


Computer Networks

CMSC 417 : Spring 2024

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

Topic: TCP Congestion-control, TCP history, Router-assisted congestion control, Link Layer (Textbook chapter 6 & 2)

Nirupam Roy

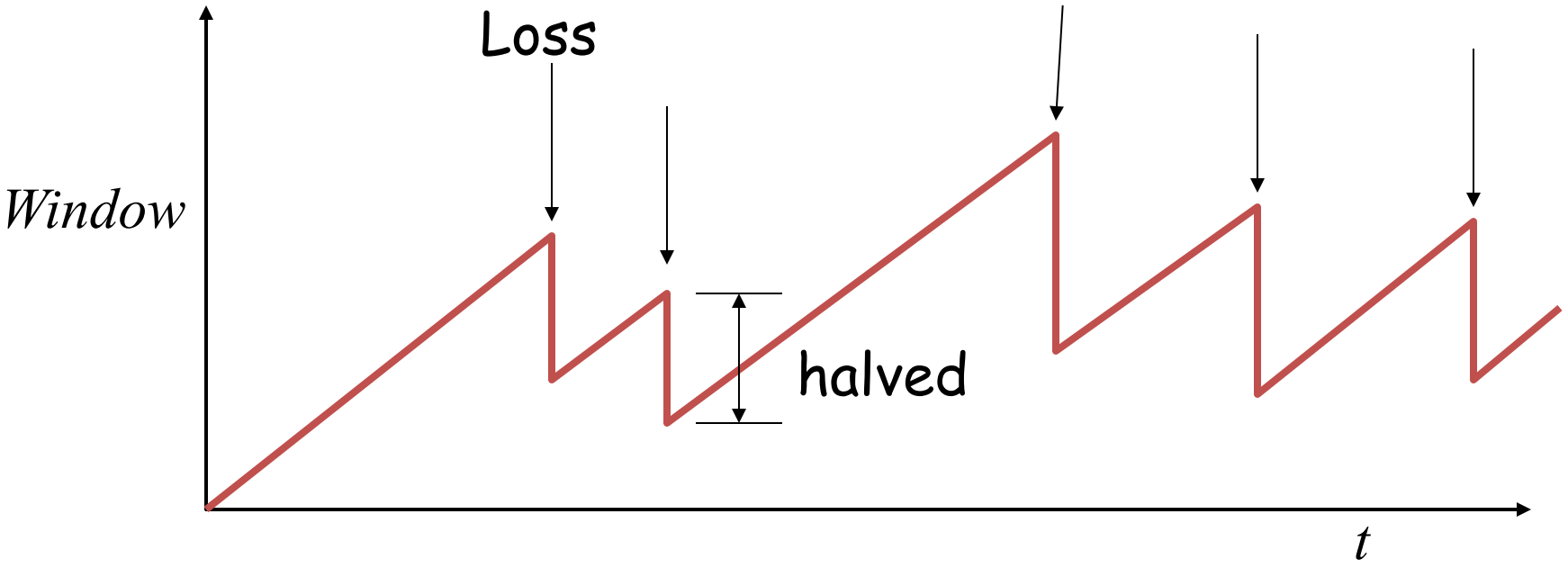
Tu-Th 2:00-3:15pm
CSI 2117

March 28th, 2024




TCP Congestion Control

- Additive increase, multiplicative decrease (AIMD)
  - On packet loss, divide congestion window in half
  - On success for last window, increase window linearly



Why Exponential decrease?

- Respond aggressively to bad news
  - Congestion is (very) bad for everyone
  - Need to react aggressively
- Nice theoretical properties
  - Makes efficient use of network resources

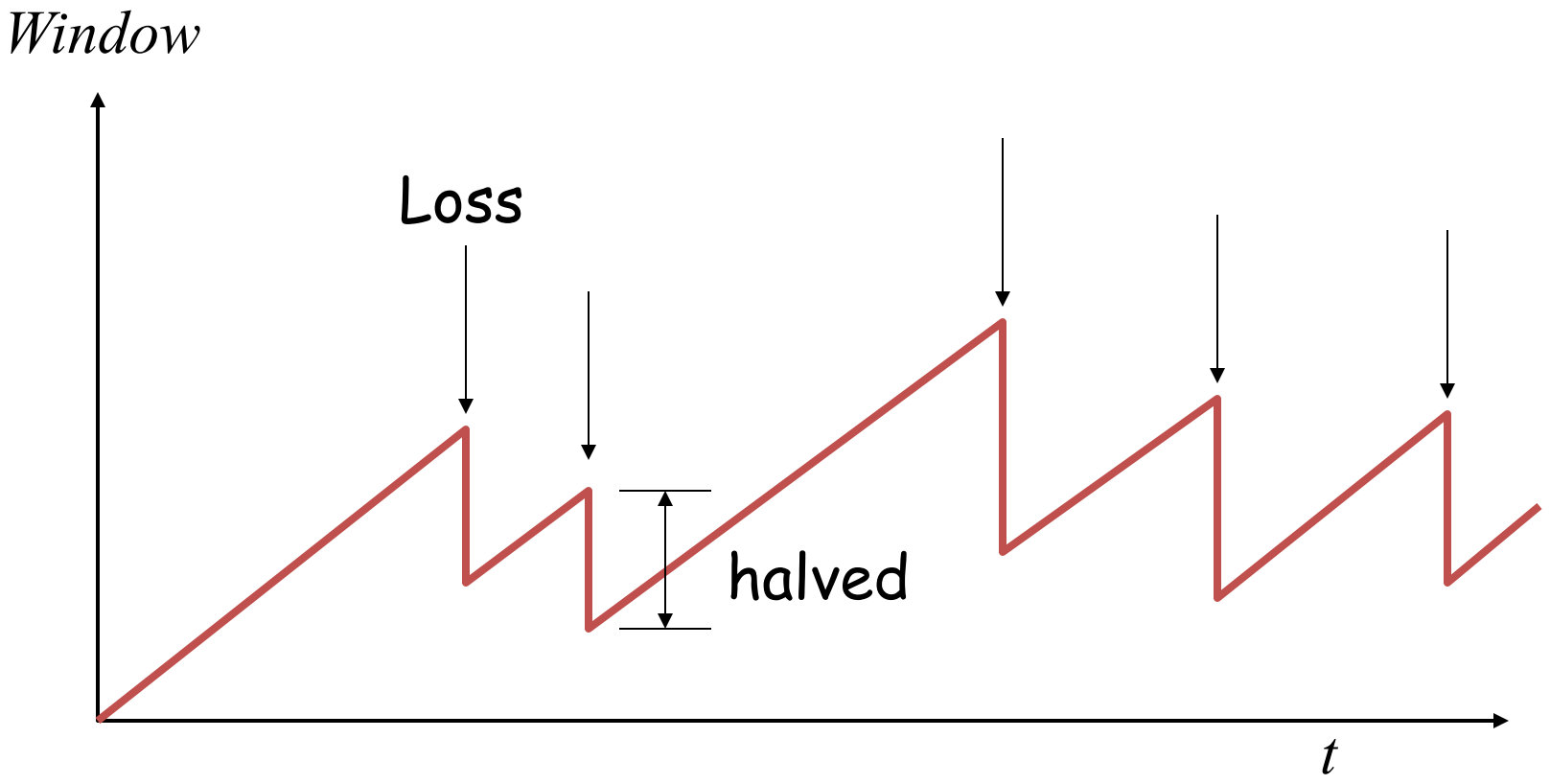
TCP Congestion Window

- Each TCP sender maintains a congestion window
  - Max number of bytes to have in transit (not yet ACK'd)
- Adapting the congestion window
  - Decrease upon losing a packet: backing off
  - Increase upon success: optimistically exploring
  - Always struggling to find right transfer rate
- Tradeoff
  - Pro: avoids needing explicit network feedback
  - Con: continually under- and over-shoots "right" rate

Additive Increase, Multiplicative Decrease

- How much to adapt?
  - Additive increase: On success of last window of data, increase window by 1 Max Segment Size (MSS)
  - Multiplicative decrease: On loss of packet, divide congestion window in half
- Much quicker to slow than speed up!
  - Over-sized windows (causing loss) are much worse than under-sized windows (causing lower thruput)
  - AIMD: A necessary condition for stability of TCP

Leads to the TCP "Sawtooth"



Receiver/Advertised Window vs. Congestion Window

- Flow control
  - Keep a *fast sender* from overwhelming *a slow receiver*
- Congestion control
  - Keep a *set of senders* from overloading the *network*

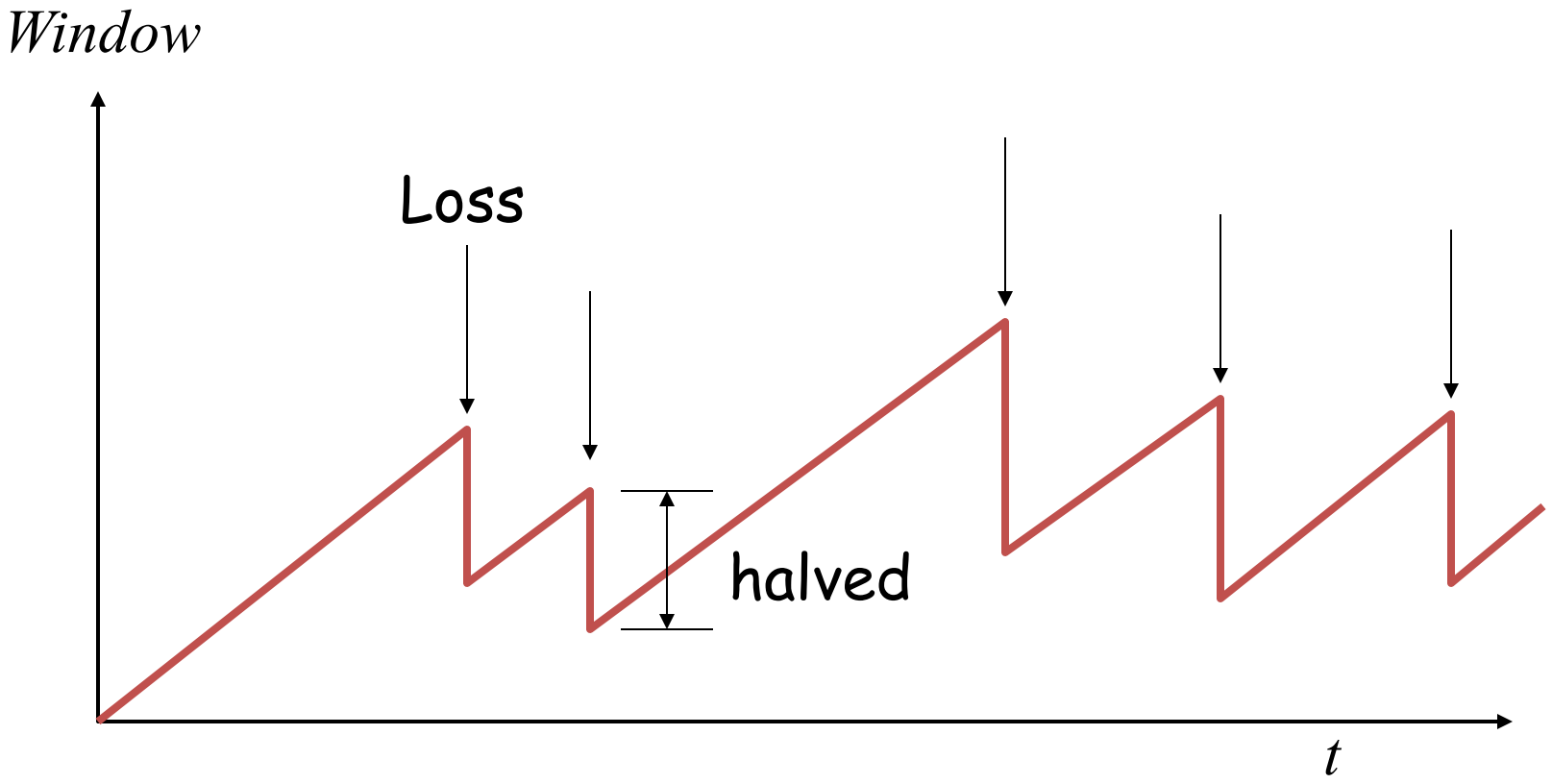
- Different concepts, but similar mechanisms
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - Sender TCP window =

    min { congestion window, receiver window }

Slow start

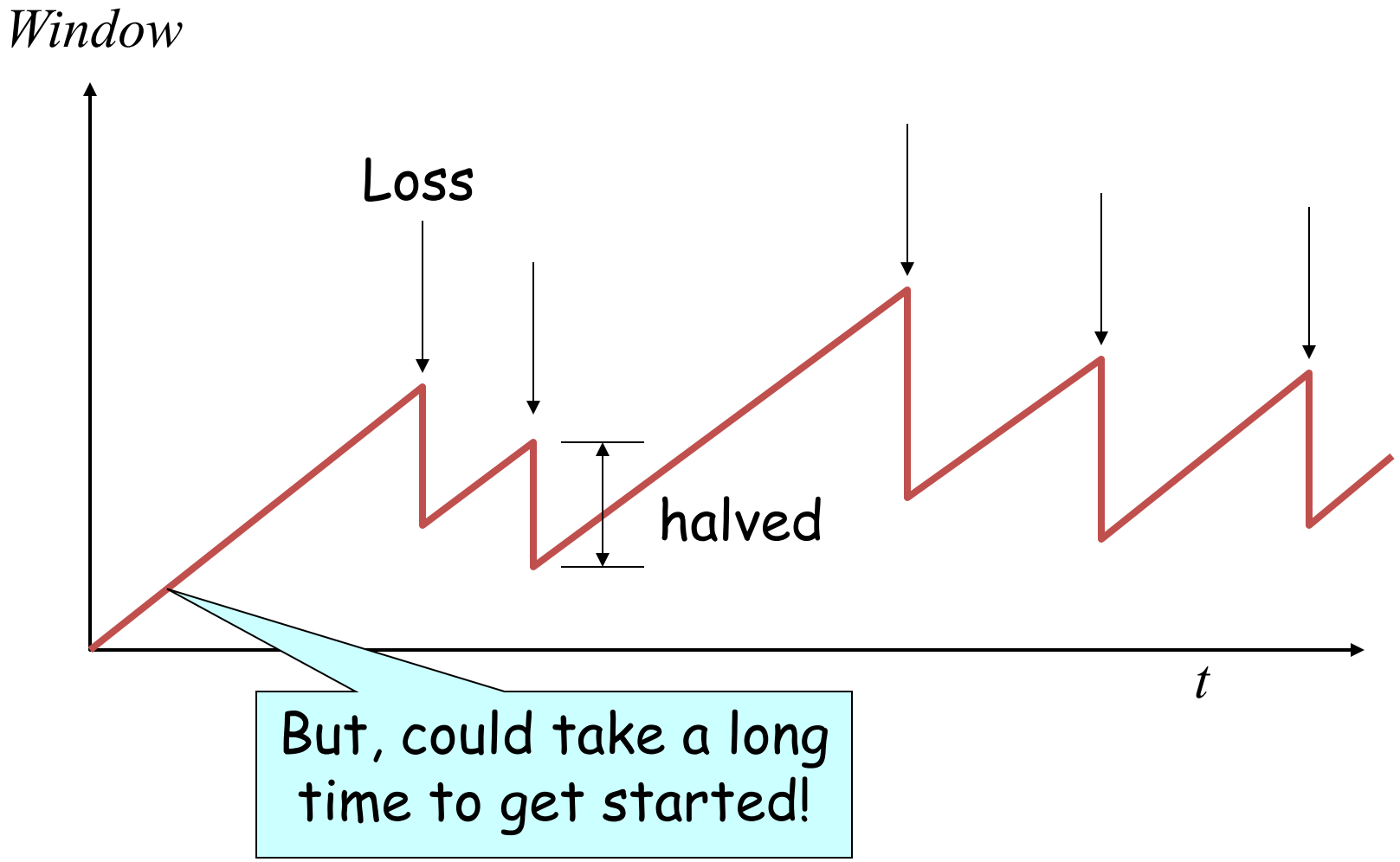
How Should a New Flow Start?

Start slow (a small CWND) to avoid overloading network



How Should a New Flow Start?

Start slow (a small CWND) to avoid overloading network

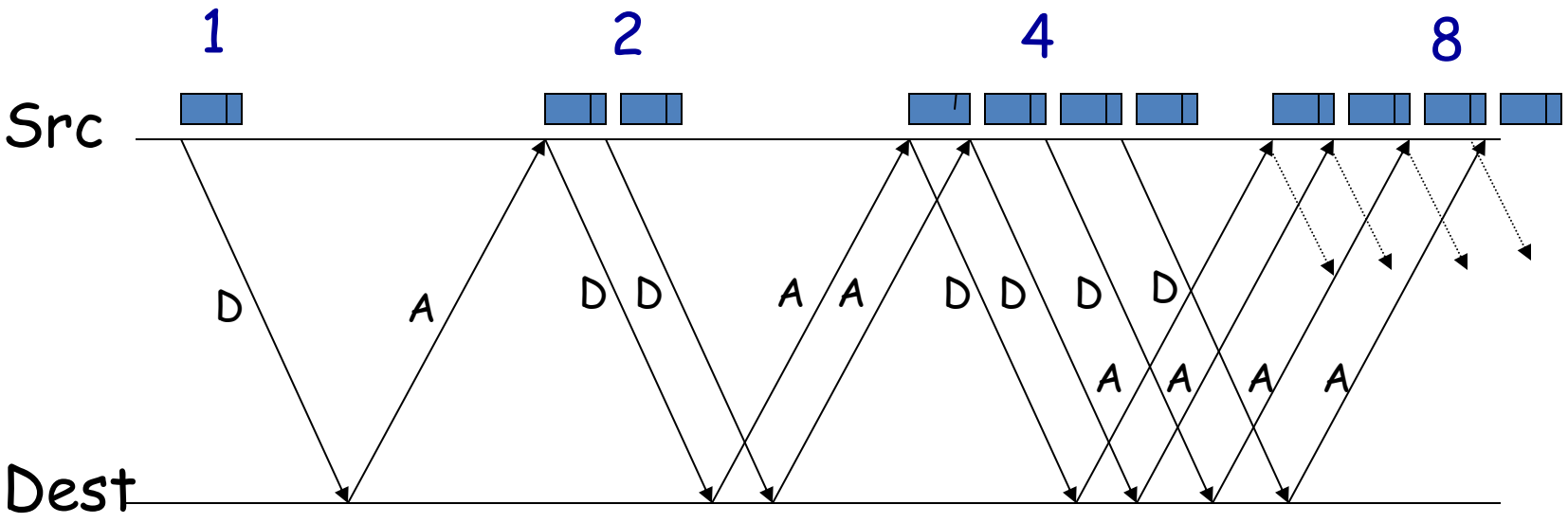


"Slow Start" Phase

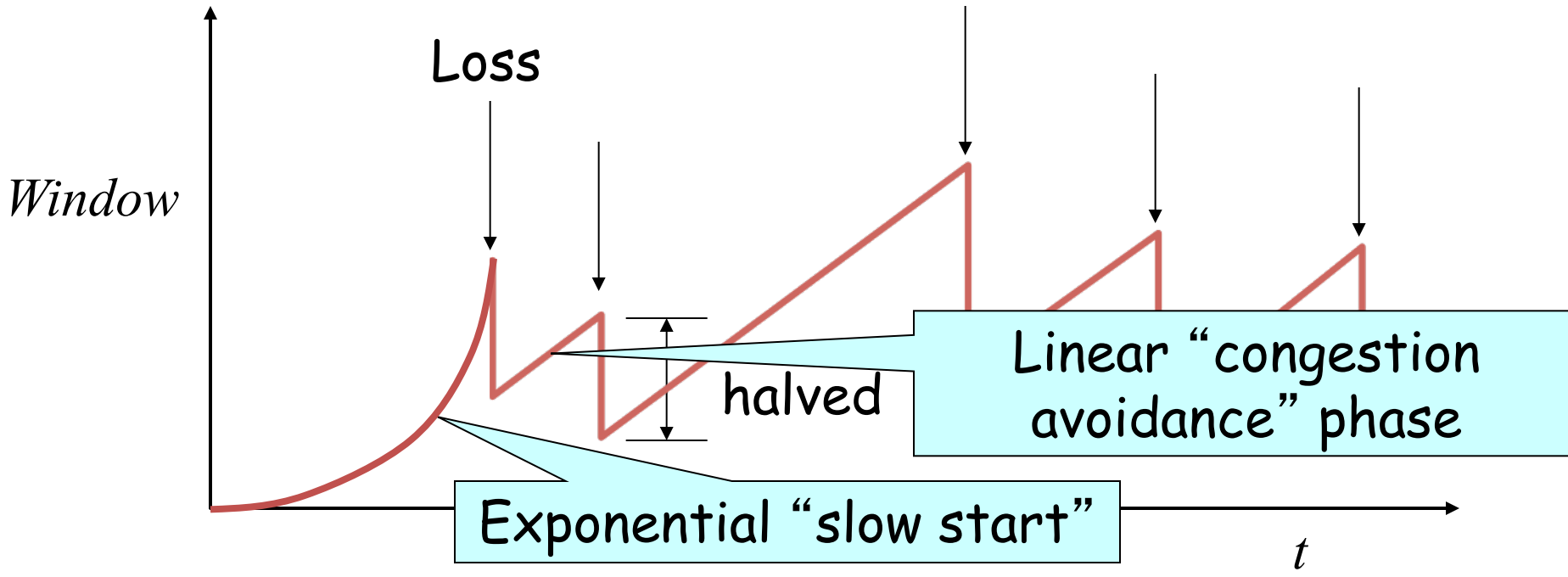
- Start with a small congestion window
  - Initially, CWND is 1 MSS
  - So, initial sending rate is MSS / RTT
- Could be pretty wasteful
  - Might be much less than actual bandwidth
  - Linear increase takes a long time to accelerate
- Slow-start phase (really "fast start")
  - Sender starts at a slow rate (hence the name)
  - … but increases rate exponentially until the first loss

Slow Start in Action

Double CWND per round-trip time



Slow Start and the TCP Sawtooth



- Why is this called "slow start"?

TCP originally had *no* congestion control

- Source would start by sending entire receiver window
- Led to congestion collapse!
- "Slow start" is, comparatively, slower

Response to the two types of losses

Response to the two types of losses

(1) Triple duplicate ACK (Fast retransmit)

- Packet n is lost, but packets n+1, n+2, etc. arrive
- Then, sender quickly resends packet n
- Do a multiplicative decrease and keep going

Fast retransmit

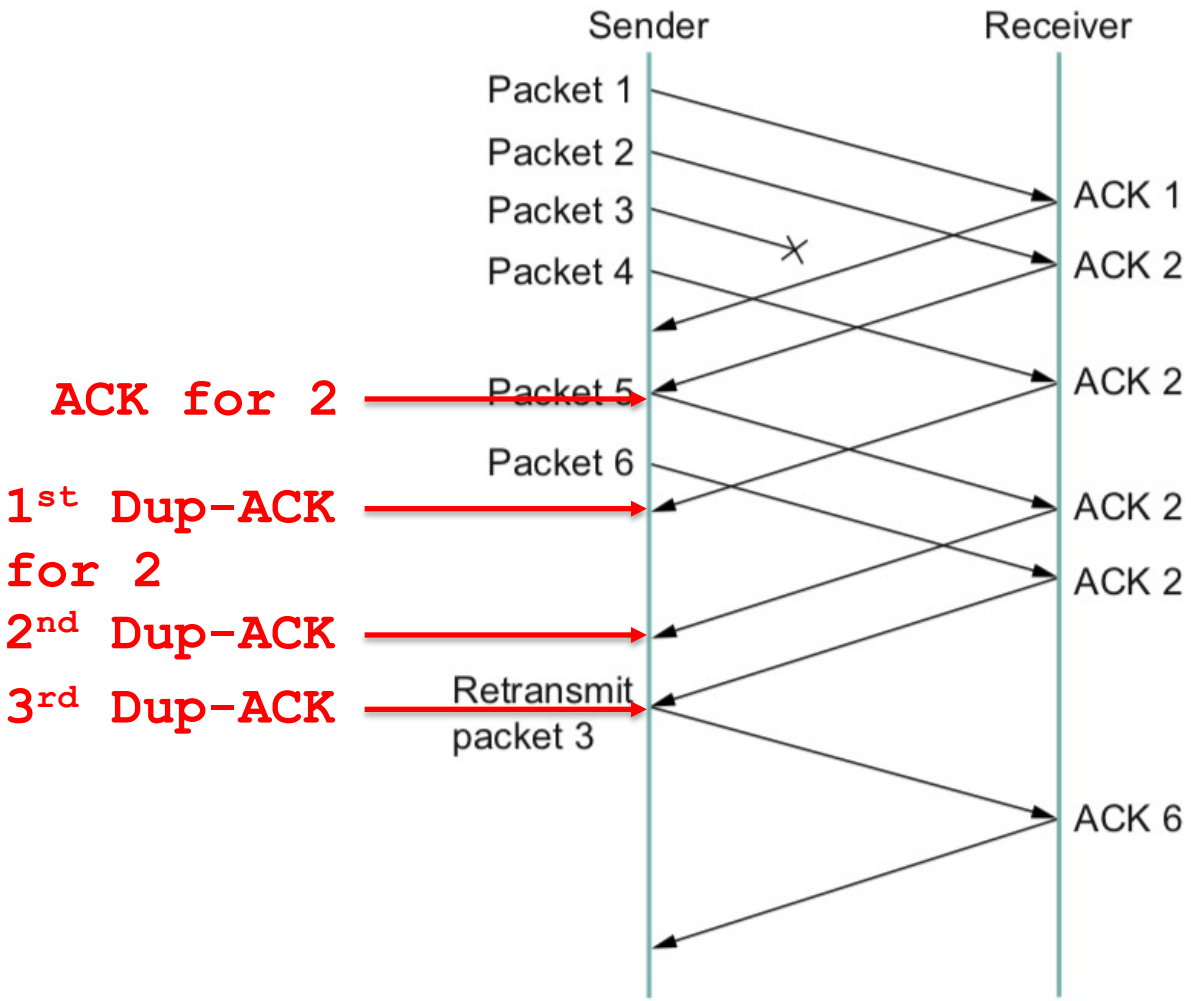
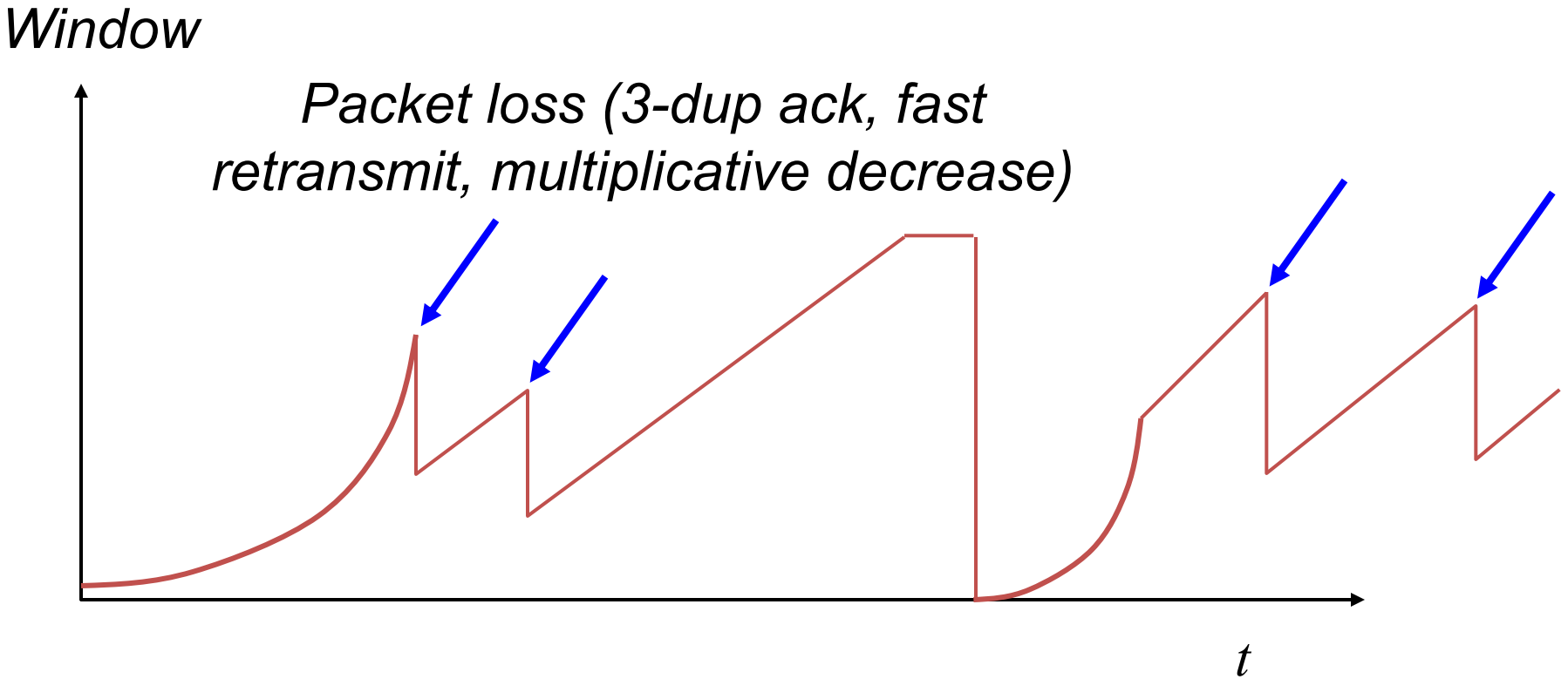


FIGURE 6.12 Fast retransmit based on duplicate ACKs.

Congestion window during Fast retransmit



Two Kinds of Loss in TCP

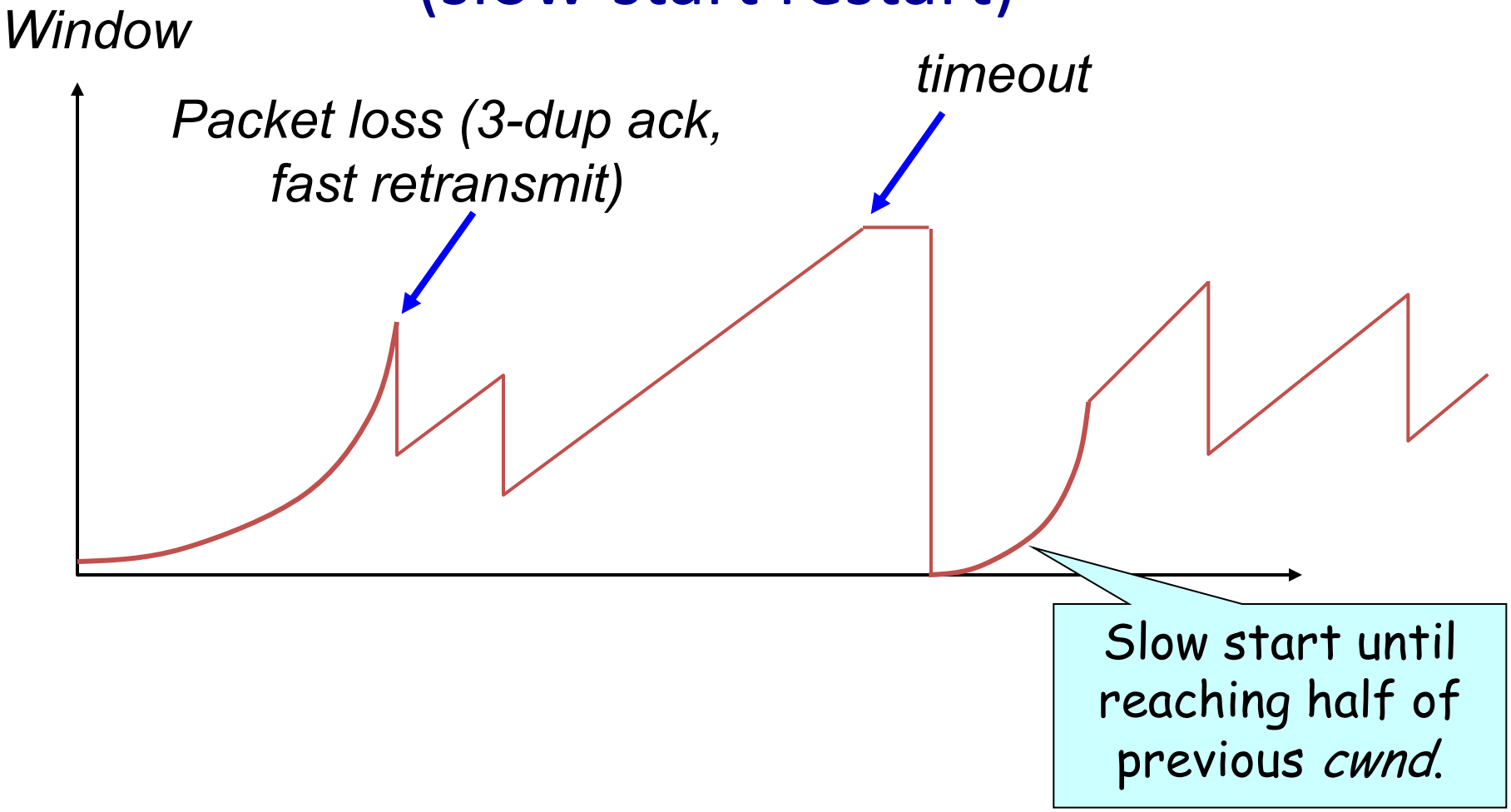
(1) Triple duplicate ACK (Fast retransmit)

- Packet n is lost, but packets n+1, n+2, etc. arrive
- Then, sender quickly resends packet n
- Do a multiplicative decrease and keep going

(2) Timeout (Repeat slow start – slow start restart)

- Packet n is lost and detected via a timeout
- Blasting entire CWND would cause another burst
- Better to start over with a low CWND

Repeating Slow Start After Timeout (slow-start restart)

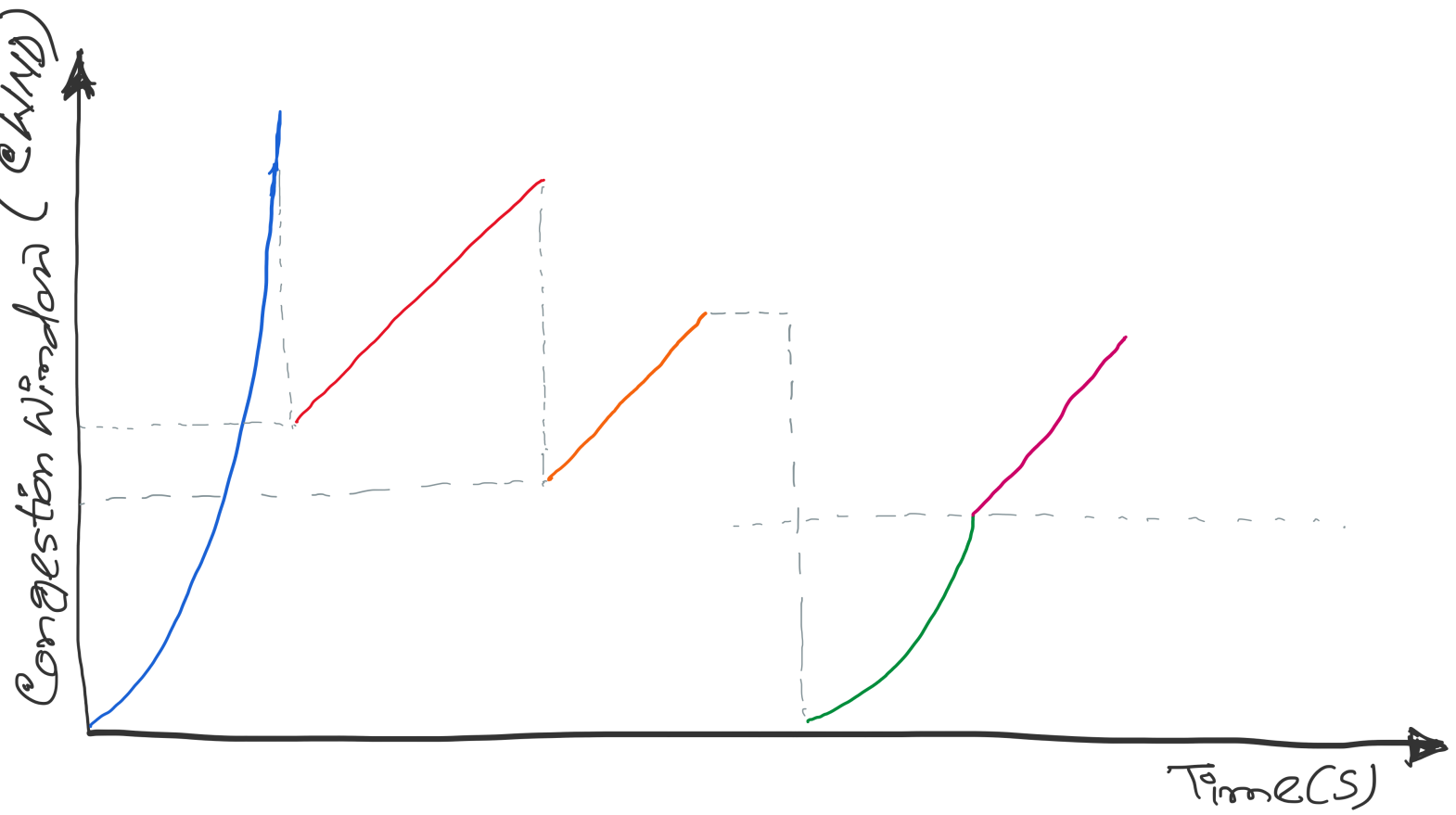


Slow-start restart: Go back to CWND of 1, but take advantage of knowing the previous value of CWND.

Repeating Slow Start After Idle Period

- Suppose a TCP connection goes idle for a while
- Eventually, the network conditions change
  - Maybe many more flows are traversing the link
- Dangerous to start transmitting at the old rate
  - Previously-idle TCP sender might blast network
  - … causing excessive congestion and packet loss
- So, some TCP implementations repeat slow start
  - Slow-start restart after an idle period

TCP congestion control: CWND graph



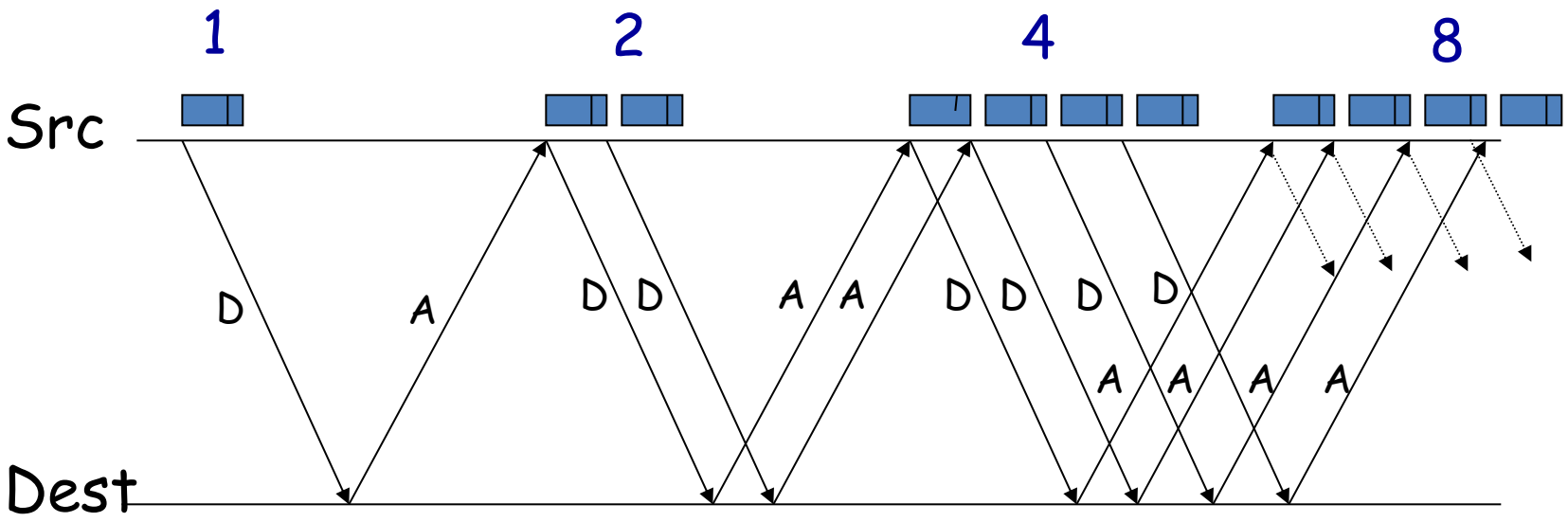
Implementation details

Update Time: Per RTT vs Per ACK

Implementation details

Update Time: Per RTT vs Per ACK

Double CWND per round-trip time

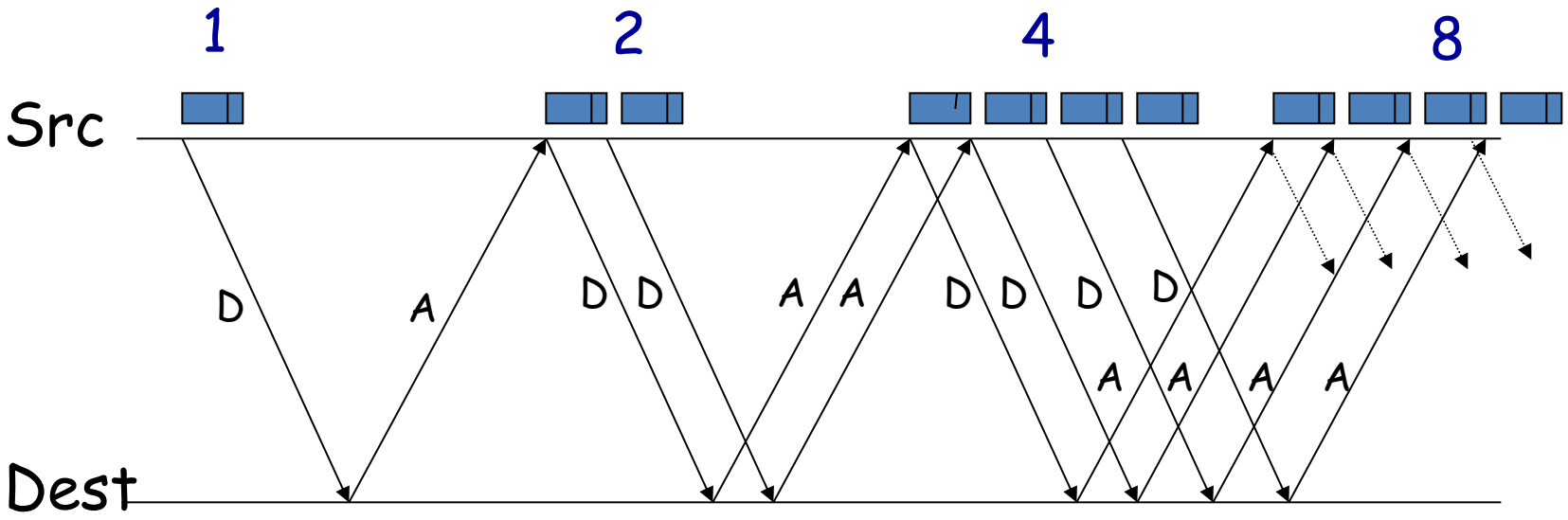


Implementation details

Update Time: Per RTT vs Per ACK

Double CWND per round-trip time

Implemented as CWND = CWND + 1 (per ACK)

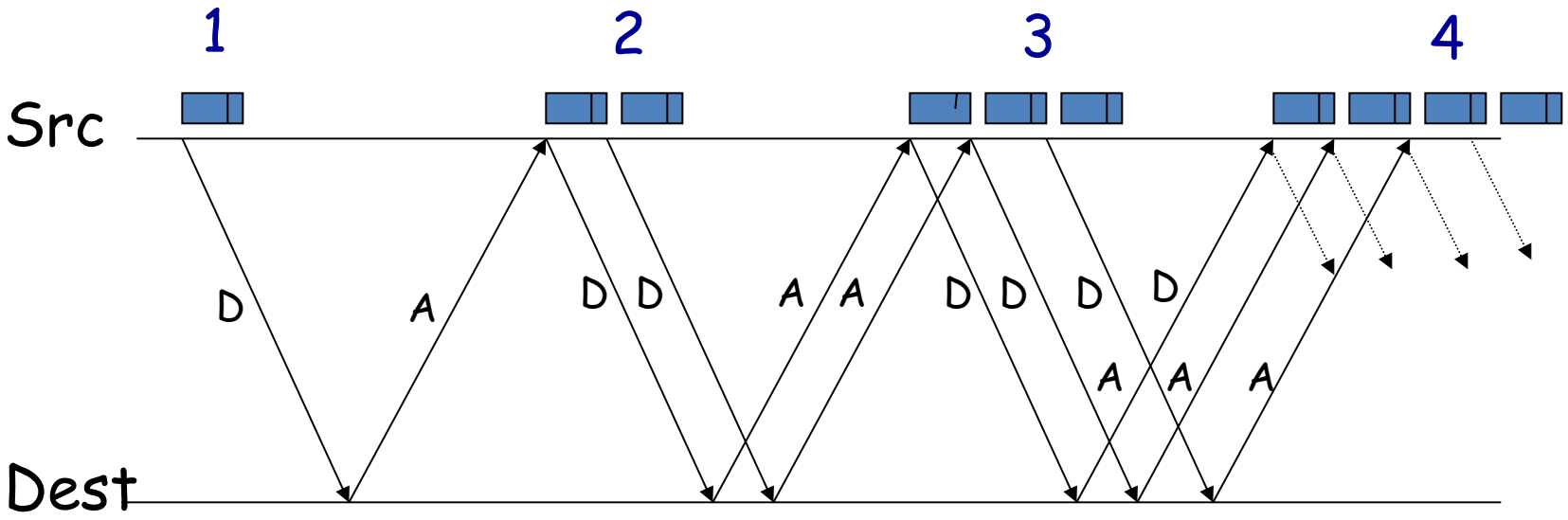


Implementation details

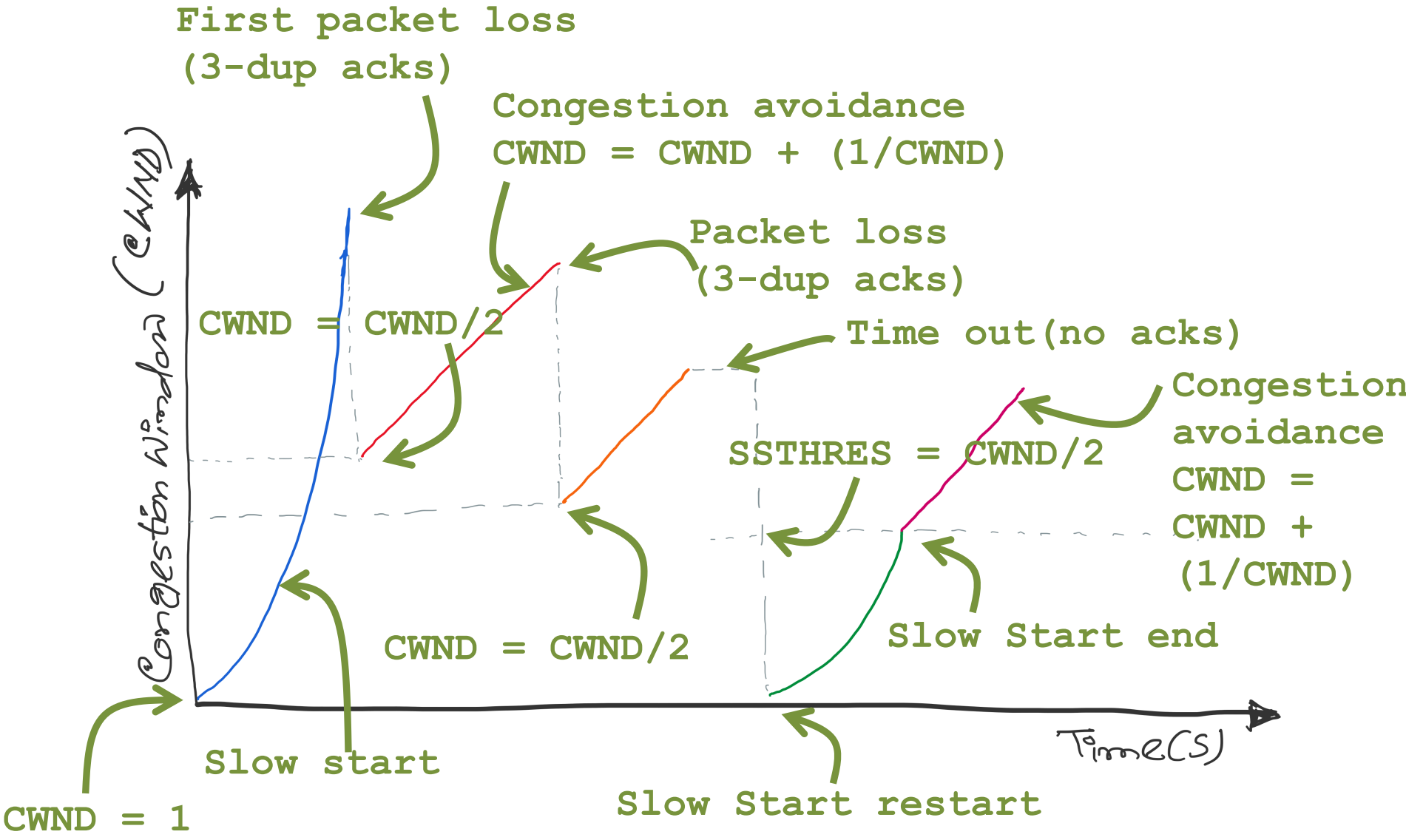
Update Time: Per RTT vs Per ACK

Linear increase of CWND per round-trip time

Implemented as CWND = CWND + 1/CWND (per ACK)

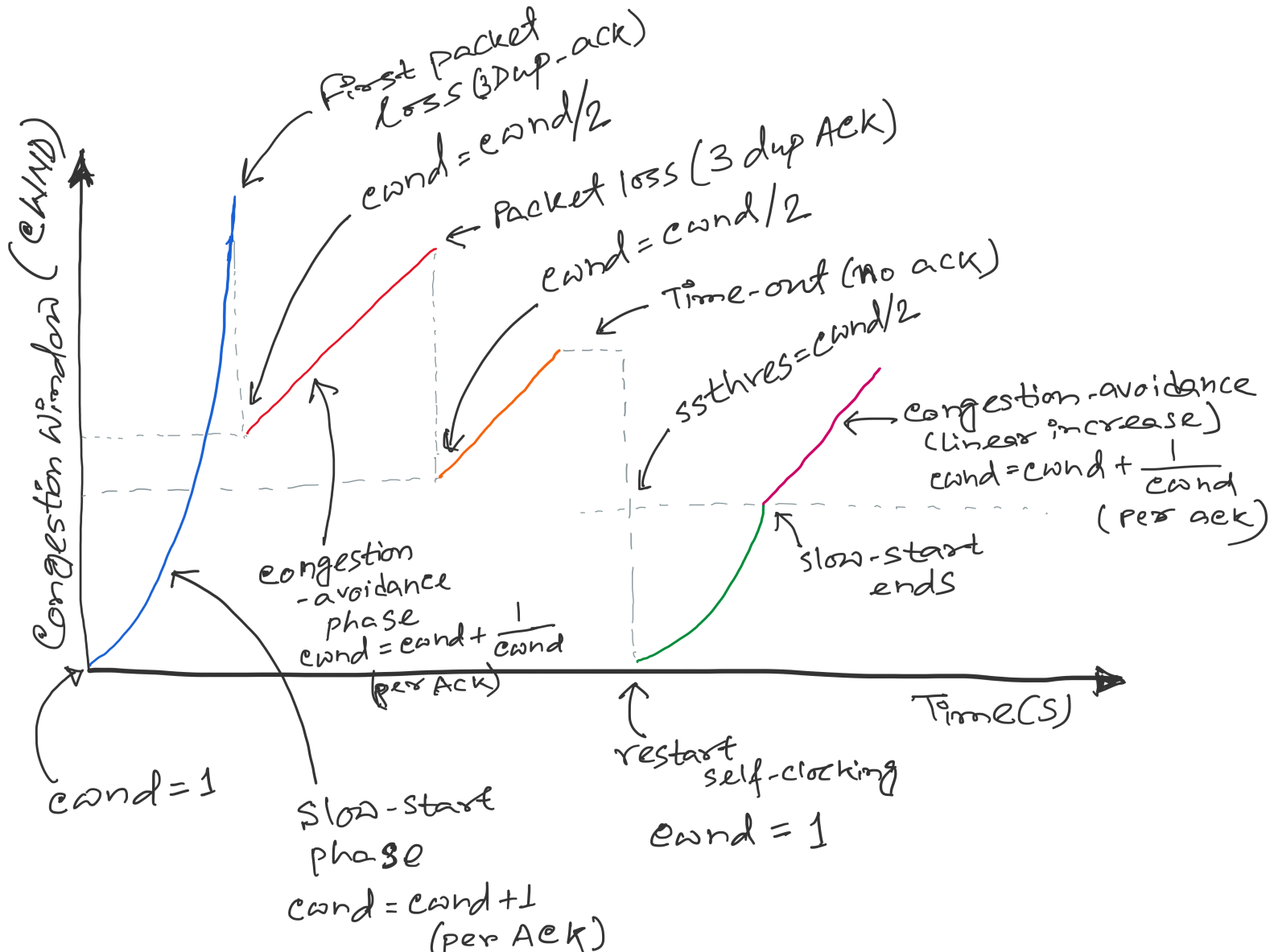


TCP congestion control: CWND graph



TCP congestion control

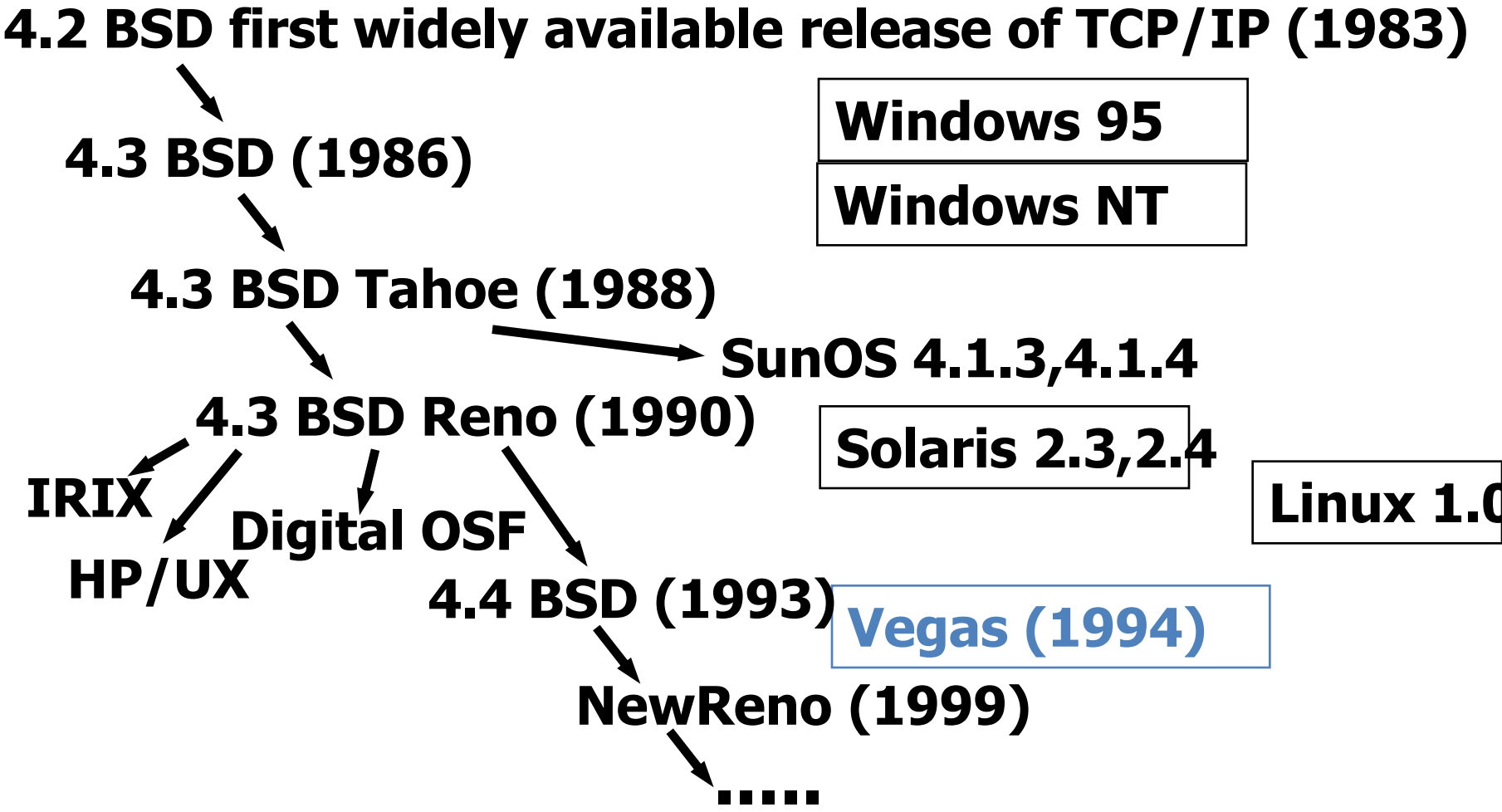
SSTHRES = Slow-start threshold



1. Development of Congestion Control over TCP versions

TCP versions

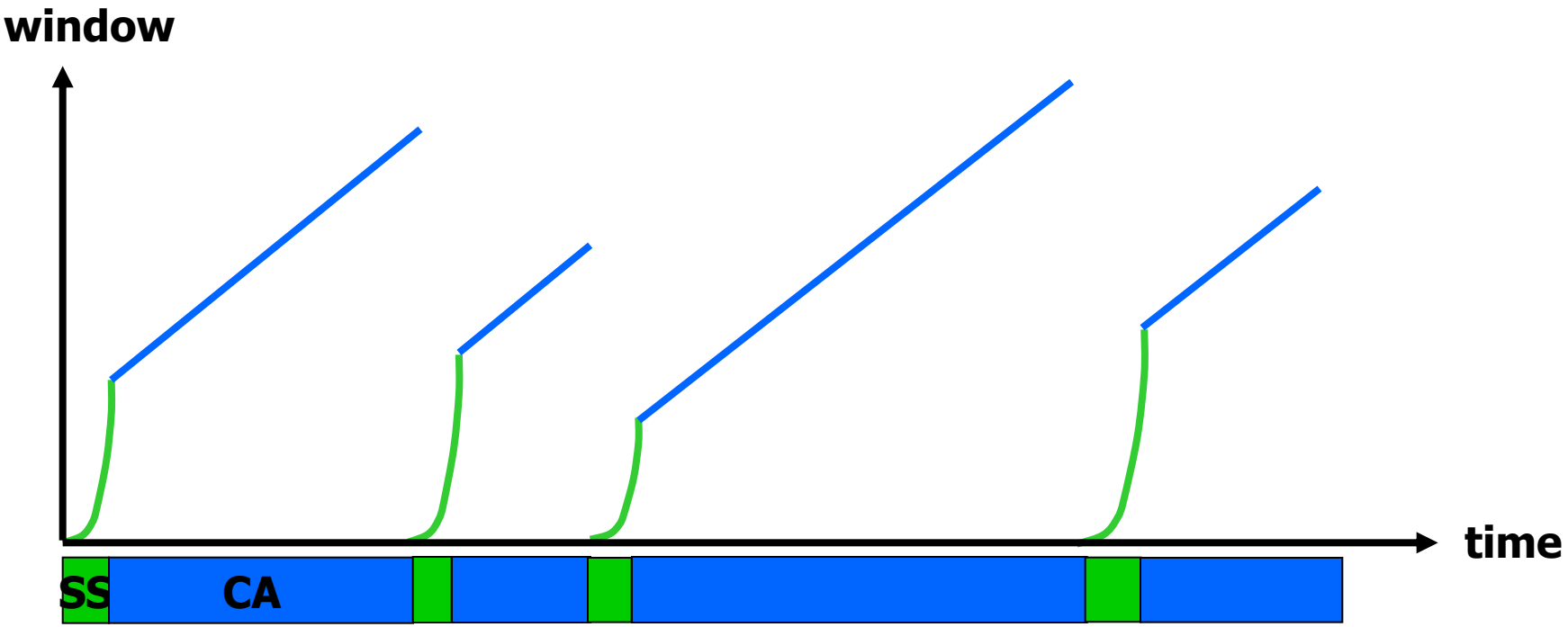
- TCP is not perfectly homogenous (200+)



TCP Congestion Controls

- Tahoe (Jacobson 1988)
  - Slow Start
  - Congestion Avoidance
  - Fast Retransmit (FR)

TCP Tahoe (Jacobson 1988)



SS: Slow Start

CA: Congestion Avoidance

Summary: Tahoe

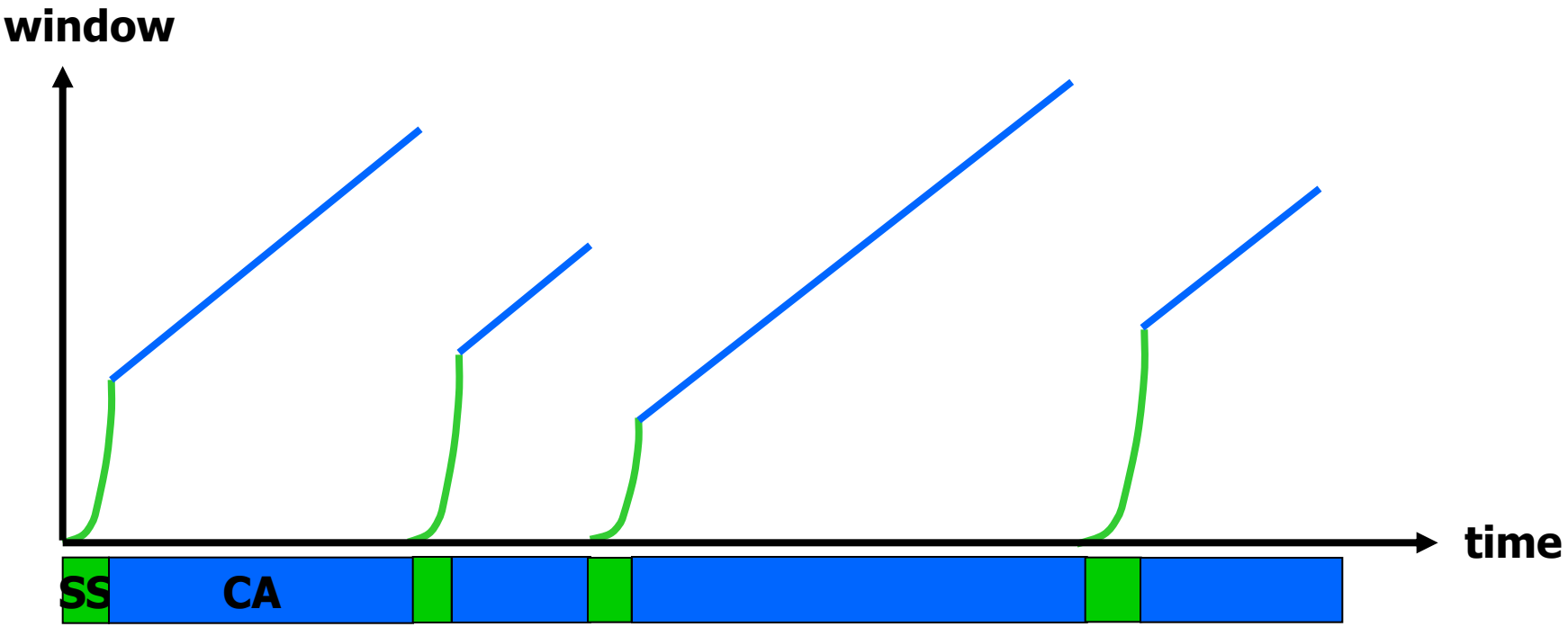
- Basic ideas
  - Gently probe network for spare capacity
  - Drastically reduce rate on congestion
  - Windowing: self-clocking
  - Other functions: round trip time estimation, error recovery

```
for every ACK {
    if (W < ssthresh) then W++    (SS)
    else  W += 1/W                (CA)
}
for every loss {
    ssthresh = W/2
          W  = 1
}
```

TCP Congestion Controls

- Tahoe (Jacobson 1988)
  - Slow Start
  - Congestion Avoidance
  - Fast Retransmit (FR)
- Reno (Jacobson 1990)
  - Fast Recovery

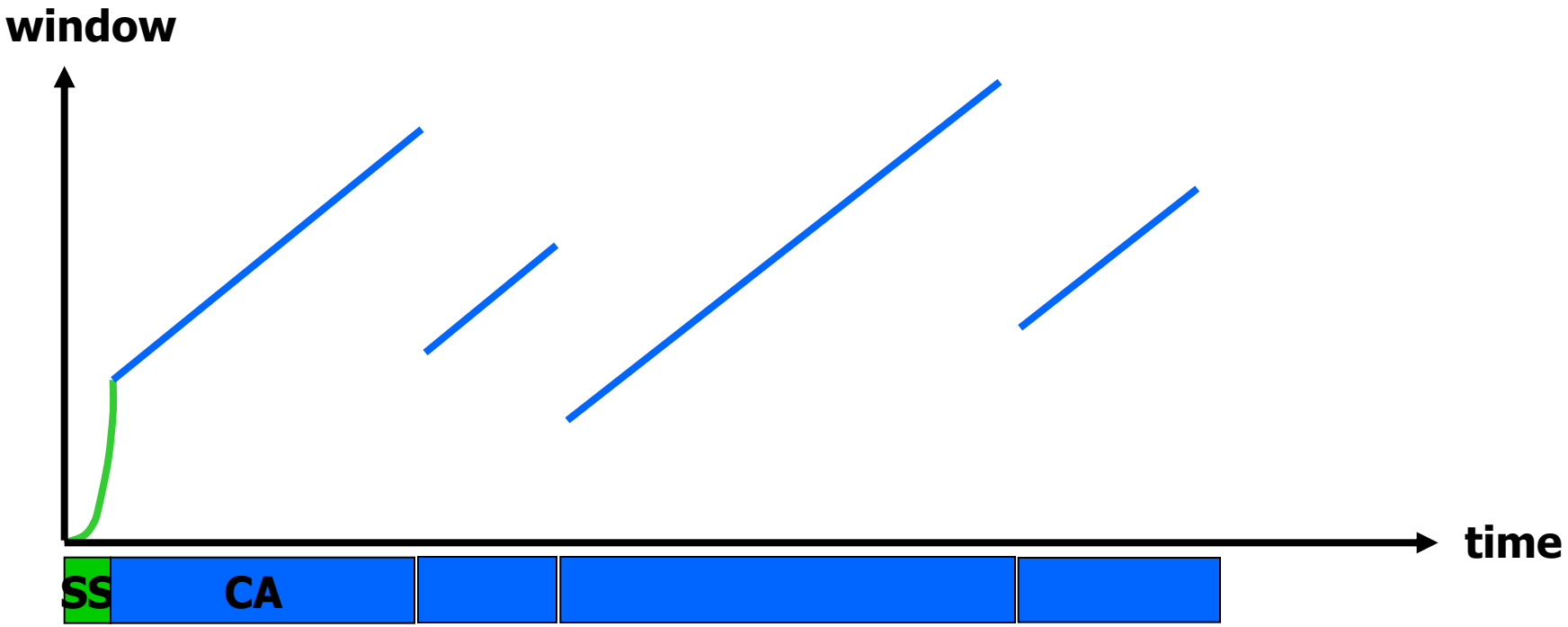
Tahoe's CWND recovers slowly



SS: Slow Start

CA: Congestion Avoidance

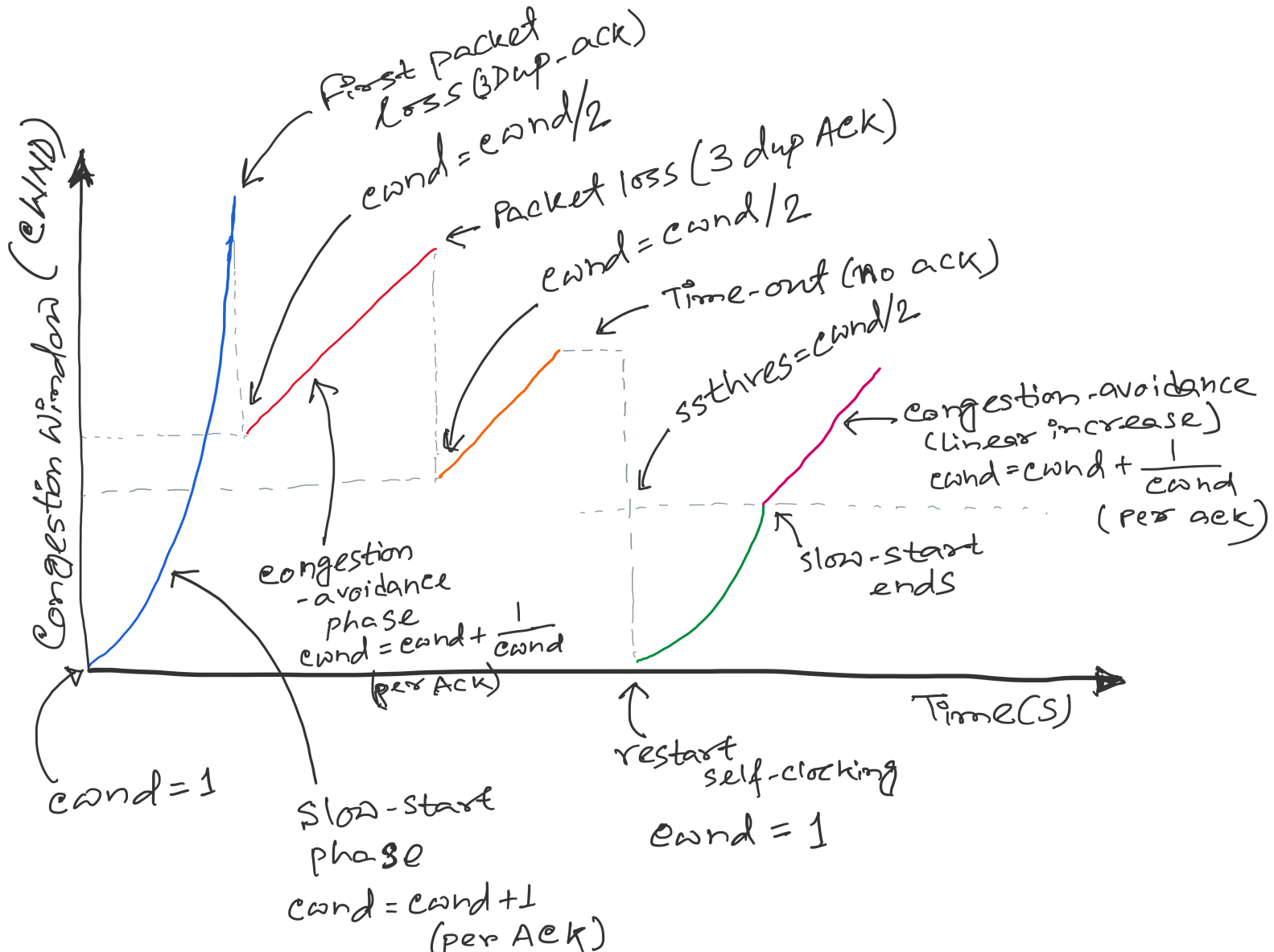
TCP Reno



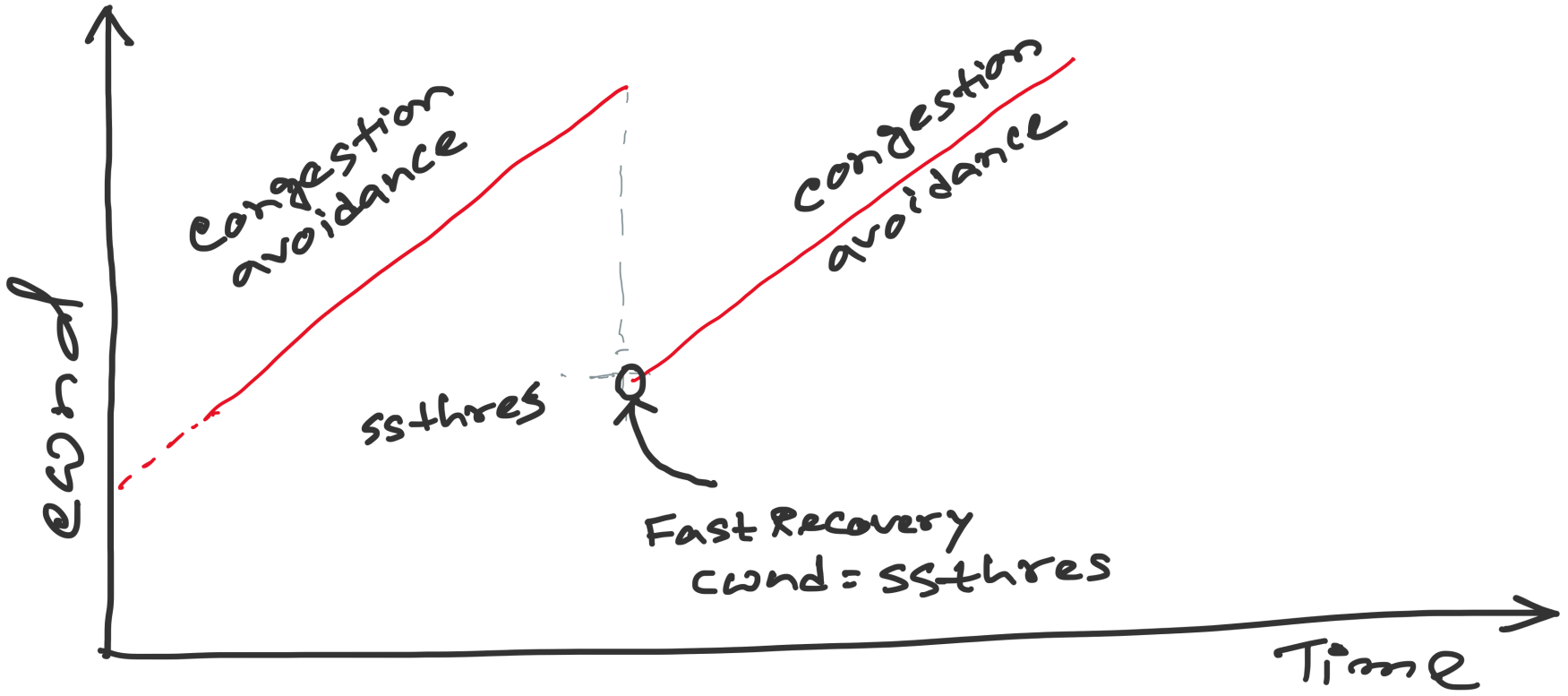
SS: Slow Start
CA: Congestion Avoidance

Fast retransmission/Fast recovery

TCP Reno


Congestion window (cwnd)
Time(s)
cwnd = 1
slow-start phase
cwnd = cwnd + 1 (per Ack)
congestion -avoidance phase
cwnd = cwnd + 1/cwnd (per ACK)
First packet loss (3Dup-ack)
cwnd = cwnd/2
Packet loss (3 dup ACK)
cwnd = cwnd/2
Time-out (no ack)
ssthres = cwnd/2
restart self-clocking
cwnd = 1
slow-start ends
congestion-avoidance (linear increase)
cwnd = cwnd + 1/cwnd (per ack)

TCP Reno: details of Fast Recovery



TCP Reno: details of Fast Recovery