CS4450

Computer Networks:
Architecture and Protocols

Lecture 24
TCP congestion control

Rachit Agarwal

Basic Components of TCP

- **Connections:** Explicit set-up and tear-down of TCP sessions/connections
- **Segments, Sequence numbers, ACKs**
    - TCP uses byte sequence numbers to identify payloads
    - ACKs referred to sequence numbers
    - Window sizes expressed in terms of # of bytes
- **Retransmissions**
    - Can’t be correct without retransmitting lost/corrupted data
    - TCP retransmits based on timeouts and duplicate ACKs
        - Timeouts based on estimate of RTT
- **Flow Control:** Ensures the sender does not overwhelm the receiver
- **Congestion Control:** Dynamic adaptation to network path’s capacity

TCP Congestion Control

TCP congestion control: high-level idea

- End hosts adjust sending rate
- Based on implicit feedback from the network
  - Implicit: router drops packets because its buffer overflows, not because it's trying to send message
- Hosts probe network to test level of congestion
  - Speed up when no congestion (i.e., no packet drops)
  - Slow down when when congestion (i.e., packet drops)
- How to do this efficiently?
  - Extend TCP's existing window-based protocol…
  - Adapt the window size based in response to congestion

All These Windows…

- **Flow control window**: Advertised Window (RWND)
    - How many bytes can be sent without overflowing receivers buffers
    - Determined by the receiver and reported to the sender
- **Congestion Window** (CWND)
    - How many bytes can be sent without overflowing routers
    - Computed by the sender using congestion control algorithm
- **Sender-side window** = minimum{CWND,RWND}
    - Assume for this lecture that RWND >> CWND

Note

- This lecture will talk about CWND in units of MSS
    - Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet
    - This is only for pedagogical purposes
- Keep in mind that real implementations maintain CWND in bytes

Basics of TCP Congestion

- Congestion Window (CWND)
  - Maximum # of unacknowledged bytes to have in flight
  - Rate ~CWND/RTT
- Adapting the congestion window
  - Increase upon lack of congestion: optimistic exploration
  - Decrease upon detecting congestion
- But how do you detect congestion?

Not All Losses the Same

- **Duplicate ACKs: isolated loss**
    - Still getting ACKs
- **Timeout: possible disaster**
    - Not enough duplicate ACKs
    - Must have suffered several losses

How to Adjust CWND?

- Consequences of over-sized window much worse than having an under-sized window
  - Over-sized window: packets dropped and retransmitted
  - Under-sized window: somewhat lower throughput
- Approach
  - Gentle increase when un-congested (exploration)
  - Rapid decrease when congested

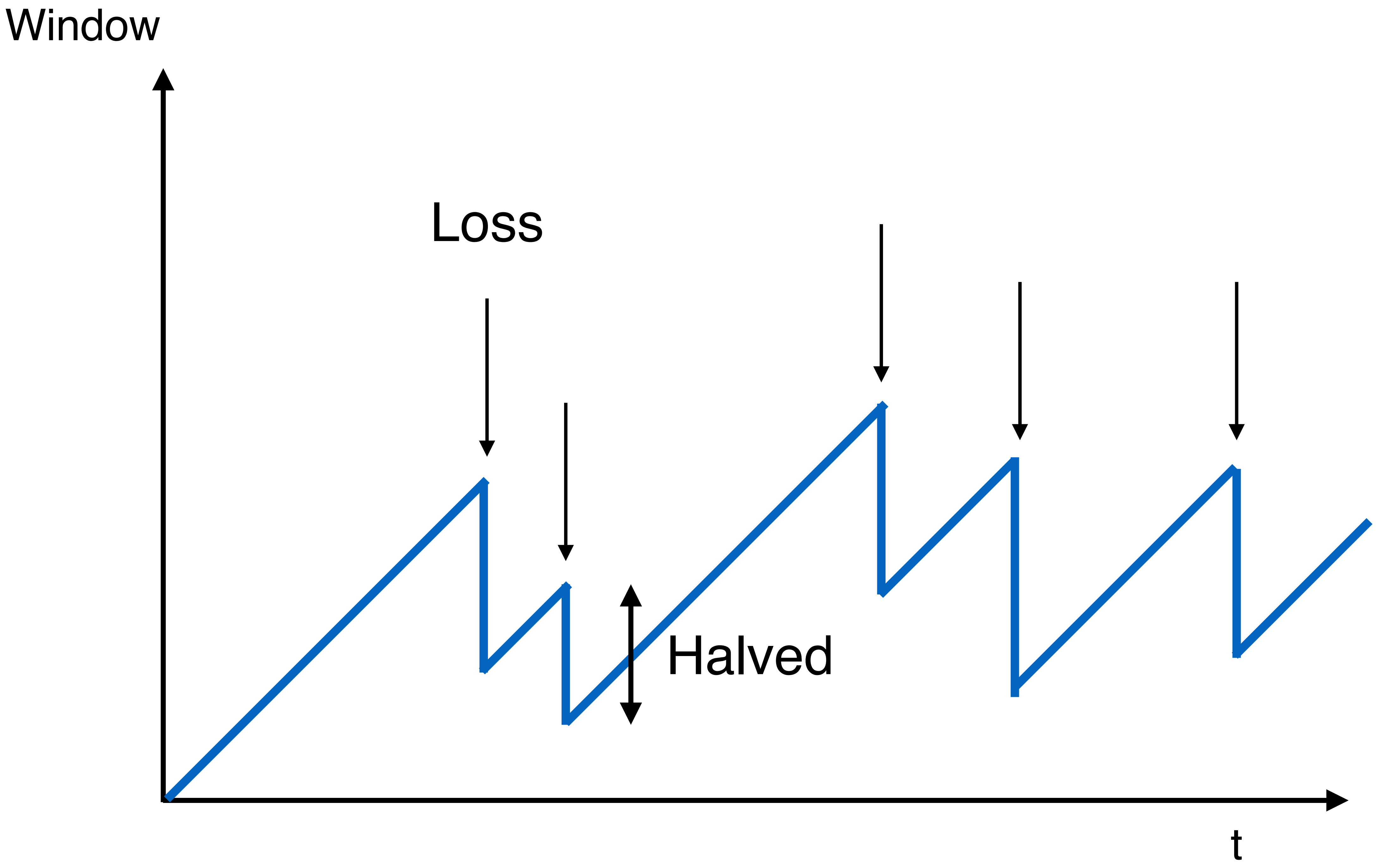
Additive Increase, Multiplicative Decrease (AIMD)

- Additive increase
  - On success of last window of data, increase by one MSS
  - If W packets in a row have been ACKed, increase W by one
  - i.e., +1/W per ACK
- Multiplicative decrease
  - On loss of packets by DupACKs, divide congestion window by half
  - Special case: when timeout, reduce congestion window to one MSS

AIMD

- ACK: CWND -> CWND + 1/CWND
  - When CWND is measured in MSS
  - Note: after a full window, CWND increase by 1 MSS
  - Thus, **CWND increases by 1 MSS per RTT**
- 3rd DupACK: CWND -> CWND/2
- Special case of timeout: CWND -> 1 MSS

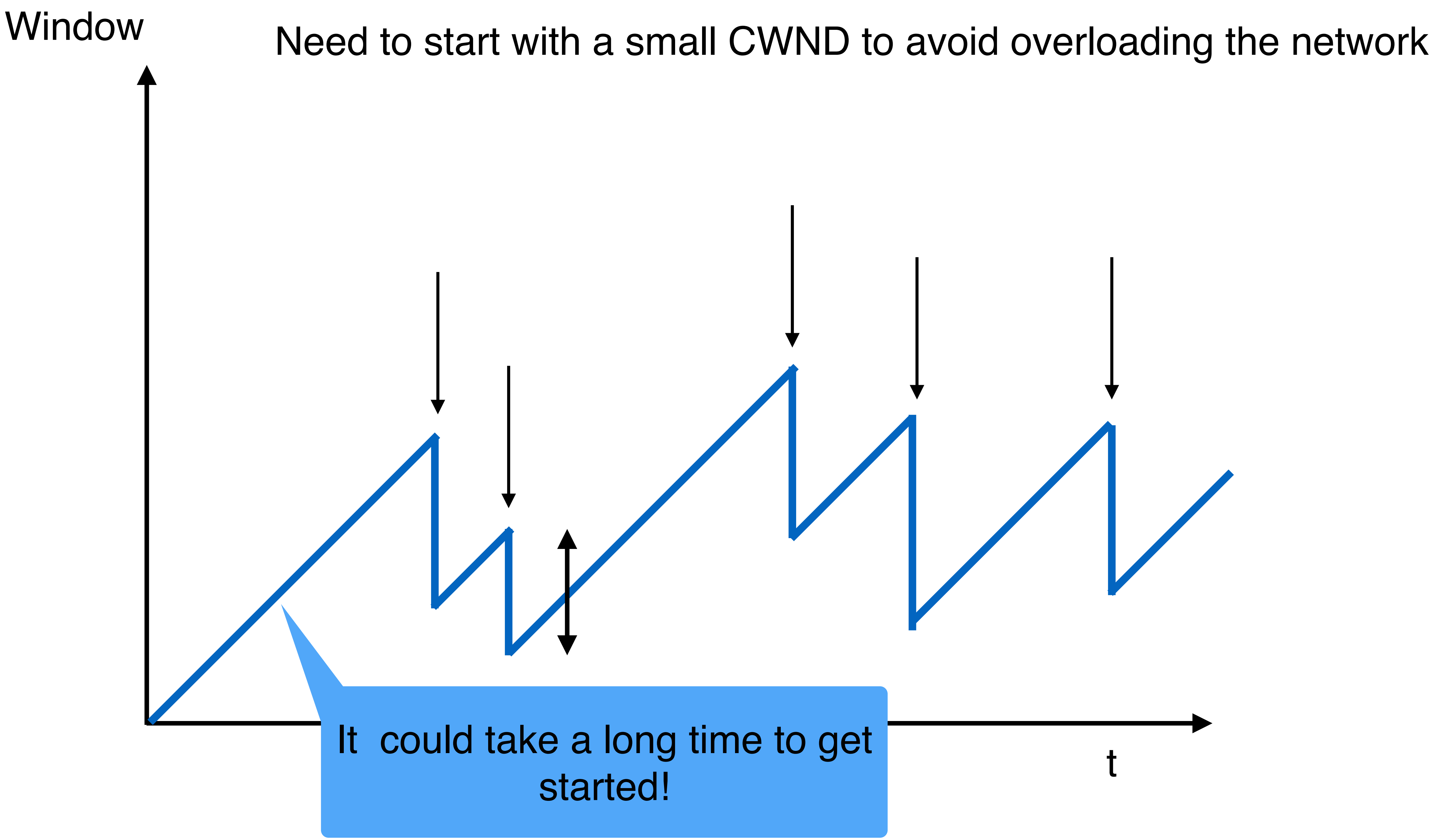
Leads to the TCP Sawtooth



Questions?

Slow Start

AIMD Starts Too Slowly



Bandwidth Discovery with Slow Start

- Goal: estimate available bandwidth
  - Start slow (for safety)
  - But ramp up quickly (for efficiency)

- Consider
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 MSS
  - Window size to fill 1 Gbps = 12,500 MSS
    - With just AIMD, it takes about 12500 RTTs to get to this window size!
    - ~21 mins

“Slow Start” Phase

- Start with a small congestion window
  - Initially, CWND is 1 MSS
  - So, initial sending rate is MSS/RTT

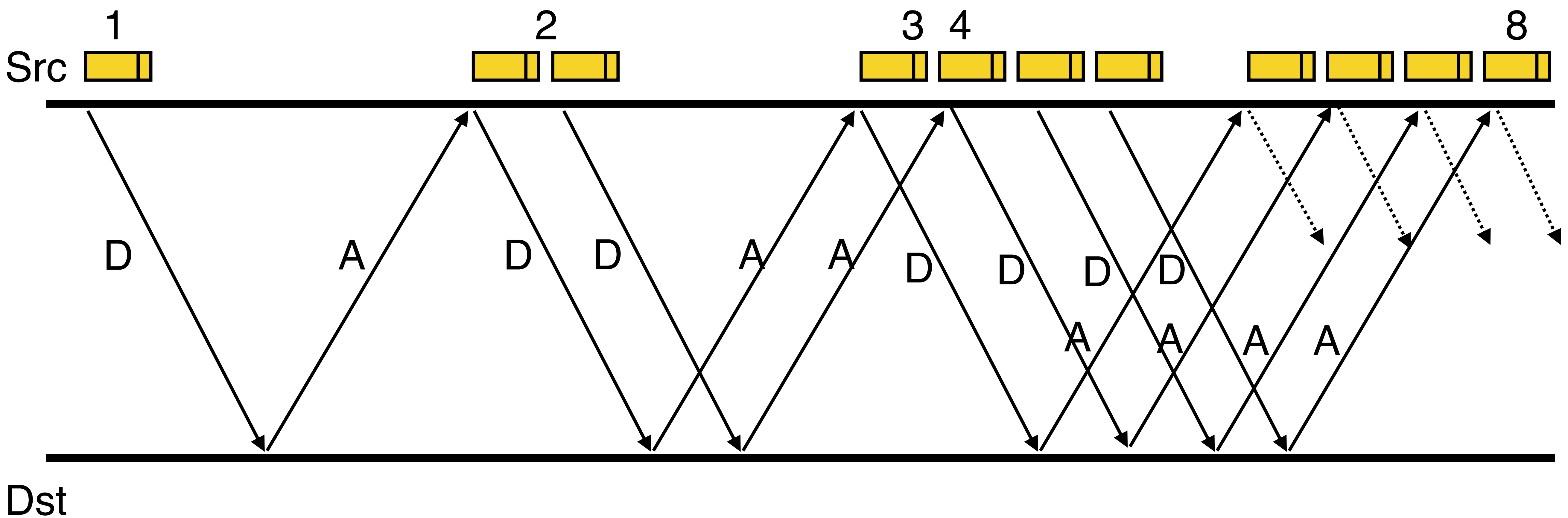
- That could be pretty wasteful
  - Might be much less than the actual bandwidth
  - Linear increase takes a long time to accelerate

- Slow-start phase (**actually “fast start”**)
  - Sender starts at a slow rate (hence the name)
  - … but increases exponentially until first loss

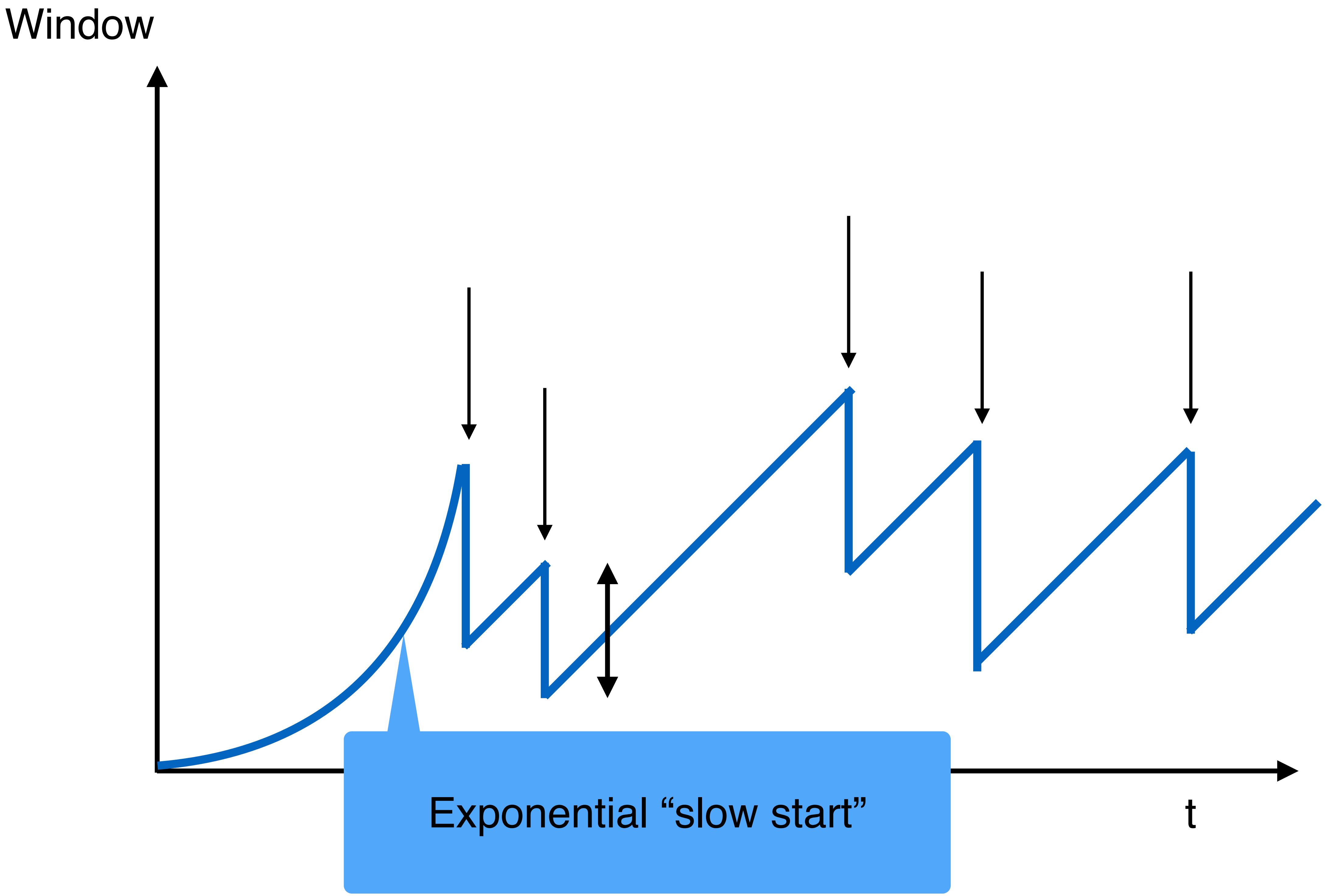
Slow Start in Action

Double CWND per round-trip time

Simple implementation: on each ACK, CWND += MSS



Slow Start and the TCP Sawtooth



Why is it called slow-start? Because TCP originally had no congestion control mechanism. The source would just start by sending a whole window's worth of data.

Slow-Start vs AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Introduce a “slow start threshold” (ssthresh)
  - Initialized to a large value
  - On timeout, ssthresh = CWND/2
- When CWND > ssthresh, sender switches from slow-start to AIMD-style increase

Timeouts

Loss Detected by Timeout

- Sender starts a timer that runs for RTO seconds
- **Restart timer whenever ACK for new data arrives**
- If timer expires
  - Set SSHTHRESH <- CWND/2 (“Slow Start Threshold”)
  - Set CWND <- 1 (MSS)
  - Retransmit **first** lost packet
  - Execute Slow Start until CWND > SSTHRESH
  - After which switch to Additive Increase

Summary of Increase

- "Slow start": increase CWND by 1 (MSS) for each ACK
  - A factor of 2 per RTT
- Leave slow-start regime when either:
  - CWND > SSTHRESH
  - Packet drop detected by dupacks
- Enter AIMD regime
  - Increase by 1 (MSS) for each window's worth of ACKed data

Summary of Decrease

- Cut CWND half on loss detected by dupacks
    - **Fast retransmit to avoid overreacting**
- Cut CWND all the way to 1 (MSS) on **timeout**
    - Set ssthresh to CWND/2
- Never drop CWND below 1 (MSS)
    - Our correctness condition: always try to make progress

TCP Congestion Control Details

Implementation

- State at sender
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
  - dupACKcount
  - Timer, as before

- Events at sender
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

- What about receiver? Just send ACKs upon arrival

Event: ACK (new data)

- If in slow start
  - CWND += 1

CWND packets per RTT
Hence after one RTT with
no drops:
CWND = 2 x CWND

Event: ACK (new data)

- If CWND <= ssthresh
  - CWND += 1
- Else
  - CWND = CWND + 1/CWND

Slow Start Phase

Congestion Avoidance Phase

(additive increase)

CWND packets per RTT
Hence after one RTT with no drops:
CWND = CWND + 1

Event: Timeout

- On Timeout
    - ssthresh <- CWND/2
    - CWND <- 1

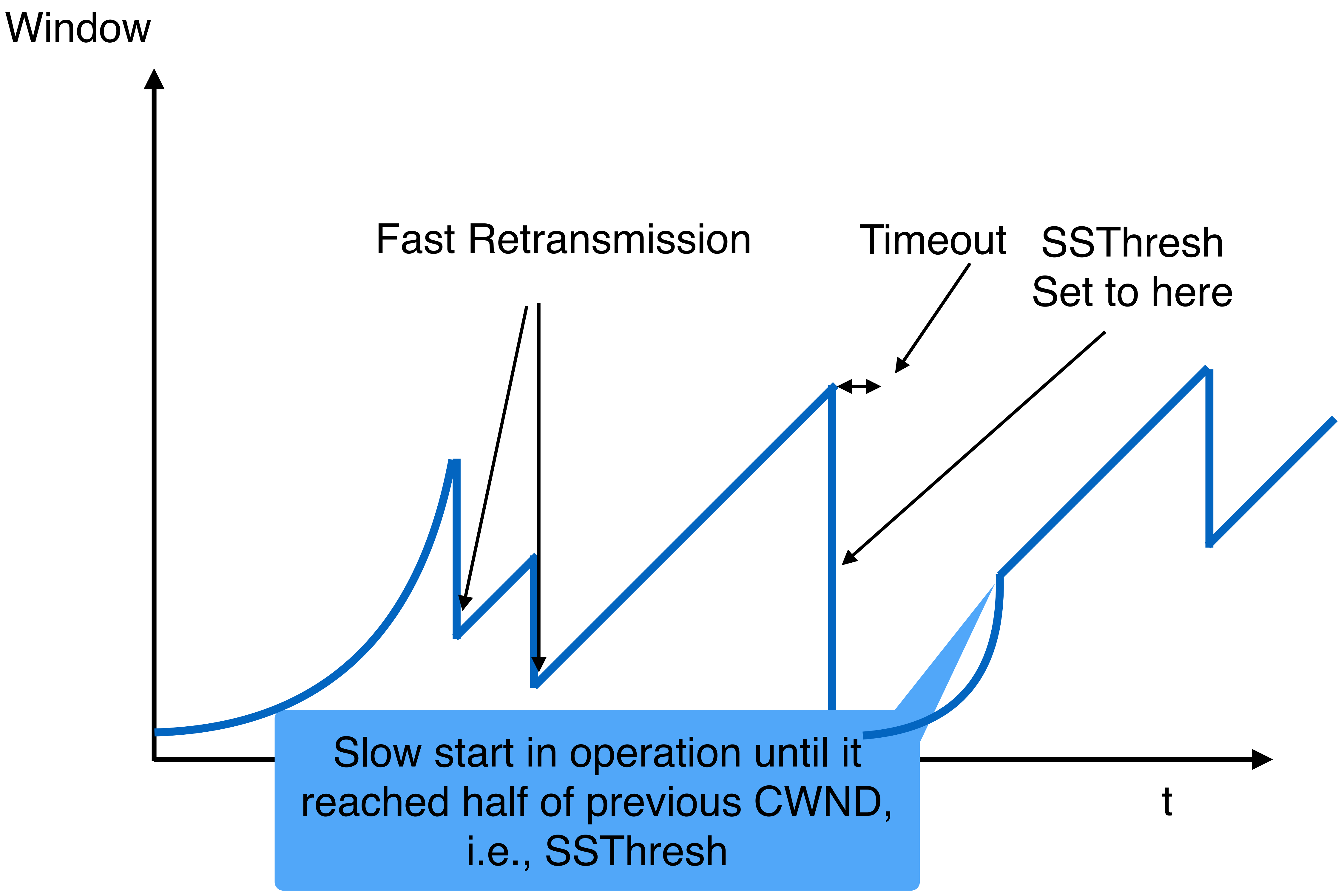
Event: dupACK

- dupACKcount++

- If dupACKcount = 3 /* fast retransmit */
    - ssthresh <- CWND/2
    - CWND <- CWND/2

Remains in congestion avoidance after fast retransmission

Time Diagram



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

TCP Flavors

- TCP Tahoe
  - CWND = 1 on triple dupACK
- TCP Reno
  - CWND = 1 on timeout
  - CWND = CWND/2 on triple dupACK
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - Incorporates selective acknowledgements

Our default assumption

TCP and fairness guarantees

Consider A Simple Model

- Flows **ask** for an amount of bandwidth r_i
  - In reality, this request is implicit (the amount they send)
- The link gives them an amount a_i
  - Again, this is implicit (by how much is forwarded)
  - $a_i <= r_i$
- There is some total capacity C
  - Sum $a_i <= C$

Fairness

- When all flows want the same rate, fair is easy
    - Fair share = C/N
    - C = capacity of link
    - N = number of flows
- Note:
    - This is fair share per link. This is not a global fair share
- When not all flows have the same demand?
    - What happens here?

Example 1

- Requests: r_i Allocations: a_i
- $C = 20$
    - Requests: $r_1 = 6$, $r_2 = 5$, $r_3 = 4$
- Solution
    - $a_1 = 6$, $a_2 = 5$, $a_3 = 4$
- When bandwidth is plentiful, everyone gets their request
- This is the easy case

Example 2

- Requests: r_i Allocations: a_i
- $C = 12$
  - Requests: $r_1 = 6$, $r_2 = 5$, $r_3 = 4$
- One solution
  - $a_1 = 4$, $a_2 = 4$, $a_3 = 4$
  - Everyone gets the same
- Why not proportional to their demands?
  - $a_i = (12/15)\ r_i$
- Asking for more gets you more!
  - Not incentive compatible (i.e., cheating works!)
  - You can’t have that and invite innovation!

Example 3

- Requests: r_i Allocations: a_i
- $C = 14$
  - Requests: $r_1 = 6$, $r_2 = 5$, $r_3 = 4$
- $a_3 = 4$ (can't give more than a flow wants)
- Remaining bandwidth is 10, with demands 6 and 5
  - From previous example, if both want more than their share, they both get half
  - $a_1 = a_2 = 5$

Max-Min Fairness

- Given a set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are $a_i = \min(f, r_i)$
  - Where f is the unique value such that $\mathrm{Sum}(a_i) = C$ or set f to be infinite if no such value exists
- **This is what round-robin service gives**
  - If all packets are MTU
- Property:
  - If you don't get full demand, no one gets more than you

Computing Max-Min Fairness

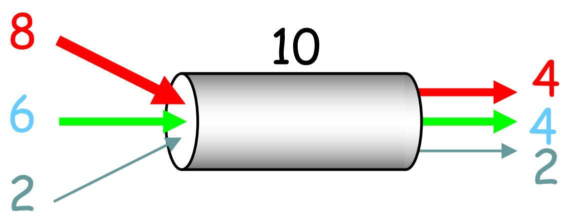
- Assume demands are in increasing order…
- If $C/N <= r_1$, then $a_i = C/N$ for all i
- Else, $a_1 = r1$, set $C = C - a_1$ and $N = N-1$
- Repeat
- Intuition: all flows requesting less than fair share get their request. Remaining flows divide equally

Example

- Assume link speed C is 10Mbps
- Have three flows:
  - Flow 1 is sending at a rate 8 Mbps
  - Flow 2 is sending at a rate 6 Mbps
  - Flow 3 is sending at a rate 2 Mbps
- How much bandwidth should each get?
  - According to max-min fairness?
- Work this out, talk to your neighbors

Example

- Requests: r_i Allocations: a_i
- Requests: $r_1 = 8$, $r_2 = 6$, $r_3 = 2$
- $C = 10$, $N = 3$, $C/N = 3.33$
  - Can serve all for r_3
  - Remove r_3 from the accounting: $C = C - r_3 = 8$, $N = 2$
- $C/2 = 4$
  - Can't service all for r_1 or r_2
  - So hold them to the remaining fair share: $f = 4$

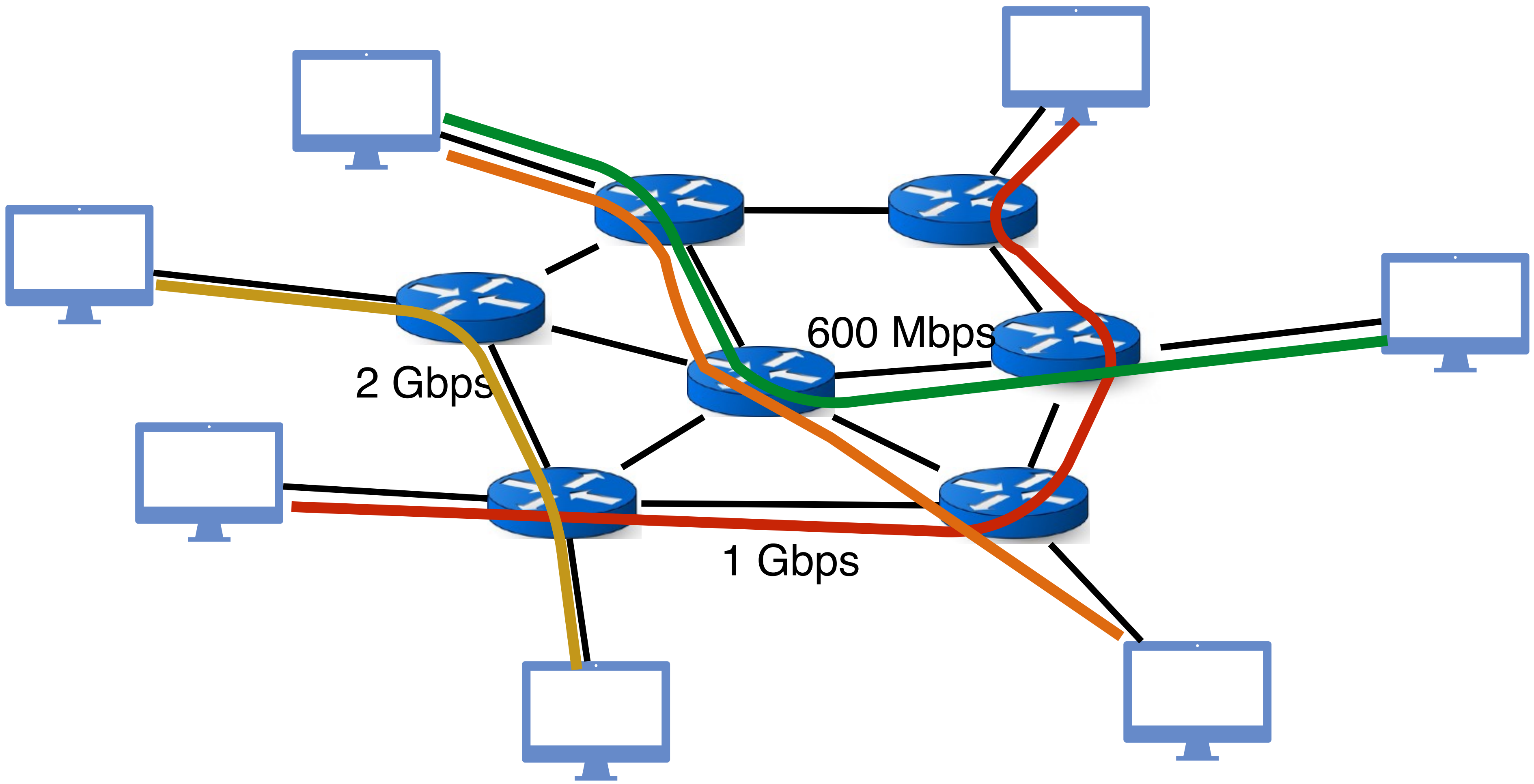


$f = 4$:
$\min(8, 4) = 4$
$\min(6, 4) = 4$
$\min(2, 4) = 2$

Max-Min Fairness

- Max-min fairness the natural per-link fairness
- Only one that is
    - Symmetric
    - Incentive compatible (asking for more doesn't help)

Reality of Congestion Control



Congestion control is a resource allocation problem involving many flows, many links and complicated global dynamics

Classical result:

In a stable state

(no dynamics; all flows are infinitely long; no failures; etc.)

TCP guarantees max-min fairness