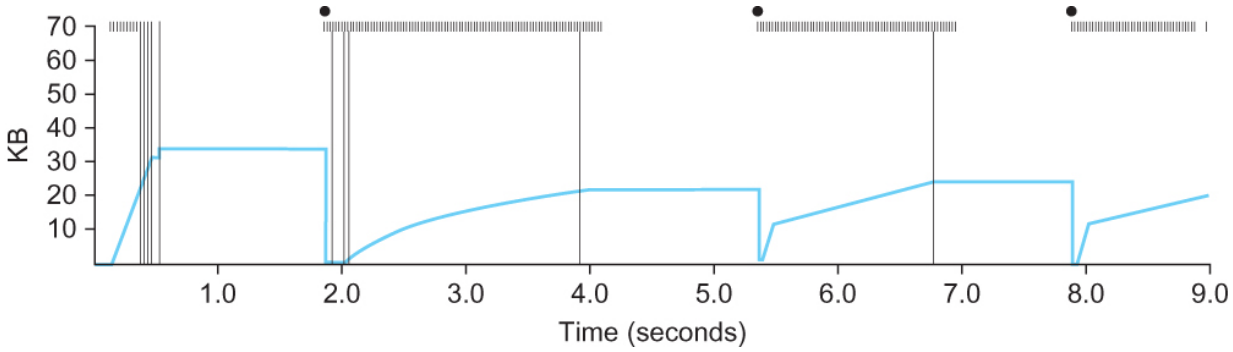
# Networks and Distributed Systems

Lecture 18 – TCP fast retransmission and fast recovery

Congestion avoidance

# Slow Start






Behavior of TCP congestion control. Colored line = value of CongestionWindow over time; solid bullets at top of graph = timeouts; hash marks at top of graph = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

# Fast Retransmit and Fast Recovery

- The mechanisms described so far were part of the original proposal to add congestion control to TCP.
- It was soon discovered, however, that the coarse-grained implementation of TCP timeouts led to long periods of time during which the connection went dead while waiting for a timer to expire.
- Because of this, a new mechanism called *fast retransmit* was added to TCP.
- Fast retransmit is a heuristic that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism.

# 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— that is, TCP cannot yet acknowledge the data the packet contains because earlier data has not yet arrived— TCP resends the same acknowledgment it sent the last time.

# Fast Retransmit and Fast Recovery

- 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.

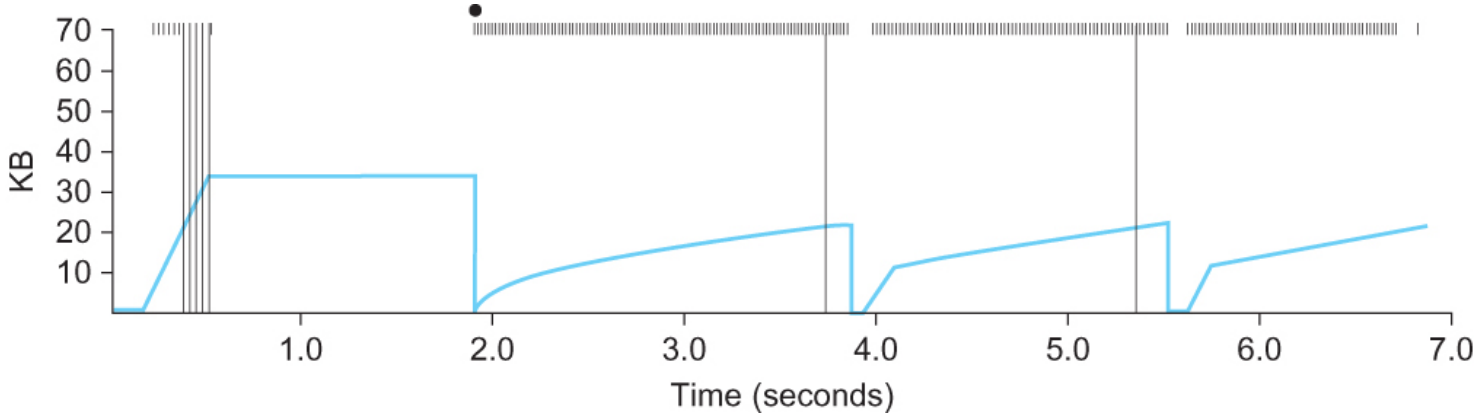
# Fast Retransmit and Fast Recovery

- 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.

# TCP Congestion Control

- Fast Retransmit and Fast Recovery




Trace of TCP with fast retransmit. Colored line = CongestionWindow; solid bullet = timeout; hash marks = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

# Next slides taken from Ion Stoica lectures from UC Berkeley

# Slow Start/AIMD Pseudocode



```
Initially:
    cwnd = 1;
    ssthresh = infinite;
New ack received:
    if (cwnd < ssthresh)
        /* Slow Start*/
        cwnd = cwnd + 1;
    else
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd;
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2;
    cwnd = 1;
```

# The big picture (with timeouts)



**Initially:**

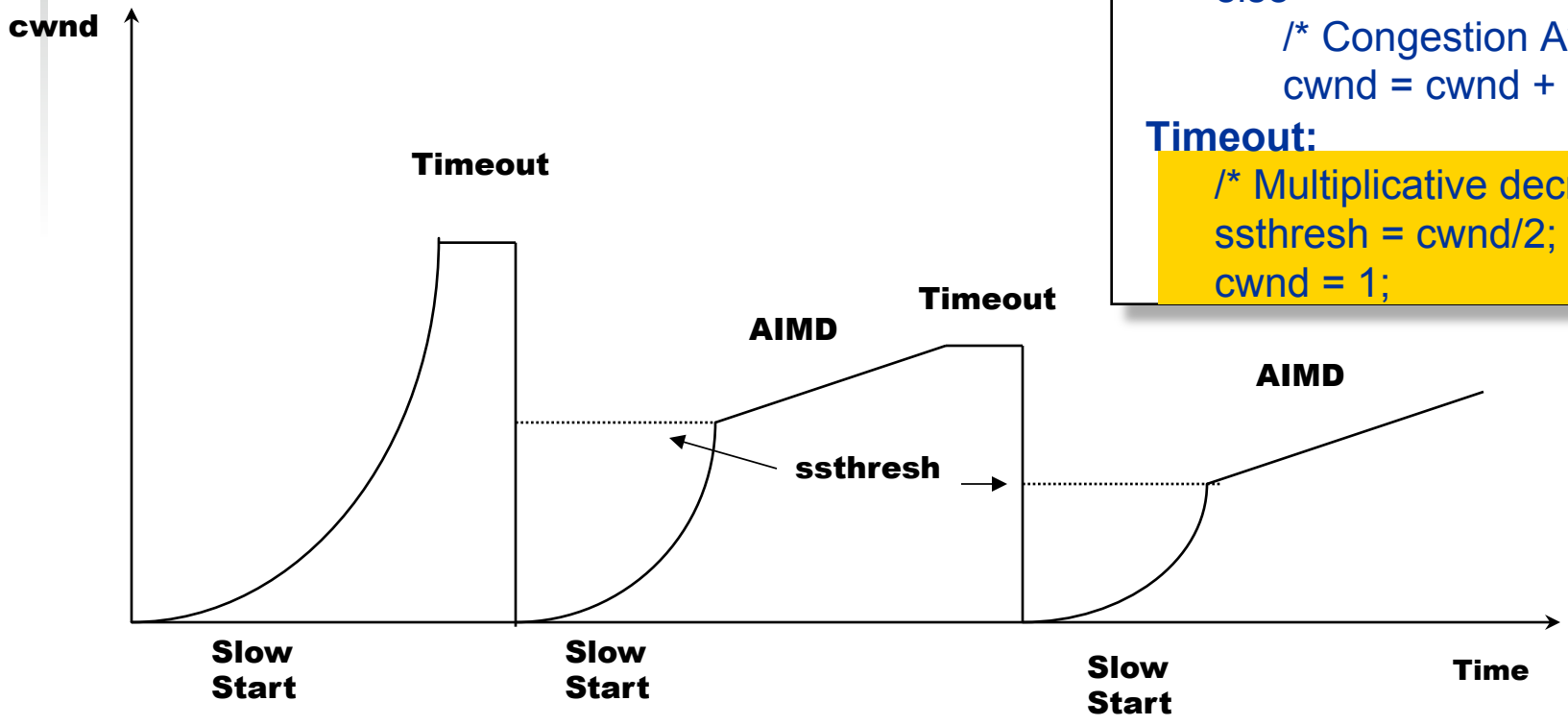
```
cwnd = 1;
ssthresh = infinite;
```

**New ack received:**

```
if (cwnd < ssthresh)
    /* Slow Start*/
    cwnd = cwnd + 1;
else
    /* Congestion Avoidance */
    cwnd = cwnd + 1/cwnd;
```

**Timeout:**

```
/* Multiplicative decrease */
ssthresh = cwnd/2;
cwnd = 1;
```



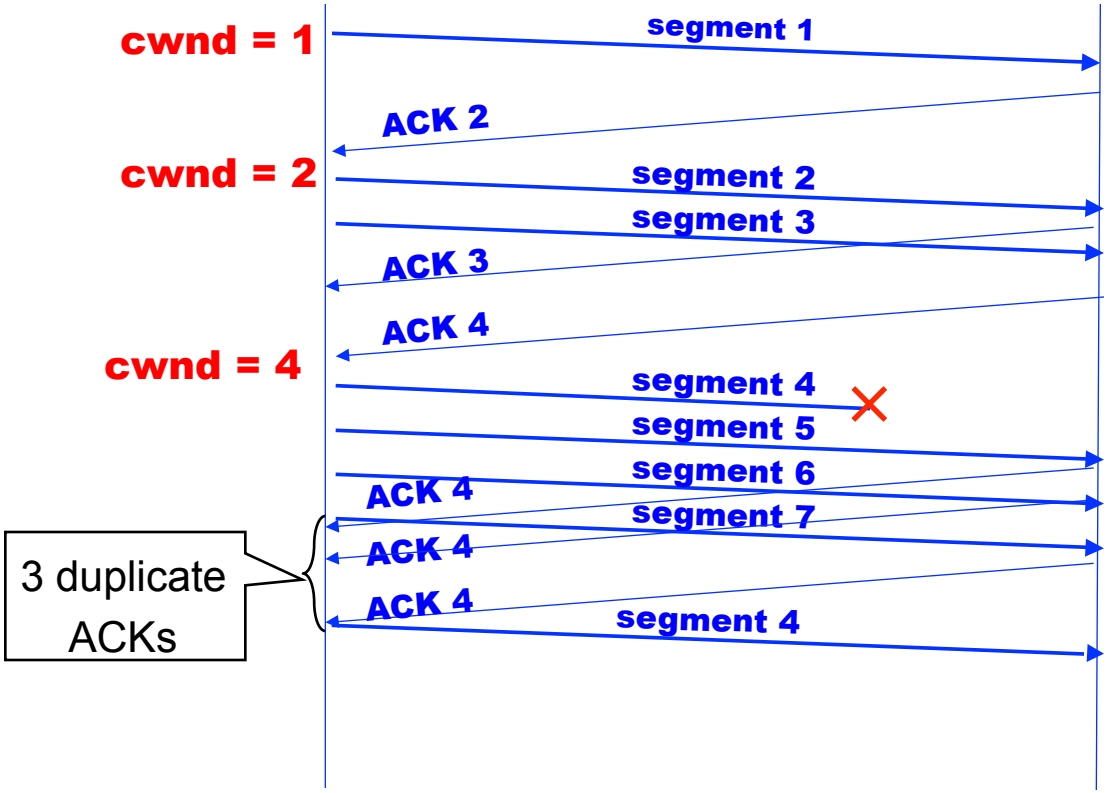
# Congestion Detection Revisited



- Wait for Retransmission Time Out (RTO)
  - RTO kills throughput
- In BSD TCP implementations, RTO is usually more than 500ms
  - The granularity of RTT estimate is 500 ms
  - Retransmission timeout is RTT + 4 * mean_deviation
- Solution: Don't wait for RTO to expire

# Fast Retransmits

- Resend a segment after 3 duplicate ACKs
  - Duplicate ACK means that an out-of sequence segment was received
- Notes:
  - ACKs are for next expected packet
  - Packet reordering can cause duplicate ACKs
  - Window may be too small to get enough duplicate ACKs

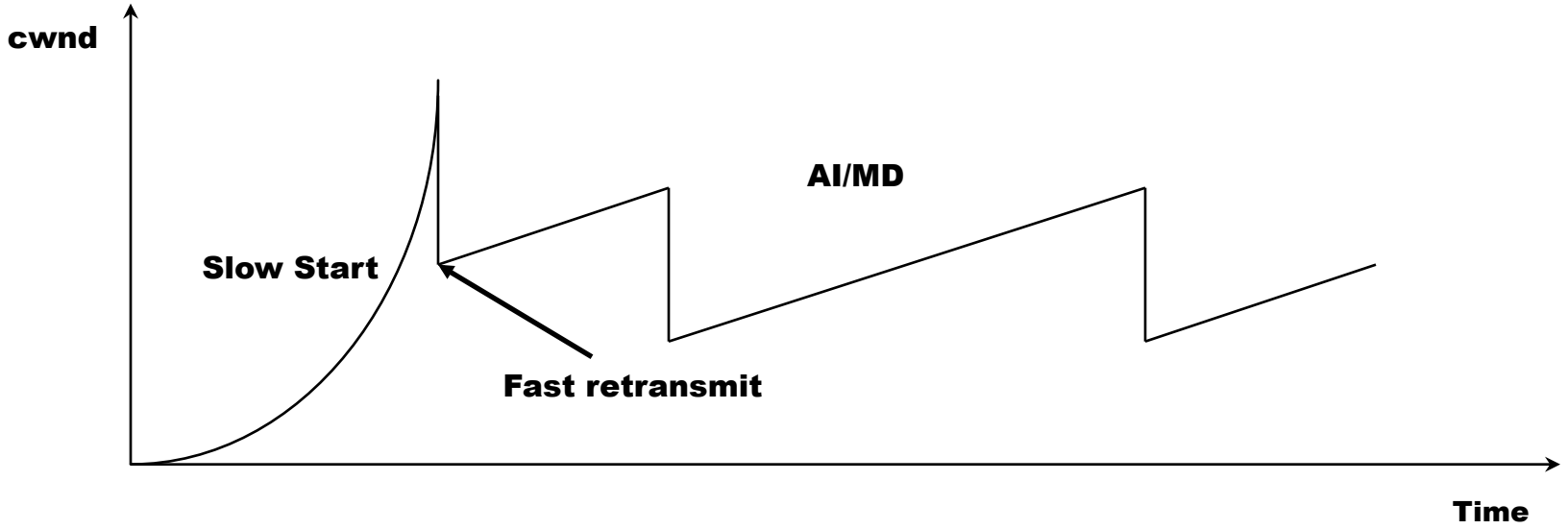


# Fast Recovery: After a Fast Retransmit



- *ssthresh* = *cwnd* / 2
- *cwnd* = *ssthresh*
  - Instead of setting *cwnd* to 1, cut cwnd in half (multiplicative decrease)
- For each dup ack arrival
  - *dupack*++
  - Indicates packet left network, so we may be able to send more
  - MaxWindow = min(*cwnd* + *dupack*, AdvWin)
- Receive ack for new data (beyond initial dup ack)
  - *dupack* = 0
  - Exit fast recovery
- But when RTO expires still do *cwnd* = 1

# Fast Retransmit and Fast Recovery




- Retransmit after 3 duplicated acks
  - Prevent expensive timeouts
- Reduce slow starts
- At steady state, *cwnd* oscillates around the optimal window size

# TCP Congestion Control Summary

- Measure available bandwidth
  - Slow start: fast, hard on network
  - AIMD: slow, gentle on network
- Detecting congestion
  - Timeout based on RTT
    - Robust, causes low throughput
  - Fast Retransmit: avoids timeouts when few packets lost
    - Can be fooled, maintains high throughput
- Recovering from loss
  - Fast recovery: don't set cwnd=1 with fast retransmits

# TCP Flavors

- TCP-Tahoe
  - cwnd =1 whenever drop is detected
- TCP-Reno
  - cwnd =1 on timeout
  - cwnd = cwnd/2 on dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK

# TCP-SACK

- SACK = Selective Acknowledgements
- ACK packets identify exactly which packets have arrived
- Makes recovery from multiple losses much easier

# Congestion Avoidance Mechanism

- It is important to understand that TCP's strategy is to control congestion once it happens, as opposed to trying to avoid congestion in the first place.
- In fact, TCP repeatedly increases the load it imposes on the network in an effort to find the point at which congestion occurs, and then it backs off from this point.
- An appealing alternative is to predict when congestion is about to happen and then to reduce the rate at which hosts send data just before packets start being discarded.
  - has not yet been widely adopted yet
- We call such a strategy *congestion avoidance, to* distinguish it from *congestion control.*

# Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - A second mechanism, called *random early detection (RED)*
  - 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, differs from the DECbit scheme in two major ways:

# Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - The first is that 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).

# Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - As the “early” part of the RED acronym suggests, the gateway drops the packet earlier than it would have to, so as to notify the source that it should decrease its congestion window sooner than it would normally have.
  - In other words, 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.

# Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - 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 idea is *called early random drop. The RED algorithm defines the details of how to monitor the queue length and* when to drop a packet.

# Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - Computes an average queue length using a weighted running average. That is, 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.
  - In most software implementations, the queue length is measured every time a new packet arrives at the gateway. In hardware, it might be calculated at some fixed sampling interval.

# Congestion Avoidance Mechanism

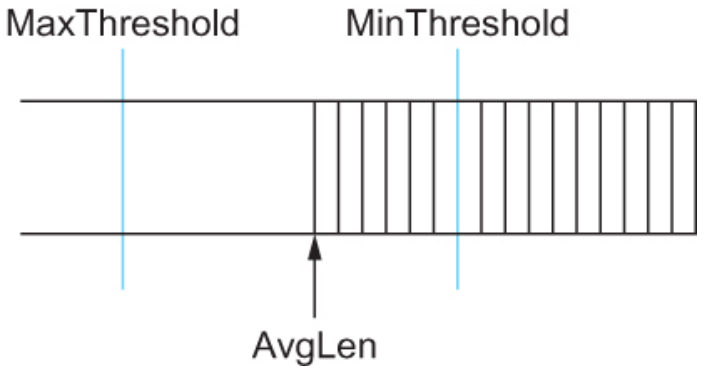
- Random Early Detection (RED)
  - Second, RED has two queue length thresholds that trigger certain activity: MinThreshold and MaxThreshold.
  - When a packet arrives at the gateway, RED compares the current AvgLen with these two thresholds, according to the following rules:
    - if AvgLen ≤ MinThreshold
      - → queue the packet
    - if MinThreshold < AvgLen < MaxThreshold
      - → calculate probability P
      - → drop the arriving packet with probability P
    - if MaxThreshold ≤ AvgLen
      - → drop the arriving packet

# Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - 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 × (AvgLen − MinThreshold)/(MaxThreshold − MinThreshold)
    - P = TempP/(1 − count × TempP)

# Congestion Avoidance Mechanism

- Random Early Detection (RED)




RED thresholds on a FIFO queue

# Congestion Avoidance Mechanism

- Random Early Detection (RED)


Drop probability function for RED