Computer Networks II

TCP Congestion Control

TCP Tahoe and TCP Reno

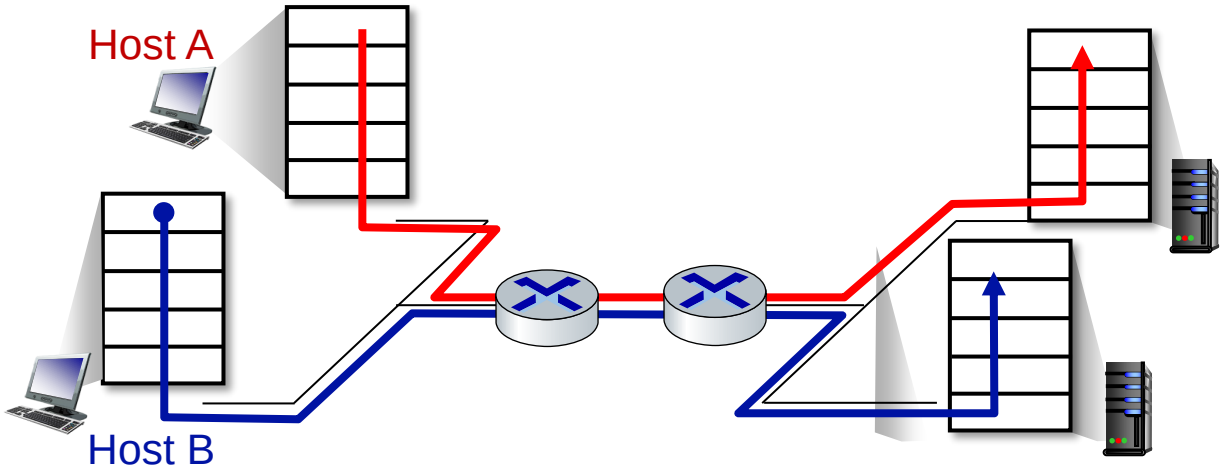
Amitangshu Pal

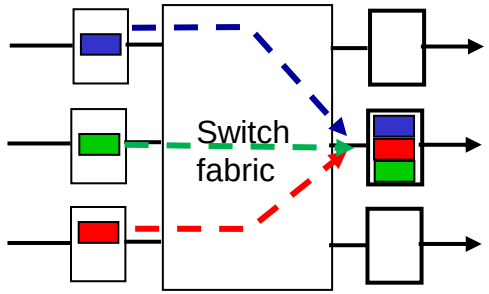
Computer Science and Engineering

IIT Kanpur

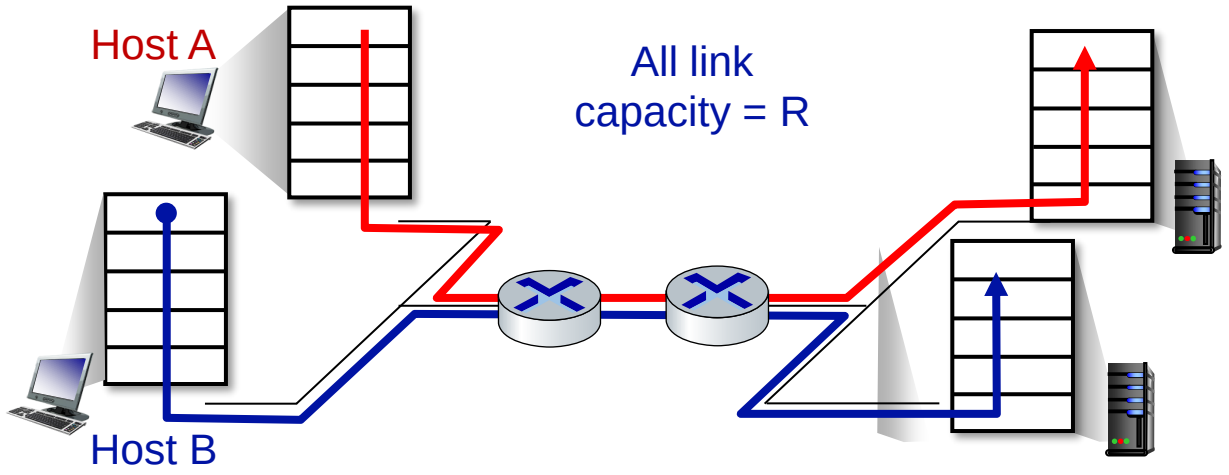
Network Congestion

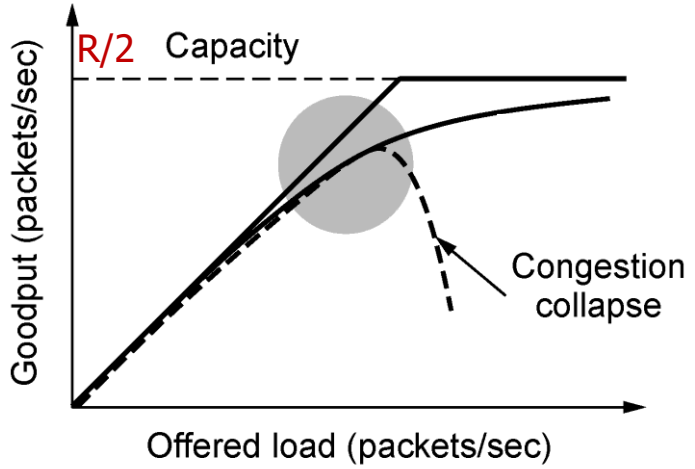
- Routers have input/output queue
  - These queues get filled up → leads to congestion

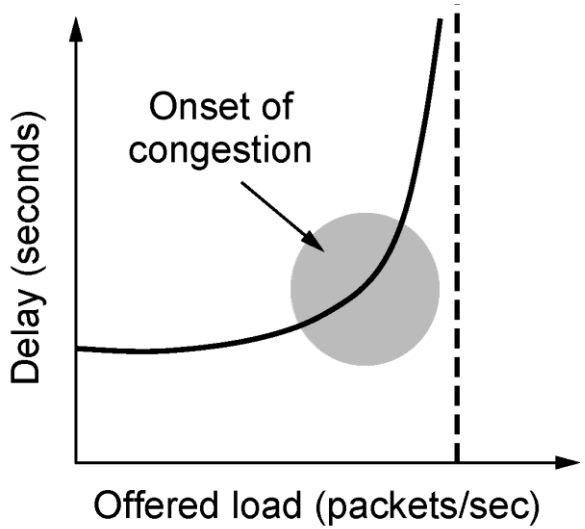




Effects of Congestion

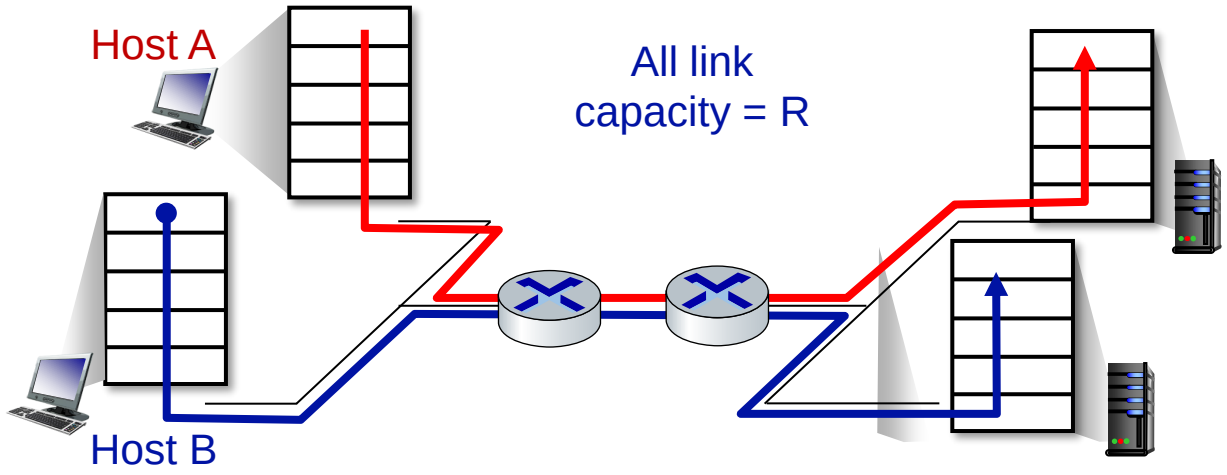


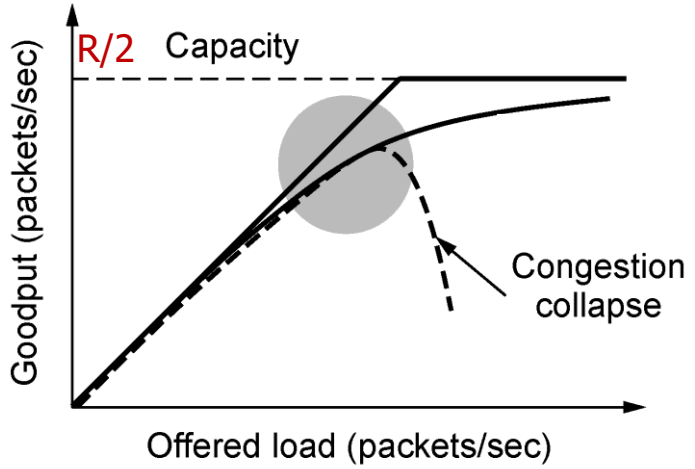


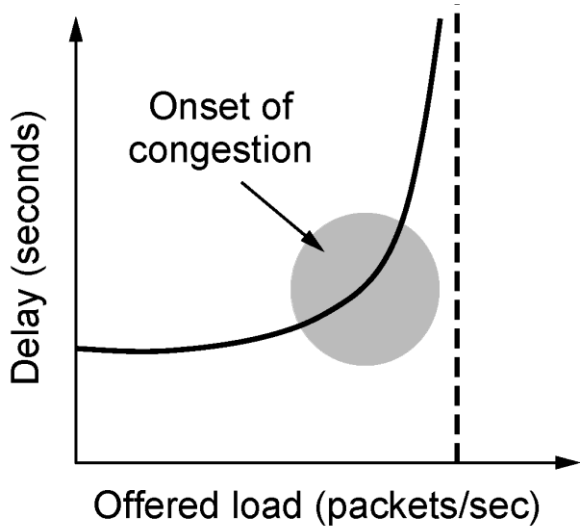


- Retransmitting undelivered packets: Packets can be lost, dropped at router due to full buffers
- Retransmitting timeout packets: Sender timers can time out prematurely, sending two copies, both of which are delivered
  - Decrease maximum achievable goodput

Effects of Congestion





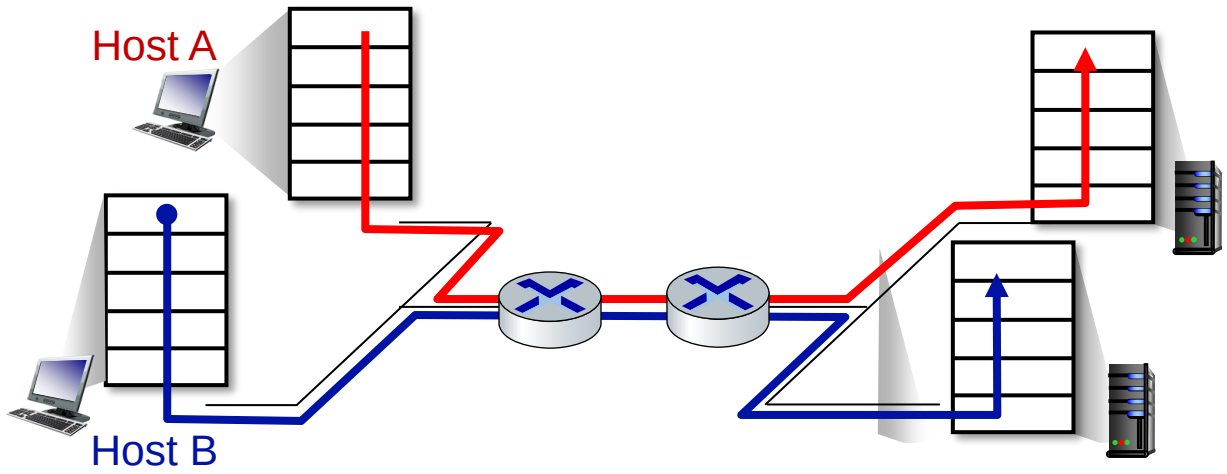


- We want to operate just before the onset of congestion
  - Avoid congestion
  - Most capacity is used
  - Needs to be fair across competitive flows

Challenges of Congestion Control

- How the senders will know whether there is a congestion or not?

- No explicit feedback from network
  - Congestion inferred from observed loss, delay

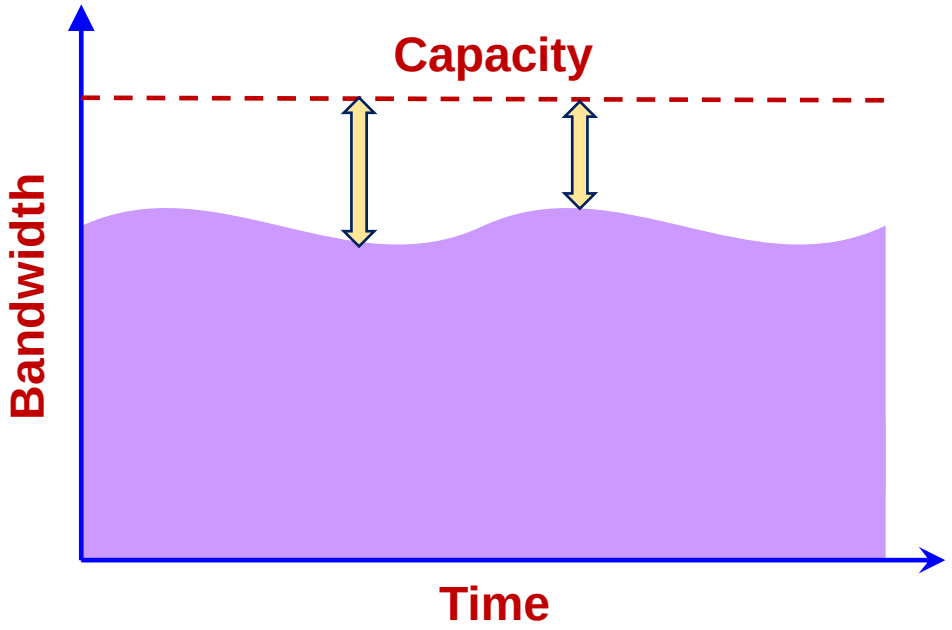


Challenges of Congestion Control

- The senders need to adapt/tune the sending rates to match the available bandwidth of the bottleneck link
  - Sending rate= function (available bandwidth of the bottleneck link)

- Which link has the lowest available bandwidth?
  - Sender does not have the global picture

- Available bandwidth varies with time
  - Number of flows and their offered load is dynamic
  - Bottleneck link can also vary



Matching Sending Rate with Available Bandwidth:
The sender needs to adjust its sending rate to match the available bandwidth of the bottleneck link in the network.
Think of it like driving a car. You need to adjust your speed to match the flow of traffic on the road. If the road is congested, you slow down; if it's clear, you can speed up.

Identifying the Bottleneck Link:
The bottleneck link is the one in the network with the lowest available bandwidth. It's like the narrowest part of a funnel where the most traffic gets congested.
However, the sender doesn't always know which link in the network is the bottleneck because it doesn't have the full picture of the network's topology.

Lack of Global Picture:
The sender only sees its own connection and doesn't have visibility into the entire network. It's like being in a crowded room but only being able to see the people immediately around you.
Variability in Available Bandwidth:
The available bandwidth on a network link can change over time. It's like traffic on a road that can vary depending on the time of day or other factors.

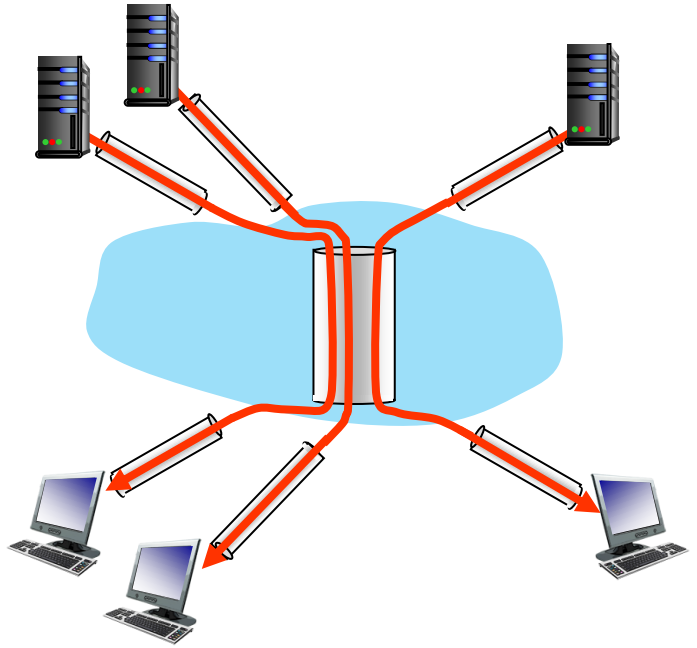
Dynamic Nature of Flows and Load:
The number of data flows and the amount of data being sent can change dynamically. It's like more cars entering or leaving the road, affecting the overall traffic flow.
Variability of Bottleneck Link:
The bottleneck link itself can change. For example, if one link becomes congested, traffic might reroute through another link, making that the new bottleneck.
Imagine you're on a highway, and suddenly there's a crash that blocks one lane. Traffic might reroute to another road, creating a new bottleneck there.

Challenges of Congestion Control

- The senders need to adapt/tune the sending rates to match the available bandwidth of the bottleneck link
  - Sending rate= function (available bandwidth of the bottleneck link)

- Which link has the lowest available bandwidth?
  - Sender does not have the global picture

- Available bandwidth varies with time
  - Number of flows and their offered load is dynamic
  - Bottleneck link can also vary

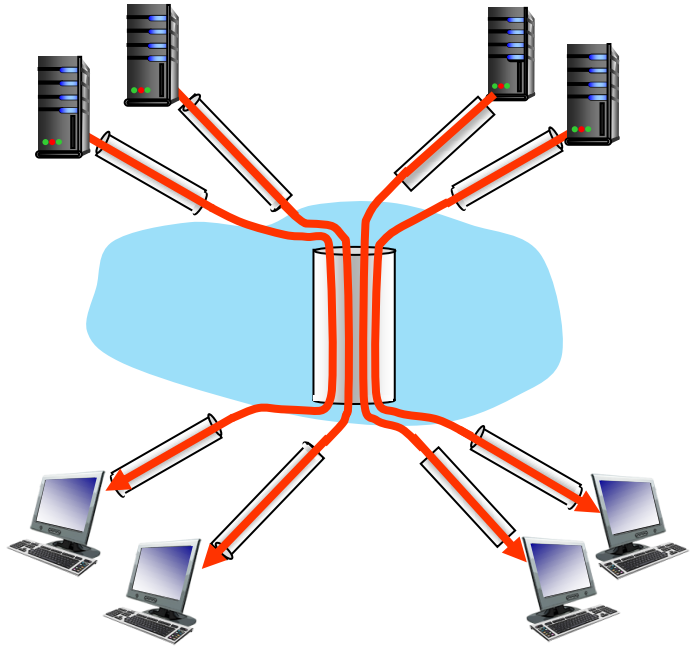


Challenges of Congestion Control

- The senders need to adapt/tune the sending rates to match the available bandwidth of the bottleneck link
  - Sending rate= function (available bandwidth of the bottleneck link)

- Which link has the lowest available bandwidth?
  - Sender does not have the global picture

- Available bandwidth varies with time
  - Number of flows and their offered load is dynamic
  - Bottleneck link can also vary



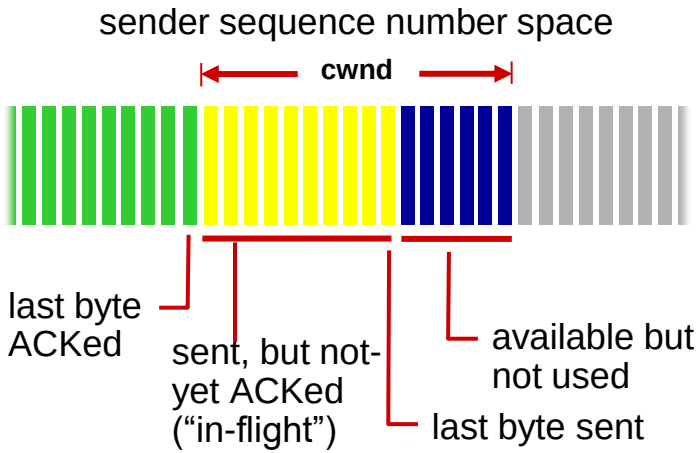
TCP Congestion Control

- Senders need to adapt/tune the sending rate
  - How many bytes are inside the network → called congestion window or cwnd

- TCP sender limits transmission: **LastByteSent - LastByteAcked ≤ cwnd**

- TCP sending behavior : send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \simeq \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$



- cwnd is dynamically adjusted in response to observed network congestion
  → Implementing TCP congestion control

---

1. Adapting Sending Rate:

TCP senders need to adjust how fast they send data based on the condition of the network.
They do this by controlling how many bytes they have "in flight" in the network at any given time. This is called the congestion window, or cwnd.
2. Congestion Window (cwnd):

The congestion window is like a limit on how much data the sender can have in the network at once.
The rule is: "LastByteSent - LastByteAcked" (the difference between the last byte sent and the last byte acknowledged) must be less than or equal to the congestion window (cwnd).
In simple terms, the sender can't send more data than what the congestion window allows.

3. TCP Sending Behavior:

TCP sends a certain amount of data, defined by the congestion window, then waits for acknowledgments (ACKs) from the receiver.
After the sender receives acknowledgments for the data sent, it can send more data.
4. TCP Rate:

The TCP sending rate is calculated by dividing the congestion window (cwnd) by the round-trip time (RTT).
This gives the rate of data transmission in bytes per second.
If the congestion window is larger, the sender can send data faster.

TCP Congestion Control: AIMD

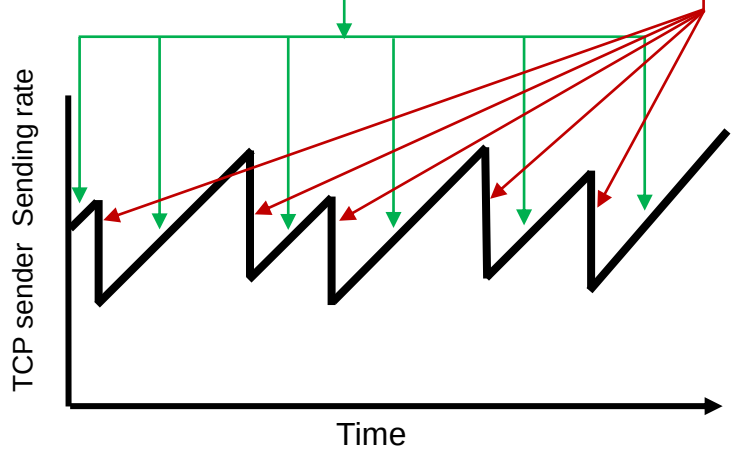
- Approach: Senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

Additively increase rate until network is not congested

Multiplicative Decrease

Multiplicatively decrease rate when congestion occurs



AIMD sawtooth behavior: probing for bandwidth

1. Additive Increase:

When the network is not congested, TCP sends data at a rate that gradually increases over time.
It does this by adding a small amount to its sending rate every time it receives an acknowledgment (ACK) from the receiver.
Think of it like driving a car: if the road is clear, you gradually accelerate to go faster.

2. Multiplicative Decrease:

If congestion occurs (e.g., packet loss is detected), TCP slows down its sending rate.
Instead of just reducing the rate by a fixed amount, it cuts it down by a larger factor.
Think of it like driving on a crowded road: if you hit traffic, you don't just slow down a little, you significantly reduce your speed.

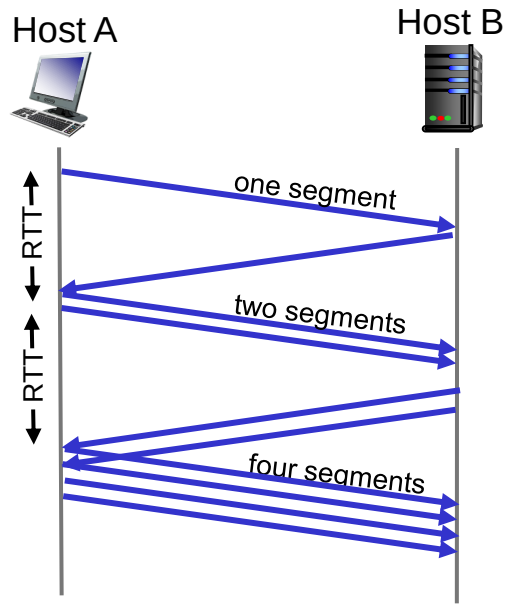
AIMD Sawtooth Behavior:

The behavior of TCP's sending rate looks like a sawtooth pattern over time.
It gradually increases until congestion is detected (a peak in the sawtooth), then drops sharply before gradually increasing again.
This pattern helps TCP find the maximum rate it can send without causing congestio

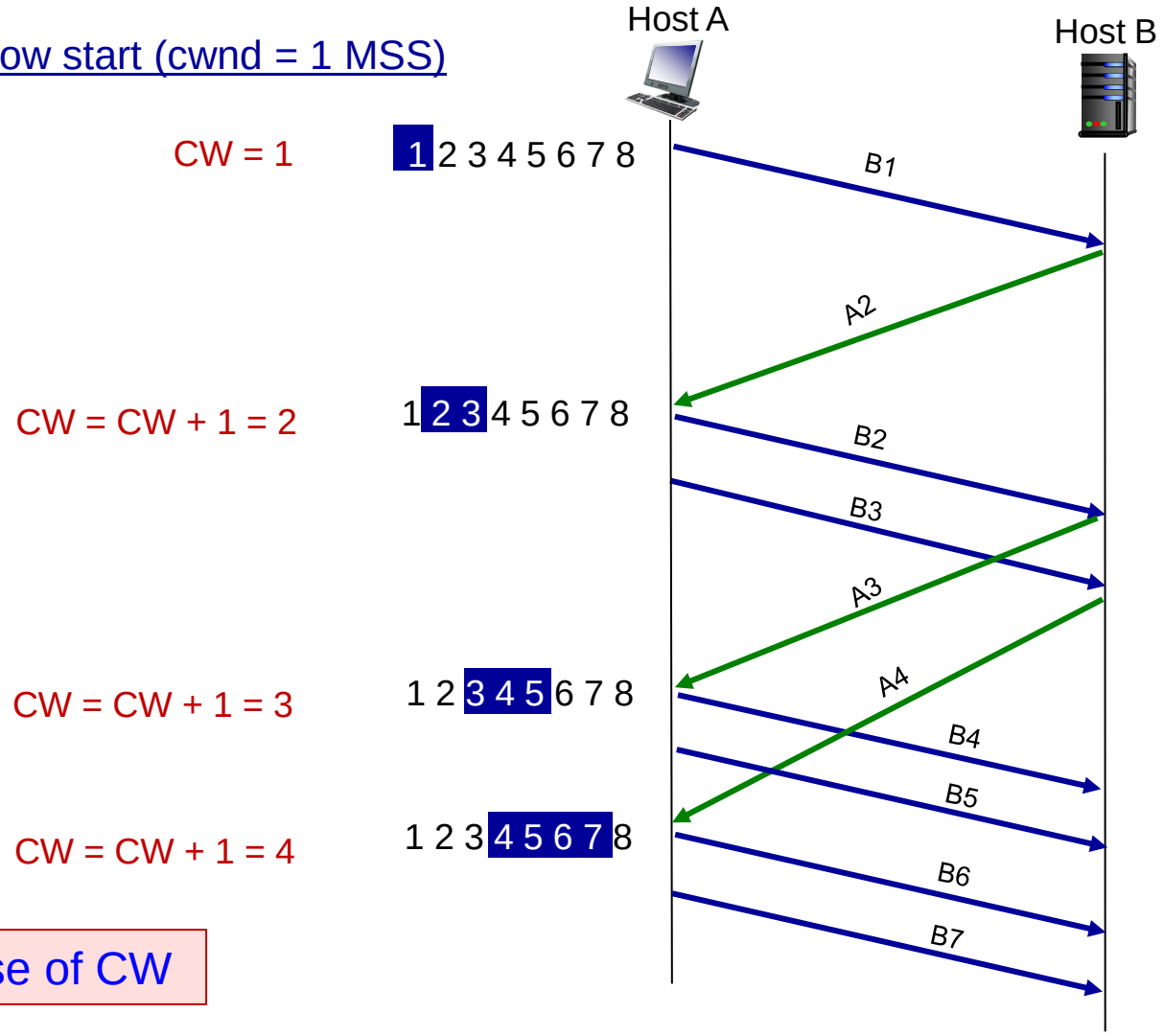
TCP Tahoe

TCP slow start

- Assumption: 1 MSS = 1 Byte



Slow start (cwnd = 1 MSS)

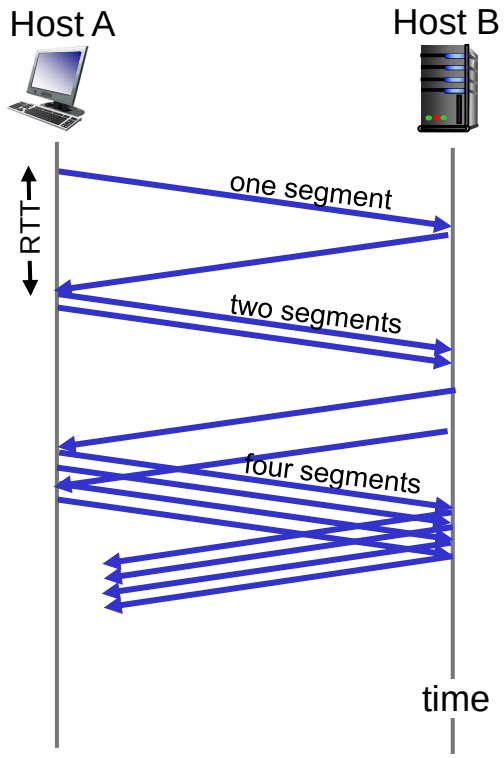


CW is doubling in every RTT → exponential increase of CW

TCP Slow Start

- When connection begins, increase rate exponentially until first loss event:
  - Initially **cwnd** = 1 MSS
  - Double **cwnd** every RTT
  - Done by incrementing **cwnd** for every ACK received

- Summary: Initial rate is slow, but ramps up exponentially fast



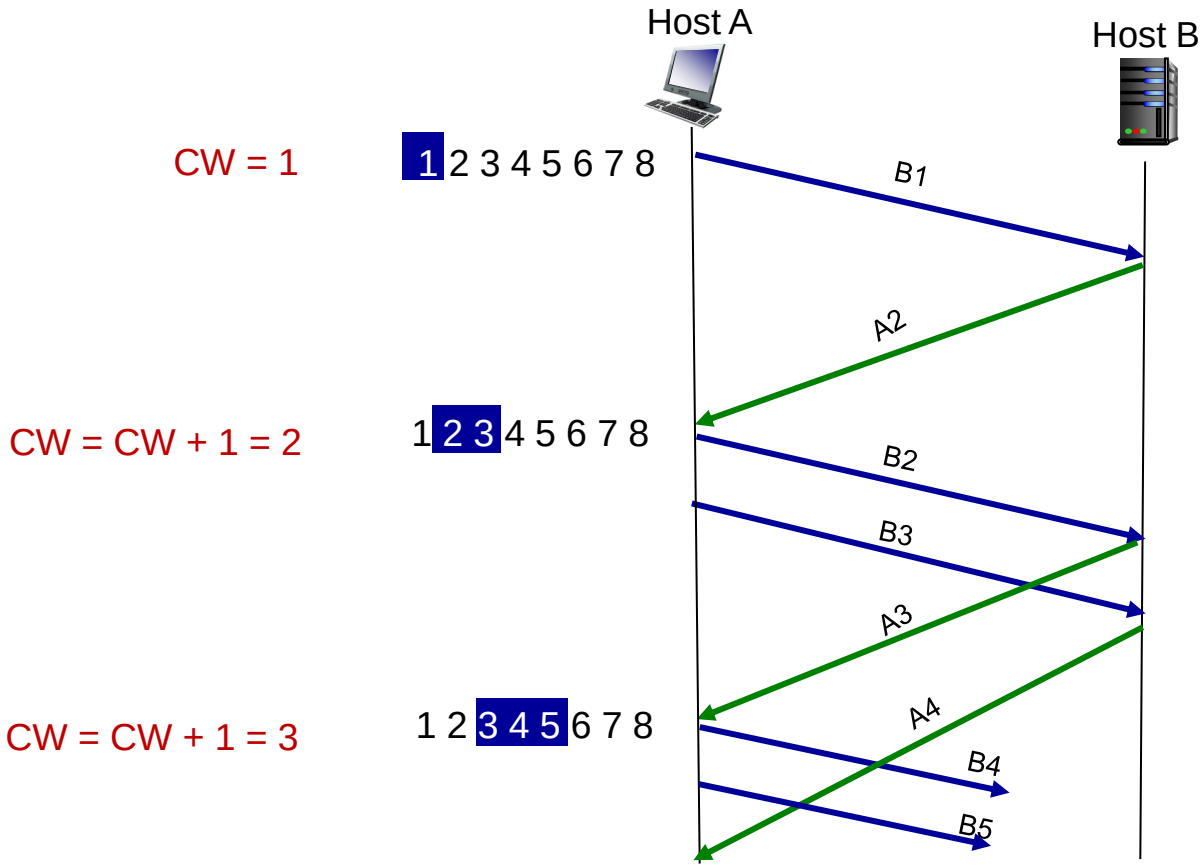
TCP Congestion Avoidance

- After ssthresh, reduce the rate of CW increase

Upon every ACK→

$$CW = CW + MSS \times \frac{MSS}{cwnd}$$

$$= CW + \frac{1}{CW}$$



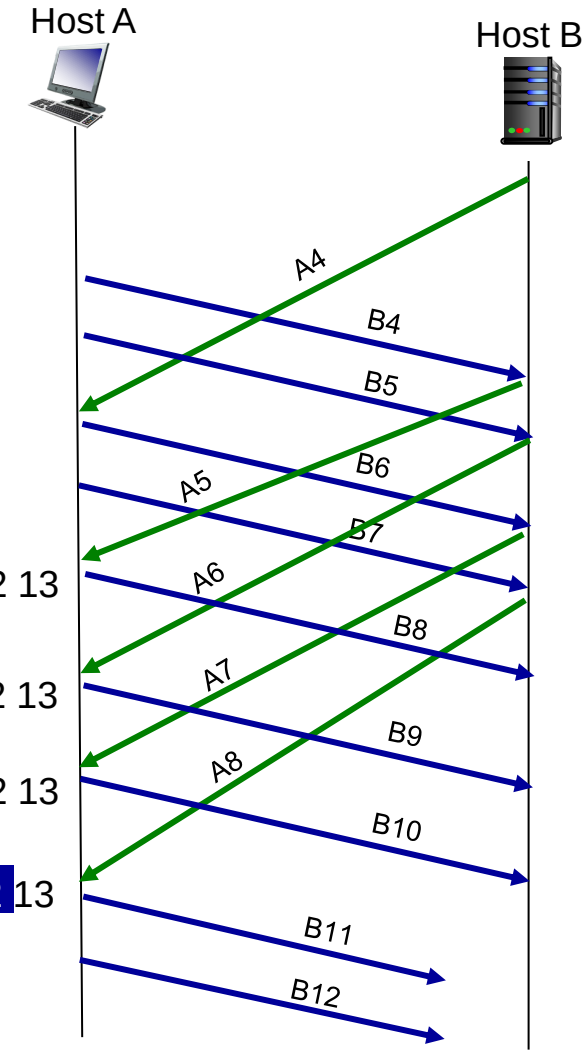
CW = CW + 1 = 4 1 2 3 4 5 6 7 8

CW = 4 + 1/4 1 2 3 4 5 6 7 8 9 10 11 12 13

CW = 4 + 2/4 1 2 3 4 5 6 7 8 9 10 11 12 13

CW = 4 + 3/4 1 2 3 4 5 6 7 8 9 10 11 12 13

CW = 4 + 4/4 = 5 1 2 3 4 5 6 7 8 9 10 11 12 13



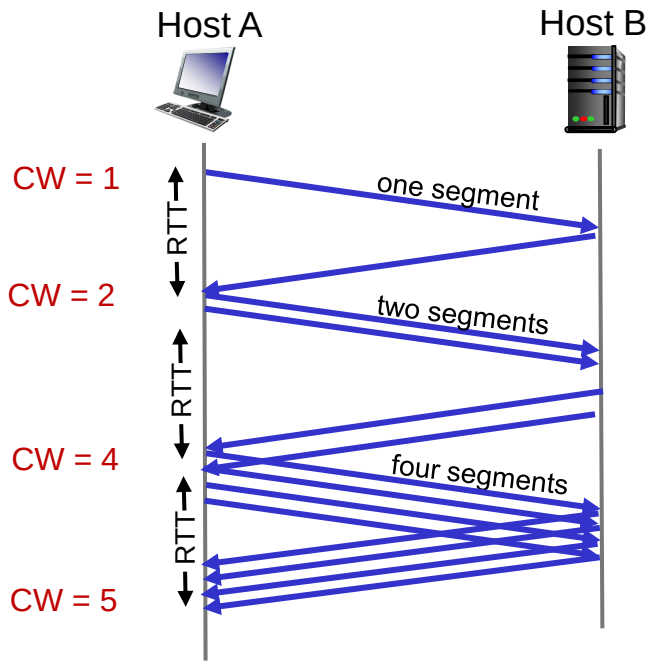
TCP Congestion Avoidance

- After ssthresh, reduce the rate of CW increase

Upon every ACK→

$$CW = CW + MSS \times \frac{MSS}{cwnd}$$

$$= CW + \frac{1}{CW}$$



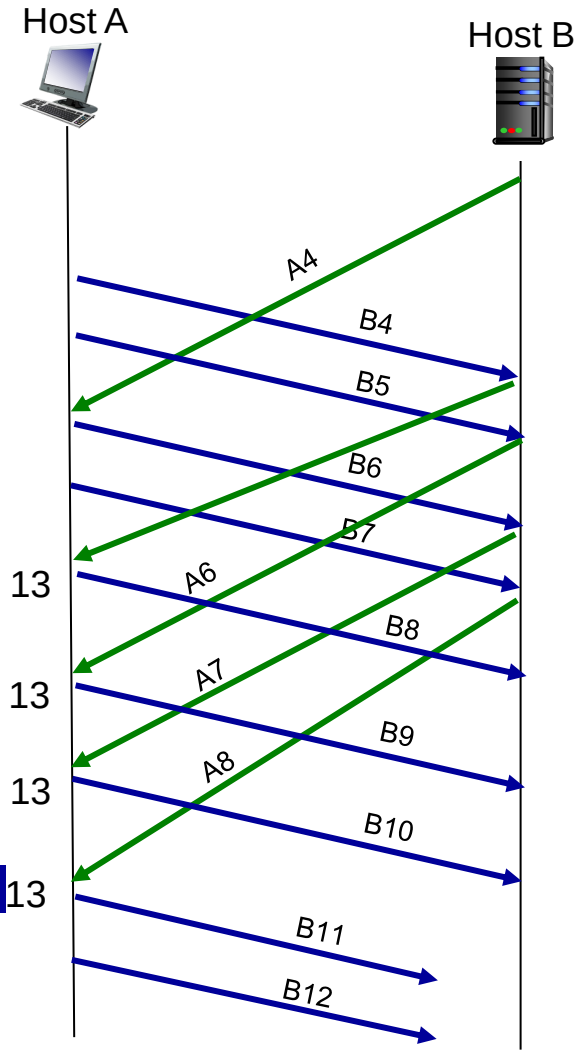
CW = CW + 1 = 4 1 2 3 **4 5 6 7** 8

CW = 4 + 1/4 1 2 3 4 **5 6 7 8** 9 10 11 12 13

CW = 4 + 2/4 1 2 3 4 5 **6 7 8 9** 10 11 12 13

CW = 4 + 3/4 1 2 3 4 5 6 **7 8 9 10** 11 12 13

CW = 4 + 4/4 = 5 1 2 3 4 5 6 7 **8 9 10 11 12** 13



After 1 RTT, the CW is approx. increasing by 1 MSS → Congestion avoidance

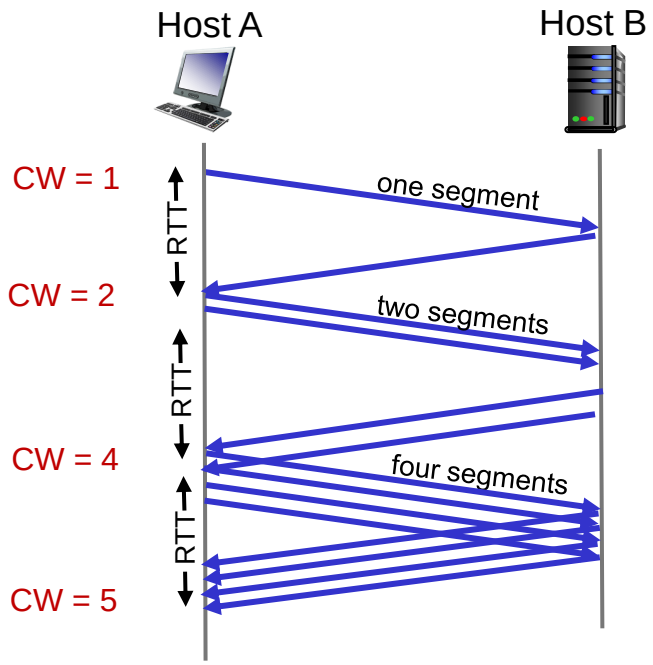
TCP Congestion Avoidance

- After ssthresh, reduce the rate of CW increase

Upon every ACK→

$$CW = CW + MSS \times \frac{MSS}{cwnd}$$

$$= CW + \frac{1}{CW}$$



If MSS = 1460 Bytes, CW = 14600 Bytes

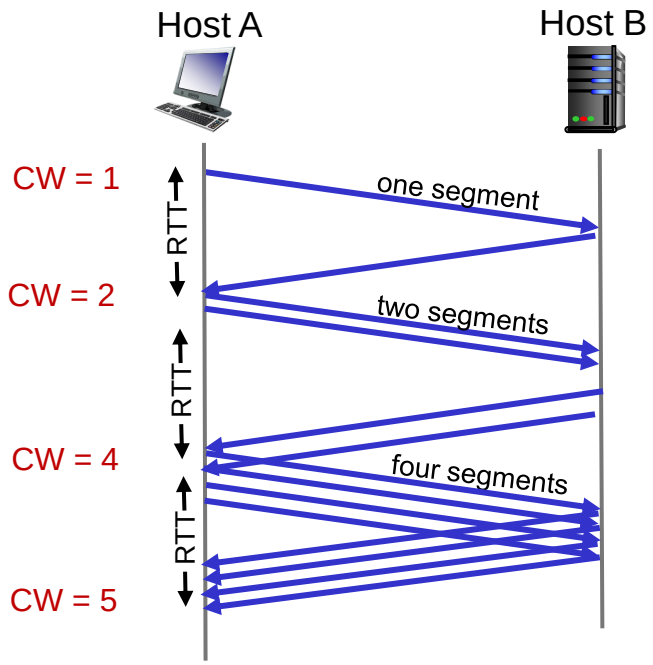
→ $CW = CW + MSS \times \frac{1}{10}$

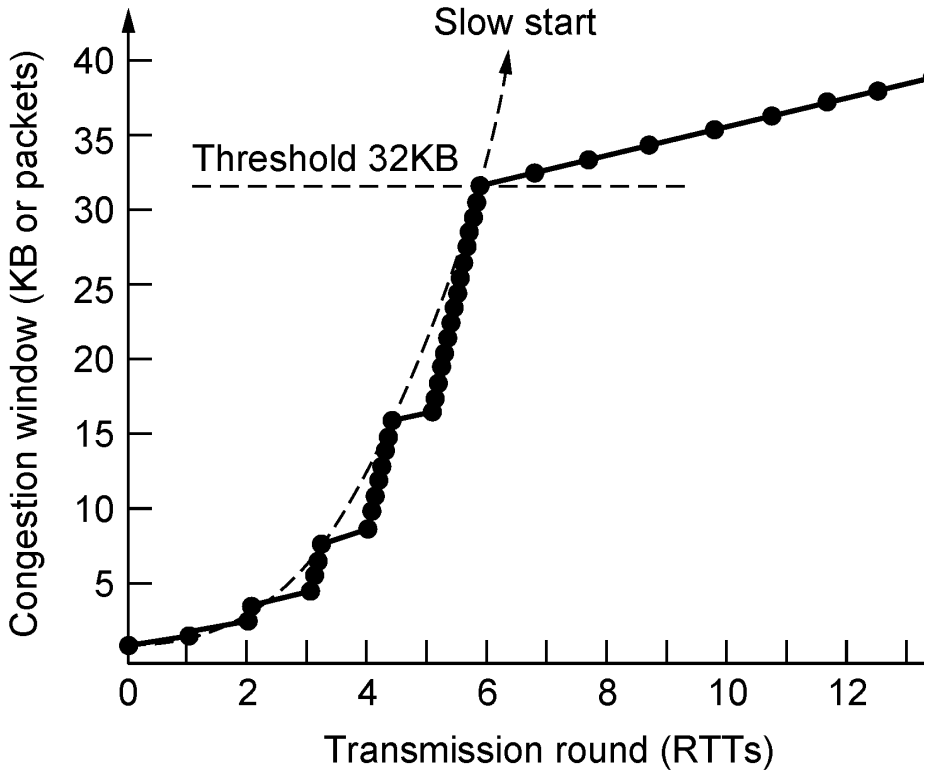
∴ CW will be increased by 1 MSS after all 10 segments will be received ≈ after 1 RTT

After 1 RTT, the CW is approx. increasing by 1 MSS → Congestion avoidance

TCP Congestion Avoidance

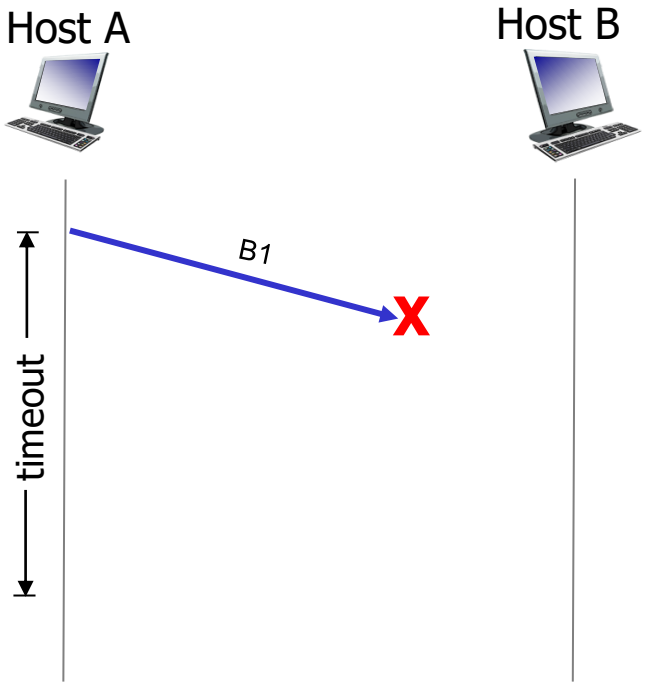
- After ssthresh, reduce the rate of CW increase

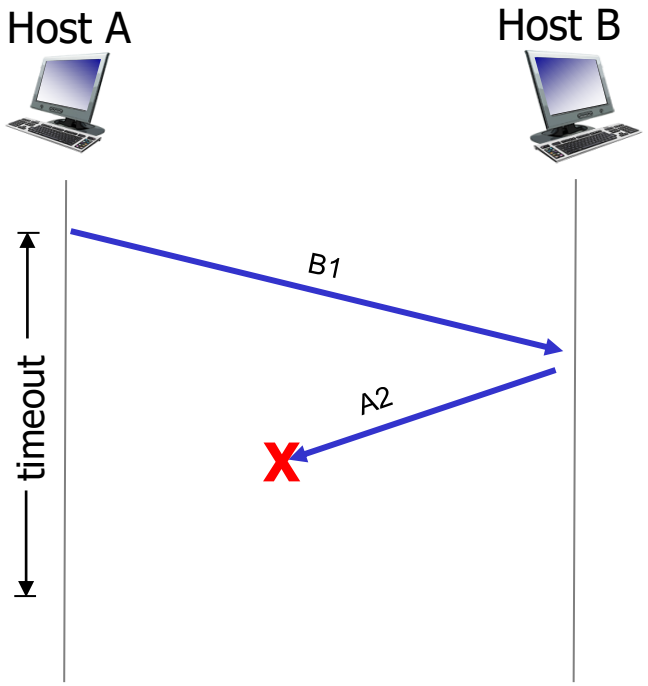


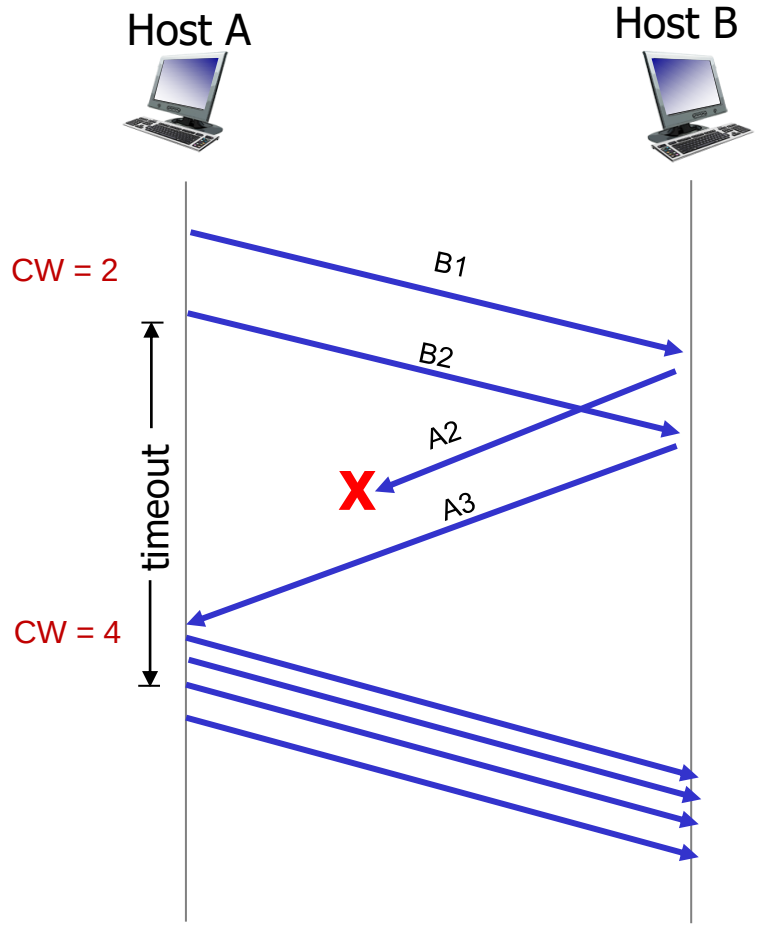


After 1 RTT, the CW is approx. increasing by 1 MSS → Congestion avoidance

Packet/ACK Drop



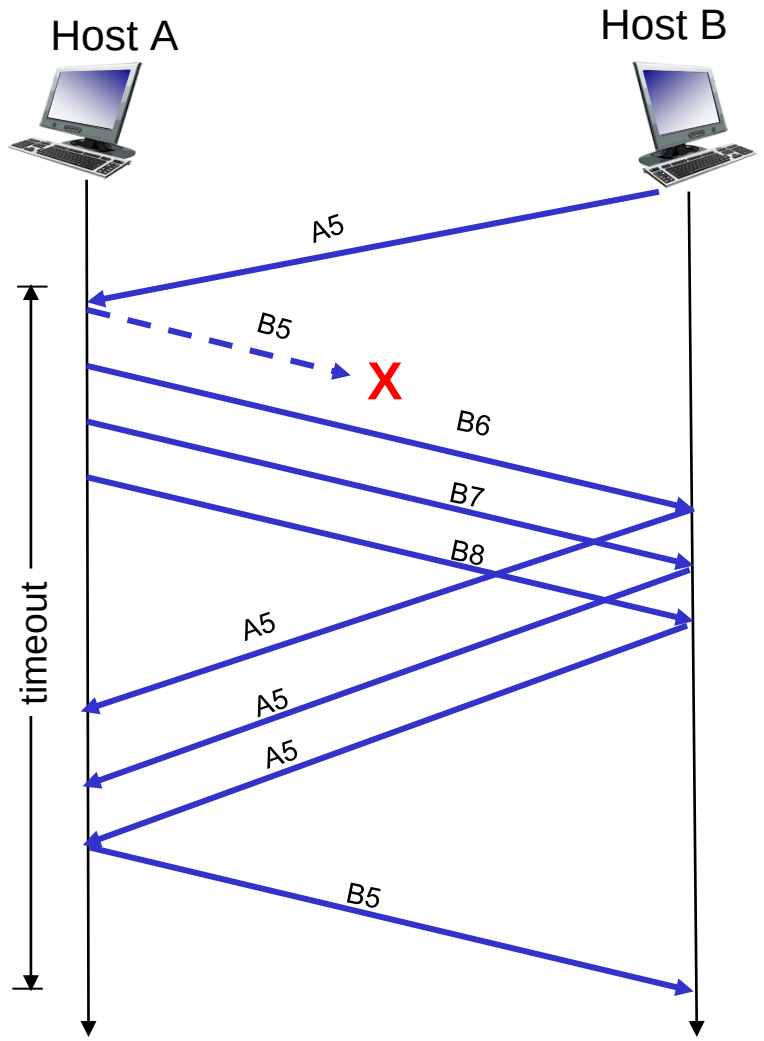




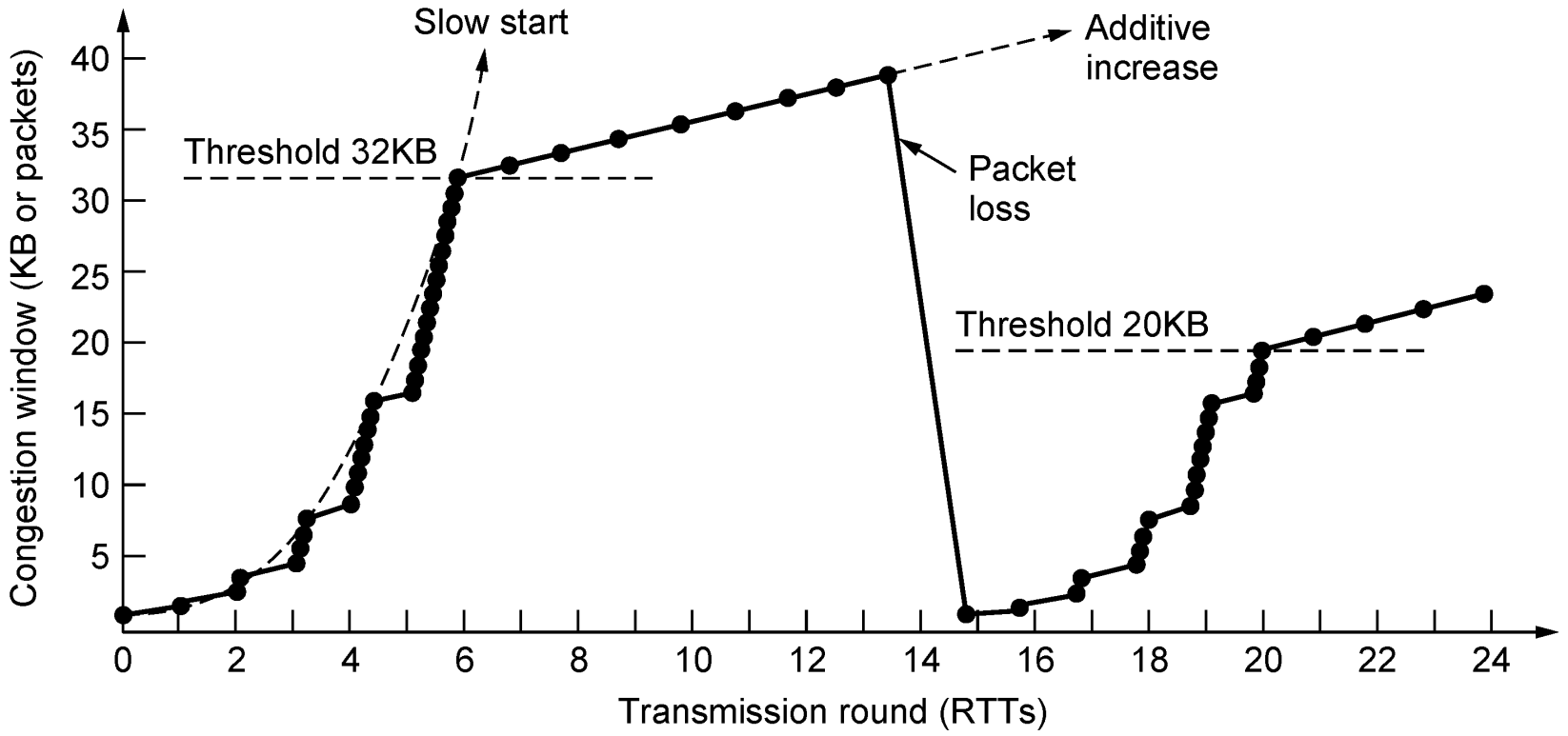
TX infers a packet loss → goes to slow start → $\begin{cases} \text{ssthresh} = \frac{\text{CW}}{2} \\ \text{CW} = 1 \end{cases}$

TCP Fast Retransmit

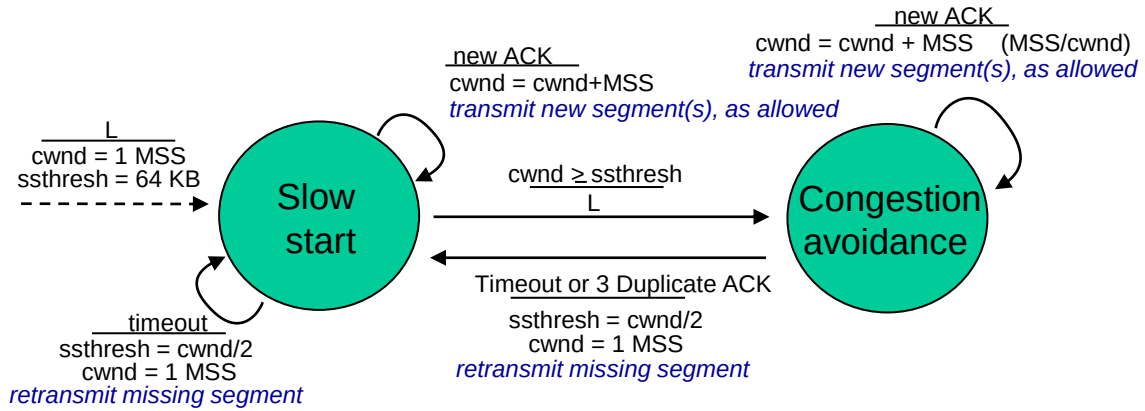
- If sender receives 3 duplicate ACKs:
  - Likely that unACKed segment lost, so don't wait for timeout
  - Resend unACKed segment with smallest seq #



TCP Tahoe: Slow start followed by additive increase



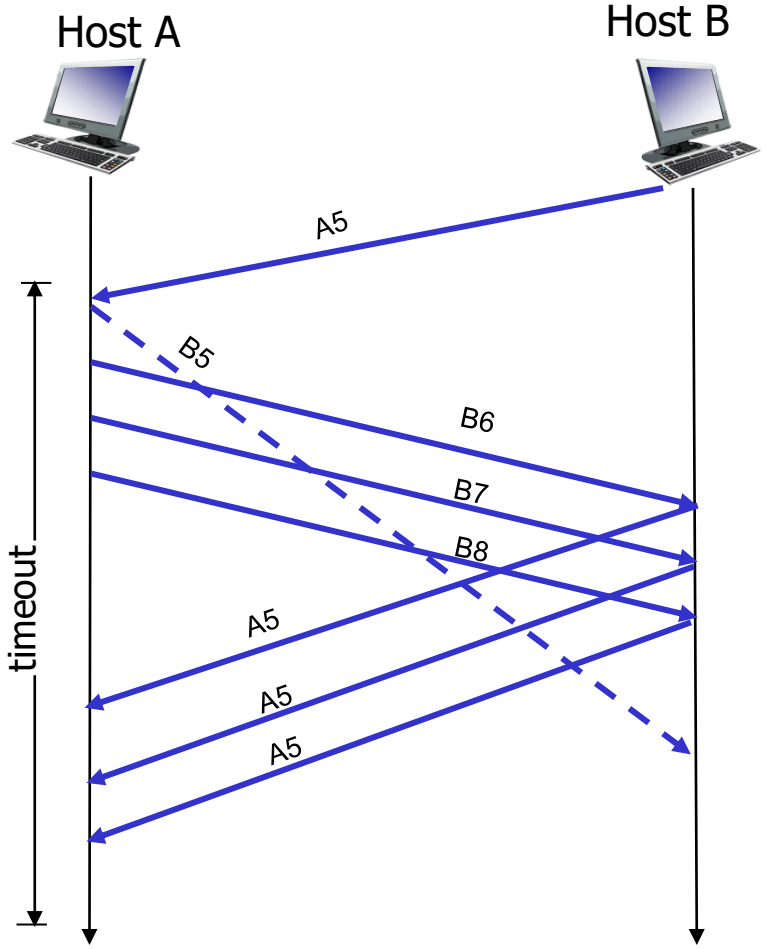
TCP Tahoe

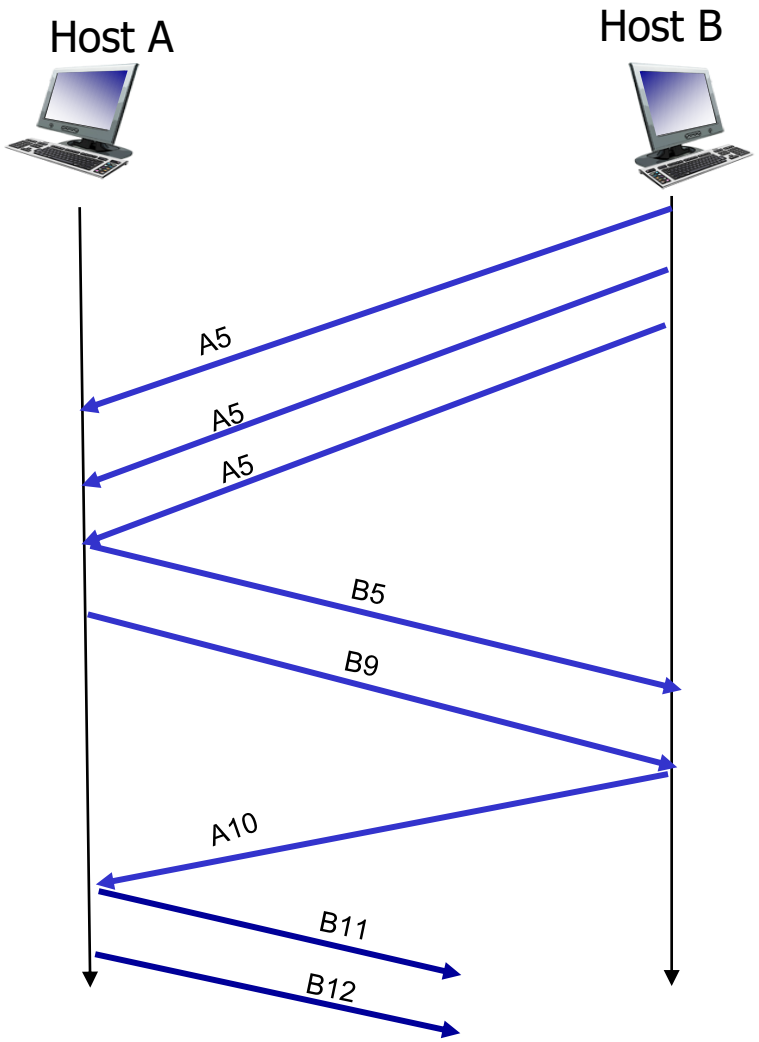

L
cwnd = 1 MSS
ssthresh = 64 KB
Slow start
new ACK
cwnd = cwnd+MSS
transmit new segment(s), as allowed
cwnd ≥ ssthresh
L
Timeout or 3 Duplicate ACK
ssthresh = cwnd/2
cwnd = 1 MSS
retransmit missing segment
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
retransmit missing segment
Congestion avoidance
new ACK
cwnd = cwnd + MSS (MSS/cwnd)
transmit new segment(s), as allowed

TCP Reno

3 Duplicate ACKs (DupACKs) Trigger Fast Retransmit:
When the sender receives three duplicate acknowledgments (DupACKs) for the same sequence number, it assumes that a packet has been lost and triggers fast retransmit.
Calculate ssthresh (Slow Start Threshold):
Set the slow start threshold (ssthresh) to half of the current congestion window (CW/2). This value is used to limit the growth of the congestion window during congestion avoidance.
Transmit Missing Segment:
Immediately retransmit the missing segment that is causing the duplicate acknowledgments. This is done to quickly recover from the packet loss.
Set CW to ssthresh + 3:
Set the congestion window (CW) to the slow start threshold (ssthresh) plus 3. This step aims to avoid slow start and achieve a more aggressive recovery from packet loss.
Transmit New Segments if CW Allows:
If the congestion window allows, the sender can transmit new segments in addition to the retransmitted segment.
More Duplicate ACKs:
If more duplicate acknowledgments are received after retransmitting the missing segment, increment the congestion window by 1 for each duplicate ACK.
This allows the sender to probe for the available network capacity without overwhelming the network.
Normal ACK:
Upon receiving a normal acknowledgment (not a duplicate), set the congestion window (CW) back to the slow start threshold (ssthresh).
This marks the end of the fast recovery phase and the beginning of congestion avoidance.
Start Congestion Avoidance:
Once the congestion window is set to ssthresh, the sender enters the congestion avoidance phase, where the congestion window grows linearly rather than exponentially.
Overall, TCP Reno's fast recovery algorithm aims to quickly recover from packet loss events by retransmitting the missing segment and adjusting the congestion window dynamically based on the network's condition. This allows TCP Reno to maintain high throughput and efficient data transmission while avoiding congestion collapse.

Fast Recovery

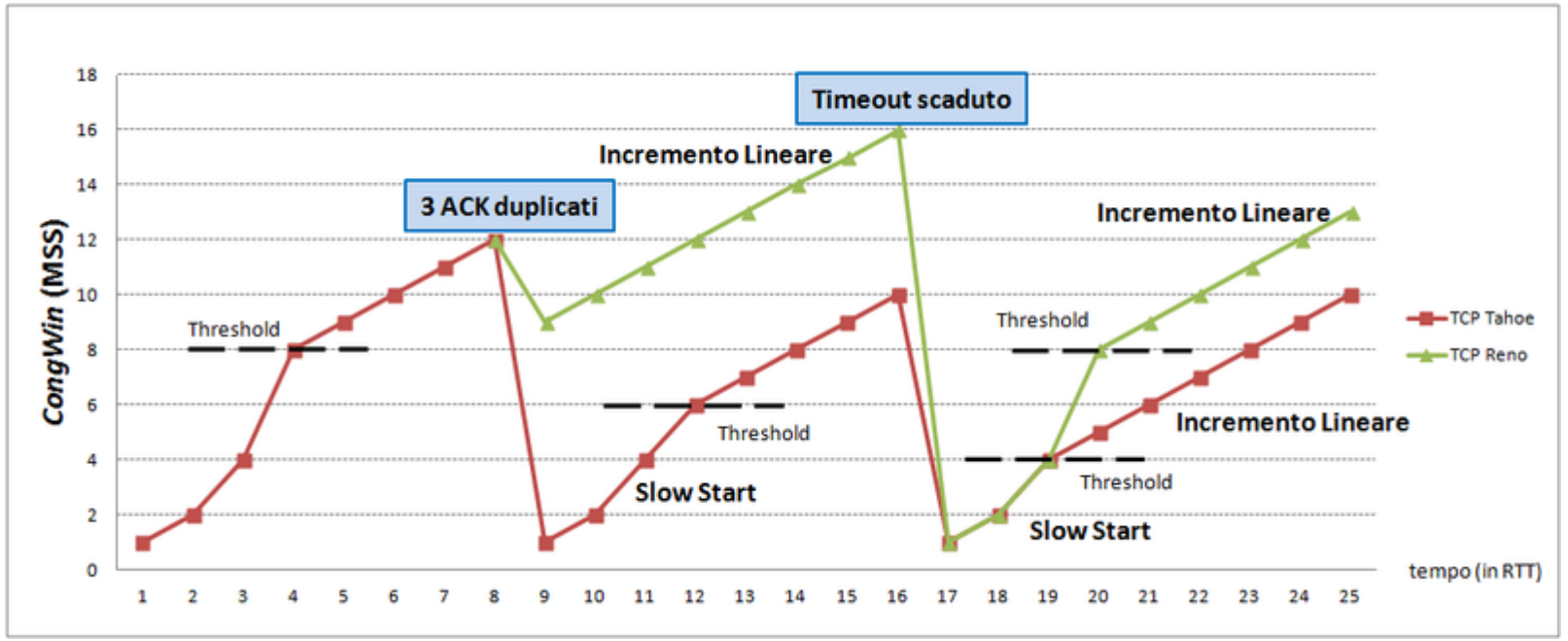




3 DupACks
→ Trigger Fast Retransmit

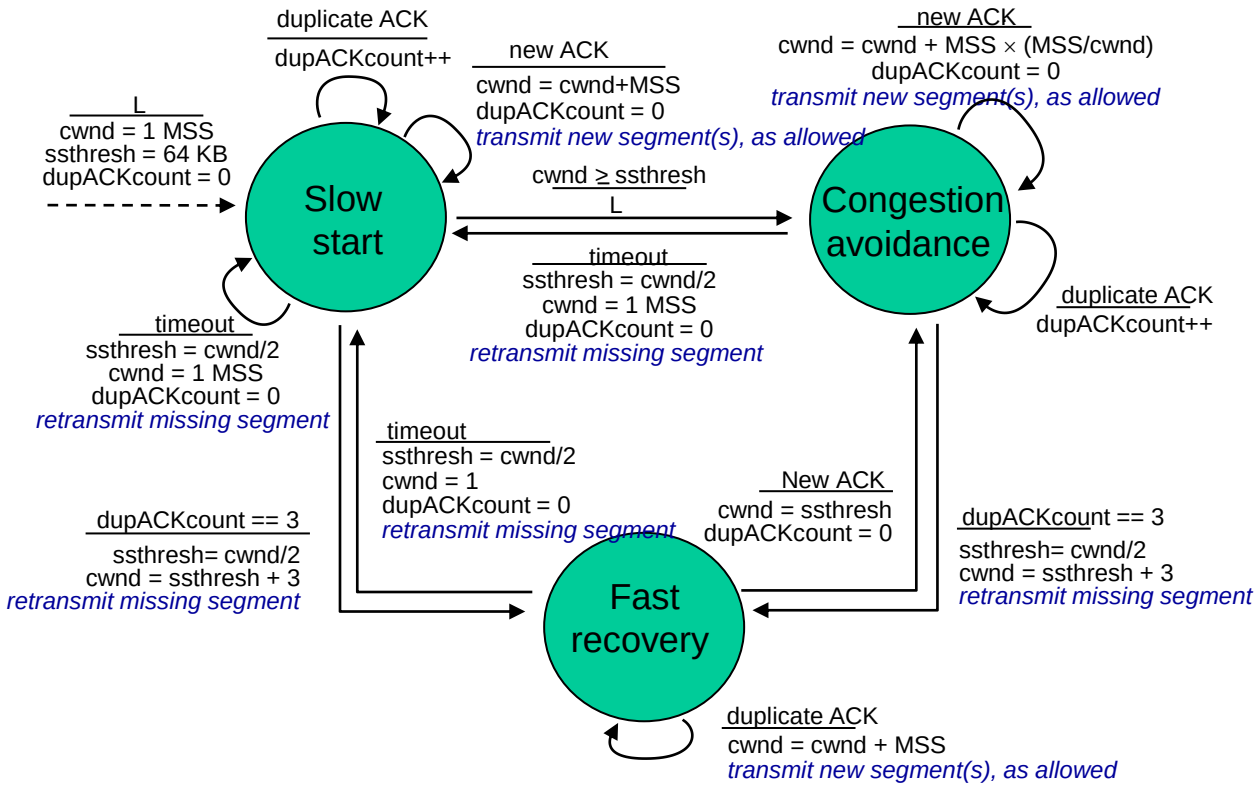
1. Ssthresh = CW/2
2. Transmit missing segment that is definitely causing DupAcks
3. CW = ssthresh + 3
4. Transmit new segments of CW allows
5. More DupAcks:
CW = CW + 1
Transmit new segments if CW allows
6. Normal ACK:
CW = ssthresh
Start congestion avoidance

TCP Reno vs TCP Tahoe



Src: https://commons.wikimedia.org/wiki/File:CongWin_in_TCP_Tahoe_e_Reno.png

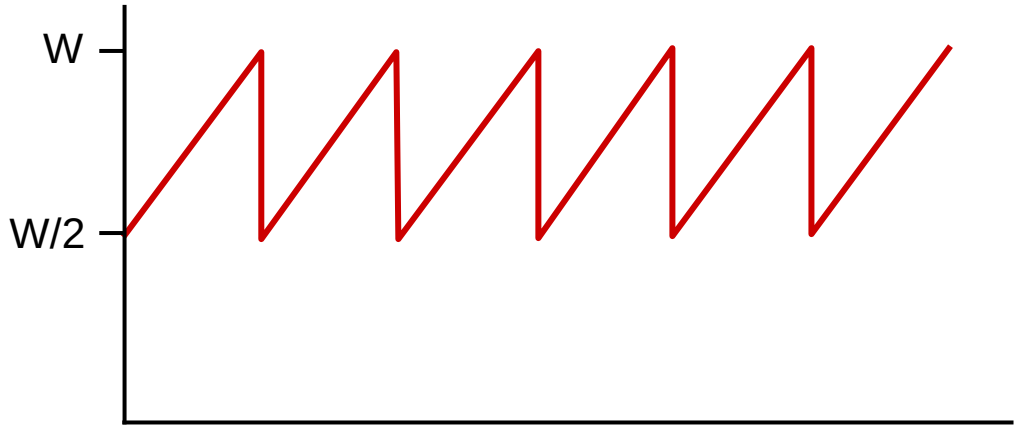
TCP Reno: TCP Tahoe + Fast recovery



TCP Throughput

- Avg. TCP throughput as function of window size, RTT?
  - Ignore slow start, assume always data to send

- W: window size (measured in bytes) where loss occurs
  - Avg. window size (# in-flight bytes) is ¾ W
  - Avg. throughput is 3/4W per RTT



Avg TCP throughput = $\frac{3}{4}\frac{W}{RTT}$ bytes/sec

Summary

- Network congestion control:
  - Challenges of congestion
  - TCP congestion control
    - TCP Tahoe and TCP Reno