# CS 43: Computer Networks

TCP Congestion Control

November 4, 2026



# Quiz!

# No Reading Quiz!

# Announcements

- Updated schedule to everything by one day
- Lab related
  - Test your server on catnip.cs.swarthmore.edu (see lab writeup for details)
  - When using printf statements to debug, always end them with a \n. Output is “buffered” by printf, so it doesn’t flush to stream until that buffer is full or you force a flush. You can do that with a \n.
- Some issues with Lab 2 grading which I’ve discussed with my graders. I will make these fixes, push updated rubrics, and I will let you all know.

# Practical Reliability Questions

- What does connection establishment look like?
- How do we choose sequence numbers?
- How should we choose timeout values?
- How do the sender and receiver keep track of outstanding pipelined segments?
- How many segments should be pipelined?

# Sliding window

- How many bytes to pipeline?
- How big do we make that window?
  - Too small: link is under-utilized
  - Too large: congestion, packets dropped
  - Other concerns: fairness

# What is a “congestion event” from the perspective of a sender in TCP?

A. A segment loss

B. Receiving duplicate acknowledgement(s)

C. A retransmission timeout firing

D. Some subset of the above

E. All of the above

What we care about is segment loss, and both B and C give us a way to know that a segment loss has occurred.

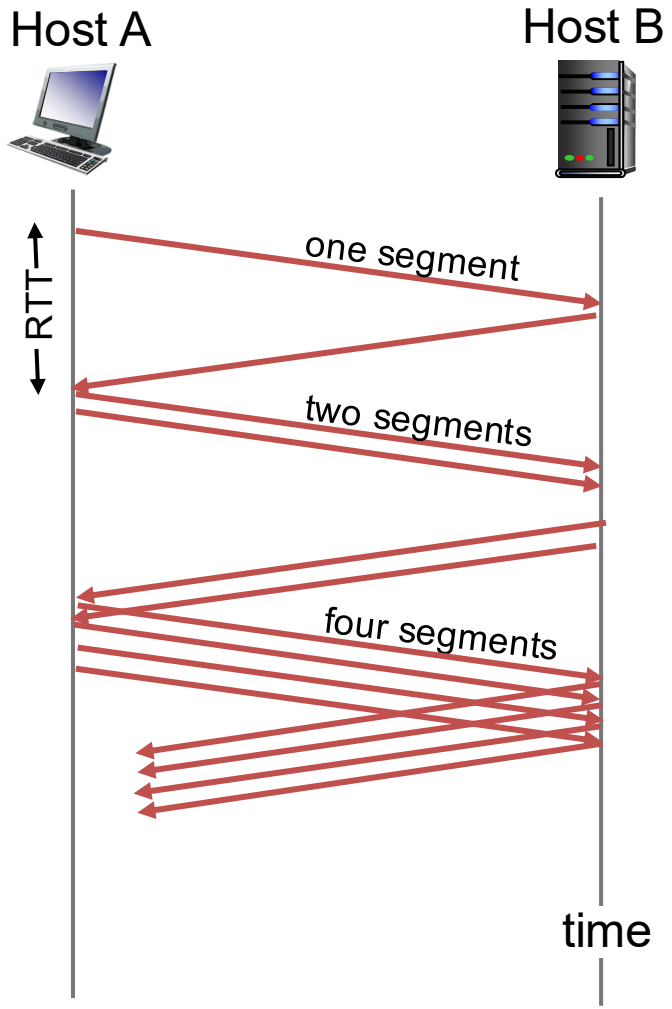
# TCP Congestion Control Phases

- Slow start
  - Sender has no idea of network's congestion
  - Start conservatively, increase rate quickly

- Congestion avoidance
  - Increase rate slowly
  - Back off when congestion occurs
    - How much depends on TCP version

# TCP Slow Start

- When do we stop?
- Initially
  - On a congestion event
- Later
  - On a congestion event
  - When we cross a previously-determined threshold

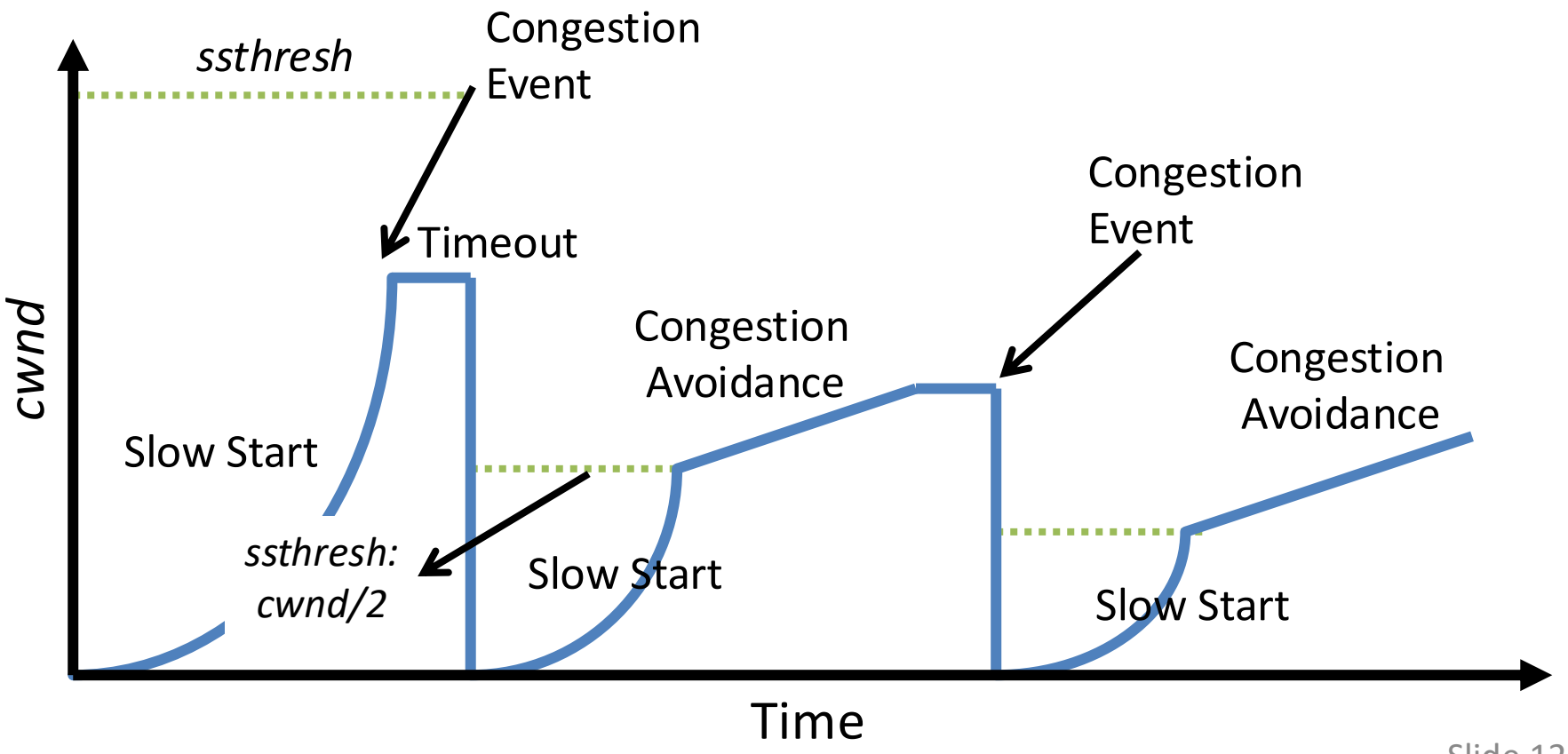


# TCP Congestion Avoidance

- `ssthresh`: Threshold where slow start ends
  - initially unlimited

- In congestion avoidance, instead of doubling, increase `cwnd` by one MSS every RTT.
  - Increase `cwnd` by MSS/cwnd bytes for each ACK
  - Back off on congestion event

# TCP: Big picture



We can determine that a packet was lost two different ways: via 3 duplicate ACKS, or via a timeout. We should…

A. Treat these events differently.

B. Treat these events the same.

# Detecting, Reacting to Loss (Tahoe vs. Reno)

Loss indicated by timeout:

Tahoe and Reno:

- cwnd set to 1 MSS;
- window then grows exponentially (as in slow start) to threshold,
- then grows linearly

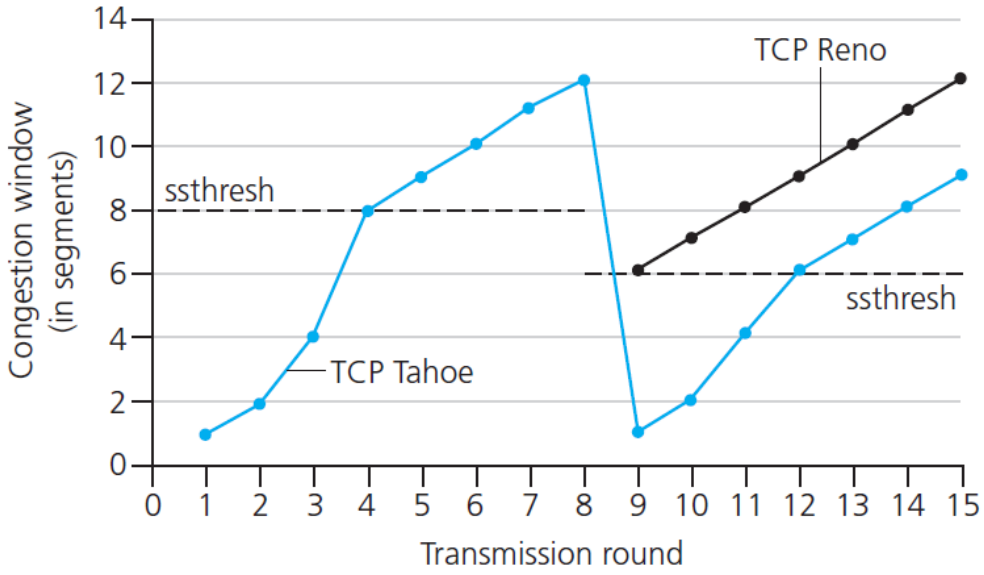
Loss indicated by 3 duplicate ACKs:

- Tahoe:
  - cwnd set to 1 MSS;
  - window grows exponentially (as in slow start) to threshold
  - then grows linearly
- Reno
  - cwnd is cut in half window then grows linearly
  - dup ACKs indicate network capable of delivering some segments

# TCP: switching from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

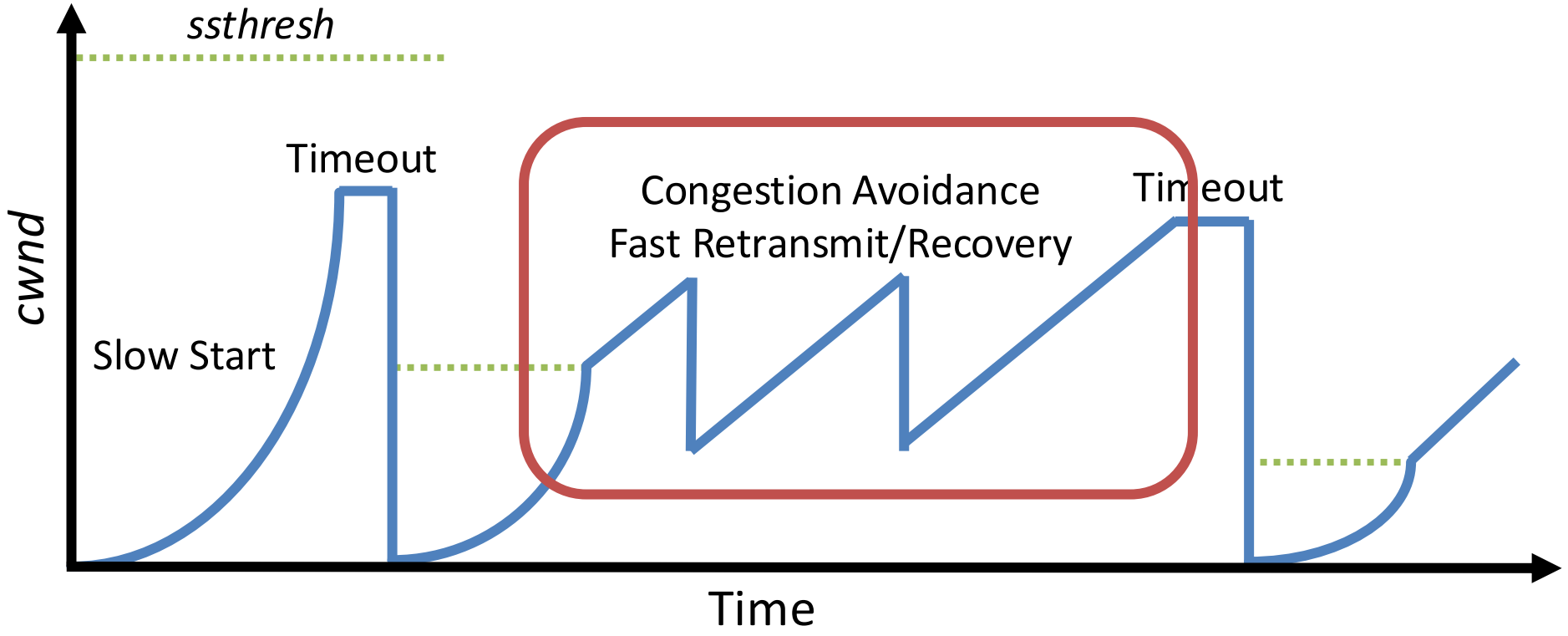
A: when `cwnd` gets to 1/2 of its value before timeout.




## Implementation:

- variable `ssthresh`
- on loss event, `ssthresh` is set to 1/2 of `cwnd` just before loss event

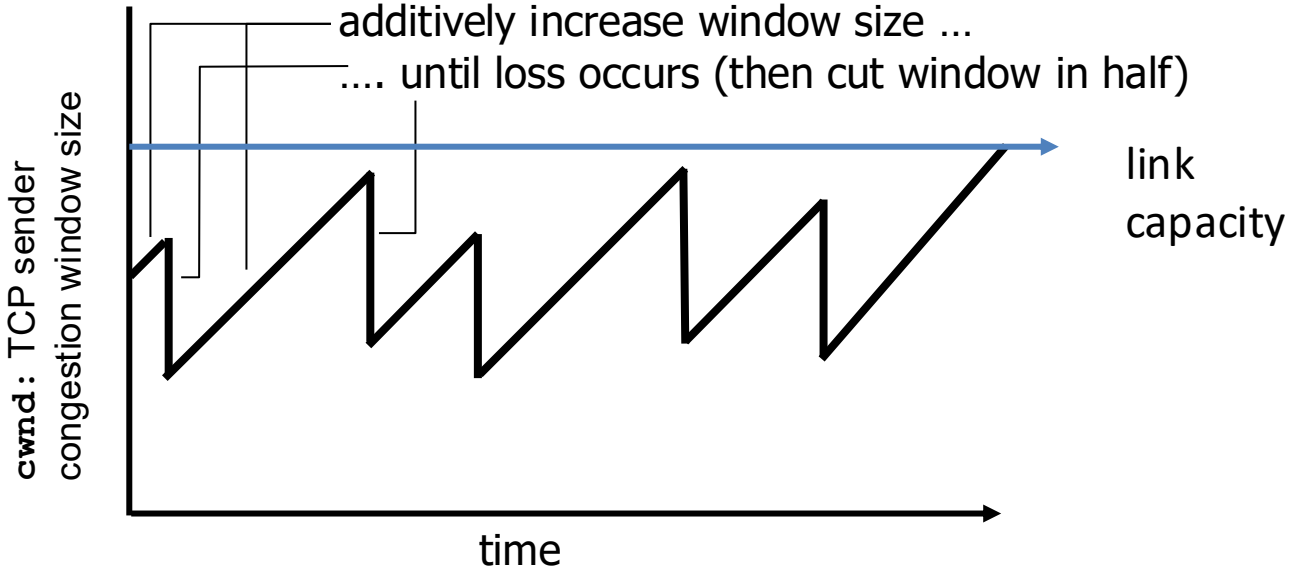
# Fast Retransmit and Fast Recovery



# Additive Increase, Multiplicative Decrease (AIMD)

- approach*:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - additive increase*:* increase **cwnd** by 1 MSS (Maximum Segment Size) every RTT until loss detected
  - multiplicative decrease: cut **cwnd** in half after loss

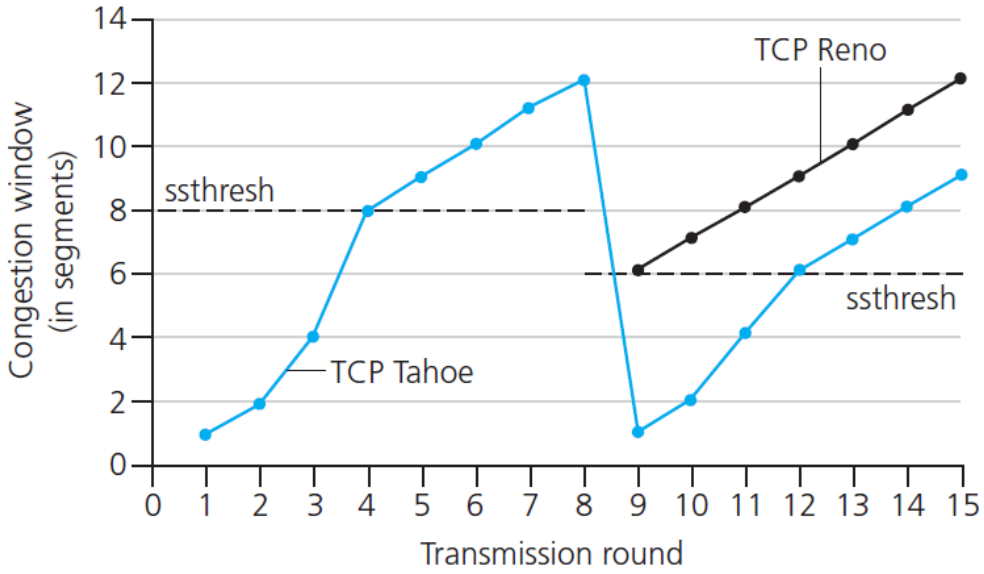
AIMD saw tooth behavior: probing for bandwidth



# TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

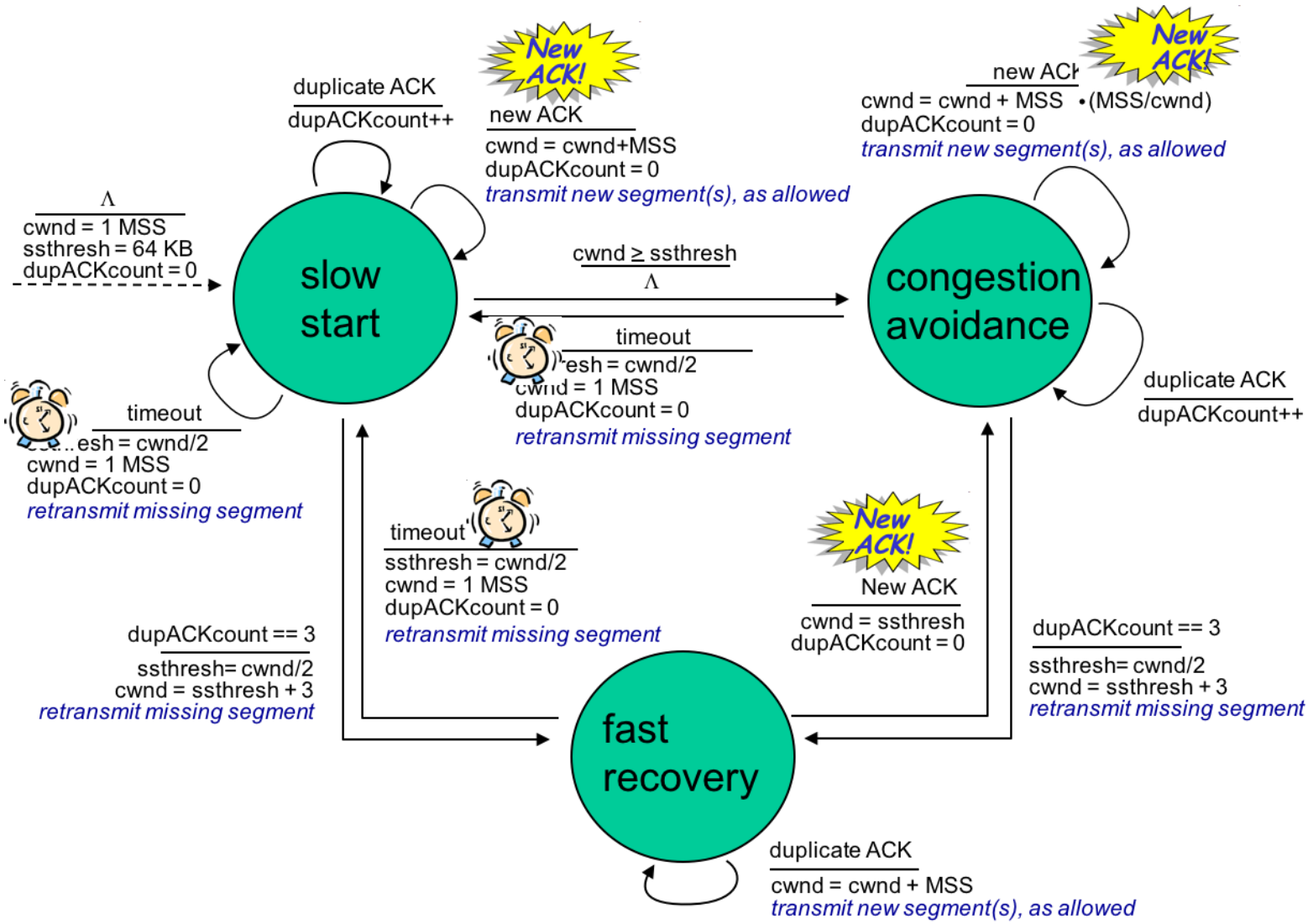
A: when **`cwnd`** gets to 1/2 of its value before timeout.




## Implementation:

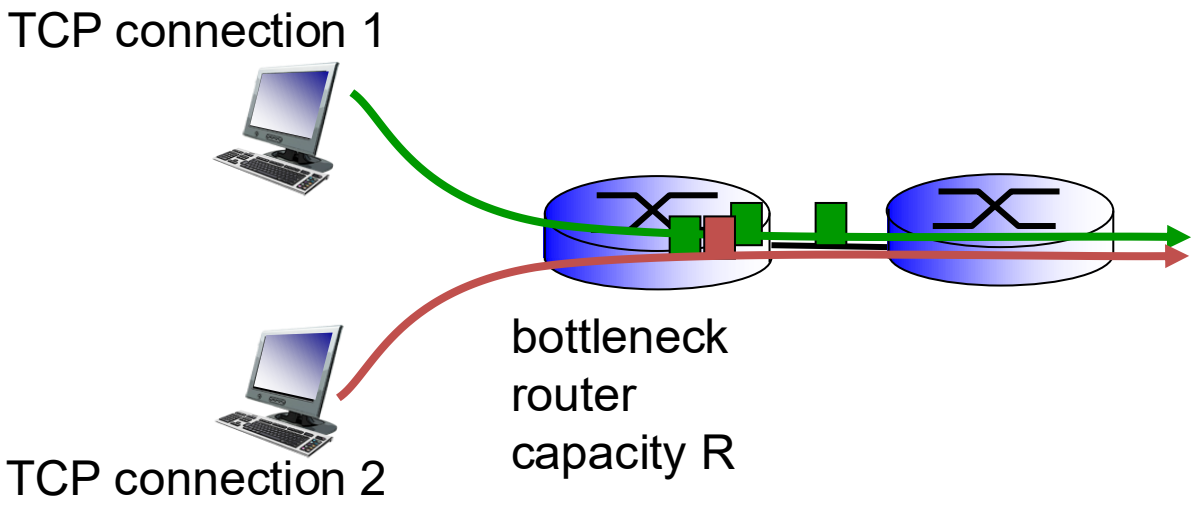
- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

# TCP Reno Congestion Control



# TCP Fairness

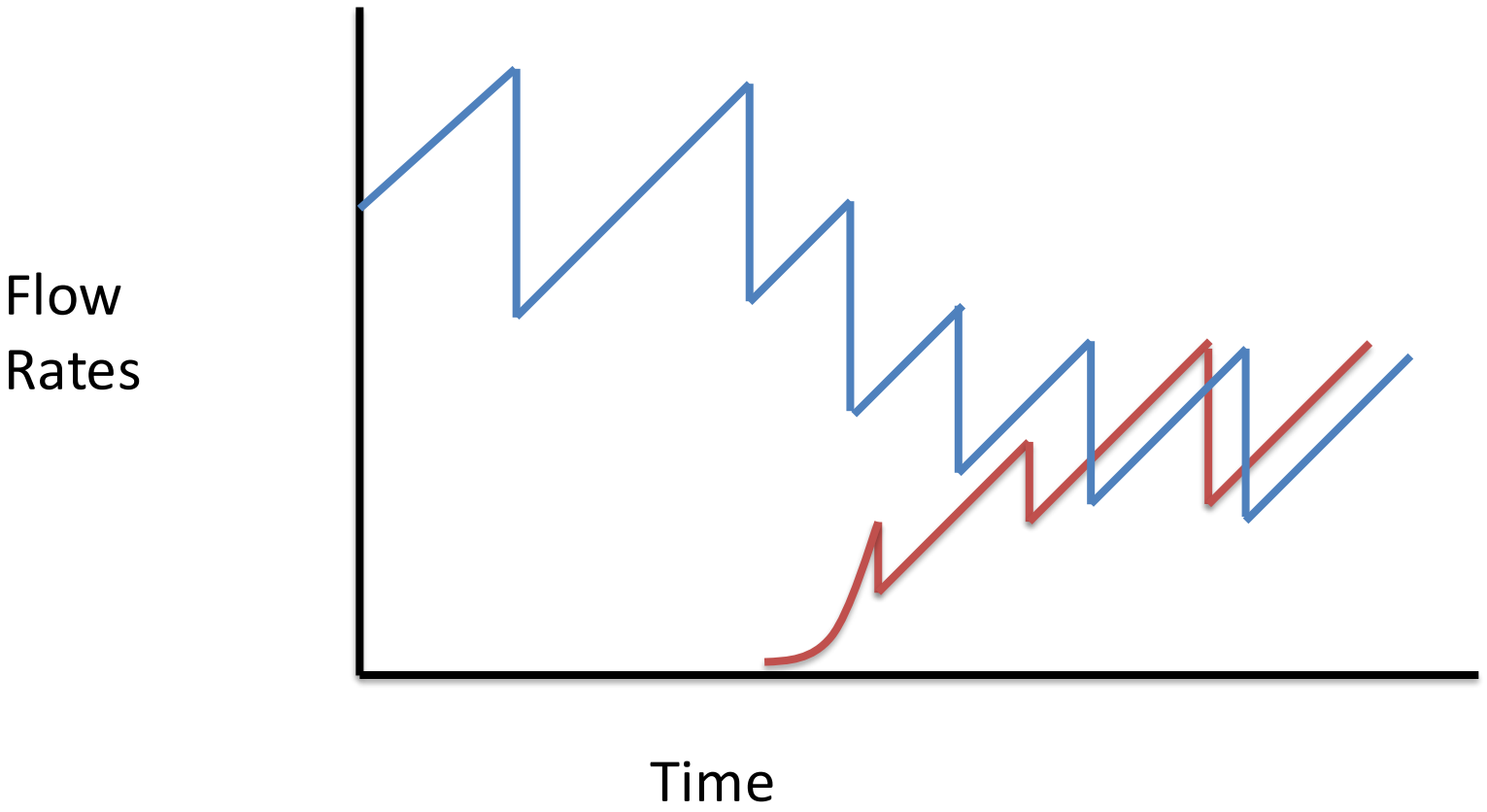
*fairness goal:* if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



# TCP Fairness

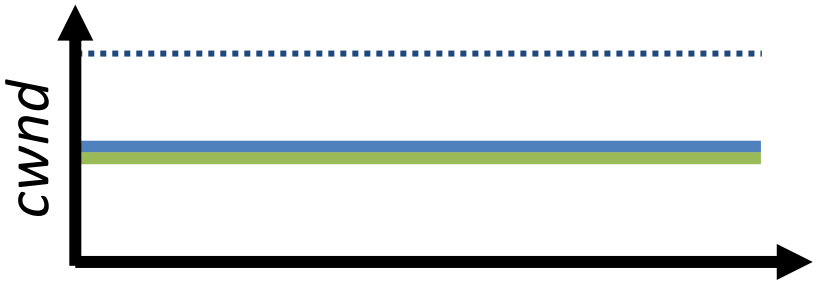
Two competing sessions:

- additive increase gives slope of 1, as throughput increases
- multiplicative decrease, decreases throughput proportionally

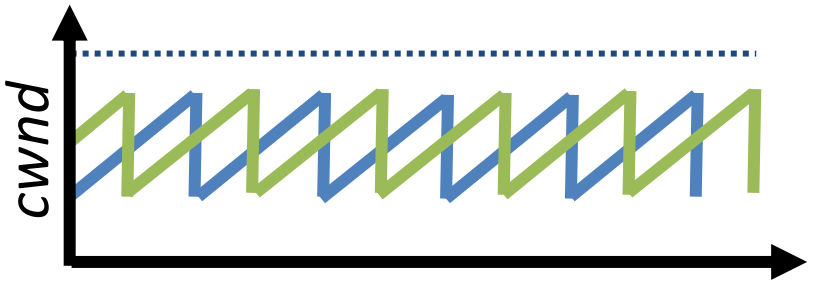


# Synchronization of Flows

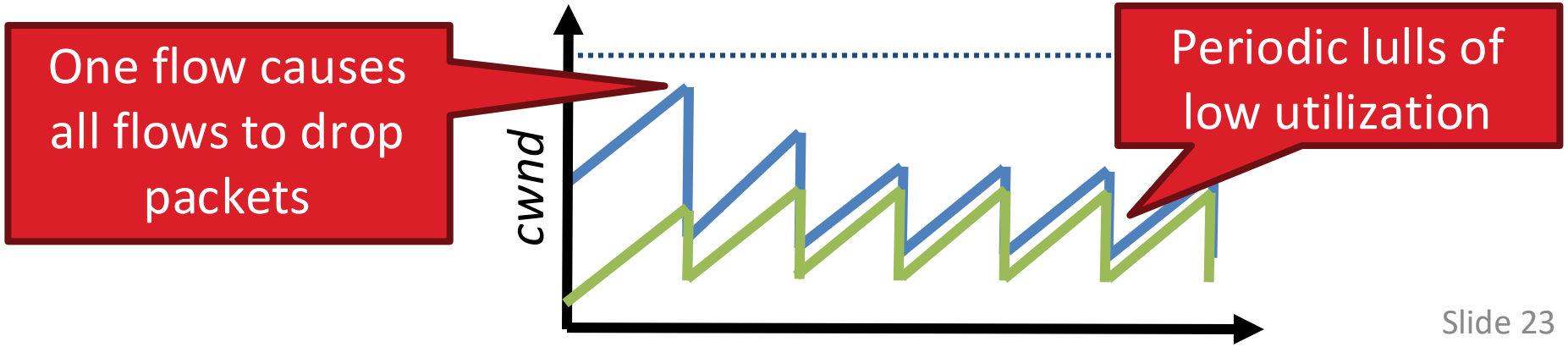
- Ideal bandwidth sharing



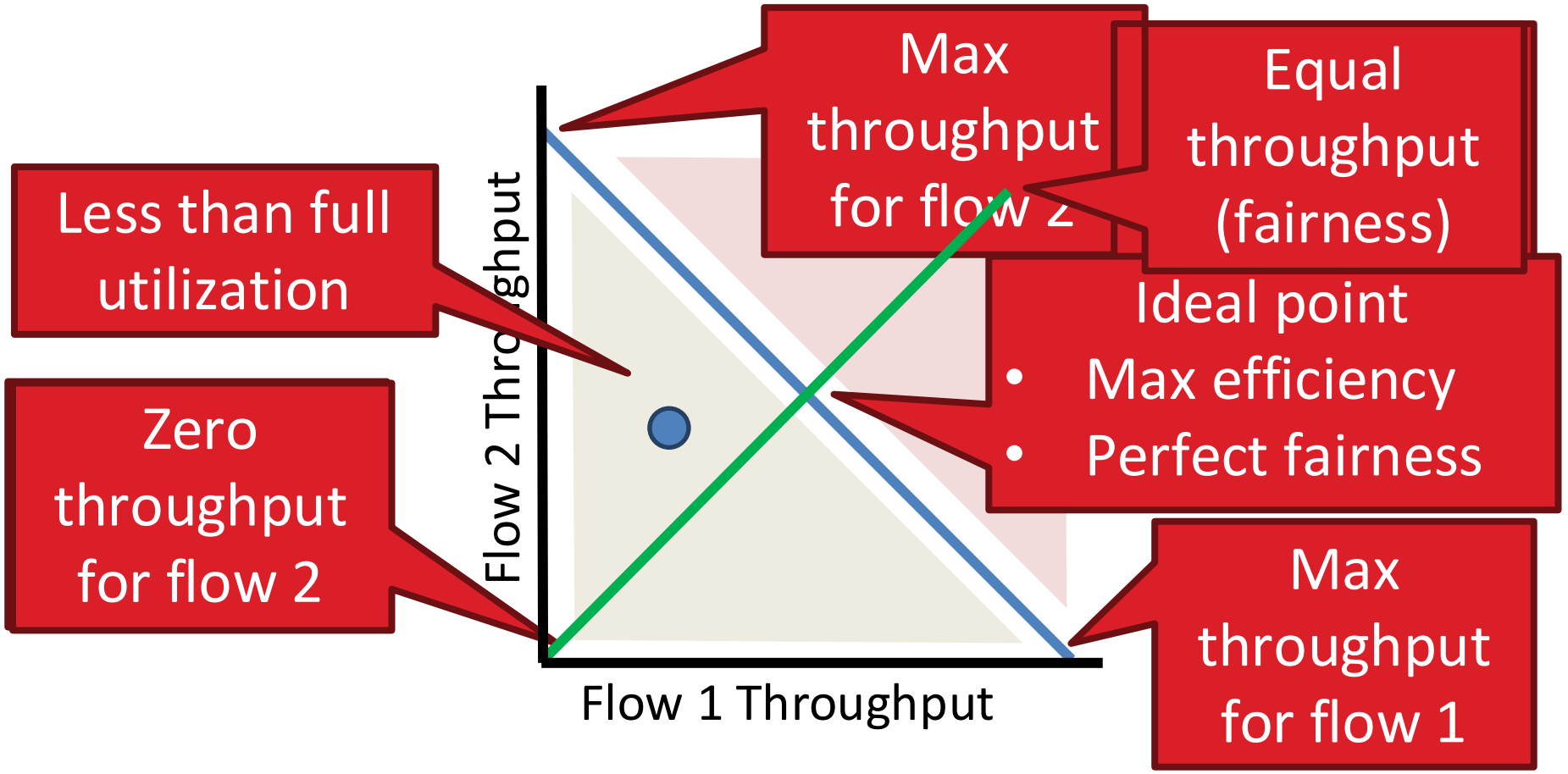

- Oscillating, but high overall utilization




- In reality, flows synchronize

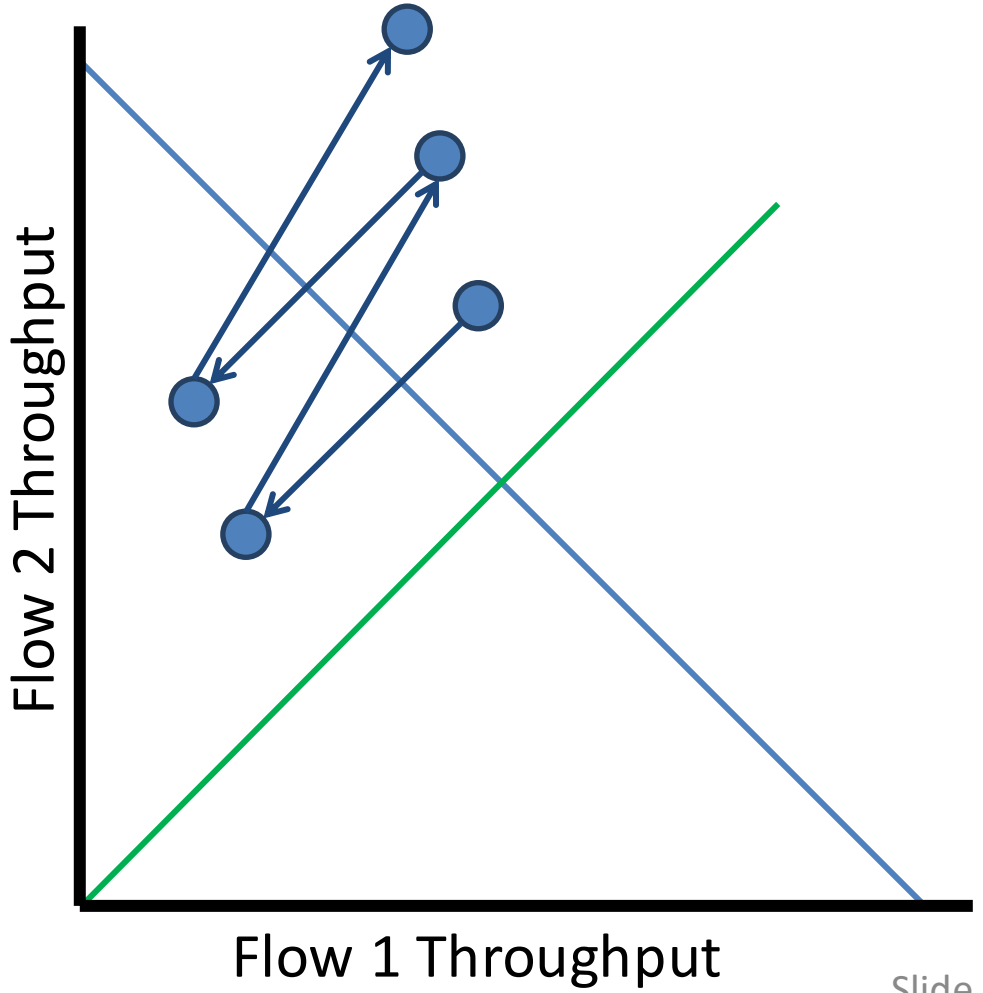


# Utilization and Fairness



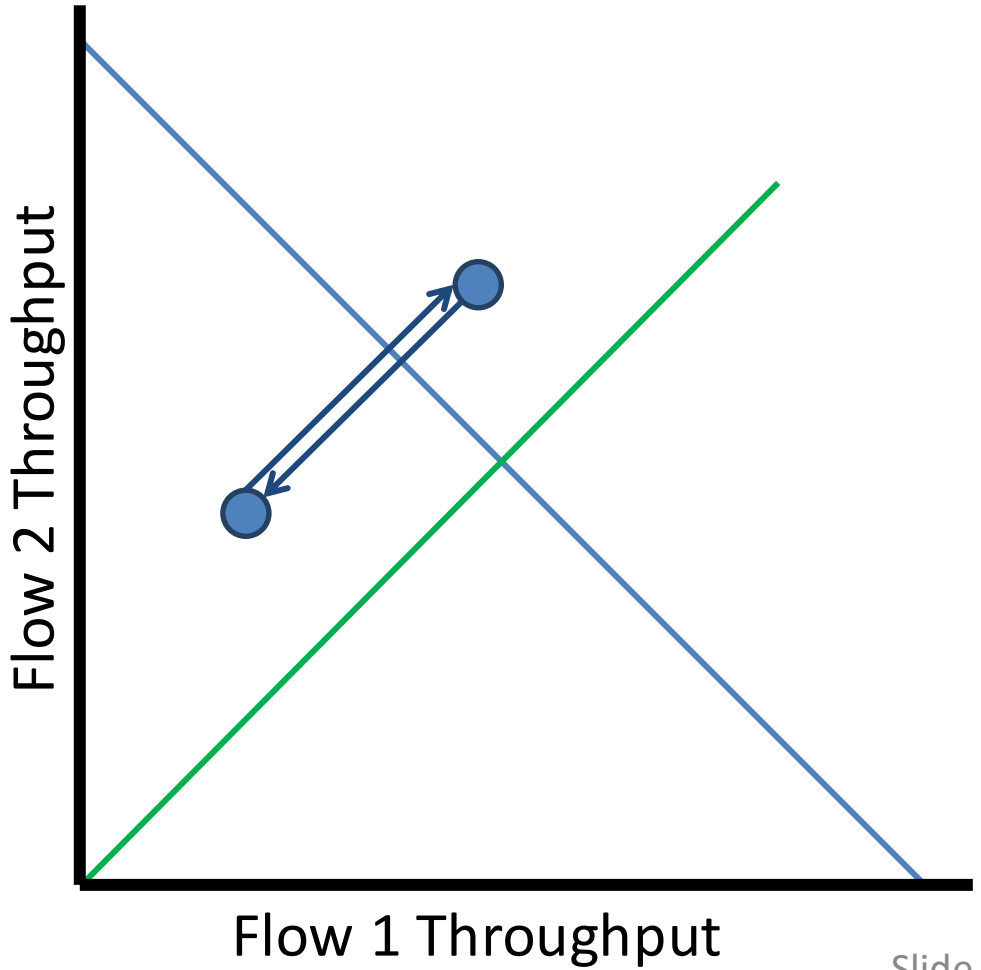
# Multiplicative Increase, Additive Decrease

- Not stable!
- Veers away from fairness



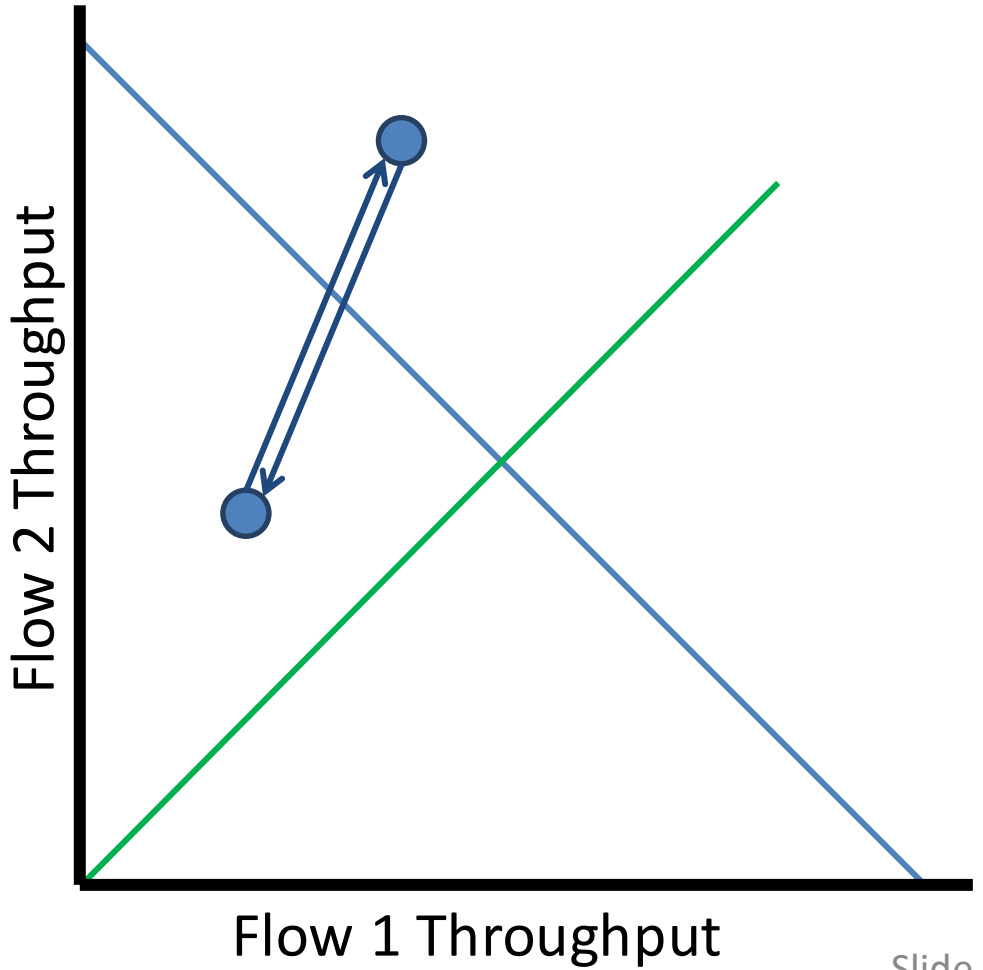
# Additive Increase, Additive Decrease

- Stable
- But does not converge to fairness



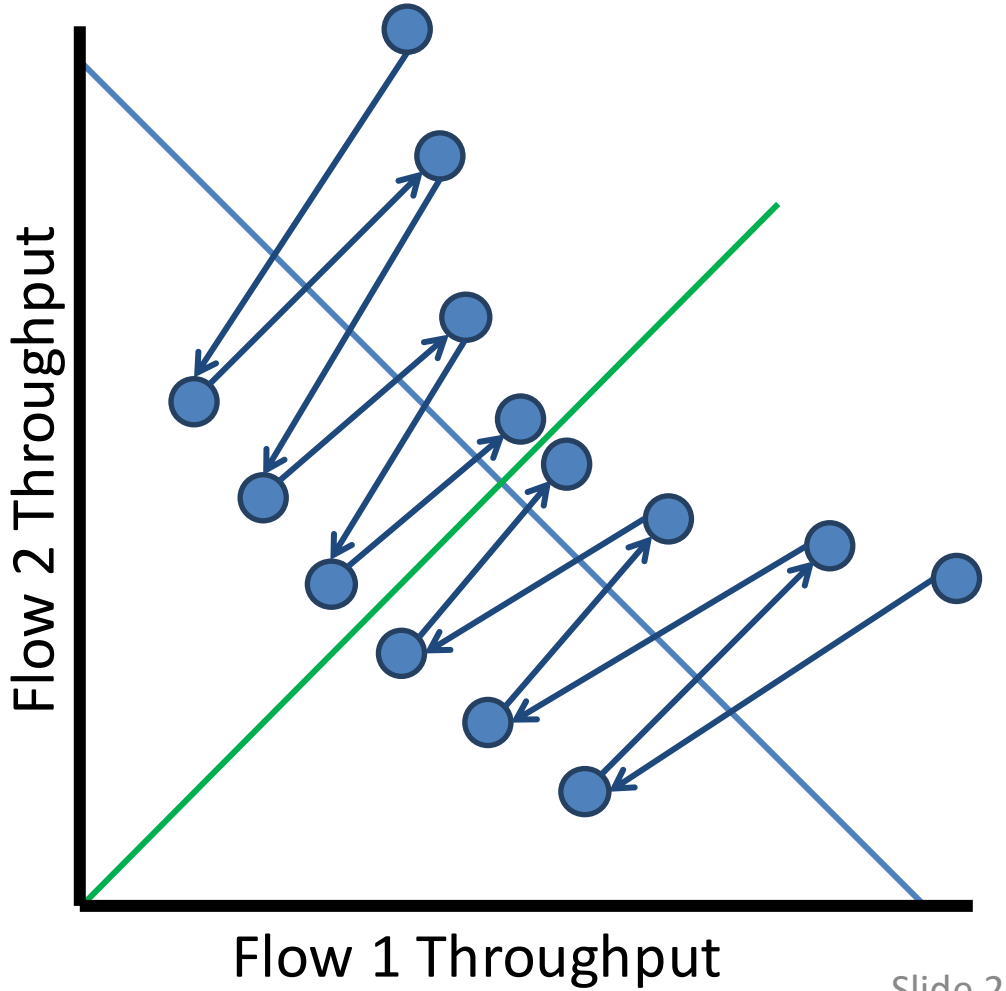
# Multiplicative Increase, Multiplicative Decrease

- Stable
- But does not converge to fairness



# Additive Increase, Multiplicative Decrease

- Converges to stable and fair cycle
- Symmetric around *y*=*x*



# Since TCP is fair, does this mean we no longer have to worry about bandwidth hogging?

A. Yep, solved it!

B. No, we can still game the system.

If you wanted to cheat to get extra traffic through, how might you do it?

# Fairness (more)

## Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

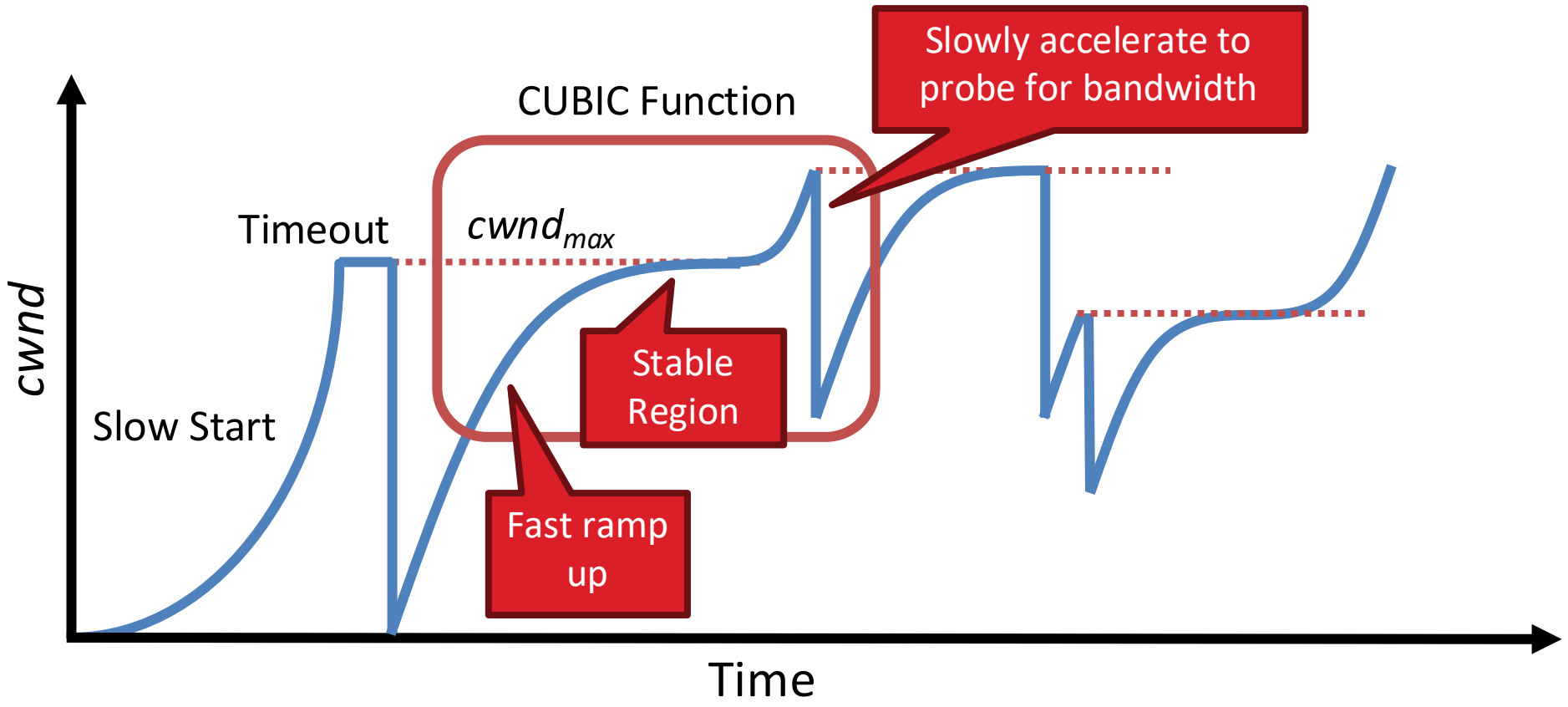
## Fairness, parallel TCP connections

- Application can open multiple parallel connections between two hosts
- Web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

# TCP Variants

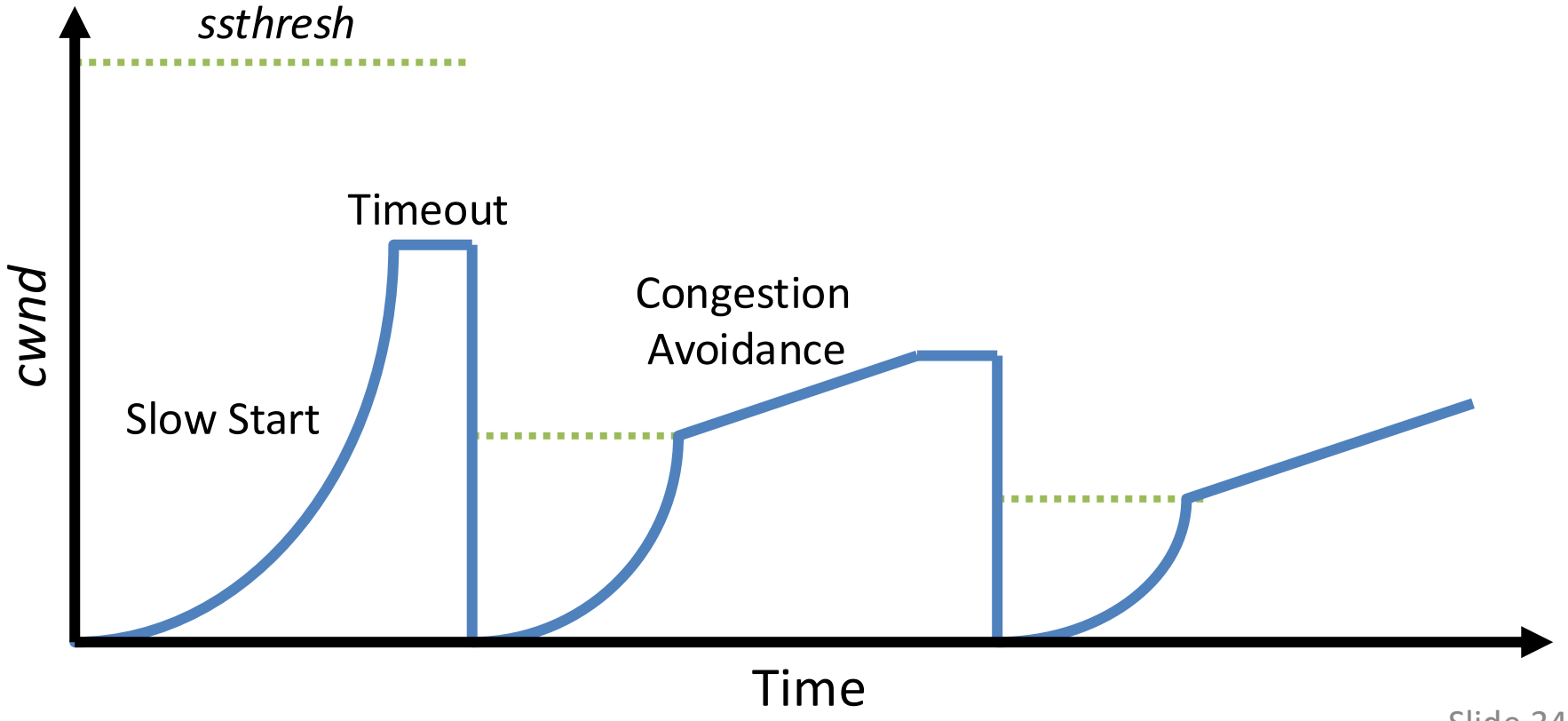
- There are tons of them!
- Tahoe, Reno, New Reno, Vegas, Hybla, BIC, CUBIC, Westwood, Compound TCP, DCTCP, YeAH-TCP, …
- Each tweaks and adjusts the response to congestion.
- Why not just find a cwnd value that works, and stick with it?

# TCP CUBIC Example




- Less wasted bandwidth due to fast ramp up
- Stable region and slow acceleration help maintain fairness
  - Fast ramp up is more aggressive than additive increase
  - To be fair to Tahoe/Reno, CUBIC needs to be less aggressive

# TCP: Big Picture



# Summary

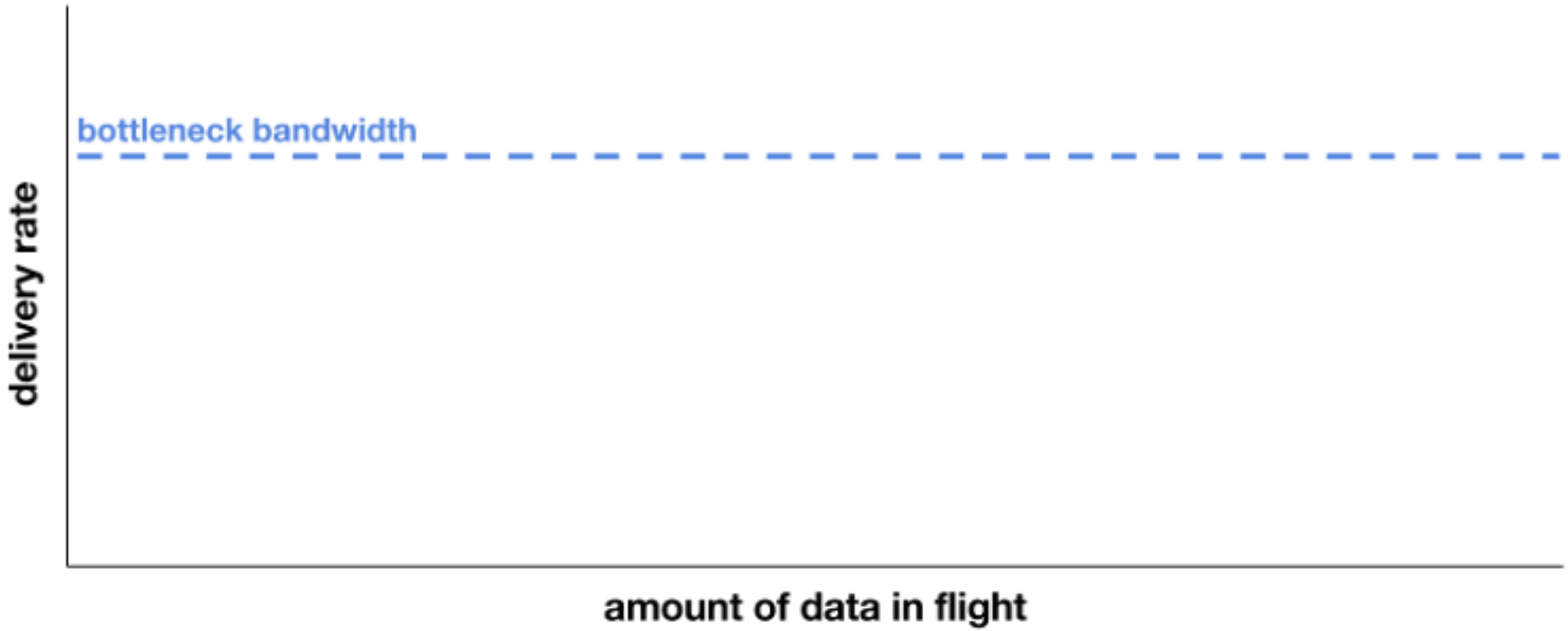
- TCP has mechanisms to control sending rate:
  - Flow control: don’t overload receiver
  - Congestion control: don’t overload network

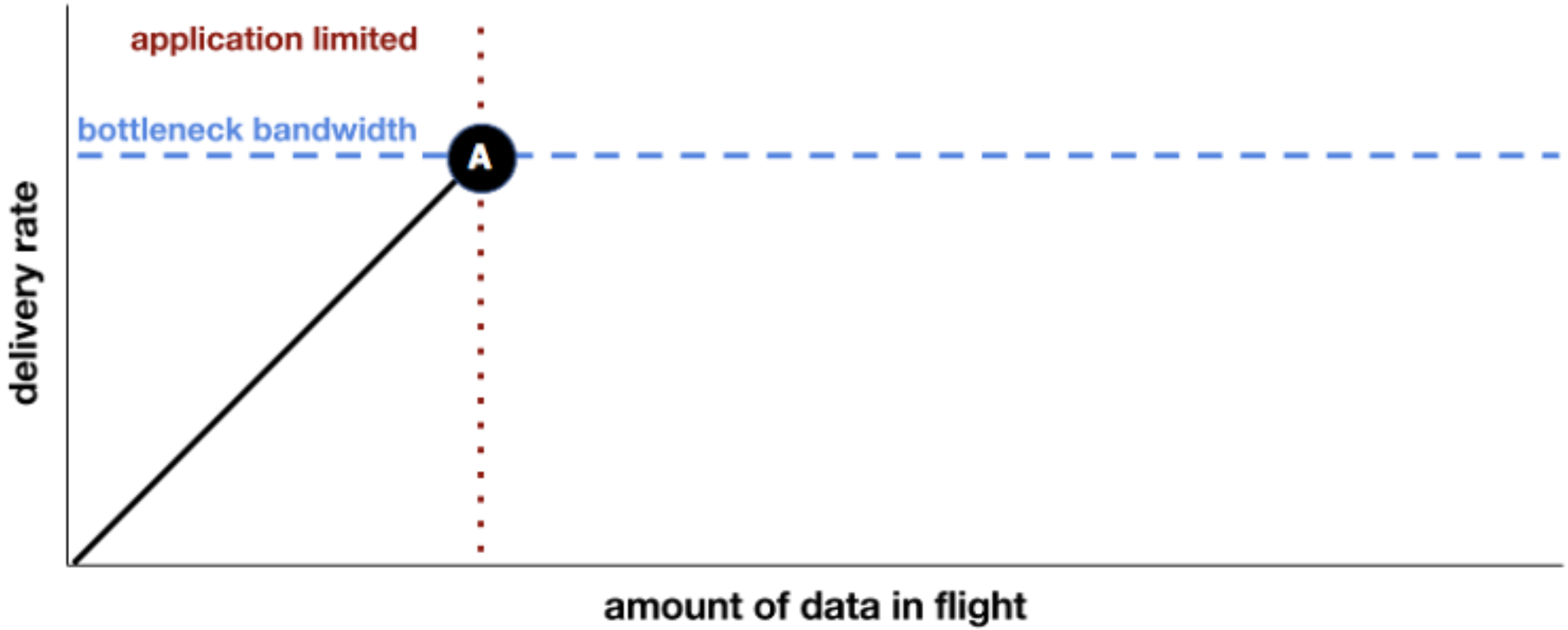
- min(rwnd, cwnd) determines window size for TCP segment pipelining (typically cwnd)

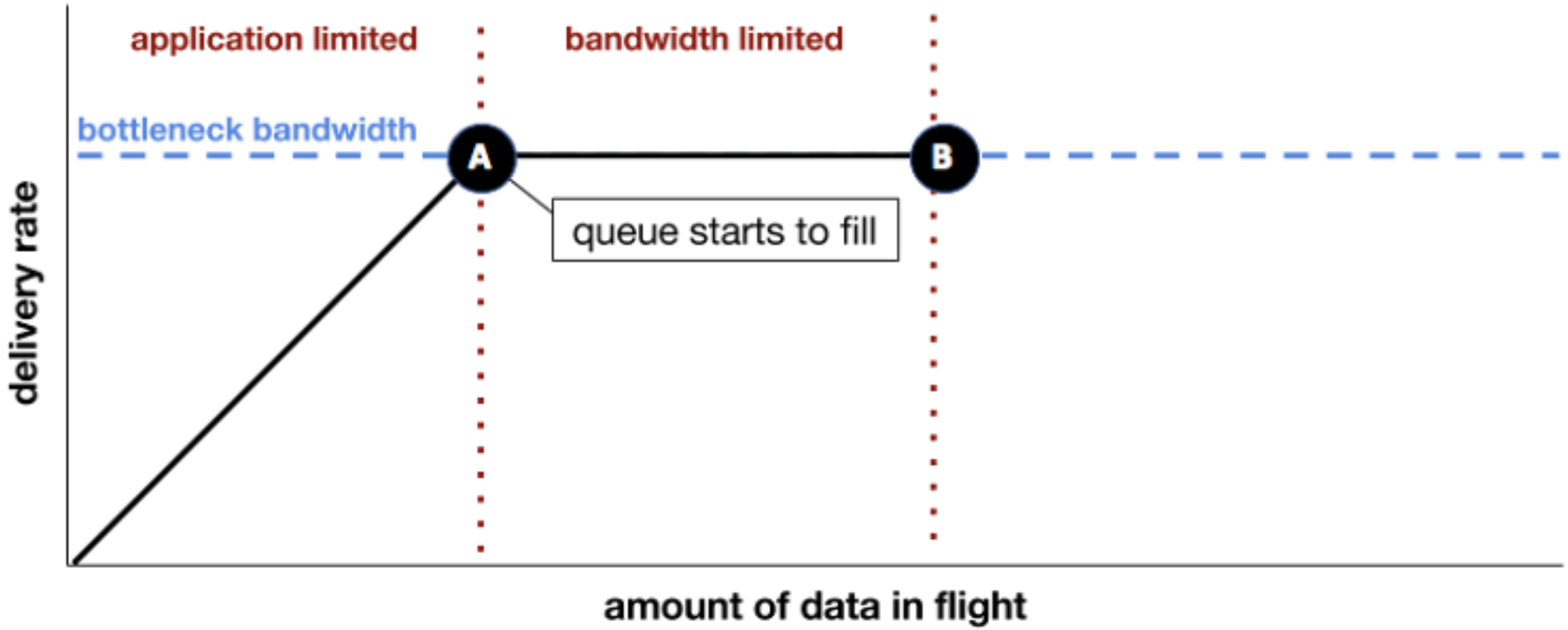
- AIMD: additive increase, multiplicative decrease
- AIMD ensures fairness

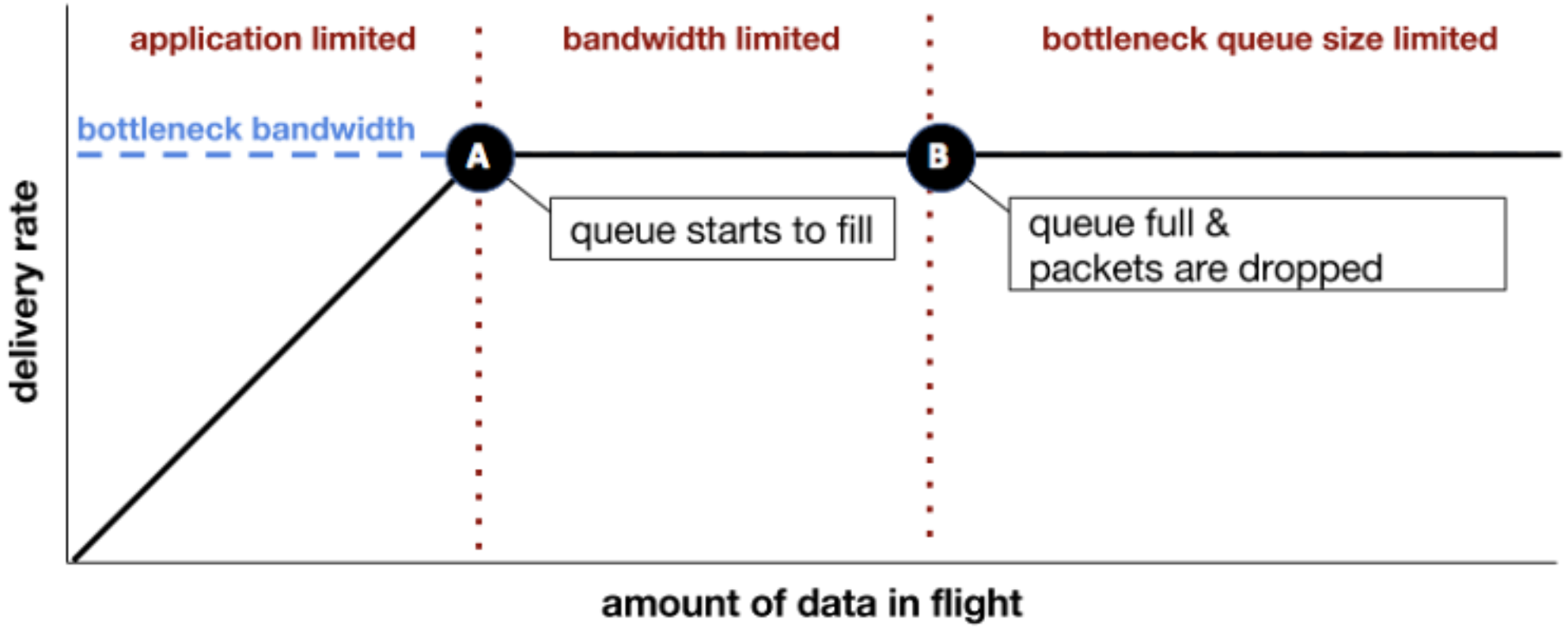
# Additional Slides

(not assessable)


bottleneck bandwidth
delivery rate
amount of data in flight

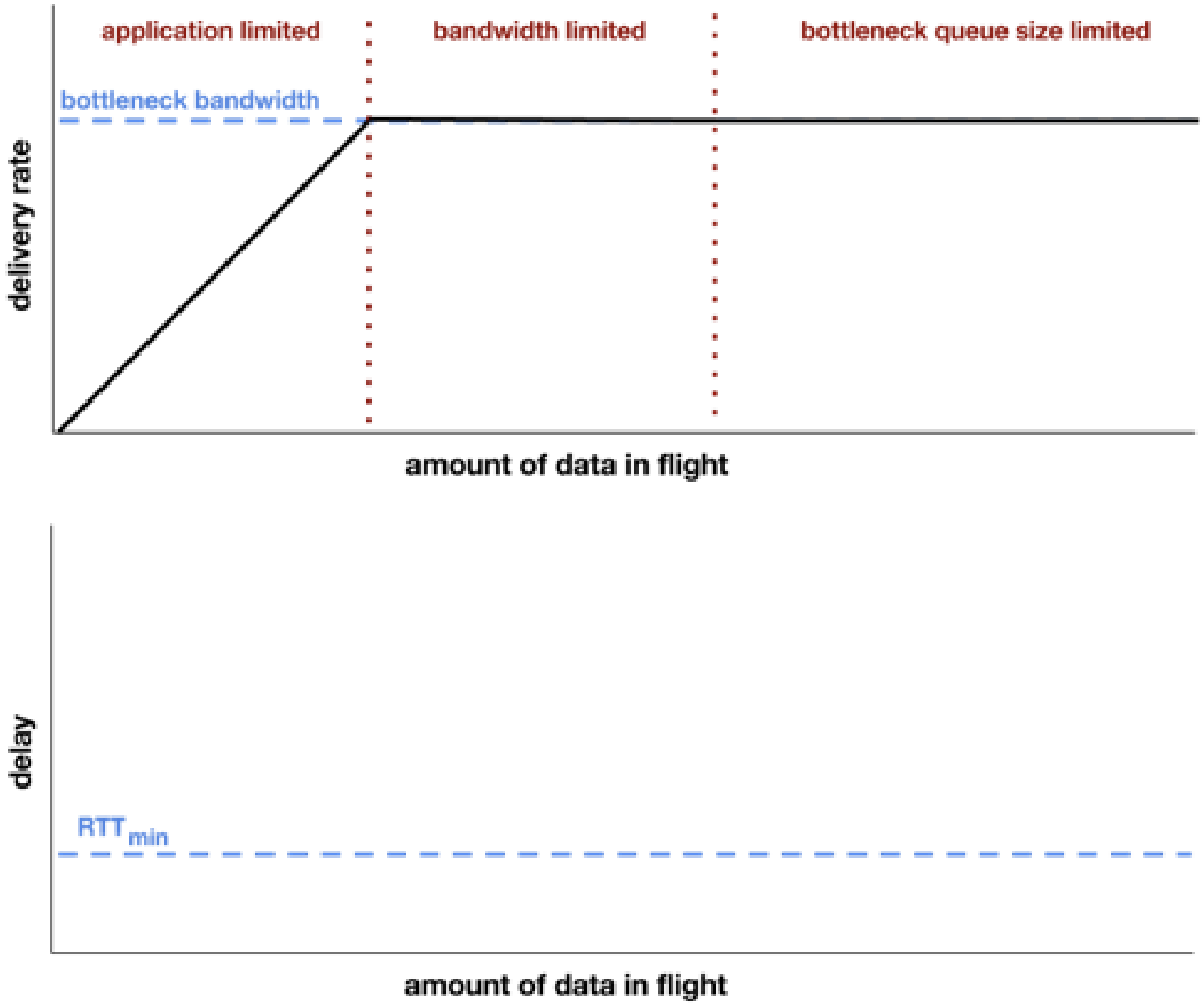

application limited
bottleneck bandwidth
A
delivery rate
amount of data in flight


application limited
bandwidth limited
bottleneck bandwidth
A
B
queue starts to fill
delivery rate
amount of data in flight

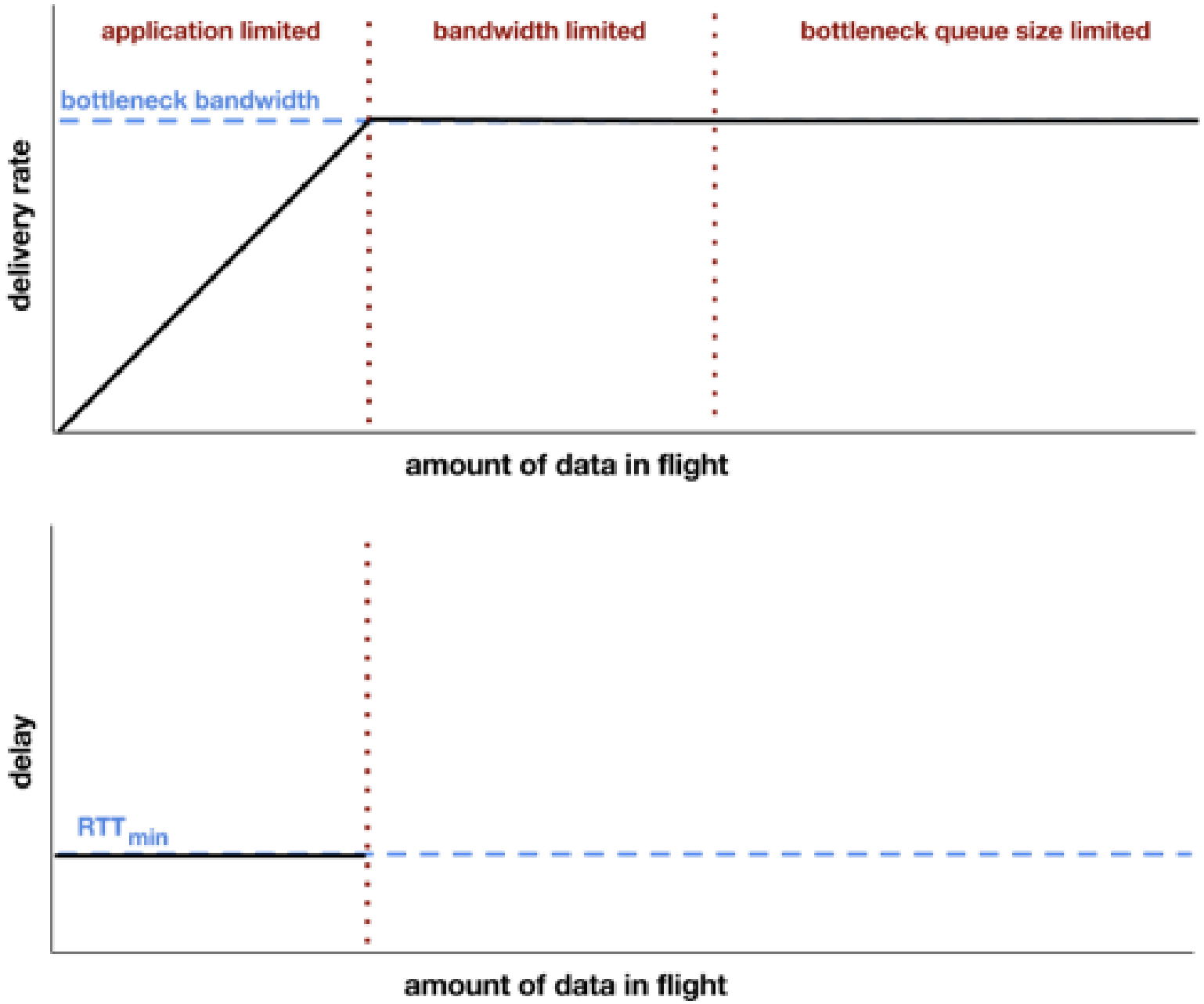



Filling the bottleneck queue, let's a flow fully utilize the available bottleneck bandwidth.

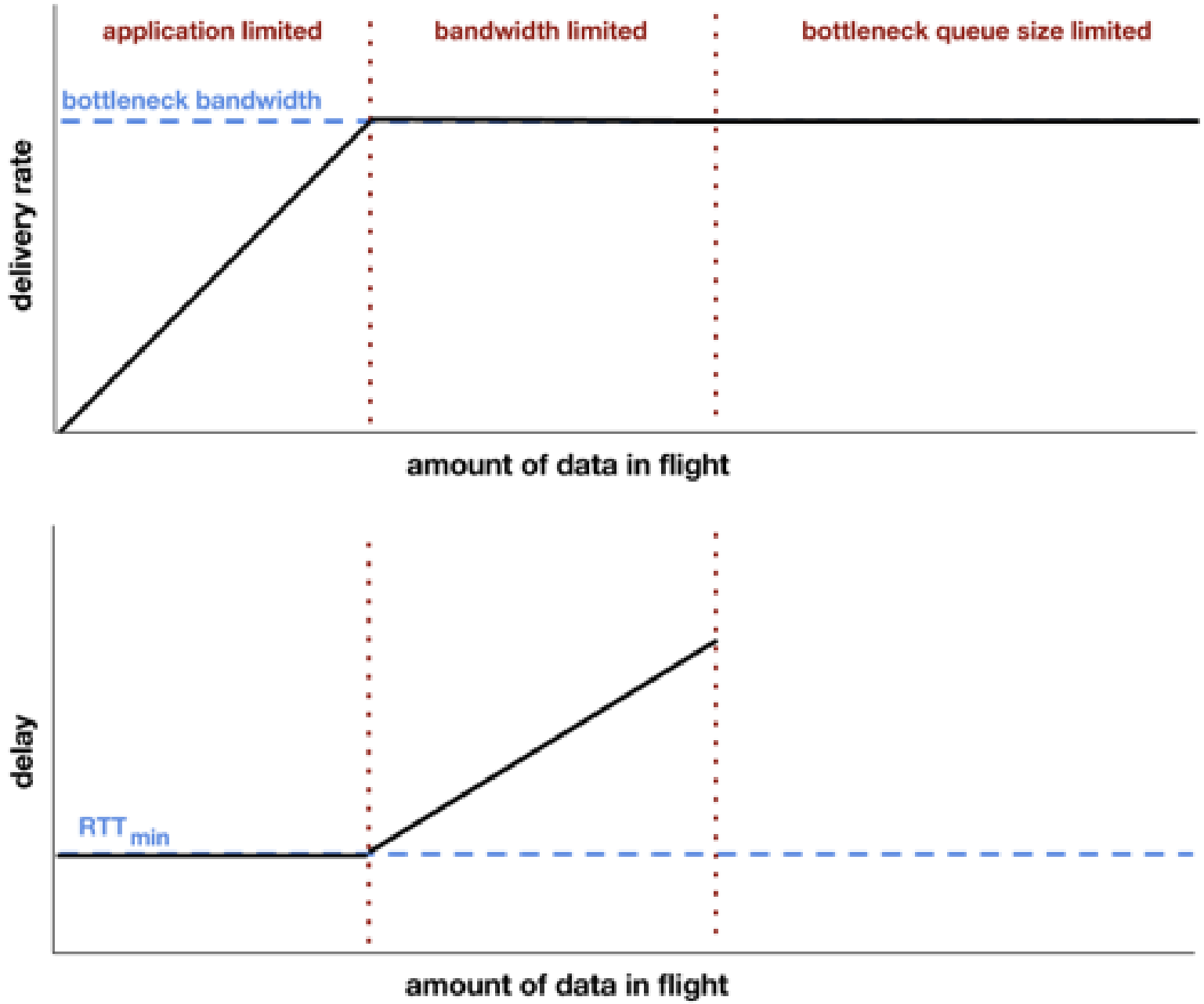
But what happens to delay when the bottleneck queue is full?



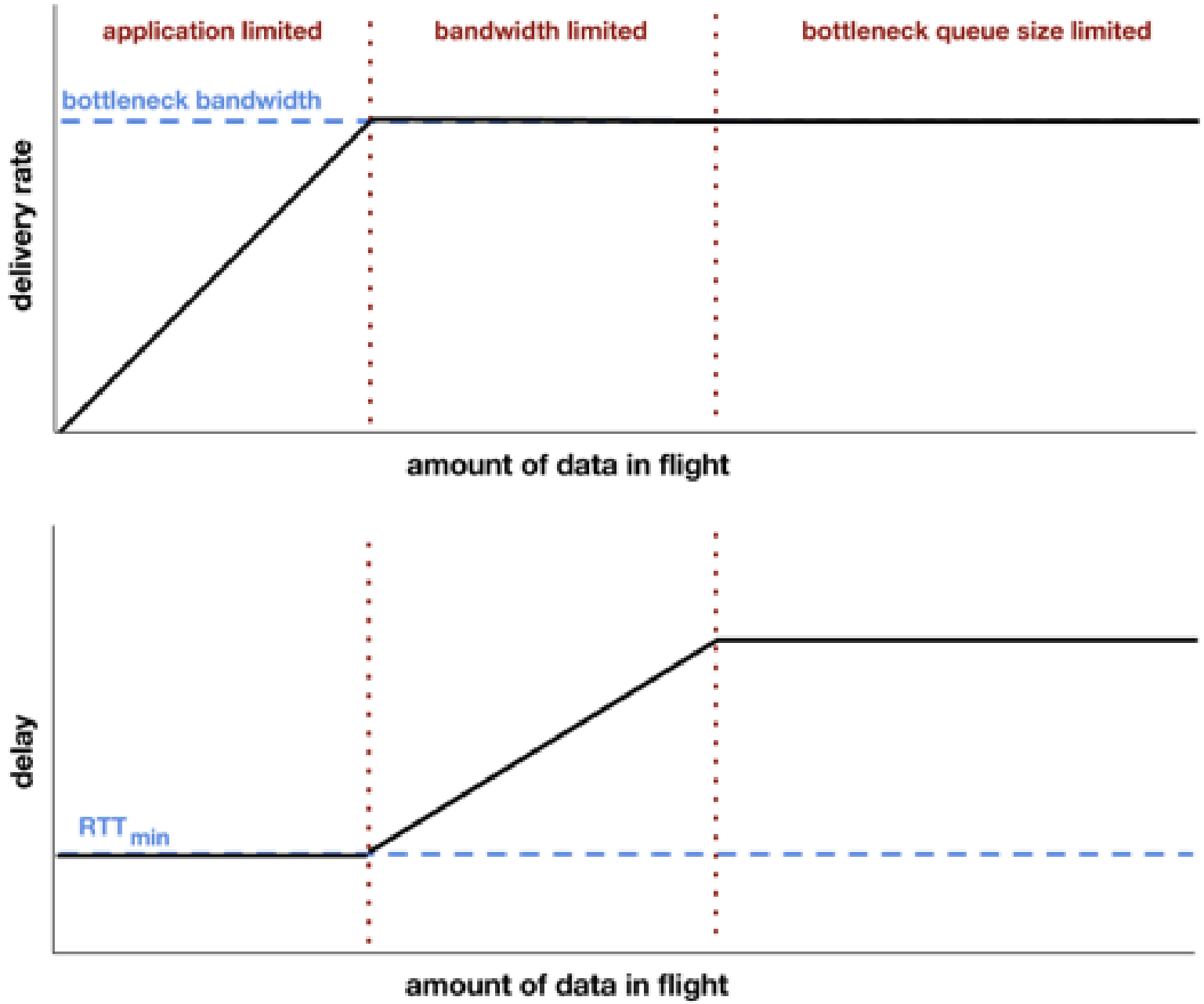
But what happens to delay when the bottleneck queue is full?



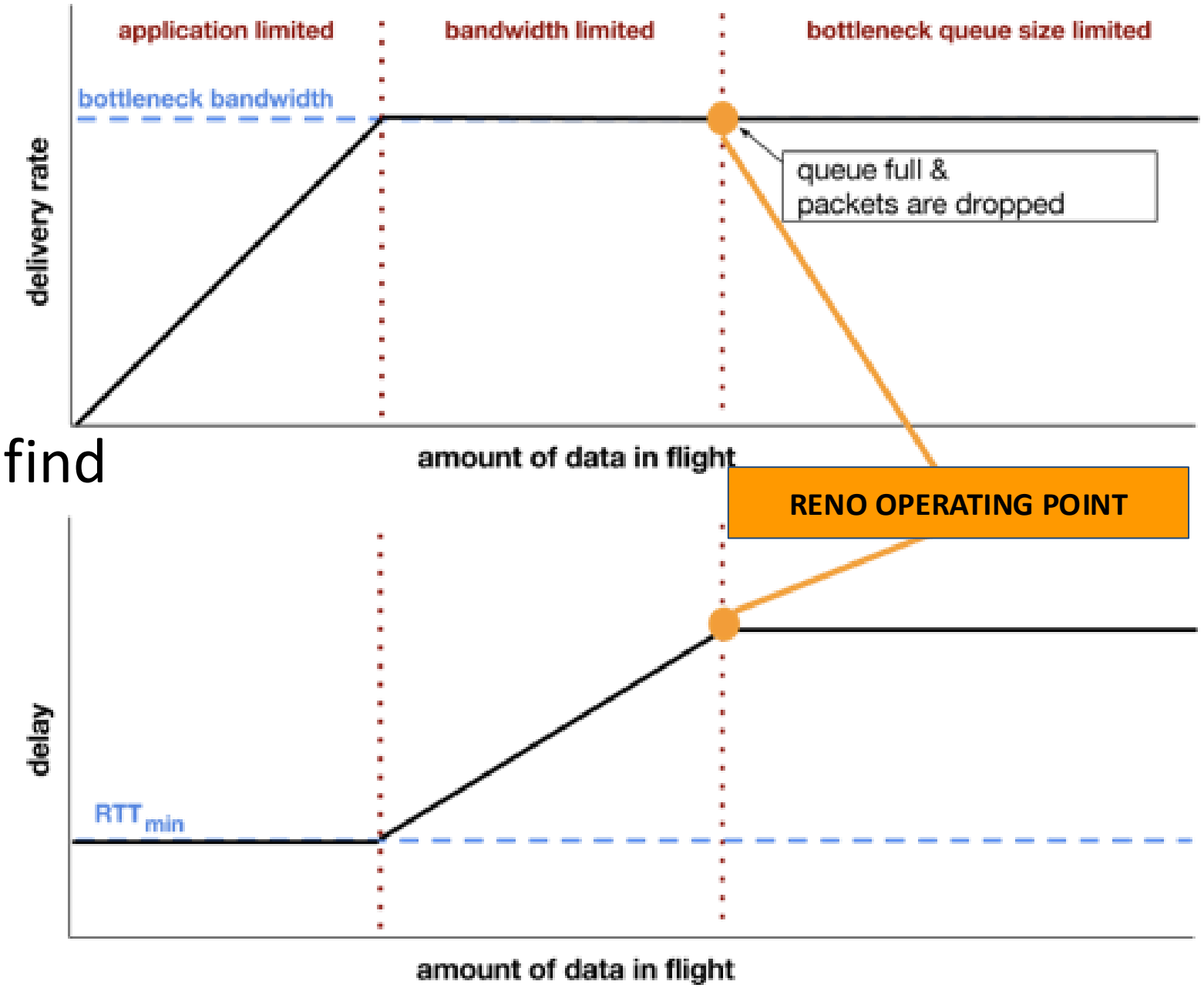
But what happens to delay when the bottleneck queue is full?



But what happens to delay when the bottleneck queue is full?



TCP Reno fills the bottleneck queue to find BDP causing large queueing delays. (bufferbloat)



In 1976, Leonard Kleinrock showed optimal operating point for a CCA is maximal throughput and minimal delay.

