Advanced Computer Networks

Congestion control in TCP

Prof. Andrzej Duda
duda@imag.fr

http://duda.imag.fr

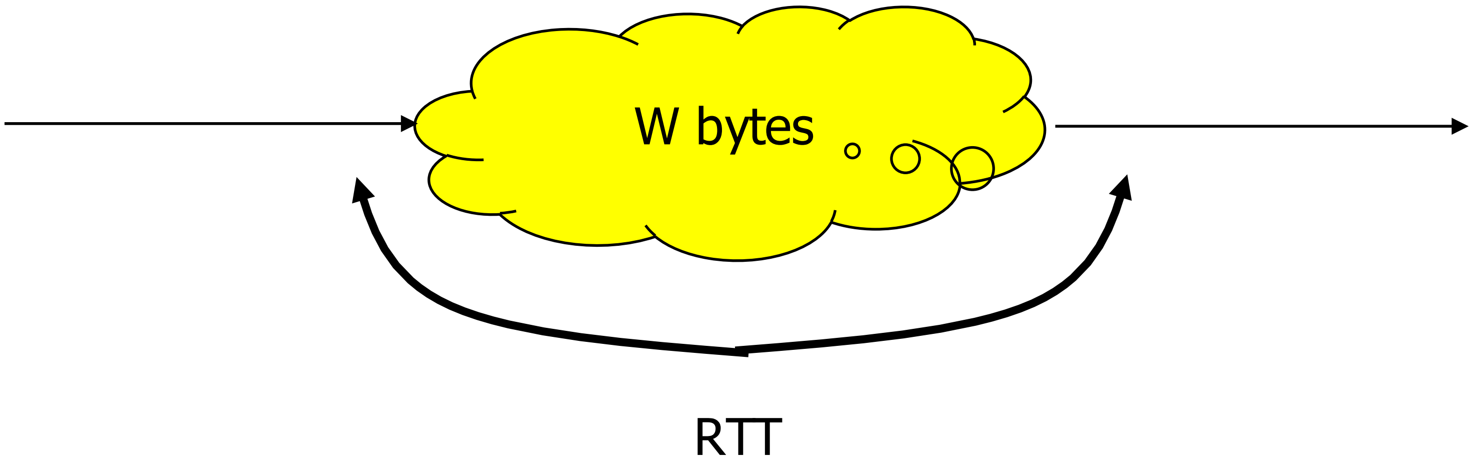
Contents

- Principles
- TCP congestion control states
  - Slow Start
  - Congestion Avoidance
  - Fast Recovery
- TCP friendly applications

TCP and Congestion Control

- TCP is used to avoid congestion in the Internet
  - a TCP source adjusts its sending window to the congestion state of the network
  - this avoids congestion collapse and ensures some fairness
- TCP sources interpret losses as a negative feedback
  - used to reduce the sending rate
- Window-based control
- UDP sources are a problem for the Internet
  - use for long lived sessions (ex: RealAudio) is a threat: congestion collapse
  - UDP sources should imitate TCP : "TCP friendly"

Sending window

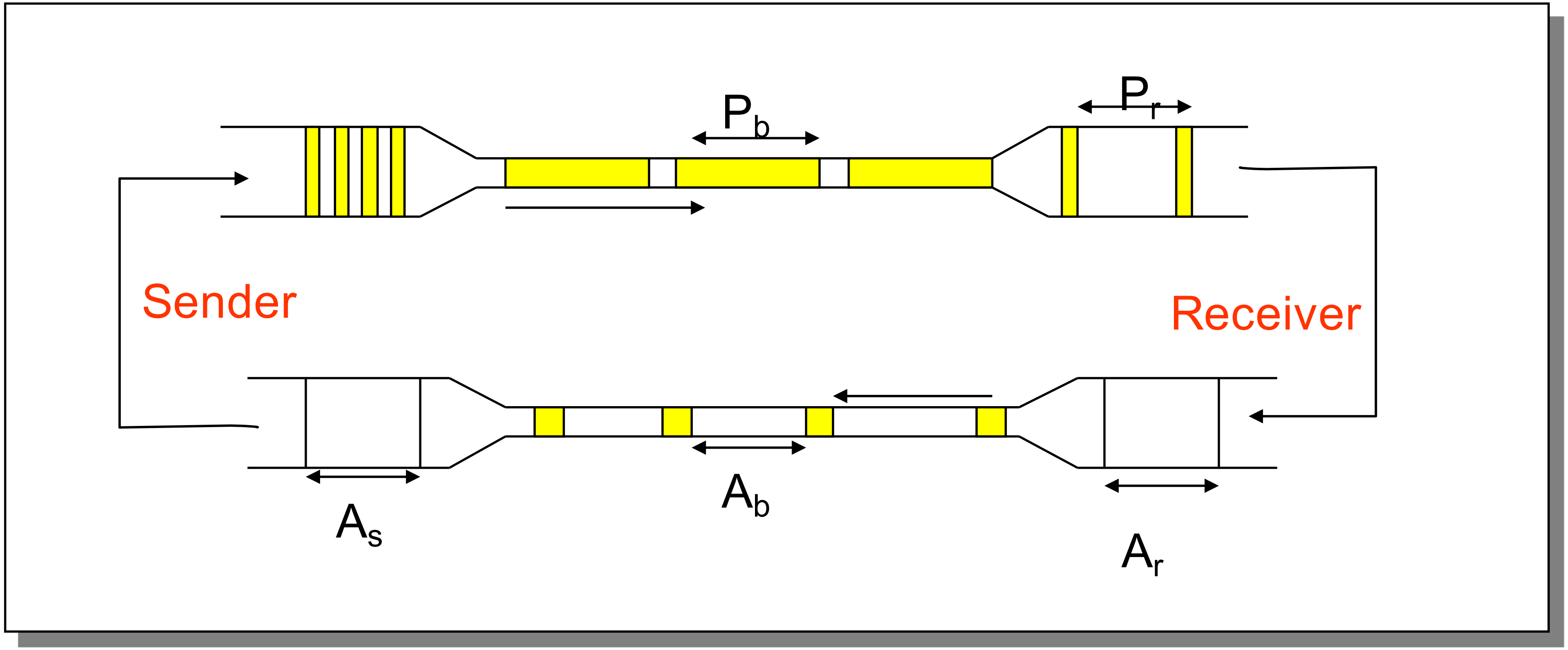


- W - the number of non ACKed bytes
  - throughput = W/RTT (Little's formulae)
- If congestion
  - RTT increases, automatic reduction of the source rate
  - additional control: decrease W

Sending window

- Sending window - number of non ACKed bytes
  - W = min (**`cwnd`**, **`OfferedWindow`**)
  - **`cwnd`**
    - congestion window - maintained by TCP source
  - **`OfferedWindow`**
    - announced by destination in TCP header
    - flow control
    - reflects free buffer space
- Same mechanism used for flow control and for congestion control

Self-clocking or ACK Clock

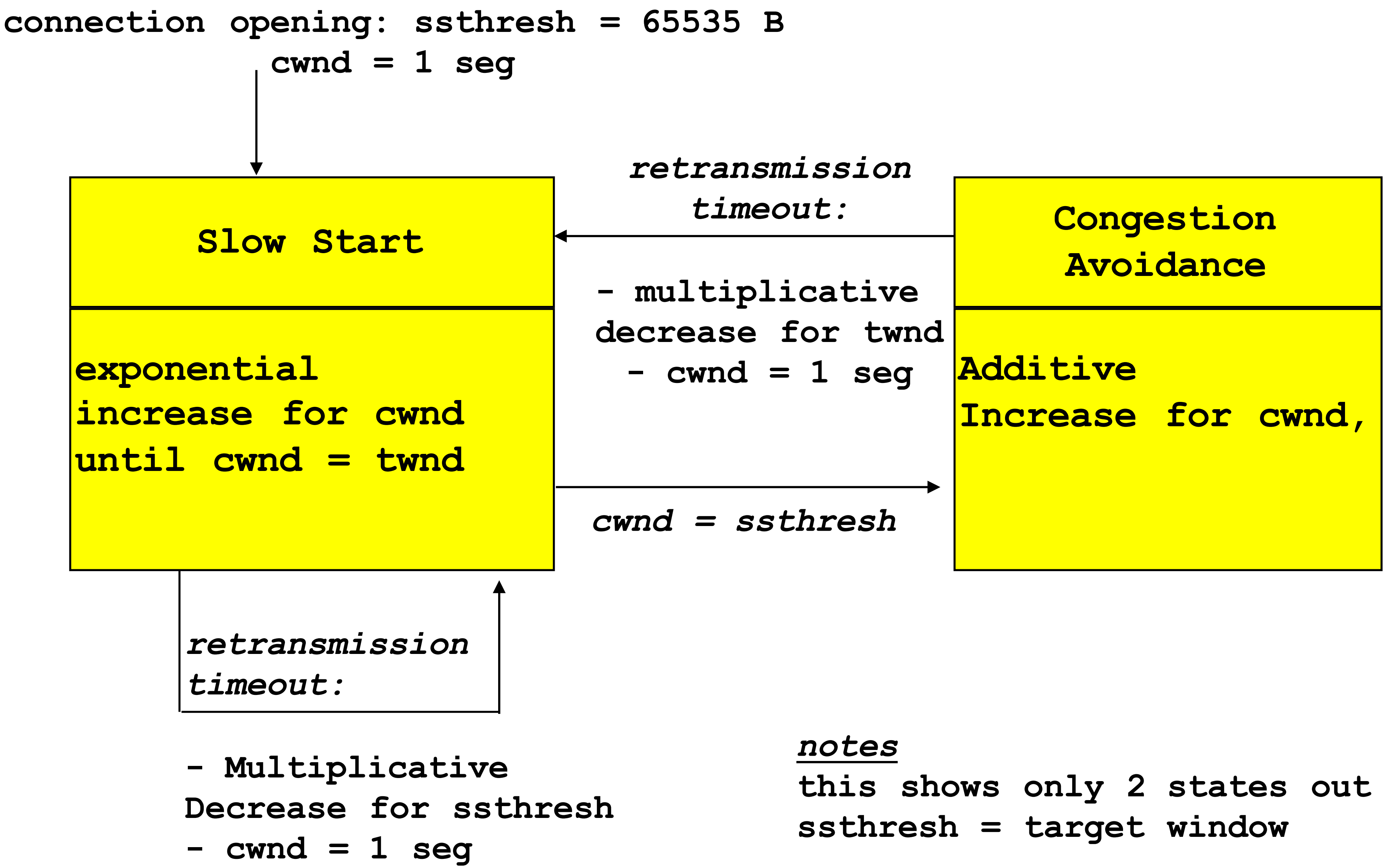


- Self-clocking systems tend to be very stable under a wide range of bandwidths and delays.
- The principal issue with self-clocking systems is getting them started.

Congestion control states

- TCP connection may be in three states with respect to congestion
  - **Slow Start** (Démarrage Lent) after loss detected by retransmission timer
  - **Fast Recovery** (Récupération Rapide) after loss detected by Fast Retransmit (three duplicated ACKs)
  - **Congestion Avoidance** (Évitement de Congestion) otherwise
- Terminology
  - *ssthresh* – target window, same as *ssthresh*
  - *flightSize* - the amount of data that has been sent but not yet acknowledged, roughly *cwnd*

Slow Start and Congestion Avoidance



notes

this shows only 2 states out of 3

ssthresh = target window

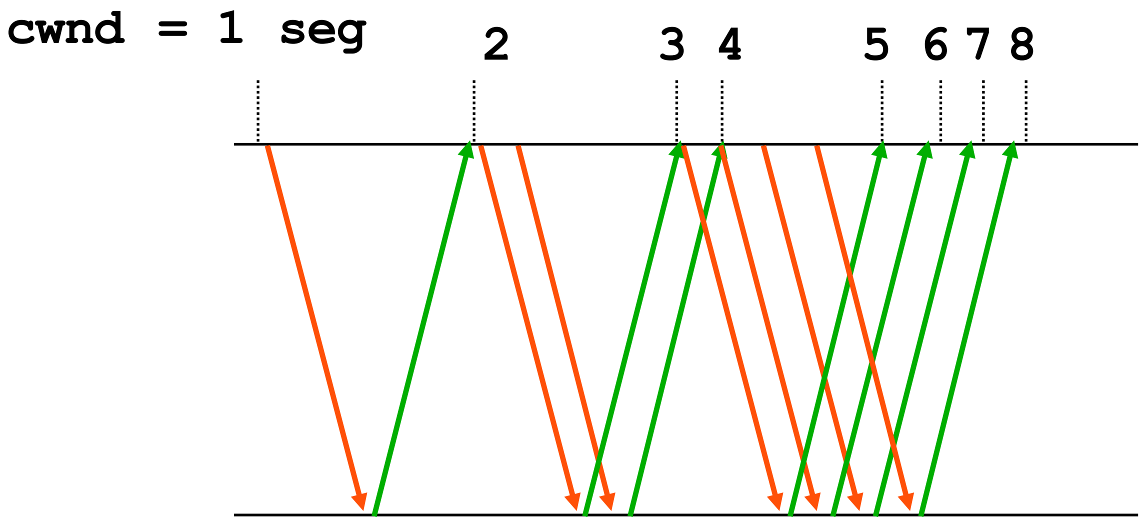
Slow Start

```
/ * exponential increase for cwnd */

  non dupl. ack received during slow start ->
          cwnd = cwnd + MSS (in bytes)
  if cwnd = ssthresh then transition to
  congestion avoidance
```

- Window increases rapidly up to the value of **`ssthresh`**
  Not so slow, rather exponential

Slow Start

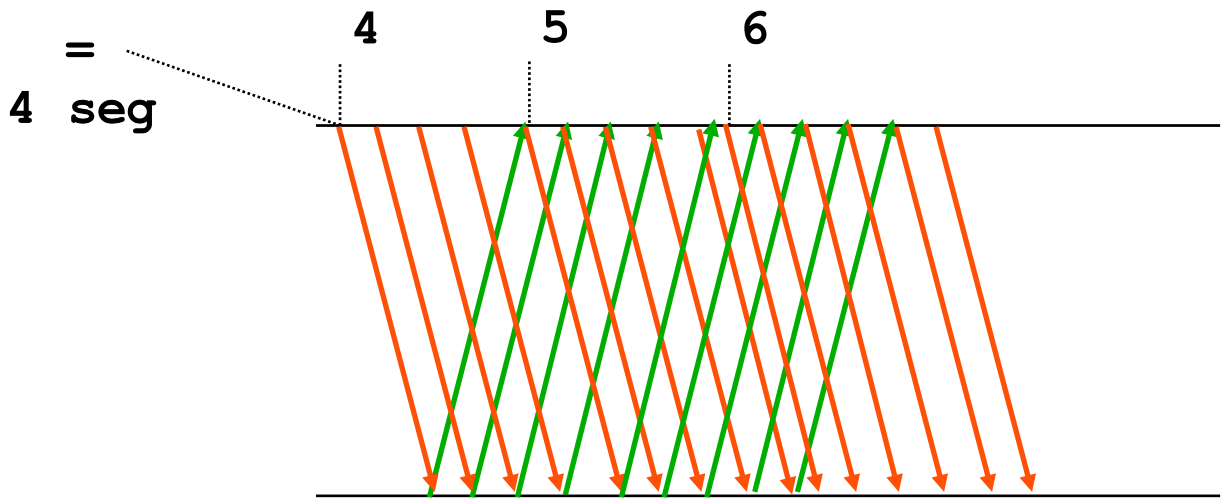


- purpose of this phase: avoid bursts of data at the beggining or after a retransmission timeout

Increase/decrease

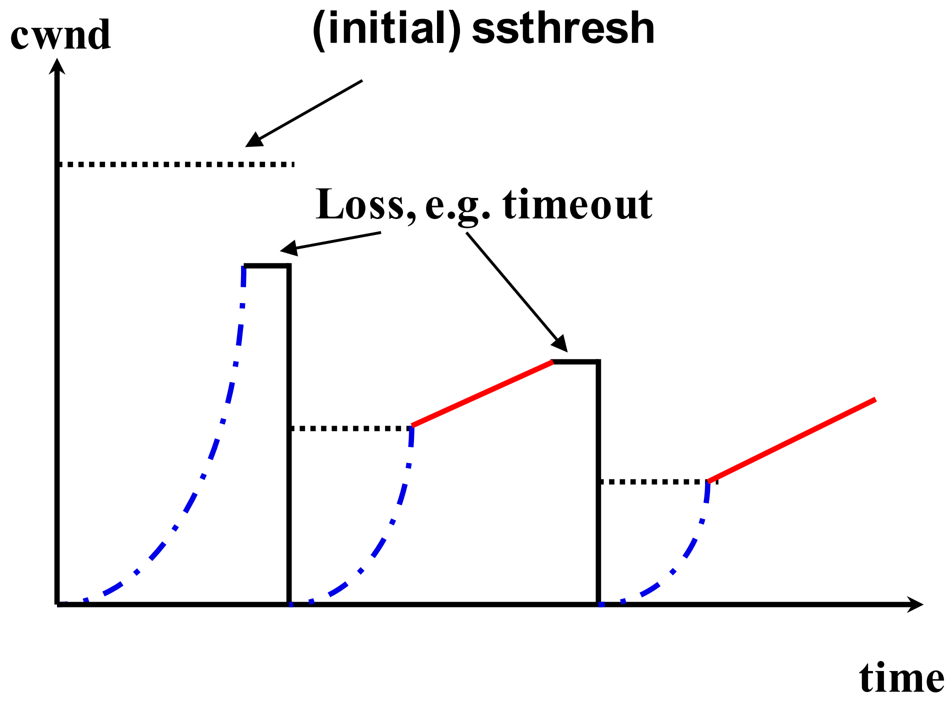
- Multiplicative decrease
  - **`ssthresh = 0.5 flightSize`**
  - **`ssthresh = max (ssthresh, 2 MSS)`**
  - **`cwnd = 1 MSS`**
- Additive increase
  - for each ACK
    - **`cwnd = cwnd + MSS×MSS / cwnd`**
    - **`cwnd = min (cwnd, max-size) (64KB)`**
  - **`cwnd`** is in bytes, counting in segments, this means that
    - we receive (**`cwnd/MSS`**) ACKs per RTT
    - for each ACK: **`cwnd/MSS`** ← 1/W
    - for a full window: W ← W + 1 MSS

cwnd Additive Increase



- during one round trip + interval between packets: increase by 1 MSS (linear increase)

Example

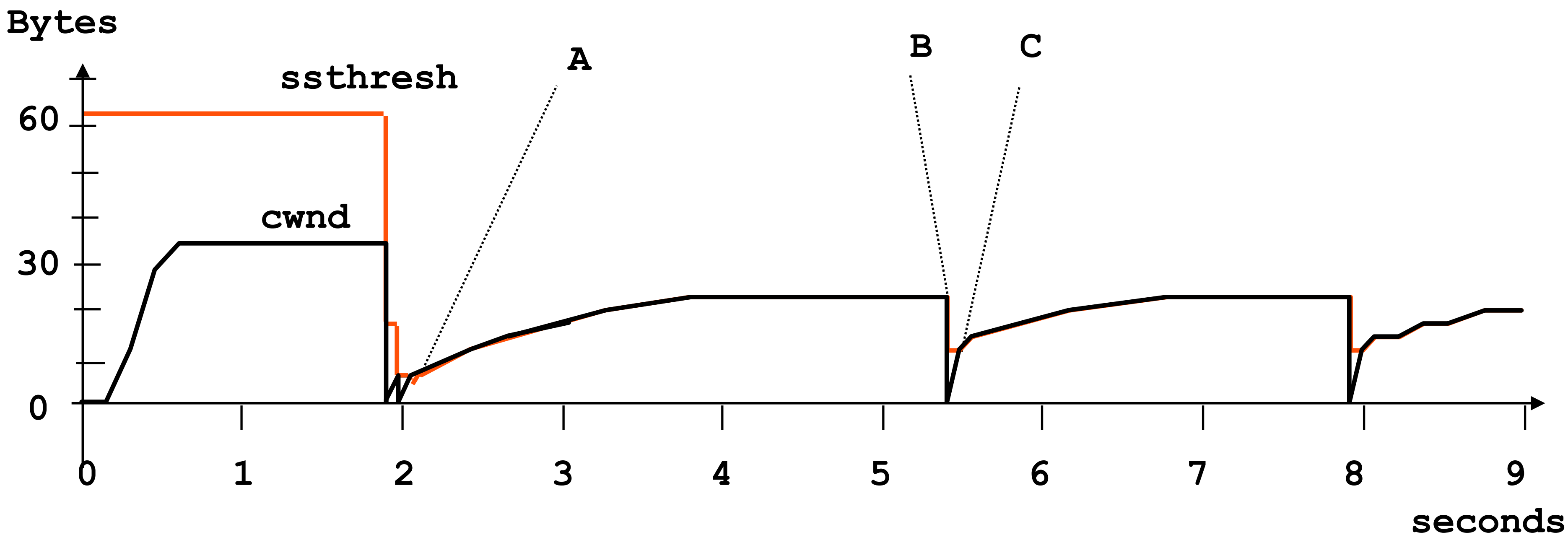


slow start – in bleu

congestion avoidance – in red

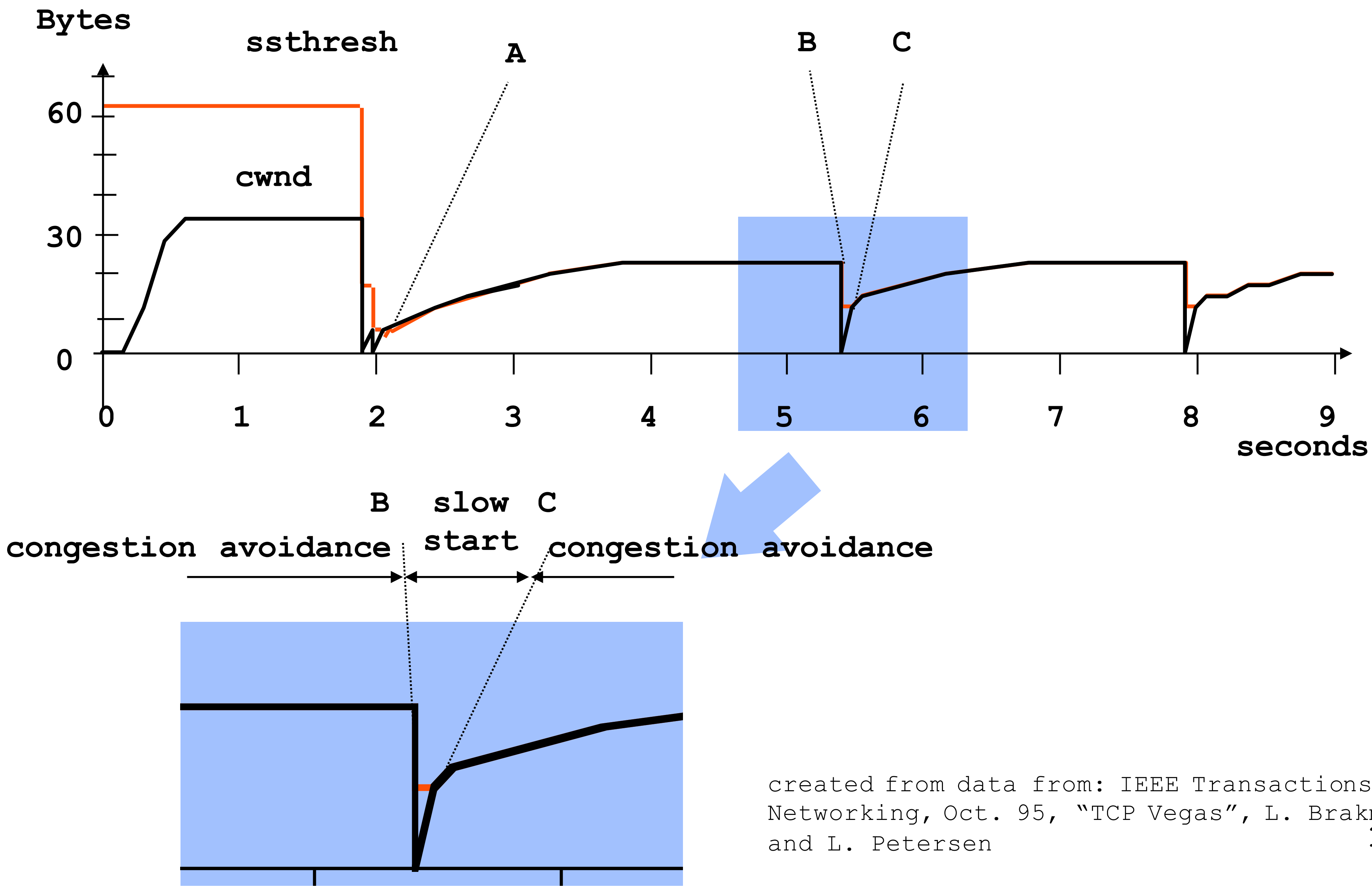
`flightSize = cwnd`

Example



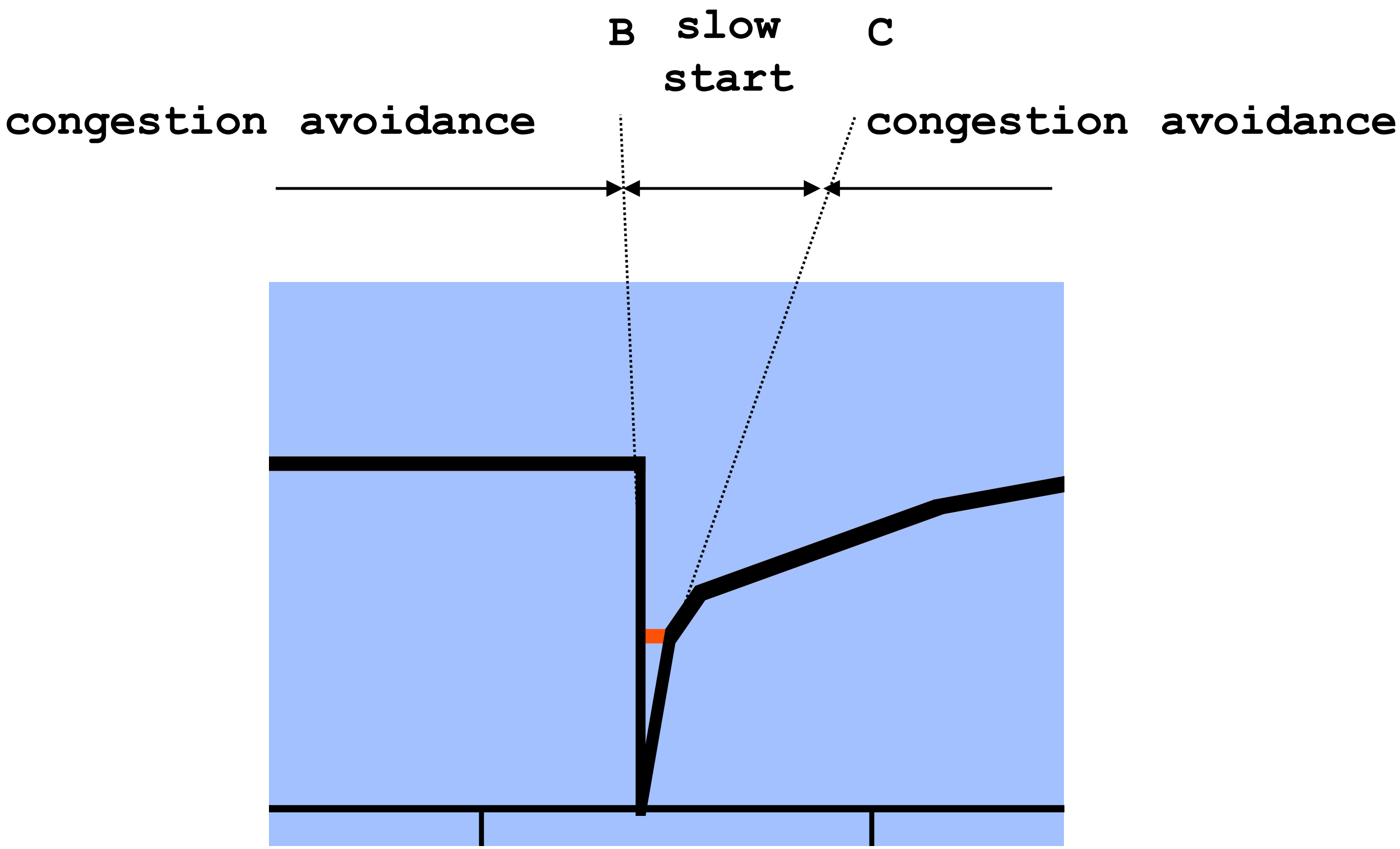
created from data from: IEEE Transactions on Networking, Oct. 95, "TCP Vegas", L. Brakmo and L. Petersen

Example



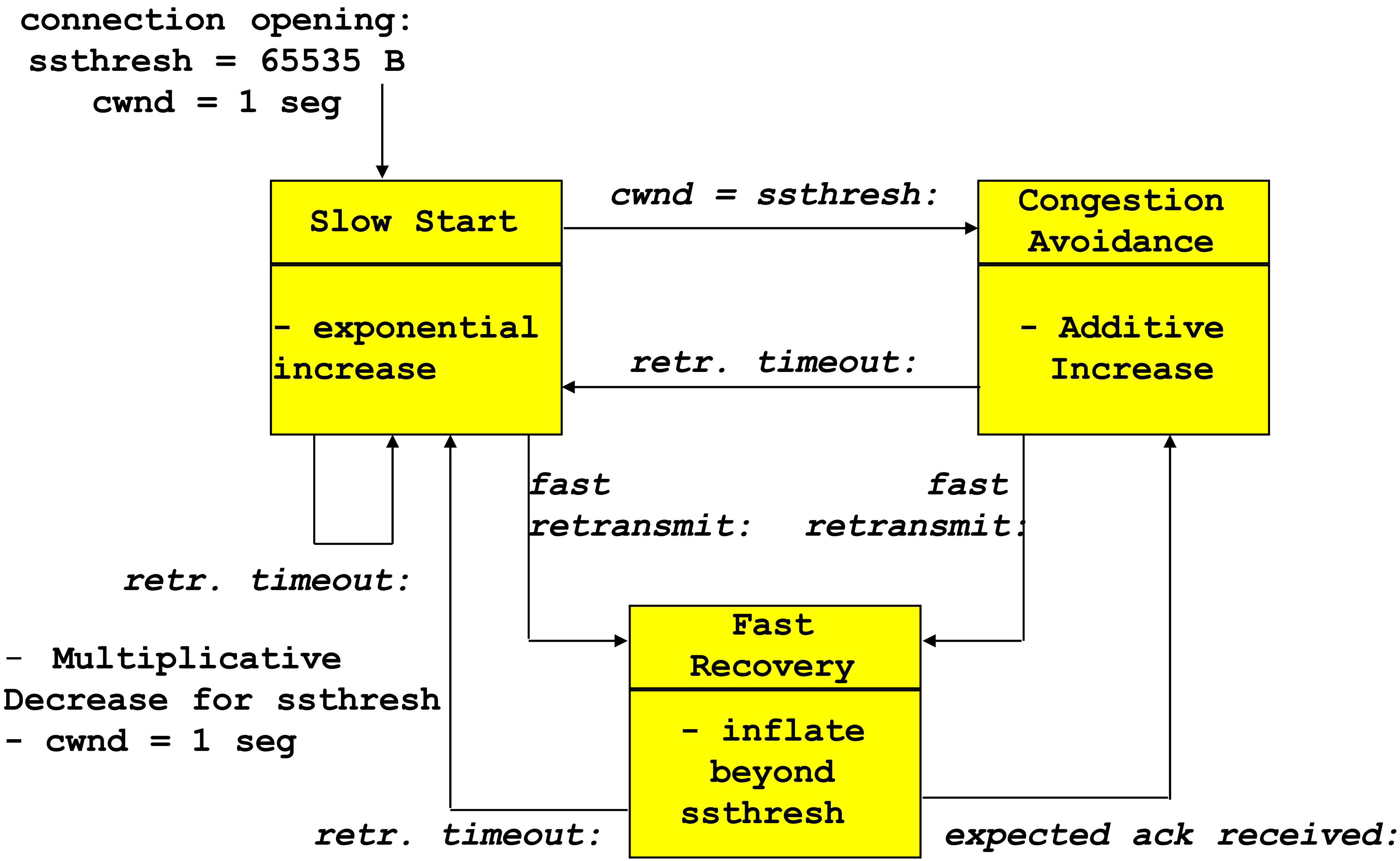
created from data from: IEEE Transactions on Networking, Oct. 95, "TCP Vegas", L. Brakmo and L. Petersen

Slow Start and Congestion Avoidance

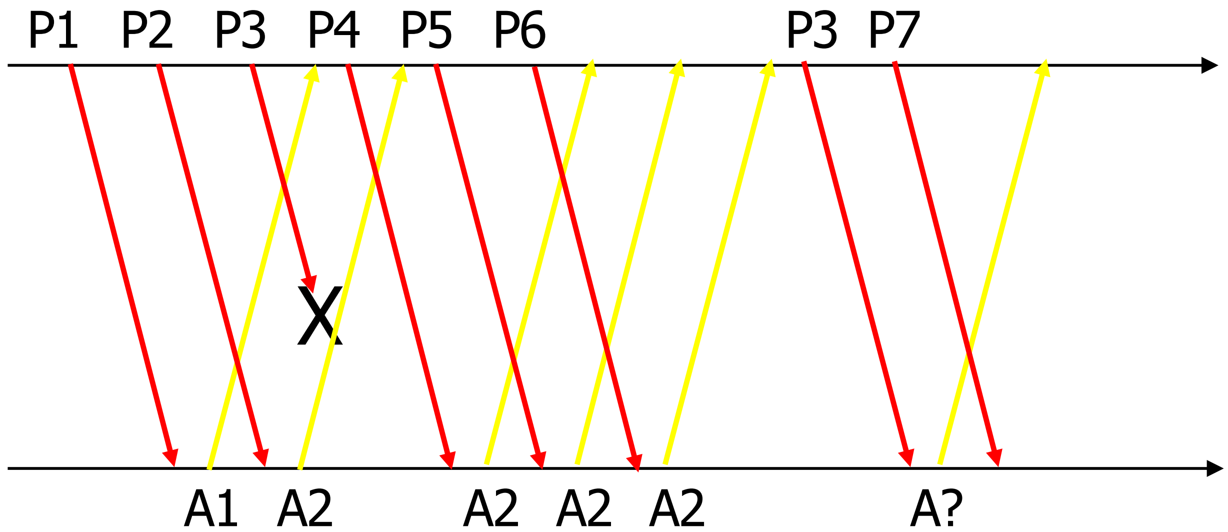


created from data from: IEEE Transactions on Networking, Oct. 95, "TCP Vegas", L. Brakmo and L. Petersen

Congestion Control States



Fast Retransmit



- Fast Retransmit
  - retransmit timer can be large
  - optimize retransmissions similarly to Selective Retransmit
  - if sender receives 3 duplicated ACKs, retransmit missing segment

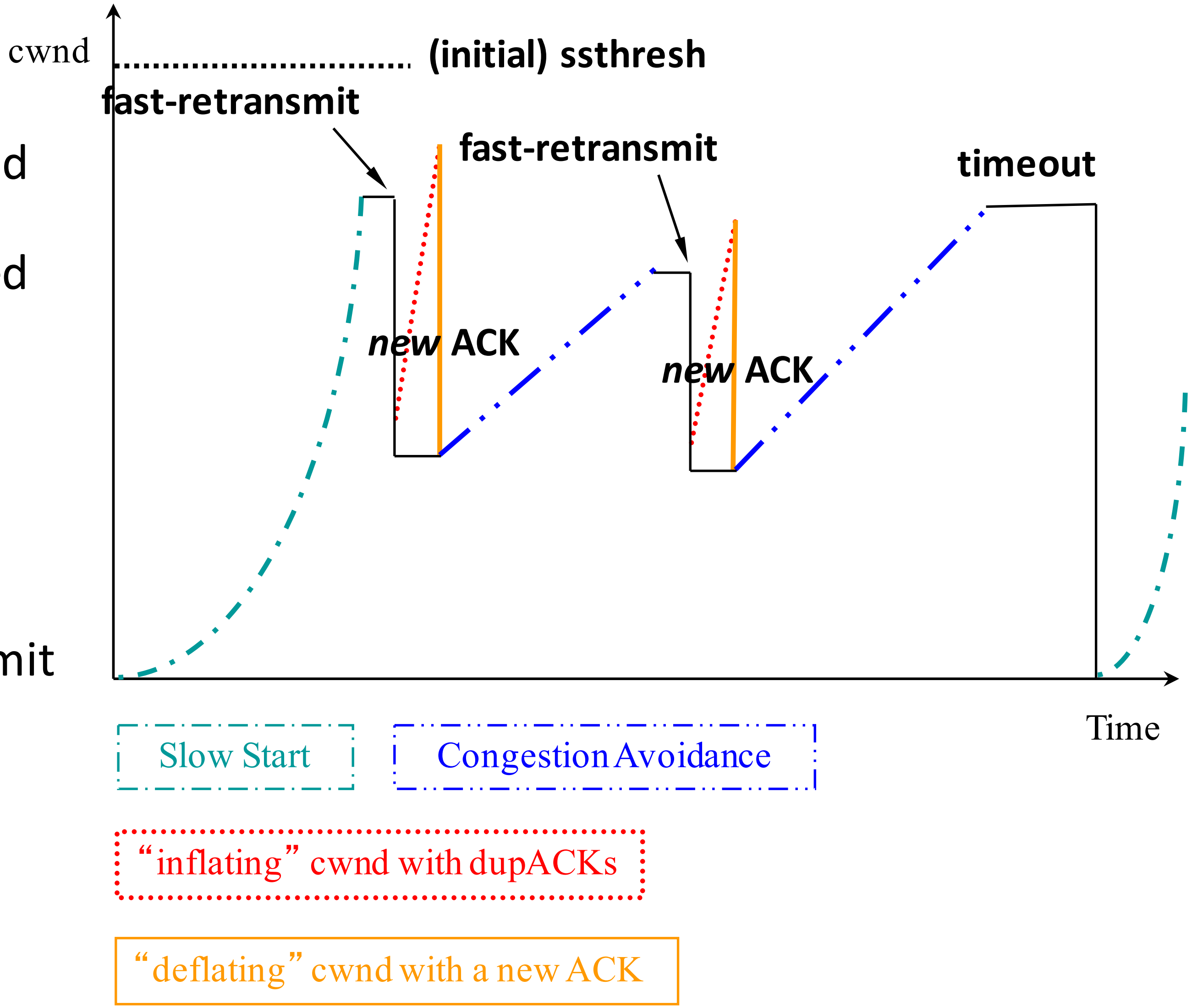
Fast Recovery

Concept:

- After fast retransmit, reduce cwnd by half, and continue sending segments at this reduced level.

Problems:

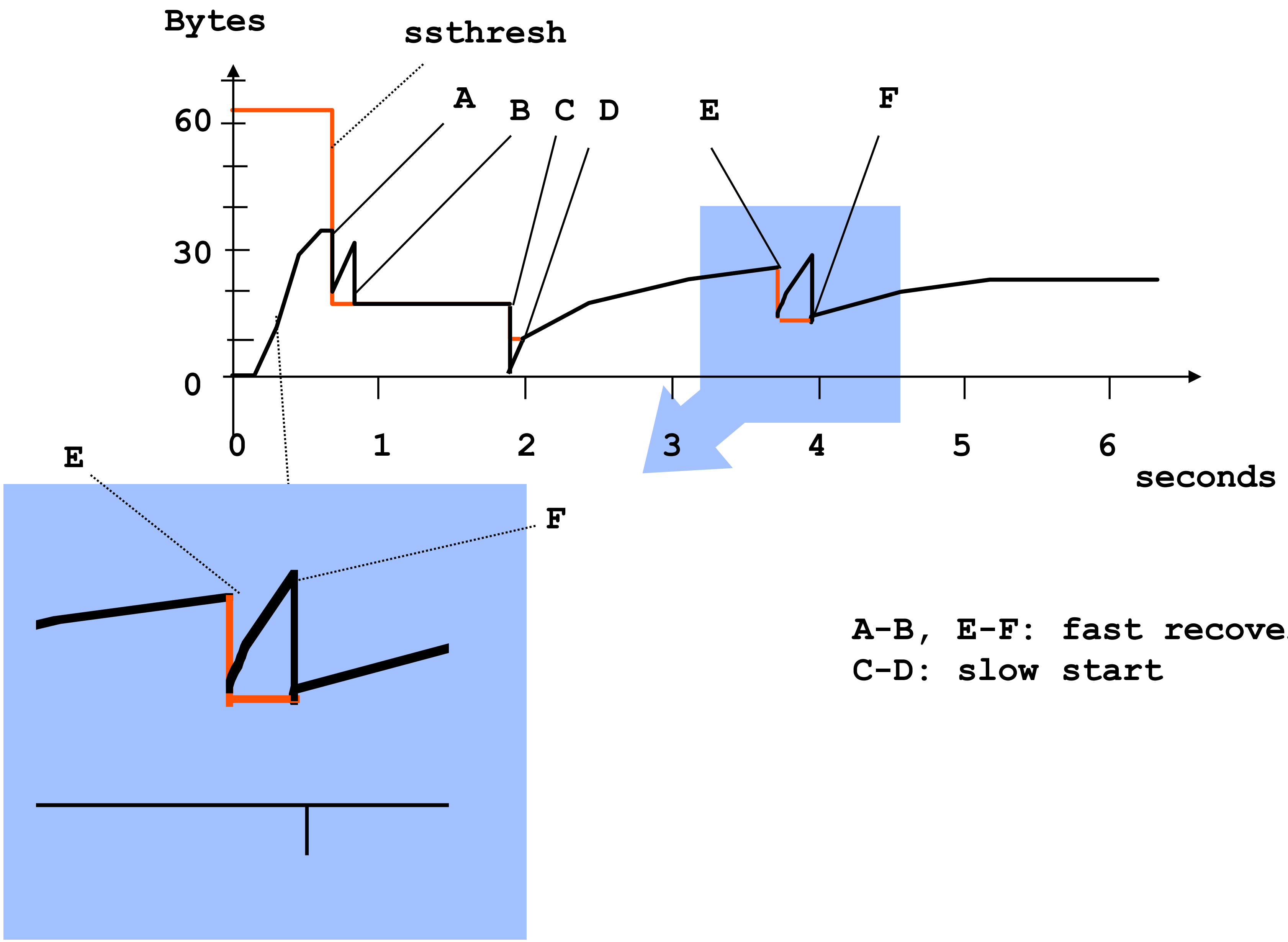
- Sender has too many outstanding segments.
- How does sender transmit packets on a dupACK? Need to use a “trick” - inflate cwnd.



Fast Recovery

- Multiplicative decrease
  - `ssthresh = 0.5 flightSize`
  - `ssthresh = max (ssthresh, 2 MSS)`
- Fast Recovery
  - `cwnd = ssthresh + 3 MSS (inflate)`
  - `cwnd = min (cwnd, 64K)`
  - `retransmit the missing segment (n)`
- For each duplicated ACK
  - `cwnd = cwnd + MSS (keep inflating)`
  - `cwnd = min (cwnd, 64K)`
  - `keep sending segments in the current window`
- For partial ACK
  - `retransmit the first unACKed segment`
  - `cwnd = cwnd – ACKed + MSS (deflate/inflate)`

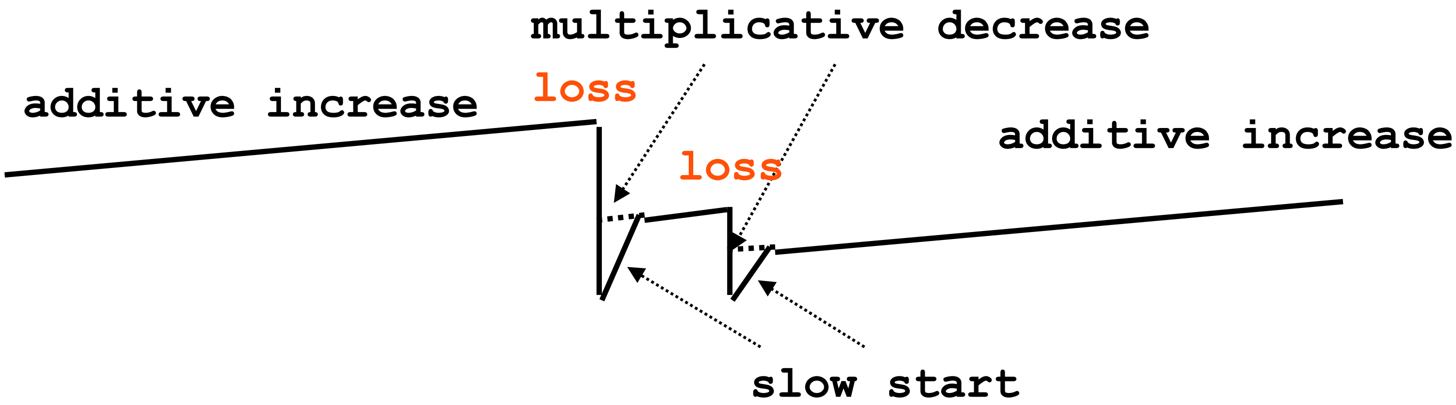
Fast Recovery Example



A-B, E-F: fast recovery
C-D: slow start

TCP Congestion Control

- TCP performs congestion control in end-systems
- Principle
  - sender increases its sending window until loss occurs, then decreases
- Target window
  - additive increase (no loss)
  - multiplicative decrease (loss)



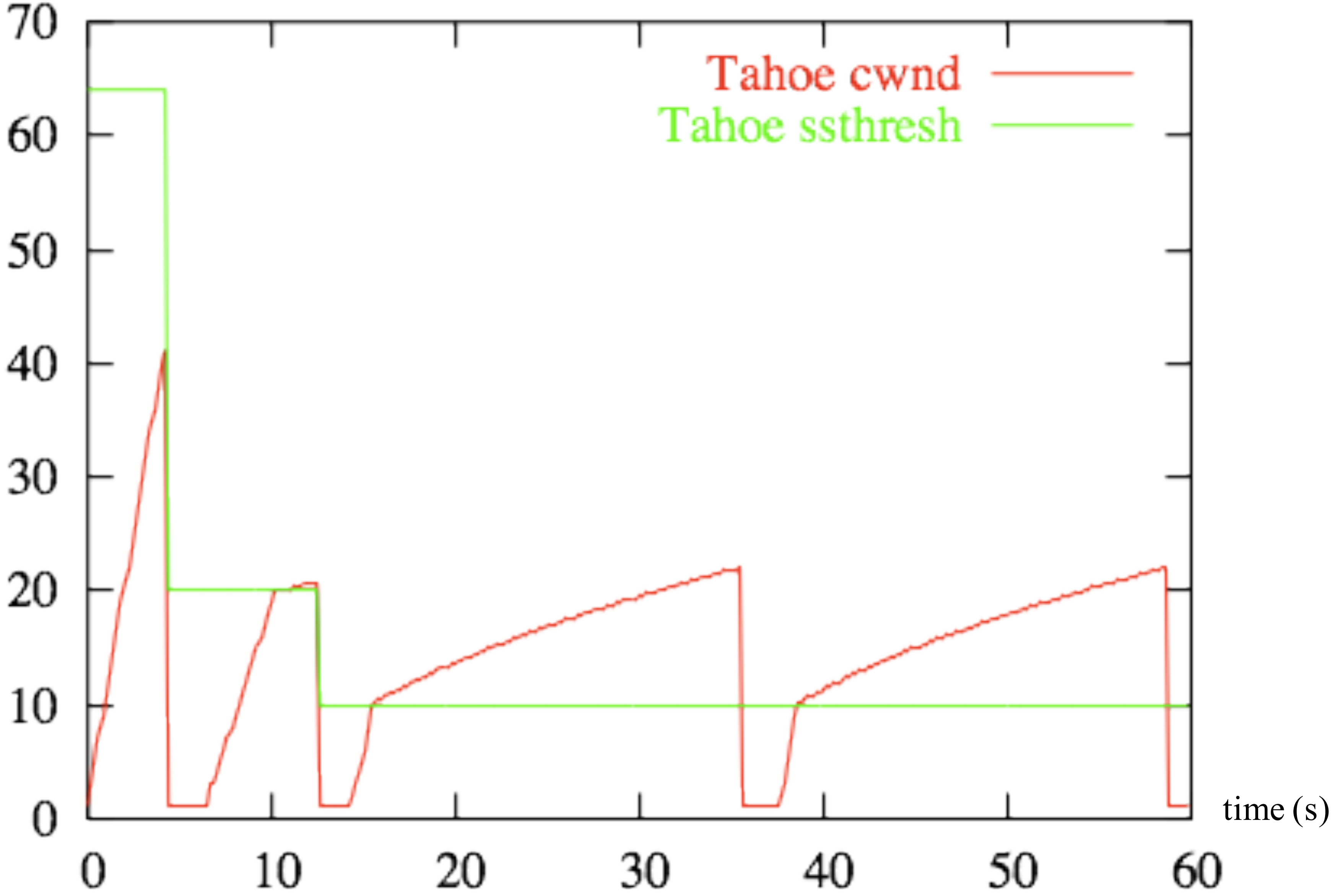
TCP Congestion Control

- 3 phases
  - **slow start**
    - starts with 1, exponential increase up to **`twnd`**
  - **congestion avoidance**
    - additive increase until loss or max window
  - **fast recovery**
    - fast retransmission of one segment
- Slow start entered at setup or after retransmission timeout
- Fast recovery entered at fast retransmit
- Congestion avoidance entered when **`cwnd = ssthresh`**

Summary of TCP Behavior

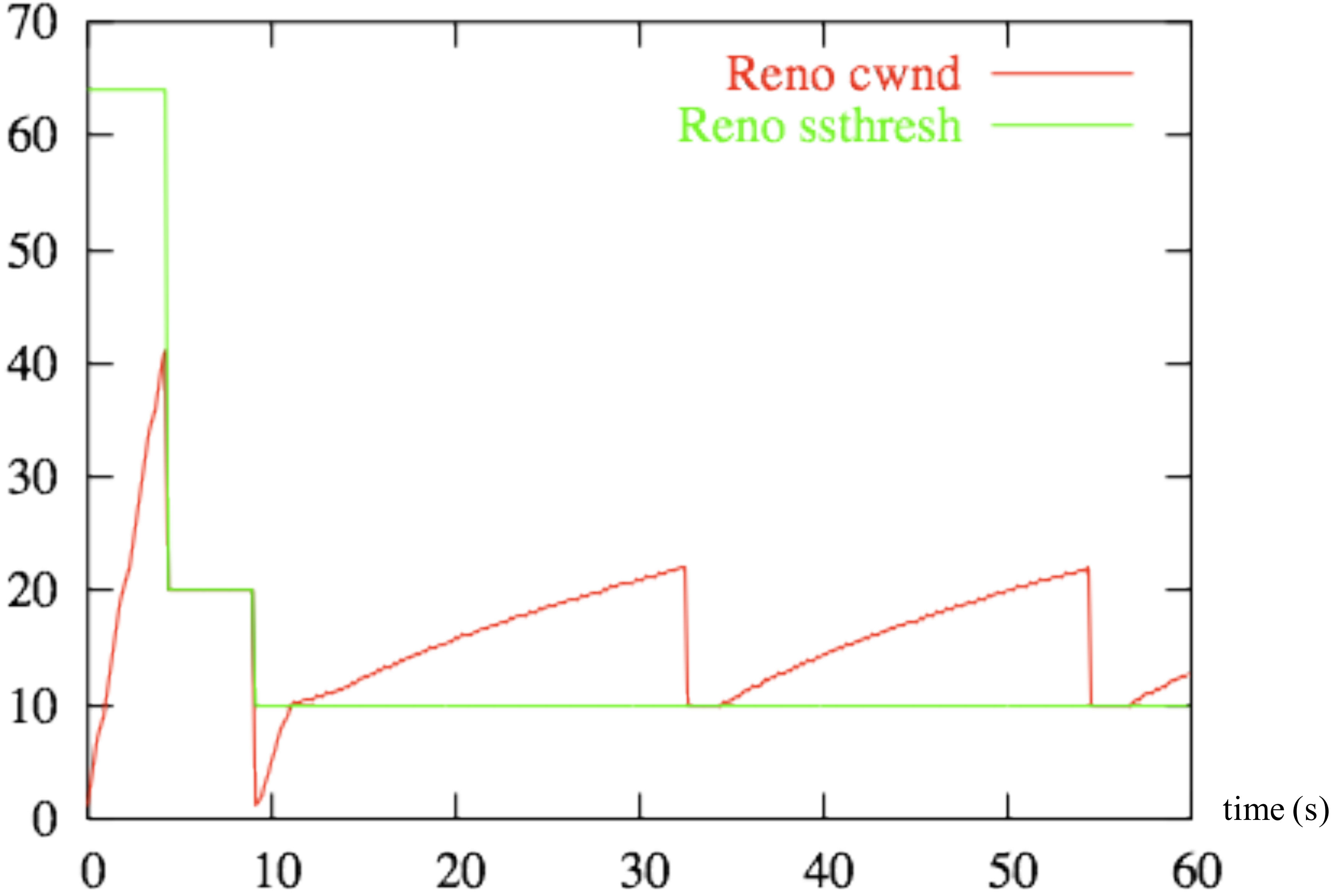
TCP Variation	Response to 3 dupACKs	Response to Partial ACK of Fast Retransmission	Response to “full” ACK of Fast Retransmission
Tahoe	Do fast retransmit, enter slow start	++cwnd	++cwnd
Reno	Do fast retransmit, enter fast recovery	Exit fast recovery, deflate window, enter congestion avoidance	Exit fast recovery, deflate window, enter congestion avoidance
NewReno	Do fast retransmit, enter modified fast recovery	Fast retransmit and deflate window – remain in modified fast recovery	Exit modified fast recovery, deflate window, enter congestion avoidance

TCP Tahoe



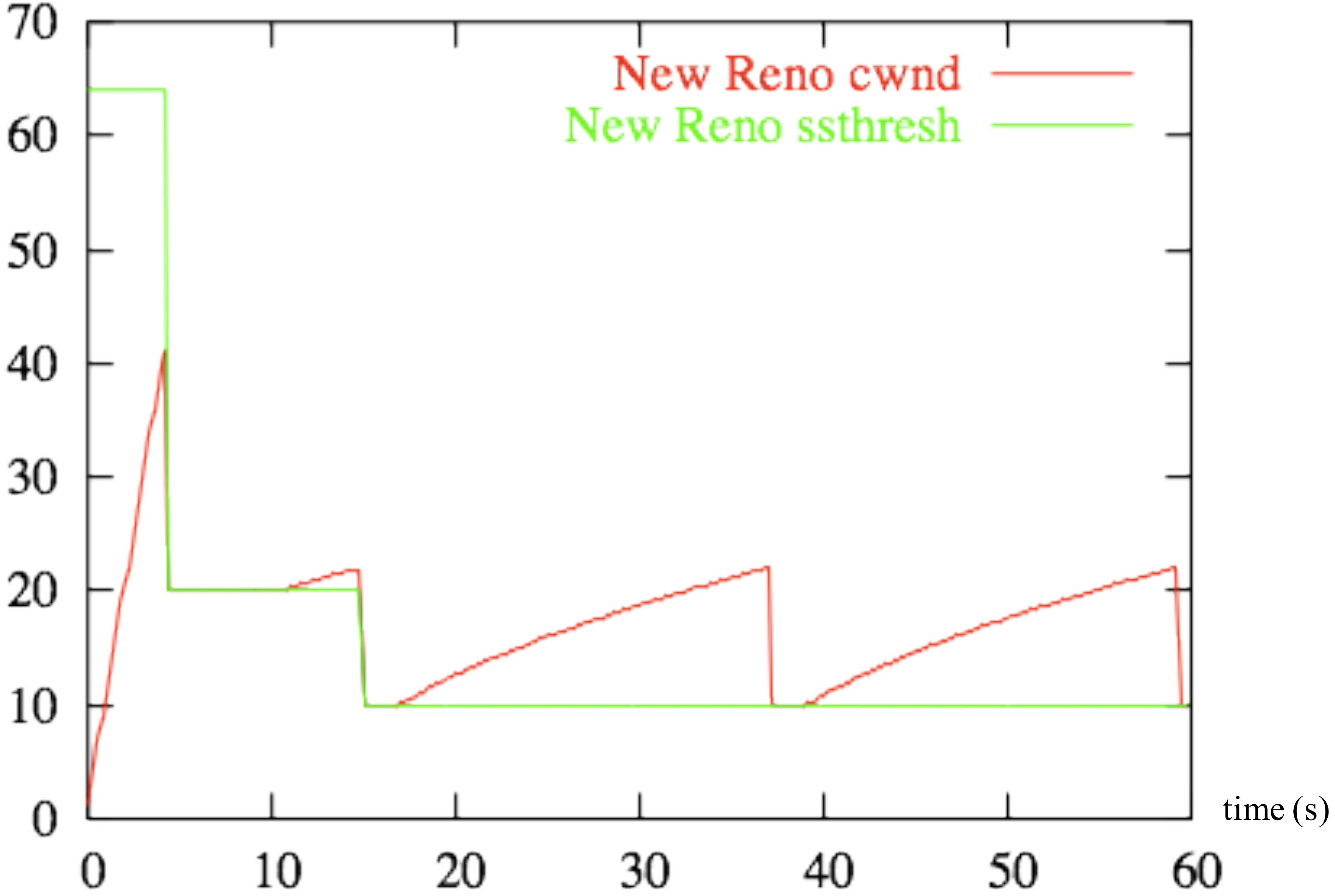
Tahoe cwnd
Tahoe ssthresh
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
time (s)

TCP Reno



Reno cwnd
Reno ssthresh
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
time (s)

TCP New Reno



New Reno cwnd
New Reno ssthresh
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
time (s)

TCP Loss - Throughput formulae

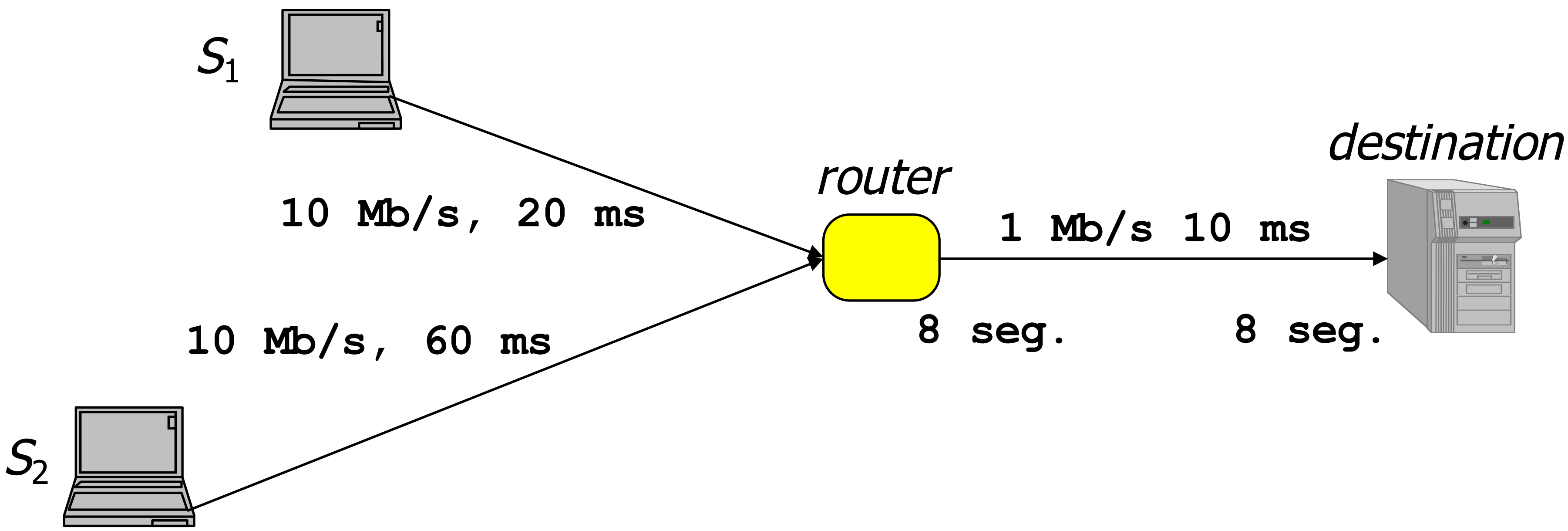
$$\theta = \frac{L}{T}\frac{C}{\sqrt{q}}$$

- TCP connection with
  - RTT *T*
  - segment size *L*
  - average packet loss ratio *q*
  - constant *C* = 1.22
- Transmission time negligible compared to RTT, losses are rare, time spent in Slow Start and Fast Recovery negligible

Fairness of the TCP

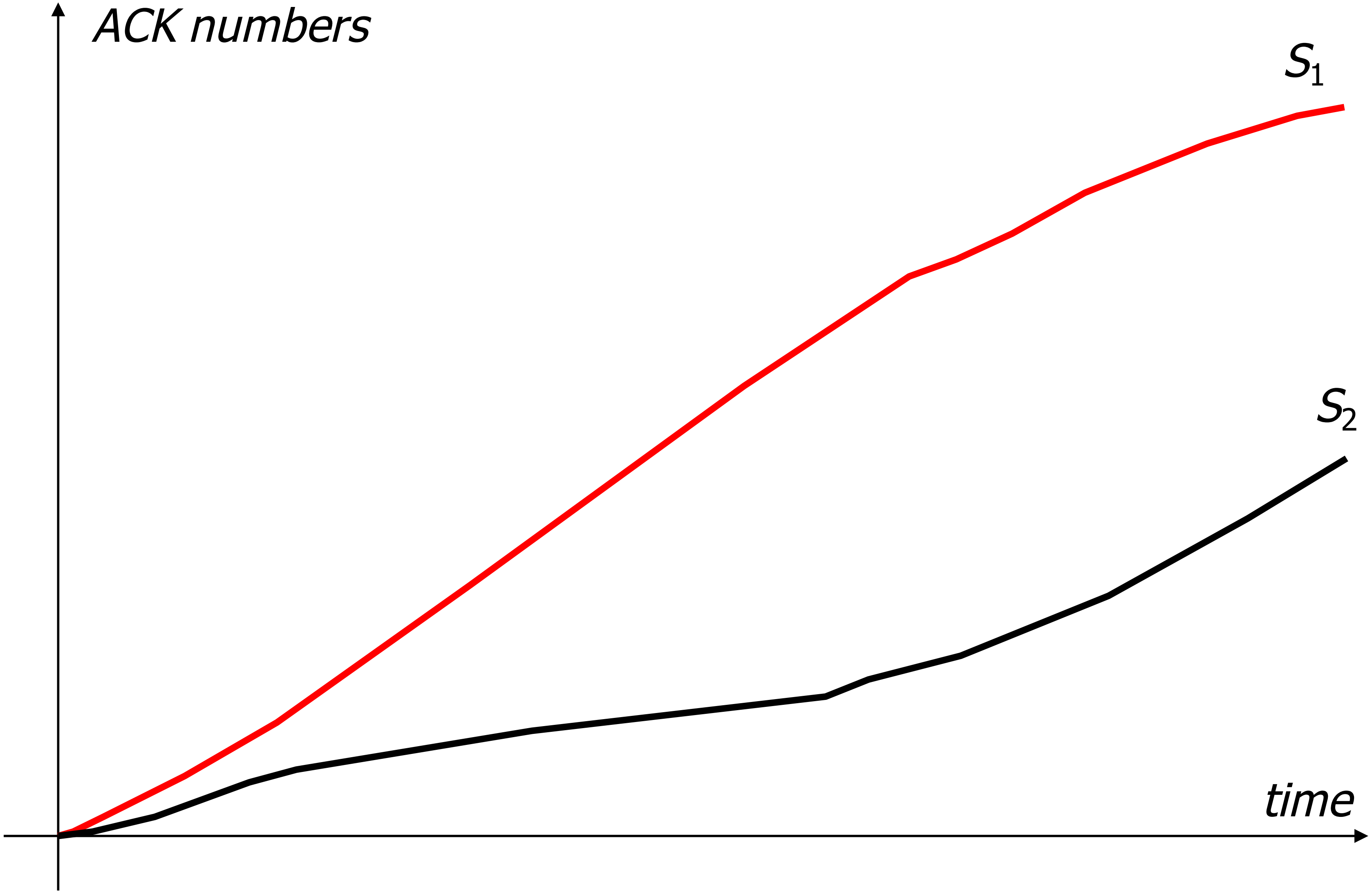
- TCP differs from the pure AI-MD principle
  - window based control, not rate based
  - increase in rate is not strictly additive - window is increased by 1/W for each ACK
- Like with **proportional fairness**, the adaptation algorithm gives less to sources using many resources
  - not the number of links, but RTT
- TCP fairness: negative bias of long round trip times

Fairness of the TCP



- Example network with two TCP sources
  - link capacity, delay
  - limited queues on the link (8 segments)
- NS simulation

Throughput in time



TCP Friendly Applications

- All TCP/IP applications that generate long lived flows should mimics the behavior of a TCP source
  - RTP/UDP flow sending video/audio data
- Adaptive algorithm
  - application determines the sending rate
  - feedback - amount of lost packets, loss ratio
  - sending rate = rate of a TCP flow experiencing the same loss ratio

Facts to remember

- TCP performs congestion control in end-systems
  - sender increases its sending window until loss occurs, then decreases
    - additive increase (no loss)
    - multiplicative decrease (loss)
- TCP states
  - slow start, congestion avoidance, fast recovery
- Negative bias towards long round trip times
- UDP applications should behave like TCP with the same loss rate