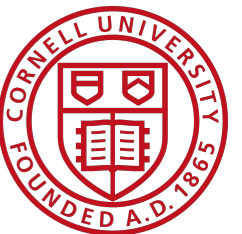


CS4450

Computer Networks: Architecture and Protocols

Lecture 22 More TCP Congestion Control

Rachit Agarwal



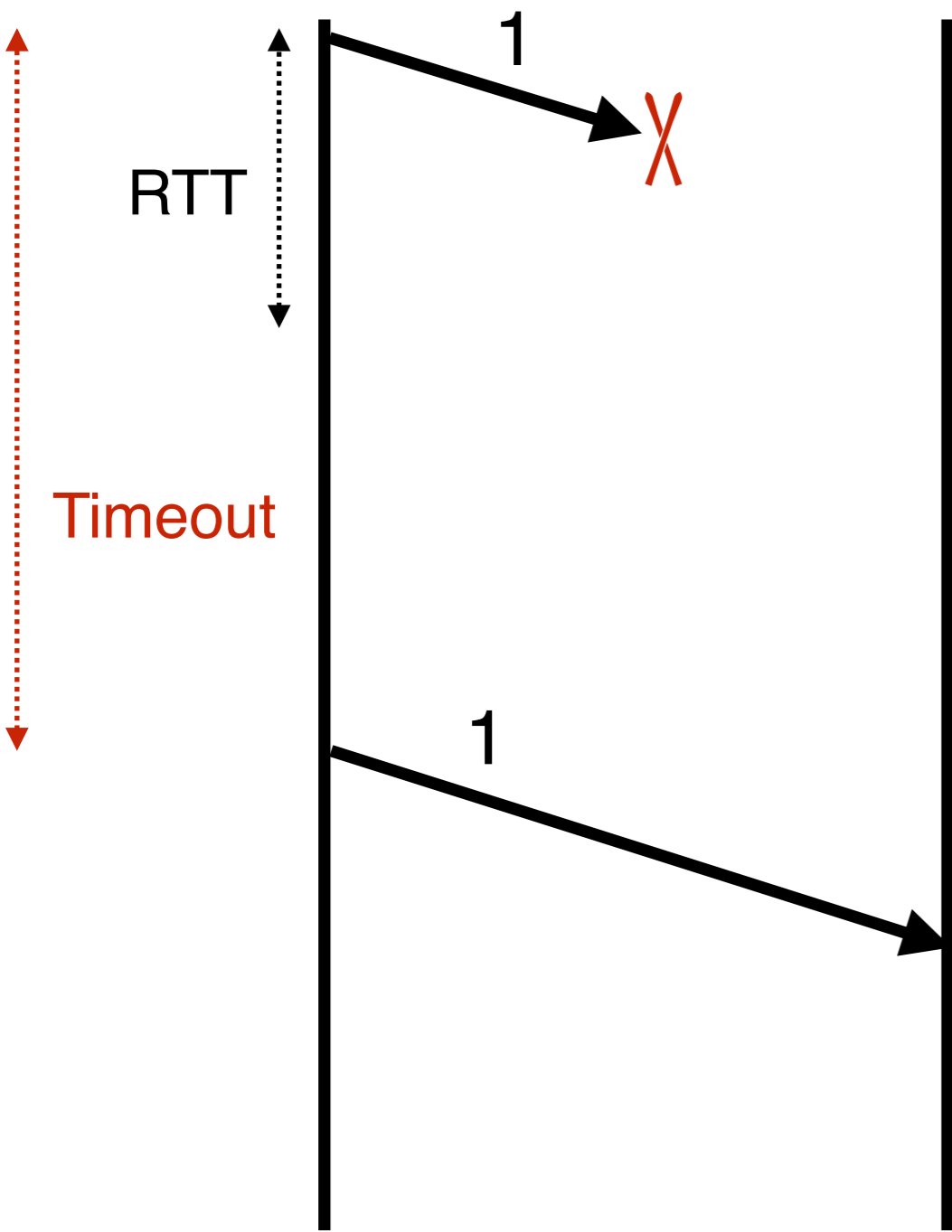
Recap: Basic Components of TCP

- **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**
- **Congestion Control**

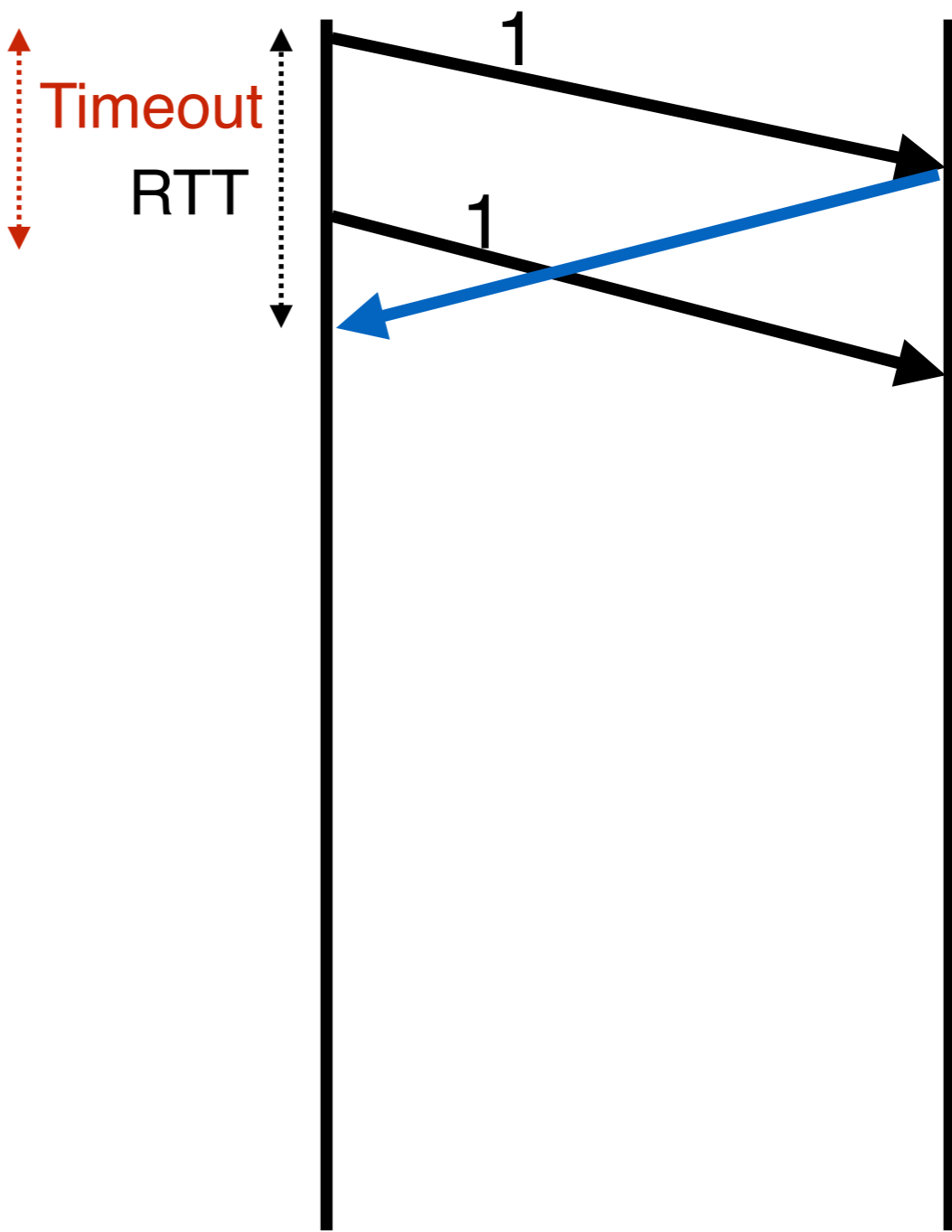
Recap: Loss with Cumulative ACKs

- Sender sends packets with 100B and seqnos
 - 100, 200, 300, 400, 500, 600, 700, 800, 900
- Assume 5th packet (seqno 500) is lost, but no others
- Stream of ACKs will be
 - 200, 300, 400, 500, 500, 500, 500, 500
- Duplicate ACKs are a sign of an isolated loss
 - The lack of ACK progress means 500 hasn't been delivered
 - Stream of ACKs means some packets are being delivered
- Therefore, could trigger resend upon receiving k duplicate ACKs
 - **Large k -> fewer retransmissions, more latency, lower rate (why?)**
 - **Small k -> more retransmissions, lower latency**

Recap: Setting the Timeout Value (RTO)



Timeout too long -> inefficient



Timeout too short -> duplicate packets

Recap: Flow Control (Sliding Window)

- Advertised Window: W
 - Can send W bytes beyond the next expected byte
- Receiver uses W to prevent sender from overflowing buffer
- Limits number of bytes sender can have in flight

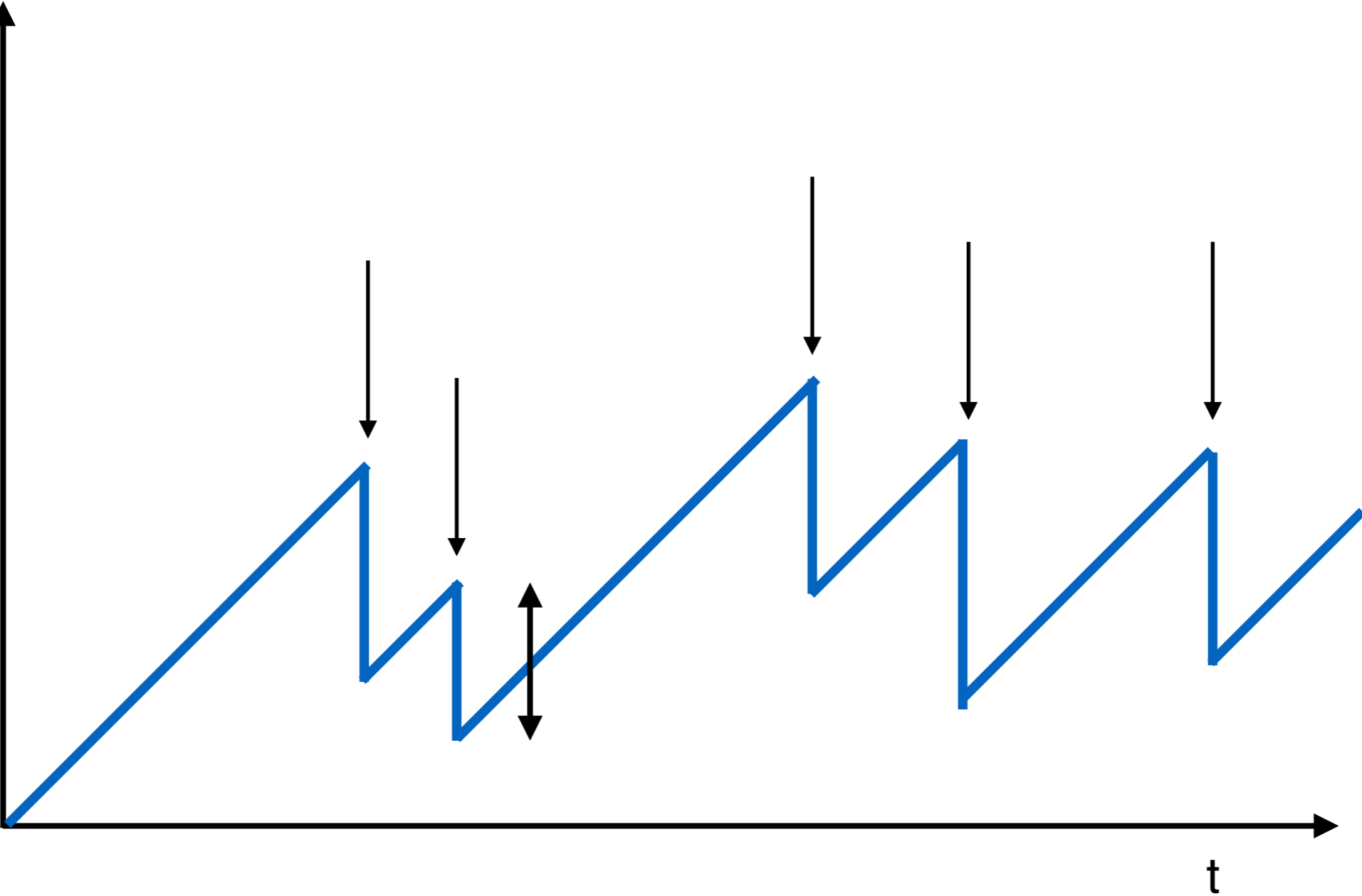
Recap: TCP congestion control: high-level idea

- 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

Recap: AIMD

Window

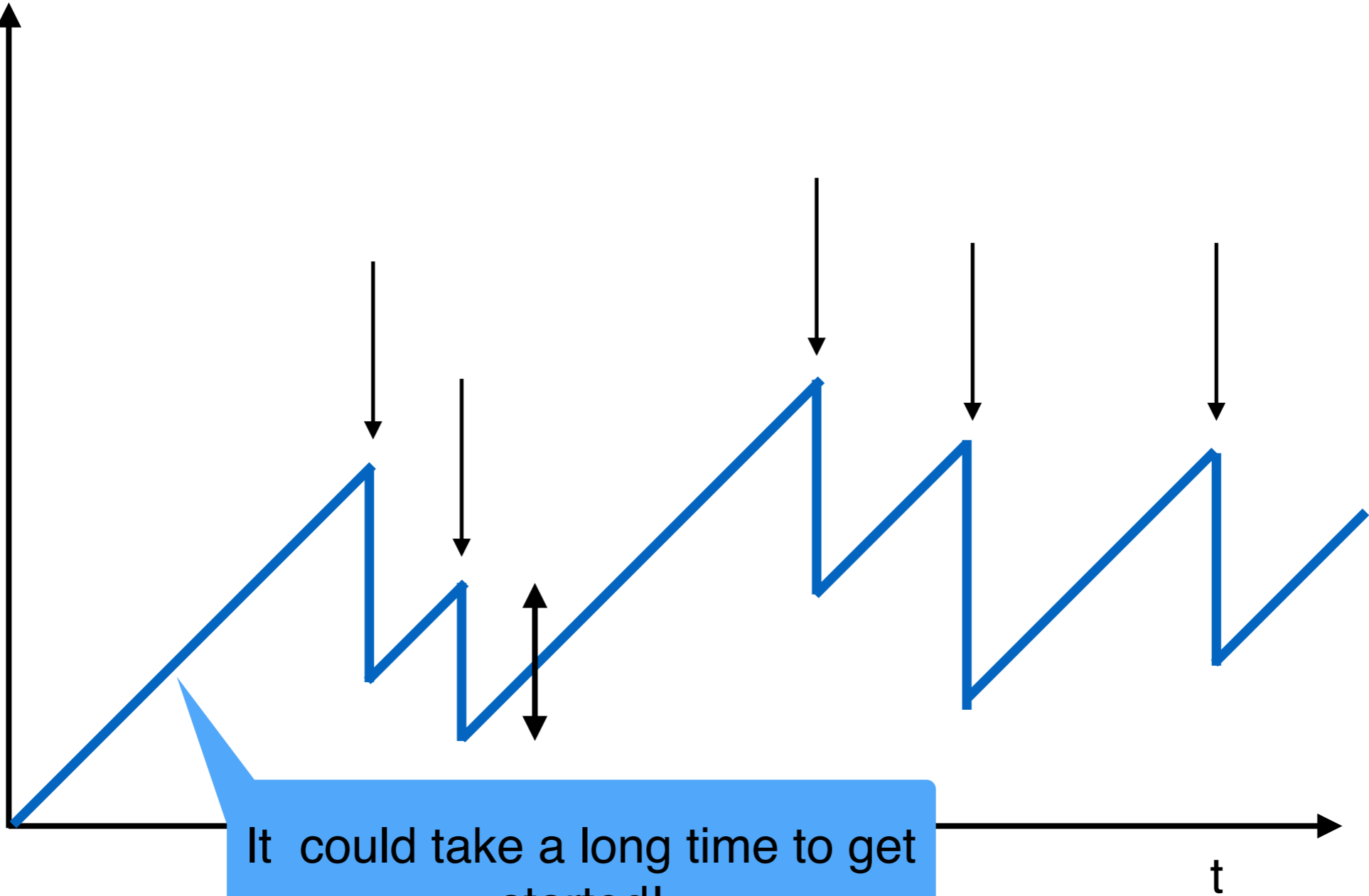
Need to start with a small CWND to avoid overloading the network



Recap: AIMD Starts Too Slowly

Window

Need to start with a small CWND to avoid overloading the network



It could take a long time to get started!

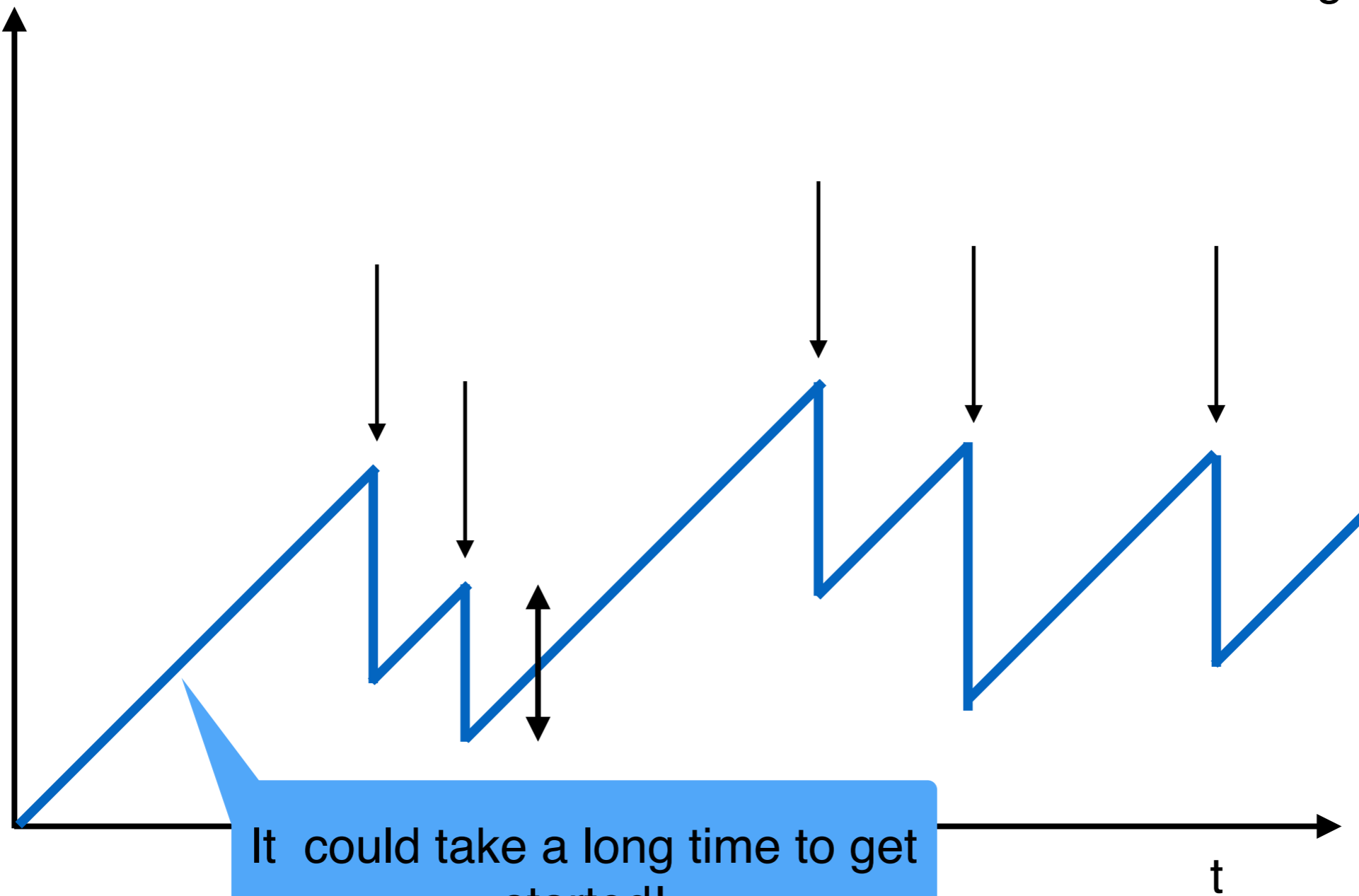
Any Questions?

Slow Start

AIMD Starts Too Slowly

Window

Need to start with a small CWND to avoid overloading the network



It could take a long time to get started!

Bandwidth Discovery with Slow Start

- Goal: estimate available bandwidth
 - Start slow (for safety)
 - But ramp up quickly (for efficiency)
- Consider
 - $RTT = 100\text{ms}$, $MSS=1000\text{bytes}$
 - 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

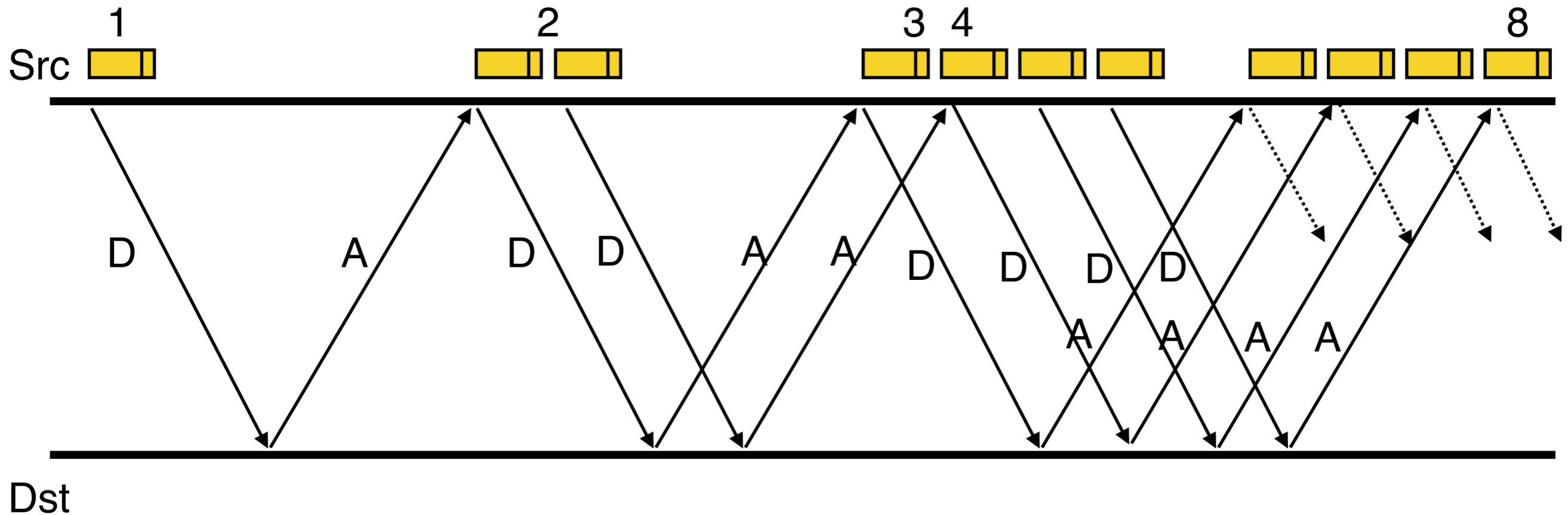
“Slow Start” Phase

- 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

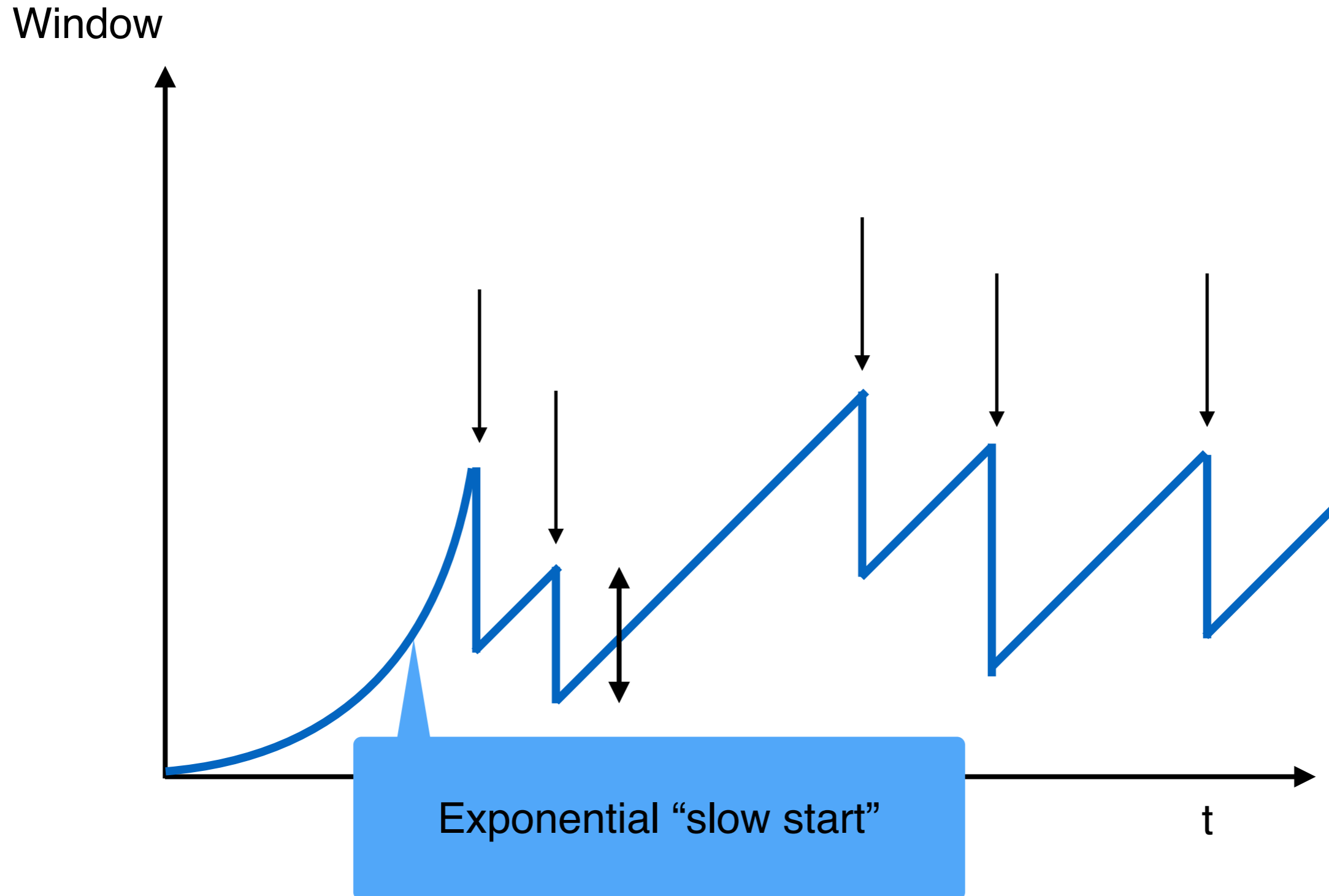
Slow Start in Action

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

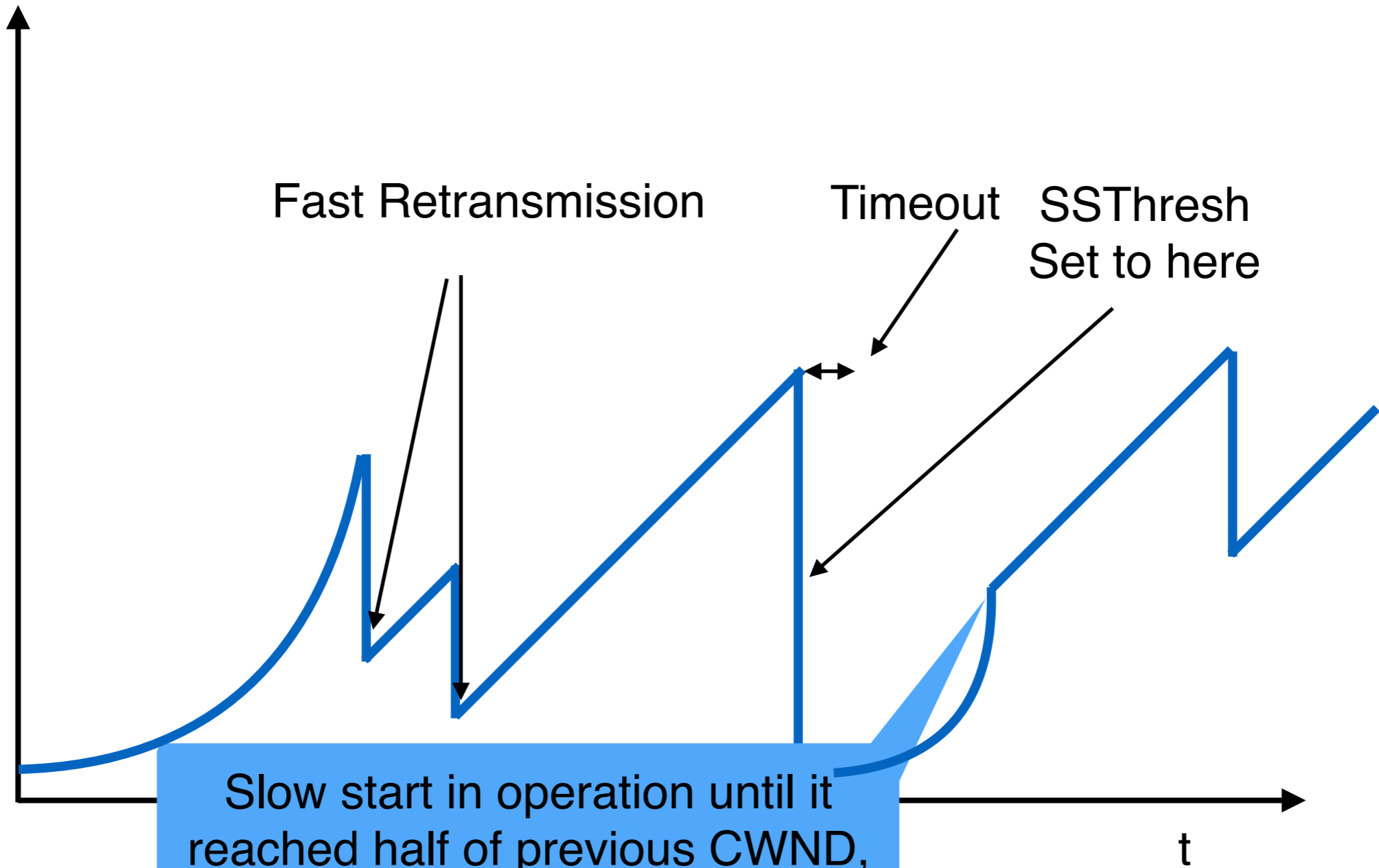
Timeouts

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 $SSTHRESH \leftarrow CWND/2$ (“Slow Start Threshold”)
 - Set $CWND \leftarrow 1$ (MSS)
 - Retransmit **first** lost packet
 - Execute Slow Start until $CWND > SSTHRESH$
 - After which switch to Additive Increase

TCP Time Diagram (with timeouts)

Window



Fast Retransmission

Timeout

Ssthresh
Set to here

Slow start in operation until it reached half of previous CWND, i.e., Ssthresh

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
- Cut CWND all the way to 1 (MSS) on **timeout**
 - Set ssthresh to $\text{CWND}/2$
- Never drop CWND below 1 (MSS)
 - Our correctness condition: always try to make progress

Any Questions?

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)
 - $a_i \leq r_i$
- There is some total capacity C
 - $\sum a_i \leq C$

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 $\text{Sum}(a_i) = C$ 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

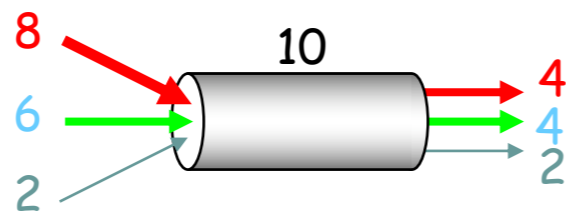
- Assume demands are in increasing order...
- If $C/N \leq r_1$, then $a_i = C/N$ for all i
- Else, $a_1 = r_1$, set $C = C - a_1$ and $N = N - 1$
- Repeat
- Intuition: all flows requesting less than fair share get their request.
Remaining flows divide equally

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: $f = 4$

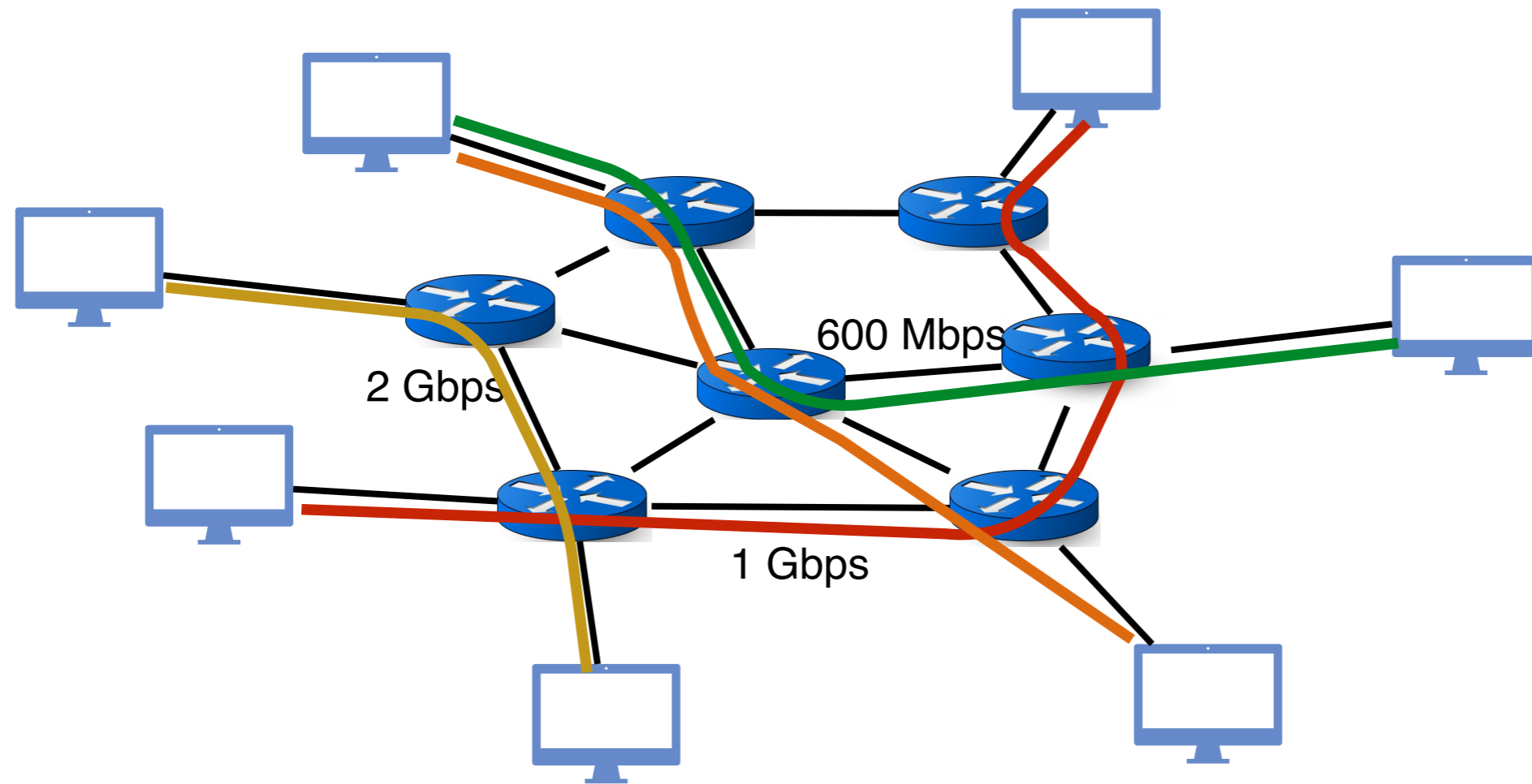


$f = 4:$
$\min(8, 4) = 4$
$\min(6, 4) = 4$
$\min(2, 4) = 2$

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

Any Questions?

The Many Failings of TCP Congestion Control

1. Fills up queues (large queueing delays)
2. Every segment not ACKed is a loss (non-congestion related losses)
3. Produces irregular saw-tooth behavior
4. Biased against long RTTs (unfair)
5. Not designed for short flows
6. Easy to cheat

(1) TCP Fills Up Queues

- TCP only slows down when queues fill up
 - High queueing delays
- Means that it is not optimized for latency
 - What is it optimized for then?
 - **Answer: Fairness**
- And many packets are dropped when buffer fills
- Alternative 1: Use small buffers
 - Is this a good idea?
 - Answer: No, bursty traffic will lead to reduced utilization
- Alternative: **Random Early Drop (RED)**
 - Drop packets on purpose **before** queue is full
 - A very clever idea

Random Early Drop (or Detection)

- Measure average queue size A with exponential weighting
 - Average: Allows for short bursts of packets without over-reacting
- Drop probability is a function of A
 - No drops if A is very small
 - Low drop rate for moderate A 's
 - Drop everything if A is too big
- Drop probability applied to incoming packets
- Intuition: link is fully utilized well before buffer is full

Advantages of RED

- Keeps queues smaller, while allowing bursts
 - Just using small buffers in routers can't do the latter
- Reduces synchronization between flows
 - Not all flows are dropping packets at once
 - Increases/decreases are more gentle
- Problem
 - Turns out that RED does not guarantee fairness

(2) Non-Congestion-Related Losses?

- For instance, RED drops packets intentionally
 - TCP would think the network is congested
- Can use **Explicit Congestion Notification (ECN)**
- Bit in IP packet header (actually two)
 - TCP receiver returns this bit in ACK
- When RED router would drop, it sets bit instead
 - Congestion semantics of bit exactly like that of drop
- Advantages
 - Doesn't confuse corruption with congestion

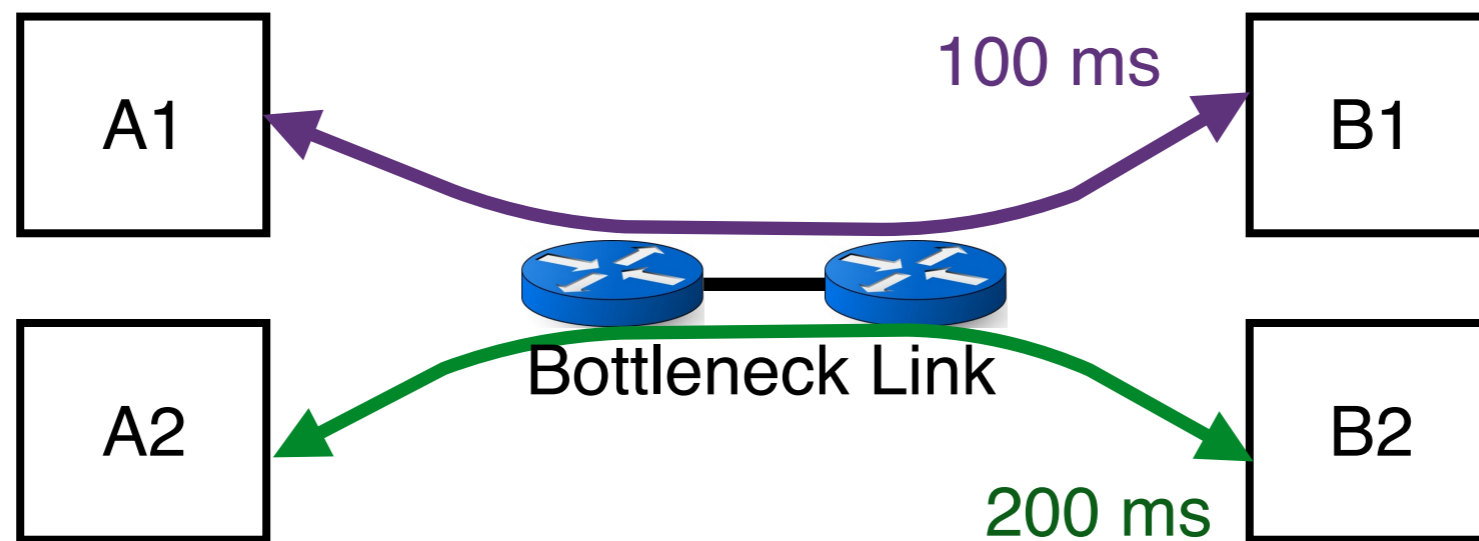
(3) Sawtooth Behavior Uneven

- TCP throughput is “choppy”
 - Repeated swings between $W/2$ to W
- Some apps would prefer sending at a steady rate
 - E.g., streaming apps
- A solution: “Equation-based congestion control”
 - Ditch TCP’s increase/decrease rules and just follow the equation:
 - **[Matthew Mathis, 1997] TCP Throughput = $MSS/RTT \sqrt{3/2p}$**
 - **Where p is drop rate**
 - Measure drop percentage p and set rate accordingly
- Following the TCP equation ensures we’re TCP friendly
 - I.e., use no more than TCP does in similar setting

Any Questions?

(4) Bias Against Long RTTs

- Flows get throughput inversely proportional to RTT
- **TCP unfair in the face of heterogeneous RTTs!**
- [Matthew Mathis, 1997] TCP Throughput = $MSS/RTT \sqrt{3/2p}$
 - Where p is drop rate
- Flows with long RTT will achieve lower throughput



Possible Solutions

- Make additive constant proportional to RTT
- But people don't really care about this...

(5) How Short Flows Fare?

- Internet traffic:
 - Elephant and mice flows
 - Elephant flows carry most bytes (>95%), but are very few (<5%)
 - Mice flows carry very few bytes, but most flows are mice
 - 50% of flows have < 1500B to send (1 MTU);
 - 80% of flows have < 100KB to send
- Problem with TCP?
 - Mice flows do not have enough packets for duplicate ACKs!!
 - Drop \approx Timeout (unnecessary high latency)
 - These are precisely the flows for which latency matters!!!
- Another problem:
 - Starting with small window size leads to high latency

Possible Solutions?

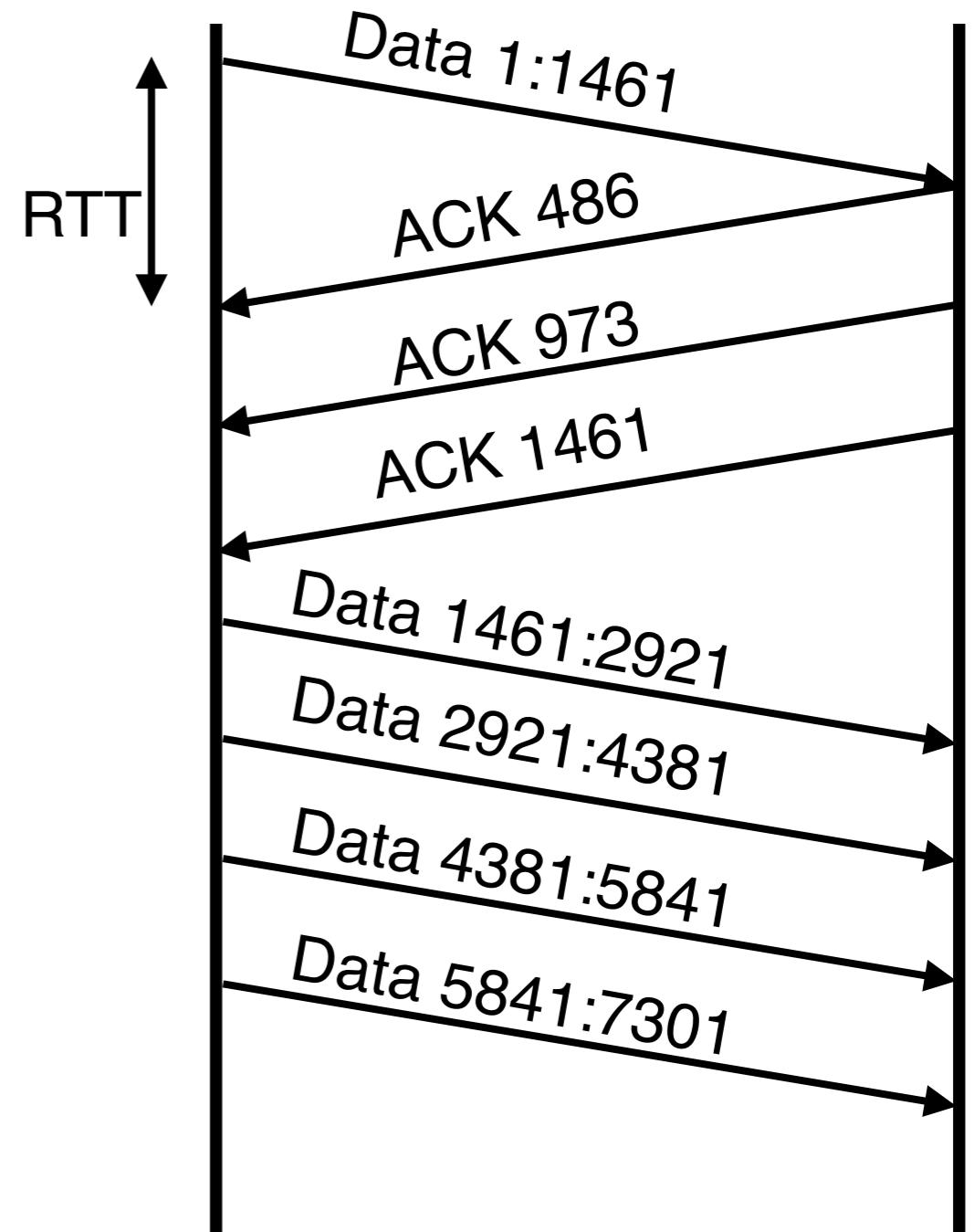
- Larger initial window?
 - Google proposed moving from ~4KB to ~15KB
 - Covers ~90% of HTTP Web
 - Decreases delay by 5%
- Many recent research papers on the timeout problem
 - Require network support

(6) Cheating

- TCP was designed assuming a cooperative world
- No attempt was made to prevent cheating
- Many ways to cheat, will present three

Cheating #1: ACK-splitting (receiver)

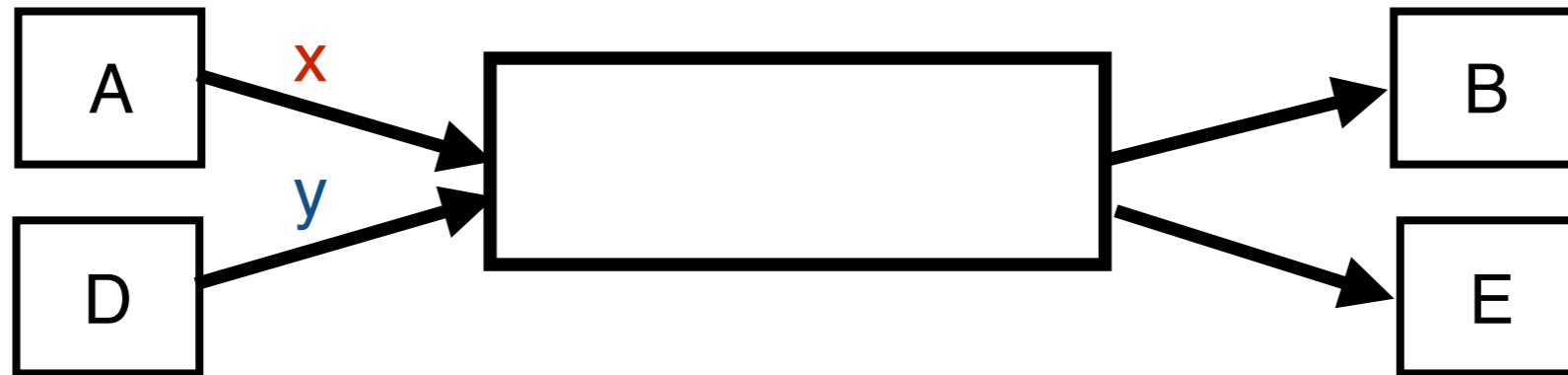
- TCP Rule: grow window by one MSS for each valid ACK received
- Send **M** (distinct) ACKs for one MSS
- Growth factor proportional to **M**



Cheating #2: Increasing CWND Faster (source)

- TCP Rule: increase window by one MSS for each valid ACK received
- Increase window by **M** per ACK
- Growth factor proportional to **M**

Cheating #3: Open Many Connections (source/receiver)



- Assume
 - A start 10 connections to B
 - D starts 1 connection to E
 - Each connection gets about the same throughput
- Then A gets 10 times more throughput than D

Cheating

- Either sender or receiver can independently cheat!
- **Why hasn't Internet suffered congestion collapse yet?**
 - Individuals don't hack TCP (not worth it)
 - Companies need to avoid TCP wars
- How can we prevent cheating
 - Verify TCP implementations
 - Controlling end points is hopeless
- Nobody cares, really

Any Questions?

How Do You Solve These Problems?

- Bias against long RTTs
- Slow to ramp up (for short-flows)
- Cheating
- Need for uniformity

Get The Network Involved!

- How can routers help?
- Routers can provide guidance for speed to send at
 - Routers have better visibility, and their feedback flows through the network, collecting from routers along path
- Routers can provide isolation/fairness
 - Via packet scheduling
- Routers can provide priority for some flows
 - Via packet scheduling

You now know Computer Networks :-)

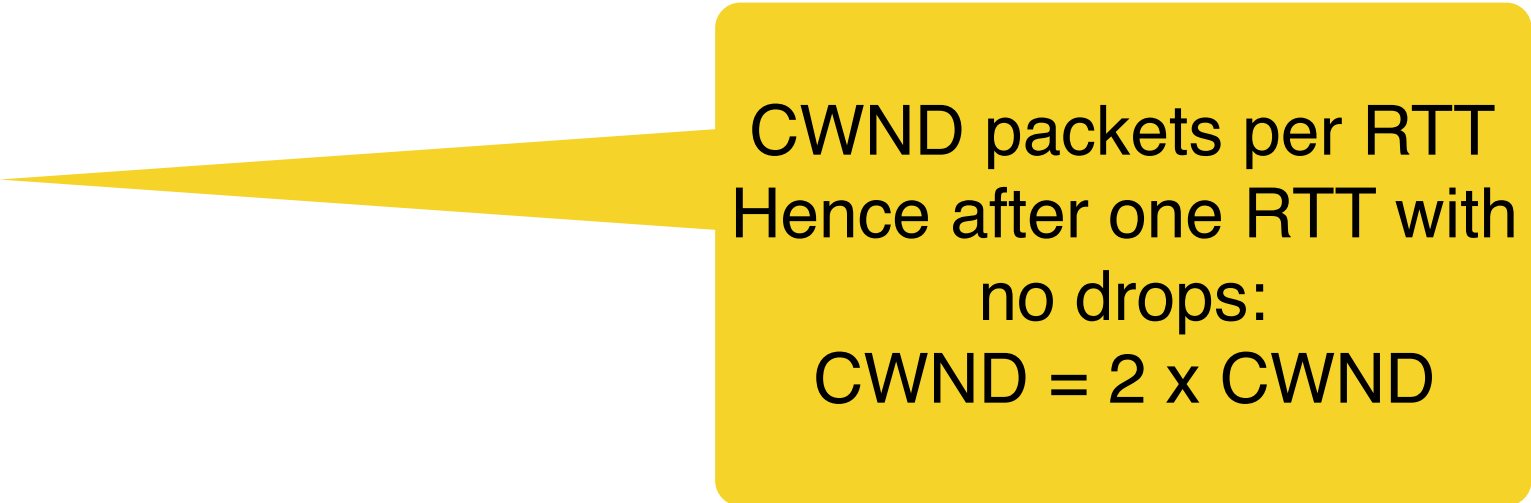
TCP Congestion Control Details

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
 - Assuming $RWND > CWND$

Event: ACK (new data)

- If in slow start
 - $CWND += 1$

A yellow callout box with a pointer pointing to the 'CWND += 1' bullet point. The text inside the box explains the consequence of this action in slow start.

CWND packets per RTT
Hence after one RTT with
no drops:
 $CWND = 2 \times CWND$

Event: ACK (new data)

- If $CWND \leq ssthresh$
 - $CWND += 1$
- 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
 - $ssthresh \leftarrow CWND/2$
 - $CWND \leftarrow 1$

Event: dupACK

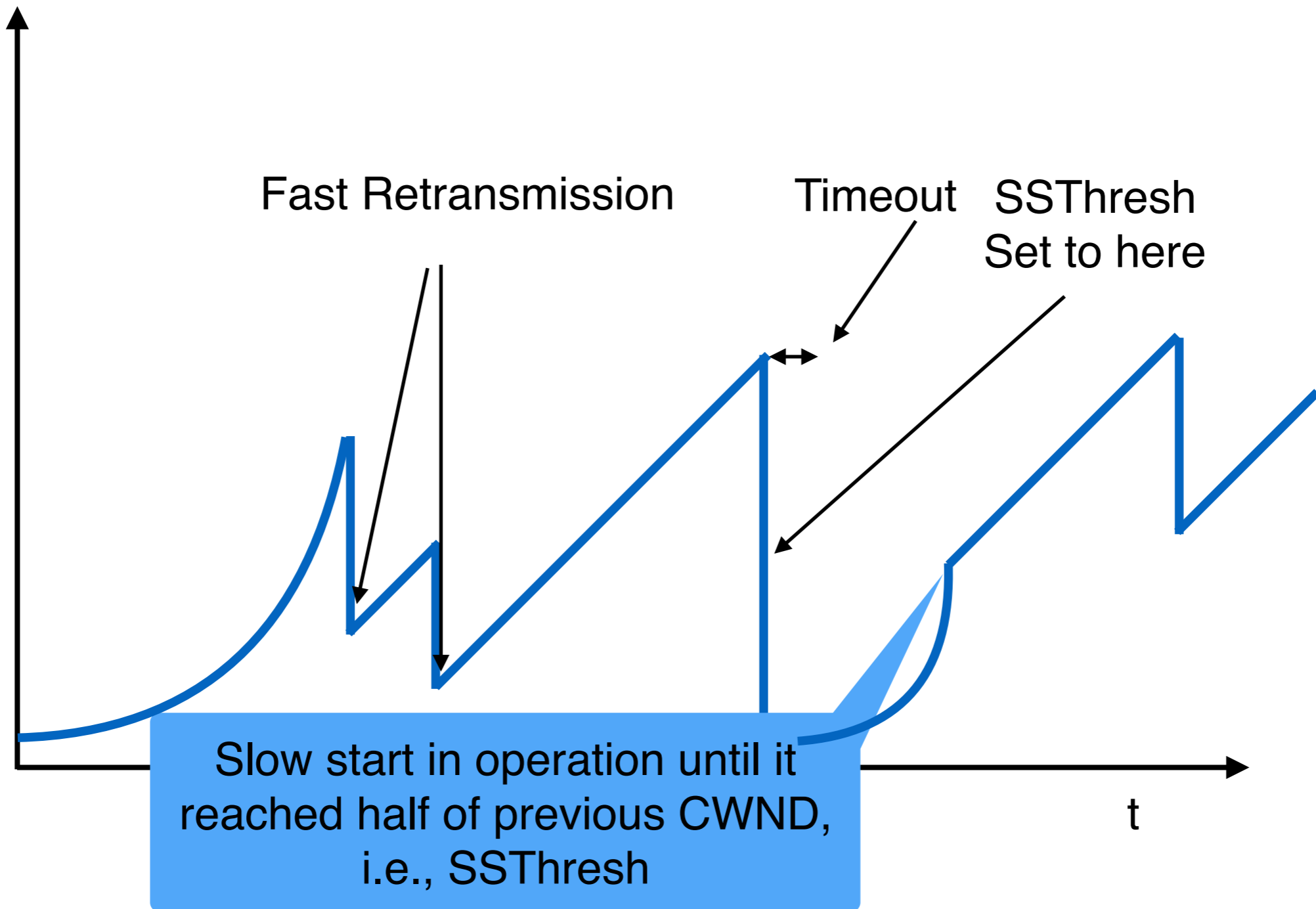
- dupACKcount++
- If dupACKcount = 3 /* fast retransmit */
 - ssthresh <- CWND/2
 - CWND <- CWND/2



Remains in congestion avoidance after fast retransmission

Time Diagram

Window



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