TCP Congestion Control (Contd.) Critiques & Advanced Techniques

CPSC 433/533, Spring 2021
Anurag Khandelwal

Administrivia

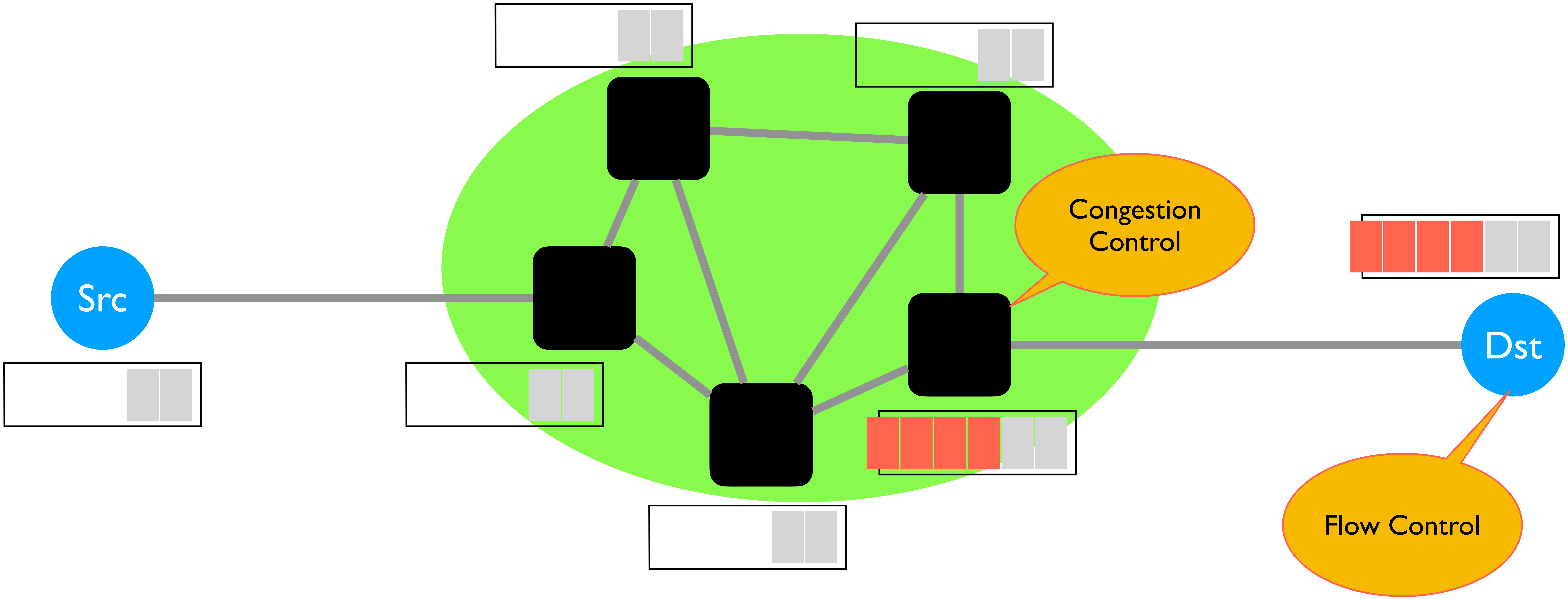
Administrivia

- **Project 1 grades out**
  - Output logs for the tests we ran have been provided
  - We will **not** reveal the actual test cases (code)

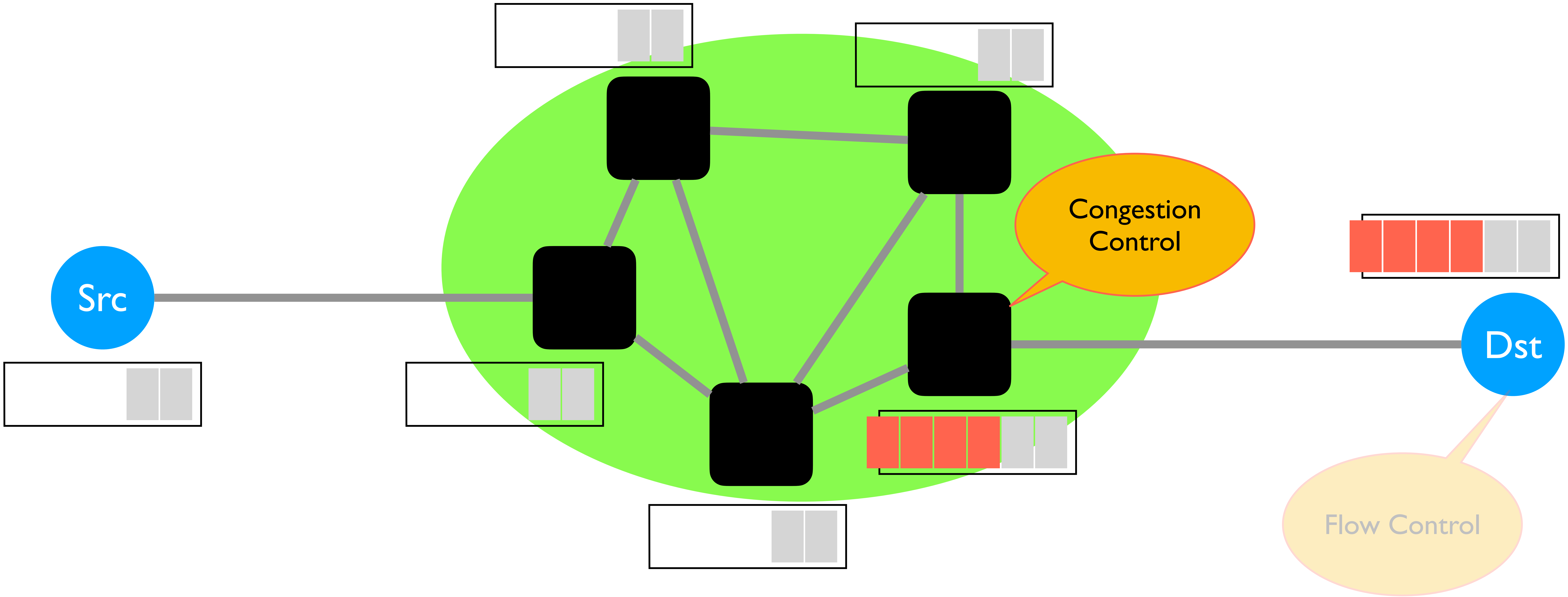
Administrivia

- **Project 1 grades out**
  - Output logs for the tests we ran have been provided
  - We will **not** reveal the actual test cases (code)
- **Midterm drawing close!**
  - We will have a review session next week
  - Be clear on concepts; clarify during OH if you don't understand them!
  - There are some topics that are covered in lectures & not in book, and vice versa
    - Only tested on things taught in class

How does TCP deal with buffer limits?



How does TCP deal with buffer limits?



Congestion: Two Basic Questions

Congestion: Two Basic Questions

- **How does the sender detect congestion?**

Losses: dupACKs, timeouts

Congestion: Two Basic Questions

- **How does the sender detect congestion?**
- **How does the sender adjust its sending rate?**
  - To address three issues:
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

Losses: dupACKs, timeouts

Congestion: Two Basic Questions

- **How does the sender detect congestion?**
- **How does the sender adjust its sending rate?**
  - To address three issues:
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

Losses: dupACKs, timeouts

Slow Start

Congestion: Two Basic Questions

- **How does the sender detect congestion?**
- **How does the sender adjust its sending rate?**
  - To address three issues:
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

Losses: dupACKs, timeouts

Slow Start

AIMD

TCP Congestion Control Details

Implementation

Implementation

- **State at sender**
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)

Implementation

- **State at sender**
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
- **Events:**
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

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

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

- **Else:**
  - CWND += 1/CWND

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

- **Else:**
  - CWND += 1/CWND

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

Event: ACK (new data)

- **If CWND < ssthresh**
  - CWND += 1

Slow Start Phase

- **Else:**
  - CWND += 1/CWND

Congestion Avoidance Phase
(Additive Increase)

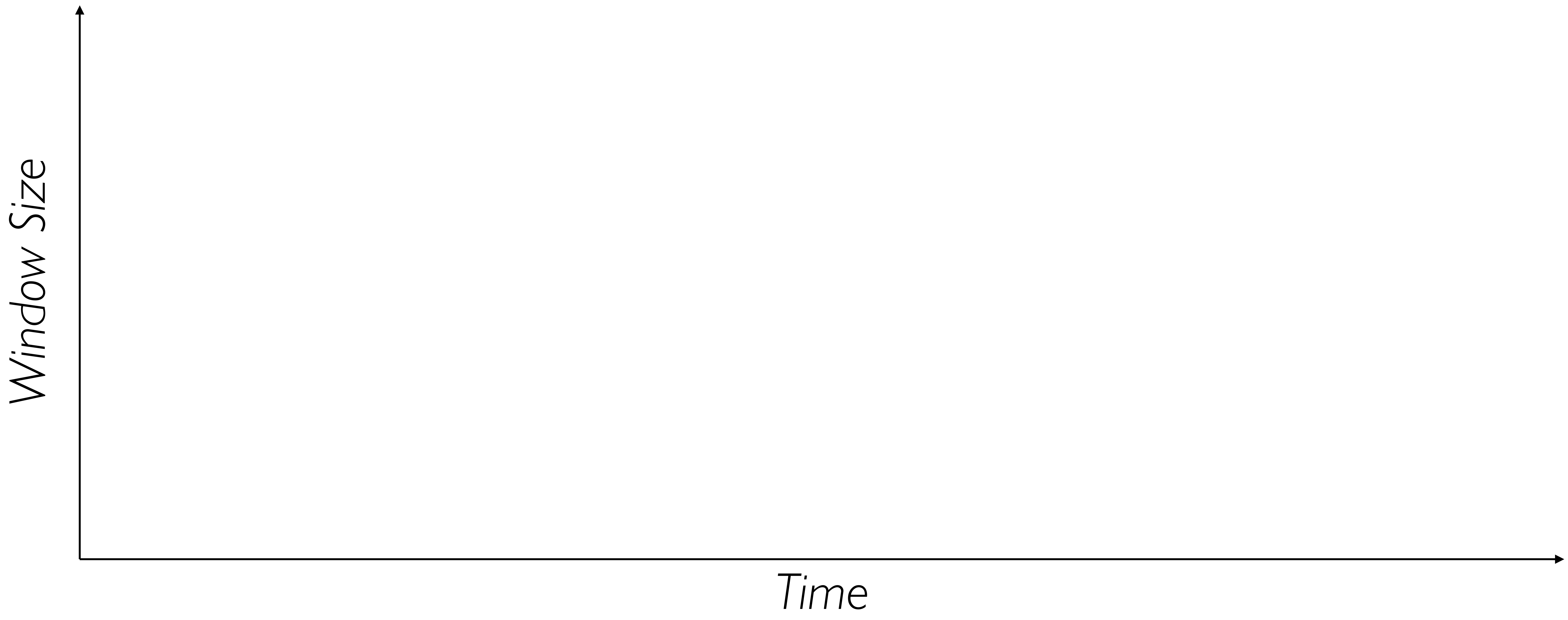
Event: Timeout

- **On timeout**
  - ssthresh ← CWND/2
  - CWND ← 1

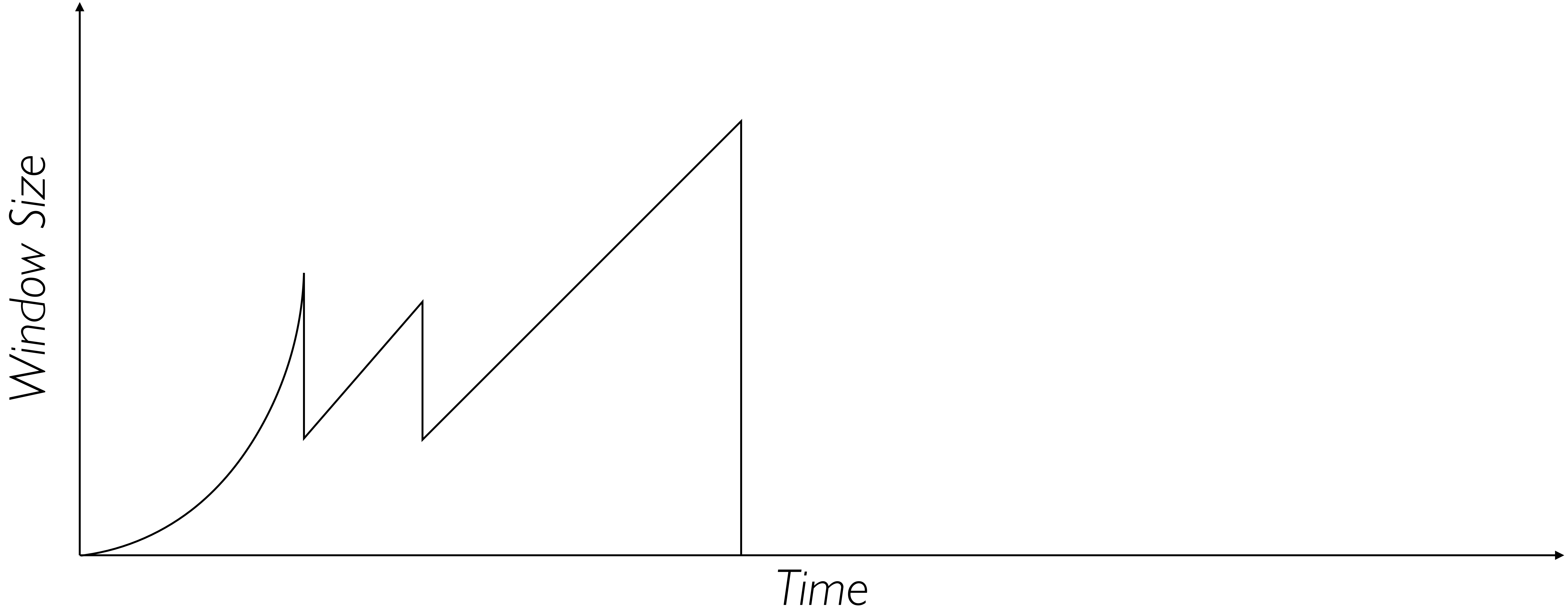
Event: dupACK

- **dupACKcount++**
- **If dupACKcount = 3 /* Fast retransmit */**
  - ssthresh ← CWND/2
  - CWND ← CWND/2

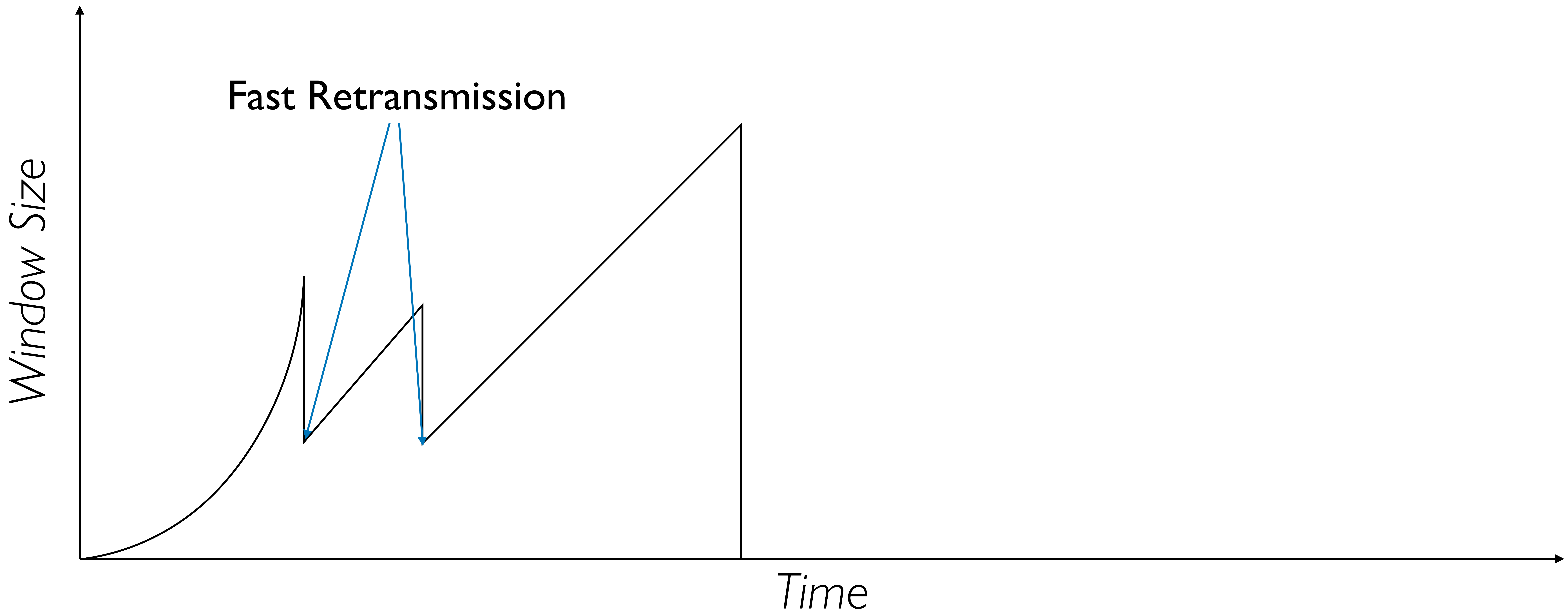
Example



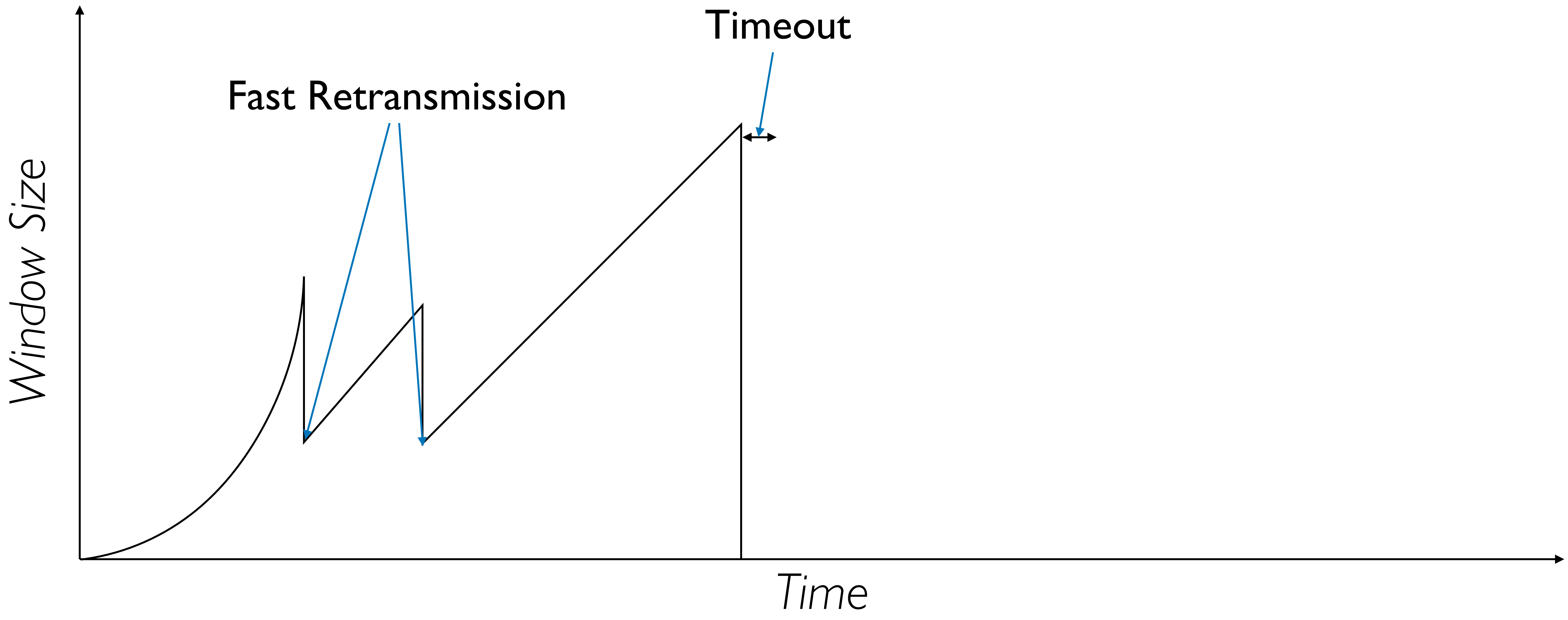
Example



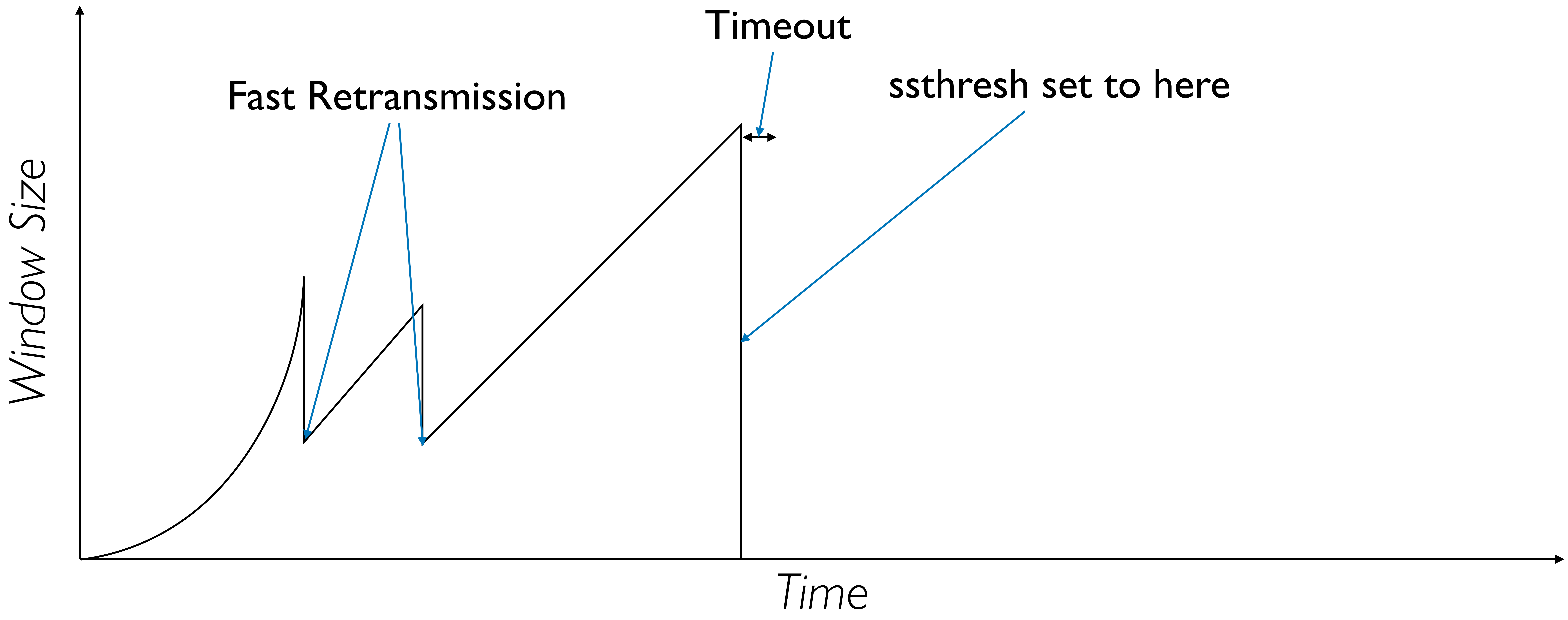
Example



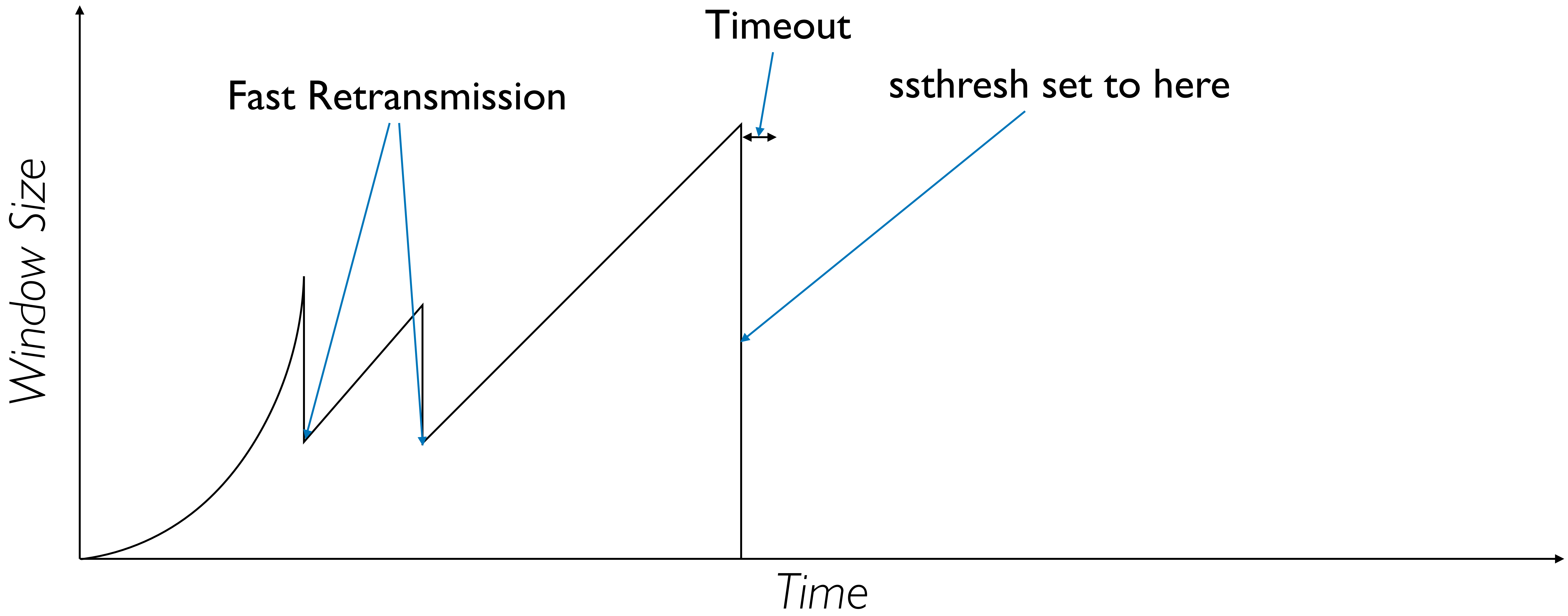
Example



Example

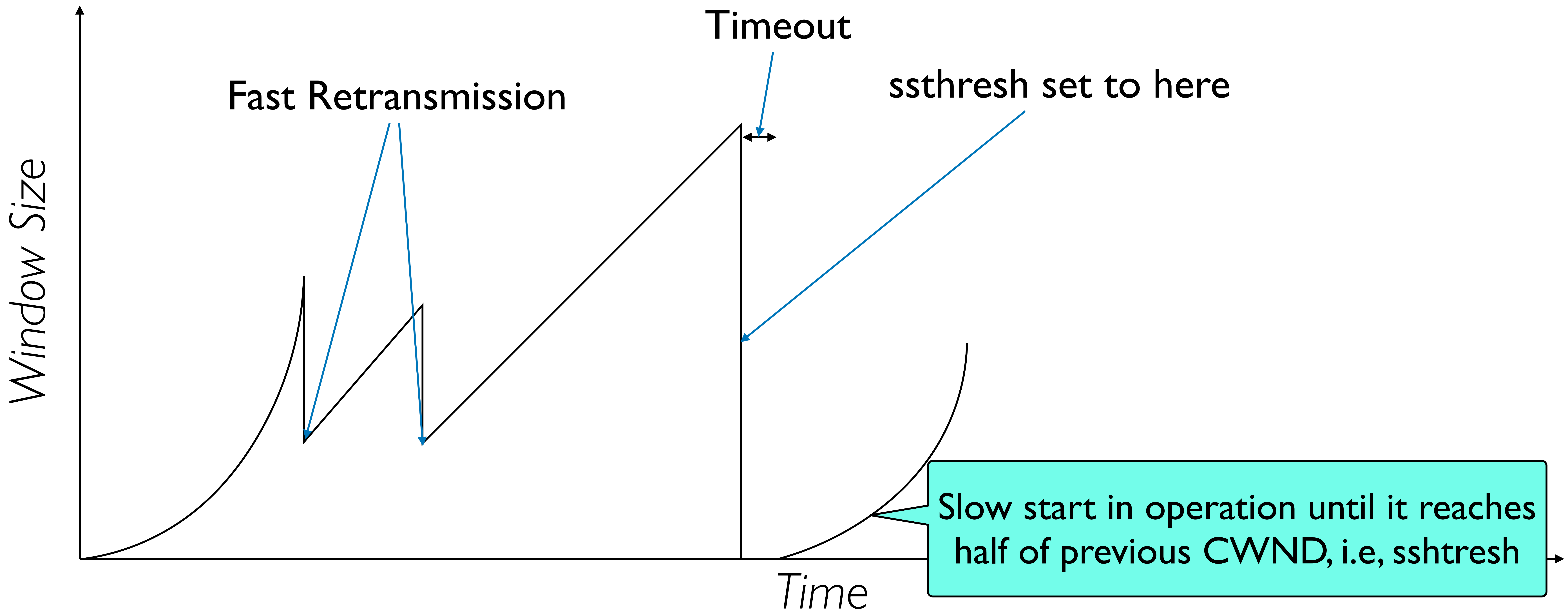

Timeout
ssthresh set to here
Fast Retransmission
Window Size
Time

Example



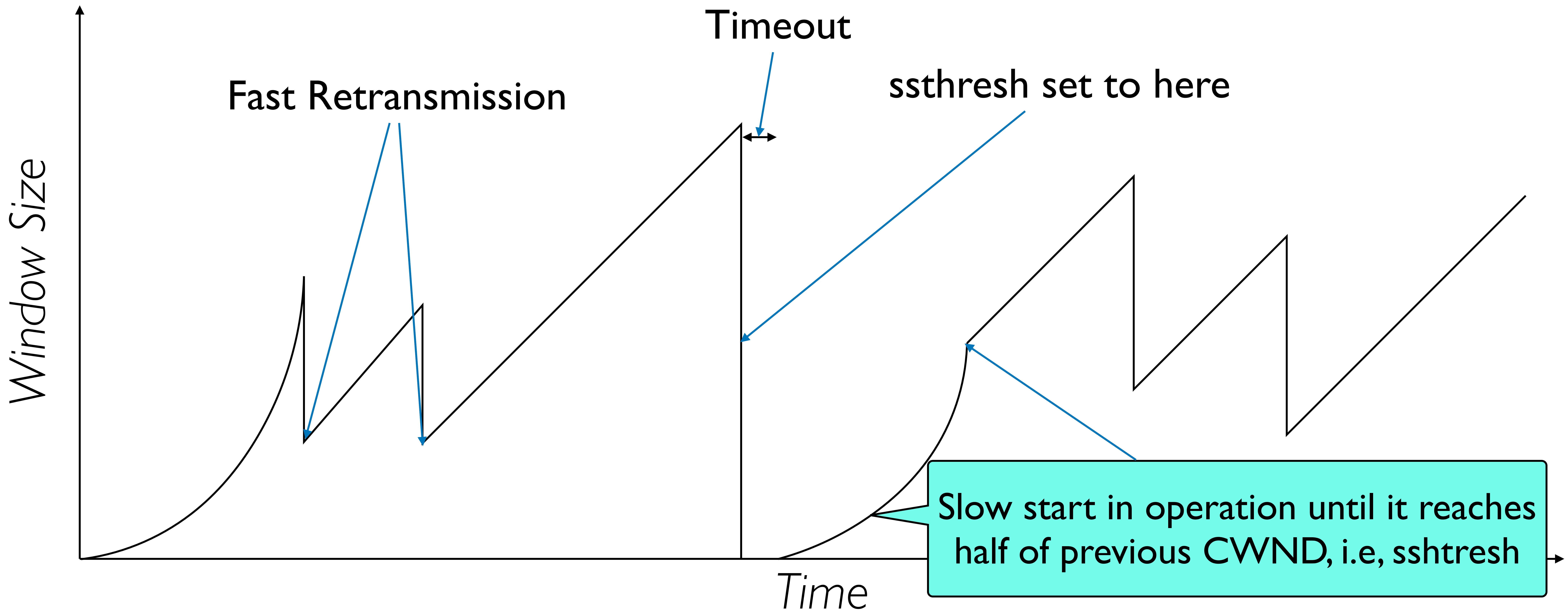
Slow Start restart: Go back to CWND=1, but take advantage of knowing the previous value of CWND

Example



Slow Start restart: Go back to CWND=1, but take advantage of knowing the previous value of CWND

Example



Slow Start restart: Go back to CWND=1, but take advantage of knowing the previous value of CWND

One Final Phase: Fast Recovery

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

Example

- **Consider a TCP connection with:**
  - CWND = 10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have sequence number 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

Timeline

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

Timeline

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

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)

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

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 + 1/5 (no xmit)

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 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)

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 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)

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 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)

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 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)

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 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)

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 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets

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 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

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 for each additional duplicate ACK

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 for each additional duplicate ACK
- **Exit fast recovery after receiving new ACK**
  - Set CWND = ssthresh

Example

- **Consider a TCP connection with:**
  - CWND = 10 packets
  - Last ACK was for packet #101
    - i.e., receiver expecting next packet to have seqno 101
- **10 packets [101, 102, 103, ..., 110] are in flight**
  - Packet 101 is dropped

Timeline

Timeline

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

Timeline

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

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)

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

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
- ACK 101 (due to 105) cwnd=9 (no xmit)

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
- ACK 101 (due to 105) cwnd=9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)

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
- 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

- 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
- 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

- 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
- 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

- 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
- 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

- 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
- 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

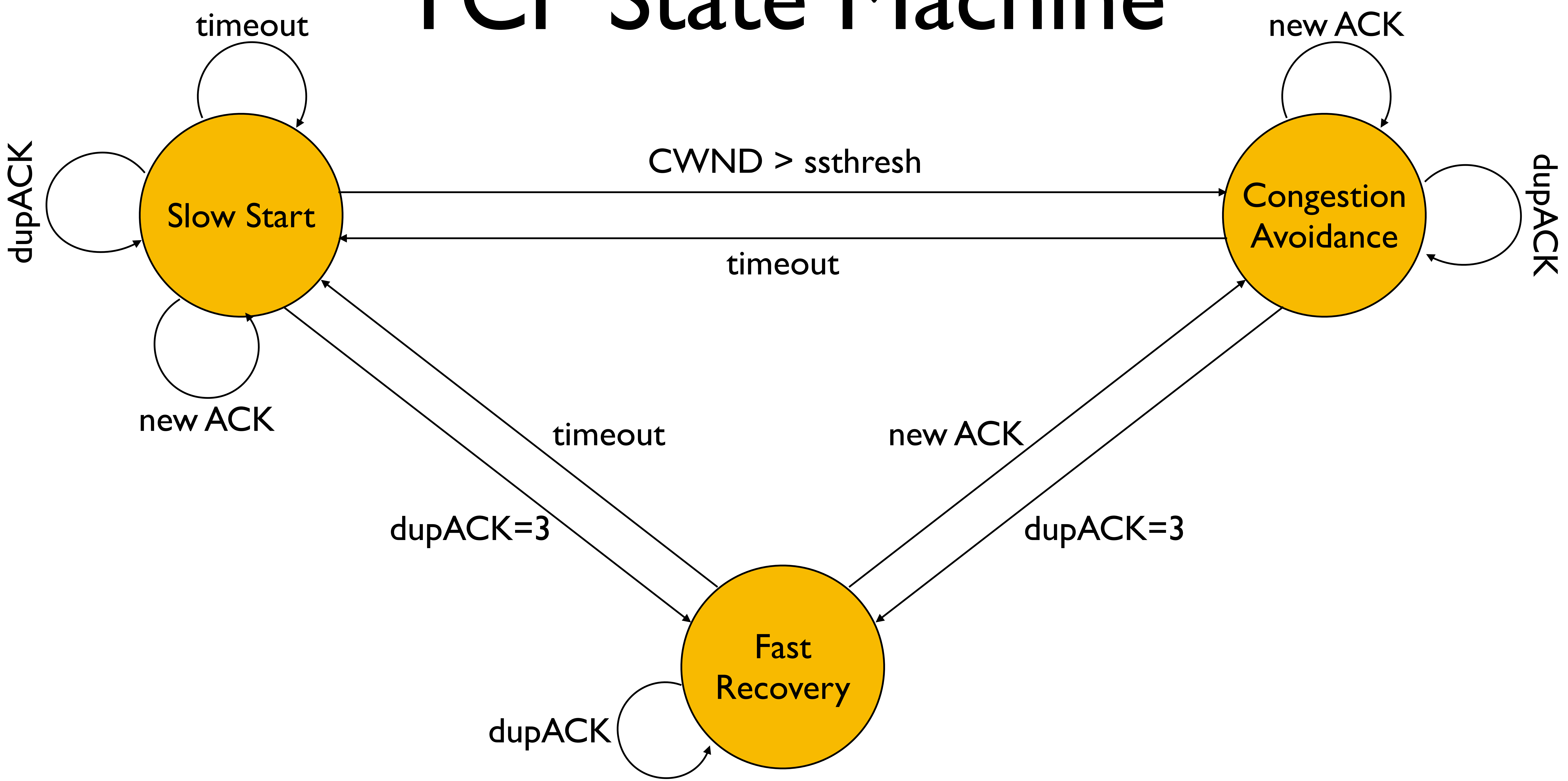
Timeline

- 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
- 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

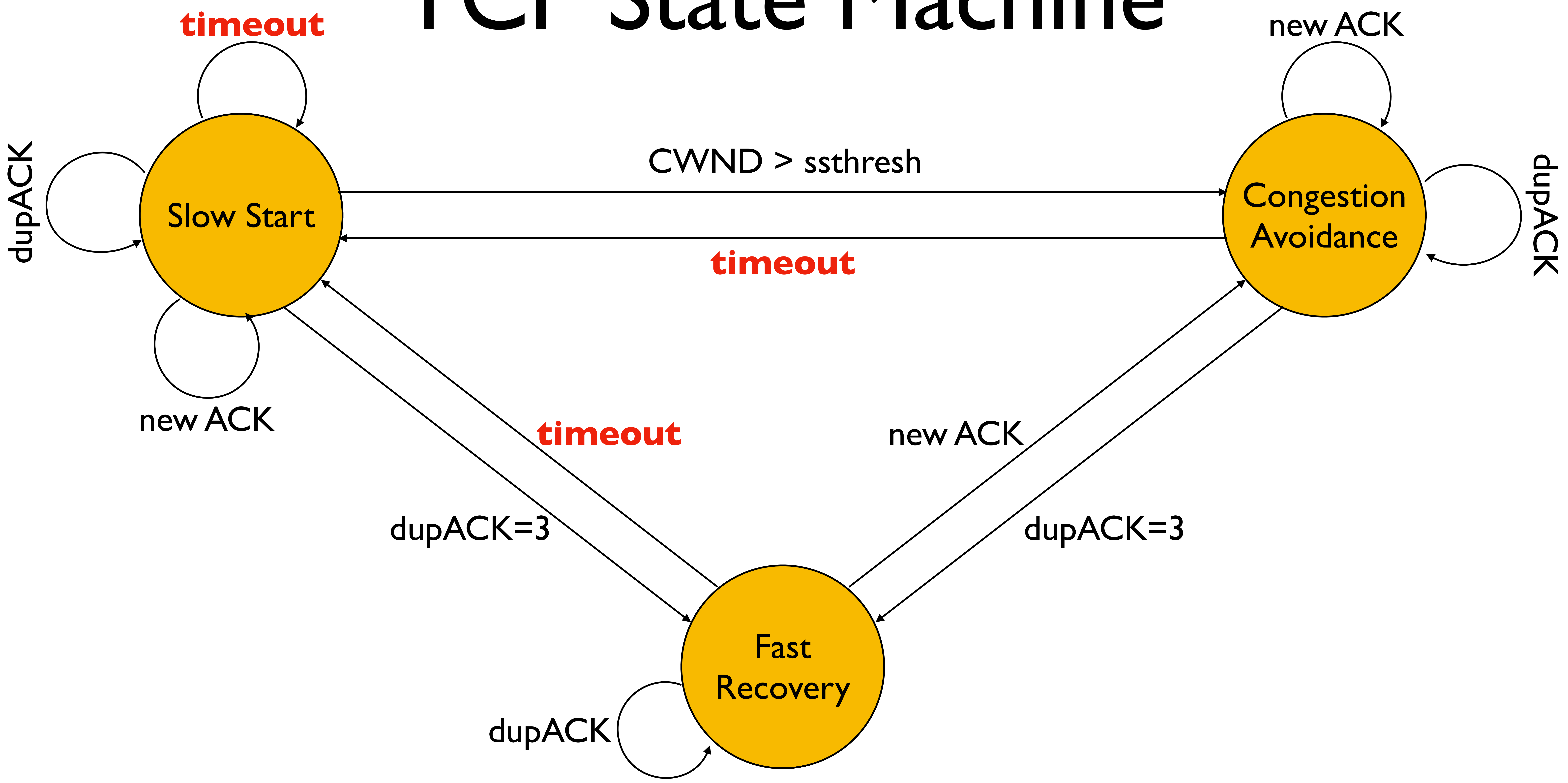
Timeline

- 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
- 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
- ACK 112 (due to 111) cwnd=5 + 1/5 ← back in congestion avoidance

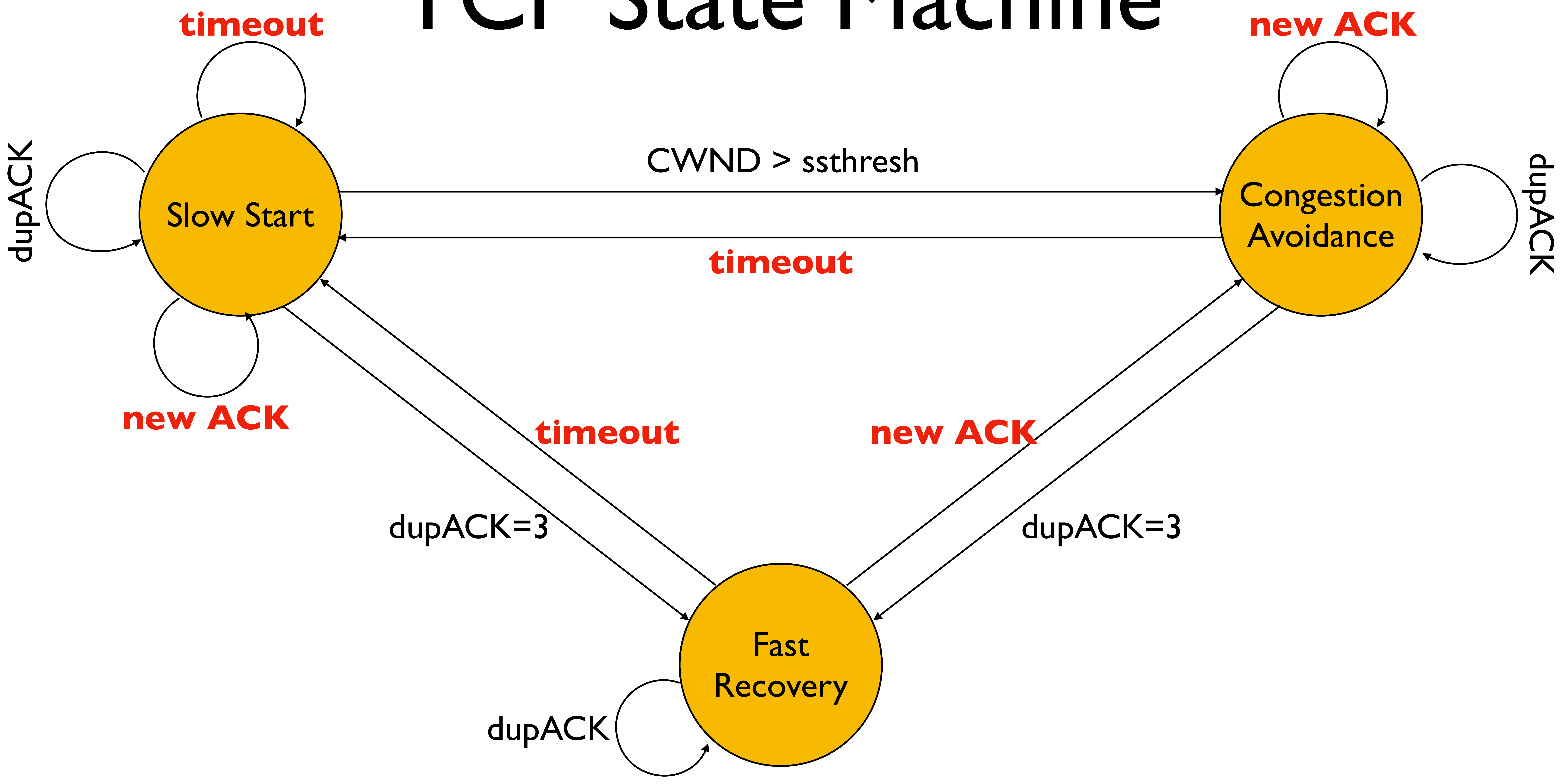
TCP State Machine



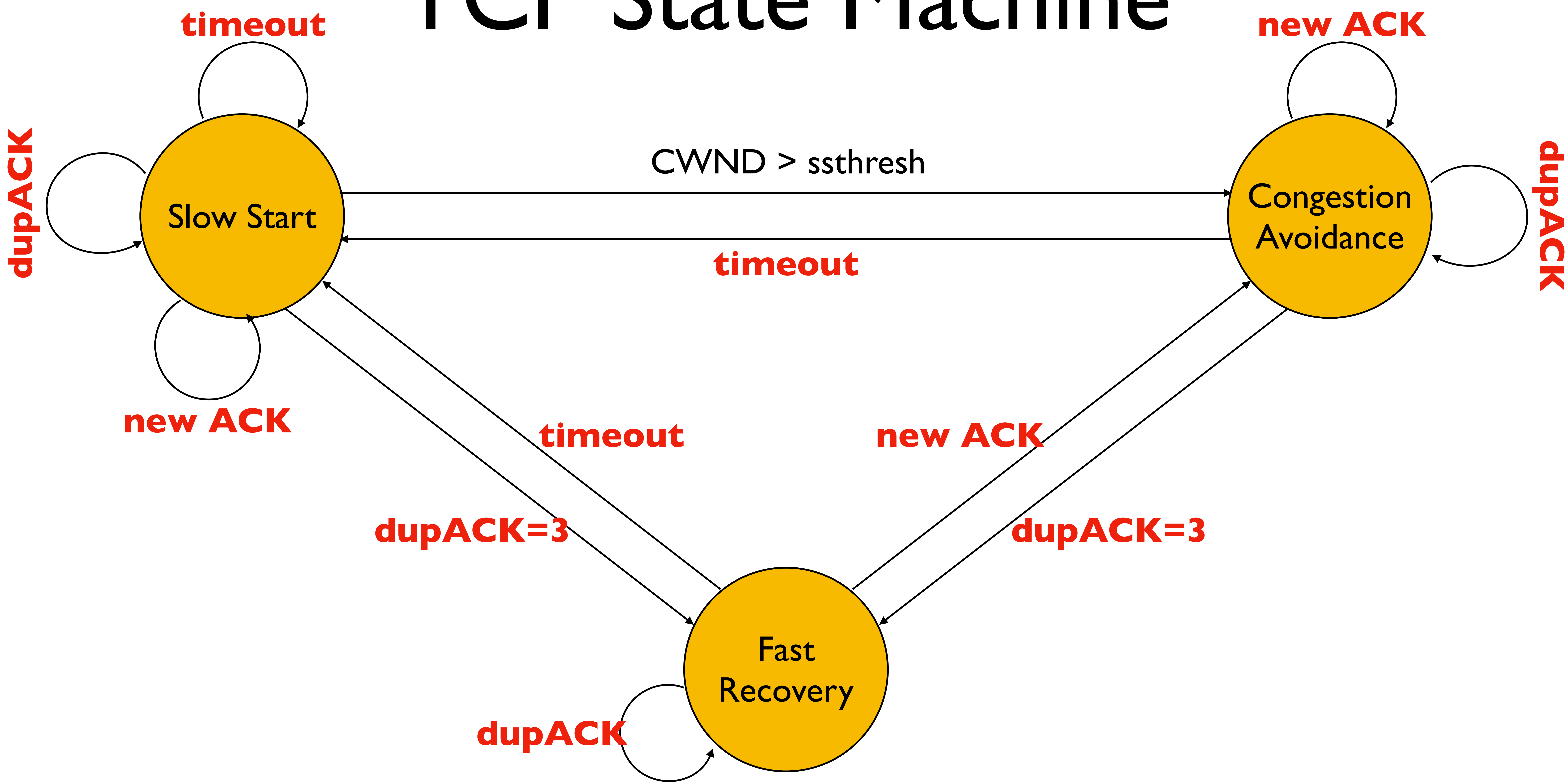
TCP State Machine



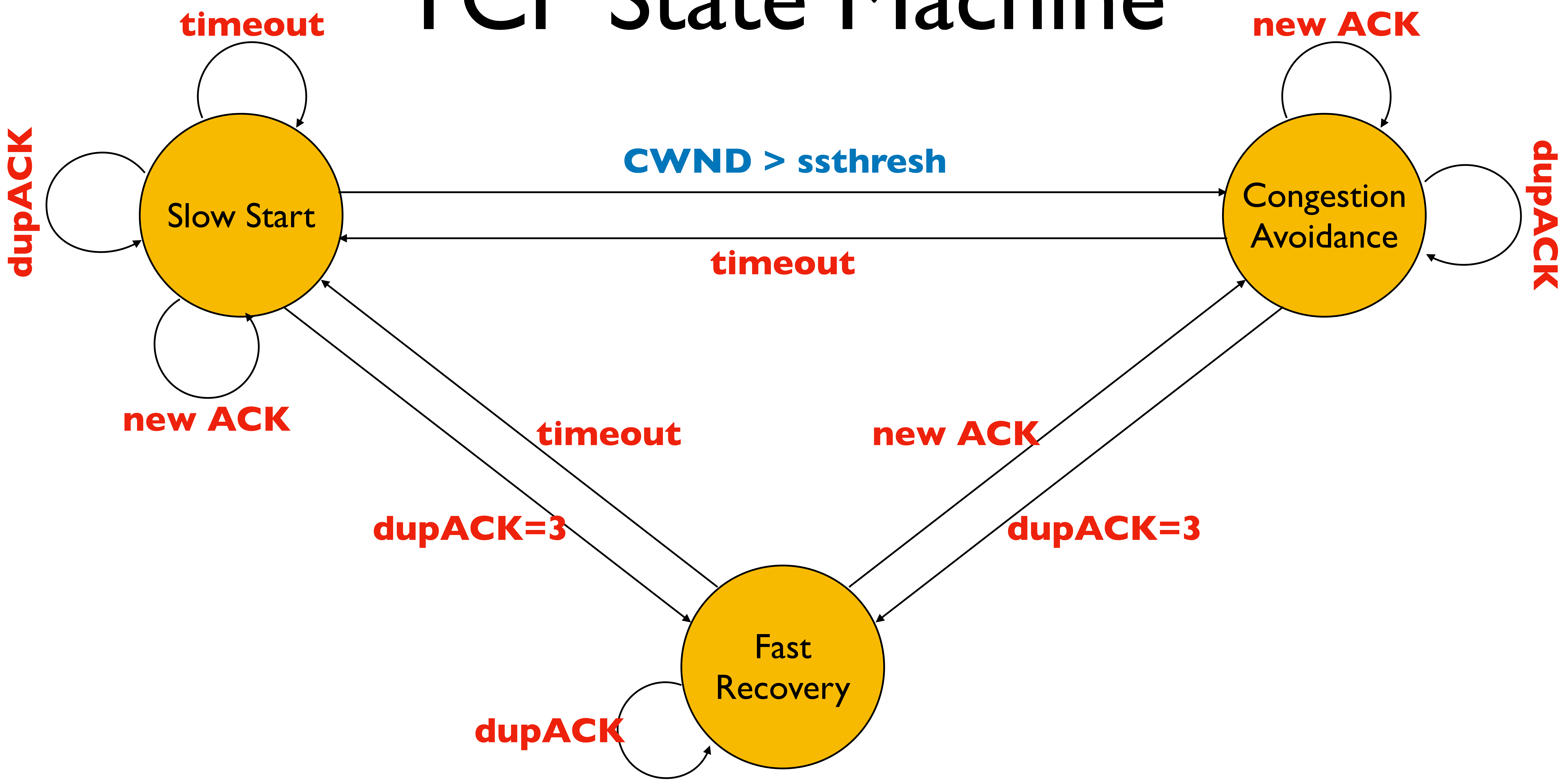
TCP State Machine



TCP State Machine



TCP State Machine



TCP Flavors

TCP Flavors

- **TCP Tahoe**
  - CWND=1 on triple dupACK

TCP Flavors

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

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 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

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

    Our Default Assumption
- **TCP SACK**
  - Incorporates selective acknowledgements

Taking Stock

- The concepts underlying TCP are simple
  - Acknowledgements
  - Timers
  - Sliding Windows
  - Buffer Management
  - Sequence Numbers

Taking Stock

- The concepts underlying TCP are simple
- But tricky in the details
  - How do we set timers
  - What is the seqno for an ACK only packet
  - What happens if the advertised window = 0
  - What if the advertised window is 1/2 an MSS
  - Should receiver acknowledge packets right away
  - What if the application generates data in units of 0.1 MSS
  - What happens if I get a duplicate SYN? Or an RST while I'm in FIN_WAIT?
  - *etc., etc., etc.*

Taking Stock

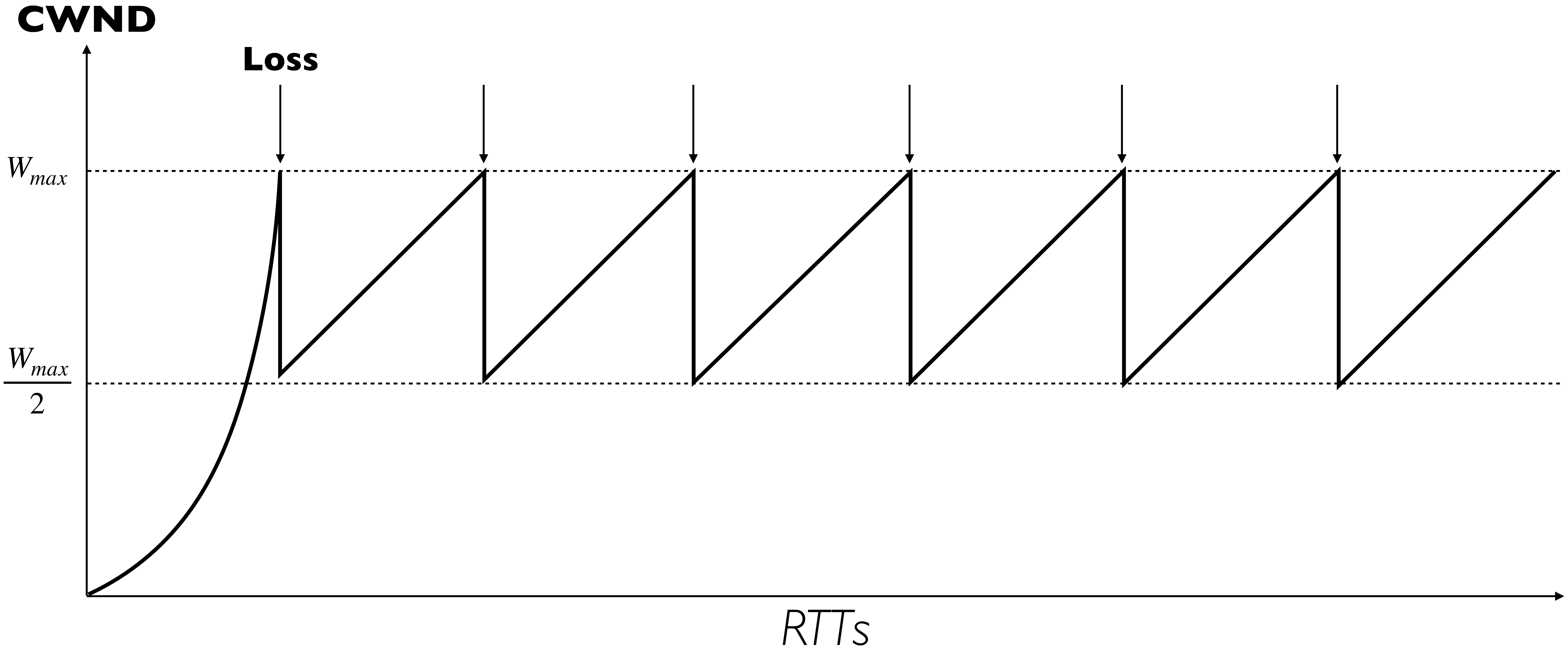
- The concepts underlying TCP are simple
- But tricky in the details
- Do the details matter?

Rest of Today's Lecture

- Critically examining TCP
- Advanced techniques

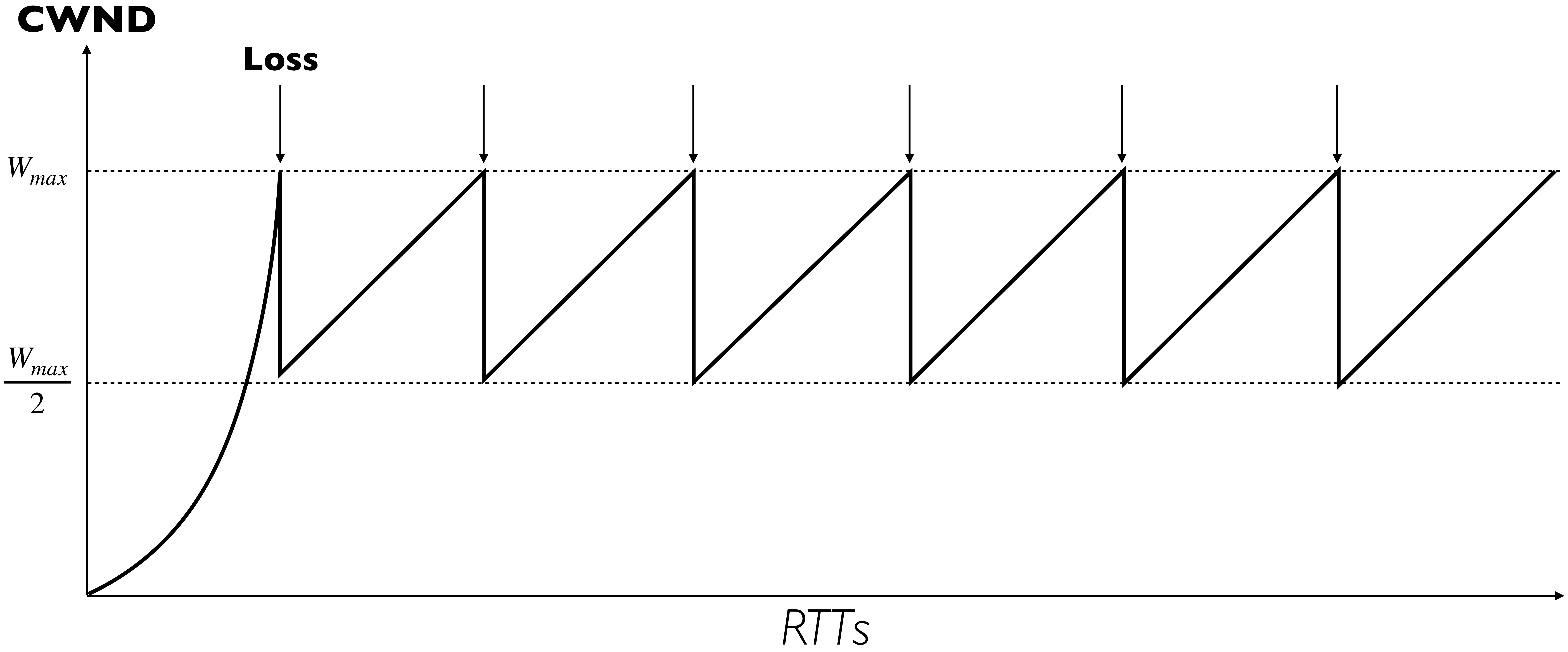
TCP Throughput Equation

A Simple Model for TCP Throughput



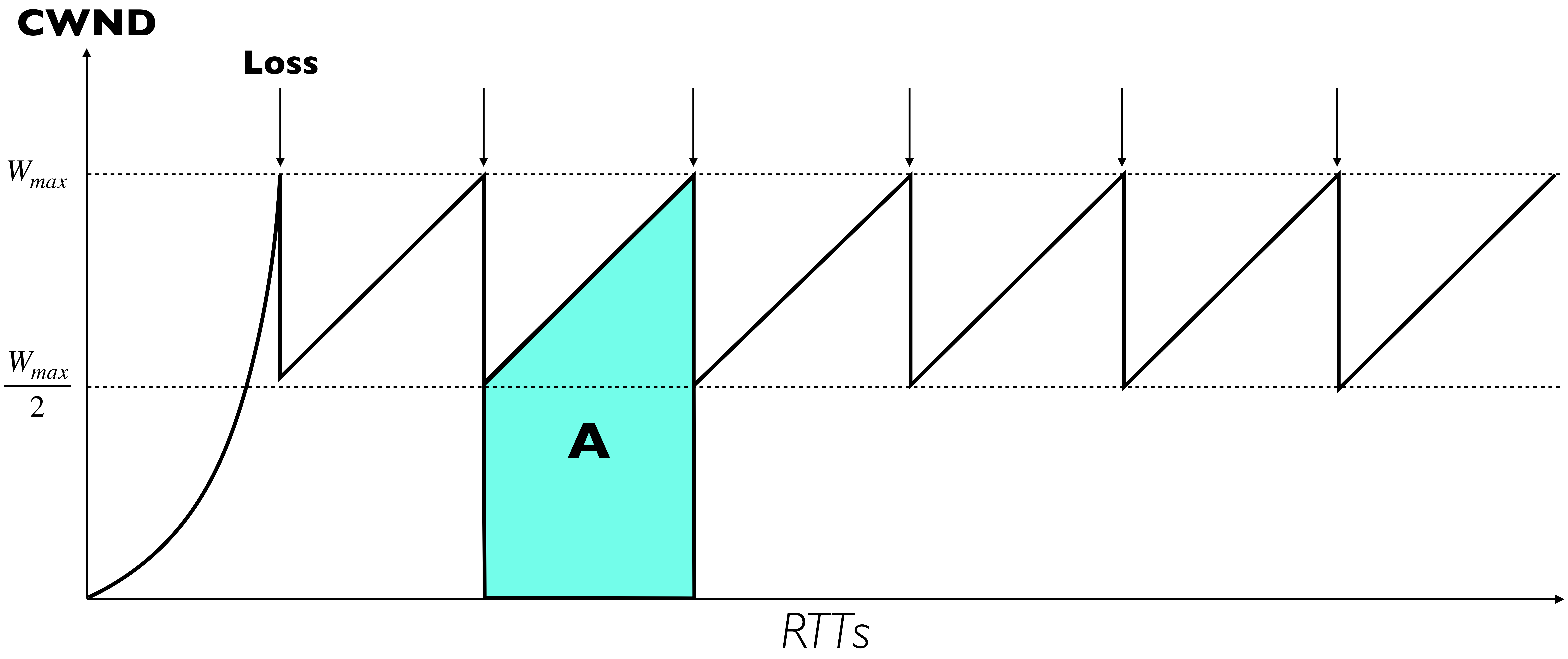
A Simple Model for TCP Throughput

Throughput = Window Size / RTT



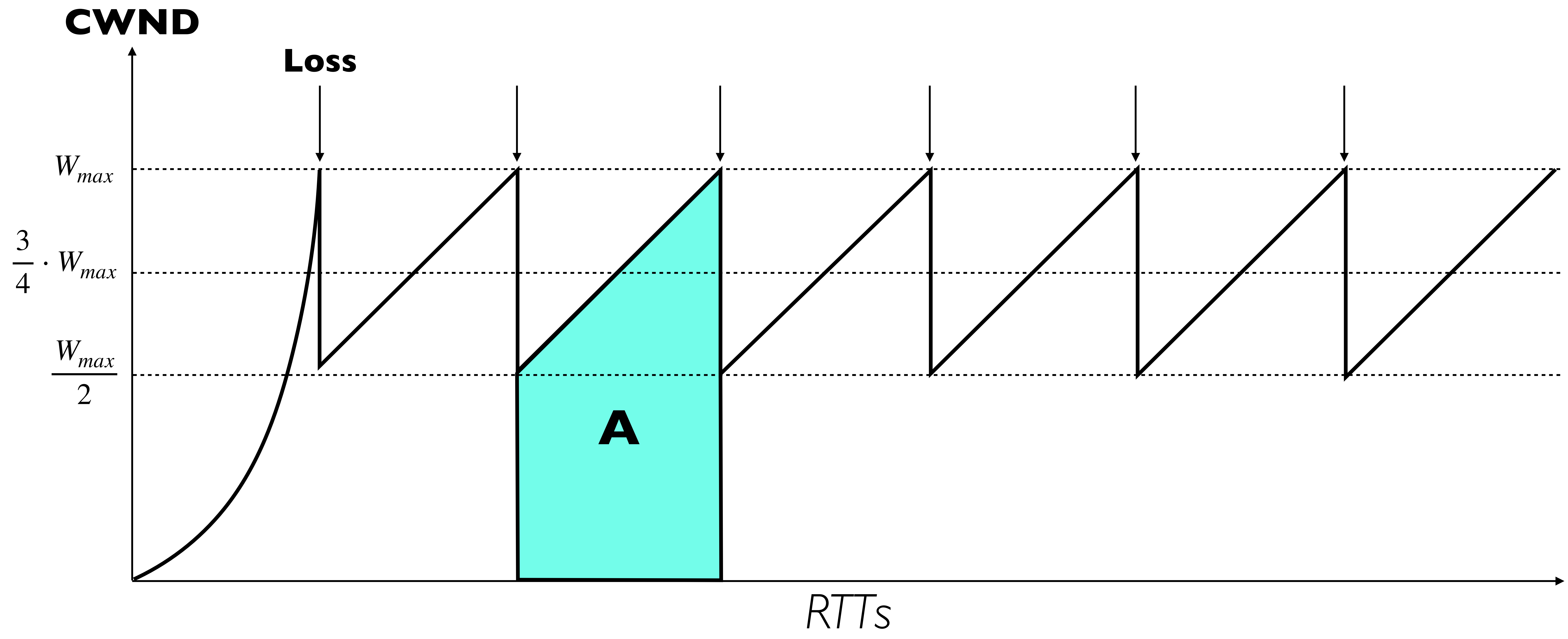
A Simple Model for TCP Throughput

Throughput = Window Size / RTT



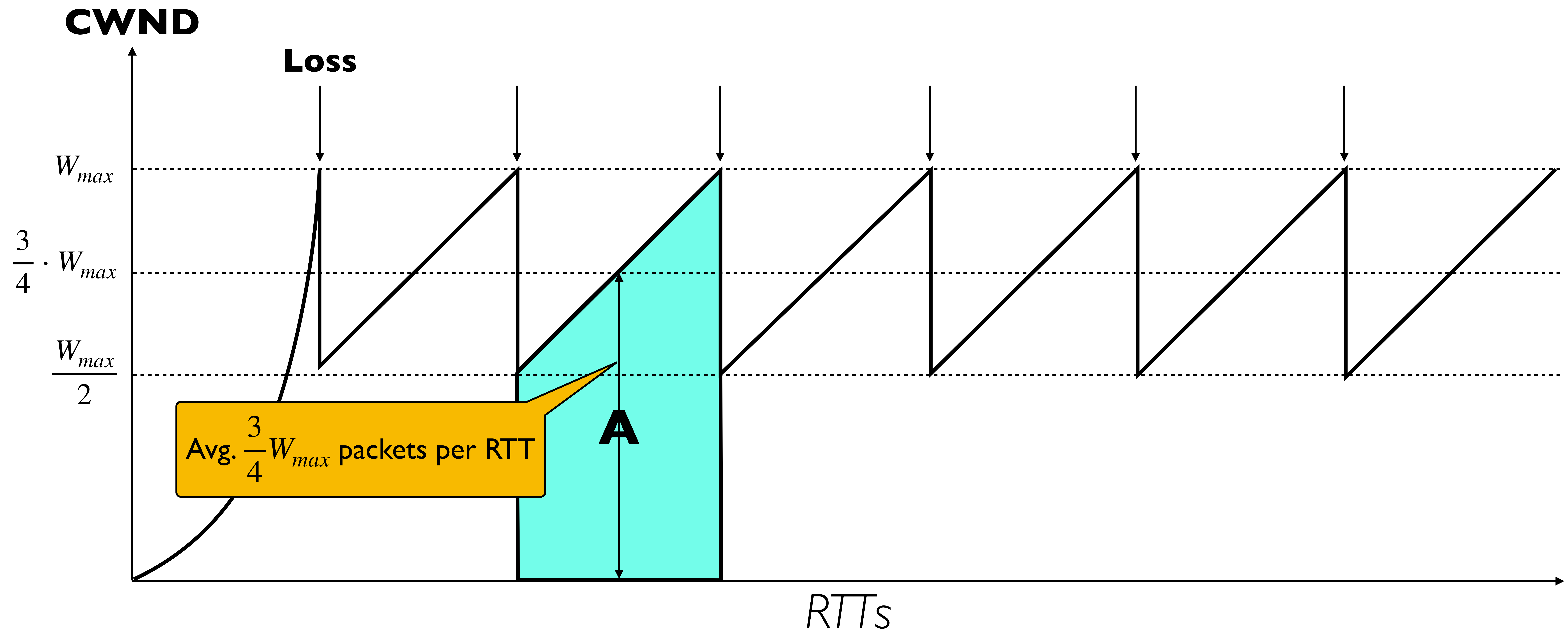
A Simple Model for TCP Throughput

Throughput = Window Size / RTT



A Simple Model for TCP Throughput

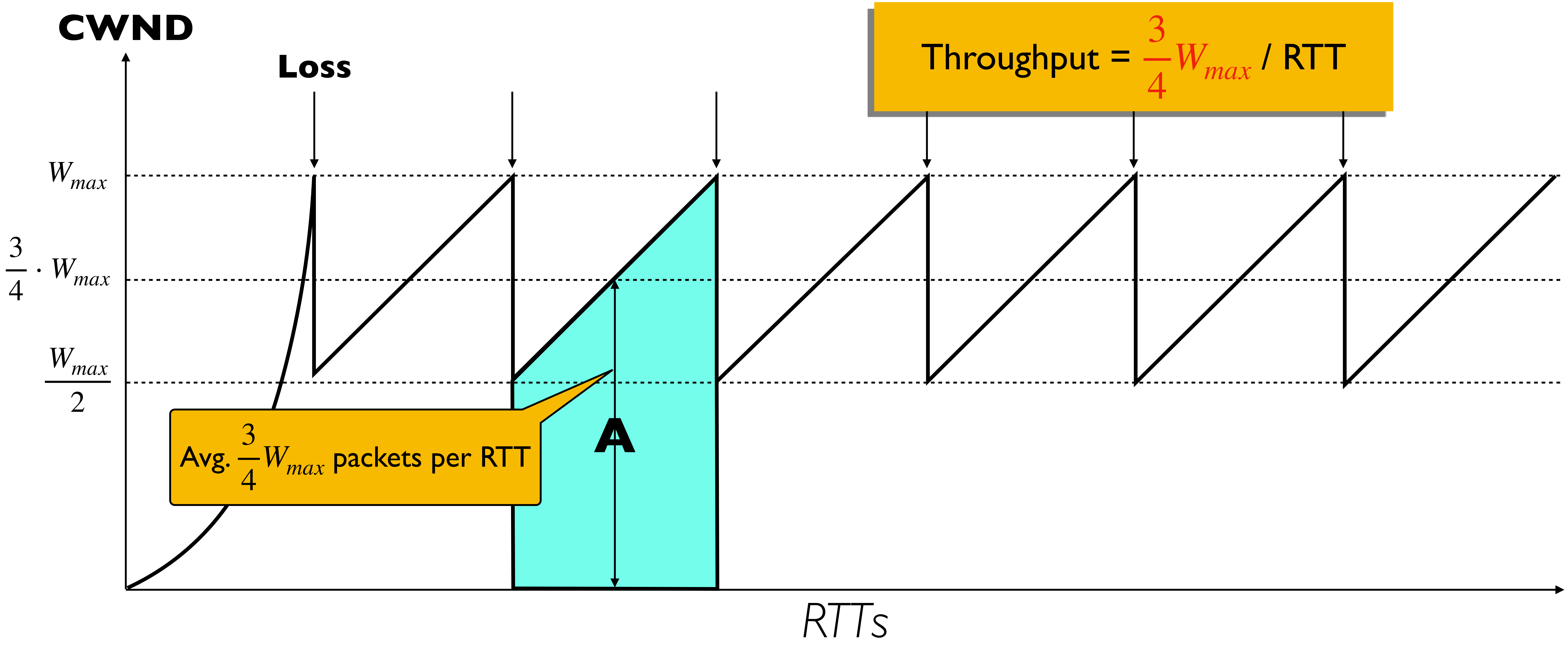
Throughput = Window Size / RTT



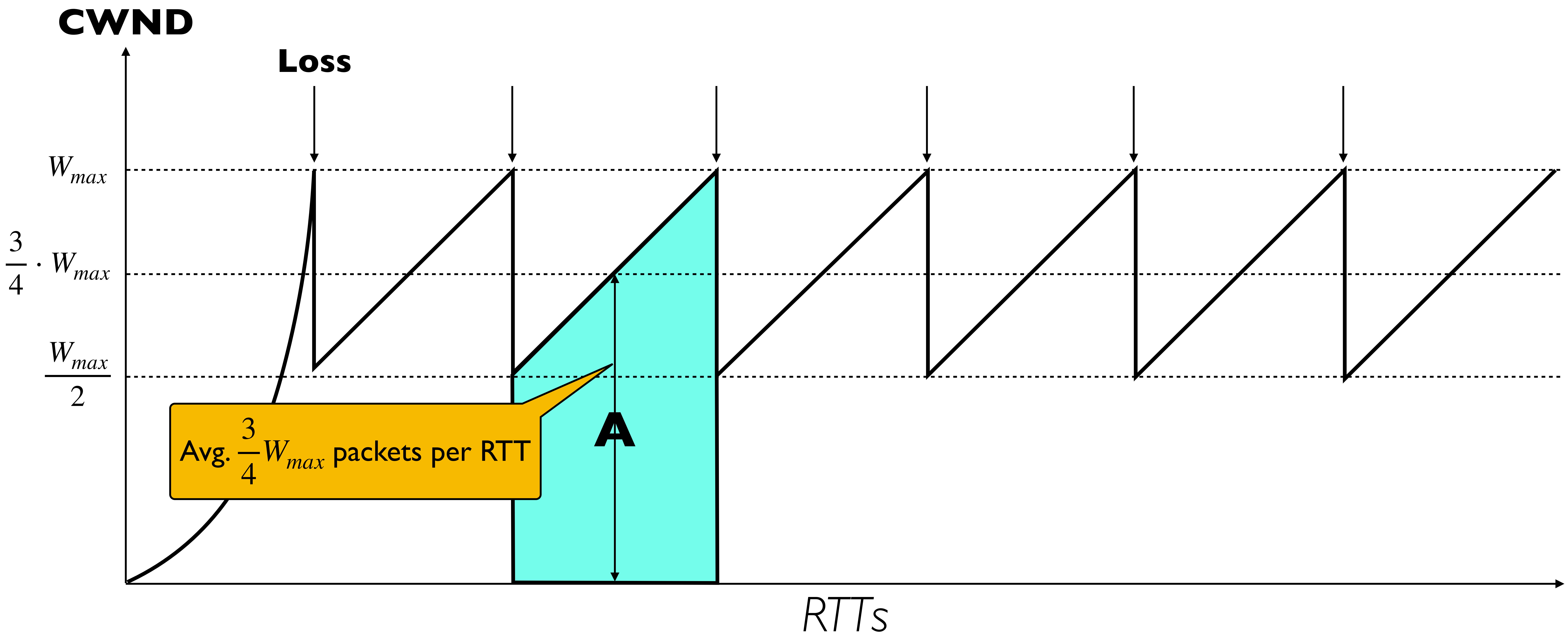
A Simple Model for TCP Throughput

Throughput = Window Size / RTT

Throughput = $\frac{3}{4} W_{max}$ / RTT

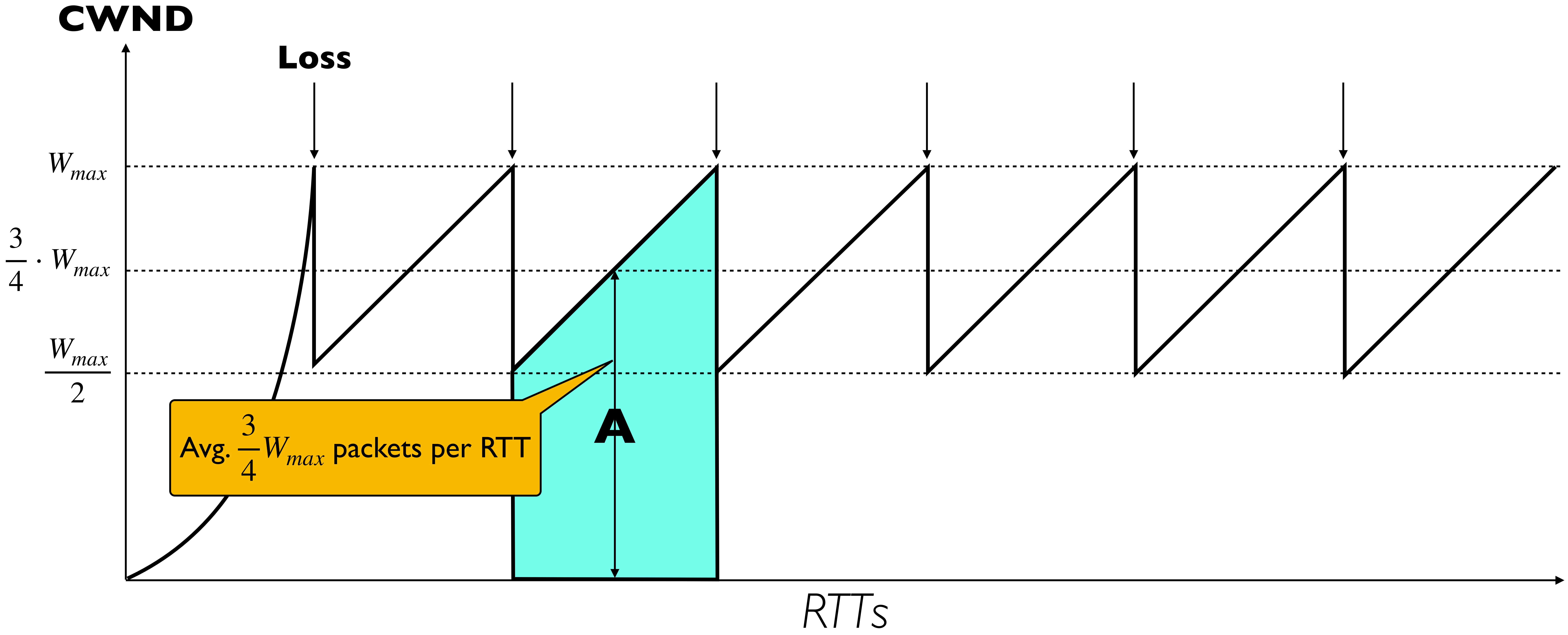


A Simple Model for TCP Throughput



A Simple Model for TCP Throughput

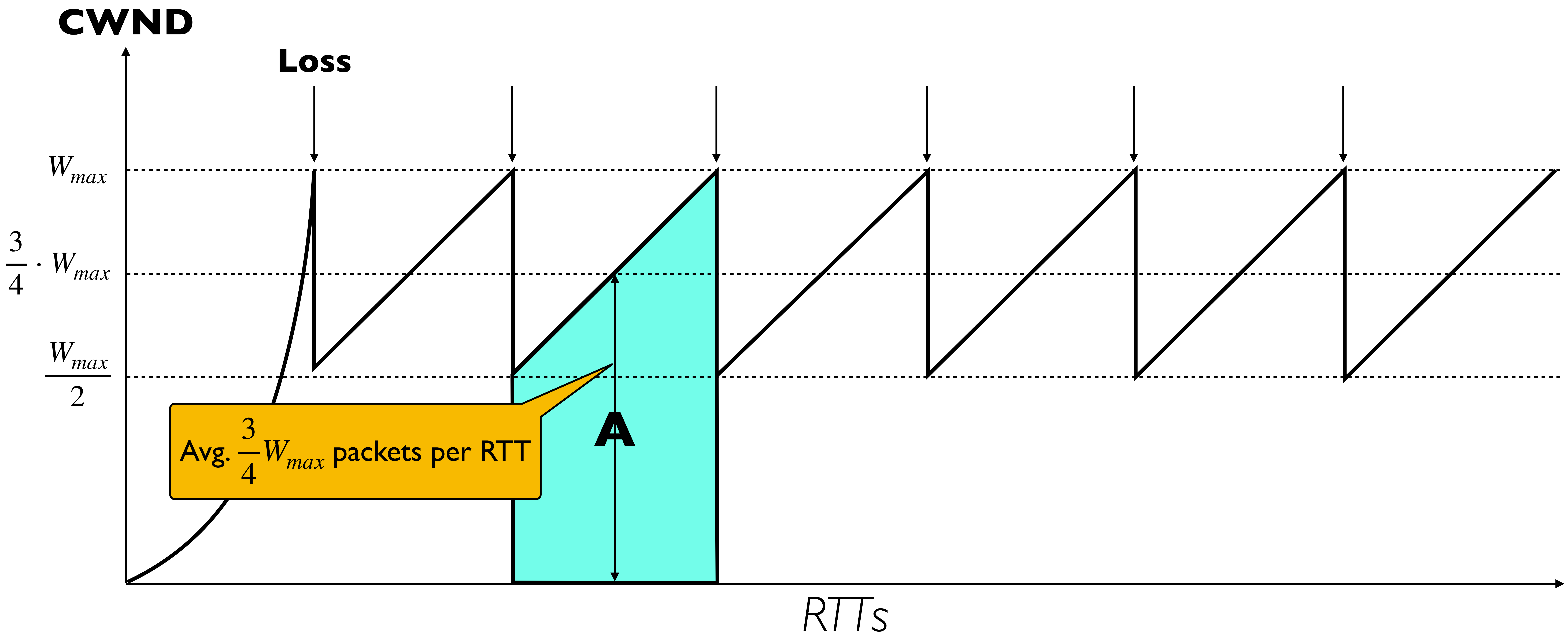
How many packets sent in the region with area A?



A Simple Model for TCP Throughput

How many packets sent in the region with area A?

A

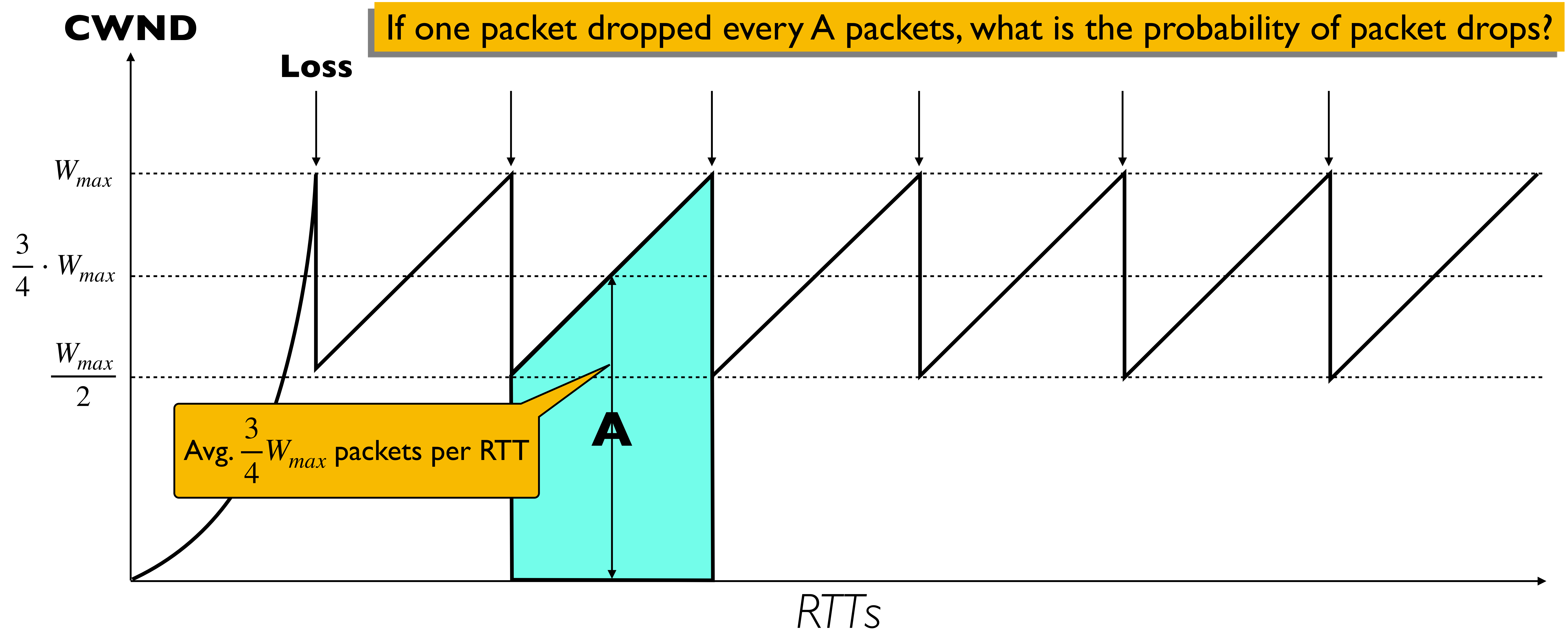


A Simple Model for TCP Throughput

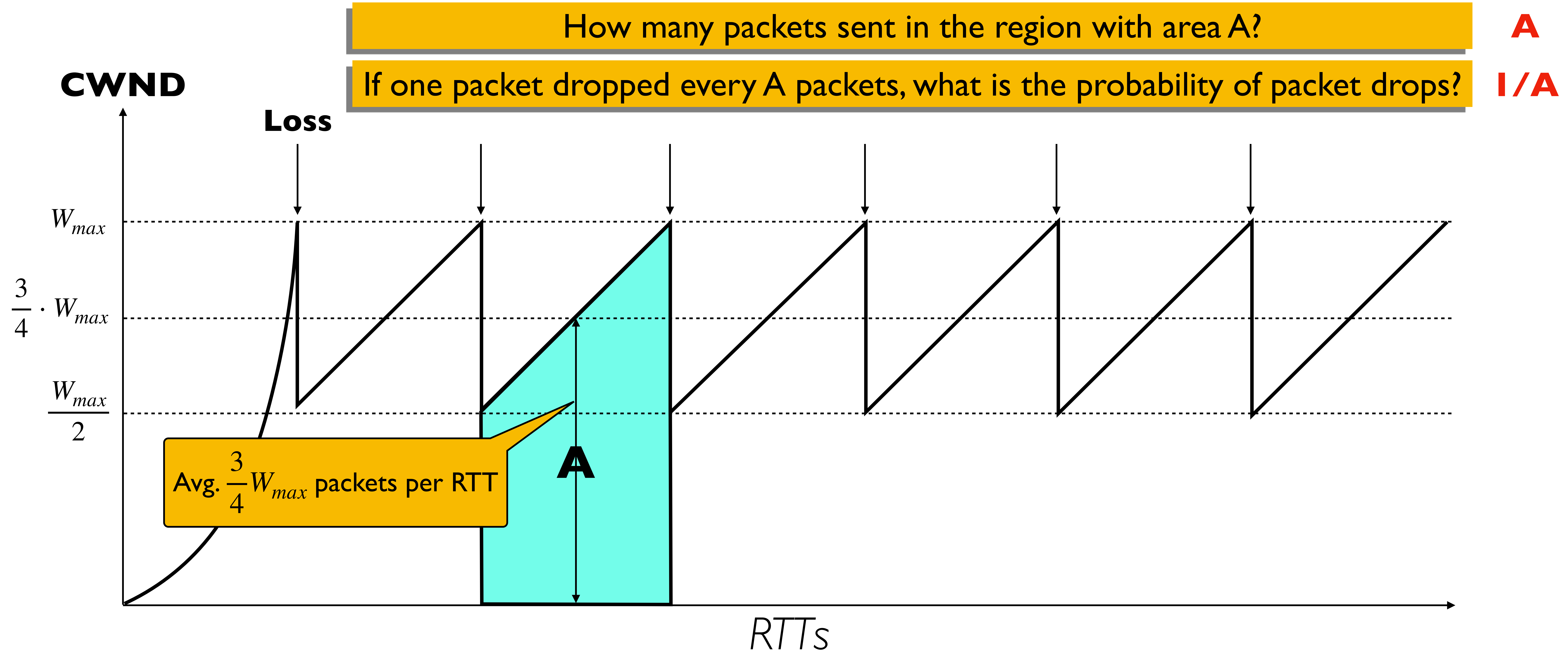
How many packets sent in the region with area A?

If one packet dropped every A packets, what is the probability of packet drops?

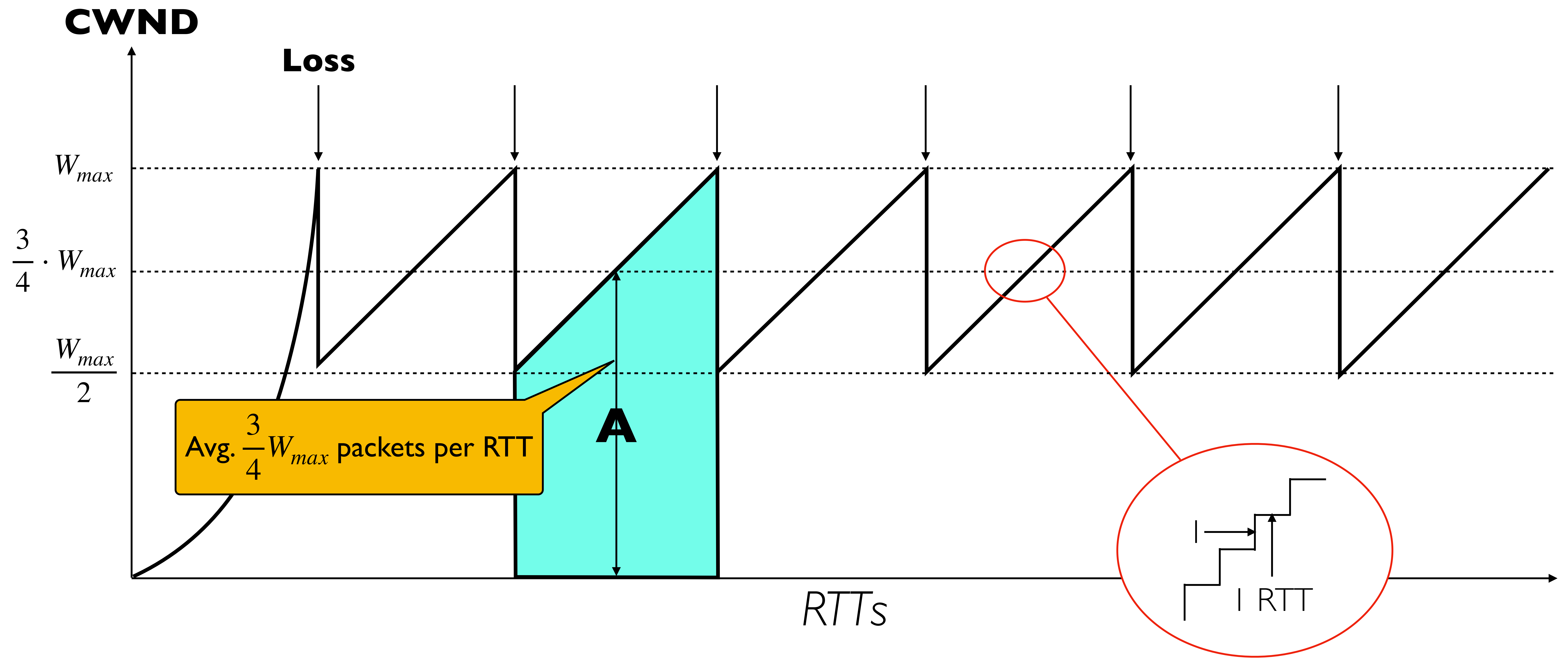
A



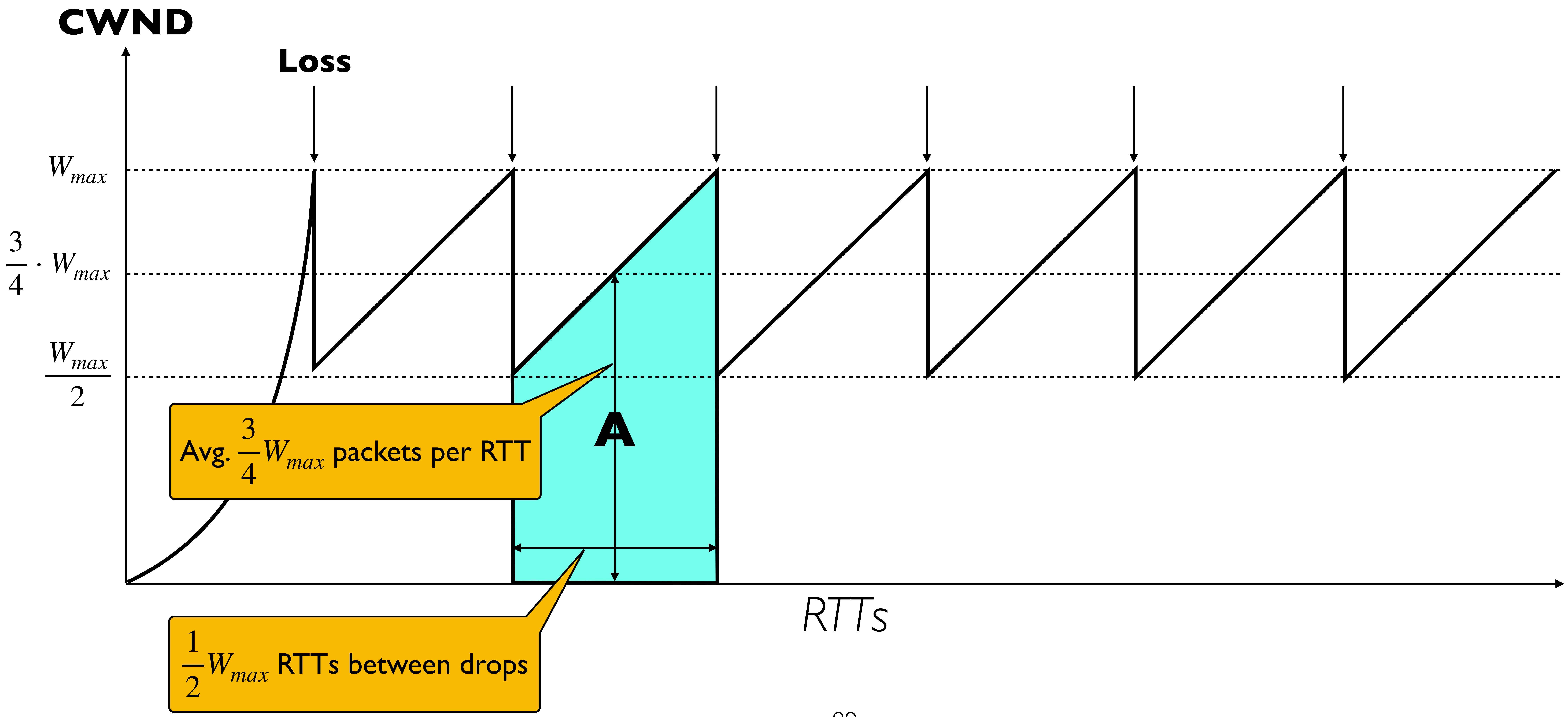
A Simple Model for TCP Throughput



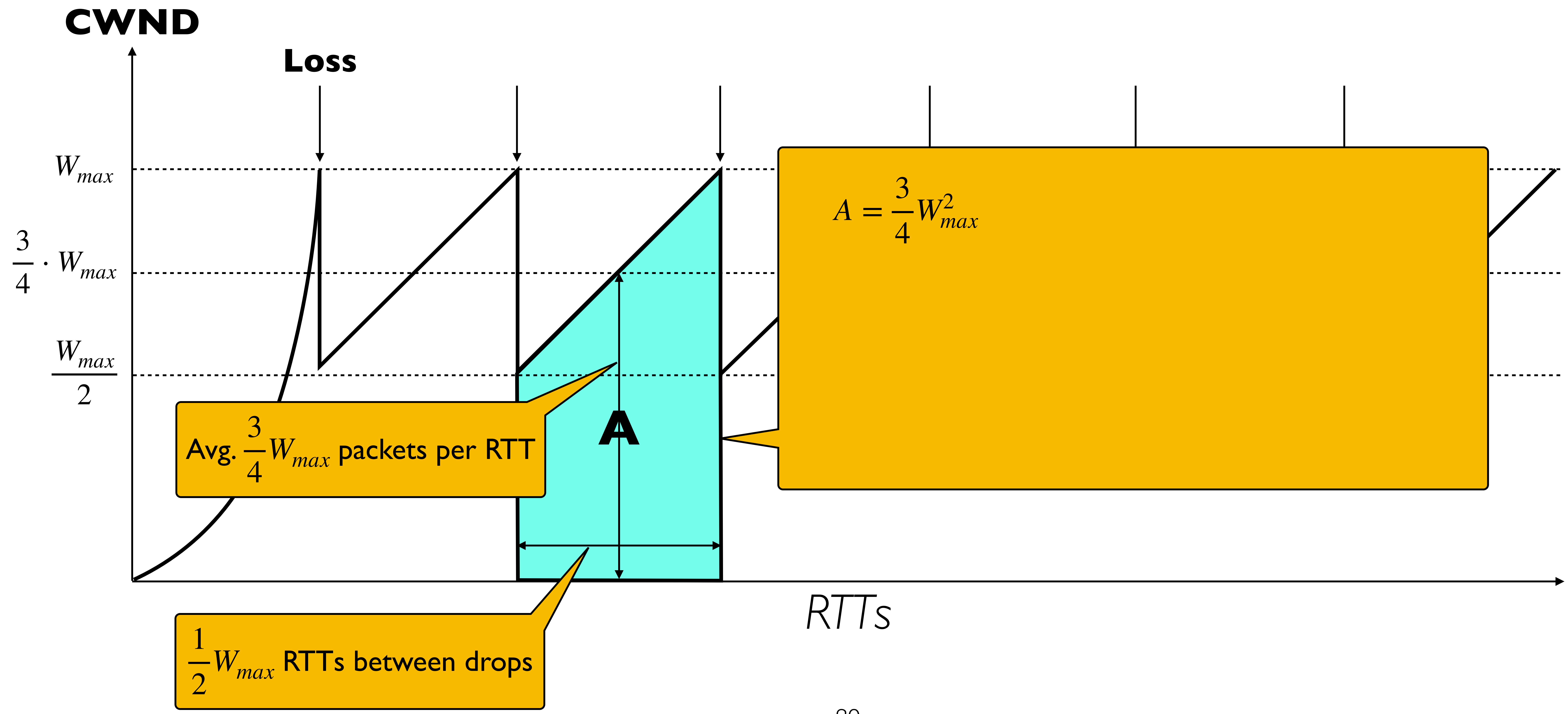
A Simple Model for TCP Throughput



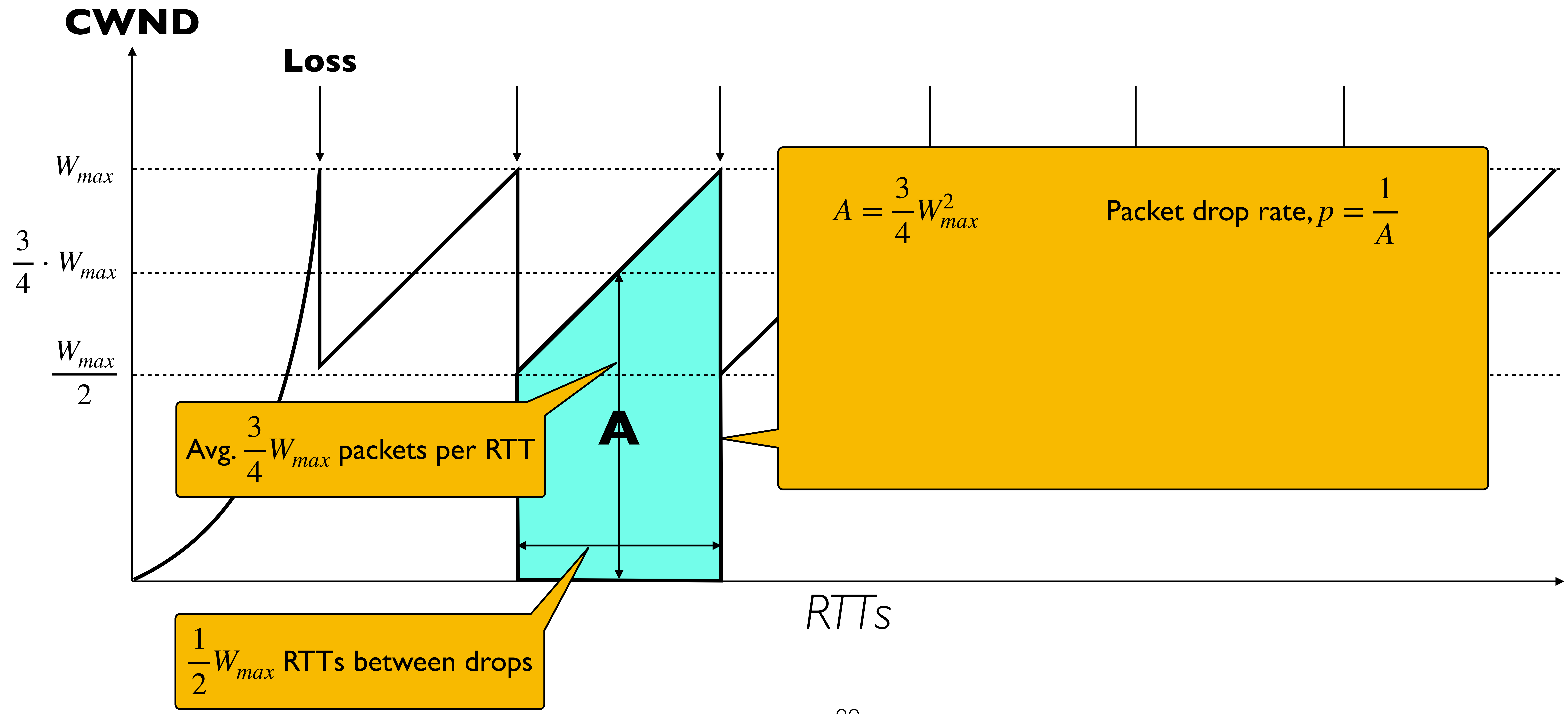
A Simple Model for TCP Throughput



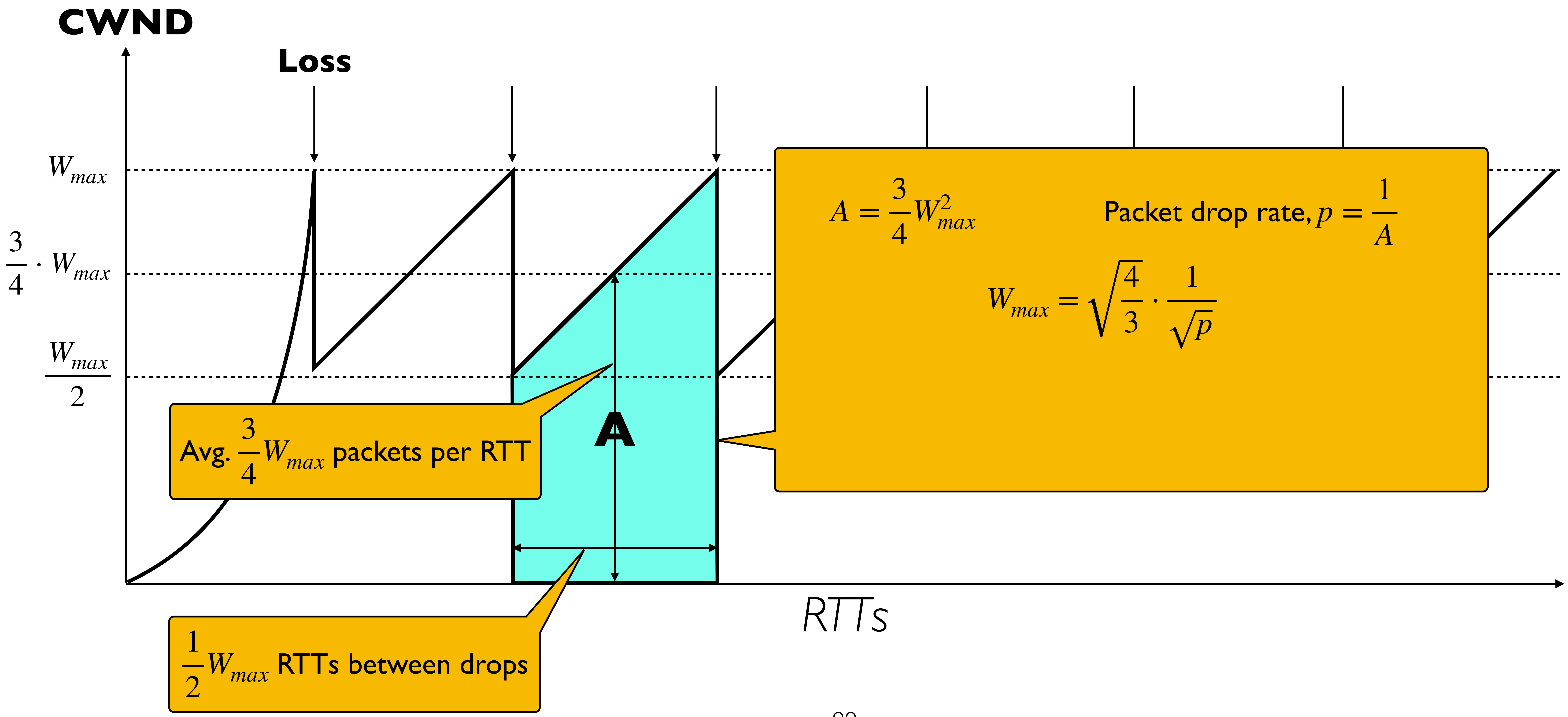
A Simple Model for TCP Throughput



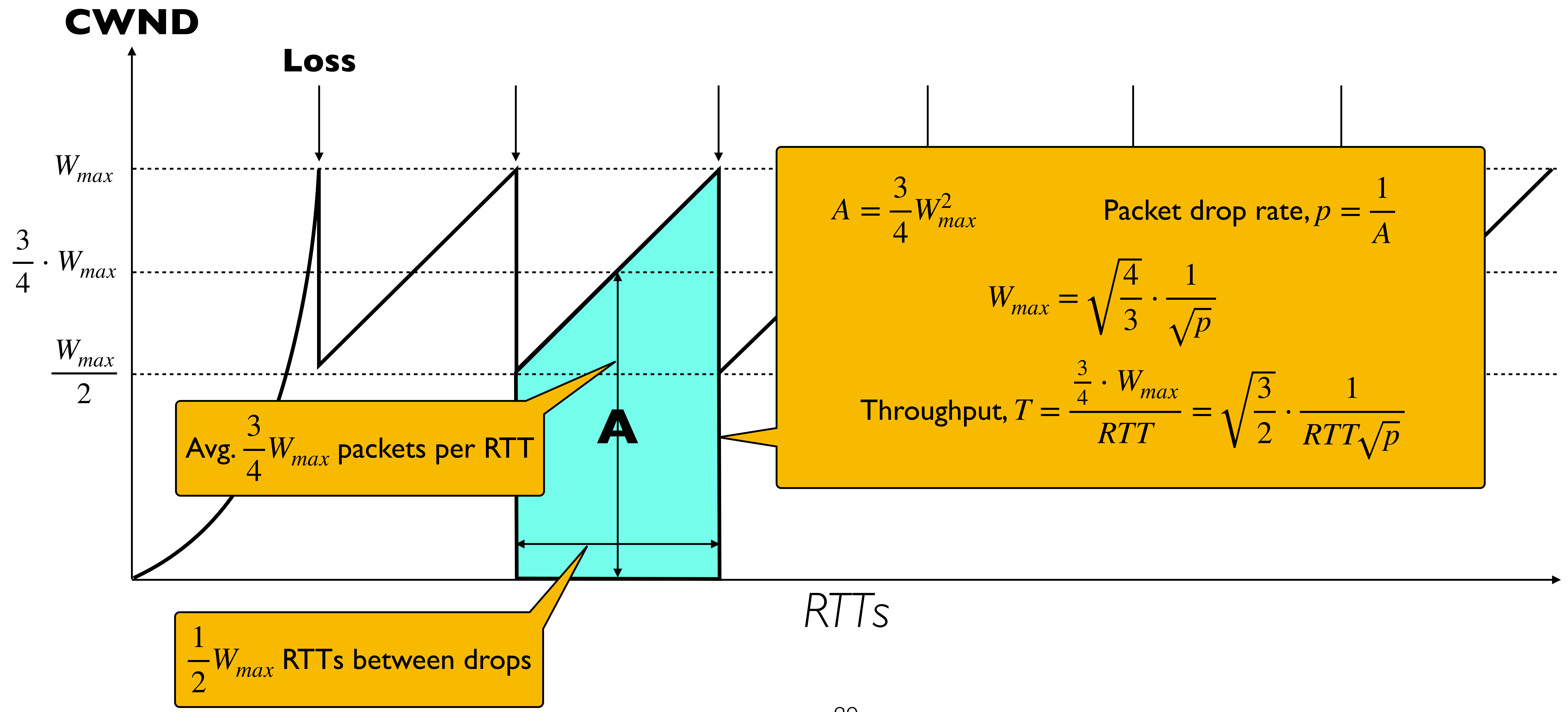
A Simple Model for TCP Throughput



A Simple Model for TCP Throughput

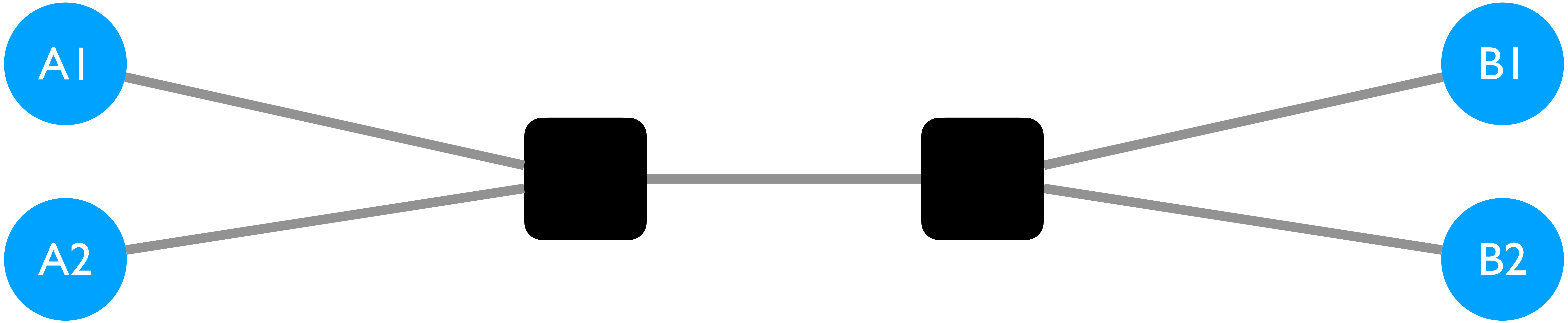


A Simple Model for TCP Throughput



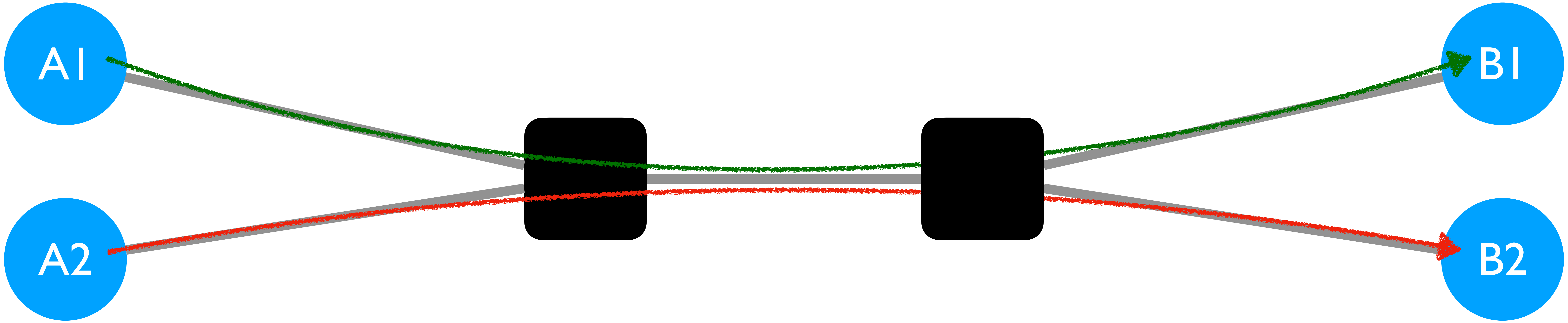
Implications (1): Different RTTs

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



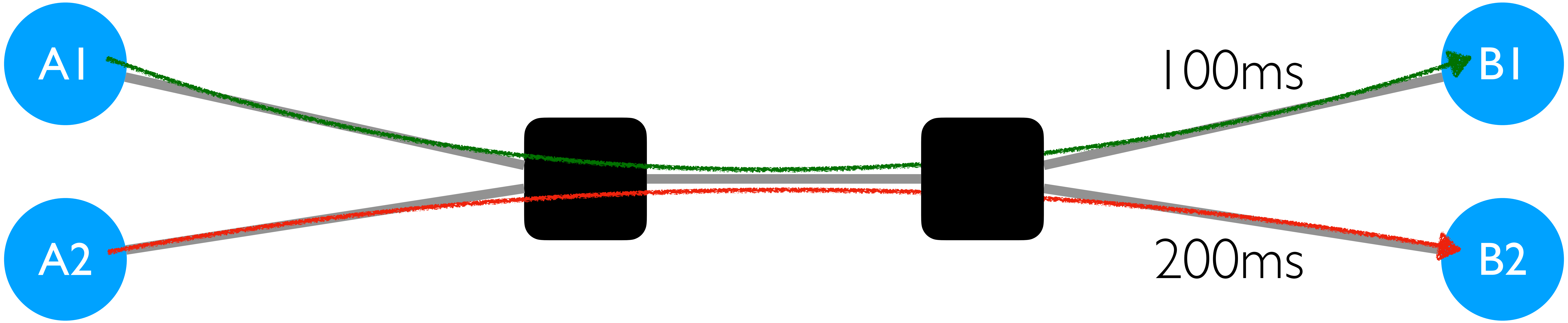
Implications (1): Different RTTs

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



Implications (1): Different RTTs

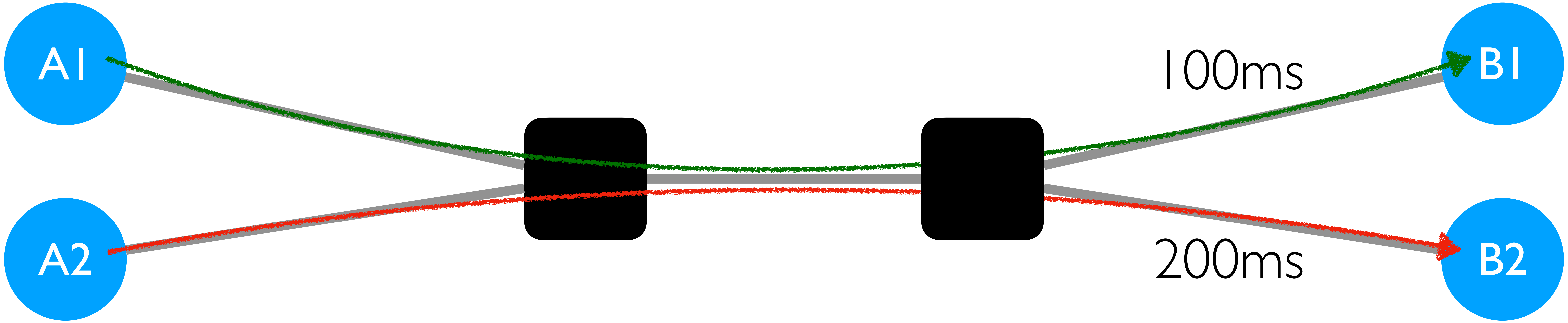
$$\text{Throughput} = \sqrt{\frac{3}{2}} \cdot \frac{1}{RTT\sqrt{p}}$$



Implications (1): Different RTTs

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

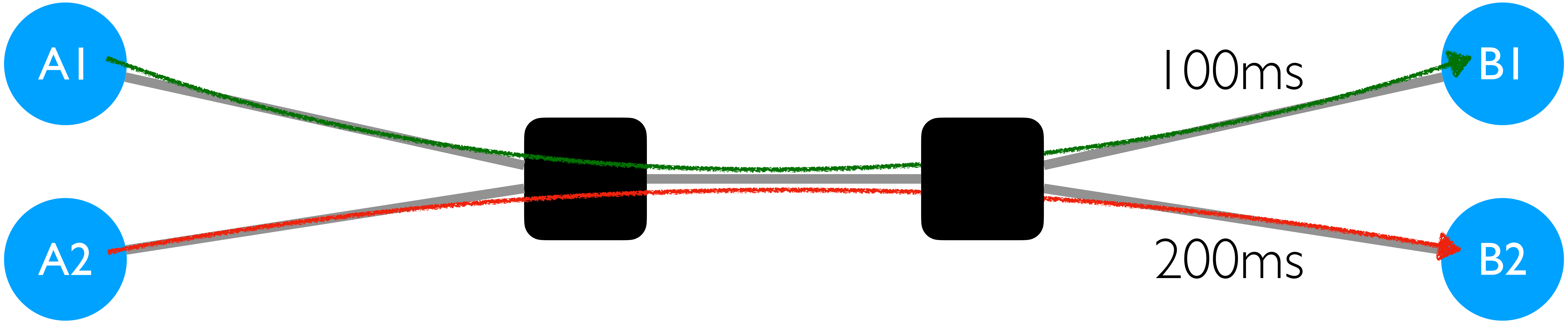
- Flows get throughput inversely proportional to RTT



Implications (1): Different RTTs

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

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



Implications (2): High Speed TCP

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

Implications (2): High Speed TCP

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

- Assume RTT = 100ms, MSS = 1500 bytes, Bandwidth = 100 Gbps

Implications (2): High Speed TCP

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

- Assume RTT = 100ms, MSS = 1500 bytes, Bandwidth = 100 Gbps
- What value of p is required to reach 100 Gbps throughput?

Implications (2): High Speed TCP

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

- Assume RTT = 100ms, MSS = 1500 bytes, Bandwidth = 100 Gbps
- What value of p is required to reach 100 Gbps throughput?
    - $\sim 2 \times 10^{-12}$

Implications (2): High Speed TCP

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

- Assume RTT = 100ms, MSS = 1500 bytes, Bandwidth = 100 Gbps
- What value of p is required to reach 100 Gbps throughput?
  - $\sim 2 \times 10^{-12}$
- How long between drops?

Implications (2): High Speed TCP

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

- Assume RTT = 100ms, MSS = 1500 bytes, Bandwidth = 100 Gbps
- What value of p is required to reach 100 Gbps throughput?
  - ~2×10^{-12}
- How long between drops?
  - ~16.6 hours

Implications (2): High Speed TCP

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

- Assume RTT = 100ms, MSS = 1500 bytes, Bandwidth = 100 Gbps
- What value of p is required to reach 100 Gbps throughput?
  - ~2×10^{-12}
- How long between drops?
  - ~16.6 hours
- How much data has been sent in this time?

Implications (2): High Speed TCP

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

- Assume RTT = 100ms, MSS = 1500 bytes, Bandwidth = 100 Gbps
- What value of p is required to reach 100 Gbps throughput?
  - $\sim 2 \times 10^{-12}$
- How long between drops?
  - ~16.6 hours
- How much data has been sent in this time?
  - ~6 petabits

Implications (2): High Speed TCP

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

- Assume RTT = 100ms, MSS = 1500 bytes, Bandwidth = 100 Gbps
- What value of p is required to reach 100 Gbps throughput?
  - $\sim 2 \times 10^{-12}$
- How long between drops?
  - ~16.6 hours
- How much data has been sent in this time?
  - ~6 petabits
- These are not practical numbers!

Adapting TCP to High Speed

Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster

Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster
- Other approaches?

Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster
- Other approaches?
  - Multiple simultaneously connections (hack but works today)

Adapting TCP to High Speed

- **Once past a threshold speed, increase CWND faster**
- **Other approaches?**
  - Multiple simultaneously connections (hack but works today)
  - Router-assisted approaches (will see shortly)

Implications (3): "Sawtooth" behavior

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

Implications (3): "Sawtooth" behavior

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

- TCP throughput is "choppy"

Implications (3): "Sawtooth" behavior

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

- TCP throughput is "choppy"
  - Repeated swings between W/2 and W

Implications (3): "Sawtooth" behavior

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

- TCP throughput is "choppy"
  - Repeated swings between W/2 and W
- Some applications would prefer sending at a steady rate

Implications (3): "Sawtooth" behavior

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

- TCP throughput is "choppy"
  - Repeated swings between W/2 and W
- Some applications would prefer sending at a steady rate
  - e.g., streaming applications

Implications (3): "Sawtooth" behavior

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

- TCP throughput is "choppy"
  - Repeated swings between W/2 and W
- Some applications would prefer sending at a steady rate
  - e.g., streaming applications
- A solution: "Equation-based congestion control"

Implications (3): “Sawtooth” behavior

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

- TCP throughput is “choppy”
  - Repeated swings between W/2 and W
- Some applications would prefer sending at a steady rate
  - e.g., streaming applications
- A solution: “Equation-based congestion control”
  - Ditch TCP’s increase/decrease rules and just follow the equation

Implications (3): "Sawtooth" behavior

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

- TCP throughput is "choppy"
  - Repeated swings between W/2 and W
- Some applications would prefer sending at a steady rate
  - e.g., streaming applications
- A solution: "Equation-based congestion control"
  - Ditch TCP's increase/decrease rules and just follow the equation
  - Measure drop percentage p, and set rate accordingly

Implications (3): "Sawtooth" behavior

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

- TCP throughput is "choppy"
  - Repeated swings between W/2 and W
- Some applications would prefer sending at a steady rate
  - e.g., streaming applications
- A solution: "Equation-based congestion control"
  - Ditch TCP's increase/decrease rules and just follow the equation
  - Measure drop percentage p, and set rate accordingly
- Following the TCP equation ensures we're "TCP friendly"

Implications (3): "Sawtooth" behavior

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

- **TCP throughput is "choppy"**
  - Repeated swings between W/2 and W
- **Some applications would prefer sending at a steady rate**
  - e.g., streaming applications
- **A solution: "Equation-based congestion control"**
  - Ditch TCP's increase/decrease rules and just follow the equation
  - 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

Other Limitations of TCP Congestion Control

Implications (4): Losses not due to congestion?

Implications (4): Losses not due to congestion?

- TCP will confuse corruption with congestion

Implications (4): Losses not due to congestion?

- TCP will confuse corruption with congestion
- Flow will cut its rate

Implications (4): Losses not due to congestion?

- TCP will confuse corruption with congestion
- Flow will cut its rate
  - Throughput ~1/sqrt(p) where p is loss probability

Implications (4): Losses not due to congestion?

- **TCP will confuse corruption with congestion**

- **Flow will cut its rate**
  - Throughput ~1/sqrt(p) where p is loss probability
  - Applies even for non-congestion losses!

Implications (4): Losses not due to congestion?

- TCP will confuse corruption with congestion
- Flow will cut its rate
  - Throughput ~1/sqrt(p) where p is loss probability
  - Applies even for non-congestion losses!
- We'll look at proposed solutions shortly…

Implications (5): How do short flows fare?

Implications (5): How do short flows fare?

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

Implications (5): How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (5.A): short flows never leave slow start!

Implications (5): How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (5.A): short flows never leave slow start!
  - Short flows never attain their fair share

Implications (5): How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (5.A): short flows never leave slow start!
  - Short flows never attain their fair share
- Implication (5.B): too few packets to trigger dupACKs & fast retransmit

Implications (5): How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (5.A): short flows never leave slow start!
  - Short flows never attain their fair share
- Implication (5.B): too few packets to trigger dupACKs & fast retransmit
  - Isolated loss may lead to timeouts

Implications (5): How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (5.A): short flows never leave slow start!
  - Short flows never attain their fair share
- Implication (5.B): too few packets to trigger dupACKs & fast retransmit
  - Isolated loss may lead to timeouts
  - At typical timeout values of ~500ms, might severely impact flow completion time

Implications (5): How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (5.A): short flows never leave slow start!
  - Short flows never attain their fair share
- Implication (5.B): too few packets to trigger dupACKs & fast retransmit
  - Isolated loss may lead to timeouts
  - At typical timeout values of ~500ms, might severely impact flow completion time
- Again, we will look at solutions in a bit…

Implications (6): TCP fills up queues → long delays

Implications (6): TCP fills up queues → long delays

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

Implications (6): TCP fills up queues → long delays

- **A flow deliberately overshoots capacity, until it experiences a drop**
  - TCP only slows down when queues fill up!

Implications (6): TCP fills up queues → long delays

- **A flow deliberately overshoots capacity, until it experiences a drop**
  - TCP only slows down when queues fill up!
  - *Everyone* experiences delays (esp. short flows)

Implications (6): TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop
    - TCP only slows down when queues fill up!
    - *Everyone* experiences delays (esp. short flows)
- Clearly, it is not optimized for latency

Implications (6): TCP fills up queues → long delays

- **A flow deliberately overshoots capacity, until it experiences a drop**
  - TCP only slows down when queues fill up!
  - *Everyone* experiences delays (esp. short flows)

- **Clearly, it is not optimized for latency**
  - What is it optimized for?

Implications (6): TCP fills up queues → long delays

- **A flow deliberately overshoots capacity, until it experiences a drop**
  - TCP only slows down when queues fill up!
  - *Everyone* experiences delays (esp. short flows)

- **Clearly, it is not optimized for latency**
  - What is it optimized for?
    - Fairness in throughput allocation!

Implications (6): TCP fills up queues → long delays

- **A flow deliberately overshoots capacity, until it experiences a drop**
  - TCP only slows down when queues fill up!
  - *Everyone* experiences delays (esp. short flows)

- **Clearly, it is not optimized for latency**
  - What is it optimized for?
    - Fairness in throughput allocation!

- **Many packets dropped when buffers are full**

Implications (6): TCP fills up queues → long delays

- **A flow deliberately overshoots capacity, until it experiences a drop**
  - TCP only slows down when queues fill up!
  - *Everyone* experiences delays (esp. short flows)

- **Clearly, it is not optimized for latency**
  - What is it optimized for?
    - Fairness in throughput allocation!

- **Many packets dropped when buffers are full**

- **Solutions?**

Implications (6): TCP fills up queues → long delays

- **A flow deliberately overshoots capacity, until it experiences a drop**
  - TCP only slows down when queues fill up!
  - *Everyone* experiences delays (esp. short flows)

- **Clearly, it is not optimized for latency**
  - What is it optimized for?
    - Fairness in throughput allocation!

- **Many packets dropped when buffers are full**

- **Solutions?**
  - Soon…

Implications (7): Cheating

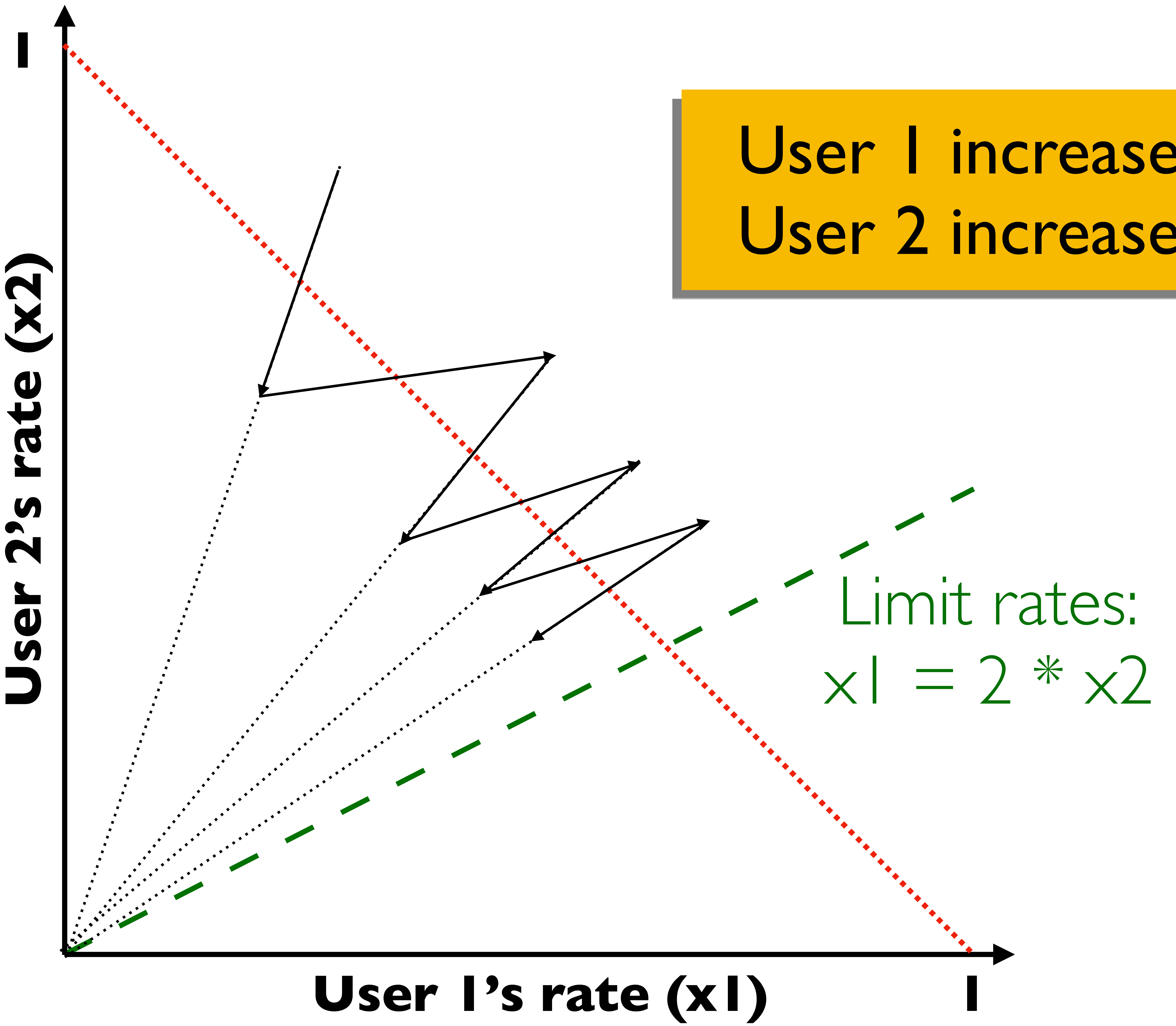
Implications (7): Cheating

- Three easy ways to cheat

Implications (7): Cheating

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

Increasing CWND Faster

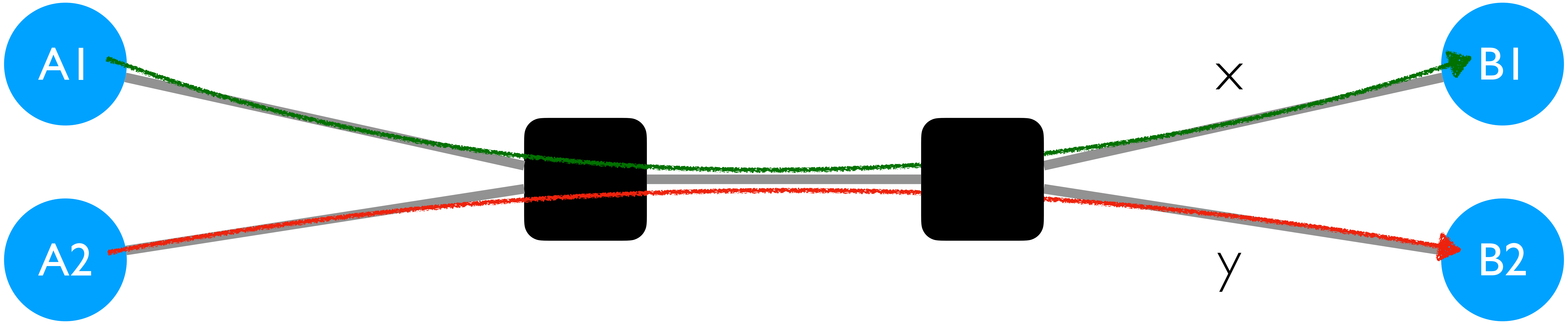


User 1 increases by 2 per RTT
User 2 increases by 1 per RTT

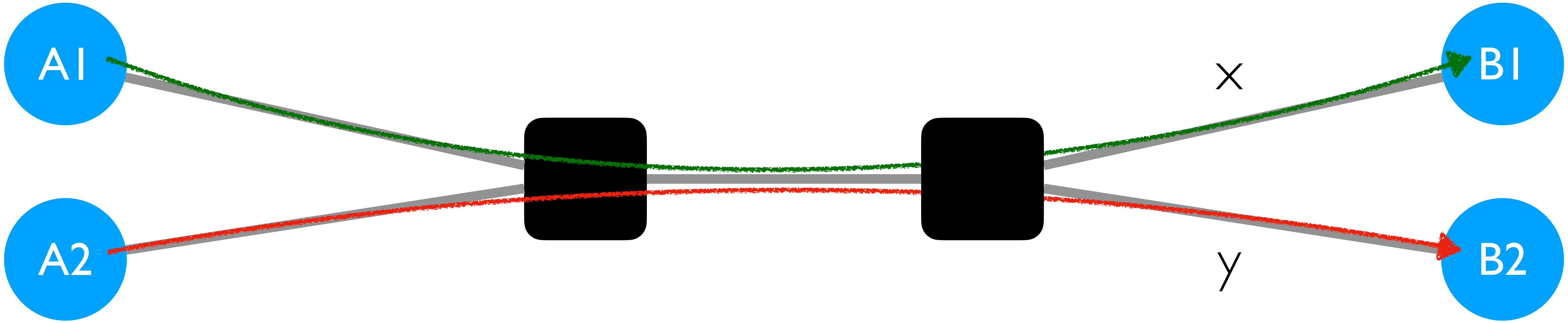
Implications (7): Cheating

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

Implications (1): Different RTTs

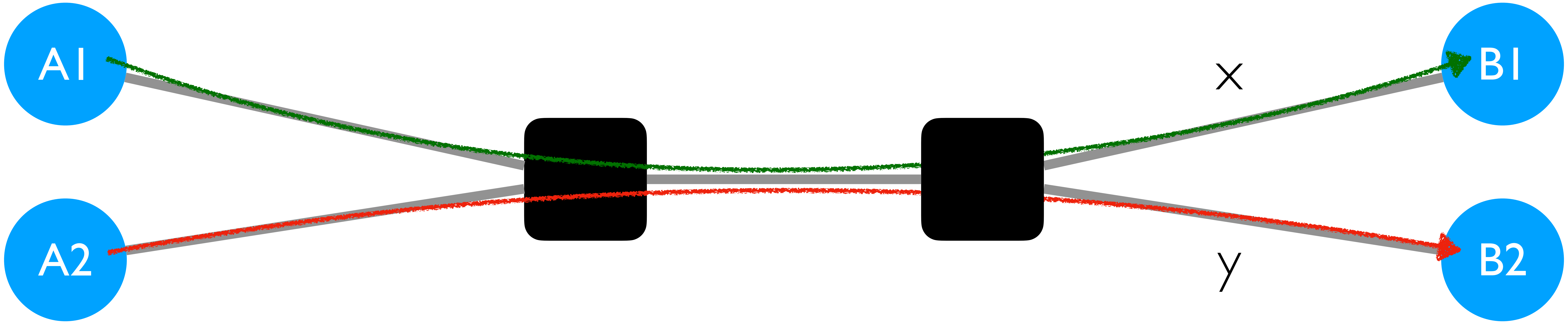


Implications (1): Different RTTs



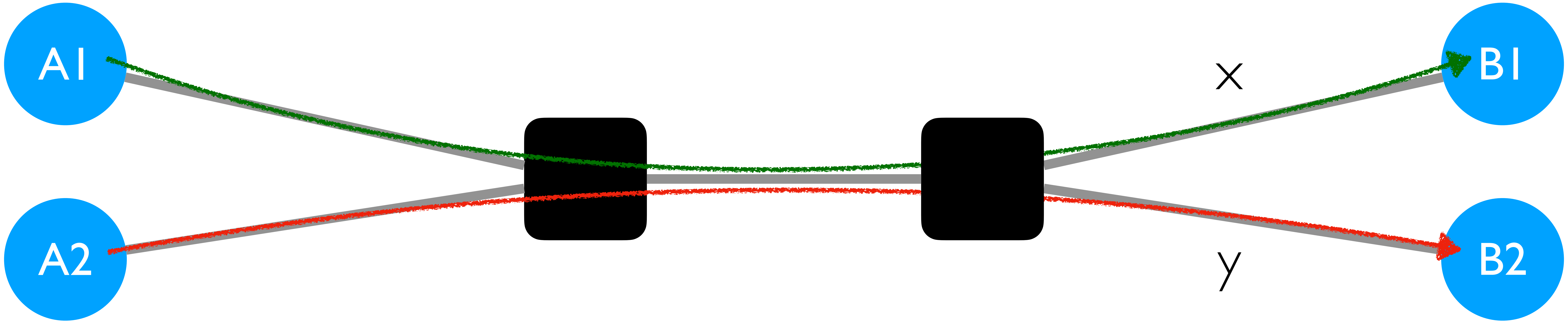
- Assume:

Implications (1): Different RTTs



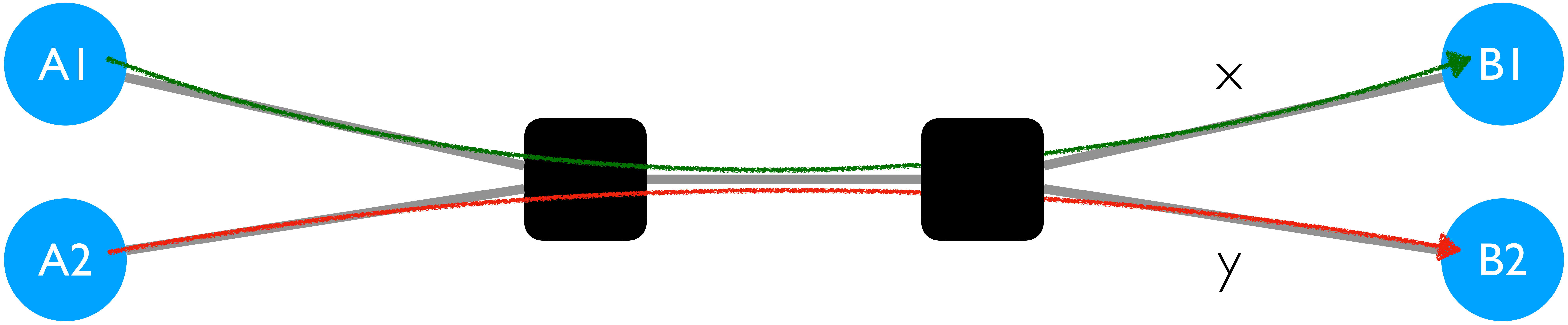
- Assume:
  - A1 starts 10 connections to B1

Implications (1): Different RTTs



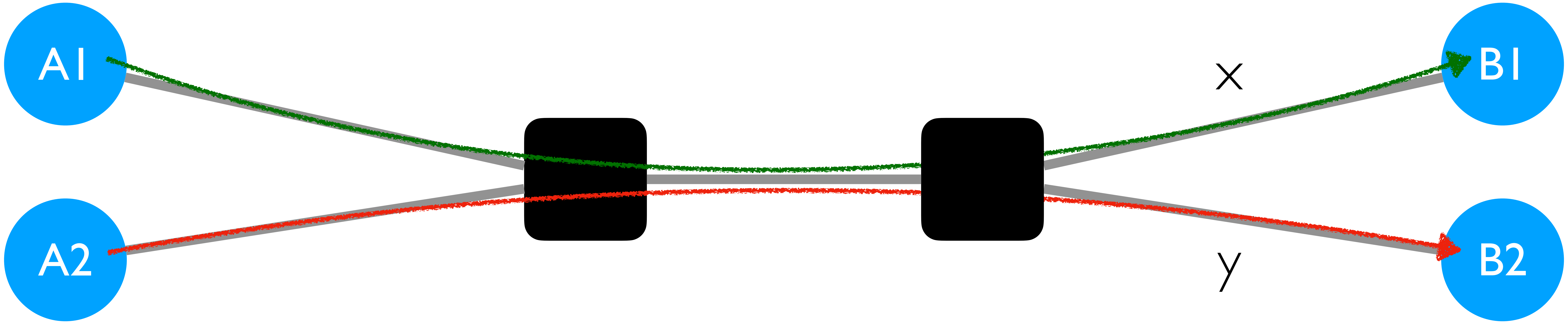
- Assume:
  - A1 starts 10 connections to B1
  - A2 starts 1 connection to B2

Implications (1): Different RTTs



- **Assume:**
  - A1 starts 10 connections to B1
  - A2 starts 1 connection to B2
  - Each connection gets about the same throughput

Implications (1): Different RTTs



- Assume:
  - A1 starts 10 connections to B1
  - A2 starts 1 connection to B2
  - Each connection gets about the same throughput
- Then A1 gets 10 times more throughput than A2!

Implications (7): Cheating

- **Three easy ways to cheat**
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections
  - ACK Splitting
    - Receiver sends multiple ACKs for same segment
    - E.g., receive segment containing bytes 1500-2400
    - Send 3 ACKs: 1801, 2101, 2401

Implications (7): Cheating

- **Three easy ways to cheat**
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections
  - ACK Splitting
    - Receiver sends multiple ACKs for same segment
    - E.g., receive segment containing bytes 1500-2400
    - Send 3 ACKs: 1801, 2101, 2401
- **Why hasn't the Internet suffered a congestion collapse yet?**

Implications (8): CC intertwined with reliability

Implications (8): 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

Implications (8): 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

Implications (8): 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., realtime applications

Implications (8): 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., realtime applications
- **Sometimes we want reliability but not CC (?)**

Recap: TCP Problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows compete before discovering available capacity
- AIMD impractical for high-speed links
- Sawtooth discovery too choppy for some applications
- Unfair under heterogenous RTTs
- Tight coupling with reliability mechanisms
- End-systems can cheat

Recap: TCP Problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows compete before discovering available capacity
- AIMD impractical for high-speed links
- Sawtooth discovery too choppy for some applications
- Unfair under heterogenous RTTs
- Tight coupling with reliability mechanisms
- End-systems can cheat

Could fix many of these with some help from routers!

Recap: TCP Problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows compete before discovering available capacity
- AIMD impractical for high-speed links
- Sawtooth discovery too choppy for some applications
- Unfair under heterogenous RTTs
- Tight coupling with reliability mechanisms
- End-systems can cheat

Routers tell end-systems if they are congested

Could fix many of these with some help from routers!

Recap: TCP Problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows compete before discovering available capacity
- AIMD impractical for high-speed links
- Sawtooth discovery too choppy for some applications
- Unfair under heterogenous RTTs
- Tight coupling with reliability mechanisms
- End-systems can cheat

Routers tell end-systems if they are congested

Routers tell end-systems what rate to send at

Could fix many of these with some help from routers!

Recap: TCP Problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows compete before discovering available capacity
- AIMD impractical for high-speed links
- Sawtooth discovery too choppy for some applications
- Unfair under heterogenous RTTs
- Tight coupling with reliability mechanisms
- End-systems can cheat

Routers tell end-systems if they are congested

Routers tell end-systems what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!

Router-assisted Congestion Control

- **Three tasks for CC:**
  - Isolation/fairness
  - Adjustment

Router-assisted Congestion Control

- Three tasks for CC:
  - Isolation/fairness
  - Adjustment
  - Detecting congestion

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

Fairness: General Approach

Fairness: General Approach

- **Routers classify packets into "flows"**
  - (For now) let's assume flows are TCP connections

Fairness: General Approach

- Routers classify packets into “flows”
  - (For now) let’s assume flows are TCP connections

- Each flow has its own FIFO queue in router

Fairness: General Approach

- Routers classify packets into “flows”
  - (For now) let’s assume flows are TCP connections
- Each flow has its own FIFO queue in router
- Router services flows in a fair-fashion
  - When line becomes free, take packet from the next flow in a fair order

Fairness: General Approach

- Routers classify packets into "flows"
  - (For now) let's assume flows are TCP connections
- Each flow has its own FIFO queue in router
- Router services flows in a fair-fashion
  - When line becomes free, take packet from the next flow in a fair order
- What does "fair" mean exactly?

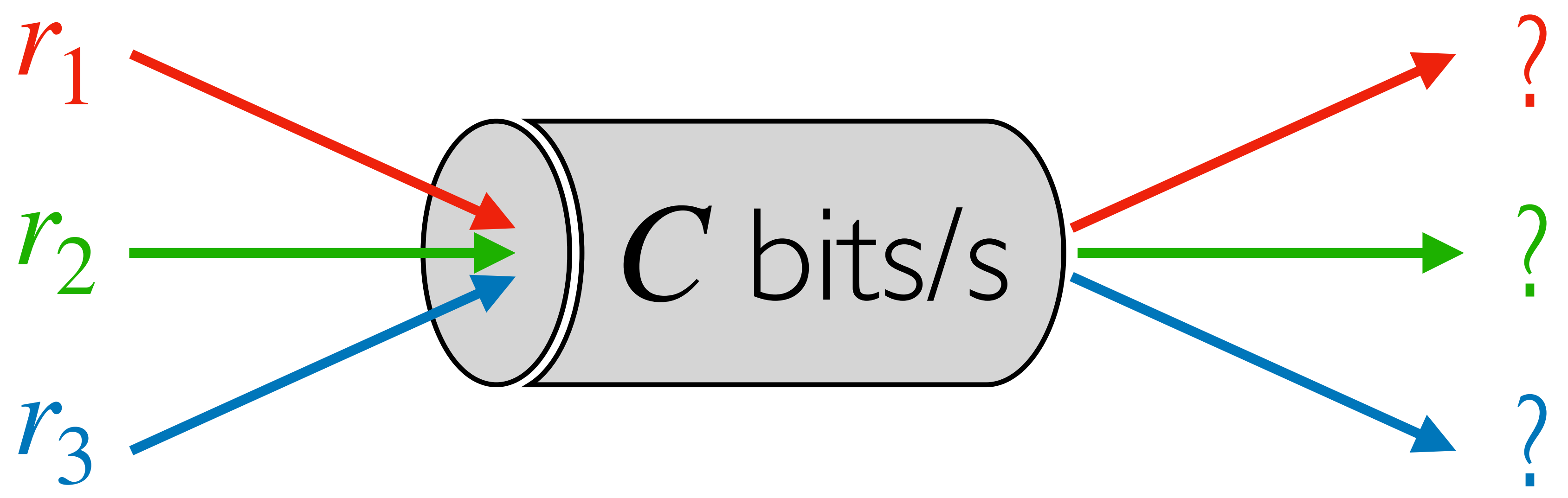
Max-Min Fairness

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 $\sum_i a_i = C$



Example

Example

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

Example

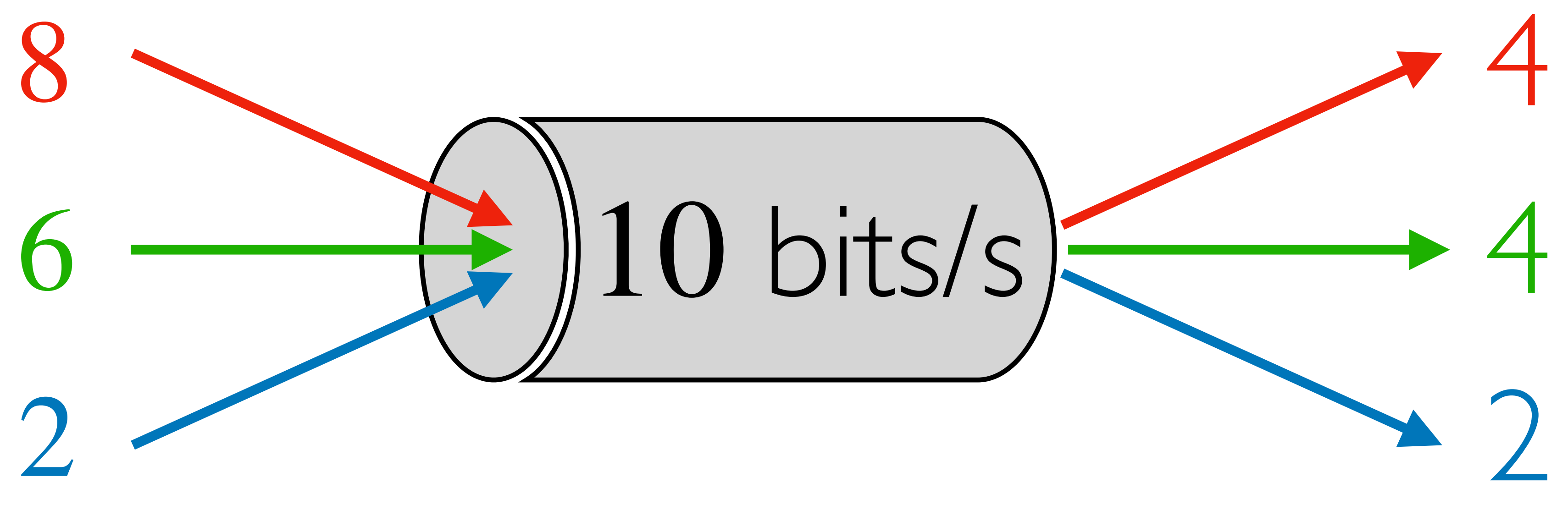
- $C = 10; r_1 = 8; r_2 = 6; r_3 = 2; N = 3$
- $C/3 = 3.33 \rightarrow$
  - But r_3's need is only 2
  - Can service all of r_3
  - Remove r_3 from the accounting: $C = C - r_3 = 8; N = 2$

Example

- $C = 10; r_1 = 8; r_2 = 6; r_3 = 2; N = 3$
- $C/3 = 3.33 \rightarrow$
    - But r_3's need is only 2
    - Can service all of r_3
    - Remove r_3 from the 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: $f = 4$

Example

- $C = 10; r_1 = 8; r_2 = 6; r_3 = 2; N = 3$
- $C/3 = 3.33 \rightarrow$
  - But r_3's need is only 2
  - Can service all of r_3
  - Remove r_3 from the 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

- 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 $\sum_i a_i = C$

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

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 $\sum_i a_i = C$

- 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”)
- Can you do this in practice?

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin (“fluid flow”)
- Can you do this in practice?
- No, packets cannot be preempted

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin (“fluid flow”)
- Can you do this in practice?
- No, packets cannot be preempted
- But we can approximate it
  - This is what “fair queueing” routers do

Fair Queueing (FQ)

Fair Queueing (FQ)

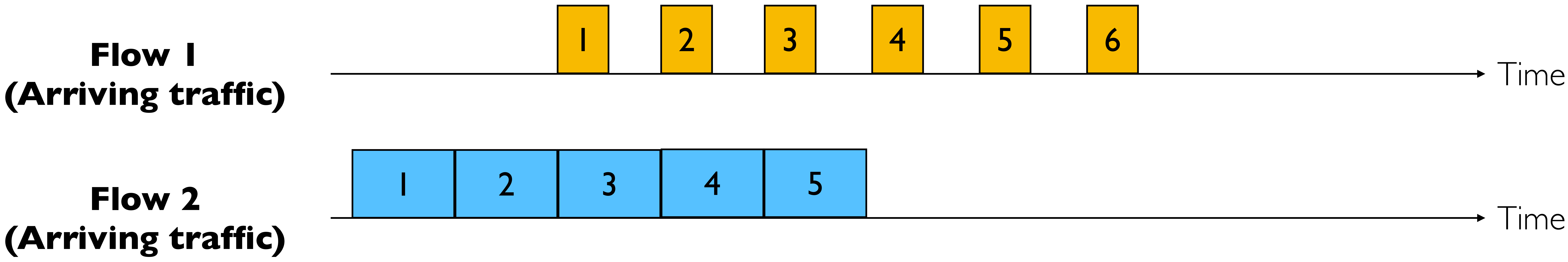
- For each packet, compute the time at which the last bit of a packet would have left if the router *if* the flows were served bit-by-bit

Fair Queueing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left if the router *if* the flows were served bit-by-bit

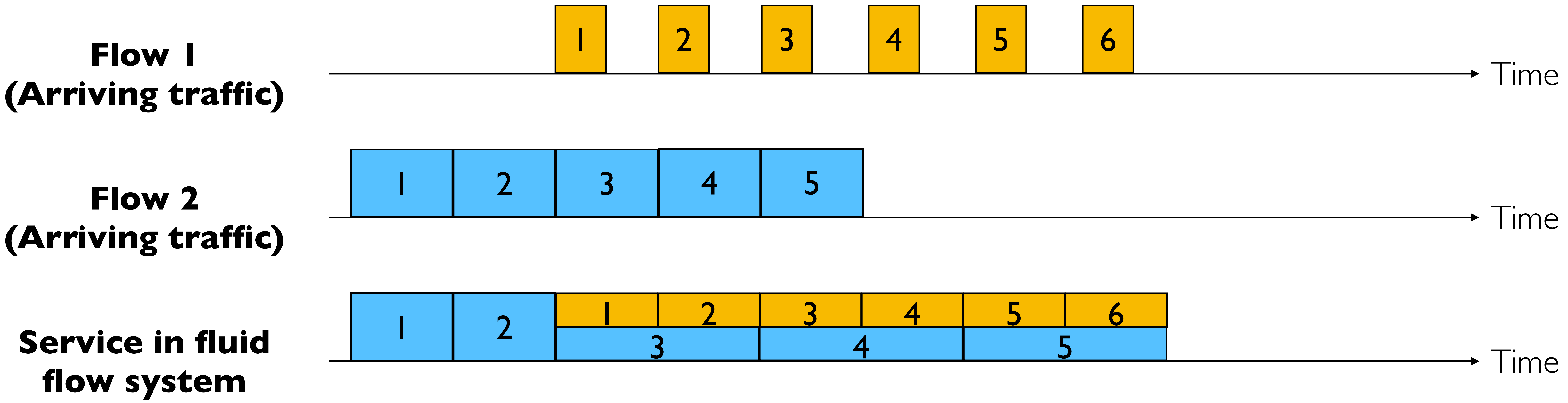
- Then service packets in the increasing order of their deadlines

Example

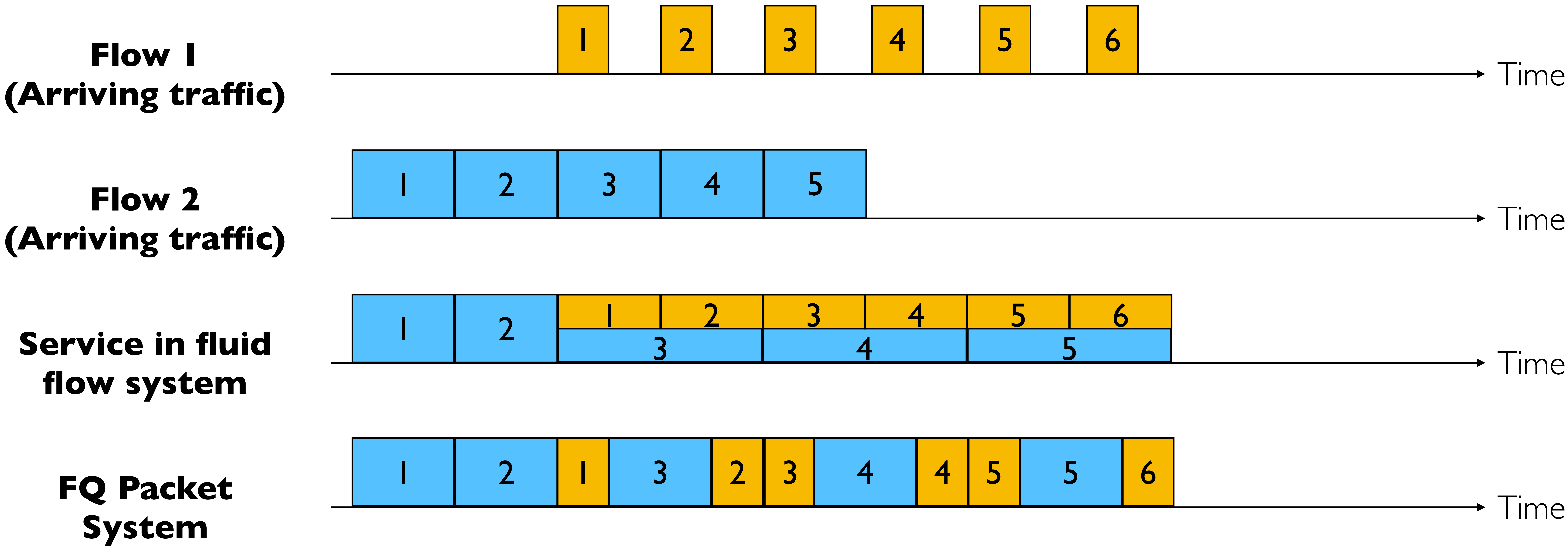


Flow 1
(Arriving traffic)
1
2
3
4
5
6
Time
Flow 2
(Arriving traffic)
1
2
3
4
5
Time

Example


Flow 1
(Arriving traffic)
1 2 3 4 5 6
Time
Flow 2
(Arriving traffic)
1 2 3 4 5
Time
Service in fluid flow system
1 2
1 2 3 4 5 6
3 4 5
Time

Example



Flow 1
(Arriving traffic)
1
2
3
4
5
6
Time
Flow 2
(Arriving traffic)
1
2
3
4
5
Time
Service in fluid flow 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

Fair Queueing (FQ)

Fair Queueing (FQ)

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

Fair Queueing (FQ)

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

- **Weighted** fair queuing (WFQ): assign different flows different shares

Fair Queueing (FQ)

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

- **Weighted** fair queuing (WFQ): assign different flows different shares

- Today, some form of WFQ implemented in almost all routers
  - Not the case in the 1980-90's, when CC was being developed
  - Mostly used to isolate traffic at larger granularities (e.g., per-prefix)

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

- Disadvantages
  - More complex than FIFO: per flow queue/state, additional per-packet book-keeping

Fairness is a controversial goal

Fairness is a controversial goal

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

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 goes only over 1?**
  - Why shouldn't you be penalized for using more scarce bandwidth?

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 goes only over 1?**
    - Why shouldn't you be penalized for using more scarce bandwidth?
- **And what is a flow anyway?**
    - TCP connection?
    - Source-destination pair?
    - Source?

Router-assisted Congestion Control

- Three tasks for CC:
  - Isolation/fairness
  - Adjustment
  - Detecting congestion

Why not just let routers tell end-systems what rate they should use?

Why not just let routers tell end-systems what rate they should use?

- Packets carry "rate field"

Why not just let routers tell end-systems what rate they should use?

- Packets carry "rate field"
- Routers insert "fair share" f in the packet header

Why not just let routers tell end-systems what rate they should use?

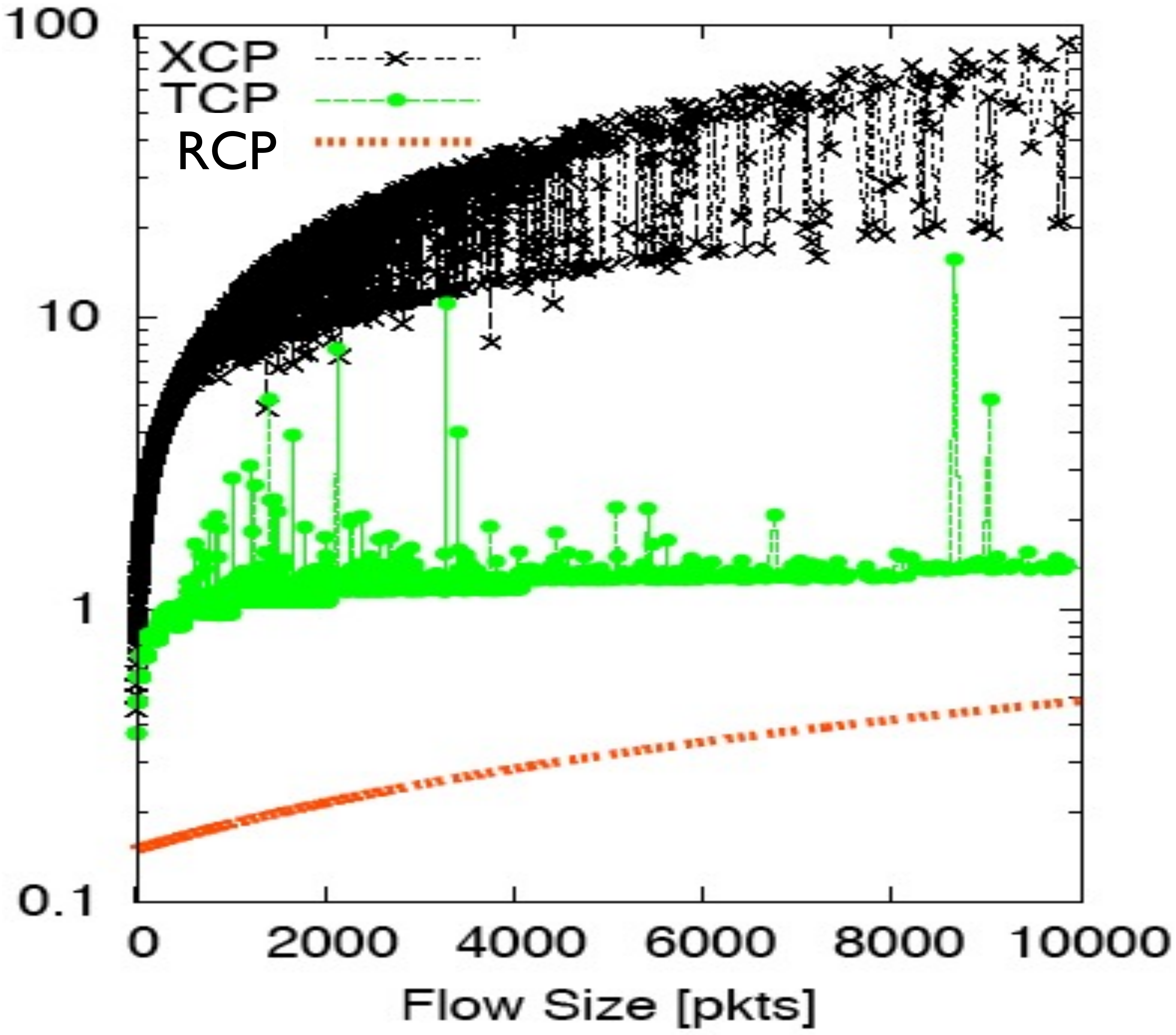
- Packets carry "rate field"
- Routers insert "fair share" f in the packet header
- End-systems set sending rate (or window size) to f
  - Hopefully (still need some policing of end-systems!)

Why not just let routers tell end-systems what rate they should use?

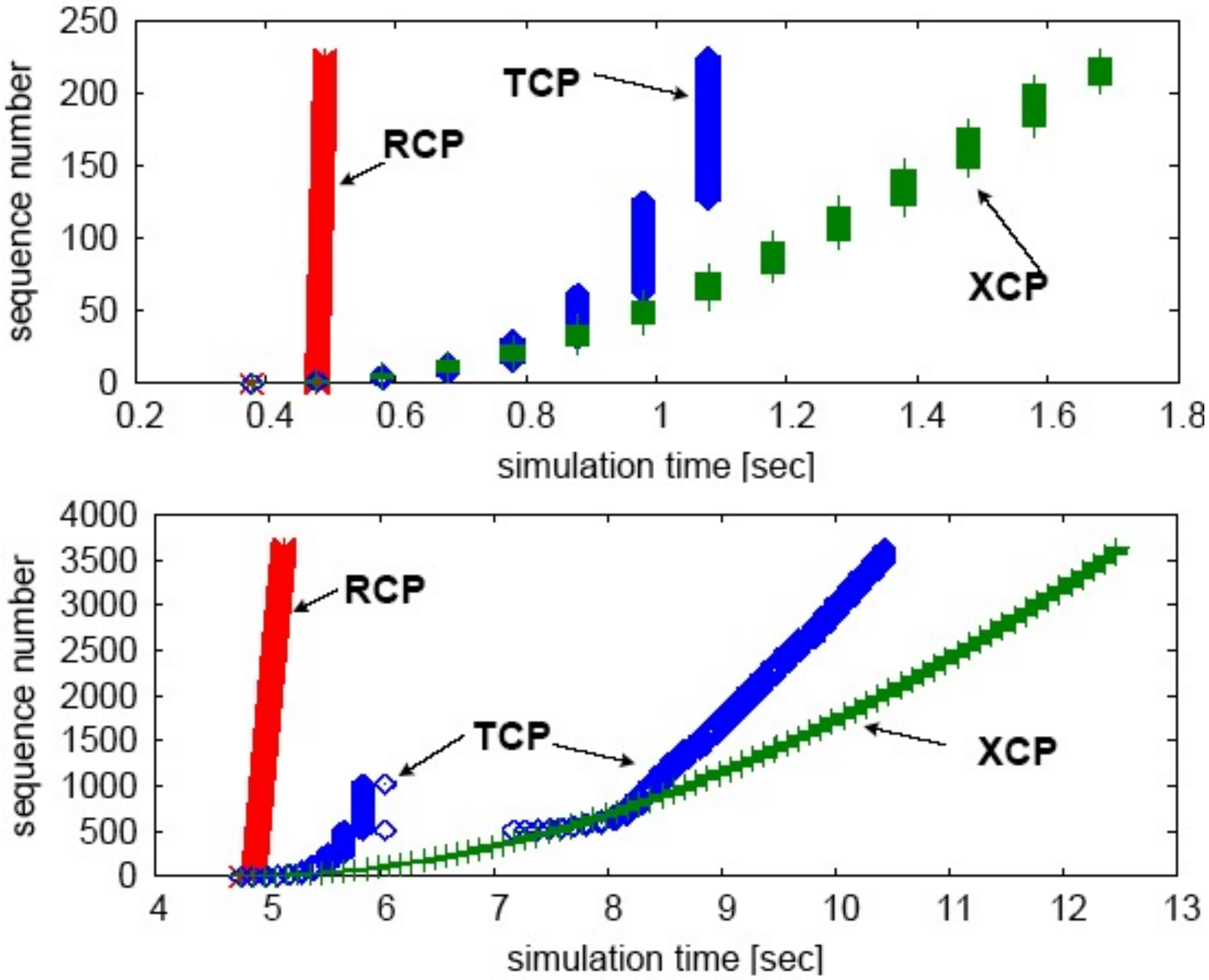
- Packets carry "rate field"
- Routers insert "fair share" f in the packet header
- End-systems set sending rate (or window size) to f
    - Hopefully (still need some policing of end-systems!)
- *Basic idea behind the "Rate Control Protocol" (RCP) from Dukkipati et. al. '07*

Flow Completion Time: TCP vs. RCP (Ignore XCP)

Flow Duration (seconds) vs. Flow Size



Why the Improvement?



Router-assisted Congestion Control

- Three tasks for CC:
  - Isolation/fairness
  - Adjustment
  - 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

- **Congestion semantics can be exactly like that of drop**
  - i.e., end-system reacts as though it saw 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
- **Congestion semantics can be exactly like that of drop**
  - i.e., end-system reacts as though it saw a drop
- **Advantages:**
  - Don't confuse corruption with congestion; recovery with rate adjustment
  - Can serve as an early indicator of congestion to avoid delays
  - Easy (easier) to incrementally deploy
    - Today: defined in RFC 3168 using **ToS/DSCP** bits in the IP header

One final proposal: Charge people for congestion!

One final proposal: Charge people for congestion!

- Use ECN as congestion markers

One final proposal: Charge people for congestion!

- Use ECN as congestion markers
- Whenever I get an ECN bit set, I have to pay $$

One final proposal: Charge people for congestion!

- Use ECN as congestion markers
- Whenever I get an ECN bit set, I have to pay $$
- Now there's no debate over what a flow is, or what fair is…

One final proposal: Charge people for congestion!

- Use ECN as congestion markers
- Whenever I get an ECN bit set, I have to pay $$
- Now there's no debate over what a flow is, or what fair is…
- Idea started by Frank Kelly at Cambridge
  - "Optimal" solution, backed by much math
  - Great idea: simple, elegant, effective
  - Unclear that it will impact practice

Recap

Recap

- TCP
  - Somewhat hacky
  - But practical/deployable
  - Good enough for Internet traffic
  - Needs of datacenters might change status quo (future lecture)