Project 2: Traceroute

- Project 2 (Traceroute) is out
- Due Friday, March 22nd at 11:59 PM PST
- Project 2 is hard(er)
  - **Start Early**
  - Don’t expect a perfect score
- Ethan Jackson is the lead TA.
- See the website for his office hours.

TCP Congestion Control (contd.)

CS 168

http://cs168.io

Sylvia Ratnasamy

TCP Congestion Control (contd.)

CS 168

http://cs168.io

Sylvia Ratnasamy

Today

- The TCP state machine
- Modeling TCP throughput
- Critiquing TCP
- Router-assisted CC (briefly)

TCP Implementation

TCP Implementation

TCP 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)

TCP 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

TCP 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)

Event: ACK (new data)

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

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

Event: ACK (new data)

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

Slow start phase

Event: ACK (new data)

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

Slow start phase

- Else
  - CWND = CWND + 1/CWND

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

Event: ACK (new data)

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

Slow start phase

"Congestion Avoidance" phase (additive increase)

Event: ACK (new data)

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

Slow start phase

- Else
  - CWND = CWND + 1/CWND

"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: TimeOut

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

Event: dupACK

Event: dupACK

Event: dupACK

- dupACKcount ++

Event: dupACK

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

Event: dupACK

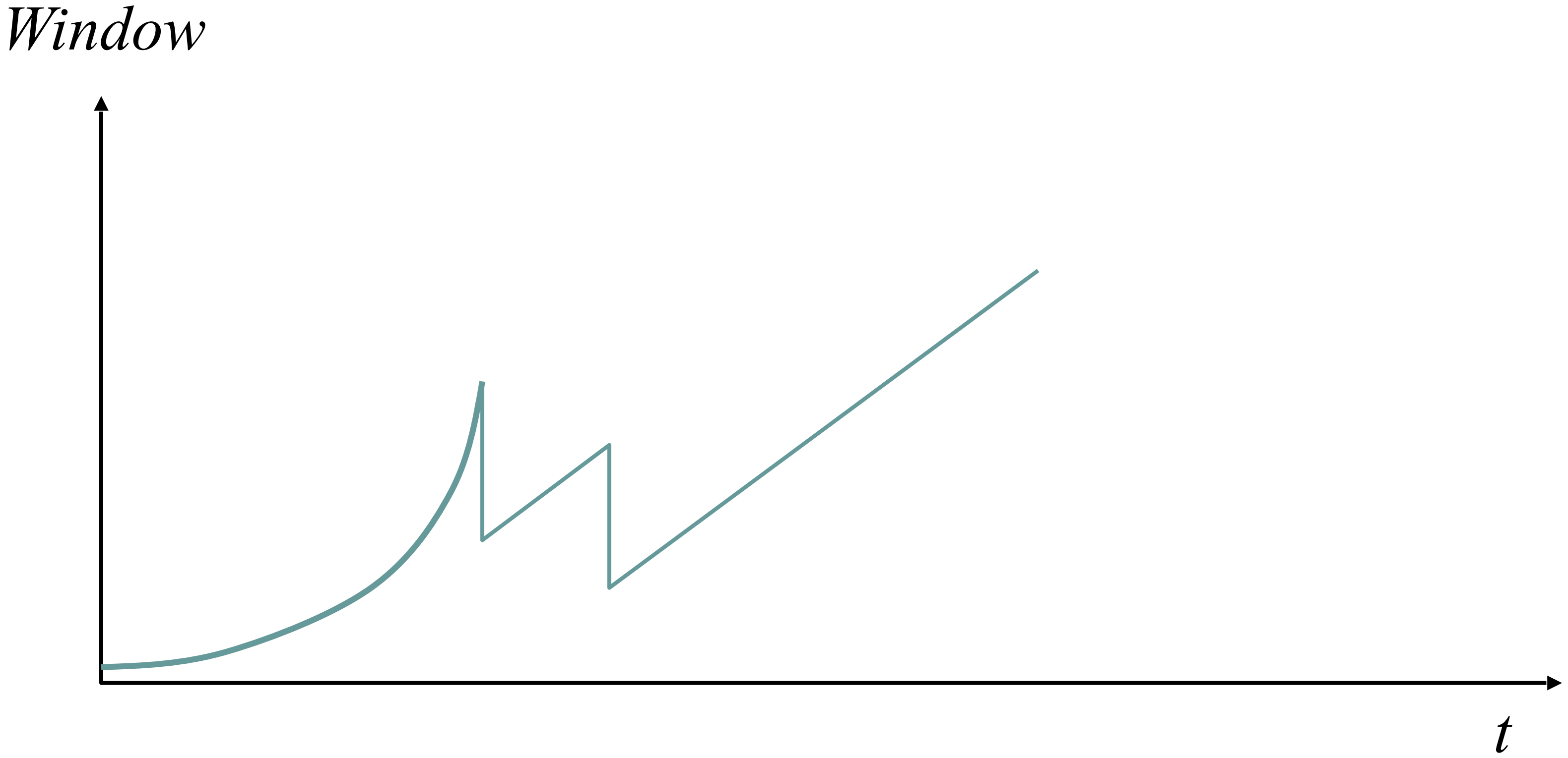
- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
  - SSTHRESH = CWND/2
  - CWND = CWND/2 (but never less than 1)
  - And retransmit packet (as always)

Remain in AIMD
after fast retransmission…

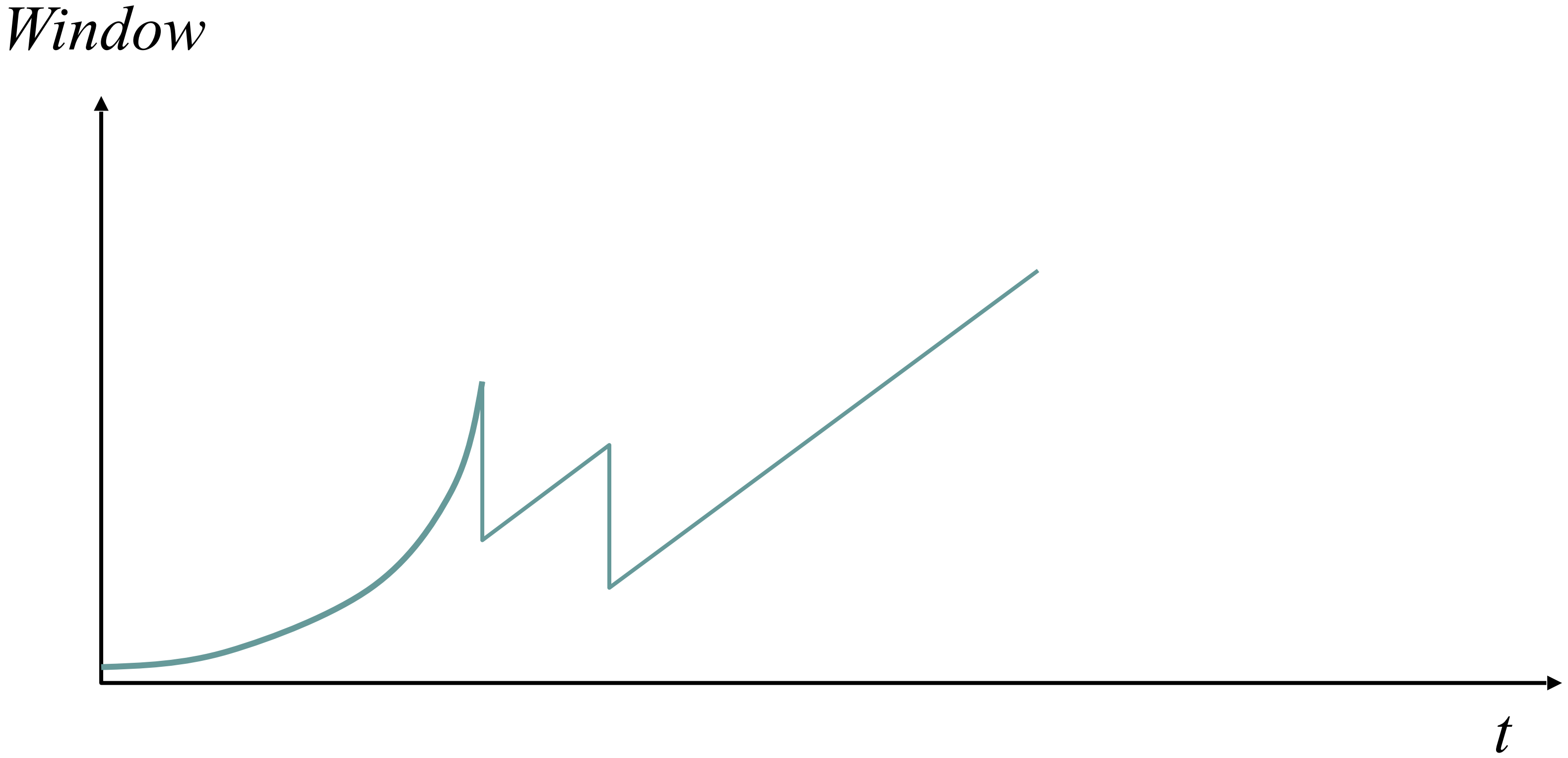
Any Questions?

Any Questions?

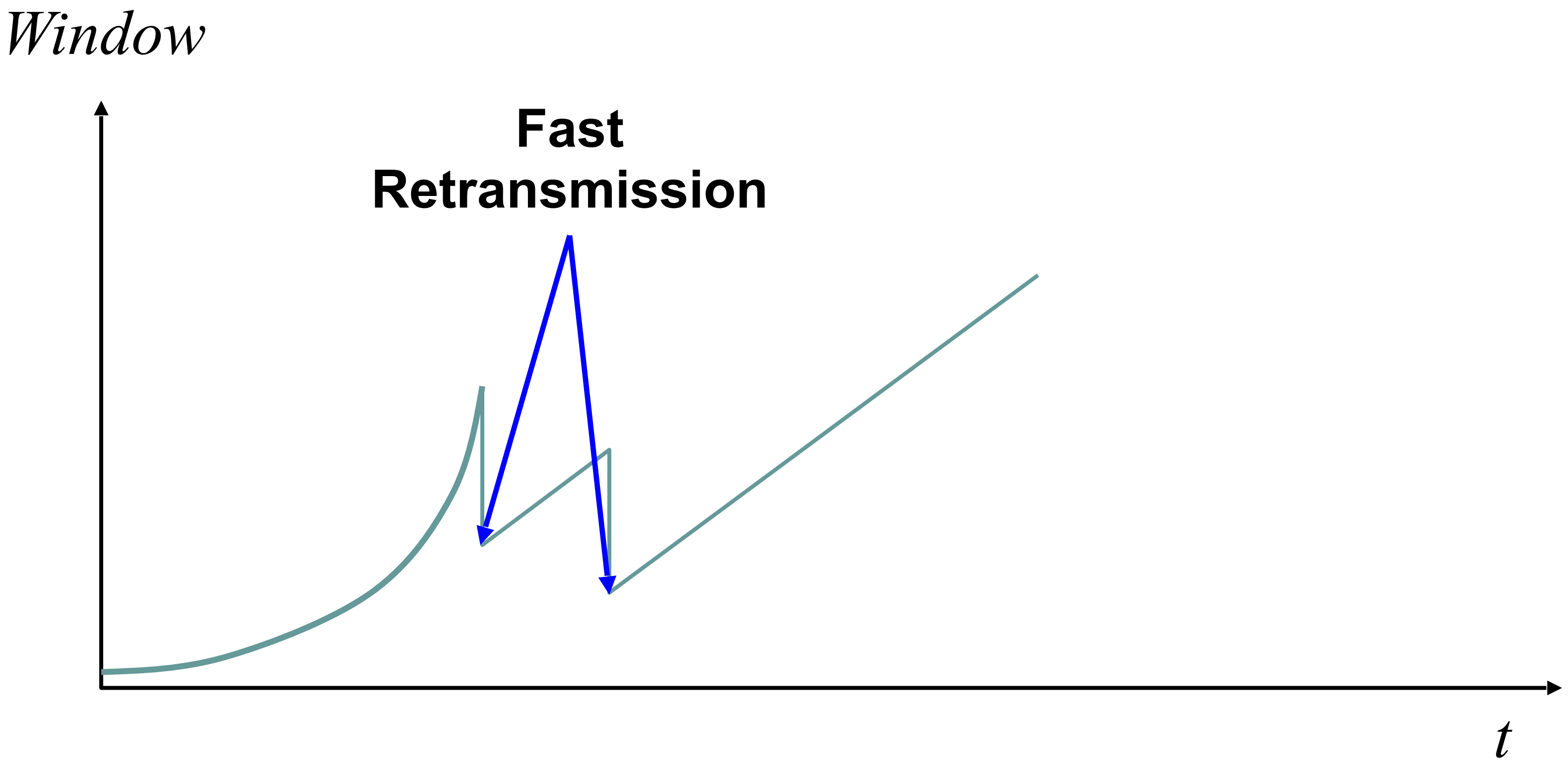
Time Diagram



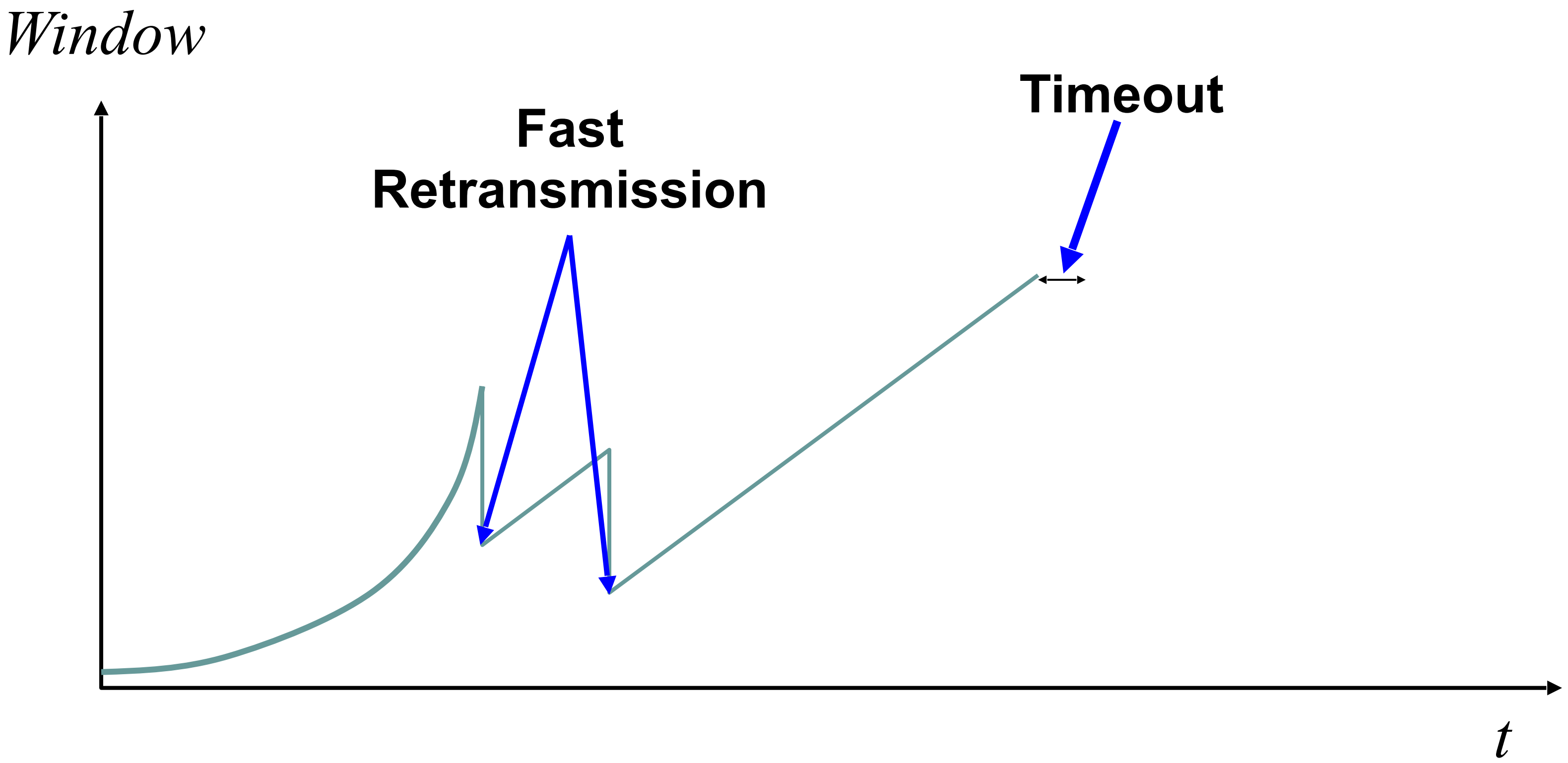
Time Diagram



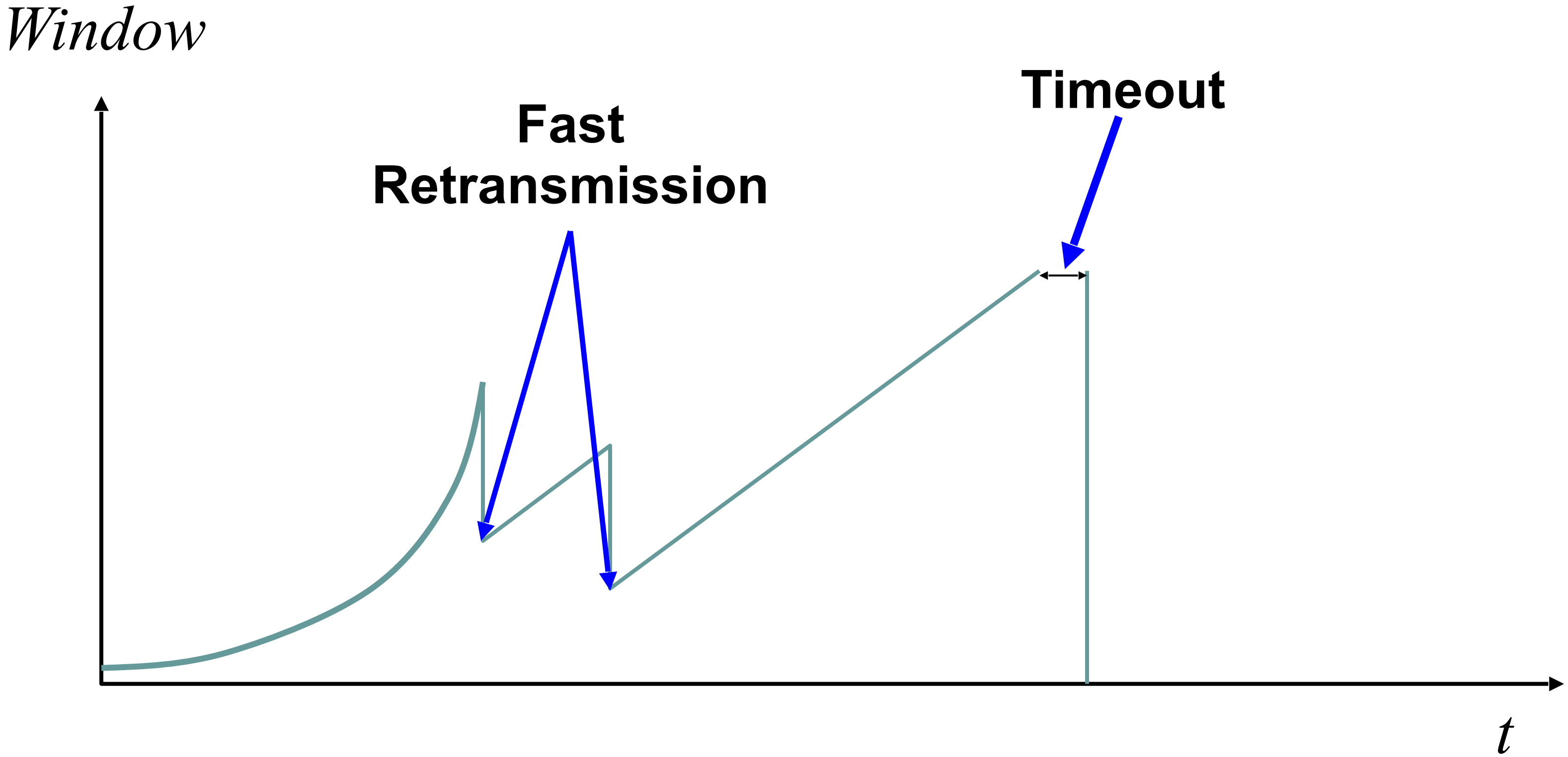
Time Diagram



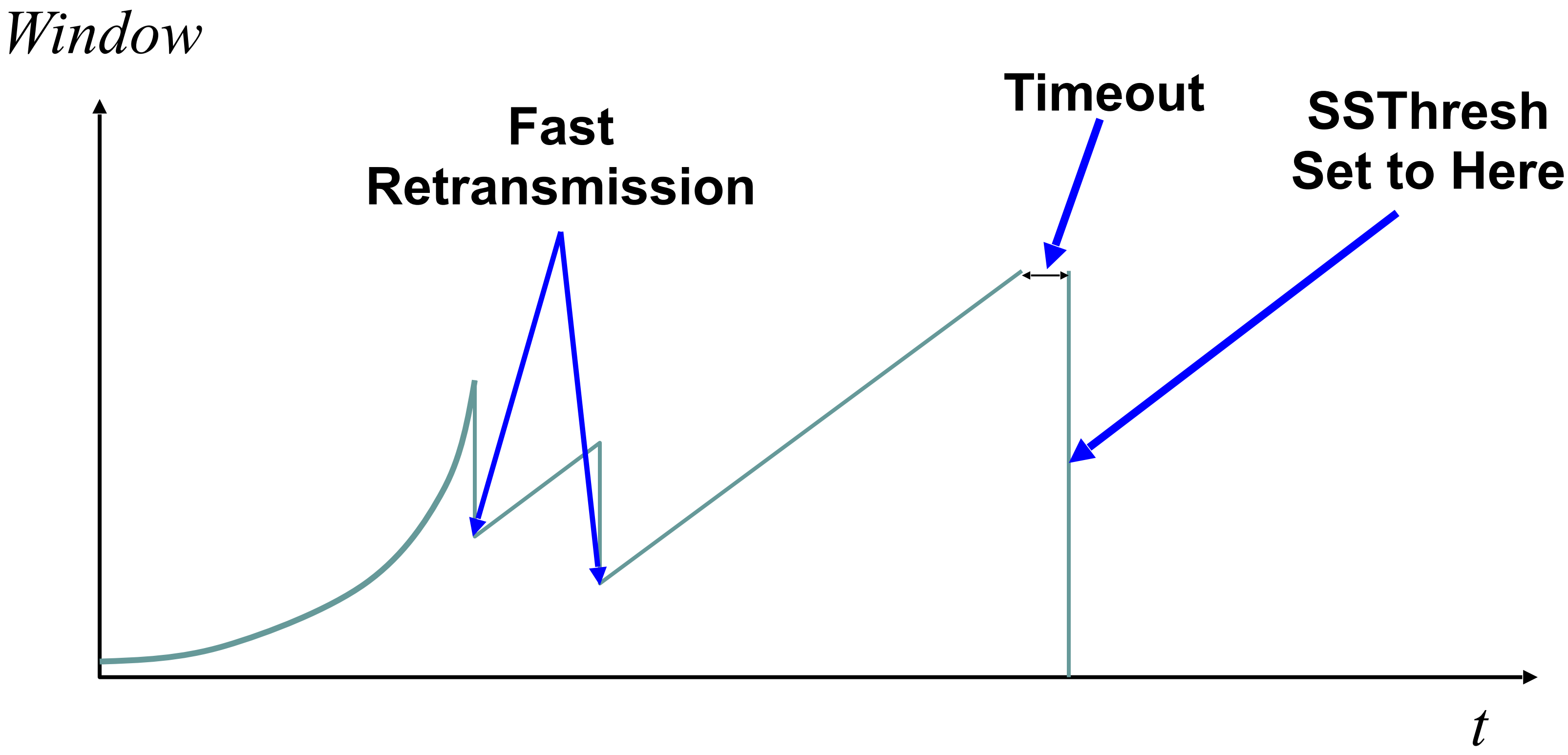
Time Diagram



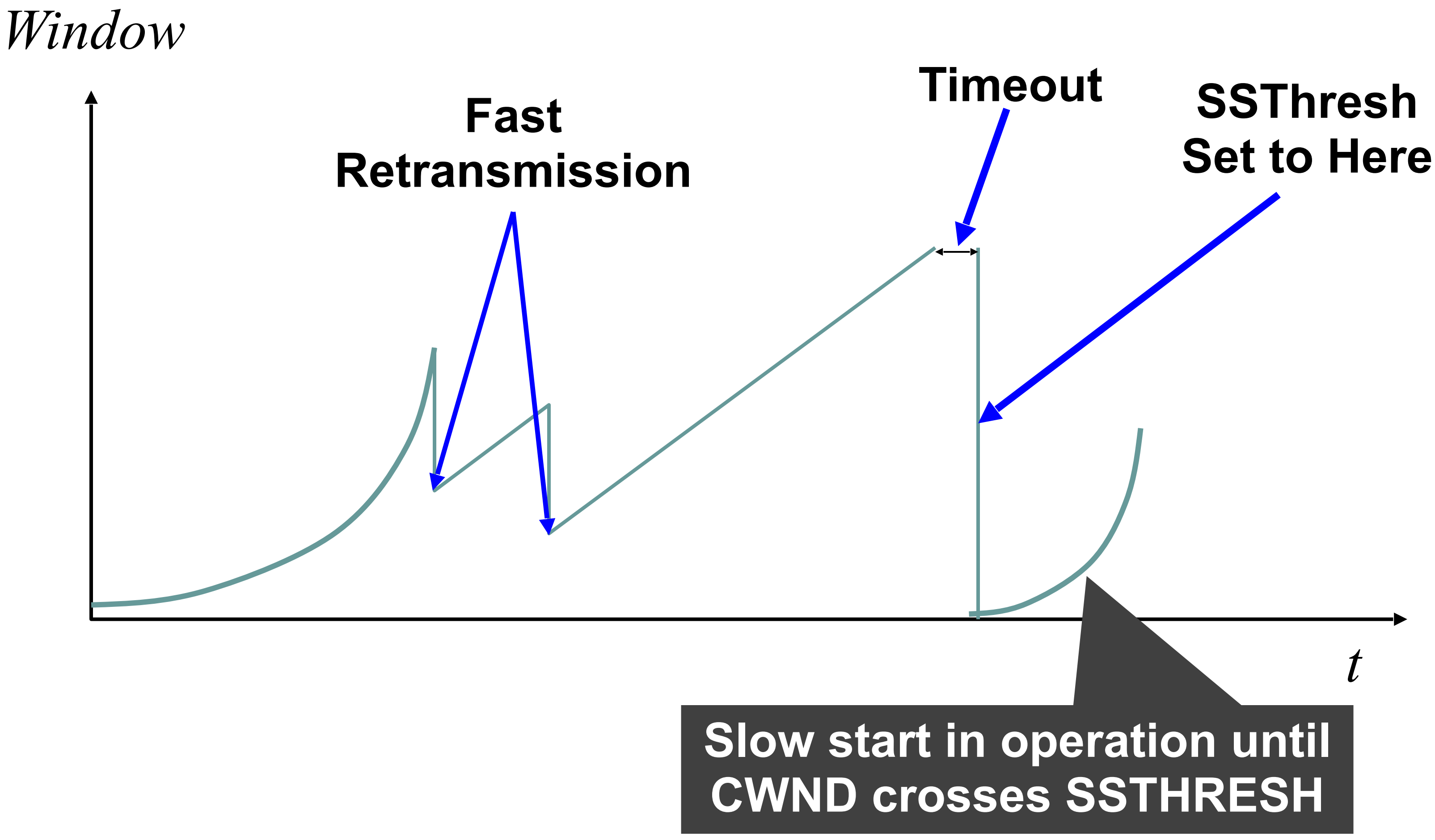
Time Diagram



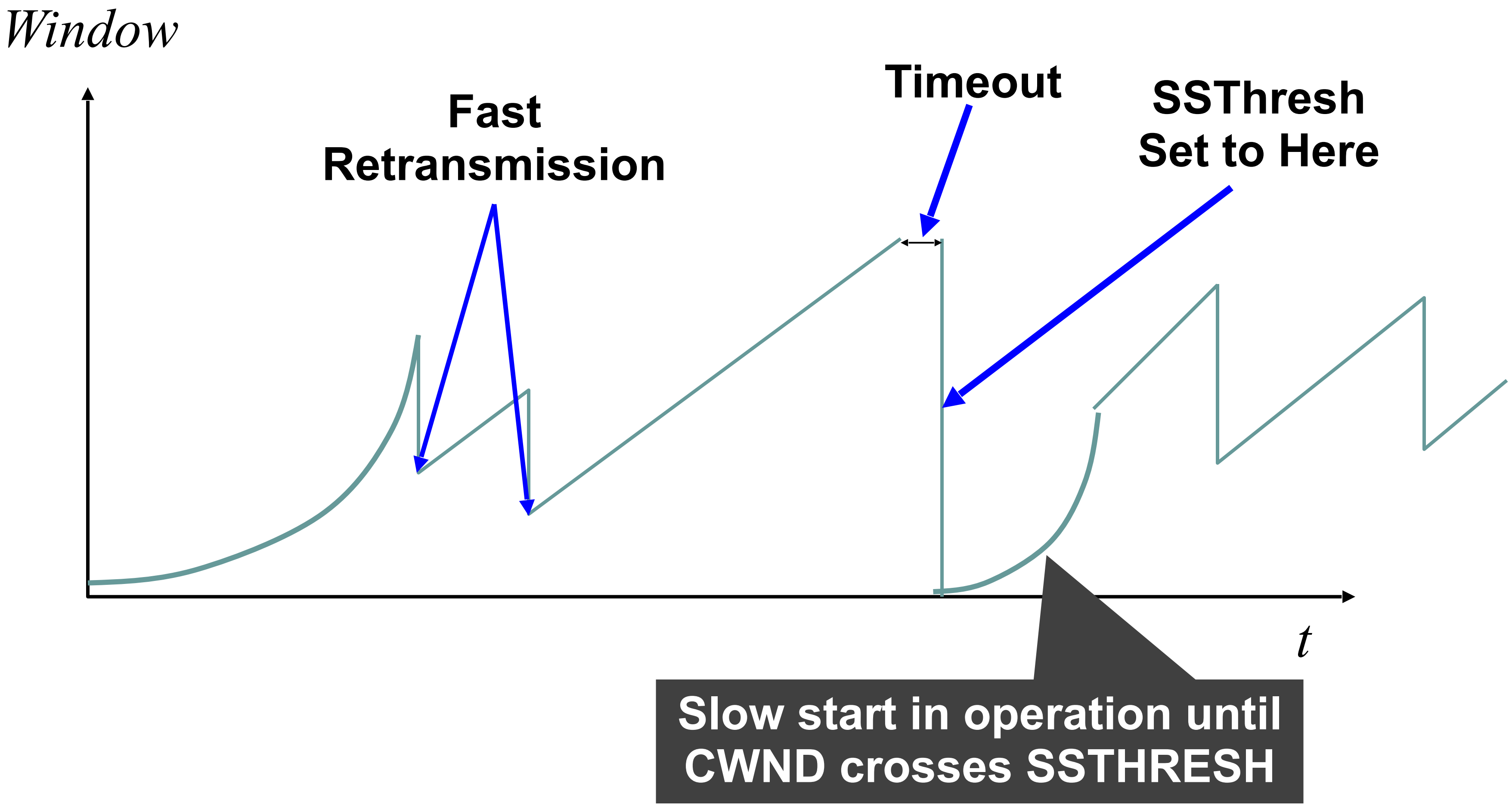
Time Diagram



Time Diagram



Time Diagram



One Final Phase: Fast Recovery

One Final Phase: Fast Recovery

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss

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

Example

Example

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

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

Timeline (at sender)

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

Timeline (at sender)

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

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)

Timeline (at sender)

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

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)

Timeline (at sender)

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

- 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)

Timeline (at sender)

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

- 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

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

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)

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)

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)

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)

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

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)

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)

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)

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)

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

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 no additional ACKs for another RTT

Two Questions

Two Questions

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

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

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

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

Solution: Fast Recovery

Solution: Fast Recovery

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

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

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)

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

- ACK 101 (due to 102) cwnd=10 dupACK#1

Timeline (at sender)

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

- ACK 101 (due to 102) cwnd=10 dupACK#1

Timeline (at sender)

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

- ACK 101 (due to 102) cwnd=10 dupACK#1
- ACK 101 (due to 103) cwnd=10 dupACK#2

Timeline (at sender)

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

- 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

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
- 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)

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
- 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)

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
- 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)

Timeline (at sender)

In flight: 101, 102, 103, 104, 105, 106, 107, 108, 109, 110 **101 111,**

- 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**)

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**)

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**)

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**)

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

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)

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

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: 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
- Plus the usual...

"Congestion Avoidance" phase (additive increase)

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)

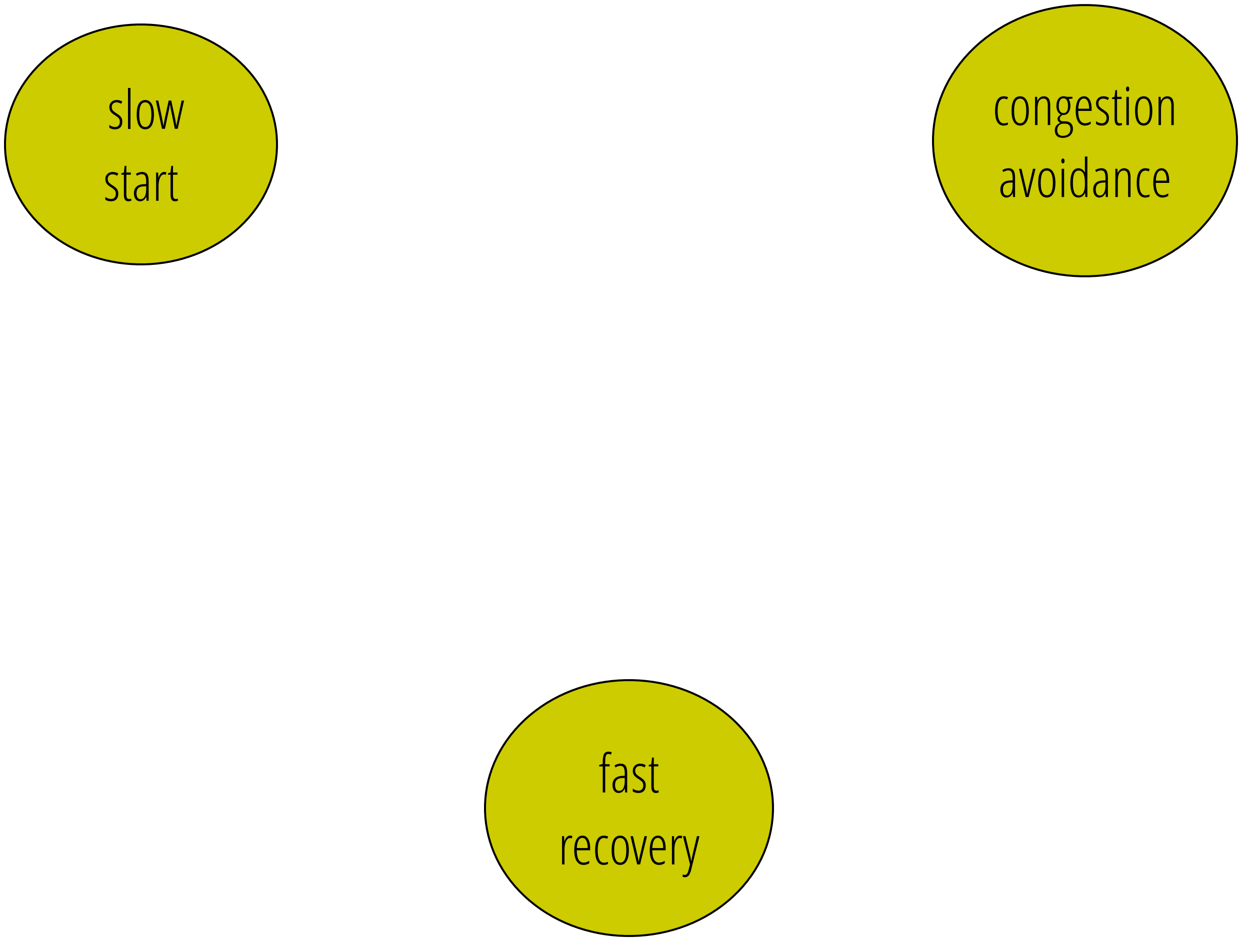
Event: dupACK

- 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)

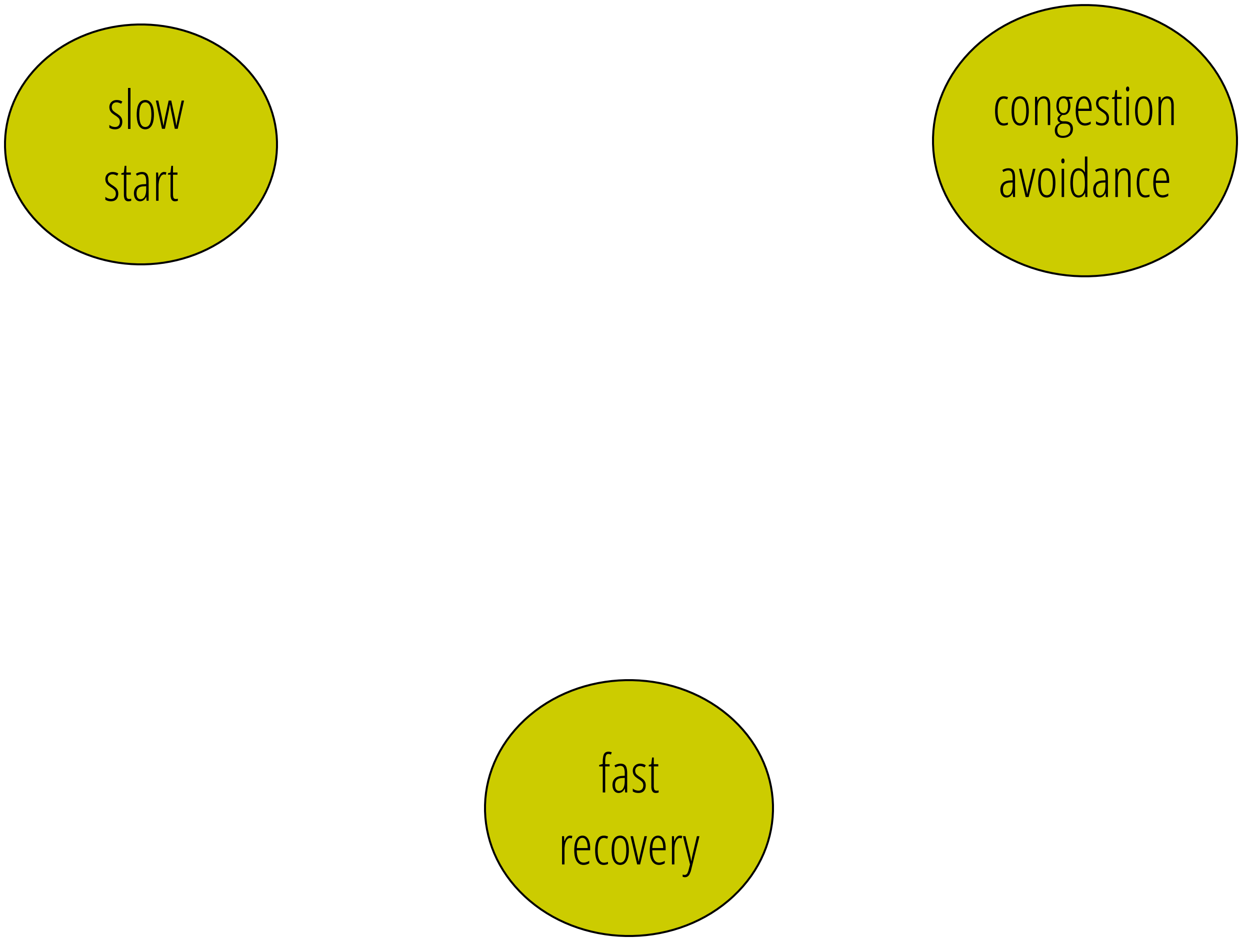
Next: TCP State Machine

Next: TCP State Machine

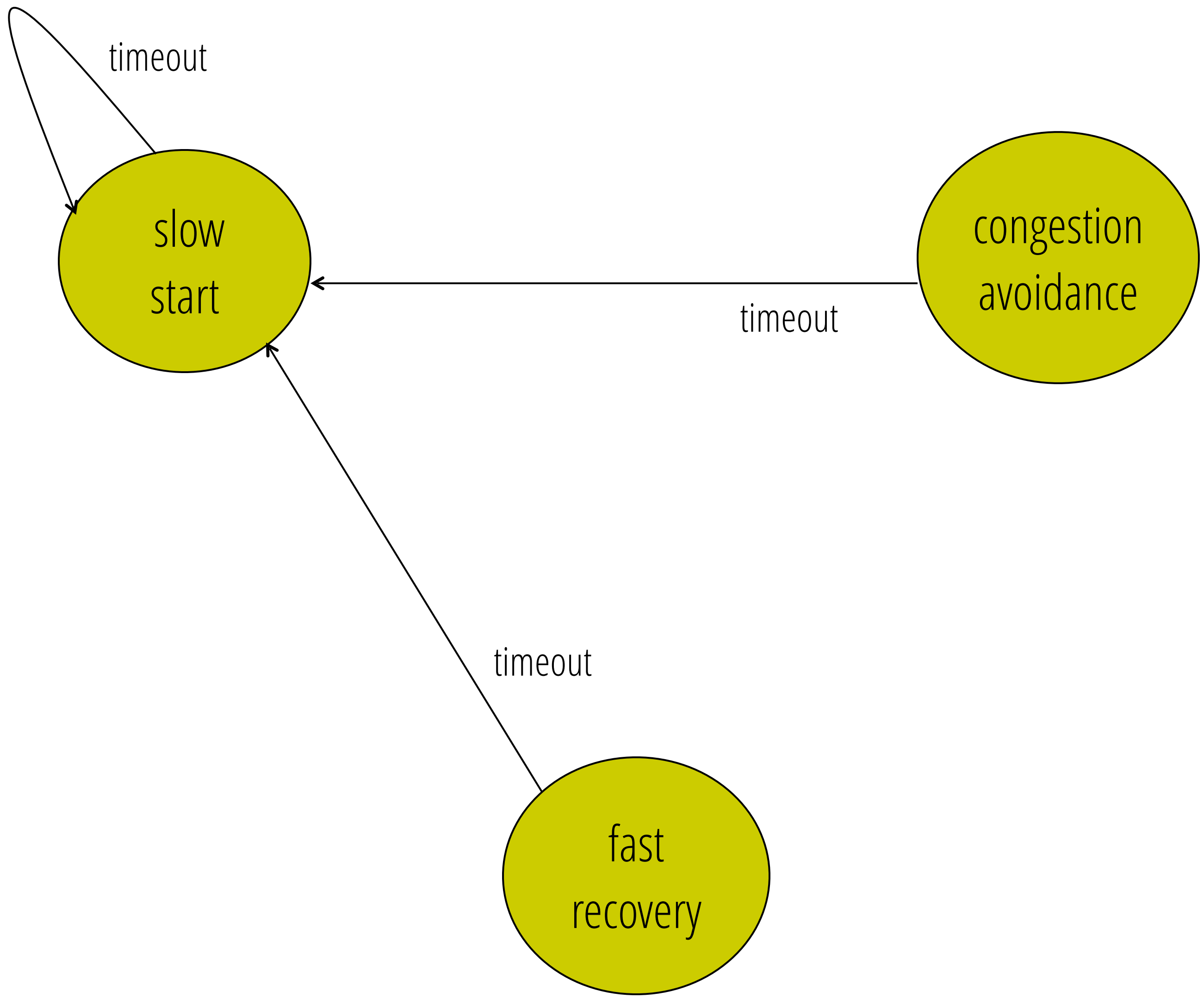
TCP State Machine



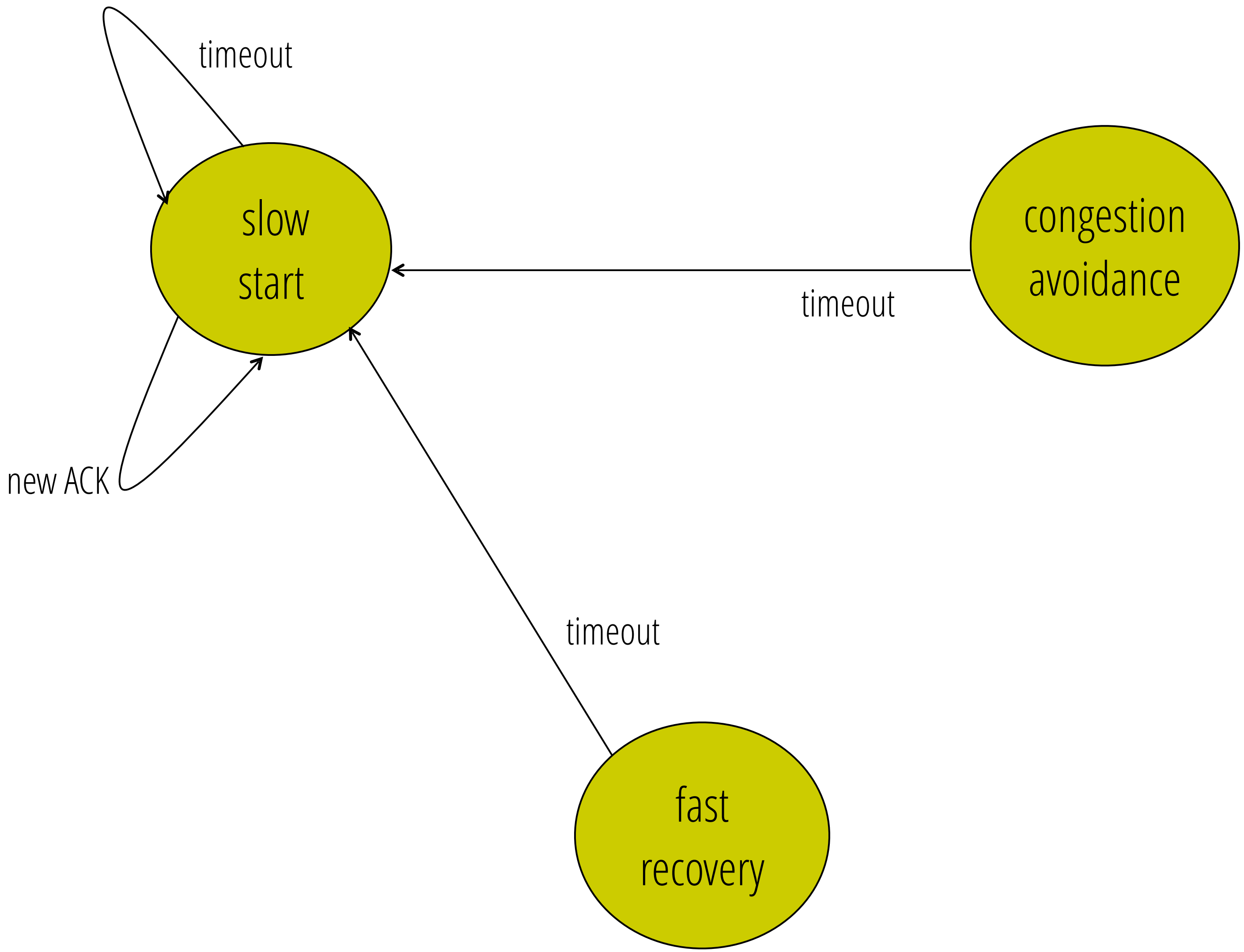
TCP State Machine



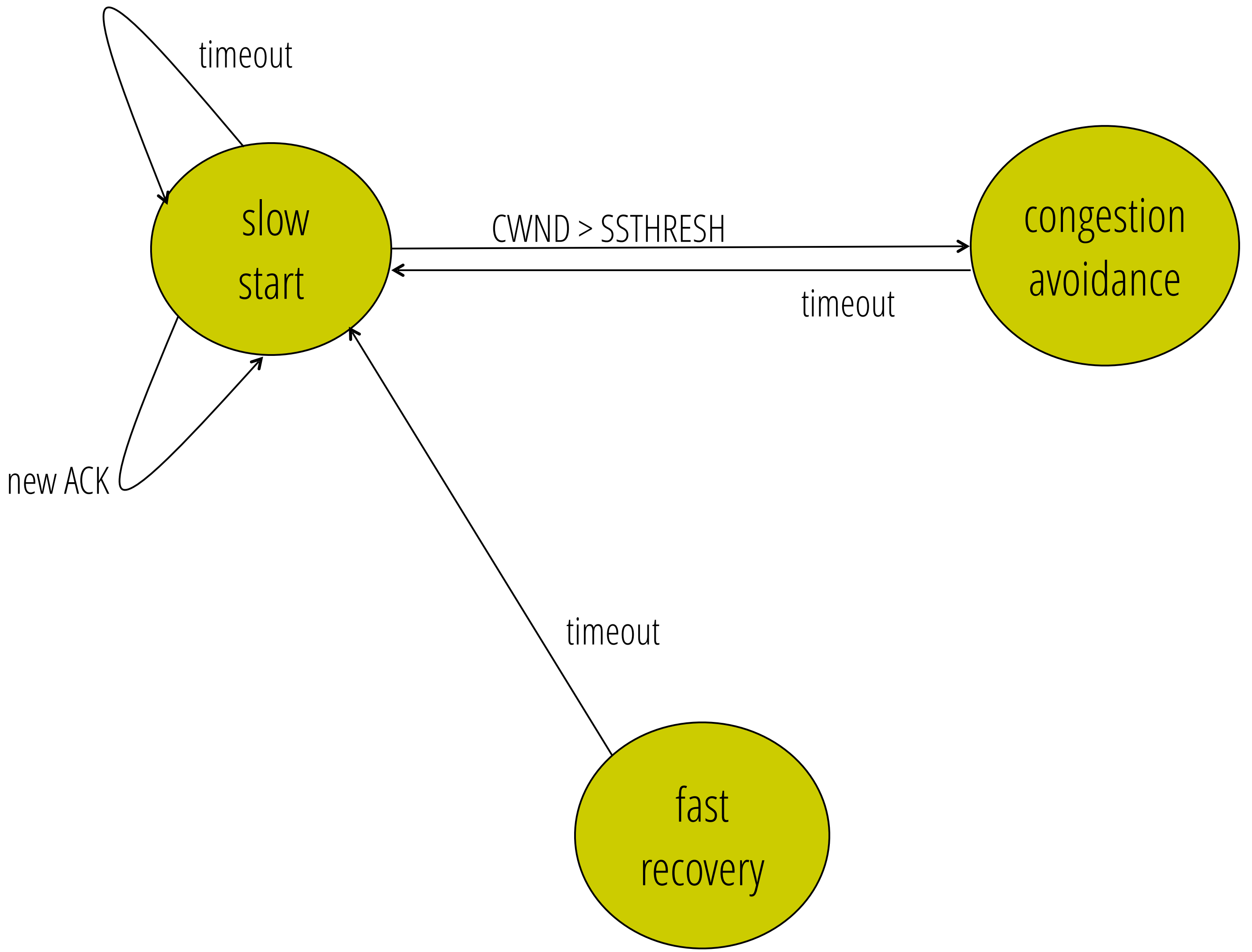
TCP State Machine



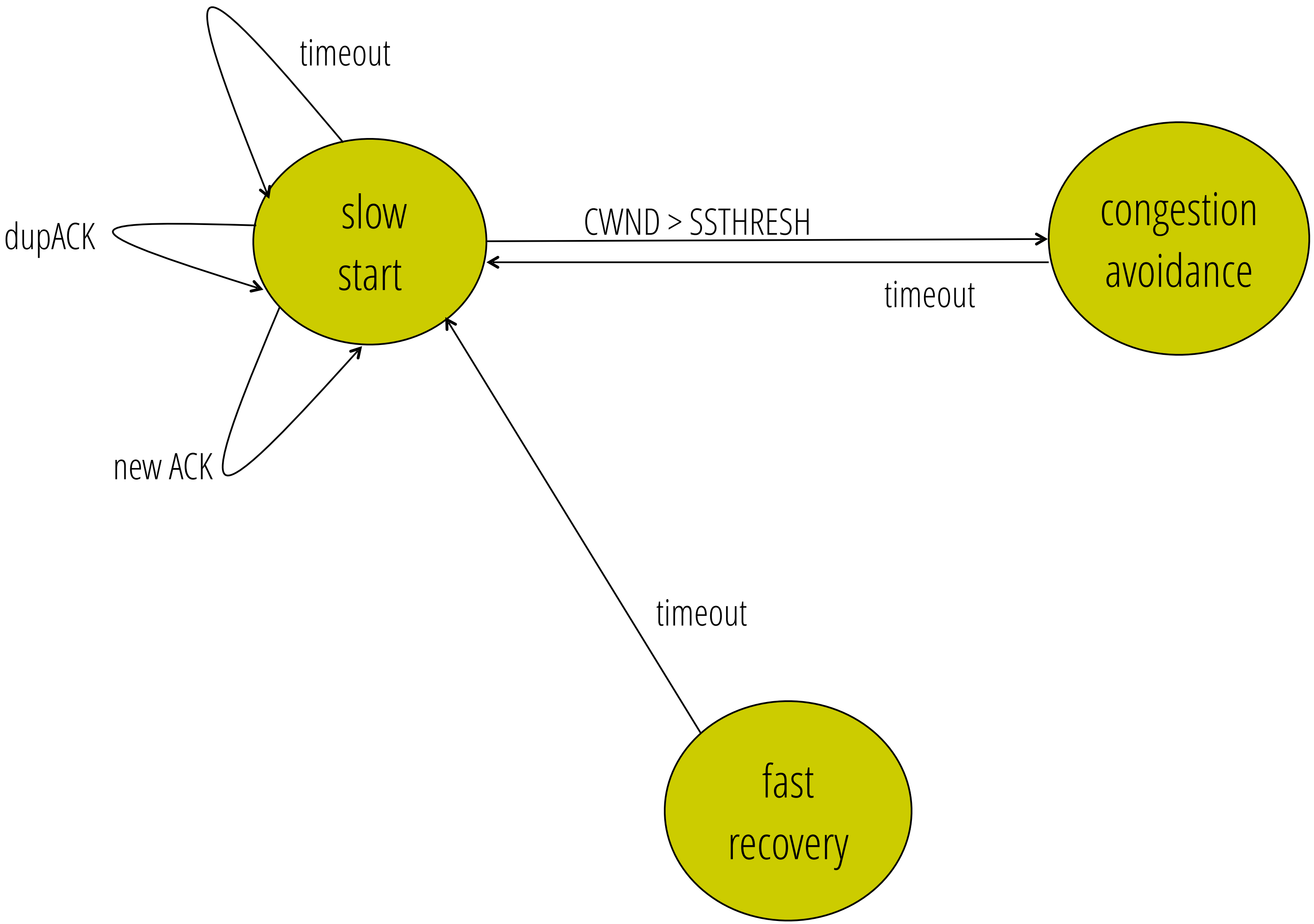
TCP State Machine



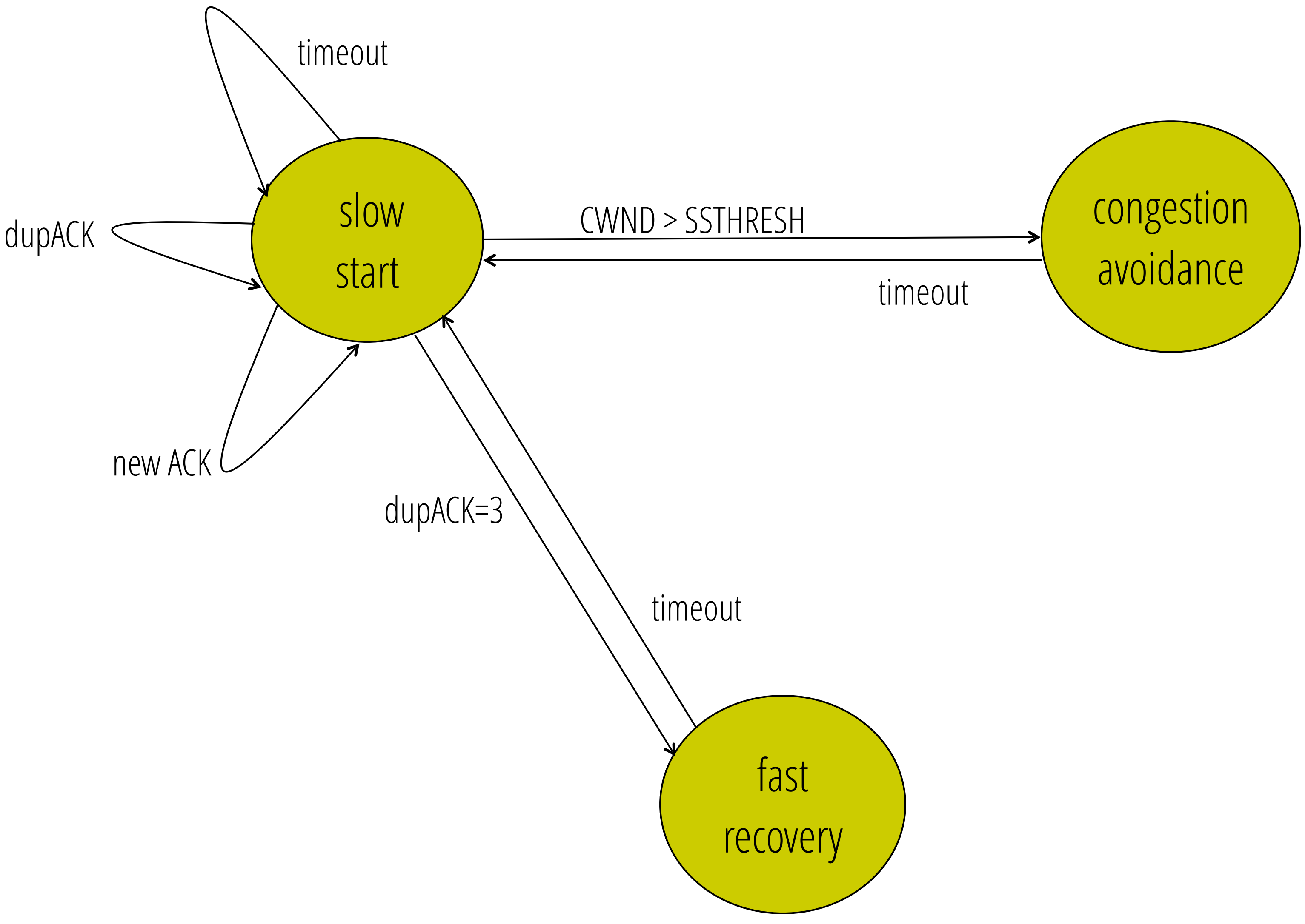
TCP State Machine



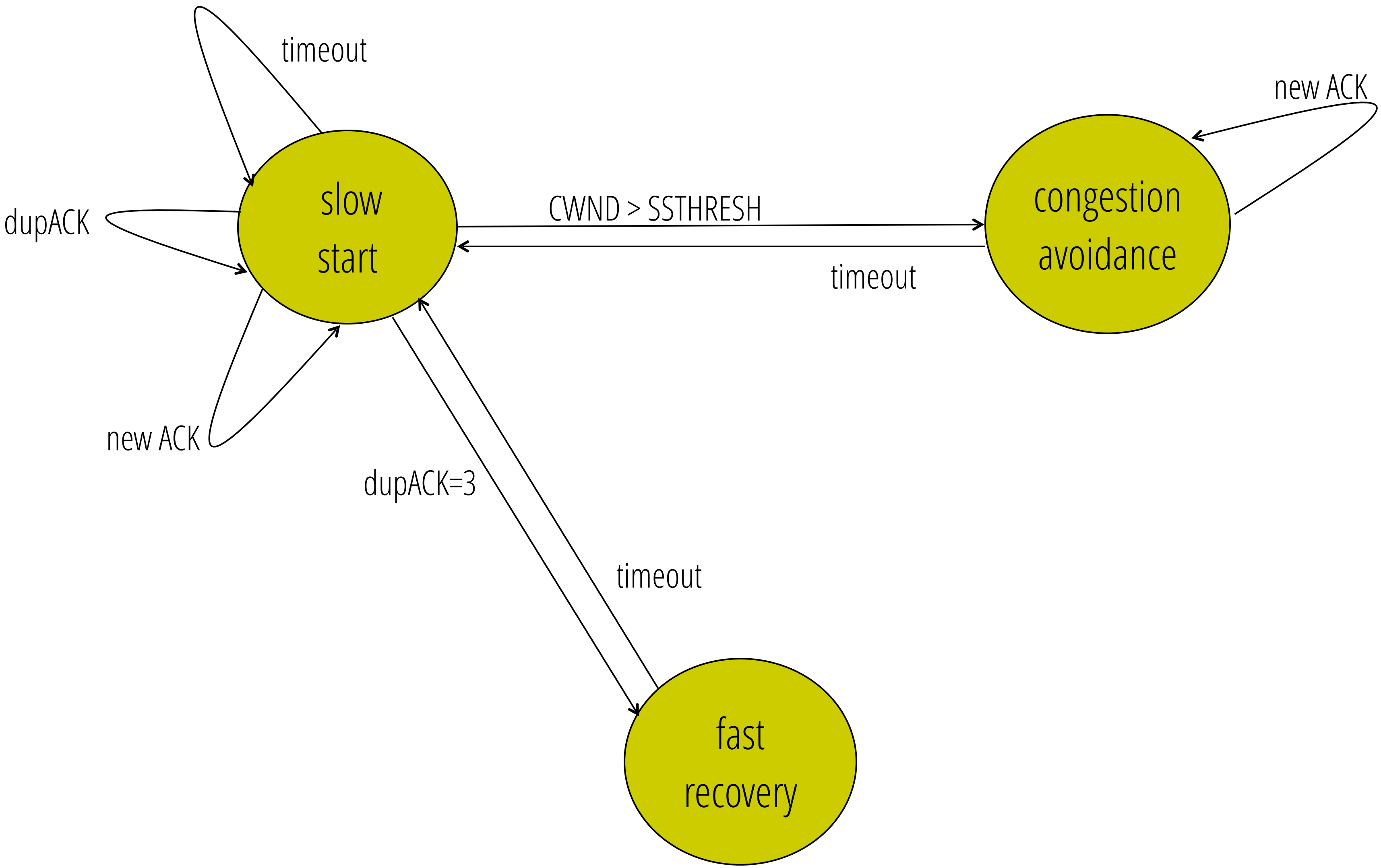
TCP State Machine



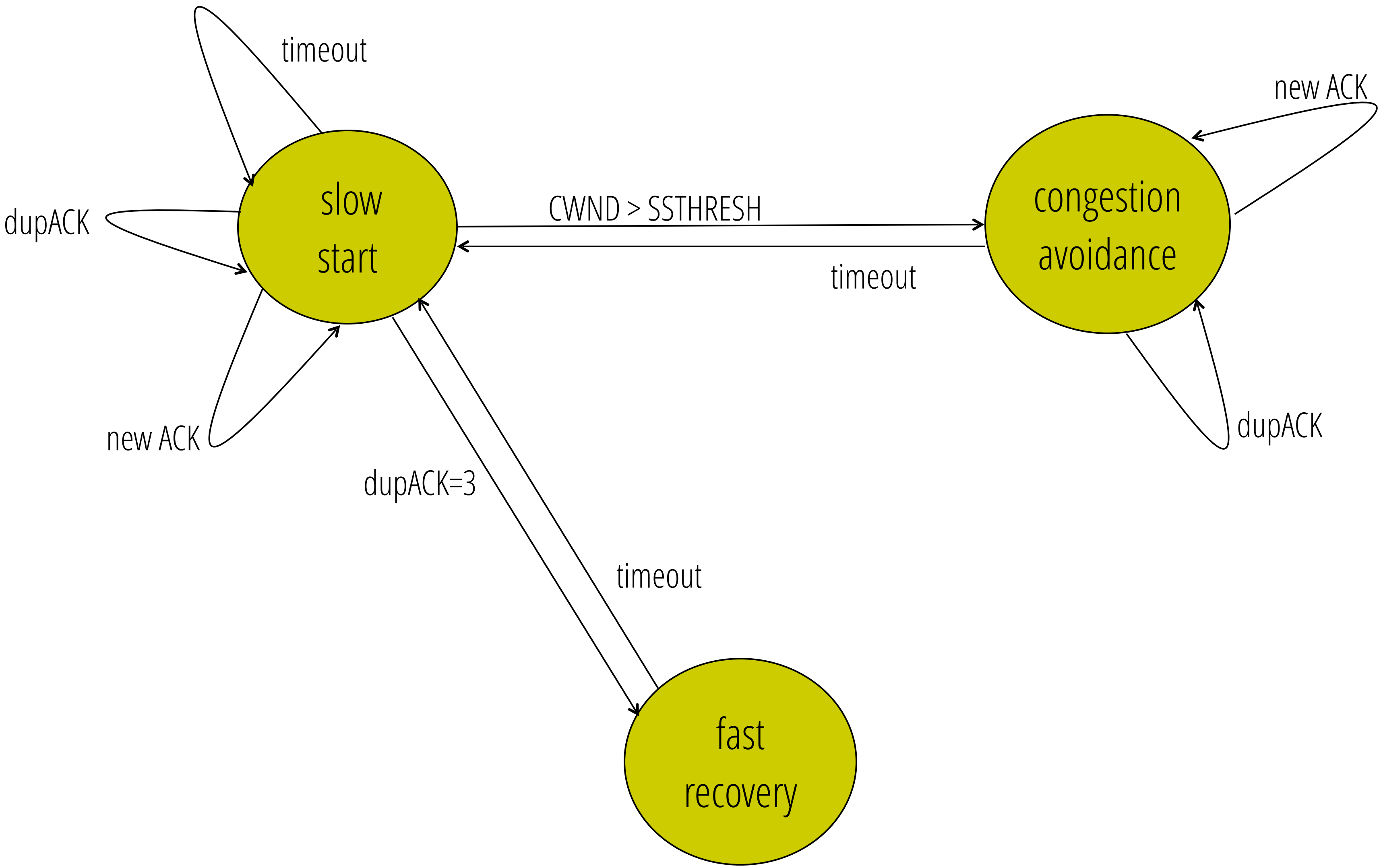
TCP State Machine



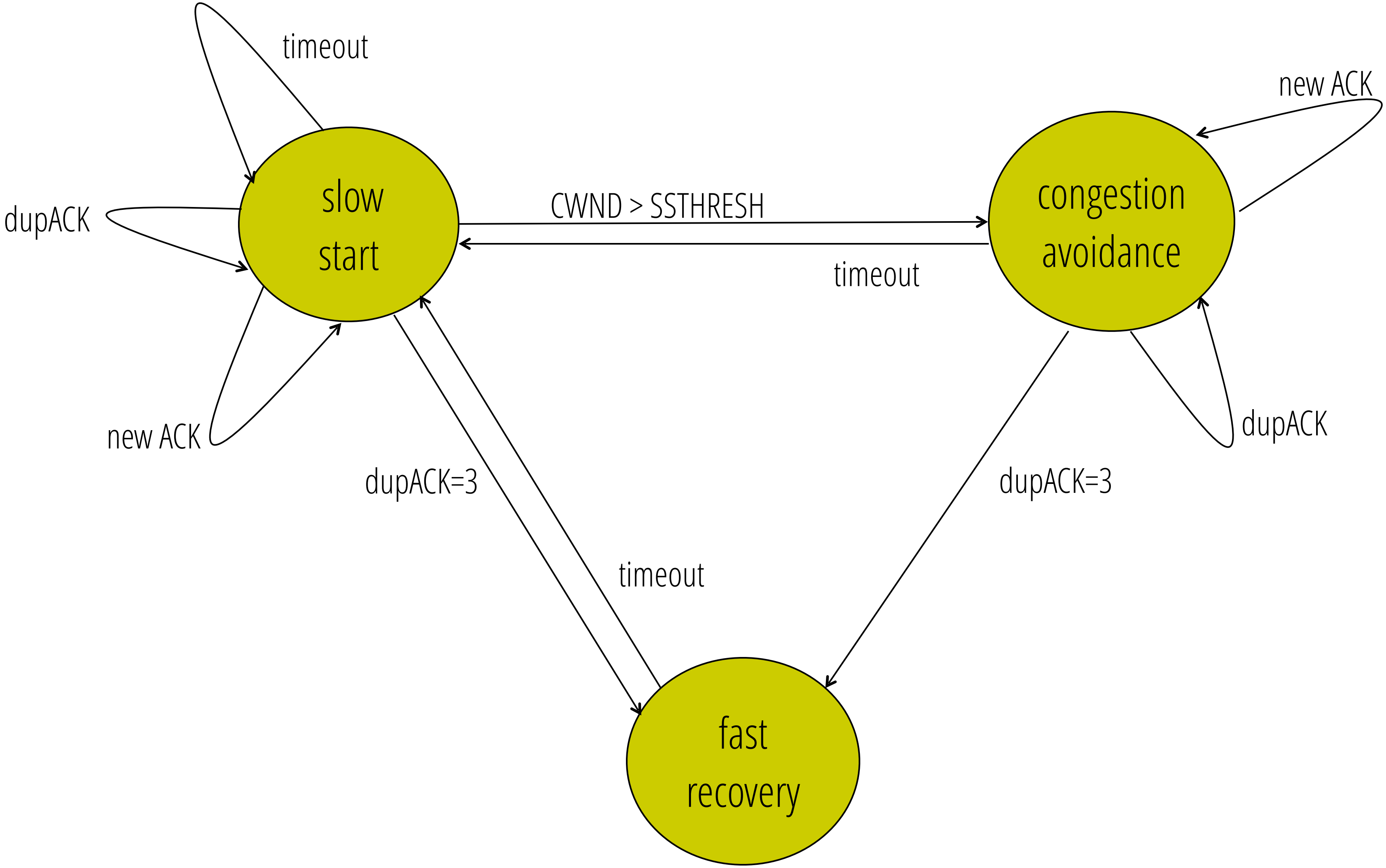
TCP State Machine



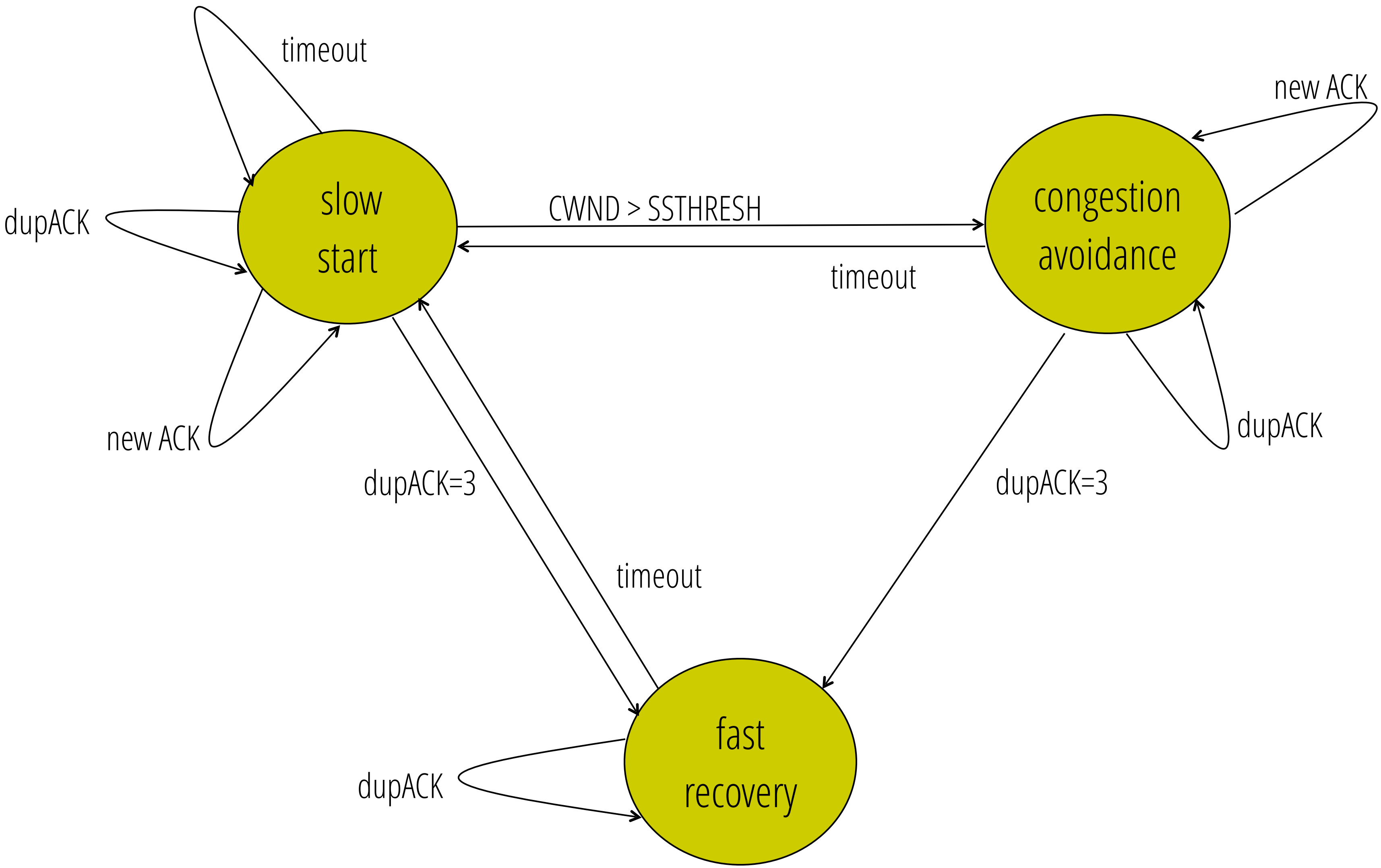
TCP State Machine



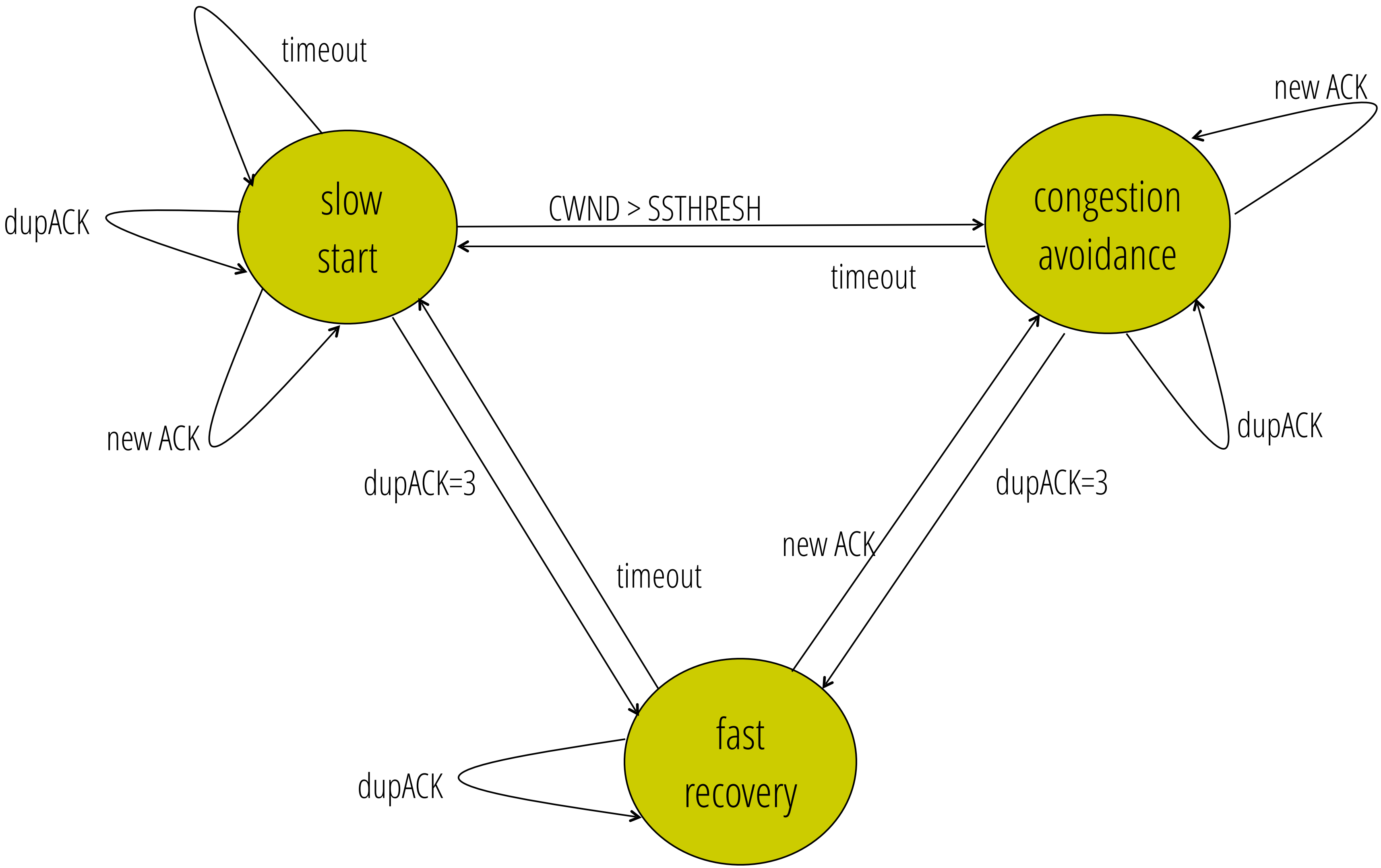
TCP State Machine



TCP State Machine



TCP State Machine



Many variants

Many variants

Many variants

- TCP-Tahoe
  - CWND =1 on triple dupACK

Many variants

- TCP-Tahoe
  - CWND =1 on triple dupACK
- TCP-Reno
  - CWND =1 on timeout
  - CWND = CWND/2 on triple dupack

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

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

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

Interoperability

Interoperability

- How can all these algorithms coexist? Don’t we need a single, uniform standard?

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?

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?

TCP Throughput Equation

TCP Throughput

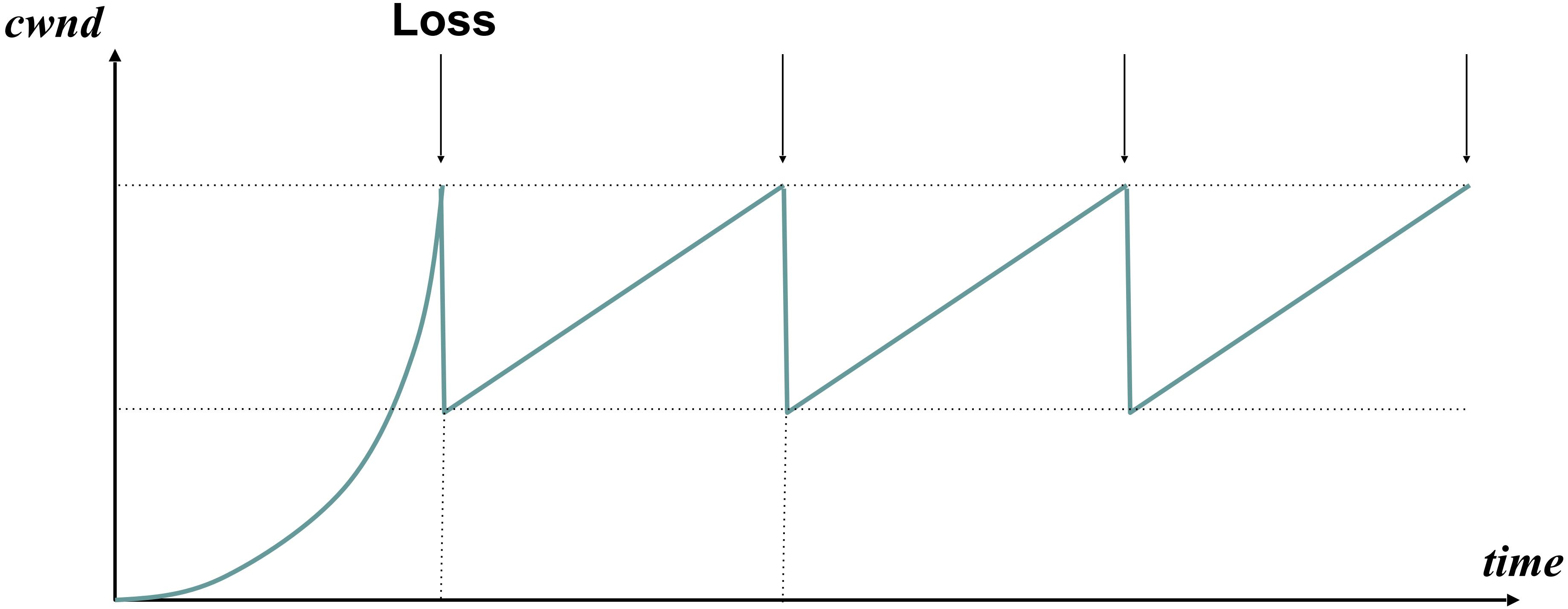
TCP Throughput

- Given a path, what TCP throughput can we expect?

TCP Throughput

- Given a path, what TCP throughput can we expect?
- We'll derive a simple model that expresses TCP throughput in terms of path properties:
  - RTT
  - Loss rate, p

A Simple Model for TCP Throughput



A Simple Model for TCP Throughput

A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}

A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}
- And is detected by duplicate ACKs (i.e., no timeouts)

A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}
- And is detected by duplicate ACKs (i.e., no timeouts)

- Hence, evolution of window size:
  - $\frac{1}{2}W_{max}$ (after detecting loss)
  - $\frac{1}{2}W_{max}$ +1 (one RTT later)
  - $\frac{1}{2}W_{max}$ +2 (two RTTs later)
  - $\frac{1}{2}W_{max}$ +3 (three RTTs later)
  - ...
  - W_{max} [drop]
  - $\frac{1}{2}W_{max}$
  - $\frac{1}{2}W_{max}$ +1

A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}
- And is detected by duplicate ACKs (i.e., no timeouts)

- Hence, evolution of window size:
  - $\frac{1}{2}W_{max}$ (after detecting loss)
  - $\frac{1}{2}W_{max}$ +1 (one RTT later)
  - $\frac{1}{2}W_{max}$ +2 (two RTTs later)
  - $\frac{1}{2}W_{max}$ +3 (three RTTs later)
  - ...
  - W_{max} [drop]
  - $\frac{1}{2}W_{max}$
  - $\frac{1}{2}W_{max}$ +1

$\frac{1}{2}W_{max}$ RTTs

A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}
- And is detected by duplicate ACKs (i.e., no timeouts)

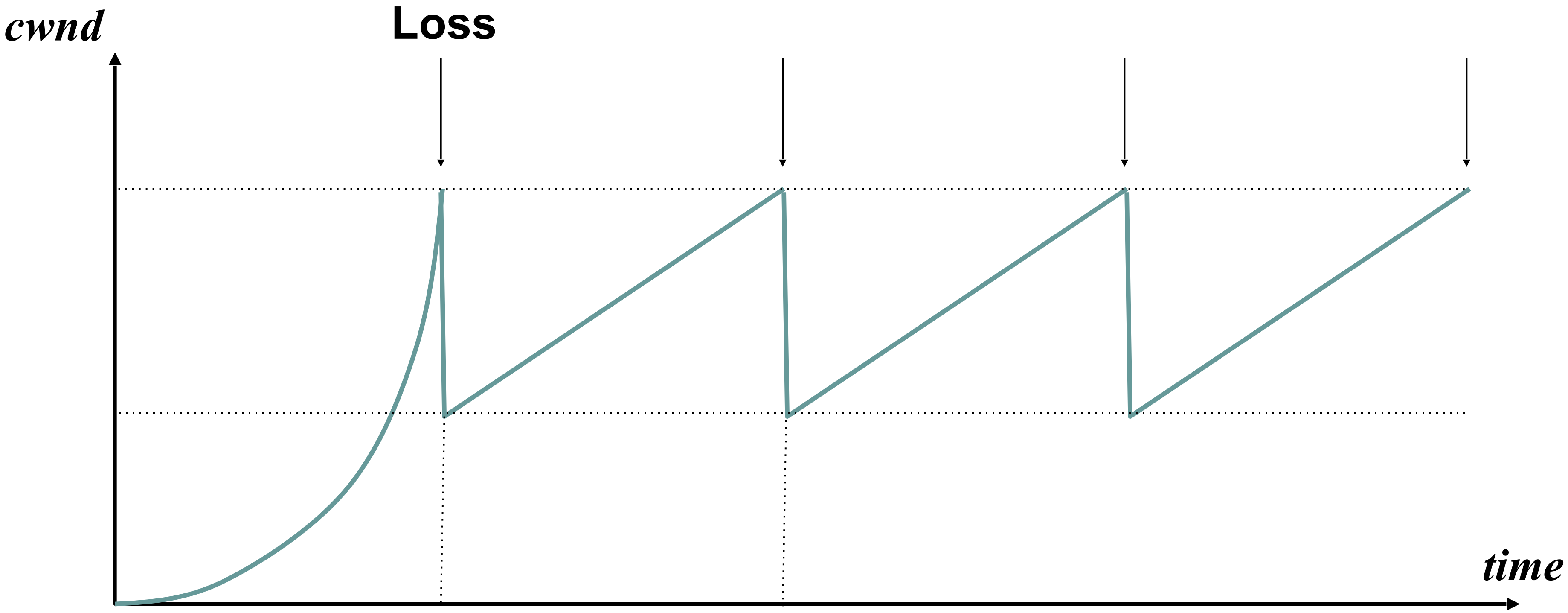
- Hence, evolution of window size:
  - Increase by 1 for $\frac{1}{2}W_{max}$ RTTs, then drop, then repeat

- Average window size per RTT = $\frac{3}{4}W_{max}$

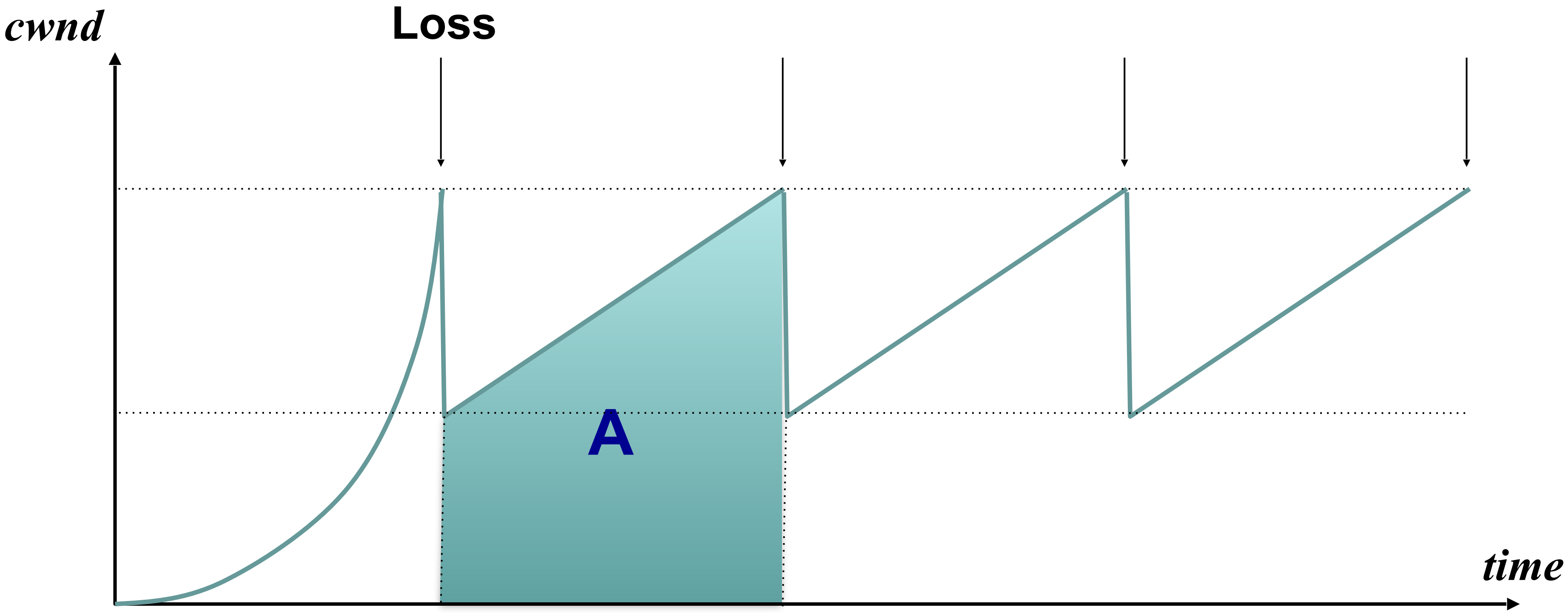
- Average throughput = $\frac{3}{4}W_{max}$ x $\frac{MSS}{RTT}$

- Remaining step: express W_{max} in terms of loss rate p

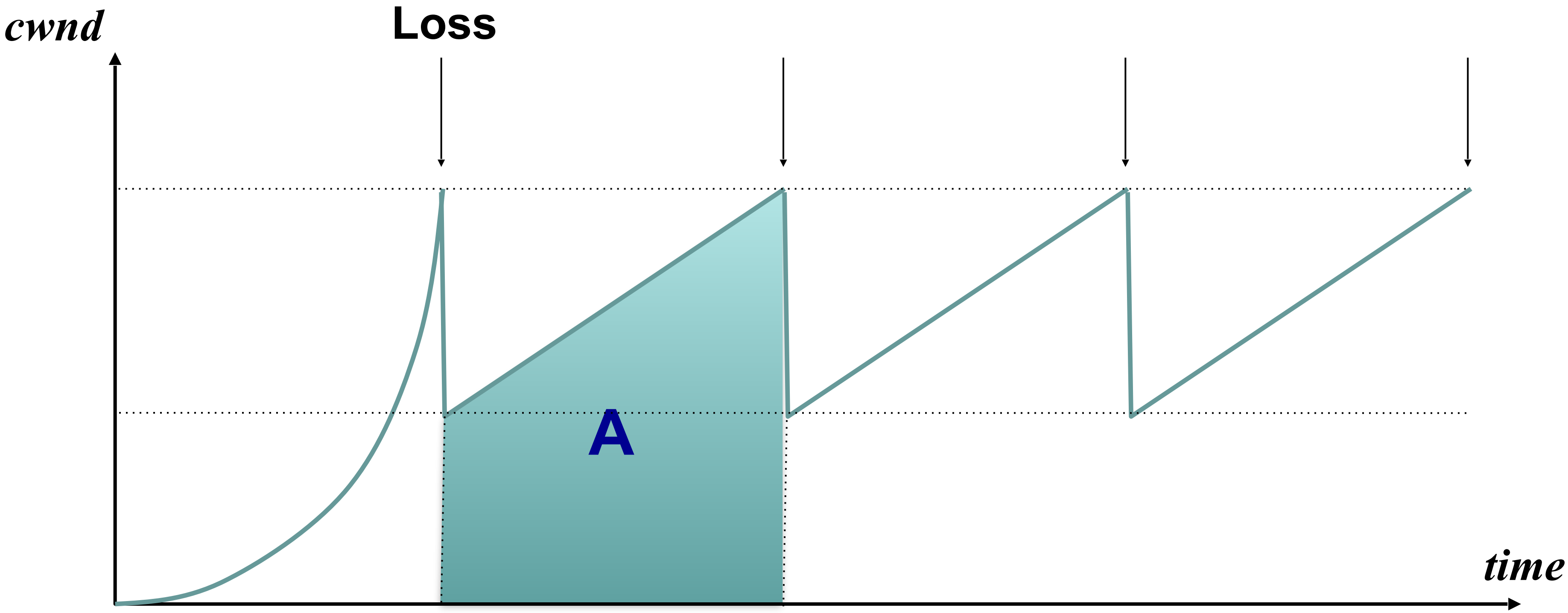
A Simple Model for TCP Throughput



A Simple Model for TCP Throughput

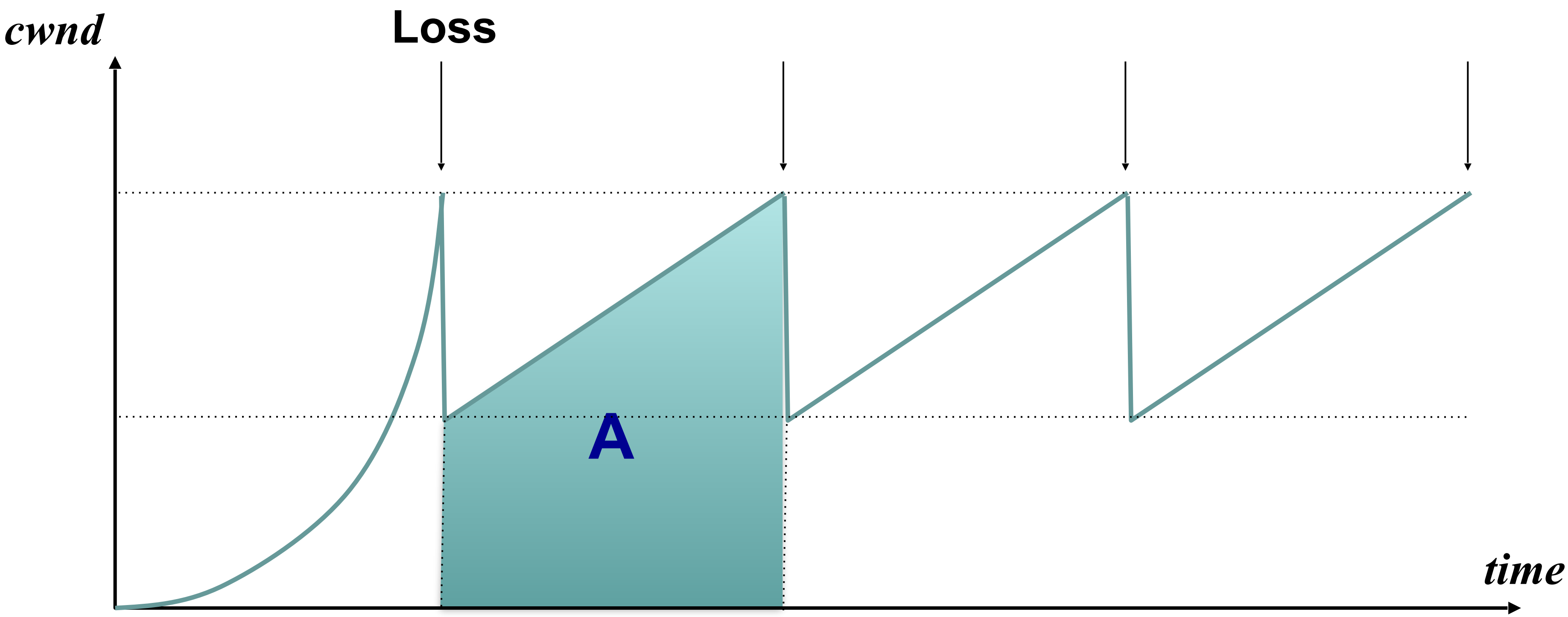


A Simple Model for TCP Throughput



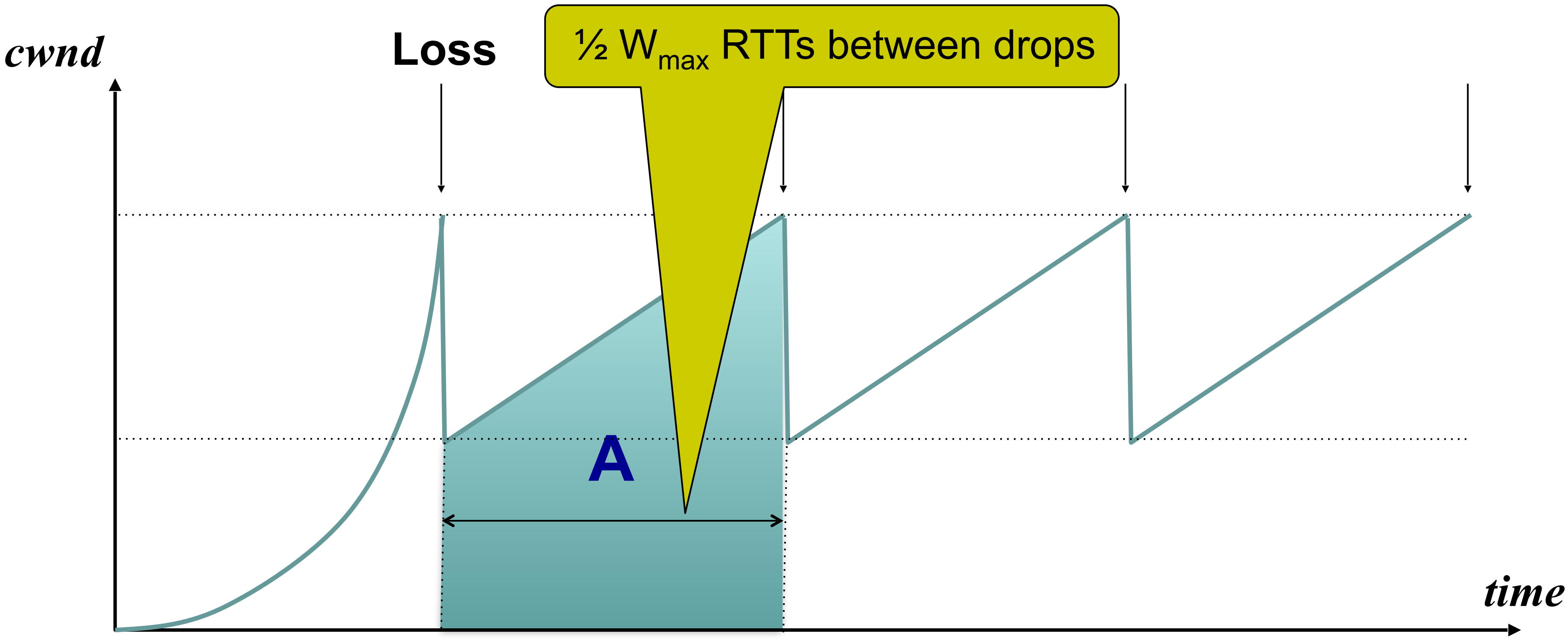
On average, one of all packets in shaded region is lost
(i.e., loss rate is 1/A, where A is #packets in shaded region)

A Simple Model for TCP Throughput



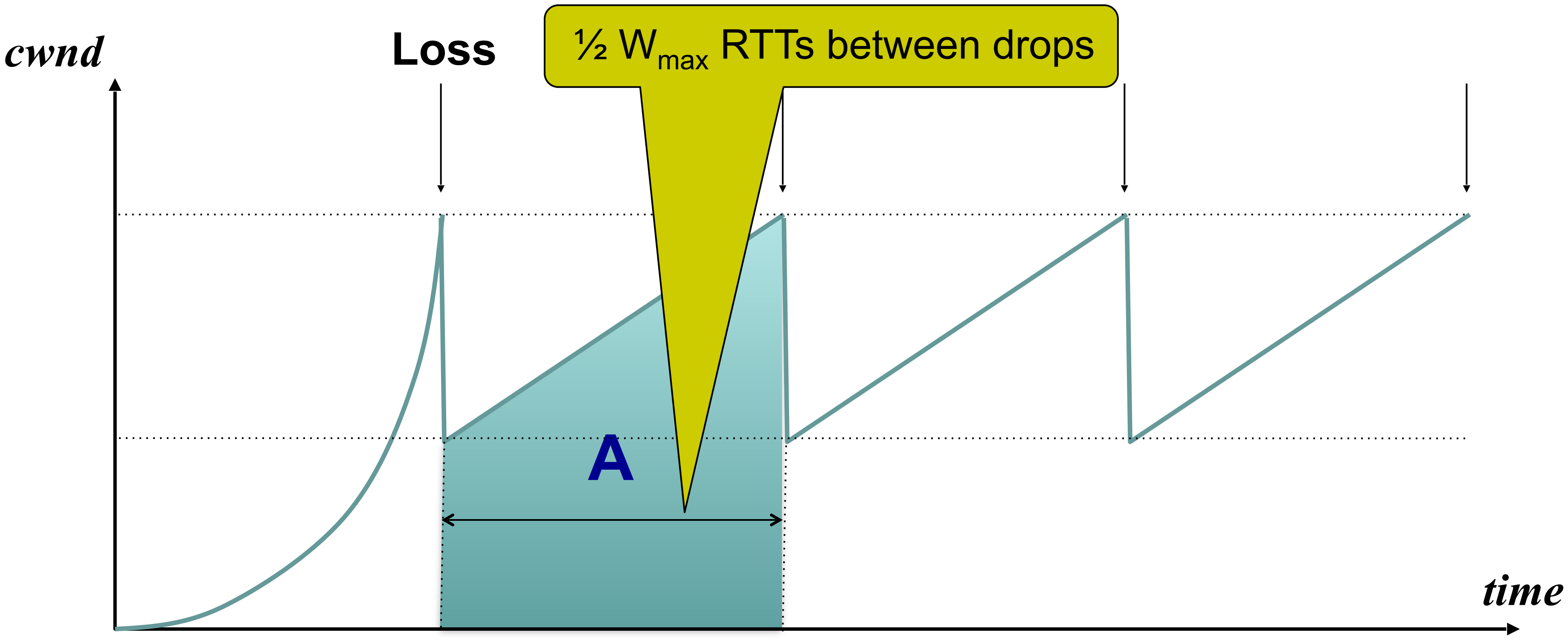
Packet drop rate, $p = \frac{1}{A}$

A Simple Model for TCP Throughput



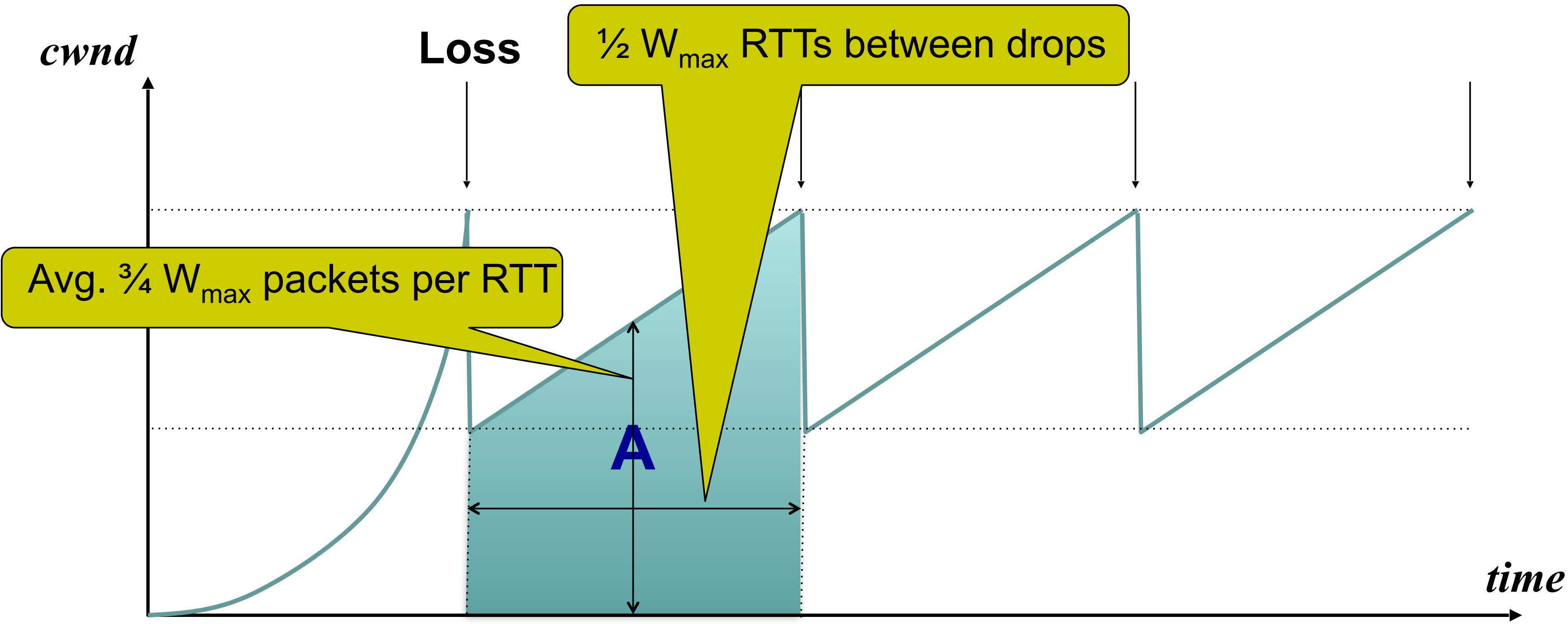
Packet drop rate, $p = \frac{1}{A}$

A Simple Model for TCP Throughput



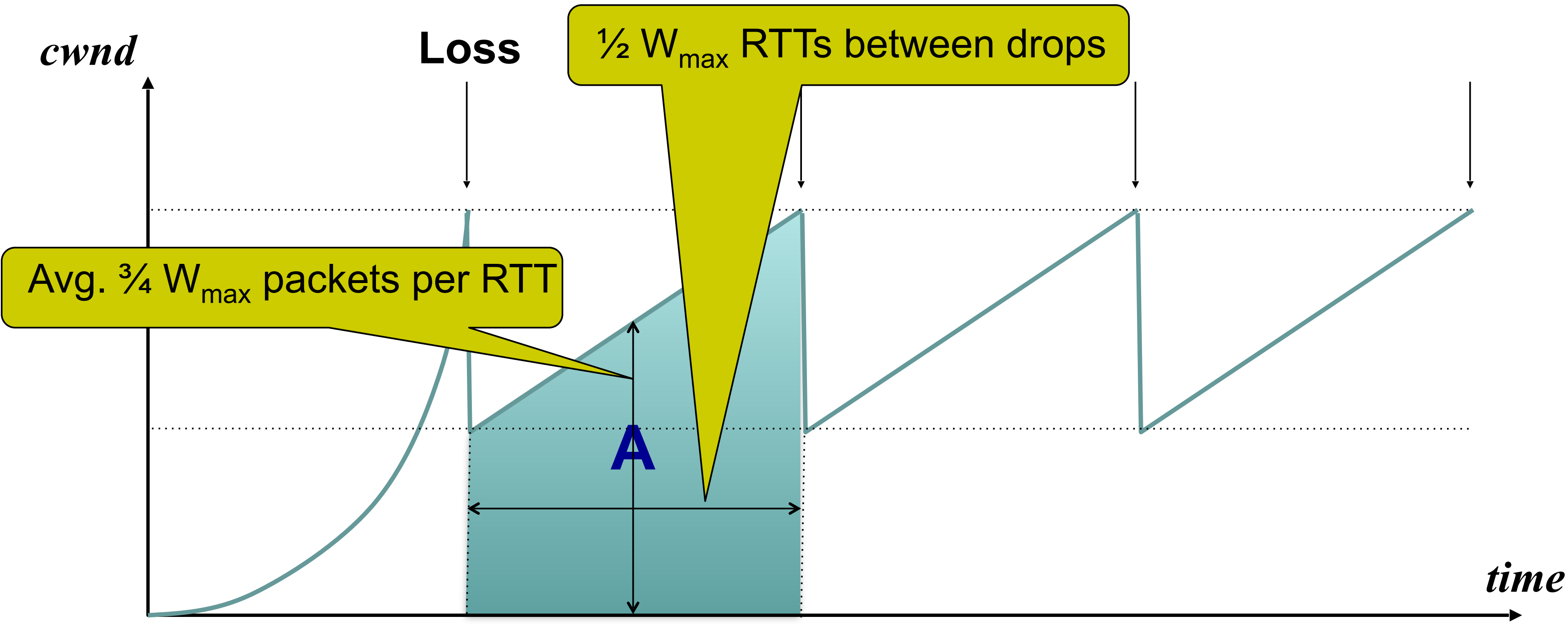
Packet drop rate, $p = \frac{1}{A}$

A Simple Model for TCP Throughput



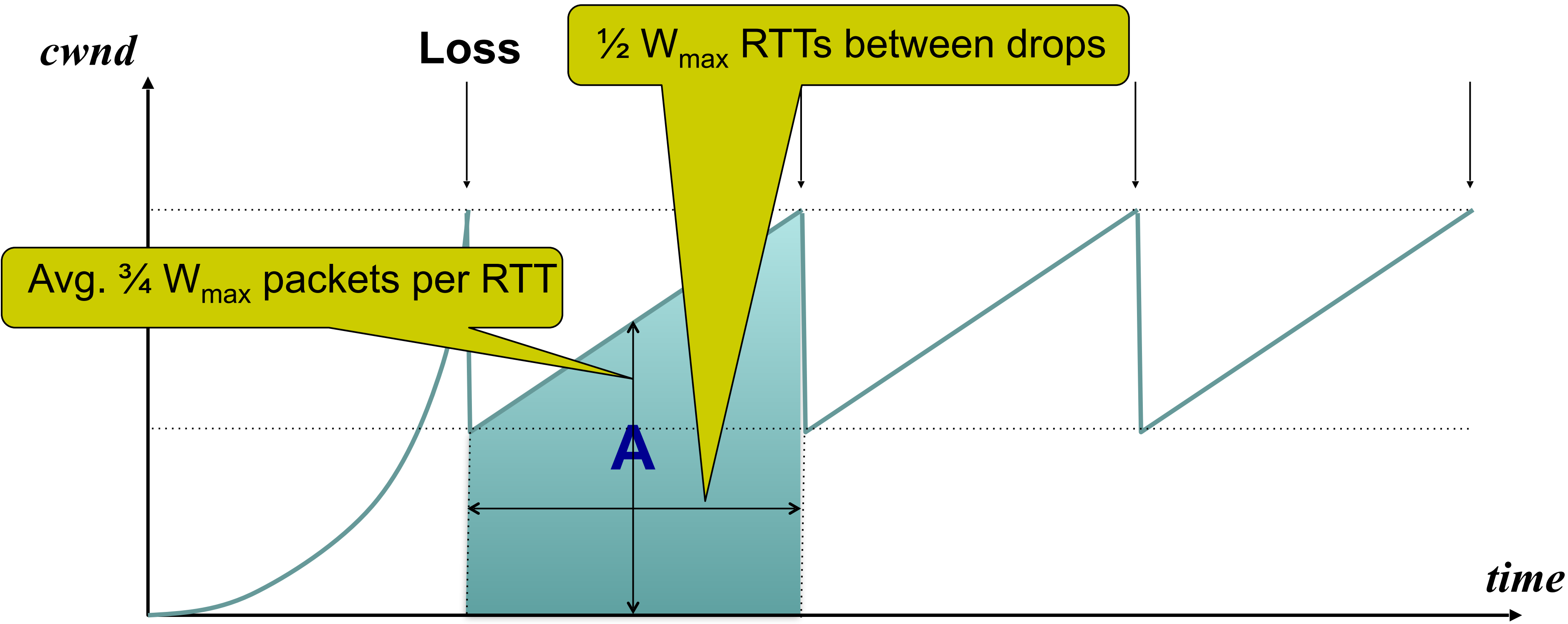
Packet drop rate, $p = \frac{1}{A}$

A Simple Model for TCP Throughput



Packet drop rate, $p = \frac{1}{A}$

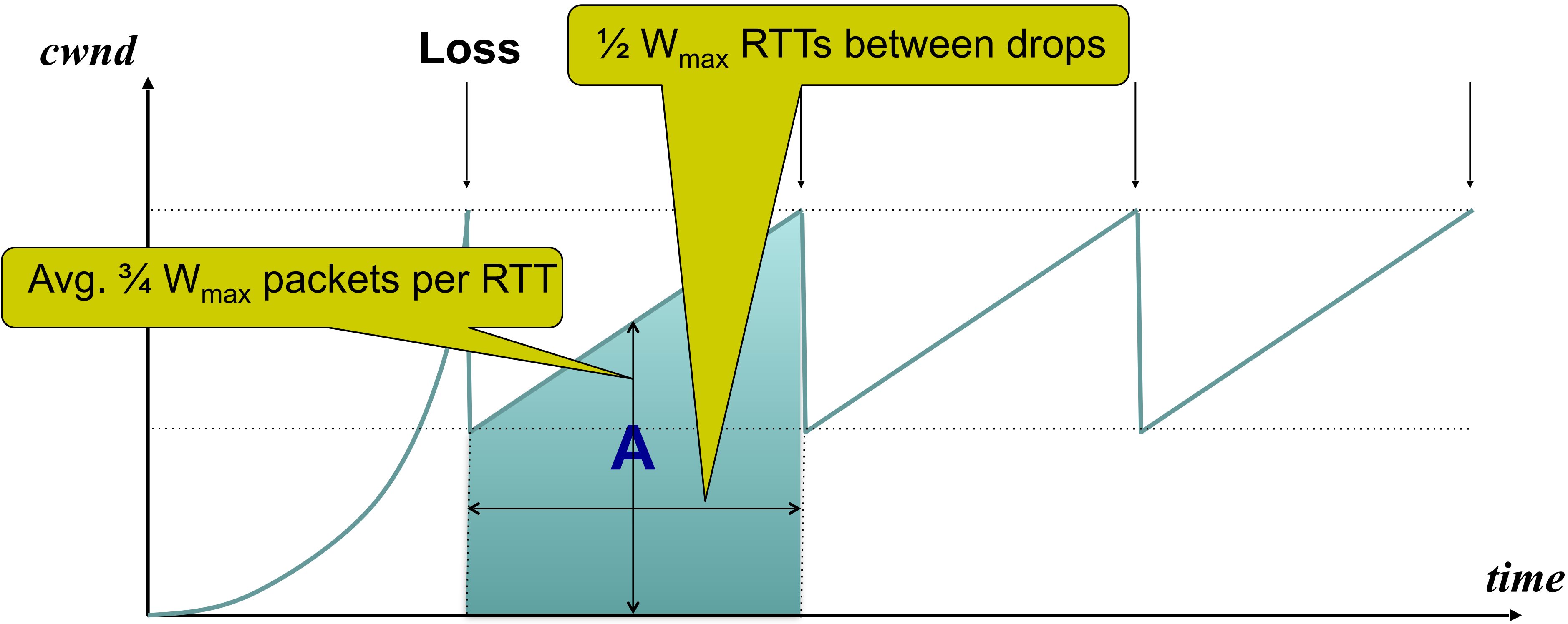
A Simple Model for TCP Throughput



Packet drop rate, $p = \frac{1}{A}$

$A = \frac{3}{8} W_{max}^2$

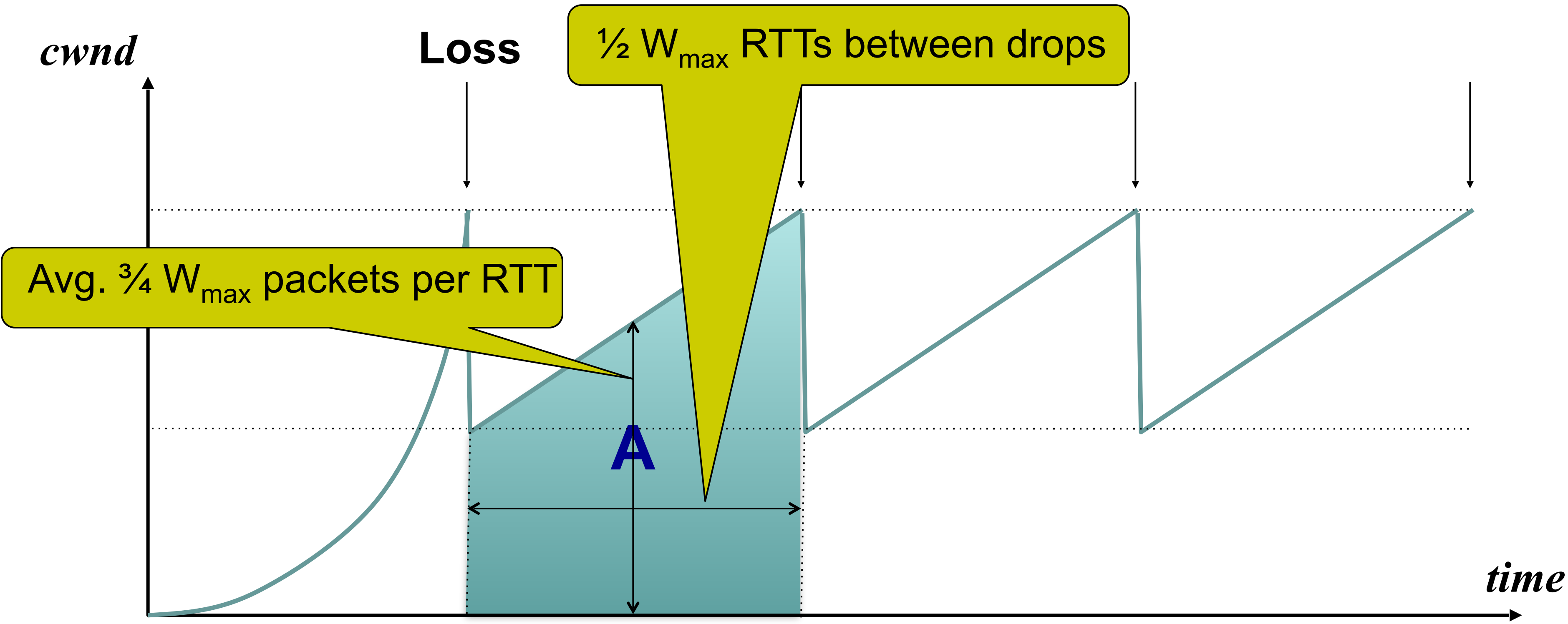
A Simple Model for TCP Throughput



Packet drop rate, $p = \frac{1}{A}$

$A = \frac{3}{8} W_{max}^2$ → $W_{max} = \frac{2\sqrt{2}}{\sqrt{3p}}$

A Simple Model for TCP Throughput

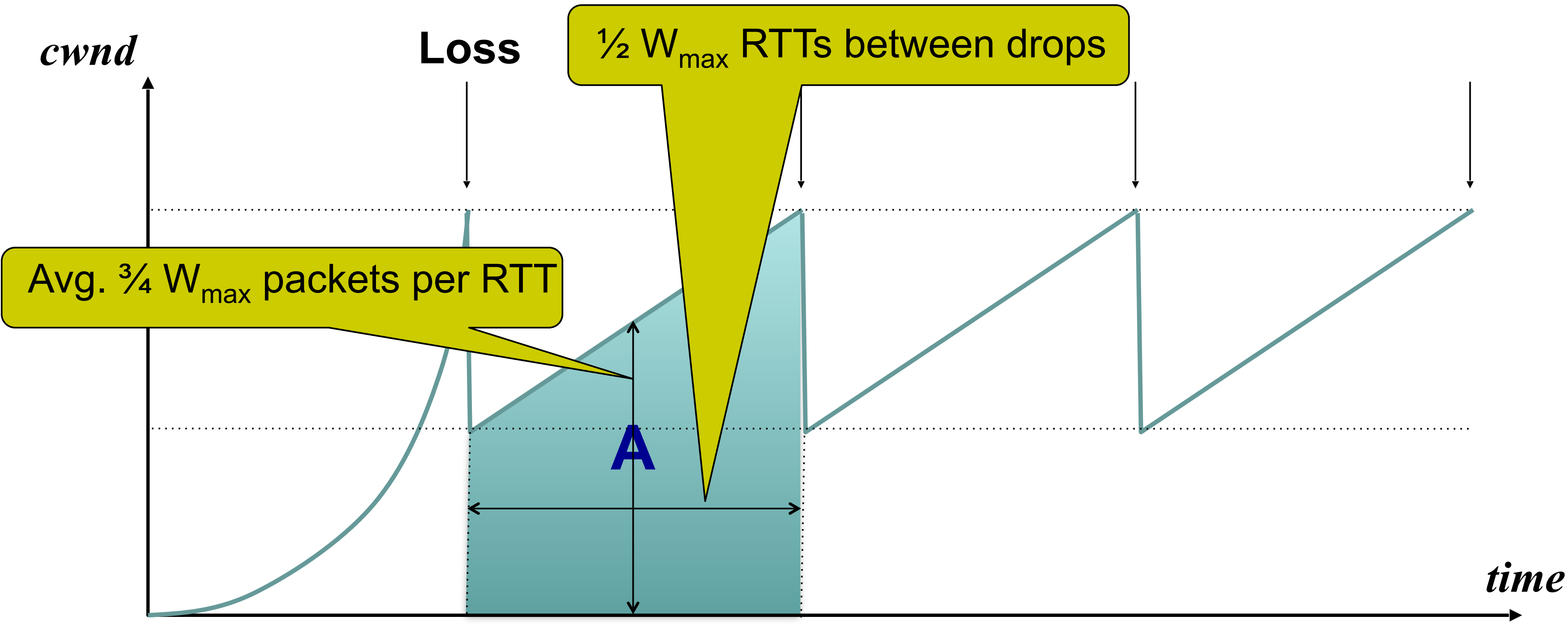


Packet drop rate, $p = \frac{1}{A}$

$A = \frac{3}{8} W_{max}^2$ → $W_{max} = \frac{2\sqrt{2}}{\sqrt{3p}}$

$$\text{Average Throughput} = \frac{\frac{3}{4} W_{max} \times MSS}{RTT}$$

A Simple Model for TCP Throughput

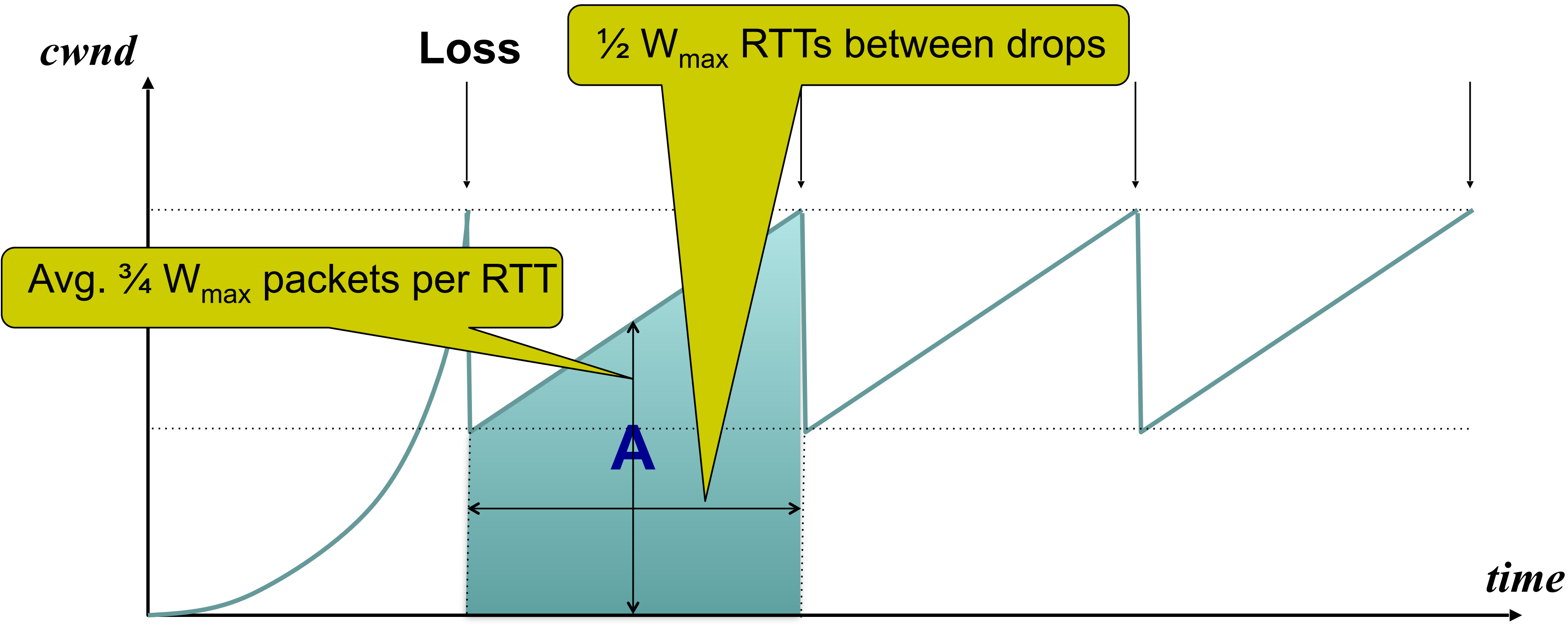


Packet drop rate, $p = \frac{1}{A}$

$$A = \frac{3}{8} W_{max}^2 \rightarrow W = \frac{2\sqrt{2}}{\sqrt{3p}}$$

$$\text{Average Throughput} = \frac{\frac{3}{4} W_{max} \times MSS}{RTT} = \frac{3}{4} \frac{2\sqrt{2}}{\sqrt{3p}} \times MSS$$

A Simple Model for TCP Throughput



Packet drop rate, $p = \frac{1}{A}$

A = $\frac{3}{8} W_{max}^2$ → $W = \frac{2\sqrt{2}}{\sqrt{3p}}$

Average Throughput = $\frac{\frac{3}{4} W_{max} \times MSS}{RTT}$ $= \sqrt{\frac{3}{2}} \frac{MSS}{RTT\sqrt{p}}$

TCP Throughput

- Given a path, what TCP throughput can we expect?

TCP Throughput

- Given a path, what TCP throughput can we expect?
- TCP throughput is proportional to $\frac{1}{\mathrm{RTT}}$ and $\frac{1}{\sqrt{p}}$
  - RTT is path round-trip time and p is the packet loss rate

TCP Throughput

- Given a path, what TCP throughput can we expect?

- TCP throughput is proportional to $\frac{1}{\mathrm{RTT}}$ and $\frac{1}{\sqrt{p}}$
  - RTT is path round-trip time and p is the packet loss rate

- Model makes many simplifying assumptions
  - Ignores slow-start, assumes fixed RTT, isolated loss, *etc.*

TCP Throughput

- Given a path, what TCP throughput can we expect?
- TCP throughput is proportional to $\frac{1}{\mathrm{RTT}}$ and $\frac{1}{\sqrt{p}}$
  - RTT is path round-trip time and p is the packet loss rate
- Model makes many simplifying assumptions
  - Ignores slow-start, assumes fixed RTT, isolated loss, *etc.*
- But leads to some insights (coming up)

Taking Stock: TCP CC

Taking Stock: TCP CC

- (Sender) host based

Taking Stock: TCP CC

- (Sender) host based
- Loss based

Taking Stock: TCP CC

- (Sender) host based
- Loss based
- Adapts every RTT

Taking Stock: TCP CC

- (Sender) host based
- Loss based
- Adapts every RTT
- Starts out in slow start (start small, double every RTT)

Taking Stock: TCP CC

- (Sender) host based
- Loss based
- Adapts every RTT
- Starts out in slow start (start small, double every RTT)
- Adapts based on AIMD (gentle increase, rapid decrease)

Taking Stock: TCP CC

- (Sender) host based
- Loss based
- Adapts every RTT
- Starts out in slow start (start small, double every RTT)
- Adapts based on AIMD (gentle increase, rapid decrease)
- TCP throughput depends on path RTT and loss rate

$$\text{Throughput} = \sqrt{\frac{3}{2}}\,\frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

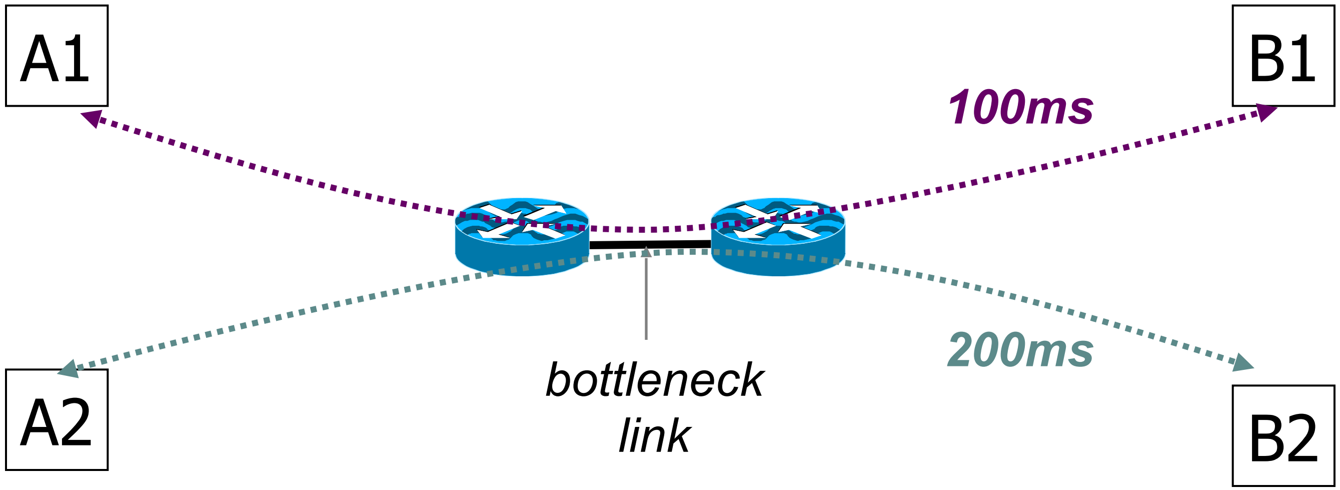
Implications (1): Different RTTs

Throughput = $$\sqrt{\frac{3}{2}}\frac{\mathrm{MSS}}{\mathrm{RTT}\sqrt{p}}$$

Implications (1): Different RTTs

Throughput =

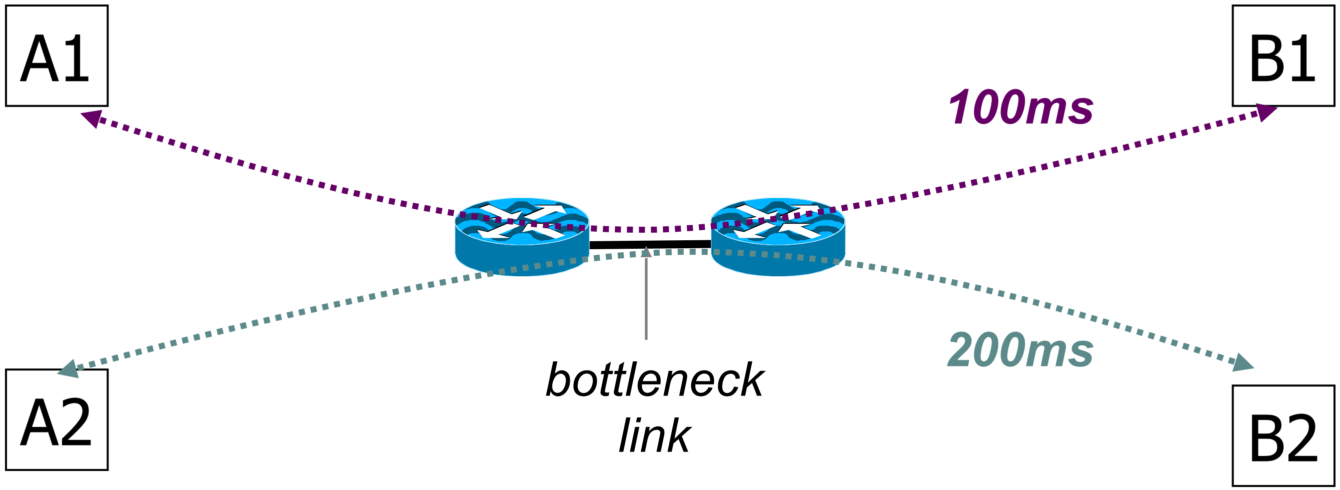
$$\sqrt{\frac{3}{2}}\frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$



Implications (1): Different RTTs

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!



Implications (2): *Rate*-based CC [RFC 5348]

Throughput =

$$\sqrt{\frac{3}{2}}\frac{1}{\mathrm{RTT}\sqrt{p}}$$

Implications (2): *Rate*-based CC [RFC 5348]

Throughput =

$$\sqrt{\frac{3}{2}}\,\frac{1}{\mathrm{RTT}\sqrt{p}}$$

- TCP throughput is "choppy"
  - repeated swings between W/2 to W

Implications (2): ***Rate***-based CC [RFC 5348]

Throughput = $$\sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- TCP throughput is “choppy”
  - repeated swings between W/2 to W
- Some apps would prefer sending at a steady rate
  - e.g., streaming apps

Implications (2): *Rate*-based CC [RFC 5348]

Throughput = $$\sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- 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
  - measure RTT and drop percentage *p*, and set rate accordingly

Implications (2): *Rate*-based CC [RFC 5348]

Throughput = $\sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$

- 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
  - measure RTT and drop percentage *p*, and set rate accordingly

Implications (2): *Rate*-based CC [RFC 5348]

Throughput = $\sqrt{\frac{3}{2}} \frac{1}{\mathrm{RTT}\sqrt{p}}$

- 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
  - measure RTT and 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

(3) Loss not due to congestion?

- TCP will confuse corruption with congestion

(3) Loss not due to congestion?

- TCP will confuse corruption with congestion
- Flow will cut its rate
  - Throughput ~ $\frac{1}{\sqrt{p}}$ even for non-congestion losses!

(4) How do short flows fare?

(4) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB

(4) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB

- Implication (1): many flows never leave slow start!
  - Short flows never attain their fair share
  - In fact, short flows are likely to suffer unduly long transfer times

(4) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (1): many flows never leave slow start!
  - Short flows never attain their fair share
  - In fact, short flows are likely to suffer unduly long transfer times

(4) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (1): many flows never leave slow start!
  - Short flows never attain their fair share
  - In fact, short flows are likely to suffer unduly long transfer times
- Implication (2): too few packets to trigger dupACKs
  - Isolated loss may lead to timeouts
  - At typical timeout values of ~500ms, might severely impact flow completion time

(4) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB

- Implication (1): many flows never leave slow start!
  - Short flows never attain their fair share
  - In fact, short flows are likely to suffer unduly long transfer times

- Implication (2): too few packets to trigger dupACKs
  - Isolated loss may lead to timeouts
  - At typical timeout values of ~500ms, might severely impact flow completion time

- A partial fix: use a higher initial CWND [RFC IW10]

(5) TCP fills up queues → long delays

(5) TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop

(5) TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop

- Recall: loss follows delay (i.e,. queue *must* fill up)

(5) TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop
- Recall: loss follows delay (i.e,. queue *must* fill up)
- Means that delays are large, for *everyone*
  - Consider a flow transferring a 10GB file sharing a bottleneck link with 10 flows transferring 100B

(5) TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop
- Recall: loss follows delay (i.e,. queue *must* fill up)
- Means that delays are large, for *everyone*
  - Consider a flow transferring a 10GB file sharing a bottleneck link with 10 flows transferring 100B
- Problem exacerbated by the trend towards adding large amounts of memory on routers (a.k.a. “bufferbloat”)

(5) TCP fills up queues → long delays

- Focus of Google's BBR algorithm[1]

[1] *BBR: Congestion-Based Congestion Control; Cardwell et al, ACM Queue 2016*

(5) TCP fills up queues → long delays

- Focus of Google's BBR algorithm[1]

- Basic idea (simplified):
  - Sender learns its minimum RTT (~ propagation RTT)
  - Decreases its rate when the observed RTT exceeds the minimum RTT

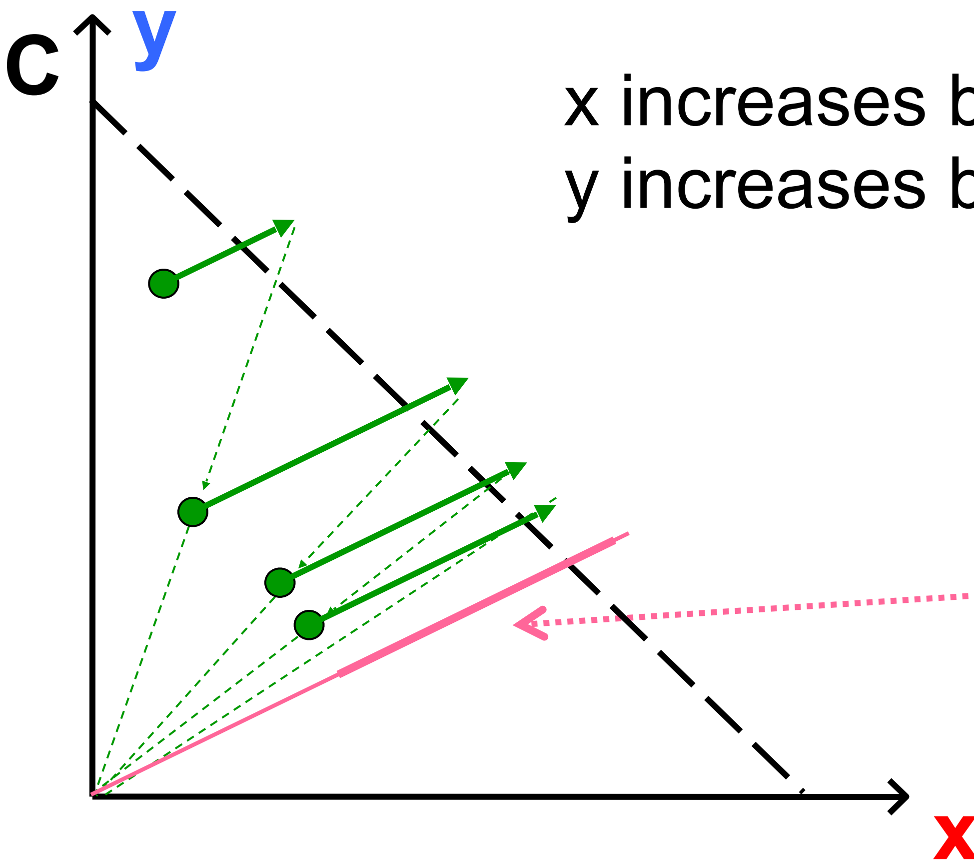
[1] *BBR: Congestion-Based Congestion Control; Cardwell et al, ACM Queue 2016*

(6) Cheating

(6) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT

Increasing CWND Faster



x increases by 2 per RTT
y increases by 1 per RTT

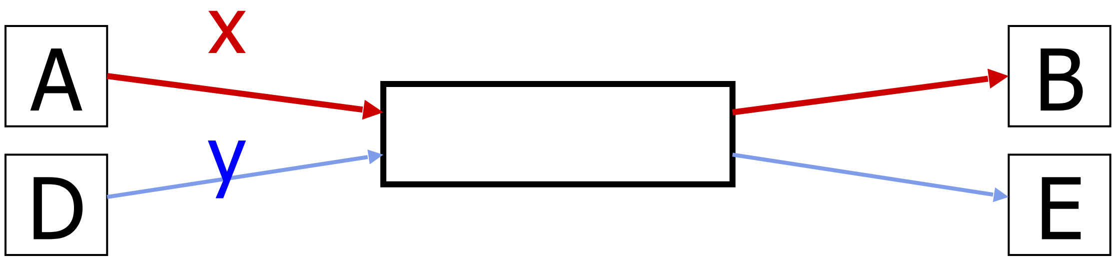
Limit rates:
x = 2y

(6) Cheating

(6) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections

Open Many Connections



Assume

- A starts 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

(6) Cheating

(6) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections
  - Using large initial CWND

Why hasn't the Internet suffered another congestion collapse?

Why hasn't the Internet suffered another congestion collapse?

- Even "cheaters" do back off!
  - Leads to unfairness, not necessarily collapse

Why hasn’t the Internet suffered another congestion collapse?

- Even “cheaters” do back off!
  - Leads to unfairness, not necessarily collapse

MOTHERBOARD
TECH BY VICE

Google’s Network Congestion Algorithm Isn’t Fair, Researchers Say

Why hasn't the Internet suffered another congestion collapse?

- Even "cheaters" do back off!
  - Leads to unfairness, not necessarily collapse

- Hard to say whether unfair behavior is common

MOTHERBOARD
TECH BY VICE

Google's Network Congestion Algorithm Isn't Fair, Researchers Say

(7) CC intertwined with reliability

(7) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
  - CWND adjusted based on ACKs and timeouts
  - Cumulative ACKs and fast retransmit/recovery rules

(7) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
  - CWND adjusted based on ACKs and timeouts
  - Cumulative ACKs and fast retransmit/recovery rules
- Complicates evolution
  - Consider changing from cumulative to selective ACKs
  - A failure of modularity, not layering

(7) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
  - CWND adjusted based on ACKs and timeouts
  - Cumulative ACKs and fast retransmit/recovery rules
- Complicates evolution
  - Consider changing from cumulative to selective ACKs
  - A failure of modularity, not layering
- Sometimes we want CC but not reliability
  - e.g., real-time audio/video

(7) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
  - CWND adjusted based on ACKs and timeouts
  - Cumulative ACKs and fast retransmit/recovery rules
- Complicates evolution
  - Consider changing from cumulative to selective ACKs
  - A failure of modularity, not layering
- Sometimes we want CC but not reliability
  - e.g., real-time audio/video
- Sometimes we want reliability but not CC (?)

(7) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
  - CWND adjusted based on ACKs and timeouts
  - Cumulative ACKs and fast retransmit/recovery rules
- Complicates evolution
  - Consider changing from cumulative to selective ACKs
  - A failure of modularity, not layering
- Sometimes we want CC but not reliability
  - e.g., real-time audio/video
- Sometimes we want reliability but not CC (?)

Recap: TCP problems

Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Could fix many of these with some help from routers!

Recap: TCP problems

Routers tell endhosts about congestion (fine- or coarse-grained feedback)

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Could fix many of these with some help from routers!

Recap: TCP problems

Routers tell endhosts about congestion (fine- or coarse-grained feedback)

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Routers enforce fair sharing

Could fix many of these with some help from routers!

Router-Assisted Congestion Control

- Three ways routers can help
  - Enforce fairness
  - More precise rate adaptation
  - Detecting congestion

How can routers ensure each flow gets its “fair share”?

Fairness: General Approach

Fairness: General Approach

- Consider a single router’s actions

Fairness: General Approach

- Consider a single router's actions
- Router classifies incoming packets into "flows"
  - (For now) let's assume flows are TCP connections

Fairness: General Approach

- Consider a single router’s actions
- Router classifies incoming packets into “flows”
  - (For now) let’s assume flows are TCP connections
- Each flow has its own FIFO queue in router
- Router picks a queue (i.e., flow) in a fair order; transmits packet from the front of the queue

Fairness: General Approach

- Consider a single router's actions
- Router classifies incoming packets into "flows"
  - (For now) let's assume flows are TCP connections
- Each flow has its own FIFO queue in router
- Router picks a queue (i.e., flow) in a fair order; transmits packet from the front of the queue

Fairness: General Approach

- Consider a single router's actions
- Router classifies incoming packets into "flows"
  - (For now) let's assume flows are TCP connections
- Each flow has its own FIFO queue in router
- Router picks a queue (i.e., flow) in a fair order; transmits packet from the front of the queue
- What does "fair" mean exactly?

Max-Min Fairness

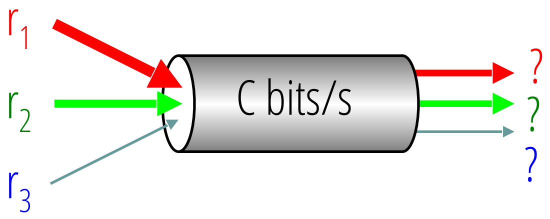
Max-Min Fairness

- Total available bandwidth C

Max-Min Fairness

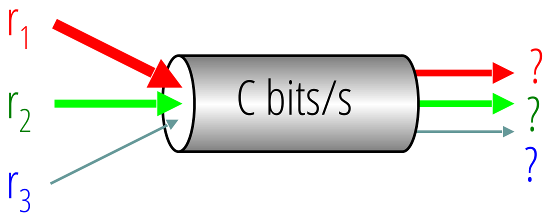
- Total available bandwidth C
- Each flow i has bandwidth demand r_i

Max-Min Fairness



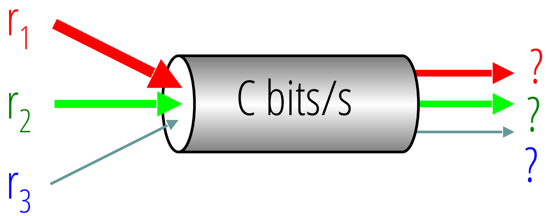
- Total available bandwidth C
- Each flow i has bandwidth demand r_i

Max-Min Fairness



- Total available bandwidth C
- Each flow i has bandwidth demand r_i
- What is a fair allocation a_i of bandwidth to each flow i ?

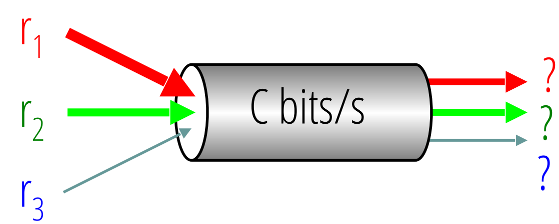
Max-Min Fairness



- Total available bandwidth C
- Each flow i has bandwidth demand r_i
- What is a fair allocation a_i of bandwidth to each flow i ?
- Max-min bandwidth allocations are:

$$a_i = \min(f, r_i)$$

Max-Min Fairness



- Total available bandwidth C
- Each flow i has bandwidth demand r_i
- What is a fair allocation a_i of bandwidth to each flow i ?
- Max-min bandwidth allocations are:

$$a_i = \min(f, r_i)$$

where f is the unique value such that $\mathrm{Sum}(a_i) = C$

Example

Example

- $C = 10$; $N = 3$; $r_1 = 8$, $r_2 = 6$, $r_3 = 2$

Example

- $C = 10$; $N = 3$; $r_1 = 8$, $r_2 = 6$, $r_3 = 2$

- C/N = 10/3 = 3.33 →
  - But r_3's need is only 2
  - Can service all of r_3
  - Allocate 2 to r_3 and remove it from accounting: $C = C – r_3 = 8$; $N = 2$

Example

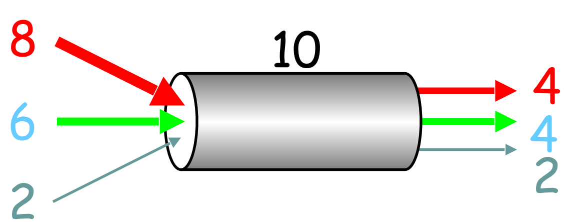
- $C = 10$; $N = 3$; $r_1 = 8$, $r_2 = 6$, $r_3 = 2$
- C/N = 10/3 = 3.33 →
  - But r_3's need is only 2
  - Can service all of r_3
  - Allocate 2 to r_3 and remove it from accounting: $C = C - r_3 = 8$; $N = 2$
- $C/2 = 4$ →
  - Can't service all of r_1 or r_2
  - So hold them to the remaining fair share: $f = 4$

Example

- $C = 10$; $N = 3$; $r_1 = 8$, $r_2 = 6$, $r_3 = 2$

- $C/\mathrm{N} = 10/3 = 3.33 \rightarrow$
  - But r_3's need is only 2
  - Can service all of r_3
  - Allocate 2 to r_3 and remove it from accounting: $C = C - r_3 = 8$; $N = 2$

- $C/2 = 4 \rightarrow$
  - Can't service all of 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

- Property:
  - If you don't get full demand, no one gets more than you

Max-Min Fairness

- Property:
  - If you don’t get full demand, no one gets more than you
- This is what round-robin service gives if all packets are the same size

How do we deal with packets of different sizes?

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin (“fluid flow”)

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin (“fluid flow”)
- Cannot do this in practice!

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin ("fluid flow")

- Cannot do this in practice!

- But we can approximate it
  - This is what **"fair queuing"** routers do

Fair Queuing (FQ)

Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit (called “deadlines”)

Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit (called “deadlines”)
- Then serve packets in increasing order of their deadlines

Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit (called "deadlines")

- Then serve packets in increasing order of their deadlines

- Think of it as an implementation of round-robin extended to the case where not all packets are equal sized

Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit (called “deadlines”)

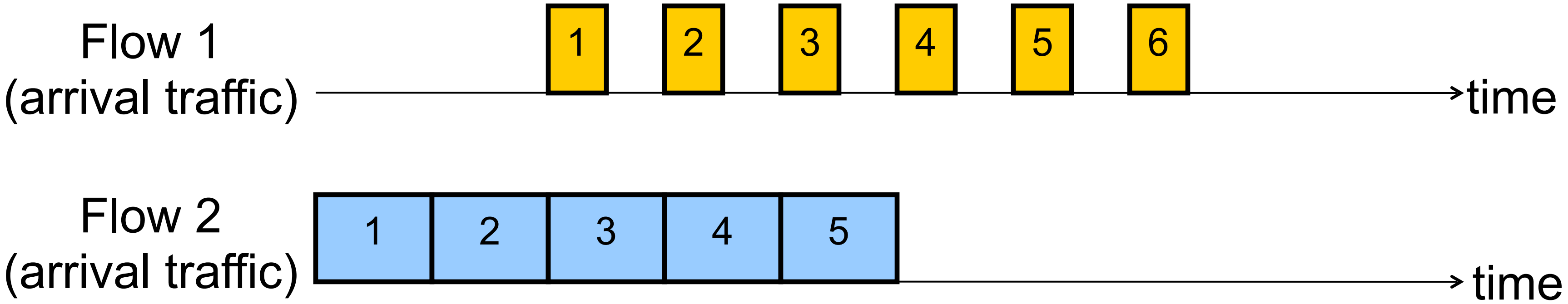
- Then serve packets in increasing order of their deadlines

- Think of it as an implementation of round-robin extended to the case where not all packets are equal sized

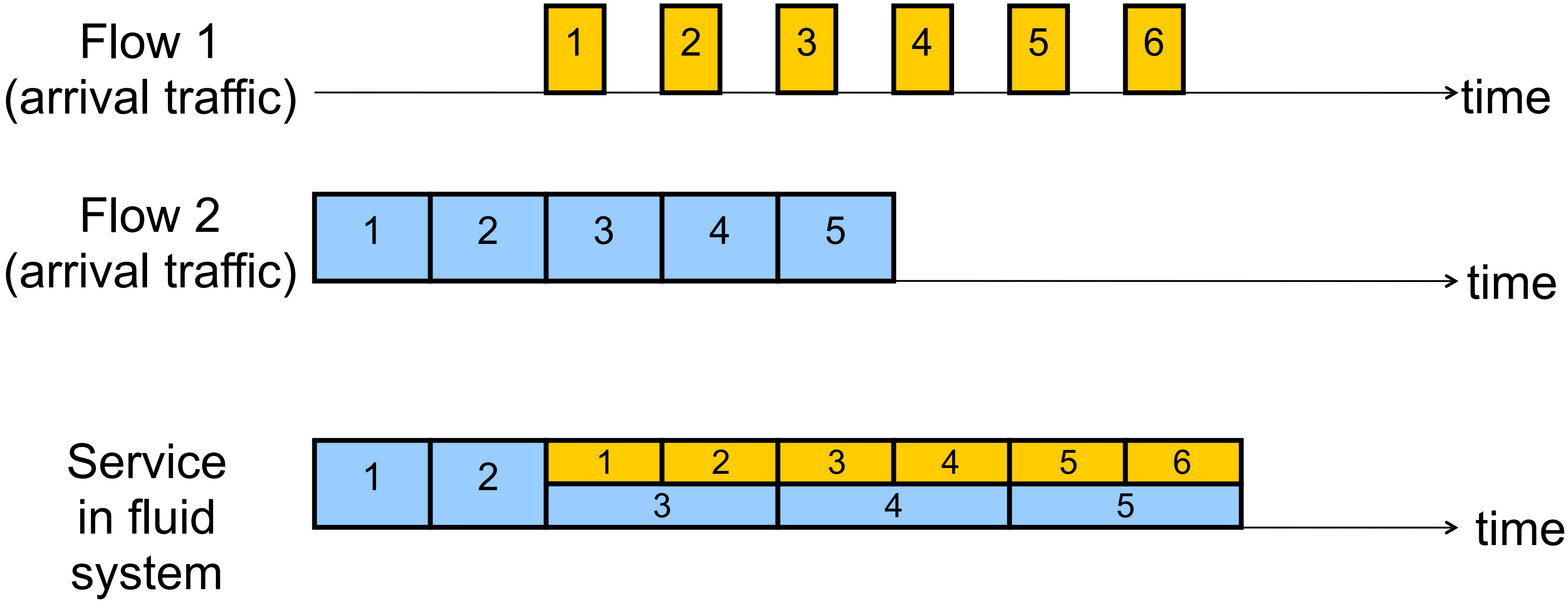
Analysis and Simulation of a Fair Queueing Algorithm

Alan Demers
Srinivasan Keshav†
Scott Shenker

Example

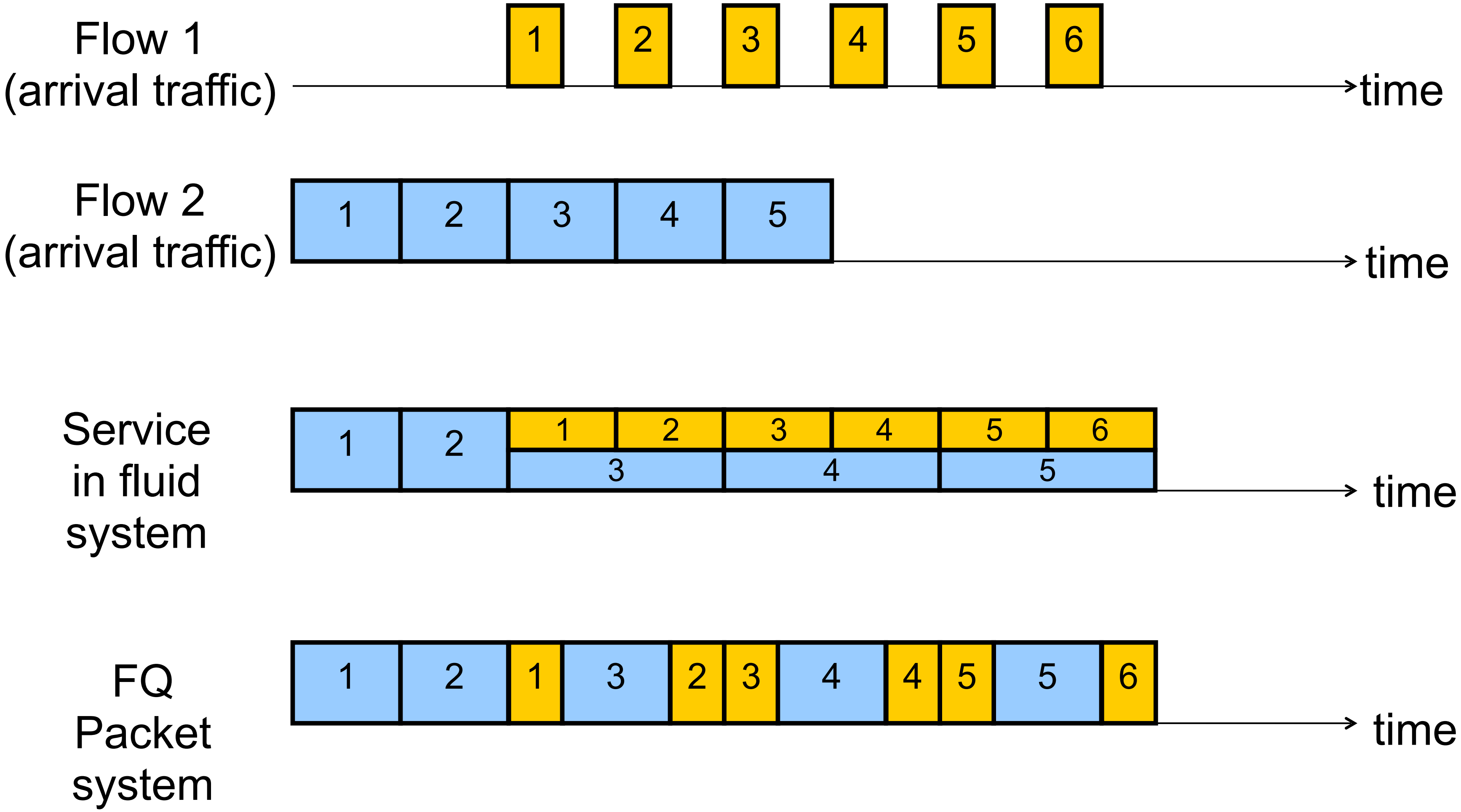

Flow 1
(arrival traffic)
1
2
3
4
5
6
time
Flow 2
(arrival traffic)
1
2
3
4
5
time

Example



Flow 1
(arrival traffic)
1
2
3
4
5
6
time
Flow 2
(arrival traffic)
1
2
3
4
5
time
Service
in fluid
system
1
2
1
2
3
4
5
6
3
4
5
time

Example


Flow 1
(arrival traffic)
1
2
3
4
5
6
time
Flow 2
(arrival traffic)
1
2
3
4
5
time
Service
in fluid
system
1
2
1
2
3
4
5
6
3
4
5
time
FQ
Packet
system
1
2
1
3
2
3
4
4
5
5
6
time

FQ vs. FIFO

FQ vs. FIFO

- FQ advantages:
  - Isolation: cheating flows don't benefit
  - Bandwidth share does not depend on RTT
  - Flows can pick any rate adjustment scheme they want

FQ vs. FIFO

- FQ advantages:
  - Isolation: cheating flows don't benefit
  - Bandwidth share does not depend on RTT
  - Flows can pick any rate adjustment scheme they want

FQ vs. FIFO

- FQ advantages:
  - Isolation: cheating flows don't benefit
  - Bandwidth share does not depend on RTT
  - Flows can pick any rate adjustment scheme they want

- Disadvantages:
  - More complex than FIFO: per flow queue/state, additional per-packet book-keeping
  - Still only a partial solution (coming up)

Fair Queuing In Practice

Fair Queuing In Practice

- “Pure” FQ too complex to implement at high speeds

Fair Queuing In Practice

- "Pure" FQ too complex to implement at high speeds
- But several approximations exist
  - E.g., Deficit Round Robin (DRR)

Fair Queuing In Practice

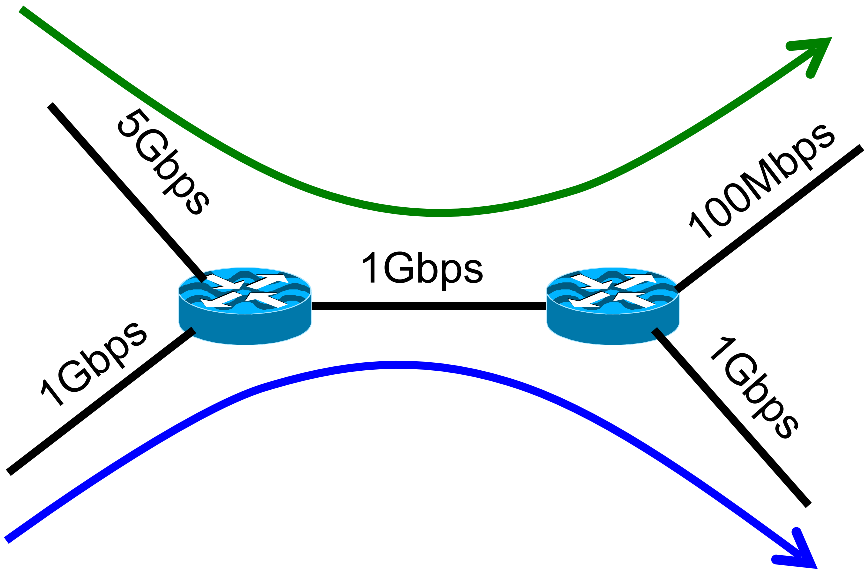
- "Pure" FQ too complex to implement at high speeds
- But several approximations exist
  - E.g., Deficit Round Robin (DRR)
- Today:
  - Routers typically implement approximate FQ (e.g., DRR)
  - For a small number of queues
  - Commonly used for coarser-grained isolation (e.g., for select customer prefixes) rather than per-flow isolation

FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion

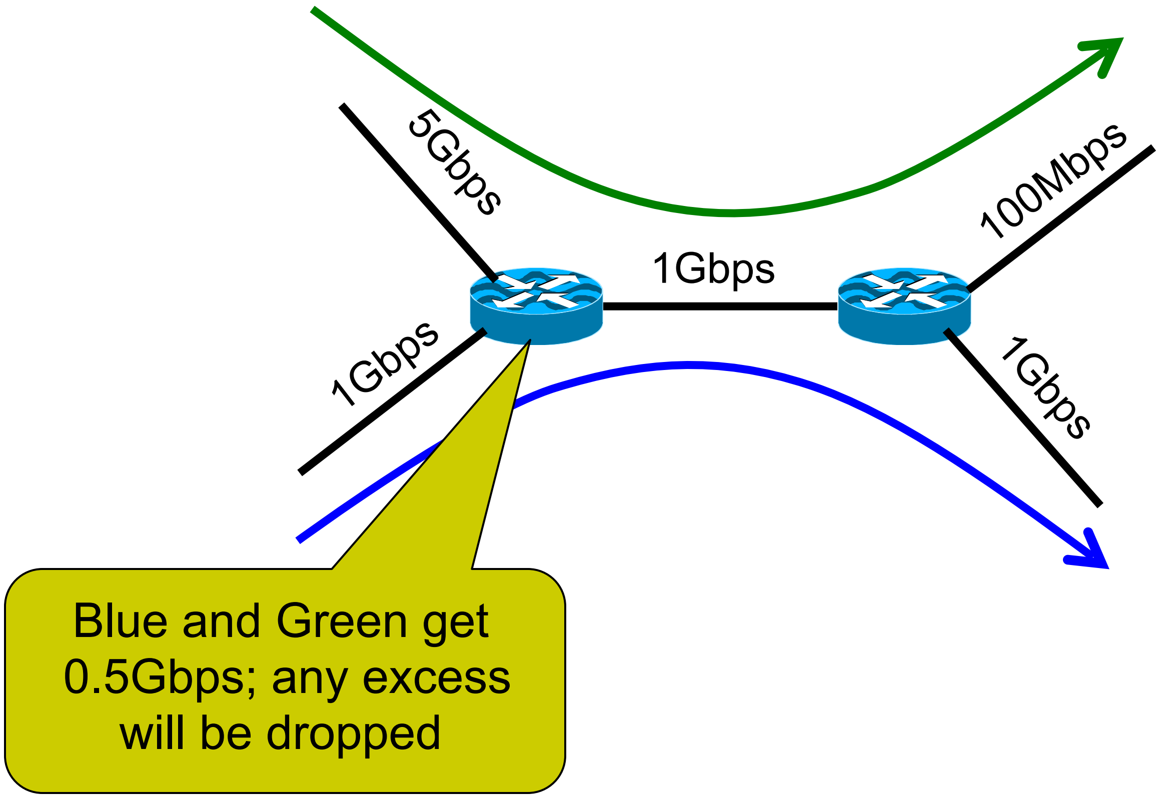
FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



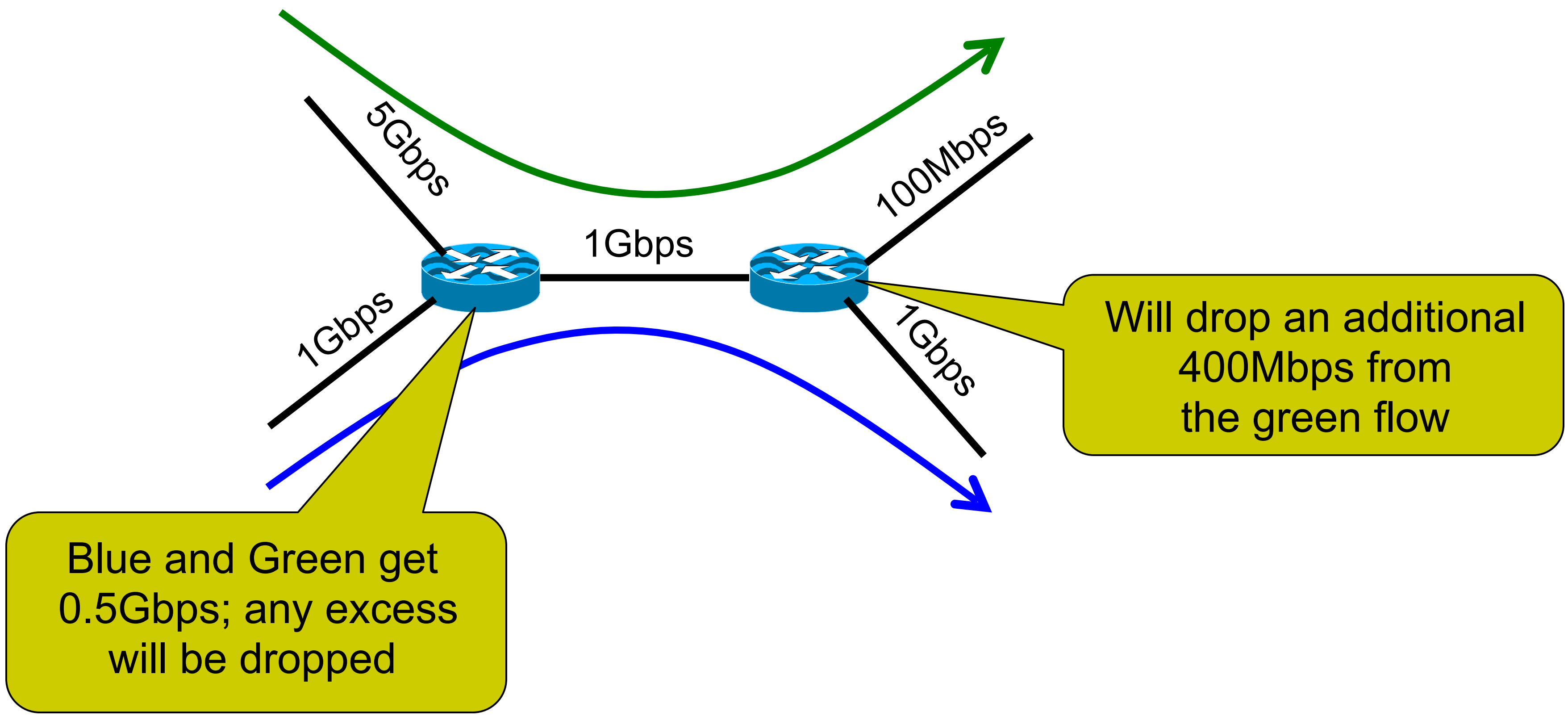
FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



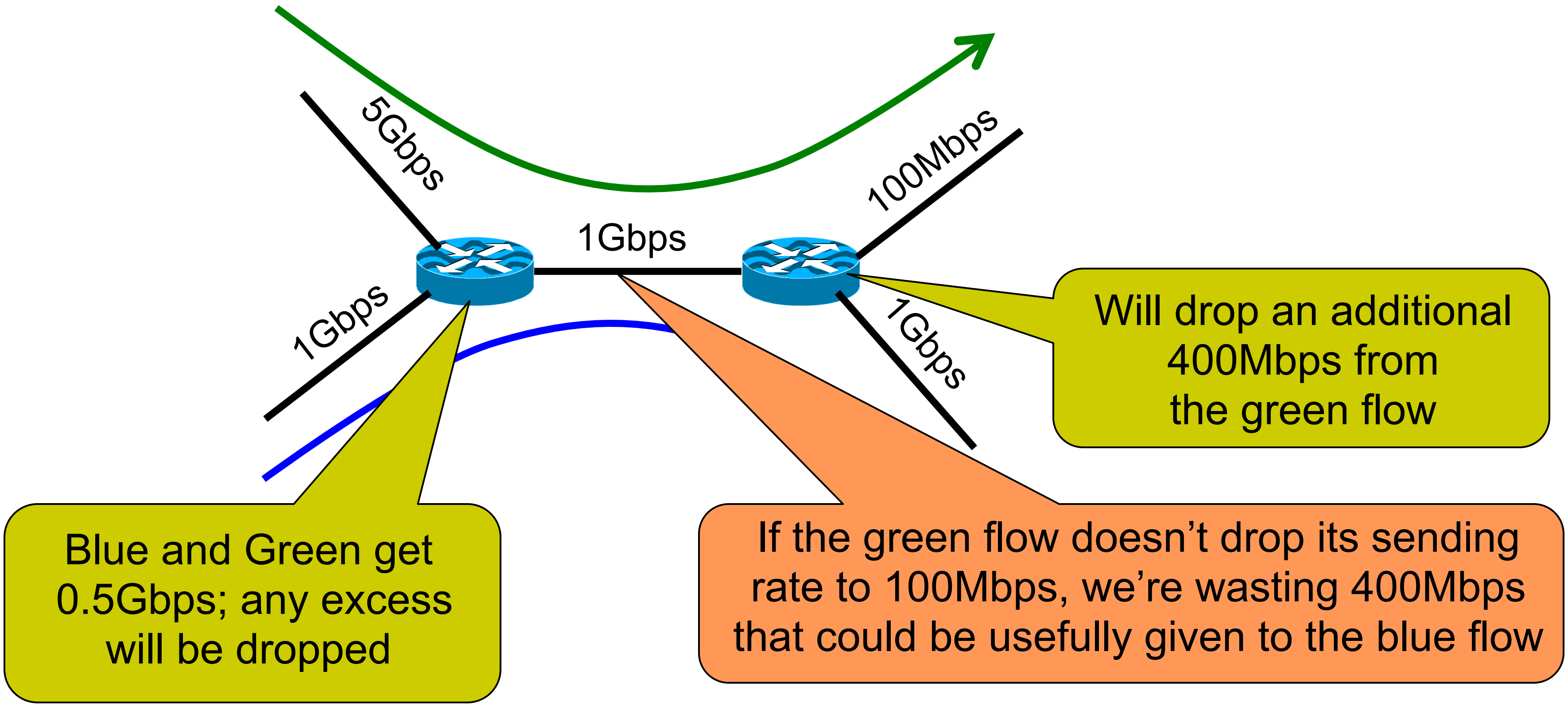
FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion
- FQ's benefit is its resilience (to cheating, variations in RTT, details of delay, reordering, *etc.*)

FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion
- FQ's benefit is its resilience (to cheating, variations in RTT, details of delay, reordering, *etc.*)
- But congestion and packet drops still occur

FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion
- FQ's benefit is its resilience (to cheating, variations in RTT, details of delay, reordering, *etc.*)
- But congestion and packet drops still occur
- And we still want end-hosts to discover/adapt to their fair share!

Per-flow fairness is a controversial goal

Per-flow fairness is a controversial goal

- What if you have 8 flows, and I have 4?
  - Why should you get twice the bandwidth

Per-flow fairness is a controversial goal

- What if you have 8 flows, and I have 4?
  - Why should you get twice the bandwidth
- What if your flow goes over 4 congested hops, and mine only goes over 1?
  - Shouldn't you be penalized for using more of scarce bandwidth?
- And at what granularity do we really want fairness?
  - TCP connection? Source-Destination pair? Source?

Per-flow fairness is a controversial goal

- What if you have 8 flows, and I have 4?
  - Why should you get twice the bandwidth
- What if your flow goes over 4 congested hops, and mine only goes over 1?
  - Shouldn't you be penalized for using more of scarce bandwidth?
- And at what granularity do we really want fairness?
  - TCP connection? Source-Destination pair? Source?
- Nonetheless, FQ/DRR is a great way to ensure **isolation**
  - Avoiding starvation even in the worst cases

Router-Assisted Congestion Control

- Three ways routers can help
  - Enforce fairness
  - More precise rate adaptation
  - Detecting congestion

Why not just let routers tell endhosts what rate they should use?

Why not just let routers tell endhosts what rate they should use?

- Packets carry "rate field"
- Routers insert a flow's fair share *f* in packet header

Why not just let routers tell endhosts what rate they should use?

- Packets carry “rate field”
- Routers insert a flow’s fair share *f* in packet header
- End-hosts set sending rate (or window size) to *f*

Router-Assisted Congestion Control

- Three ways routers can help
  - Enforce fairness
  - More precise rate adaptation
  - Detecting congestion

Explicit Congestion Notification (ECN)

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for *when* routers set the bit
  - Tradeoff between link utilization and packet delay

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for *when* routers set the bit
  - Tradeoff between link utilization and packet delay
- Host can react as though it was a drop

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for *when* routers set the bit
  - Tradeoff between link utilization and packet delay
- Host can react as though it was a drop

- Advantages:
  - Don't confuse corruption with congestion
  - Early indicator of congestion → avoid delays
  - Lightweight to implement

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for *when* routers set the bit
  - Tradeoff between link utilization and packet delay
- Host can react as though it was a drop

- Advantages:
  - Don't confuse corruption with congestion
  - Early indicator of congestion → avoid delays
  - Lightweight to implement

- Today:
  - Widely implemented in routers
  - Commonly used in datacenters (e.g., Azure)

Recap: Router-Assisted CC

- FQ: routers *enforce* per-flow fairness
- RCP: routers *inform* endhosts of their fair share
- ECN: routers set “I’m congested” bit in packets

Perspective: Router-Assisted CC

Perspective: Router-Assisted CC

- Can be highly effective, approaching optimal perf.

Perspective: Router-Assisted CC

- Can be highly effective, approaching optimal perf.
- But deployment is more challenging
  - Need support at hosts and routers
  - Some require more complex book-keeping at routers
  - Some require deployment at *every* router

Perspective: Router-Assisted CC

- Can be highly effective, approaching optimal perf.
- But deployment is more challenging
  - Need support at hosts and routers
  - Some require more complex book-keeping at routers
  - Some require deployment at *every* router
- Though worth revisiting in datacenter contexts

Perspective: TCP CC

Perspective: TCP CC

- Not perfect, a little ad-hoc

Perspective: TCP CC

- Not perfect, a little ad-hoc
- But deeply practical/deployable

Perspective: TCP CC

- Not perfect, a little ad-hoc
- But deeply practical/deployable
- Good enough to have raised the bar for the deployment of new, more optimal, approaches

Perspective: TCP CC

- Not perfect, a little ad-hoc
- But deeply practical/deployable
- Good enough to have raised the bar for the deployment of new, more optimal, approaches
- Though datacenters are the CC agenda
  - different needs and constraints (future lecture)