TCP Congestion Control

CS 168

http://cs168.io

Sylvia Ratnasamy

Last time:

- We narrowed our exploration of the design space to a CC solution that is based on:
  - Implemented only by end-hosts
  - Dynamic rate adjustment
  - Uses loss to detect congestion
- Today: TCP CC
  - An example of the above design

Plan

- Review TCP's window-based operation
- Extending the above for CC

Review:

- Sender maintains a window of packets in flight
- Window size **W** is picked to balance three goals
  - Take advantage of network capacity (“fill the pipe”)
  - Avoid overloading the receiver (flow control)
  - Avoid overloading links (congestion control)

Review:

- Sender maintains a window of packets in flight

- Window size **W** is picked to balance three goals
  - Take advantage of network capacity (“fill the pipe”)
  - Avoid overloading the receiver (flow control)
  - Avoid overloading links (congestion control)

- Flow control: sender maintains an **advertised window**; also called a **receiver window (RWND)**

- CC: sender maintains a **congestion window (CWND)**

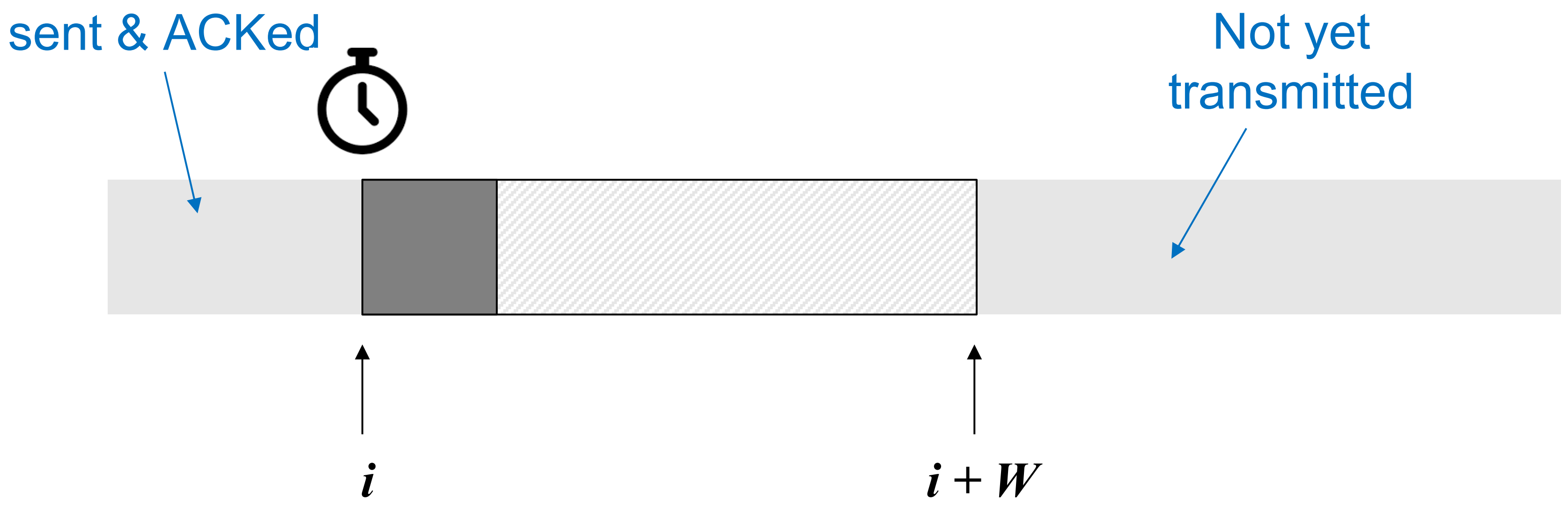
All These Windows…

- Congestion Window: **CWND**
  - How many bytes can be sent without overloading links
  - Computed by the sender using CC algorithm
- Flow control window: **RWND**
  - How many bytes can be sent without overflowing the receiver’s buffers
  - Implemented by having the receiver tell the sender
- **Sender-side window = min{CWND, RWND}**
  - Assume for this lecture that RWND > CWND

Note

- **Recall: TCP operates on bytestreams**
- **Hence, real implementations maintain CWND in bytes**
- **This lecture will talk about CWND in units of MSS**
  - MSS: Maximum Segment Size, the max number of bytes of data that one TCP packet can carry in its payload
  - This is only for pedagogical purposes

Review:

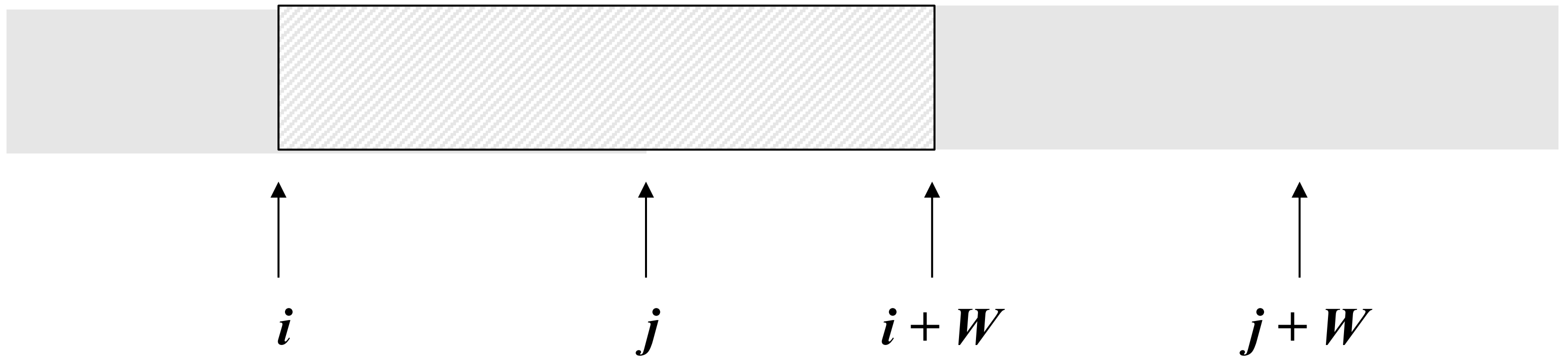


Sender maintains a **sliding** window of W **contiguous** bytes

Sender maintains a single timer, for the LHS of window

On timeout, sender retransmits the packet starting at i

Review:



Receiver sends cumulative ACKs; sender counts #dupACKs

Fast Retransmit: Sender retransmits when #dupACKs = 3

Sender slides window on receiving an ACK for new data ($j > i$)

Extending TCP with CC

- Add a congestion window parameter (CWND)
- Adapt CWND based on current congestion level
- How do we adapt CWND?
  - Last lecture: how sender adapts its transmission *rate*
- In TCP, sender's rate is simply CWND/RTT
  - (Since we're assuming RWND > CWND)
- Adapting CWND every RTT → adapting sender's rate

Recall: how we adapt rate

- Detecting congestion
  - **Loss-based**
- Discovering an initial rate
  - **Slow start**
- Adapting rate to congestion (or lack thereof)
  - **AIMD**

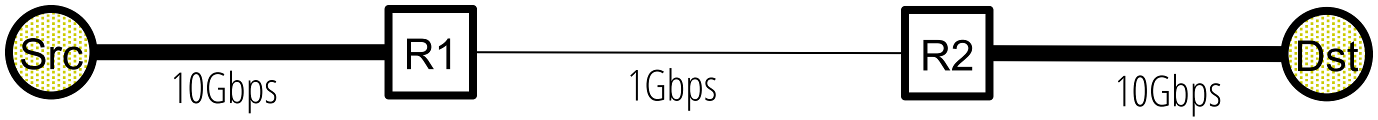
What follows is all about how TCP implements the above

Theme: CWND updates driven by ACK arrivals ("ACK clocking")

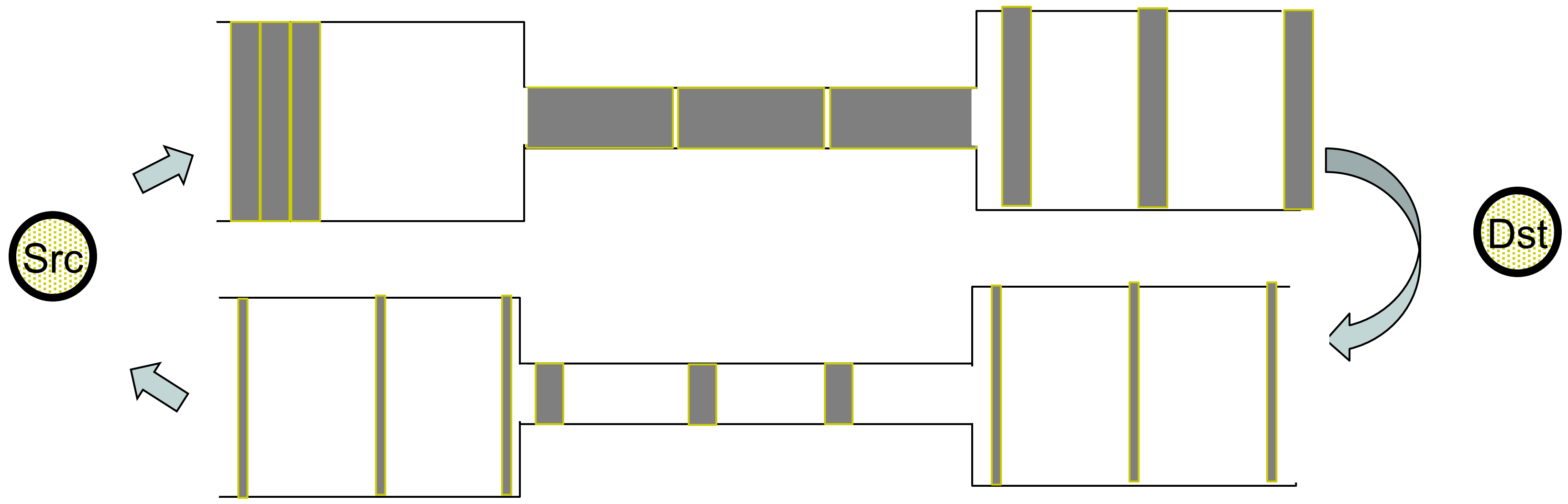
ACK Clocking

- A new ACK advances the sliding window and lets a new data segment enter the network
  - I.e., ACKs “clock” data segments
- What’s the benefit of ACK clocking?

ACK Clocking



ACK Clocking

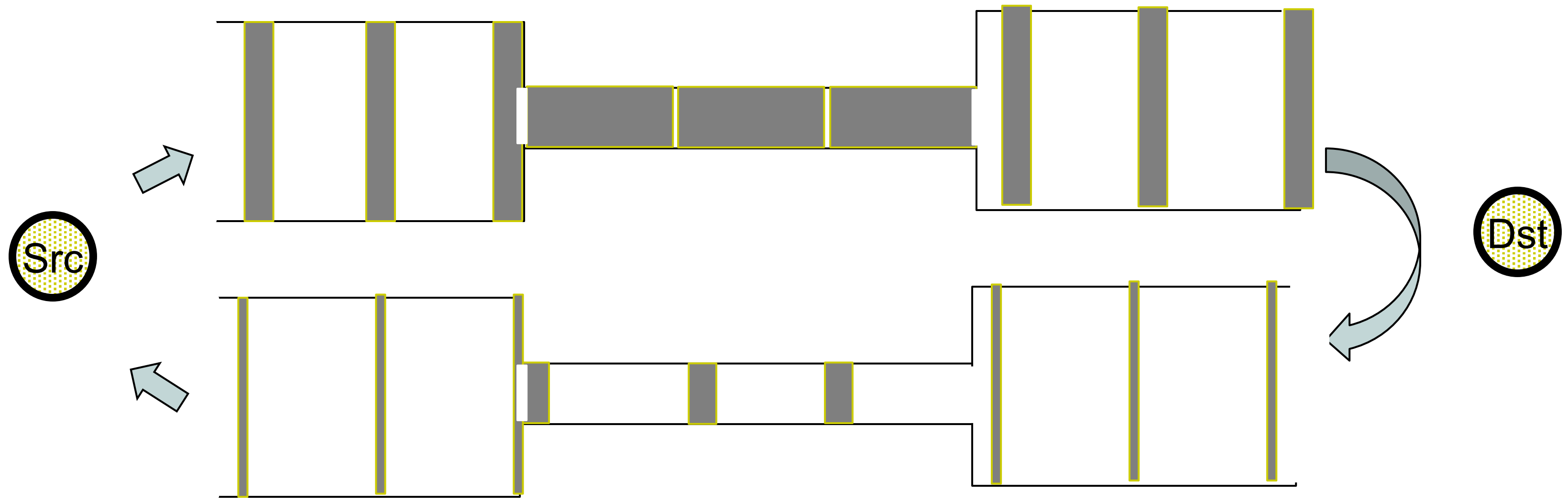


Consider: source sends a burst of packets

Packets are queued and "spread out" at slow link

ACKs maintain the spread on the return path

ACK Clocking



Sender clocks new packets with the spread

Now sending without queuing at the bottleneck link!

Recall: how we adapt rate

- Detecting congestion
  - **Loss-based**
- Discovering an initial rate
  - **Slow start**
- Adapting rate to congestion (or lack thereof)
  - **AIMD**

What follows is all about how TCP implements the above

Theme: CWND updates driven by ACK arrivals ("ACK clocking")

How TCP Detects Loss

- **3 duplicate ACKs**: typically indicates isolated loss
- **Timeout**: typically indicates loss of several packets

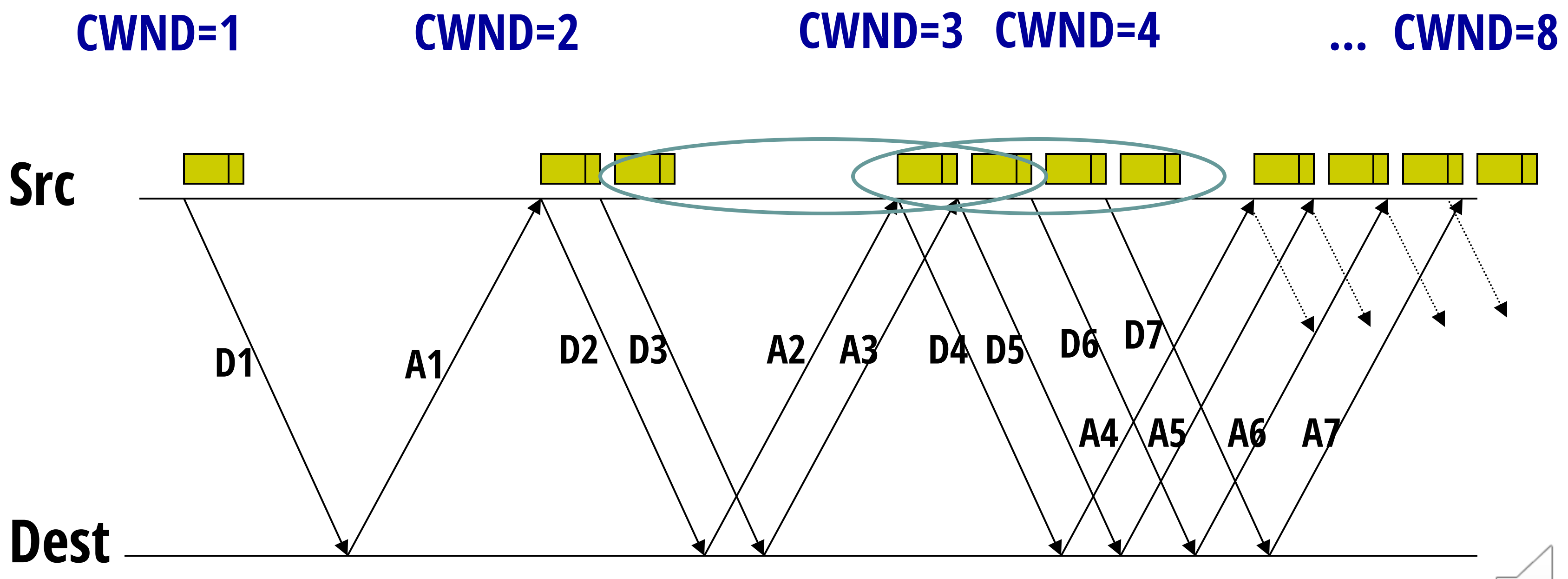
How TCP Implements Slow Start

- Sender starts at a slow rate; increases rate **exponentially** until first loss
- In TCP: start with a small CWND = 1 (MSS)
  - So, initial sending rate is MSS/RTT
- Then double CWND every RTT until first loss
- Implemented as: On each ACK: CWND += 1 (MSS)

Slow Start in Action

Goal: Double CWND every round-trip time

Simple implementation: On each ACK, CWND += 1 (MSS)



How TCP Implements Slow Start (contd.)

- Double CWND every RTT until first loss
- Introduce a “slow start threshold” parameter
  - SSTHRESH, used to remember last “safe” rate
- On first loss: SSTHRESH = CWND/2

Recall: how we adapt rate

- Detecting congestion
  - Loss-based
- Discovering an initial rate
  - Slow start
- Adapting rate to congestion (or lack thereof)
  - AIMD

AIMD in TCP

- Additive increase:
  - No loss → increase CWND by **1 MSS every RTT**
- Multiplicative decrease
  - Loss detected by 3 dupACKs → divide CWND in **half**
- What about timeouts? Will **exit AIMD** (coming up)

Implementing Additive Increase

- Implementation works by adding a fraction of an MSS every time we receive an ACK

- On receiving an ACK (for new data)
  - $CWND \rightarrow CWND + \frac{1}{CWND}$
  - $CWND \rightarrow CWND + MSS \times \frac{MSS}{CWND}$ *if counting CWND in bytes*

- NOTE: after full window, CWND increases by 1 MSS
  - Thus, CWND increases by 1 MSS per RTT

Implementing Multiplicative Decrease

- On receiving 3rd dupACK:
  - $CWND \rightarrow \frac{CWND}{2}$

On Timeout

- Rationale: lost multiple packets in a window
  - Current CWND may be way off
  - Hence, need to rediscover a good rate from scratch
  - Design decision that errs on the side of caution

- Hence, on timeout:
  - Retransmit first missing packet (as usual)
  - Set SSTHRESH $\leftarrow \frac{CWND}{2}$
  - Set CWND $\leftarrow$ 1 MSS & enter Slow Start mode

Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Determined by SSTHRESH
- When CWND > SSTHRESH, sender switches from slow-start to AIMD’s additive increase

Summary of Decrease

- Cut CWND in **half** on loss detected by dupACKs
- Cut CWND **all the way to 1 (MSS)** on timeout
- Never drop CWND below 1 (MSS)

Summary of Increase

- When in Slow-Start phase
  - Increase CWND by 1 MSS for each new ack
- When in AIMD phase
  - Increase by 1 (MSS) for each window’s worth of acked data

TCP Congestion Control Details

In what follows refer to CWND in units of MSS

Implementation

- **State at sender**
  - CWND (initialized to a 1 MSS)
  - SSTHRESH (initialized to a large constant)
  - dupACKcount (initialized to zero, as before)
  - Timer (as before)
- **Events at sender**
  - ACK (for new data)
  - dupACK (duplicate ACK for old data)
  - Timeout
- What about receiver?
  - Just send ACKs like before

Event: ACK (new data)

- If in slow start
  - CWND += 1 (MSS)

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*

  CWND = 2xCWND

Event: ACK (new data)

- If in slow start
  - CWND += 1 (MSS)

Slow start phase

- Else
  - CWND = CWND + 1/CWND

- CWND packets per RTT
- Hence after one RTT with no drops:

  CWND = CWND + 1

“Congestion Avoidance” phase (additive increase)

- Plus the usual ...
  - Reset timer, dupACKcount
  - Send new data packets (if CWND allows)

Event: TimeOut

- On Timeout
  - SSTHRESH ← CWND/2
  - CWND ← 1
  - And retransmit packet (as always)

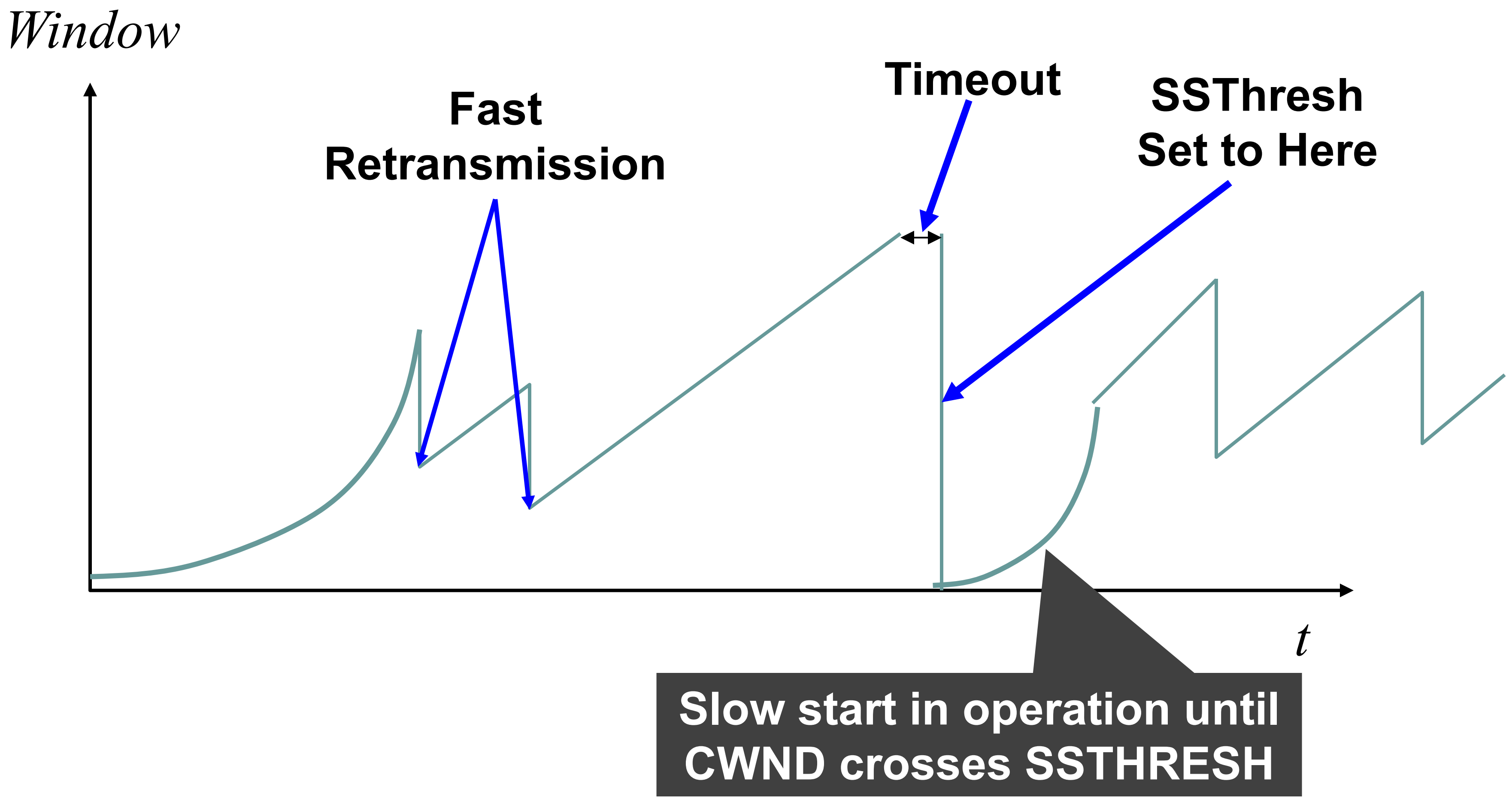
Event: dupACK

- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
  - SSTHRESH = CWND/2
  - CWND = CWND/2 (but never less than 1)
  - And retransmit packet (as always)

Remain in AIMD
after fast retransmission…

Any Questions?

Time Diagram



One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss
- This last feature is an optimization to improve performance
    - Bit of a hack, but effective

Example

- Again: counting packets, not bytes
  - If you want example in bytes, assume MSS=1000 and add three zeros to all sequence numbers

- 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 (at sender)

In flight: ~~101~~, 102, 103, 104, 105, 106, 107, 108, 109, 110 101

- 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 ACK “clocking” stalls for another RTT

Note that you do not restart dupACK counter on same packet!

Two Questions

- Do you understand the problem?
  - Have to wait a long time before sending again
  - When you finally send, you have to send full window

- How would you fix it?

Solution: Fast Recovery

Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - SSTHRESH = CWND/2
  - CWND = SSTHRESH + 3
- While in fast recovery
  - CWND = CWND + 1 (MSS) for each additional duplicate ACK
  - This allows source to send an additional packet…
  - …to compensate for the packet that arrived (generating dupACK)
- Exit fast recovery after receiving new ACK
  - set CWND = SSTHRESH

Timeline (at sender)

In flight: ~~101~~, 102, 103, 104, 105, 106, 107, 108, 109, 110 101 111, 112, ...

- ACK 101 (due to 102) cwnd=10 dupACK#1
- ACK 101 (due to 103) cwnd=10 dupACK#2
- ACK 101 (due to 104) cwnd=10 dupACK#3
- REXMIT 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 in congestion avoidance

Updated Event-Actions

Event: ACK (new data)

- If in slow start
  - CWND += 1 (MSS)

Slow start phase

- If in fast recovery
  - CWND = SSTHRESH

Leaving Fast Recovery

- Else
  - CWND = CWND + 1/CWND

"Congestion Avoidance" phase (additive increase)

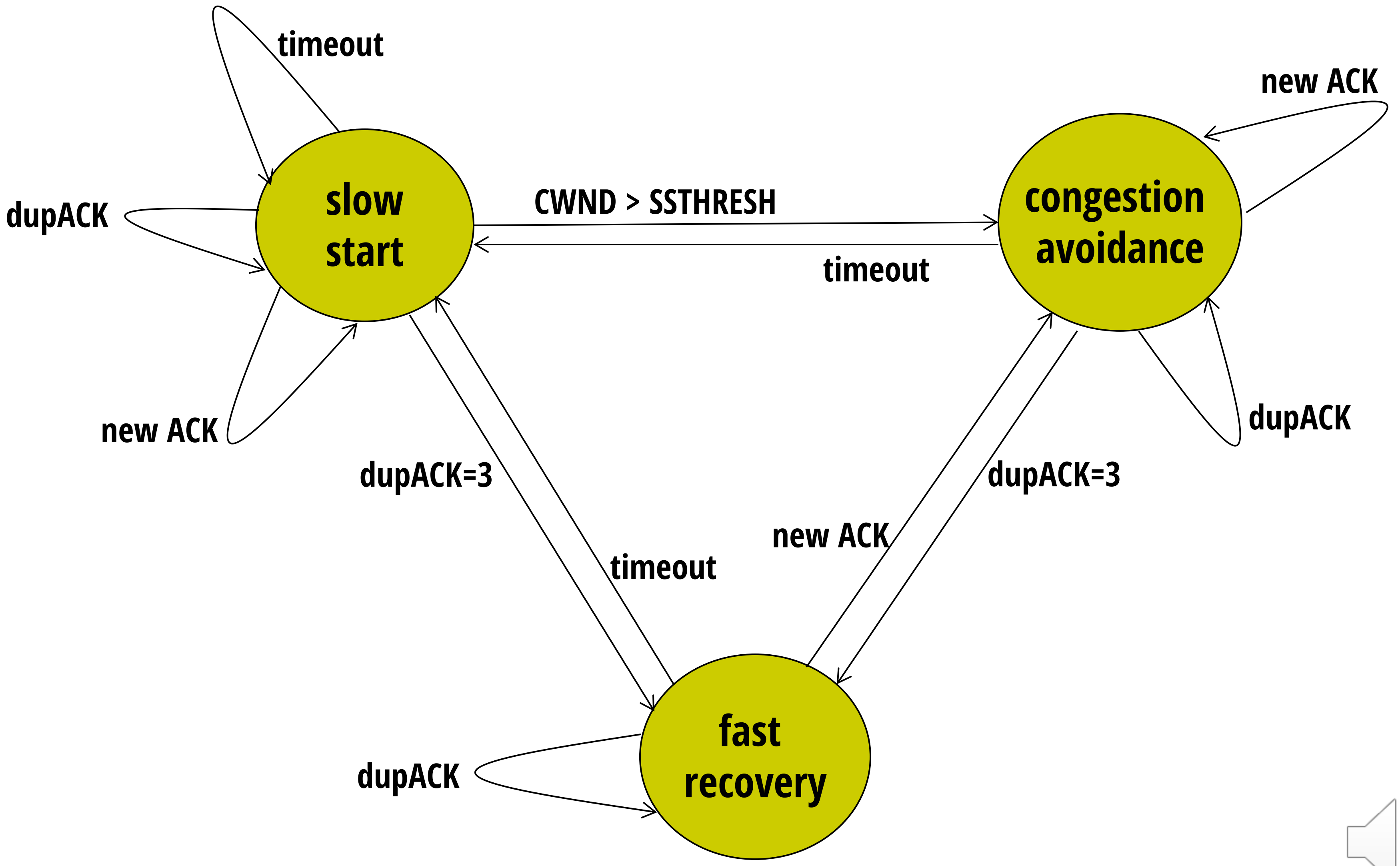
- Plus the usual...

Event: dupACK

- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
  - ssthresh = CWND/2
  - CWND = CWND/2 +3
  - And retransmit packet
- If dupACKcount > 3 /* fast recovery */
  - CWND = CWND + 1 (MSS)

Next: TCP State Machine

TCP State Machine



Many variants

- 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”
  - ACKs describe byte ranges received

Our default assumption

Interoperability

- How can all these algorithms coexist? Don’t we need a single, uniform standard?
- What happens if I’m using Reno and you are using Tahoe, and we try to communicate?
- What happens if I’m using Tahoe and you are using SACK?

Next Lecture

- Modeling TCP
- Advanced congestion control techniques