CS4450

Computer Networks:
Architecture and Protocols

Lecture 24
Transport Layer Wrap Up

Rachit Agarwal

Recap

Recap: Four Goals for Reliable Transfer

- **Correctness**
    - As defined
- **“Fairness”**
    - Every flow must get a fair share of network resources
- **Flow Performance**
    - Latency, jitter, etc.
- **Utilization**
    - Would like to maximize bandwidth utilization
    - If network has bandwidth available, flows should be able to use it!

Recap: Complete Correctness Condition

A transport mechanism is "reliable" if and only if

(a) It resends all dropped or corrupted packets

(b) It attempts to make progress

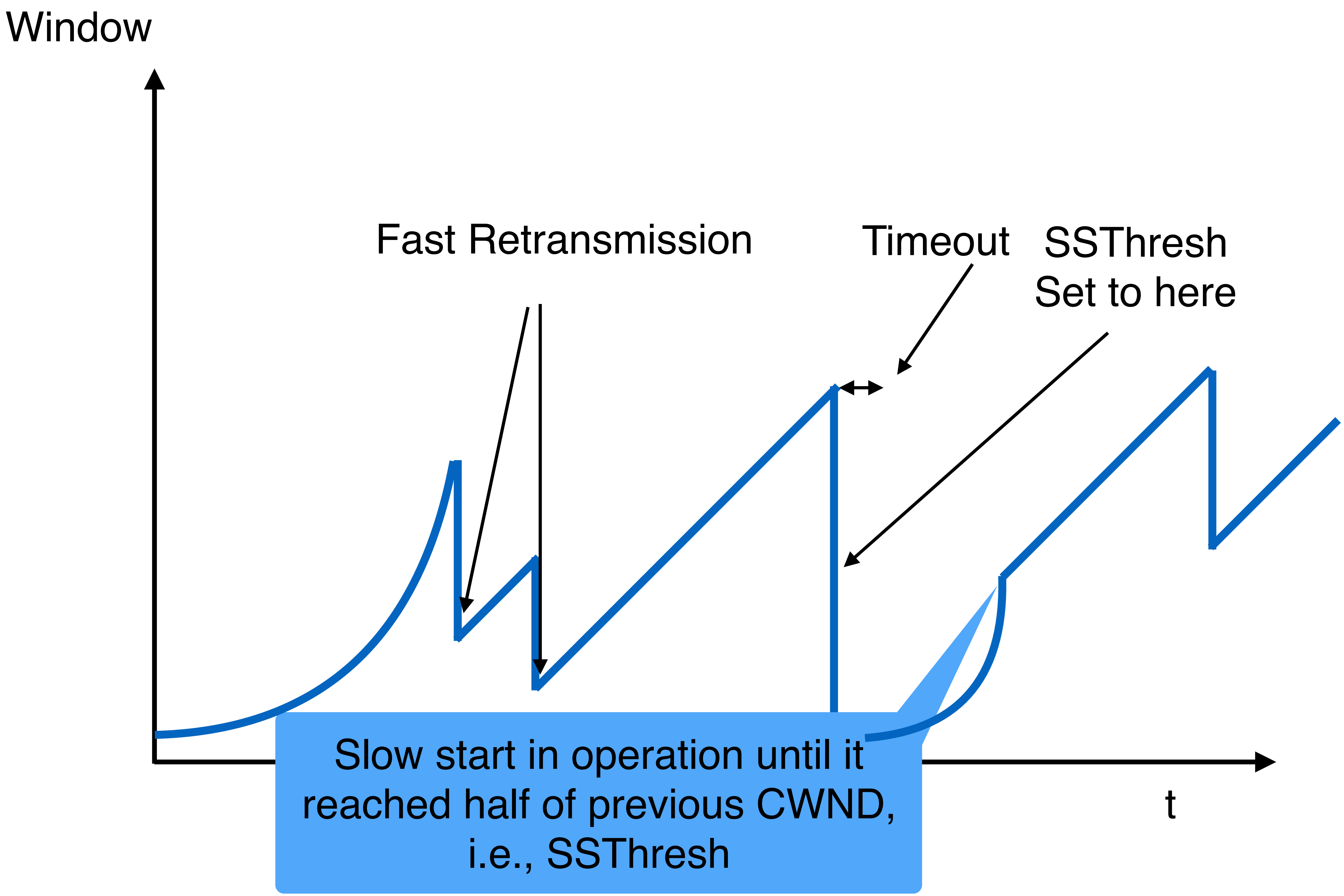
Recap: WHYs behind Transport design

- Started from first principles
    - Correctness condition for reliable transport
- … to understanding **why feedback from receiver is necessary** (sol-v1)
- … to understanding **why timers may be needed** (sol-v2)
- … to understanding **why window-based design may be needed** (sol-v3)
- … to understanding **why cumulative ACKs may be a good idea**
    - Very close to modern TCP

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: Final Time Diagram



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

Recap Classical result:

In a stable state

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

TCP guarantees max-min fairness

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss
- This last feature improves performance, but is not conceptually important

Example

- Consider a TCP connection with:
  - CWND = 10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq no 101
- 10 packets [101, 102, 103, …, 110] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

Timeline

- ACK 101(due to 102) CWND = 10 dupACK #1 (no xmit)
- ACK 101(due to 103) CWND = 10 dupACK #2 (no xmit)
- ACK 101(due to 104) CWND = 10 dupACK #3 (no xmit)
- RETRANSMIT 101 ssthresh = 5 CWND = 5
- ACK 101 (due to 105) CWND=5 (no xmit)
- ACK 101 (due to 106) CWND=5 (no xmit)
- ACK 101 (due to 107) CWND=5 (no xmit)
- ACK 101 (due to 108) CWND=5 (no xmit)
- ACK 101 (due to 109) CWND=5 (no xmit)
- ACK 101 (due to 110) CWND=5 (no xmit)
- ACK 111 (due to 101)<- only now can we transmit new packets
- Plus no packets in flight so no ACKs for another RTT

Note that you do not restart dupACKcounter on same packet!

Solution: Fast Recovery

- Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight (each ACK due to arriving pkt)
- If dupACKcount = 3
    - ssthresh = CWND / 2
    - CWND = ssthresh + 3
- While in fast recovery
    - CWND = CWND + 1 for each additional duplicate ACK
- Exit fast recovery after receiving **new** ACK
    - Set CWND = ssthresh (which had been set to CWND/2 after loss)

Example

- Consider a TCP connection with:
  - CWND = 10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq no 101

- 10 packets [101, 102, 103, …, 110] are in flight
  - Packet 101 is dropped

Timeline

- ACK 101(due to 102) CWND = 10 dupACK #1 (no xmit)
- ACK 101(due to 103) CWND = 10 dupACK #2 (no xmit)
- ACK 101(due to 104) CWND = 10 dupACK #3 (no xmit)
- RETRANSMIT 101 ssthresh = 5 CWND = 8 (5 + 3)
- ACK 101 (due to 105) CWND=9 (no xmit)
- ACK 101 (due to 106) CWND=10 (no xmit)
- ACK 101 (due to 107) CWND=11 (xmit 111)
- ACK 101 (due to 108) CWND=12 (xmit 112)
- ACK 101 (due to 109) CWND=13 (xmit 113)
- ACK 101 (due to 110) CWND=14 (xmit 114)
- ACK 111 (due to 101) CWND = 5 (xmit 115) <- exiting fast recovery
- Packets 111-114 already in flight (and now sending 115)
- ACK 112 (due to 111) CWND = 5 + 1/5 <- back to congestion avoidance

TCP “Phases”

- Slow-start
  - Enter on timeout
  - Leave when CWND > ssthresh (to Cong. Avoid.)
    - The > only applies here…
- Congestion Avoidance
  - Leave when timeout
- Fast recovery
  - Enter when dupACK=3
  - Leave when New ACK or Timeout

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

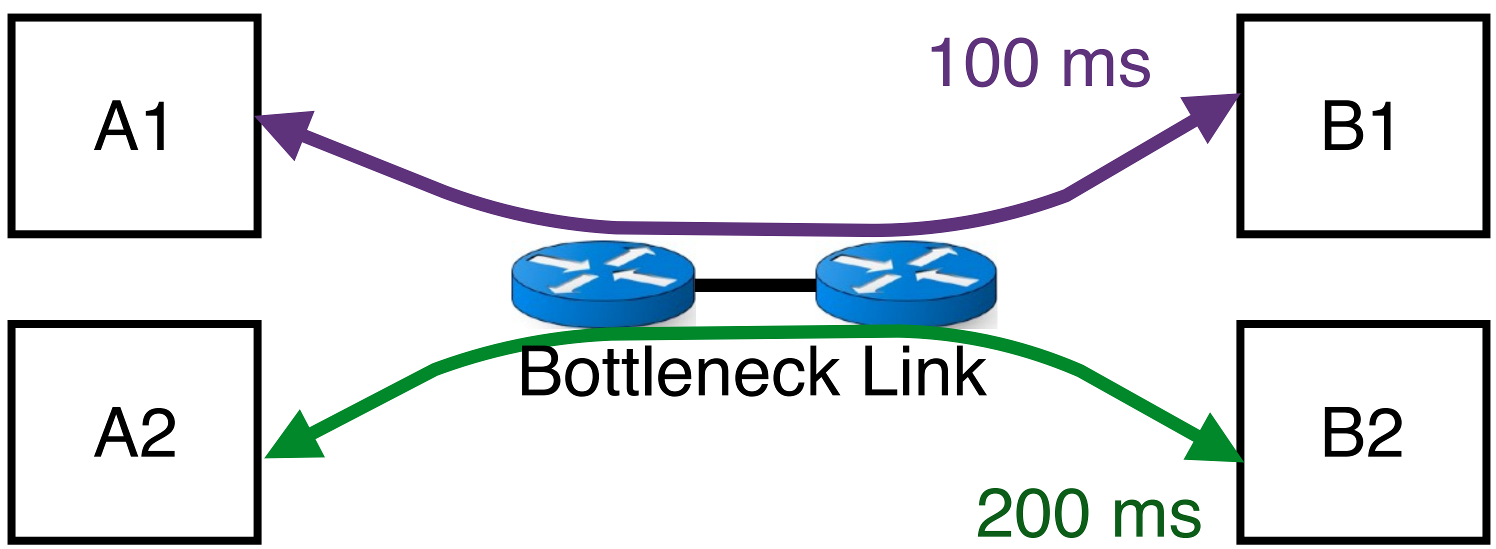
(3) Sawtooth Behavior Uneven

- 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 ~=~ 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?

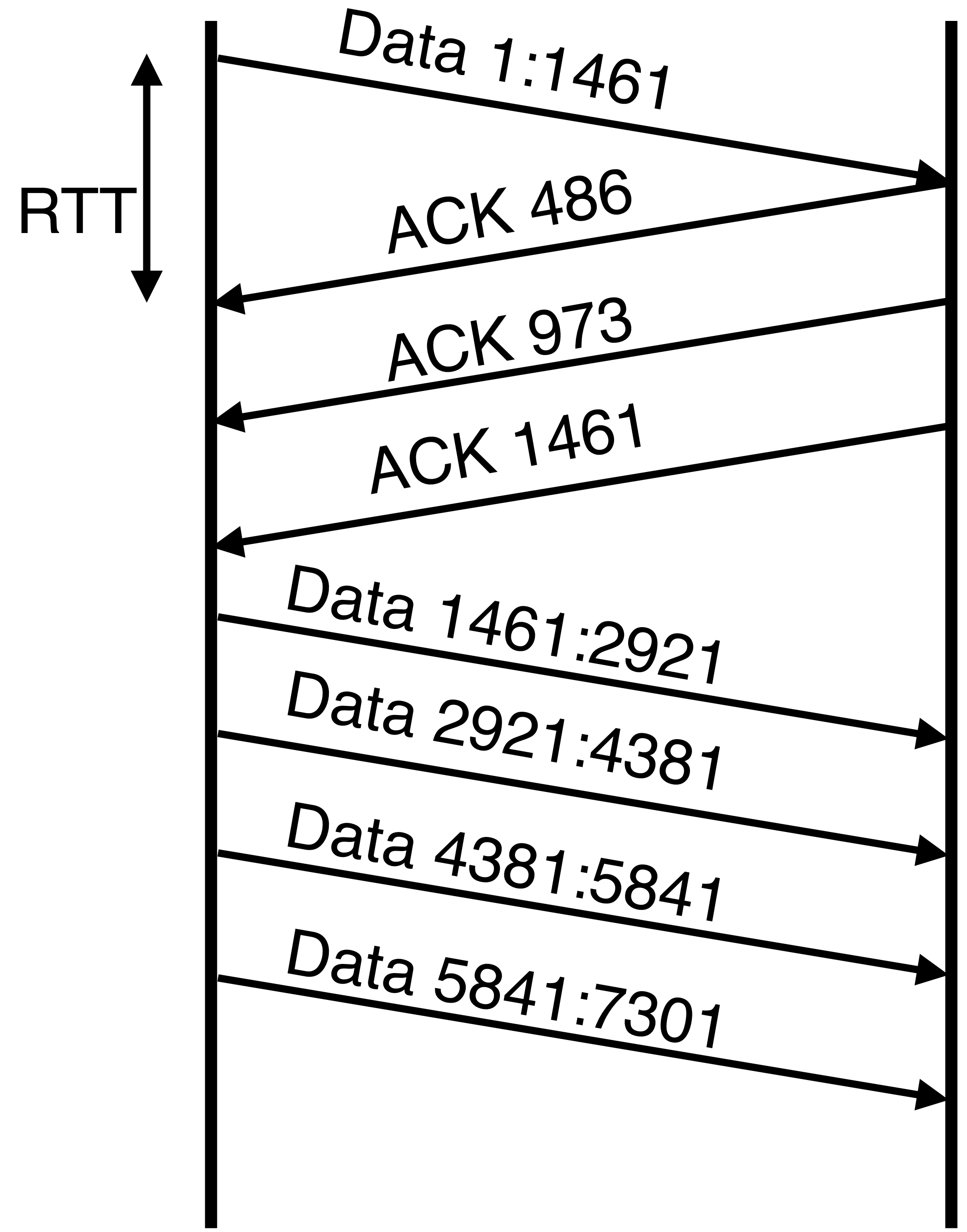
- Larger initial window?
    - Google proposed moving from ~4KB to ~15KB
    - Covers ~90% of HTTP Web
    - Decreases delay by 5%
- Many recent research papers on the timeout problem
    - Require network support

(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)

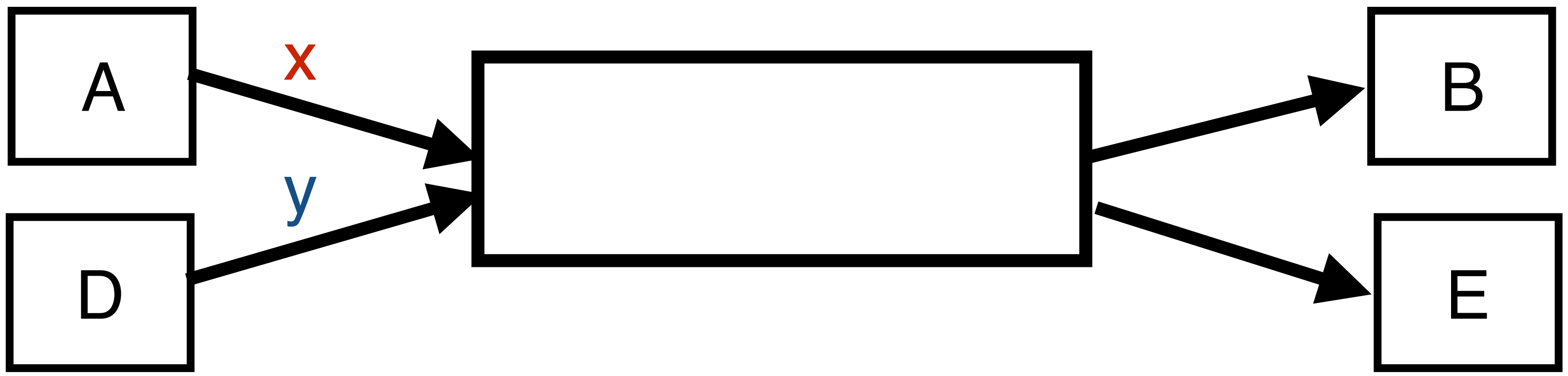
- TCP Rule: grow window by one MSS for each valid ACK received
- Send **M** (distinct) ACKs for one MSS
- Growth factor proportional to **M**



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