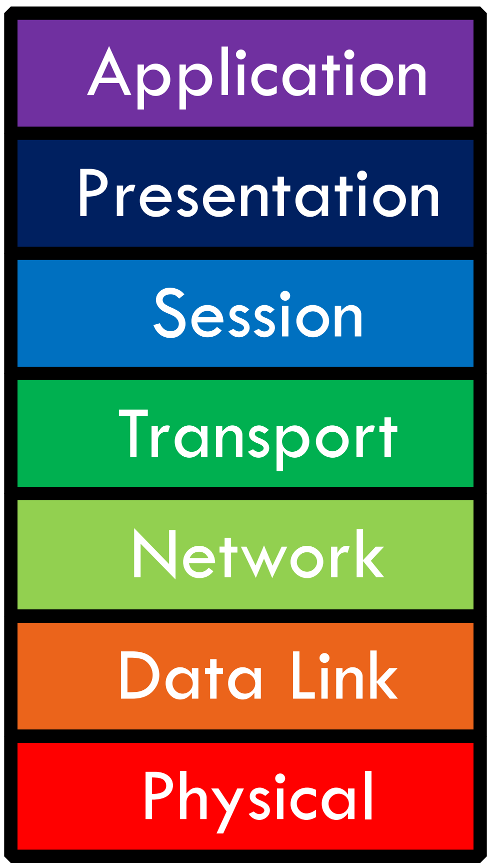
CS 3700

Networks and Distributed Systems

Lecture 10: Congestion Control

Revised 2/9/2014

Transport Layer



- Function:
  - Demultiplexing of data streams
- Optional functions:
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control
- Key challenges:
  - Detecting and responding to congestion
  - Balancing fairness against high utilization

Outline

- Congestion Control
- Evolution of TCP
- Problems with TCP

What is Congestion?

- Load on the network is higher than capacity

What is Congestion?

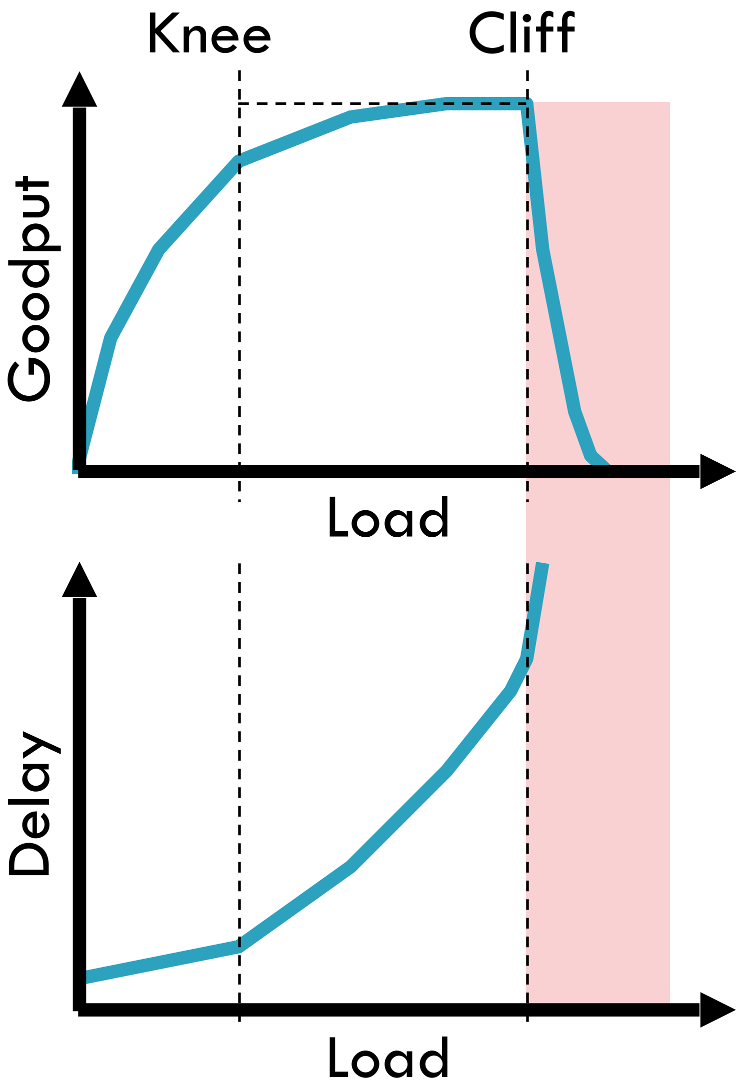
- Load on the network is higher than capacity
  - Capacity is not uniform across networks
    - Modem vs. Cellular vs. Cable vs. Fiber Optics
  - There are multiple flows competing for bandwidth
    - Residential cable modem vs. corporate datacenter
  - Load is not uniform over time
    - 10pm, Sunday night = Bittorrent Game of Thrones

Why is Congestion Bad?

- Results in packet loss
  - Routers have finite buffers
  - Internet traffic is self similar, no buffer can prevent all drops
  - When routers get overloaded, packets will be dropped
- Practical consequences
  - Router queues build up, delay increases
  - Wasted bandwidth from retransmissions
  - Low network goodput

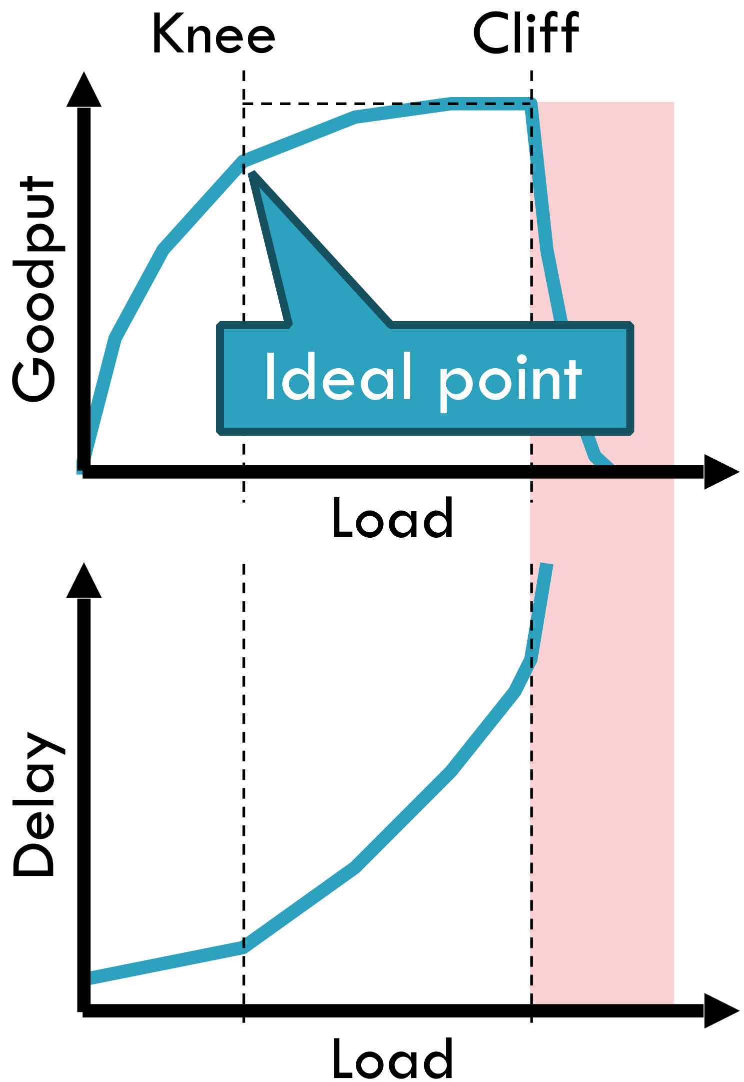
The Danger of Increasing Load

- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- In an M/M/1 queue
  - Delay = 1/(1 – utilization)
- Cliff – point after which
  - Throughput → 0
  - Delay → ∞



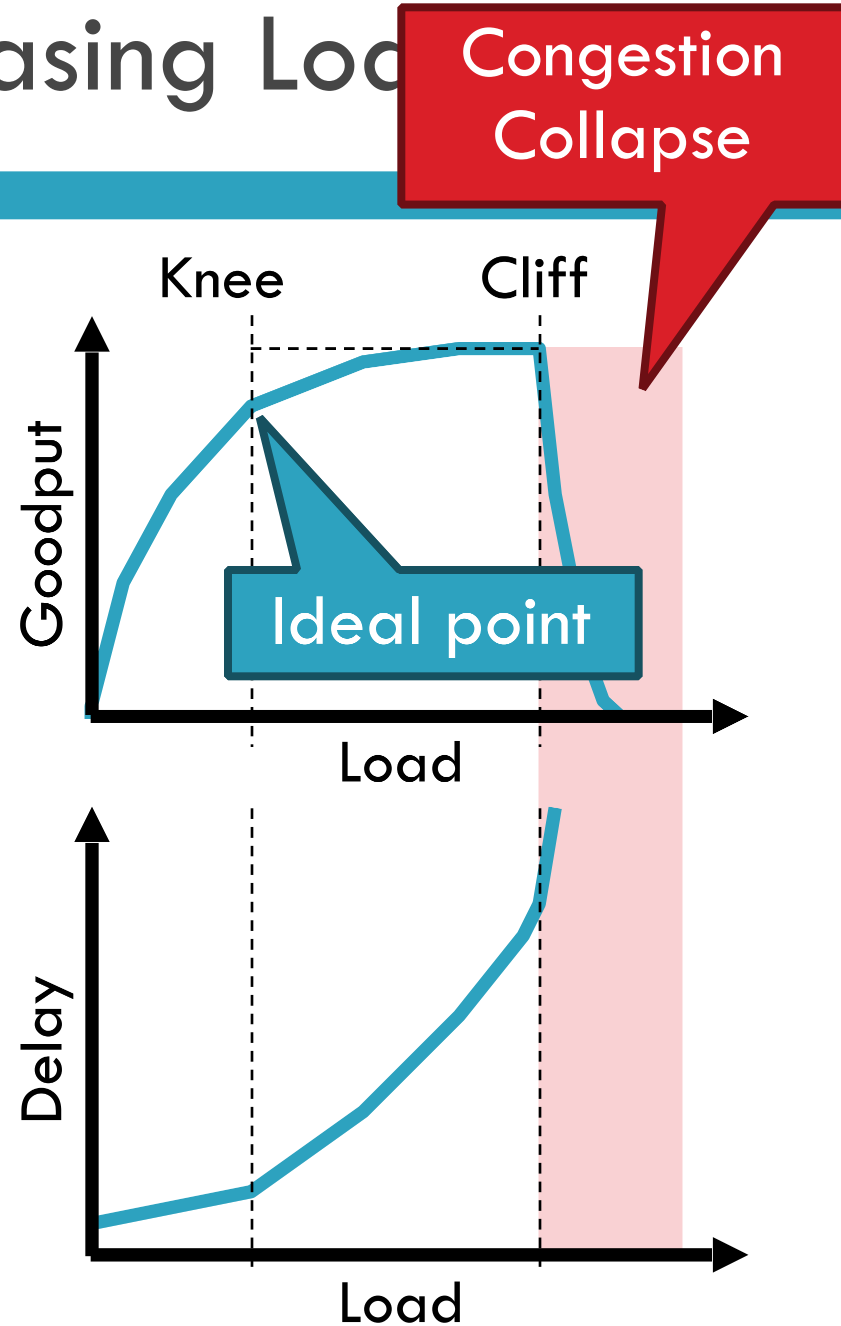
The Danger of Increasing Load

- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- In an M/M/1 queue
  - Delay = 1/(1 – utilization)
- Cliff – point after which
  - Throughput → 0
  - Delay → ∞

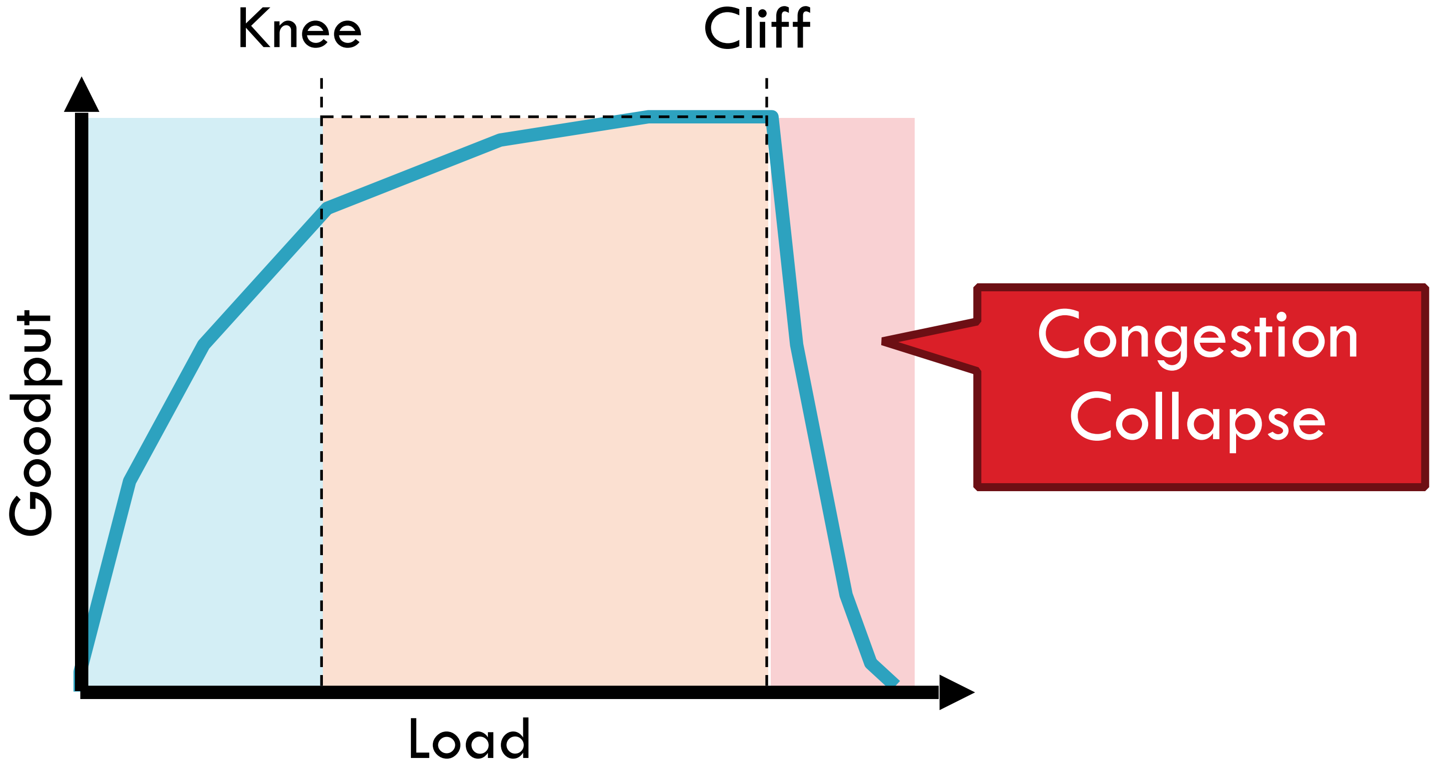


The Danger of Increasing Load

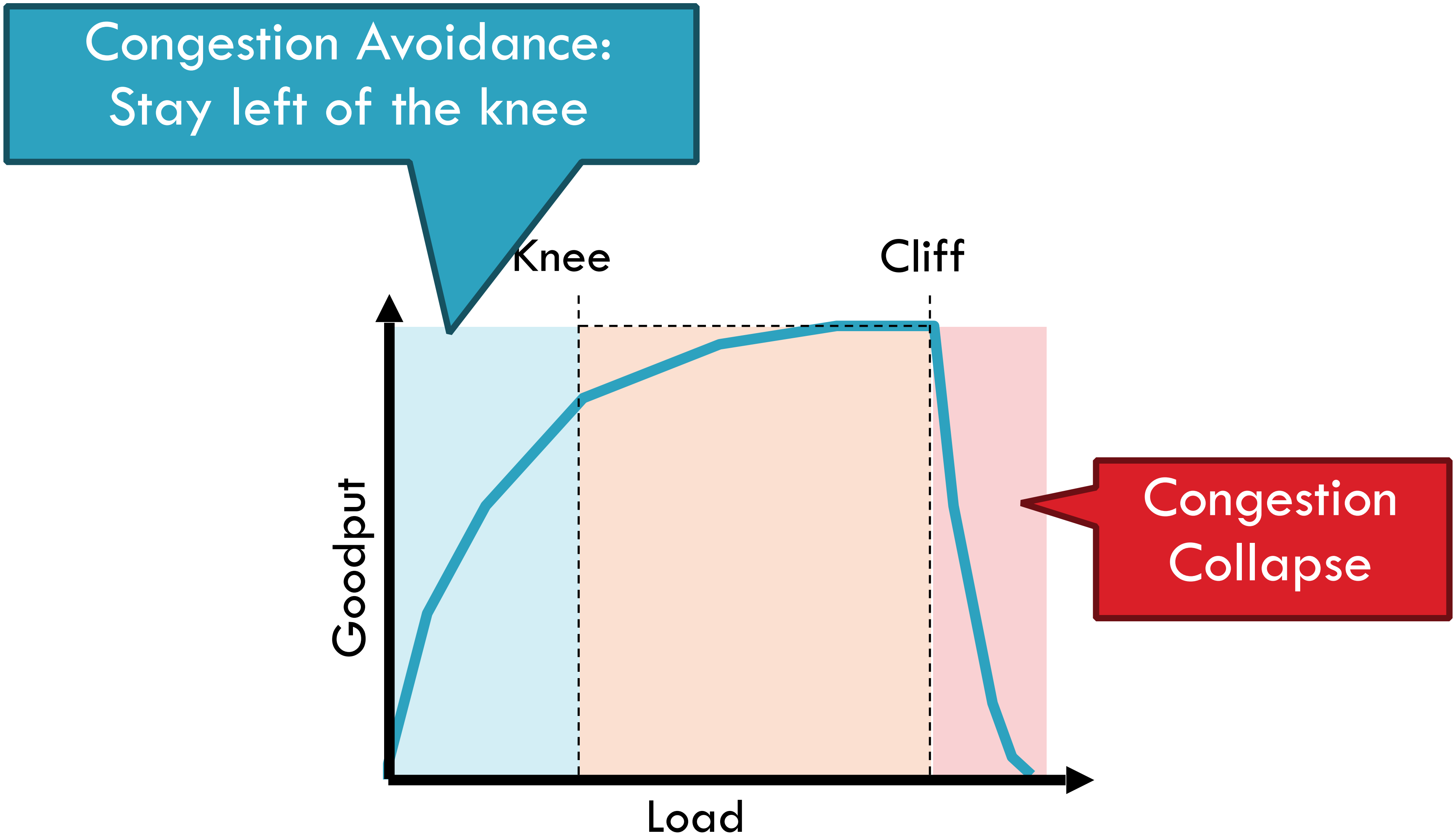
- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- In an M/M/1 queue
  - Delay = 1/(1 – utilization)
- Cliff – point after which
  - Throughput → 0
  - Delay → ∞



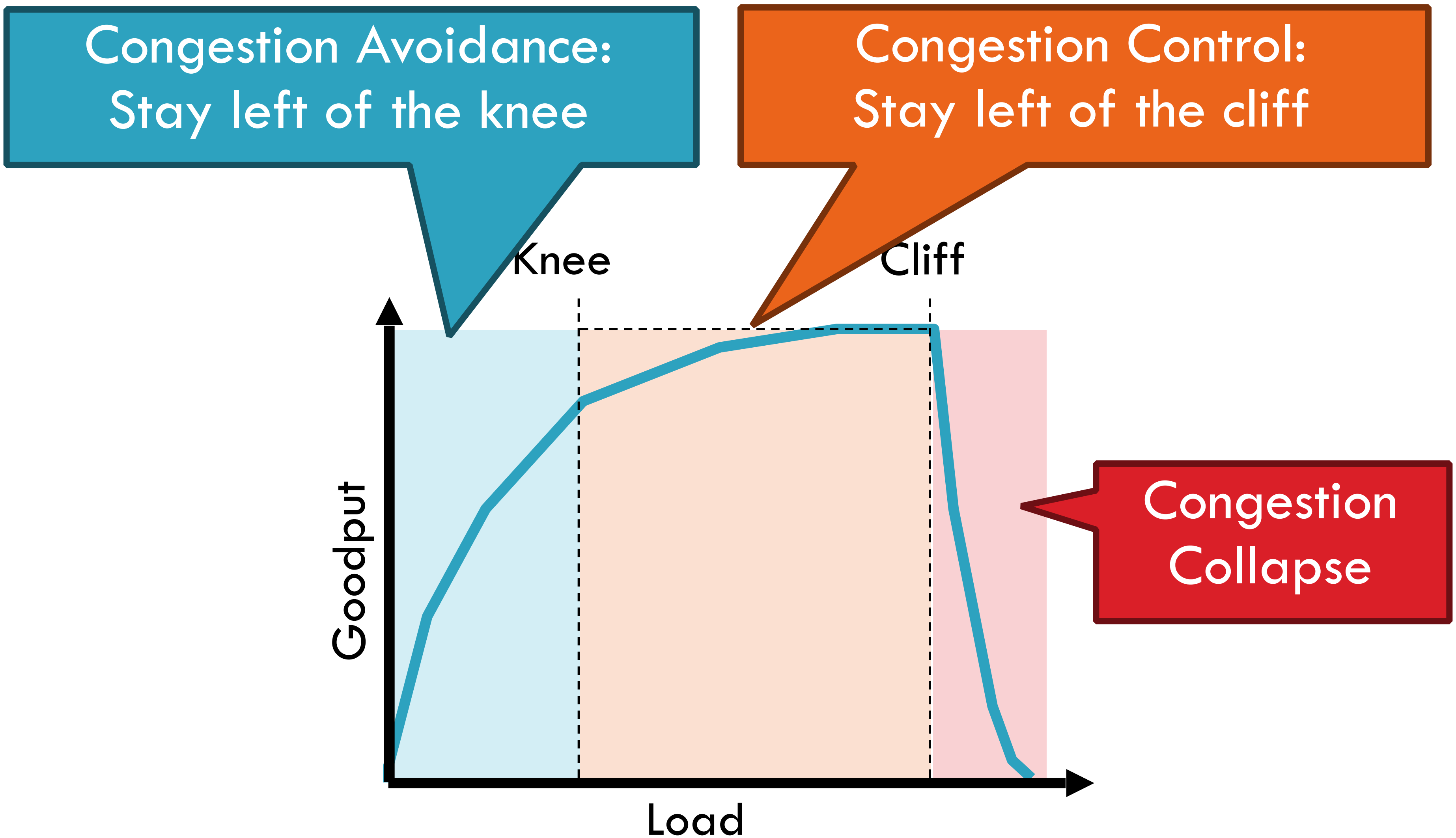
Cong. Control vs. Cong. Avoidance



Cong. Control vs. Cong. Avoidance



Cong. Control vs. Cong. Avoidance



Advertised Window, Revisited

- Does TCP's advertised window solve congestion?

Advertised Window, Revisited

- Does TCP's advertised window solve congestion?

NO

- The advertised window only protects the receiver
- A sufficiently fast receiver can max the window
  - What if the network is slower than the receiver?
  - What if there are other concurrent flows?

Advertised Window, Revisited

- Does TCP's advertised window solve congestion?

  NO

- The advertised window only protects the receiver
- A sufficiently fast receiver can max the window
  - What if the network is slower than the receiver?
  - What if there are other concurrent flows?
- Key points
  - Window size determines send rate
  - Window must be adjusted to prevent congestion collapse

Goals of Congestion Control

Goals of Congestion Control

1. Adjusting to the bottleneck bandwidth
2. Adjusting to variations in bandwidth
3. Sharing bandwidth between flows
4. Maximizing throughput

General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse

General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse
- Reservations
  - Pre-arrange bandwidth allocations for flows
  - Requires negotiation before sending packets
  - Must be supported by the network

General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse
- Reservations
  - Pre-arrange bandwidth allocations for flows
  - Requires negotiation before sending packets
  - Must be supported by the network
- Dynamic adjustment
  - Use probes to estimate level of congestion
  - Speed up when congestion is low
  - Slow down when congestion increases
  - Messy dynamics, requires distributed coordination

General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse
- Reservations
  - Pre-arrange bandwidth allocations for flows
  - Requires negotiation before sending packets
  - Must be supported by the network
- Dynamic adjustment
  - Use probes to estimate level of congestion
  - Speed up when congestion is low
  - Slow down when congestion increases
  - Messy dynamics, requires distributed coordination

TCP Congestion Control

- Each TCP connection has a window
  - Controls the number of unACKed packets
- Sending rate is ~ window/RTT
- Idea: vary the window size to control the send rate

TCP Congestion Control

- Each TCP connection has a window
  - Controls the number of unACKed packets
- Sending rate is ~ window/RTT
- Idea: vary the window size to control the send rate
- Introduce a congestion window at the sender
  - Congestion control is sender-side problem

Congestion Window (*cwnd*)

- Limits how much data is in transit
- Denominated in bytes

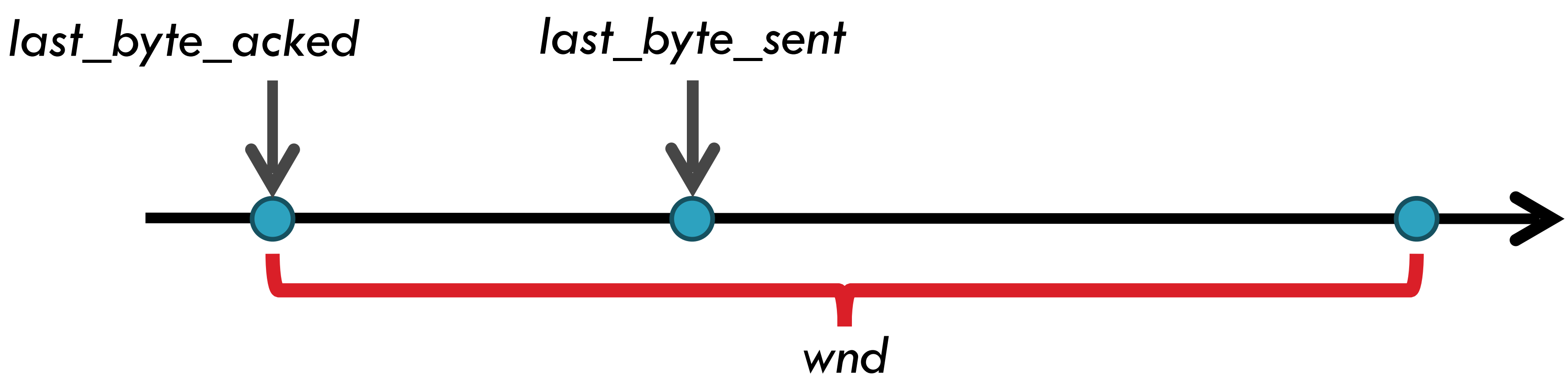
1. *wnd* = min(*cwnd*, *adv_wnd*);
2. *effective_wnd* = *wnd* –

   (*last_byte_sent* – *last_byte_acked*);

Congestion Window (*cwnd*)

- Limits how much data is in transit
- Denominated in bytes

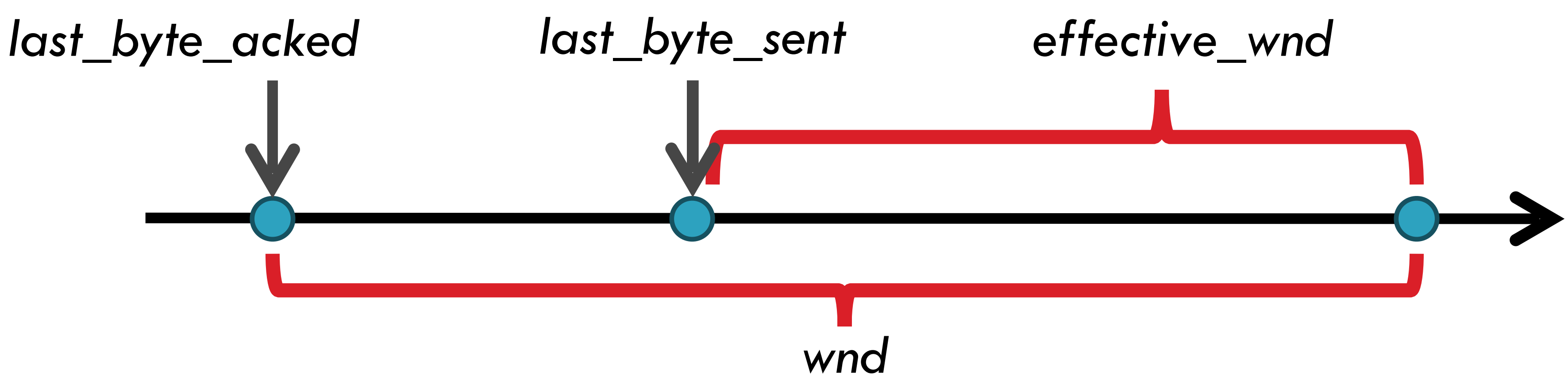
1. *wnd* = min(*cwnd*, *adv_wnd*);
2. *effective_wnd* = *wnd* –
   (*last_byte_sent* – *last_byte_acked*);



Congestion Window (*cwnd*)

- Limits how much data is in transit
- Denominated in bytes

1. *wnd* = min(*cwnd*, *adv_wnd*);
2. *effective_wnd* = *wnd* –
   (*last_byte_sent* – *last_byte_acked*);



Two Basic Components

1. Detect congestion

Two Basic Components

1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)

Two Basic Components

1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)

Except on wireless networks

Two Basic Components

Except on wireless networks

1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)
2. Rate adjustment algorithm
   - Modify *cwnd*
   - Probe for bandwidth
   - Responding to congestion

Rate Adjustment

- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly

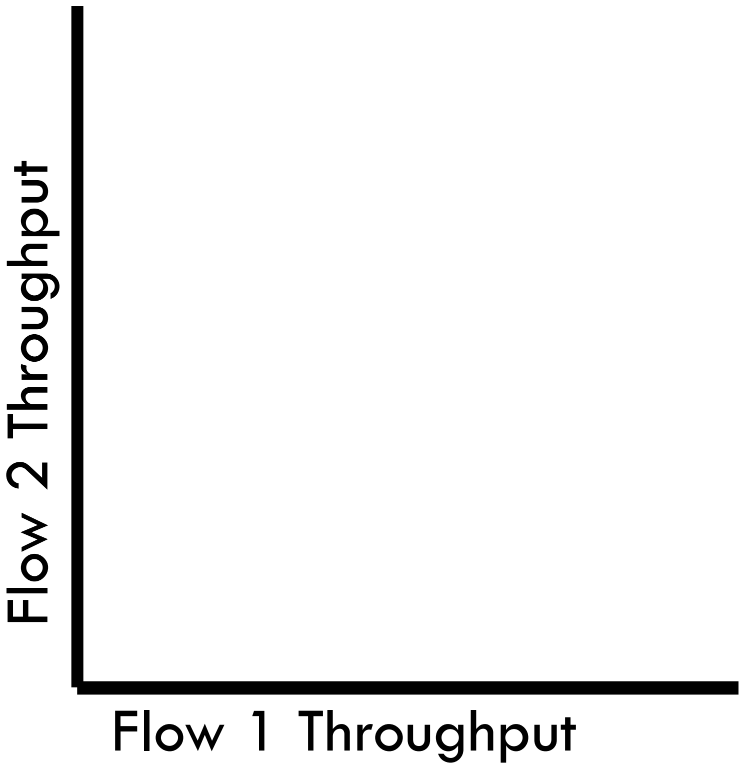
Rate Adjustment

- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly
- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - *cwnd* growth is proportional to RTT
  - On loss: decrease cwnd
    - Data is being lost, there must be congestion

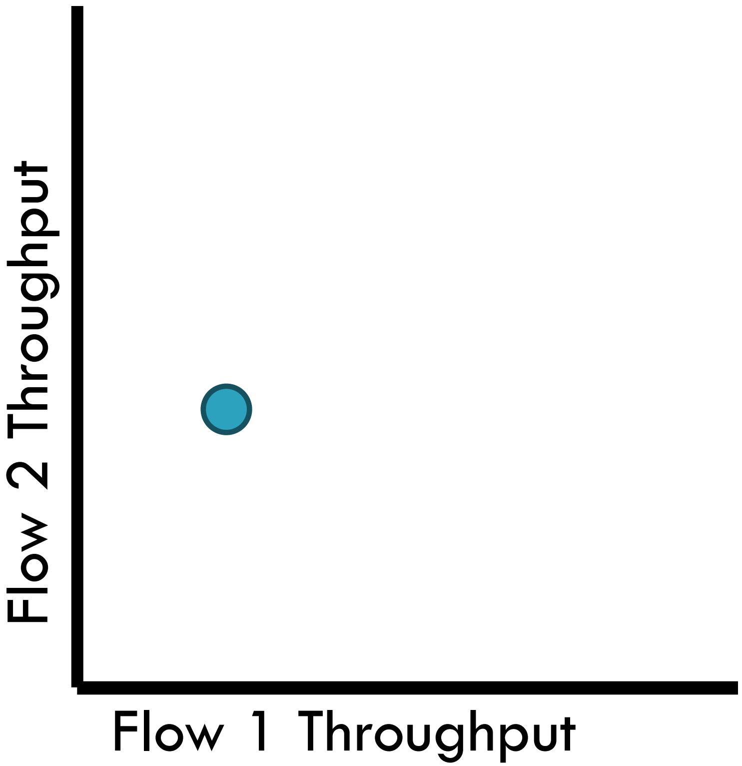
Rate Adjustment

- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly
- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - *cwnd* growth is proportional to RTT
  - On loss: decrease cwnd
    - Data is being lost, there must be congestion
- Question: increase/decrease functions to use?

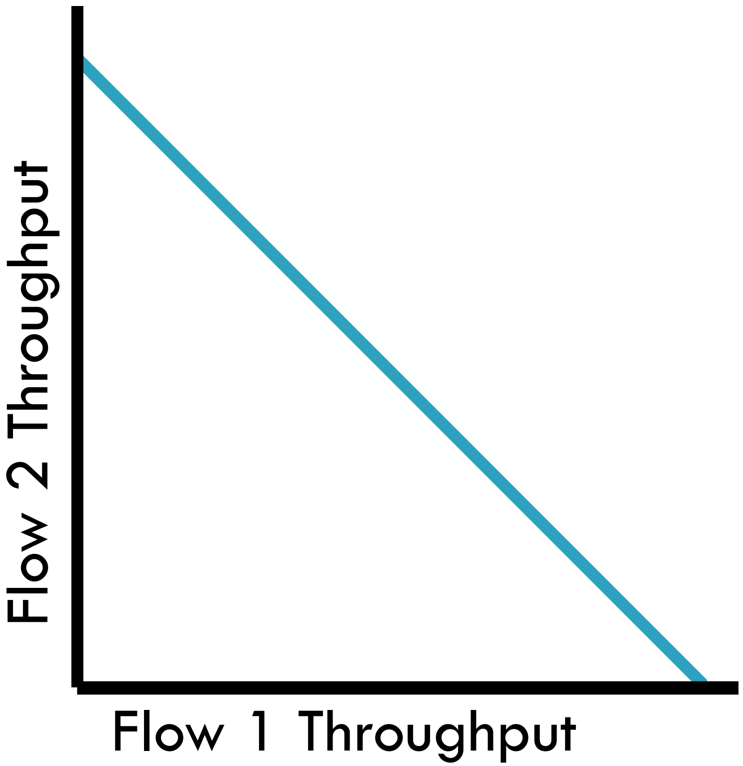
Utilization and Fairness



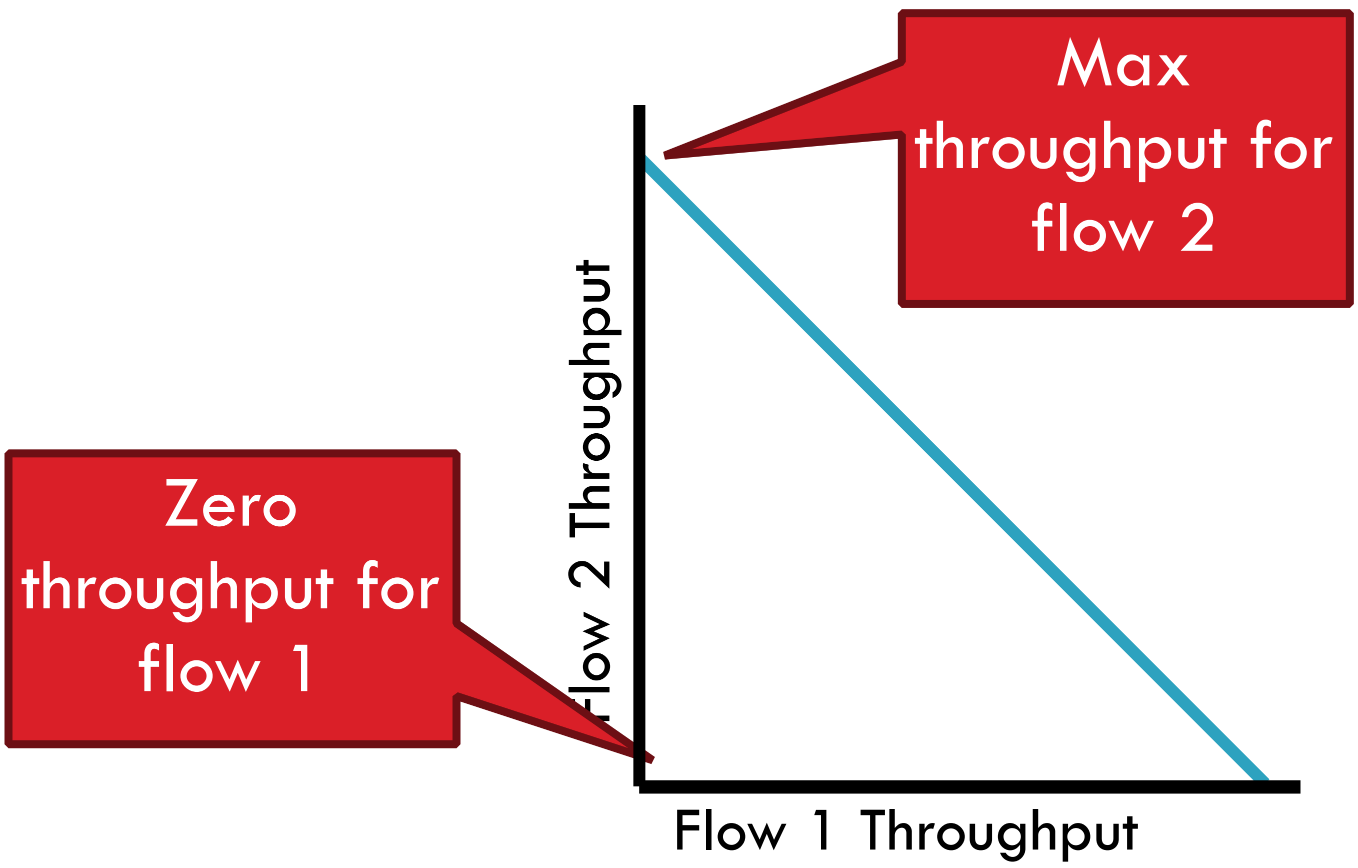
Utilization and Fairness



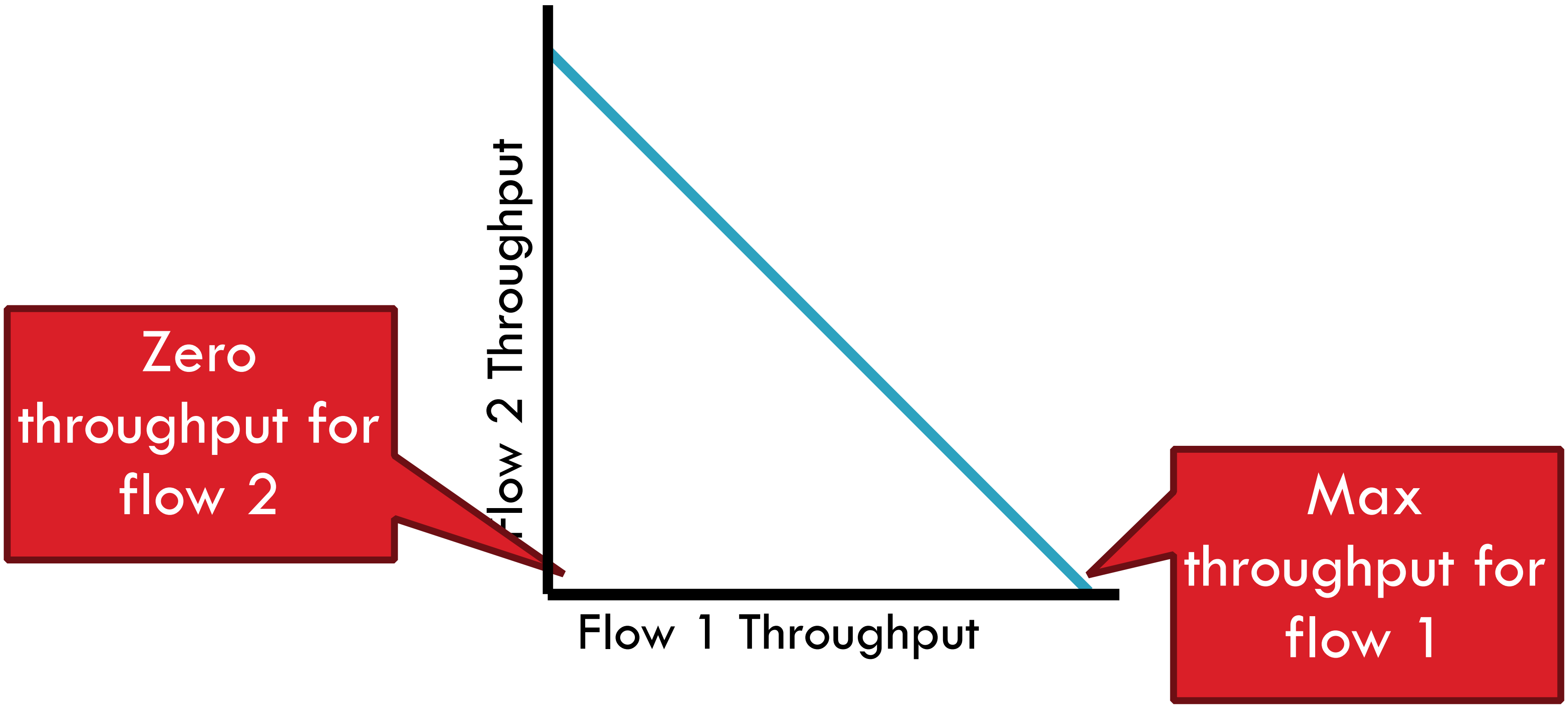
Utilization and Fairness



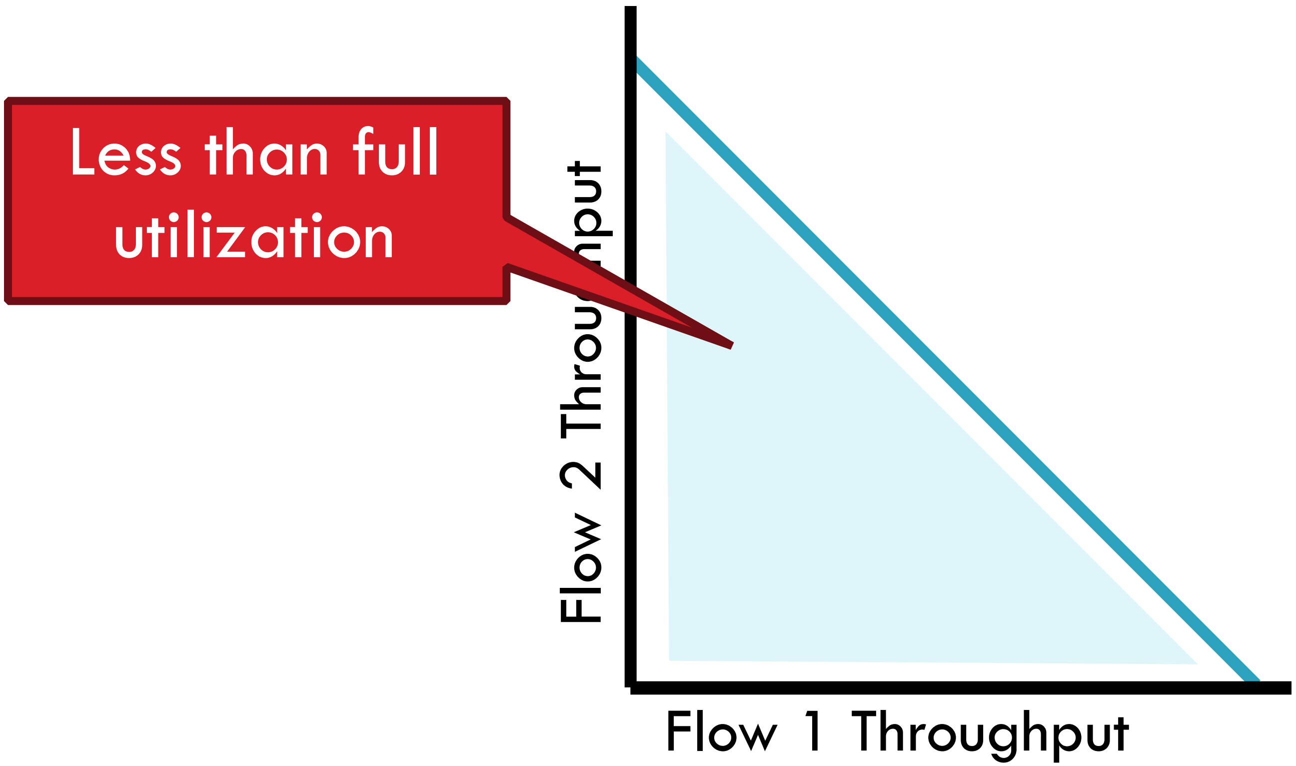
Utilization and Fairness



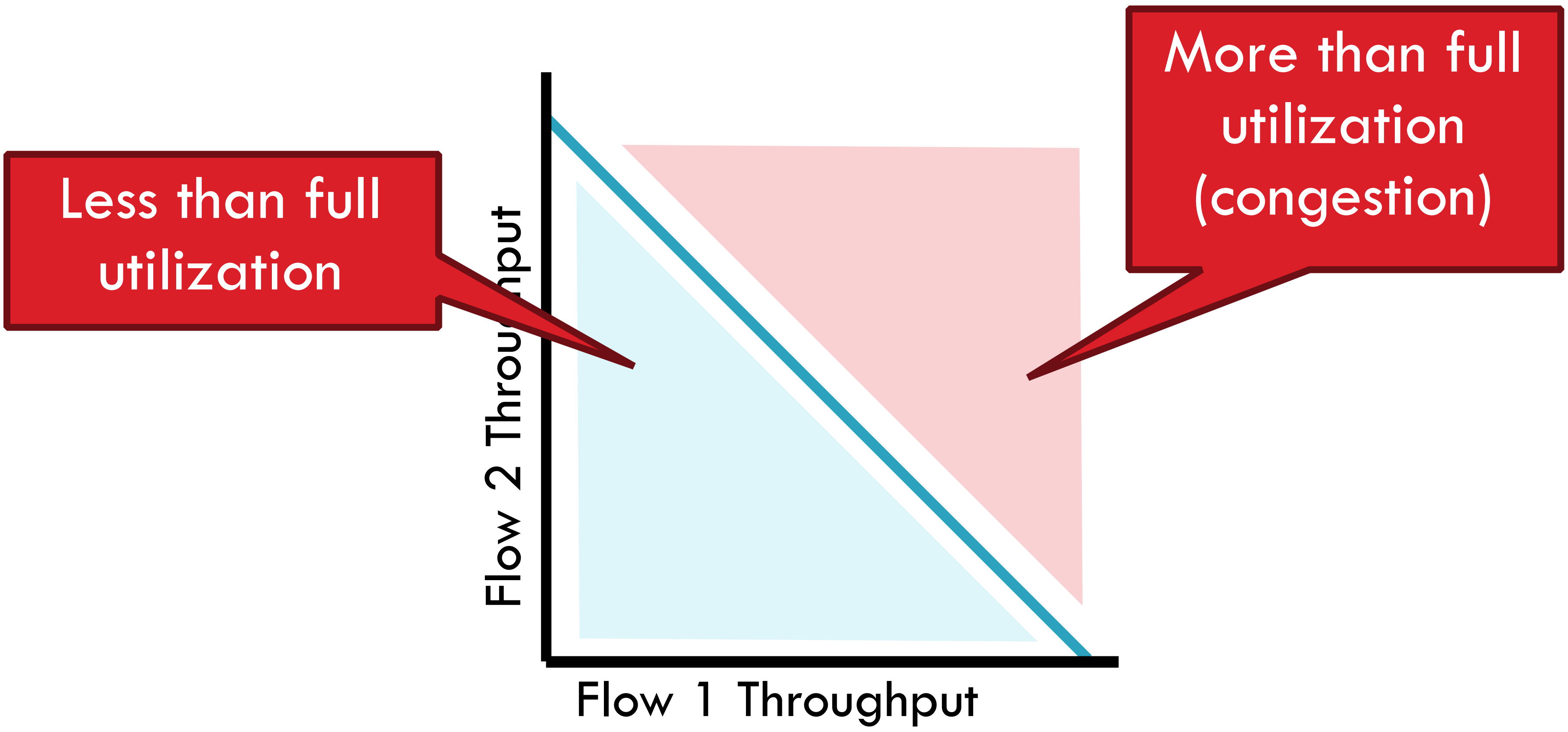
Utilization and Fairness



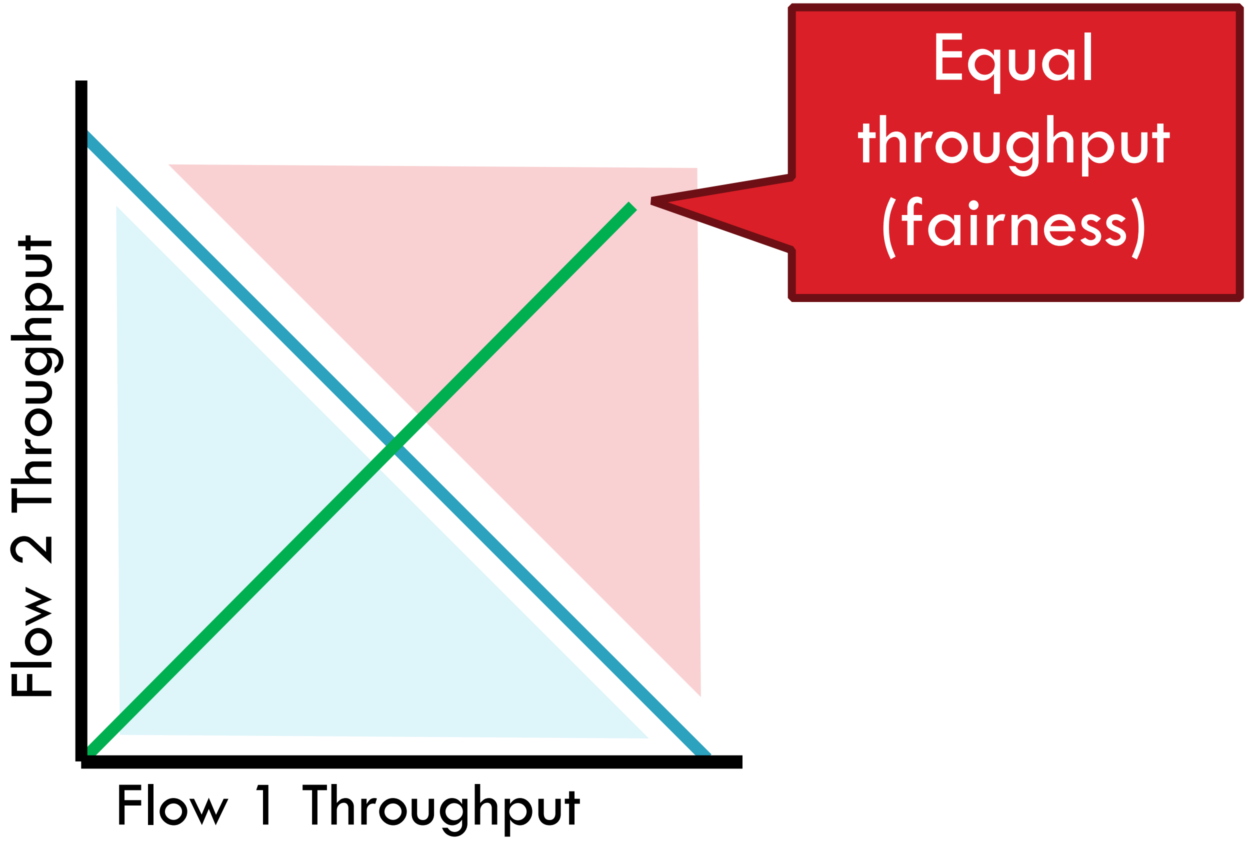
Utilization and Fairness



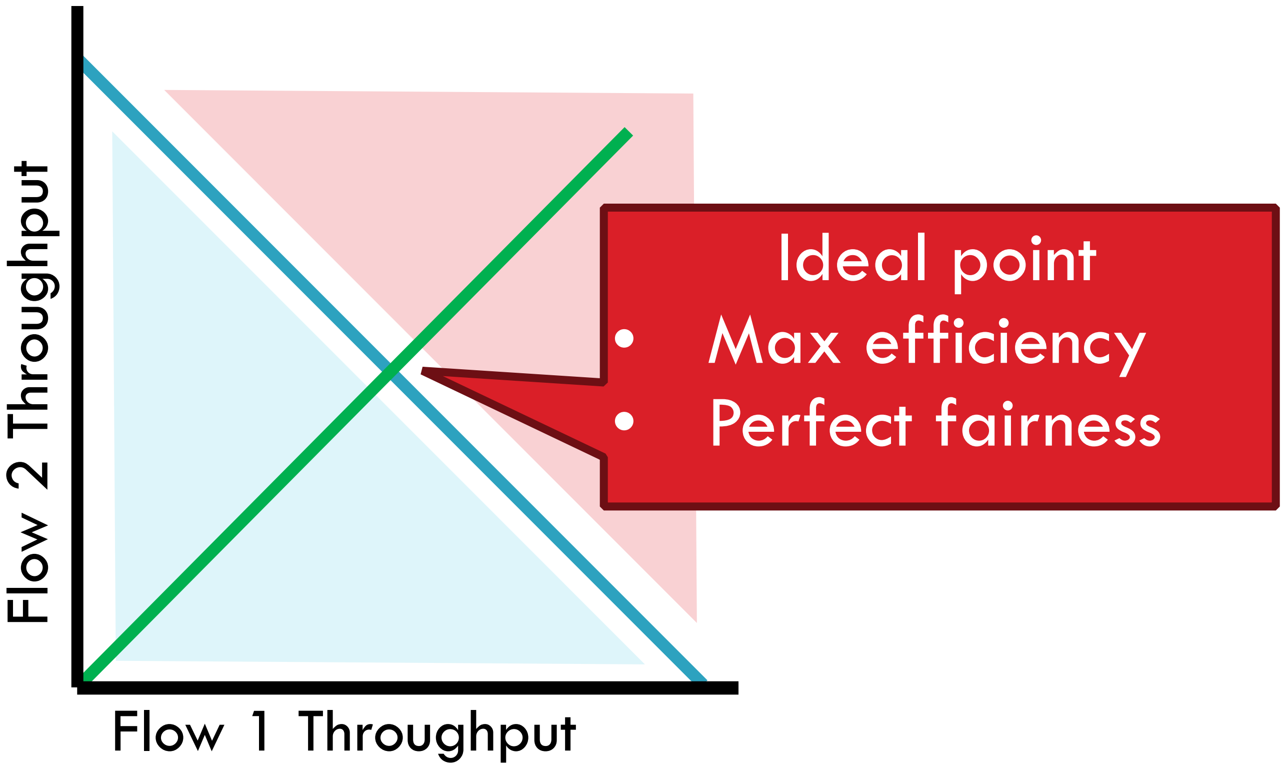
Utilization and Fairness



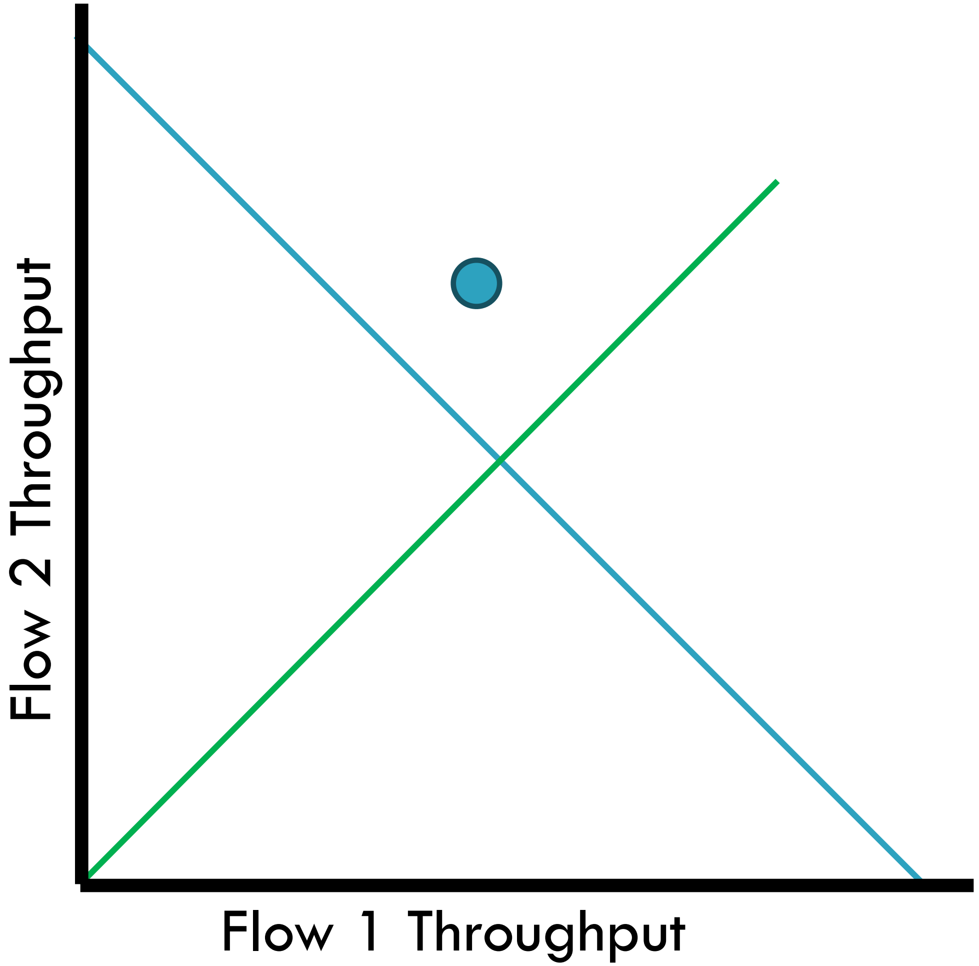
Utilization and Fairness



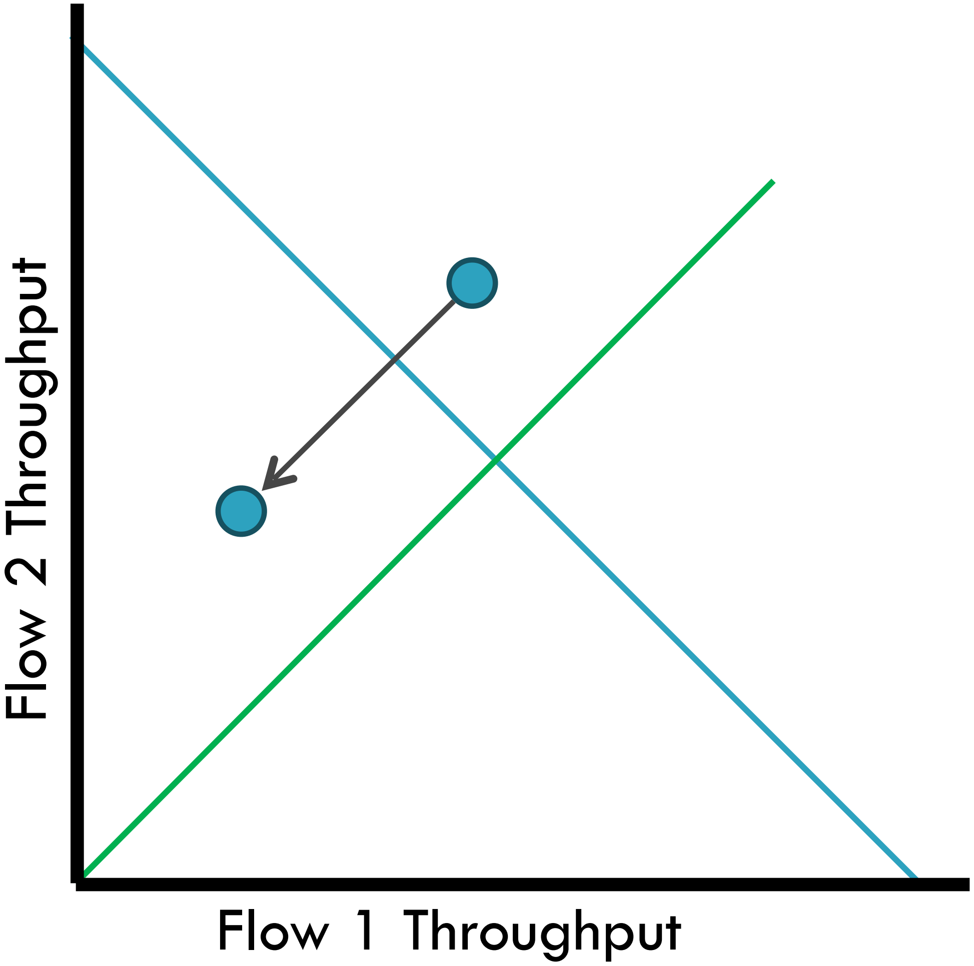
Utilization and Fairness



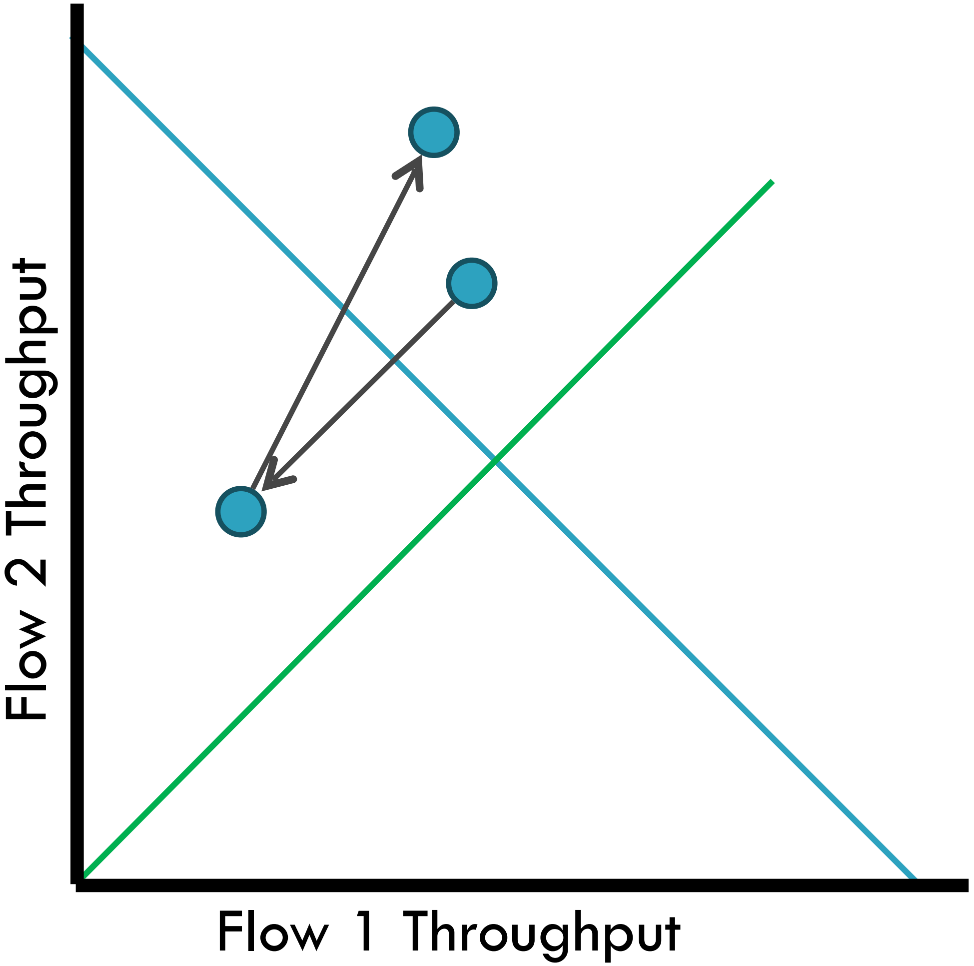
Multiplicative Increase, Additive Decrease



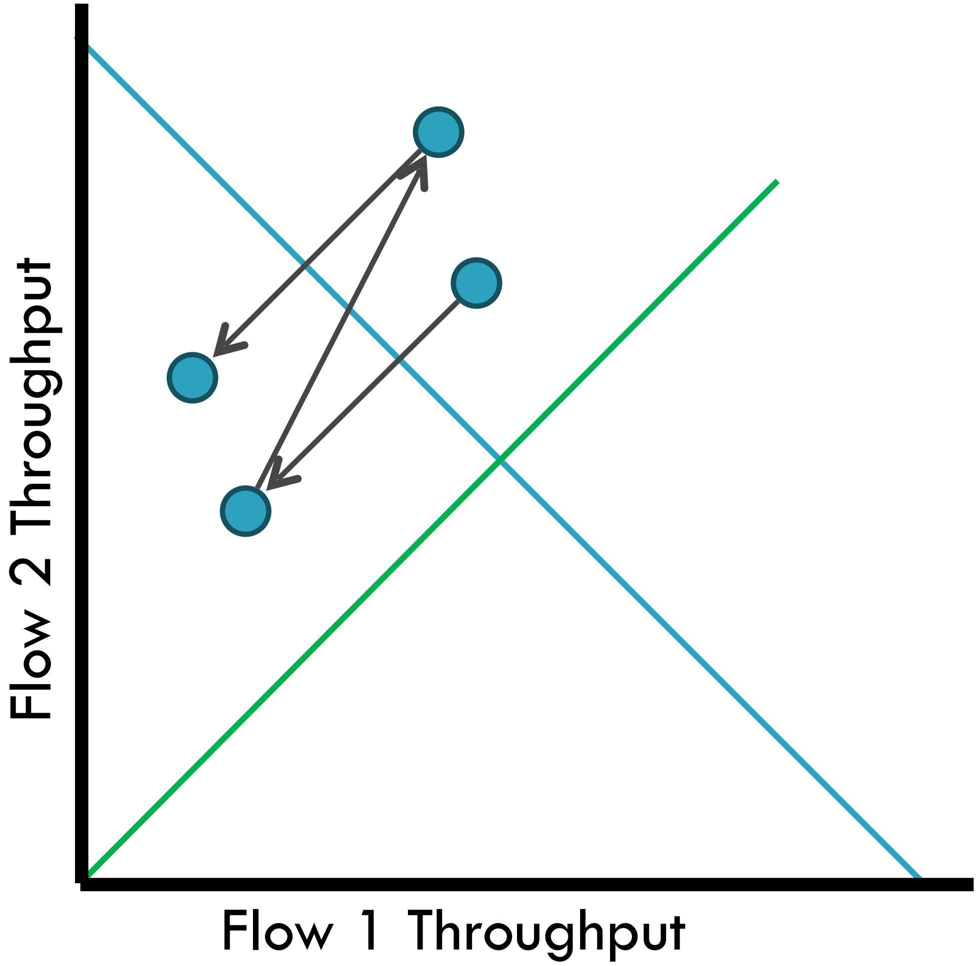
Multiplicative Increase, Additive Decrease



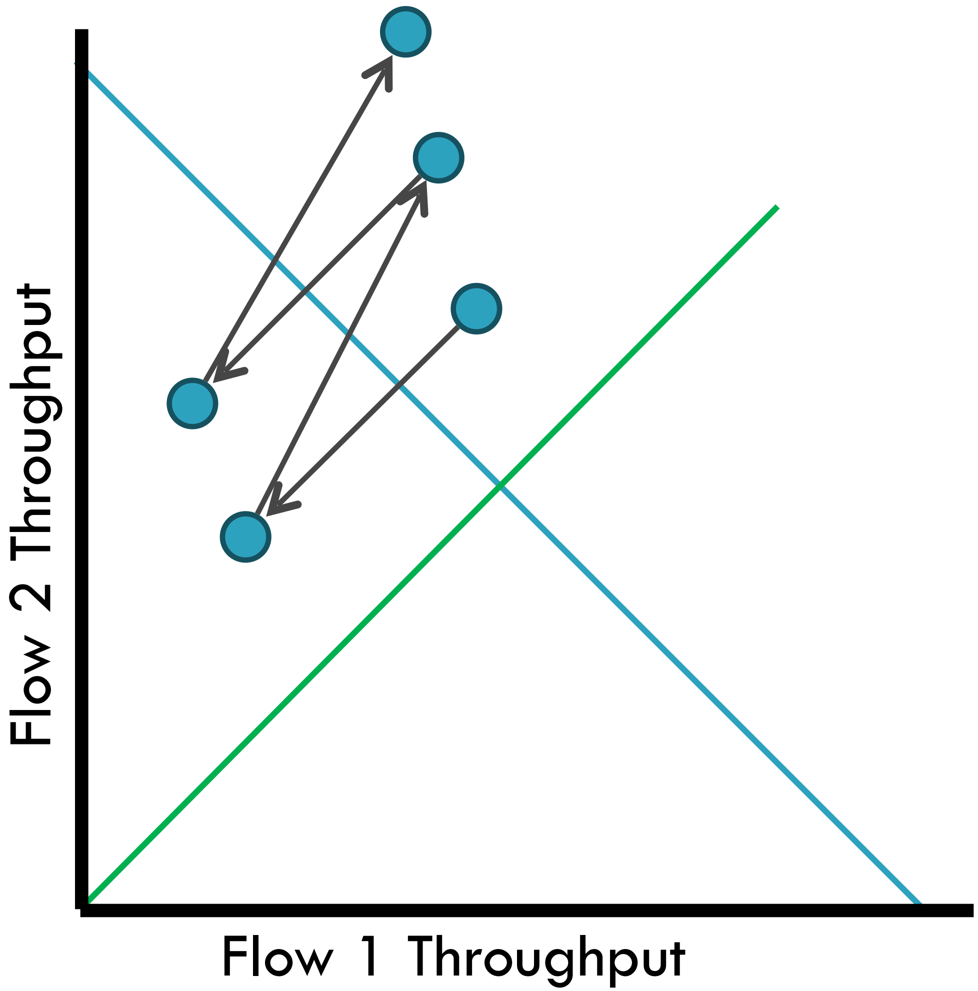
Multiplicative Increase, Additive Decrease



Multiplicative Increase, Additive Decrease

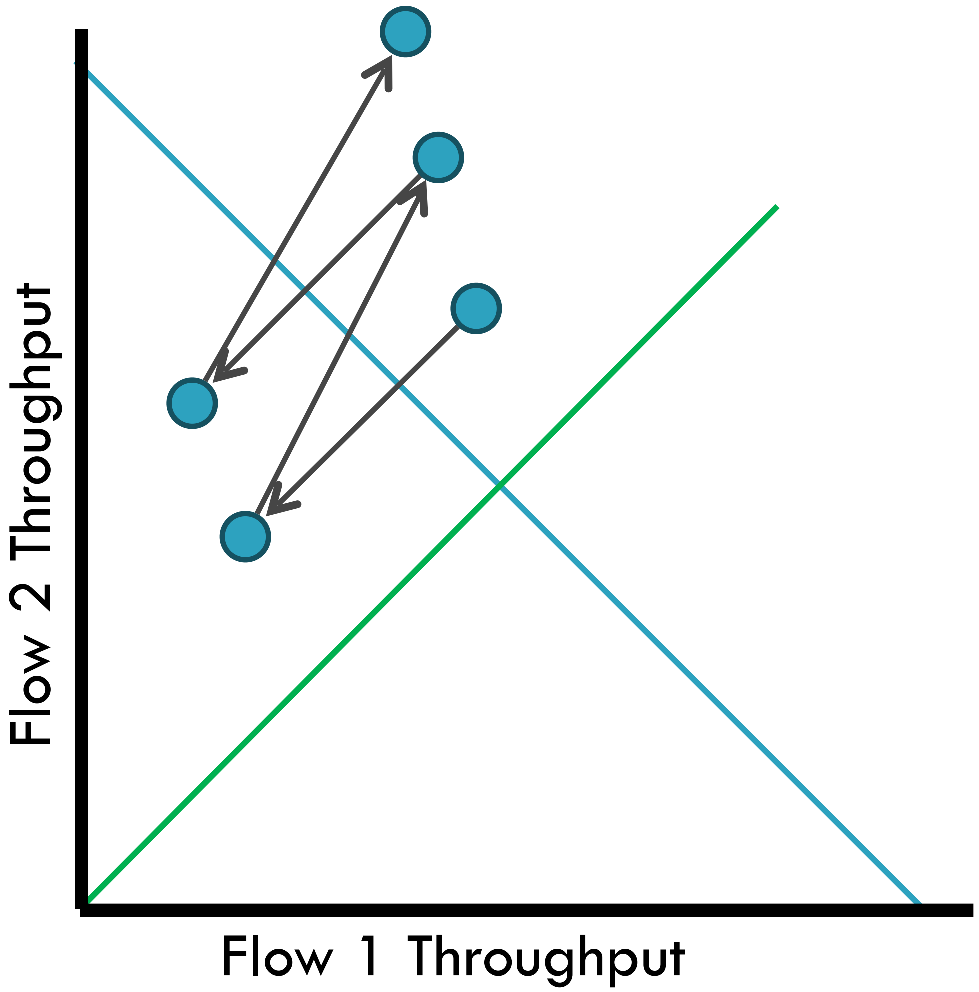


Multiplicative Increase, Additive Decrease



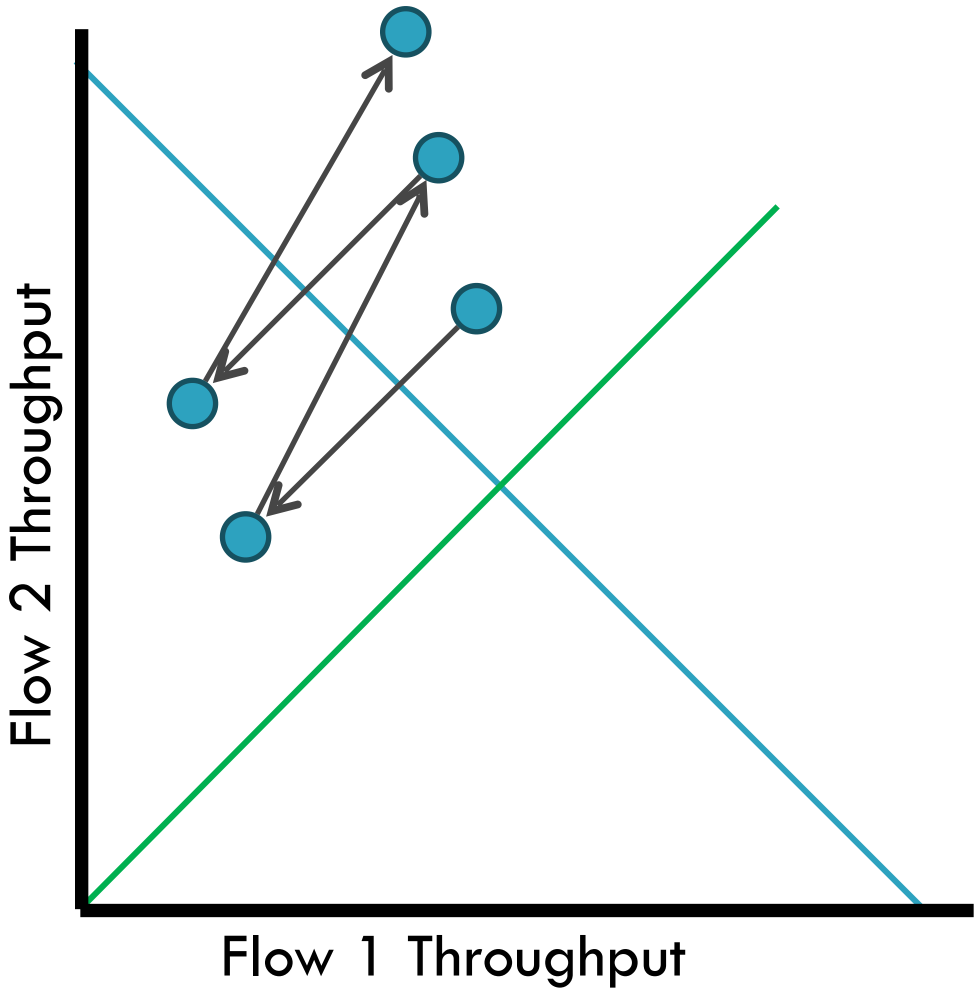
Multiplicative Increase, Additive Decrease

- Not stable!

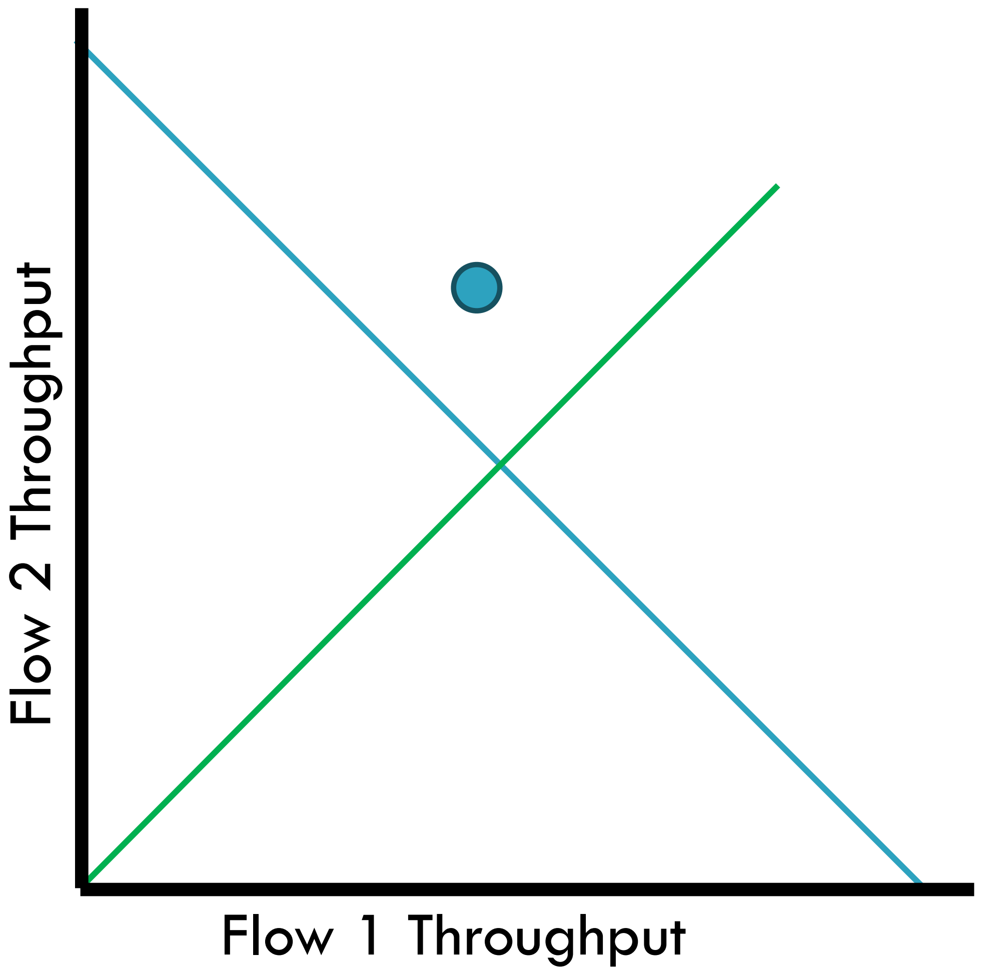


Multiplicative Increase, Additive Decrease

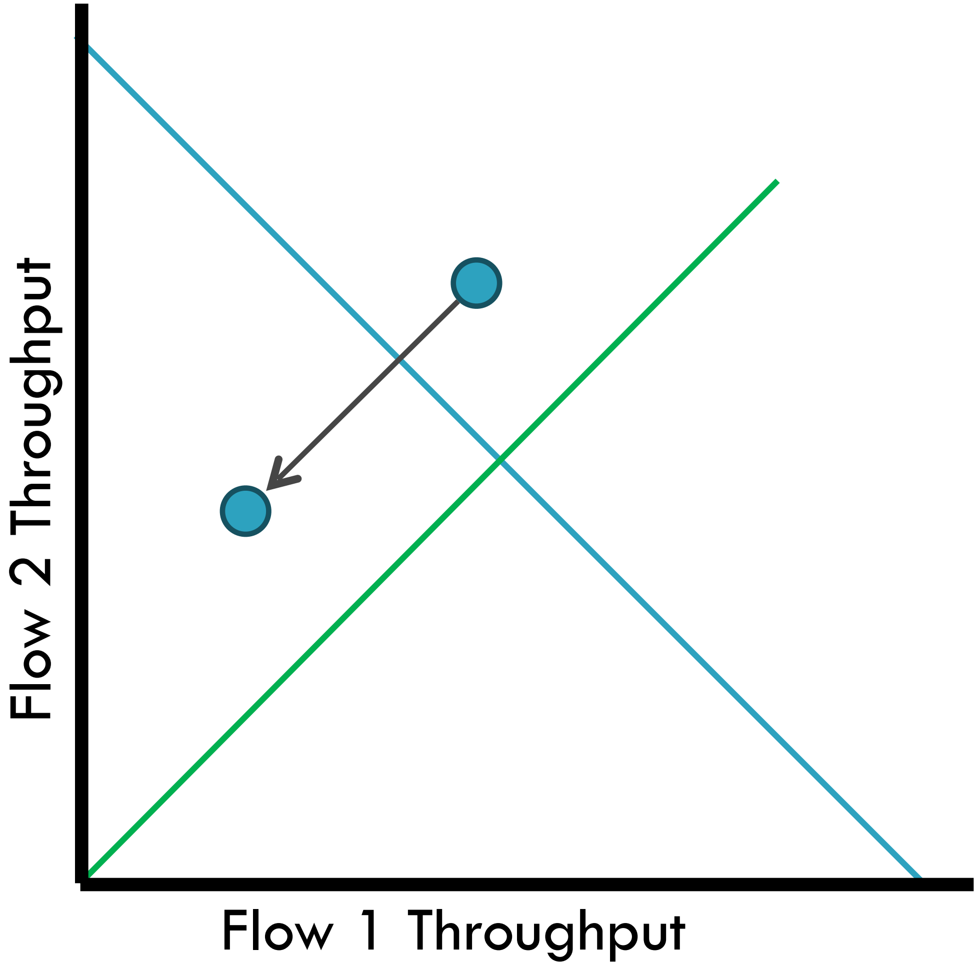
- Not stable!
- Veers away from fairness



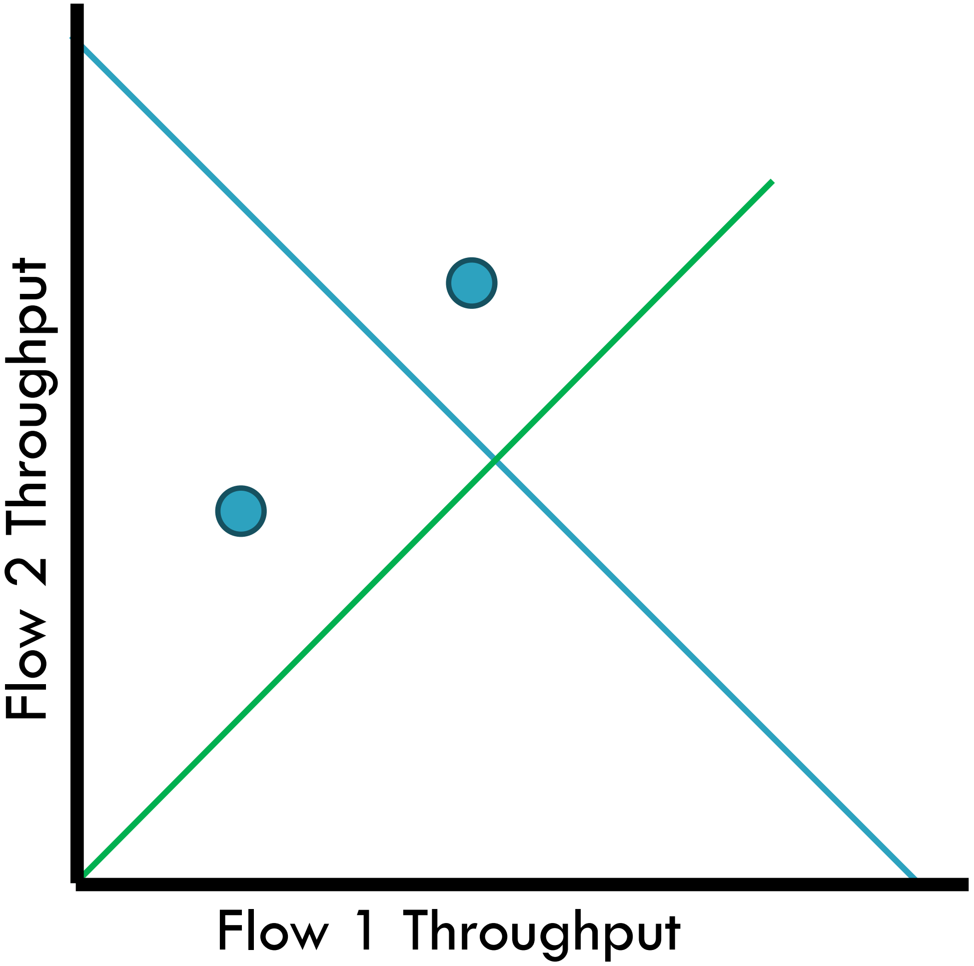
Additive Increase, Additive Decrease



Additive Increase, Additive Decrease

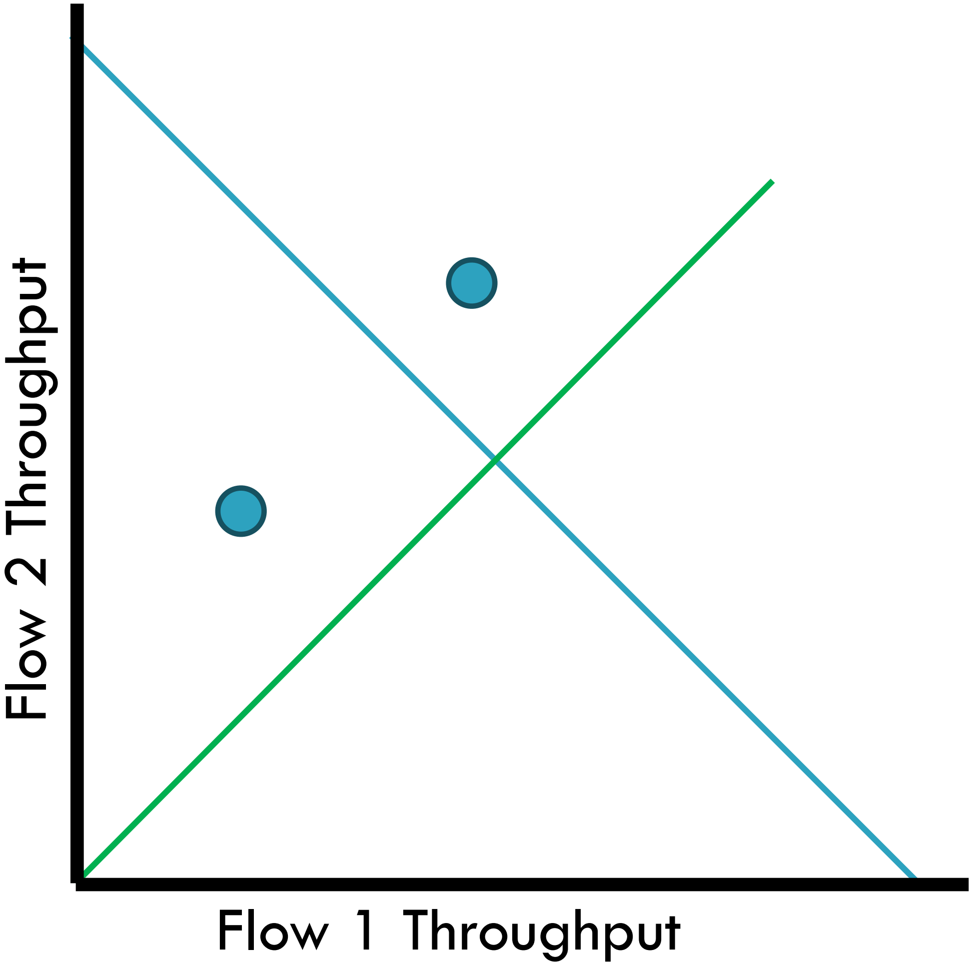


Additive Increase, Additive Decrease



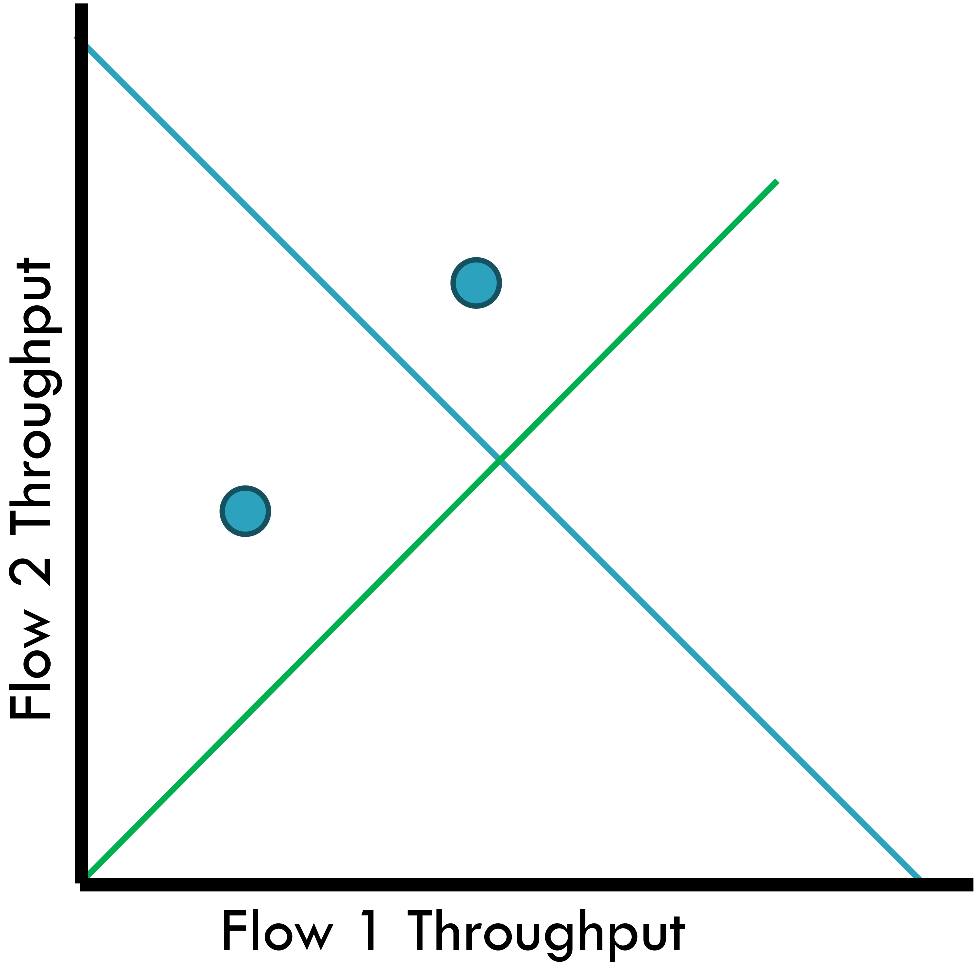
Additive Increase, Additive Decrease

- Stable

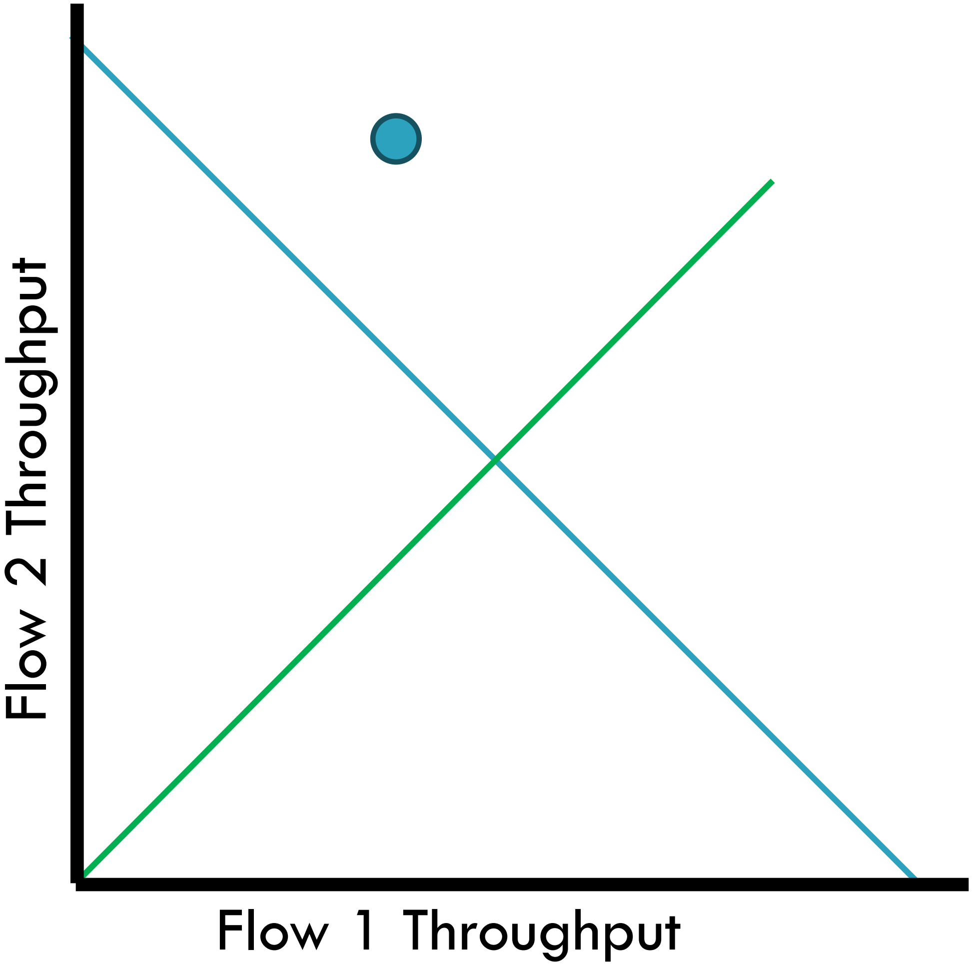


Additive Increase, Additive Decrease

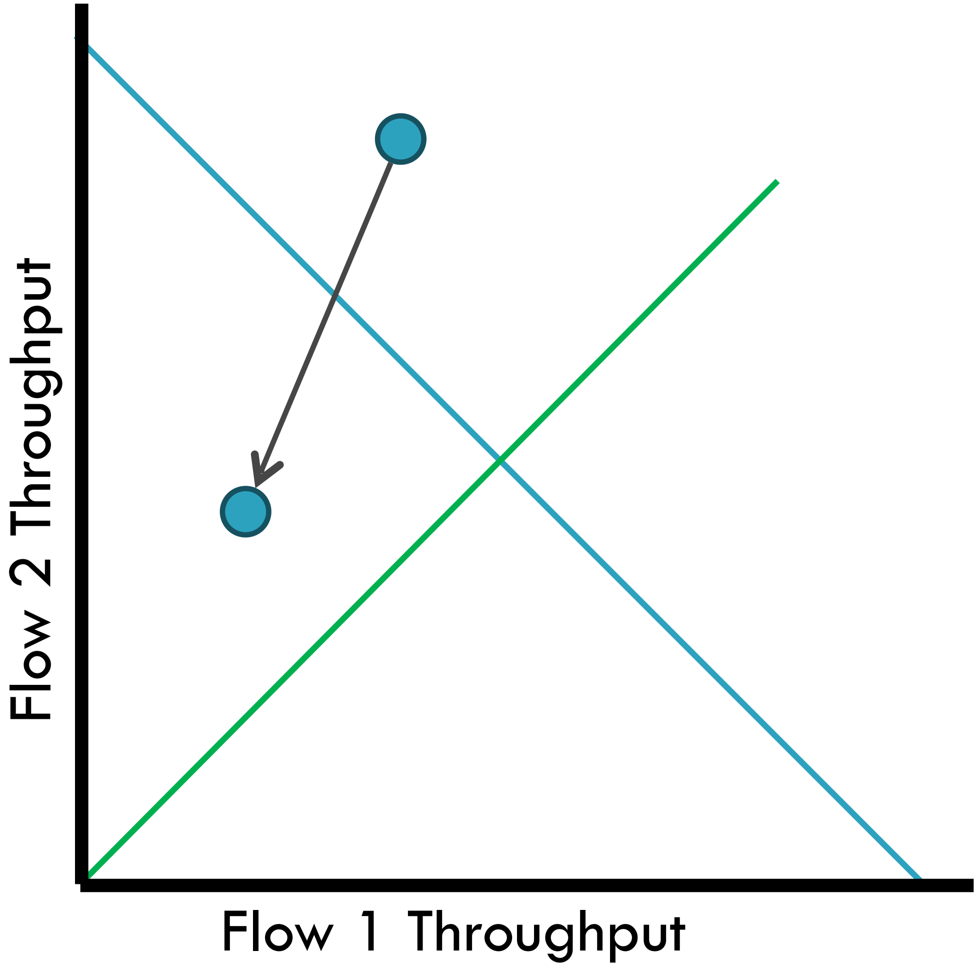
- Stable
- But does not converge to fairness



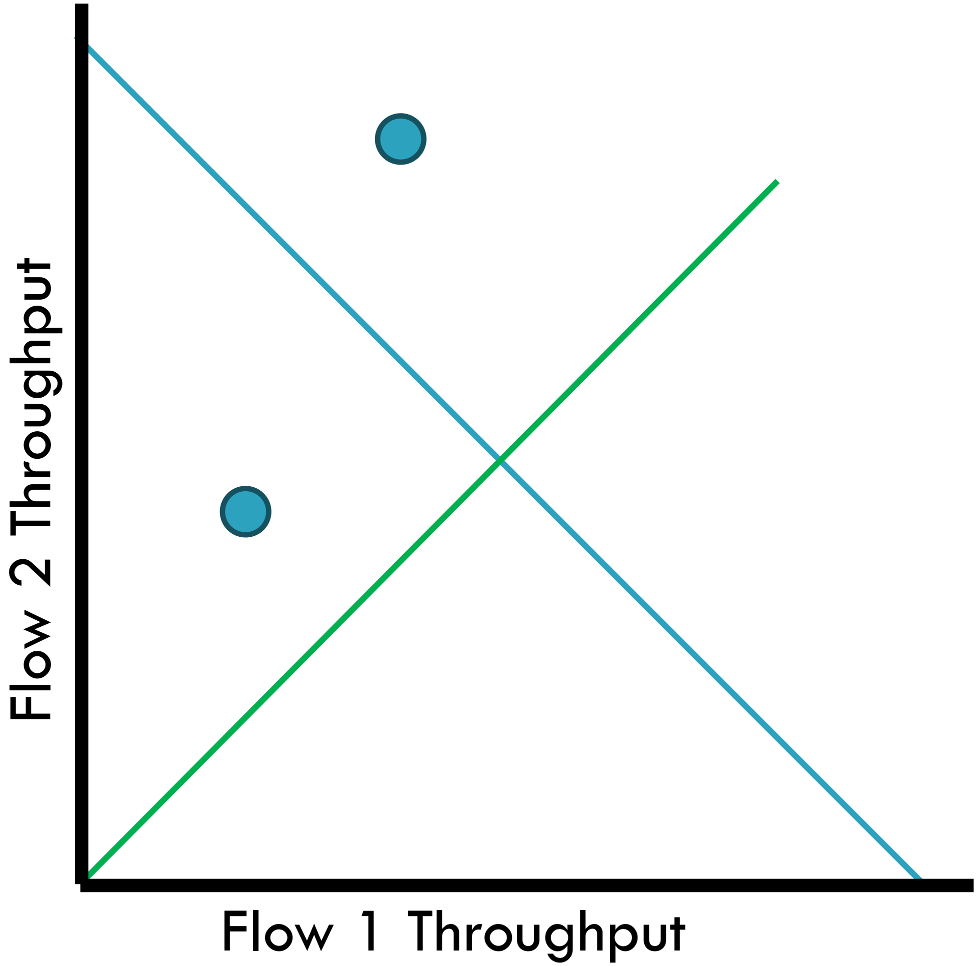
Multiplicative Increase, Multiplicative Decrease



Multiplicative Increase, Multiplicative Decrease

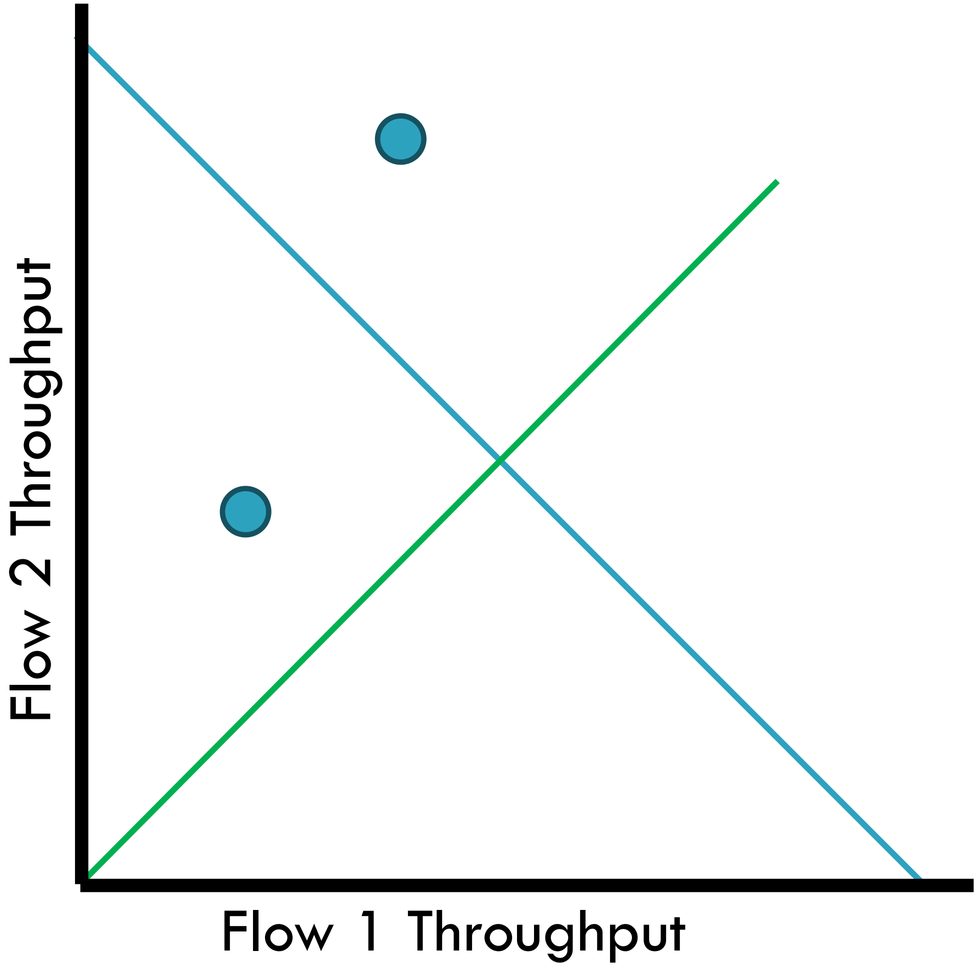


Multiplicative Increase, Multiplicative Decrease



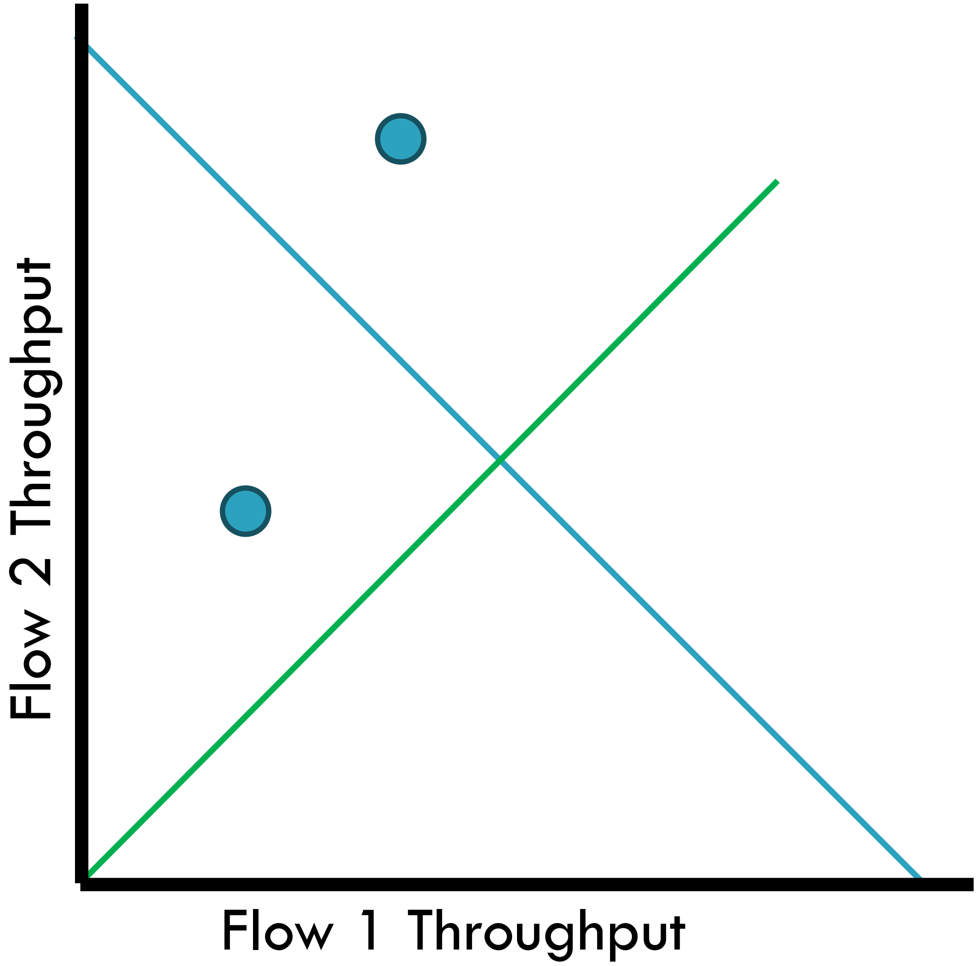
Multiplicative Increase, Multiplicative Decrease

- Stable

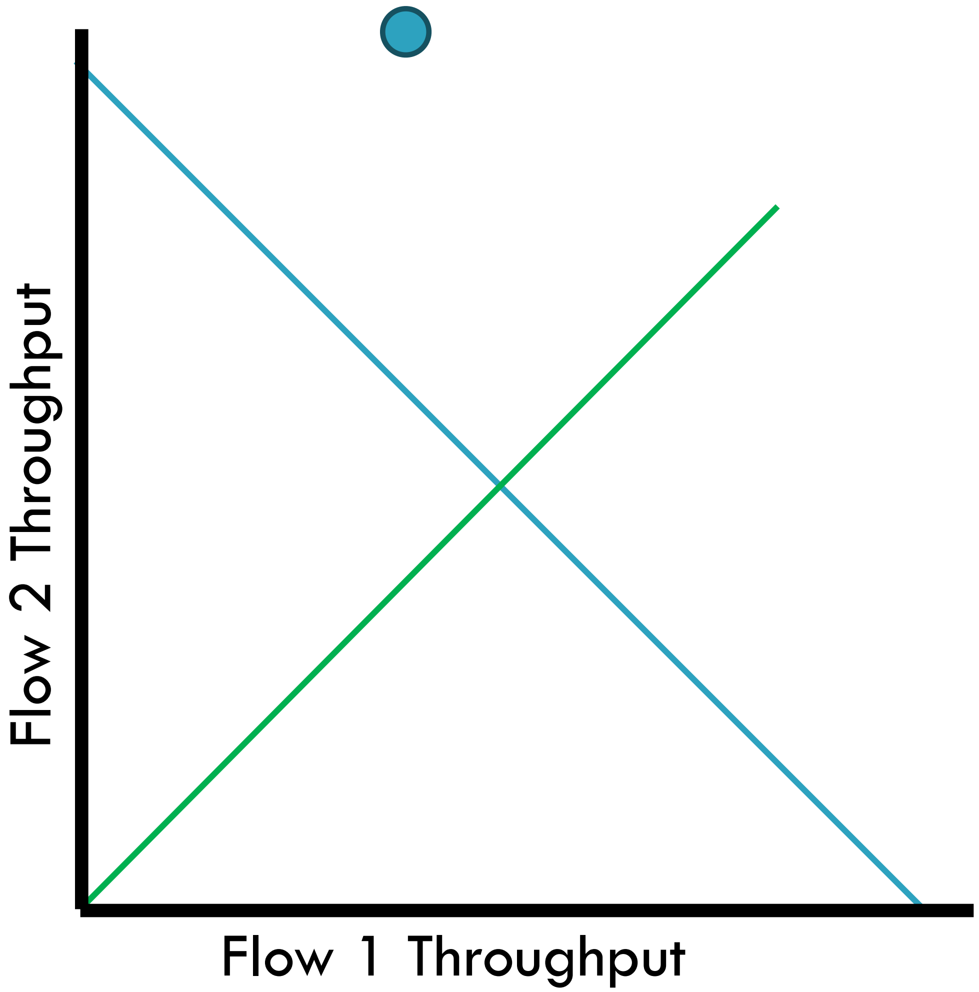


Multiplicative Increase, Multiplicative Decrease

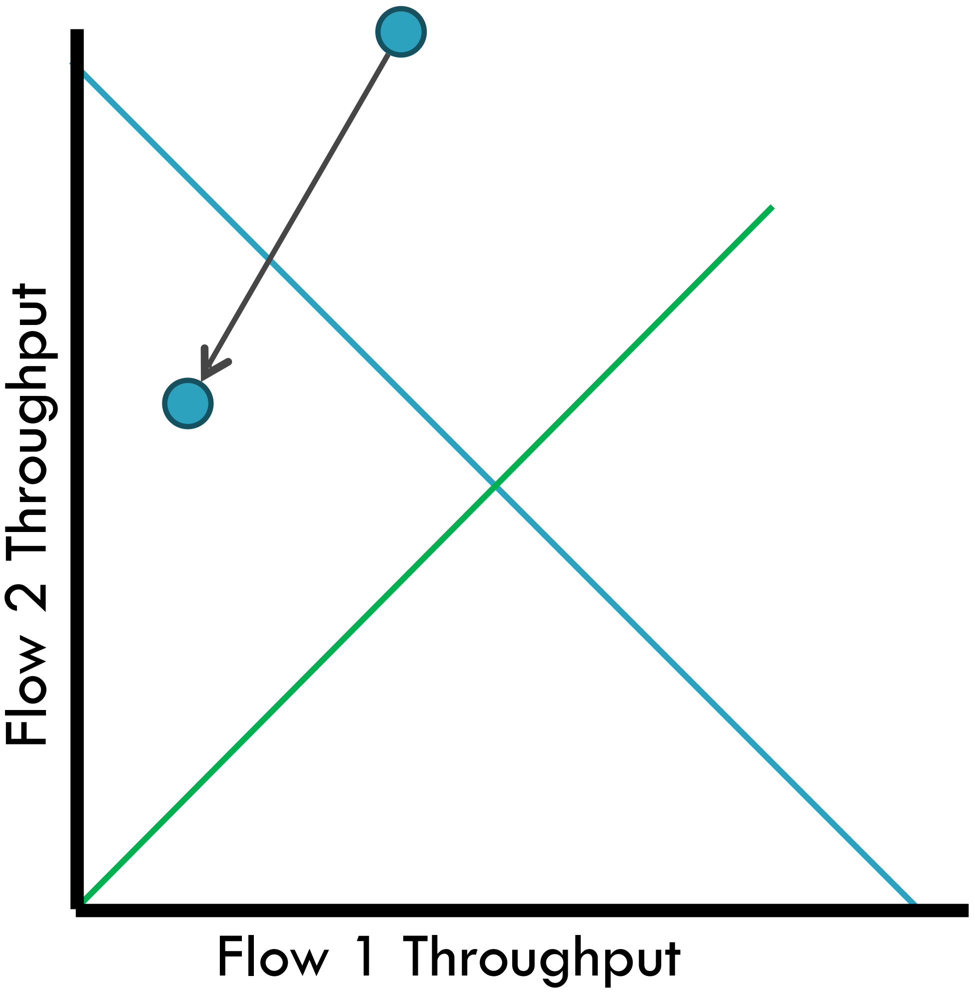
- Stable
- But does not converge to fairness



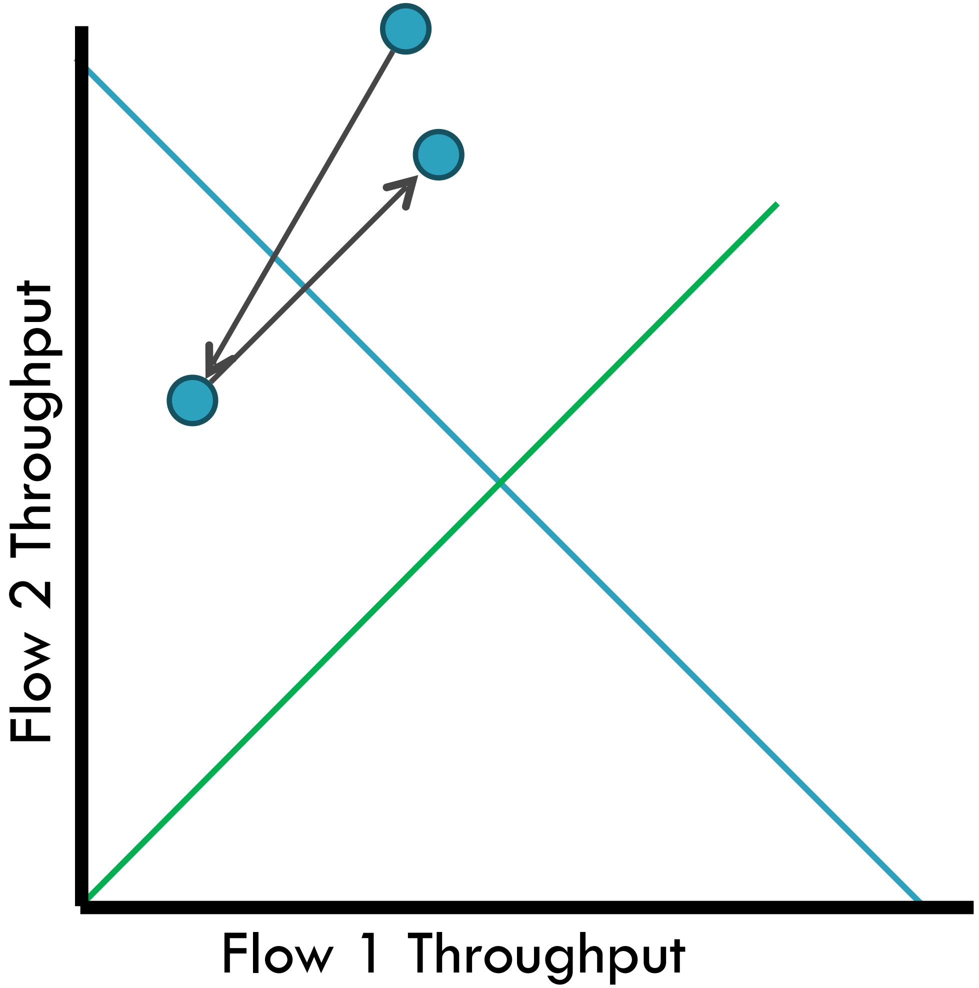
Additive Increase, Multiplicative Decrease



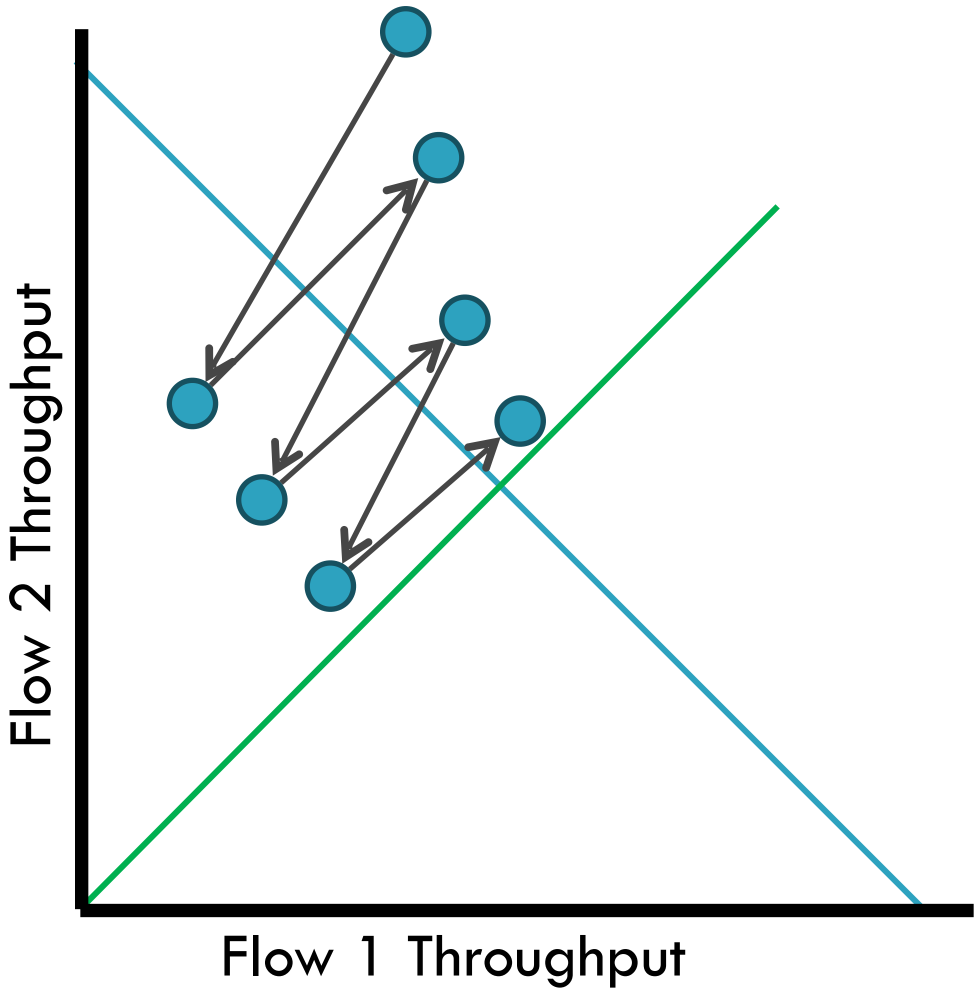
Additive Increase, Multiplicative Decrease



Additive Increase, Multiplicative Decrease

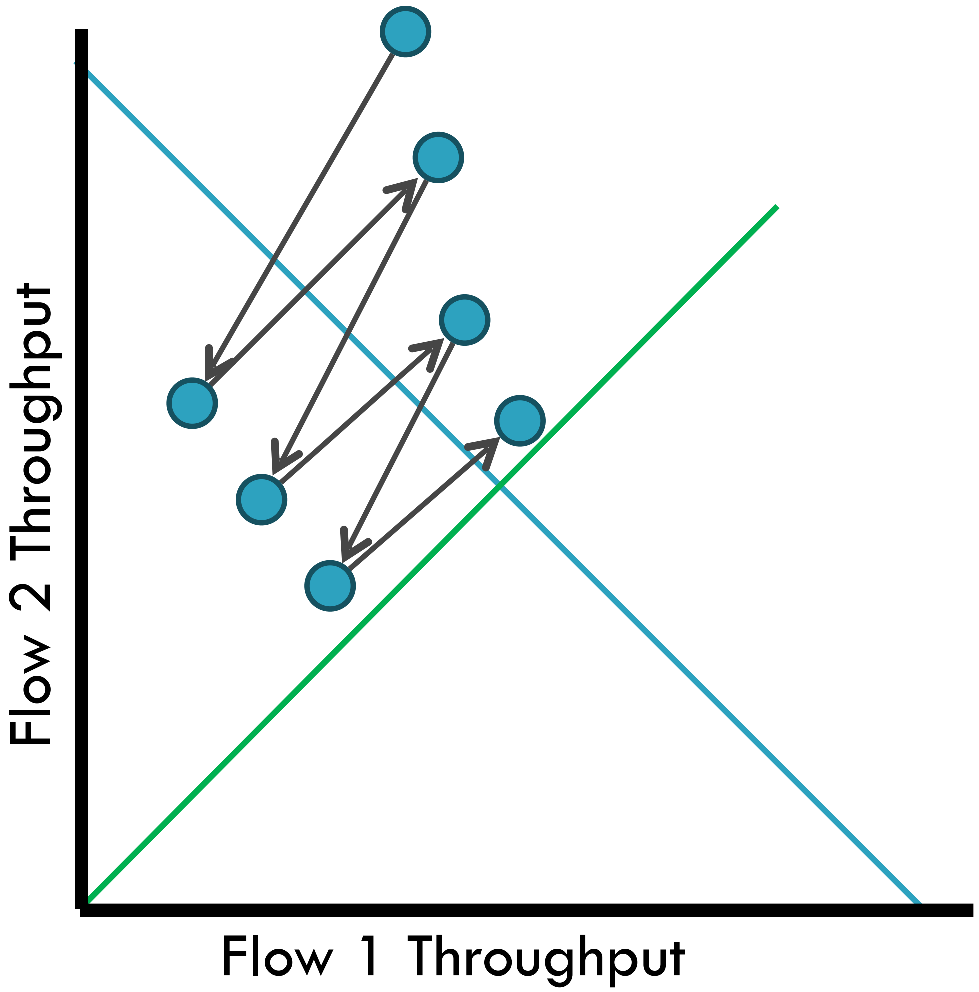


Additive Increase, Multiplicative Decrease



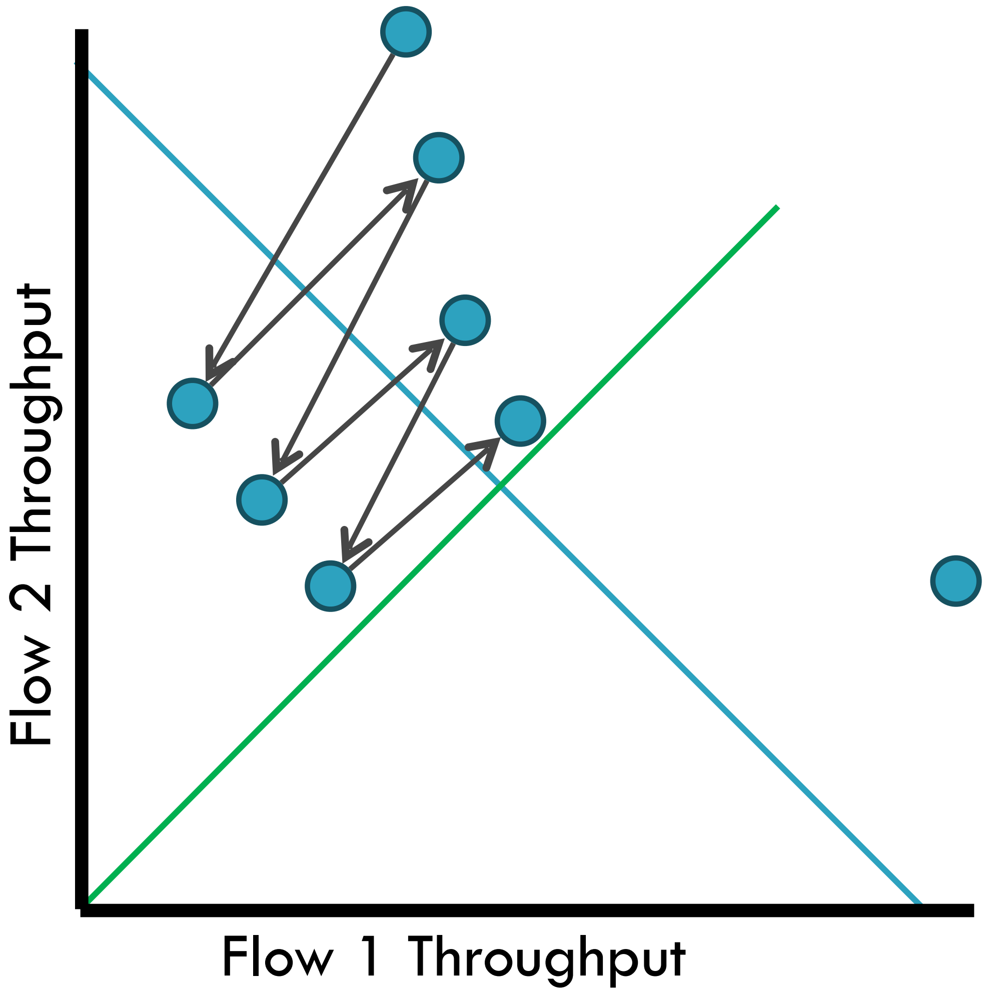
Additive Increase, Multiplicative Decrease

- Converges to stable and fair cycle



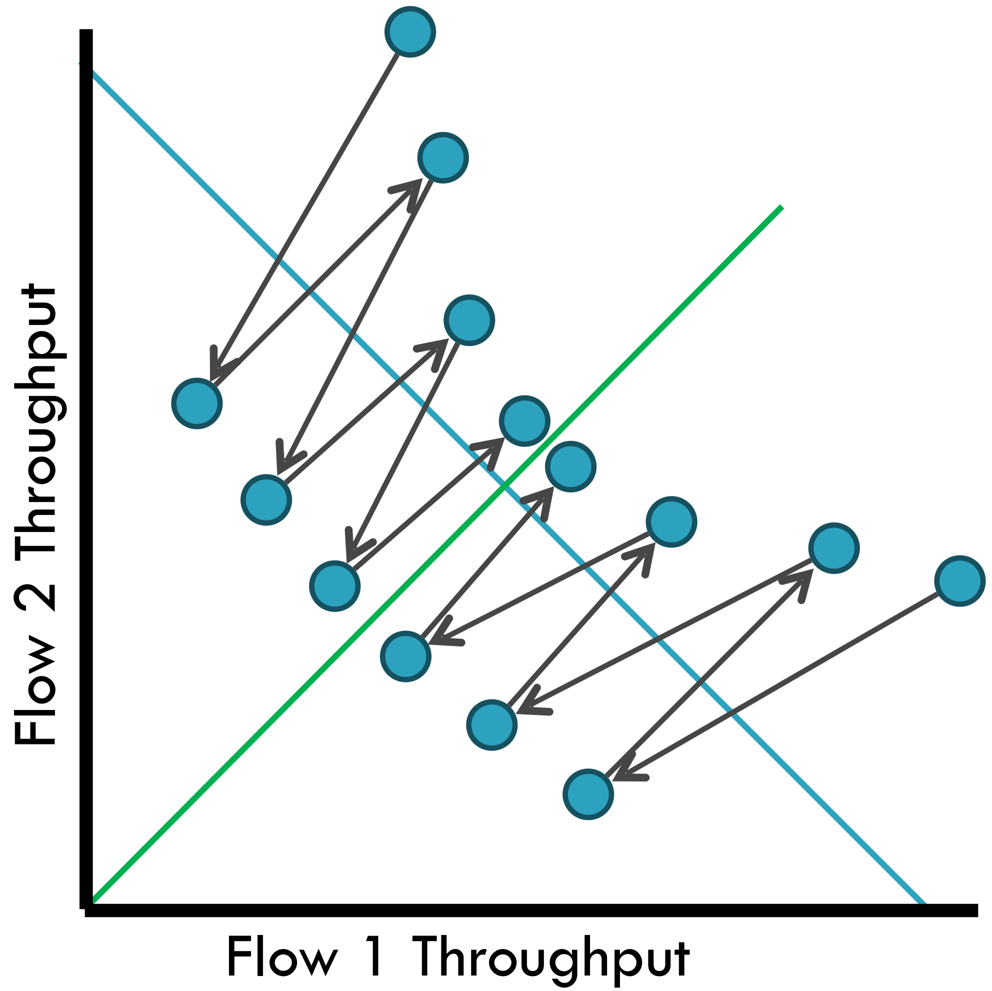
Additive Increase, Multiplicative Decrease

- Converges to stable and fair cycle



Additive Increase, Multiplicative Decrease

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



Implementing Congestion Control

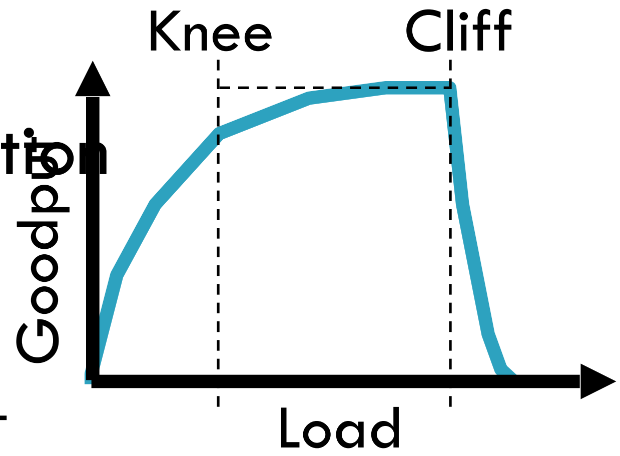
- Maintains three variables:
  - *cwnd*: congestion window
  - *adv_wnd*: receiver advertised window
  - *ssthresh*: threshold size (used to update *cwnd*)
- For sending, use: $wnd = min(cwnd, adv\_wnd)$

Implementing Congestion Control

- Maintains three variables:
  - *cwnd*: congestion window
  - *adv_wnd*: receiver advertised window
  - *ssthresh*: threshold size (used to update *cwnd*)
- For sending, use: *wnd = min(cwnd, adv_wnd)*
- Two phases of congestion control
  1. Slow start (*cwnd* < *ssthresh*)
     - Probe for bottleneck bandwidth
  2. Congestion avoidance (*cwnd* >= *ssthresh*)
     - AIMD

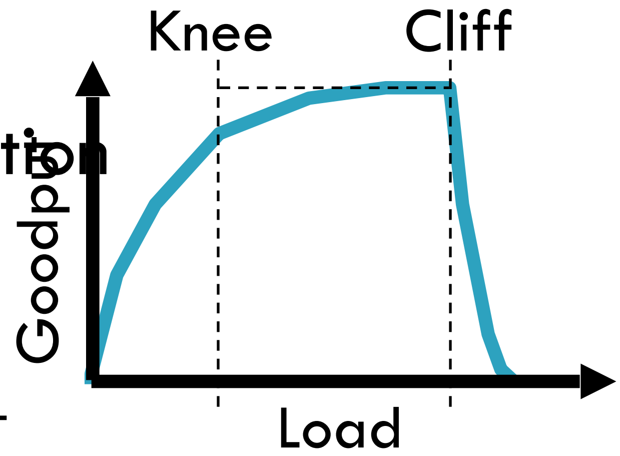
Slow Start

- Goal: reach knee quickly
- Upon starting (or restarting) a connection
  - *cwnd* =1
  - *ssthresh* = *adv_wnd*
  - Each time a segment is ACKed, *cwnd*++



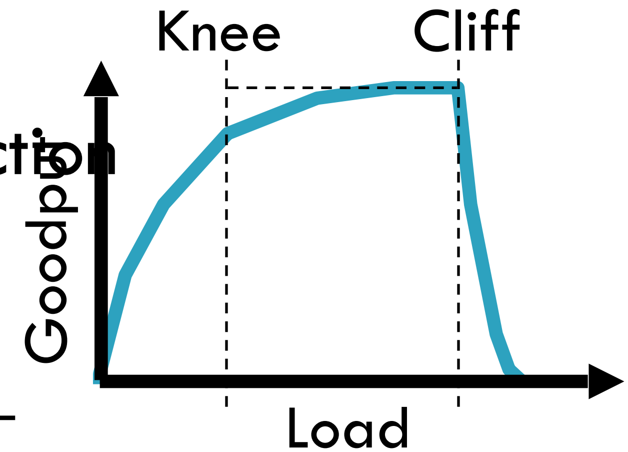
Slow Start

- Goal: reach knee quickly
- Upon starting (or restarting) a connection
  - *cwnd* =1
  - *ssthresh* = *adv_wnd*
  - Each time a segment is ACKed, *cwnd*++
- Continues until…
  - *ssthresh* is reached
  - Or a packet is lost

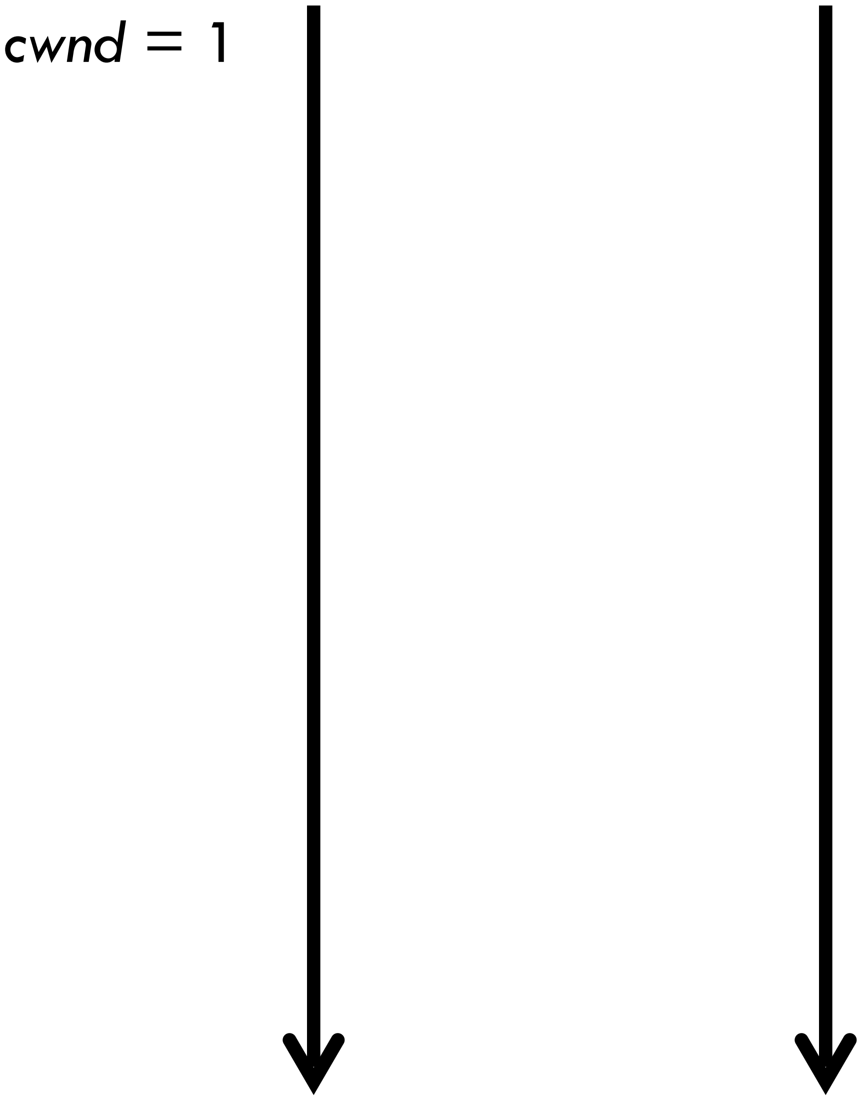


Slow Start

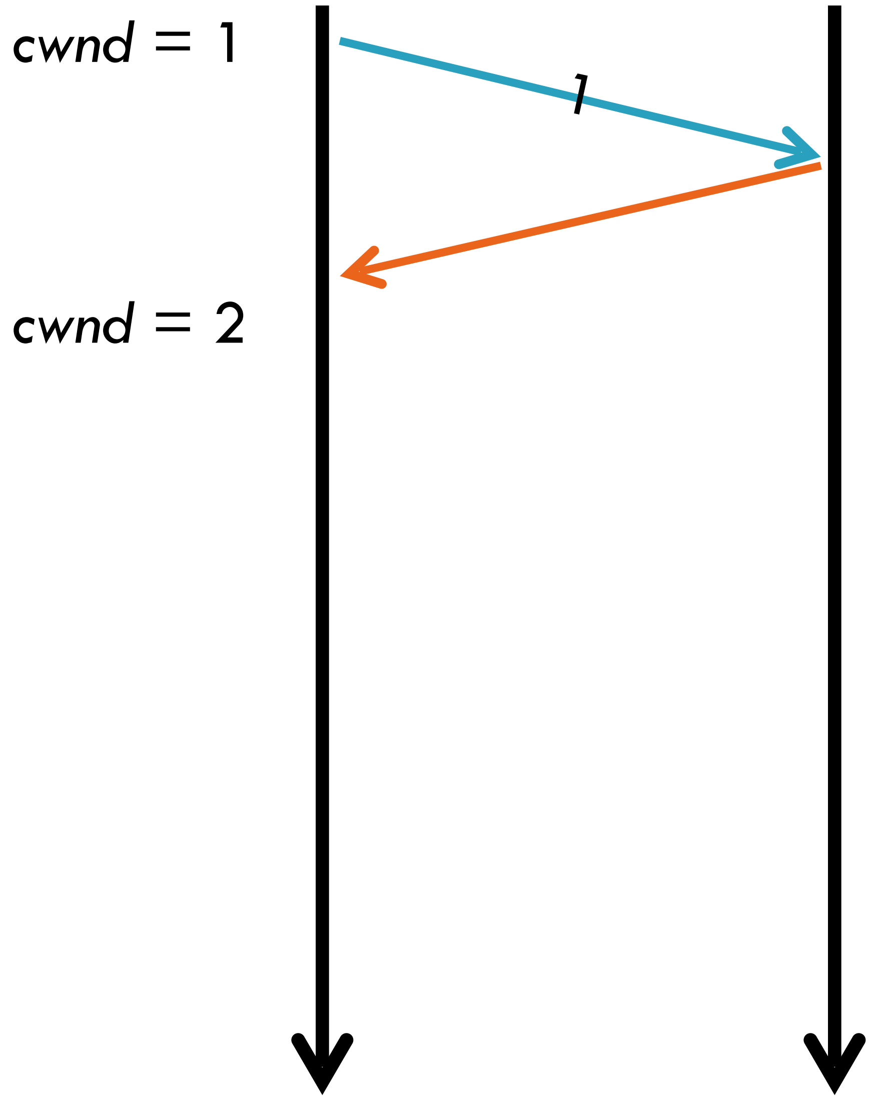
- Goal: reach knee quickly
- Upon starting (or restarting) a connection
  - *cwnd* =1
  - *ssthresh* = *adv_wnd*
  - Each time a segment is ACKed, *cwnd*++
- Continues until…
  - *ssthresh* is reached
  - Or a packet is lost
- Slow Start is not actually slow
  - *cwnd* increases exponentially



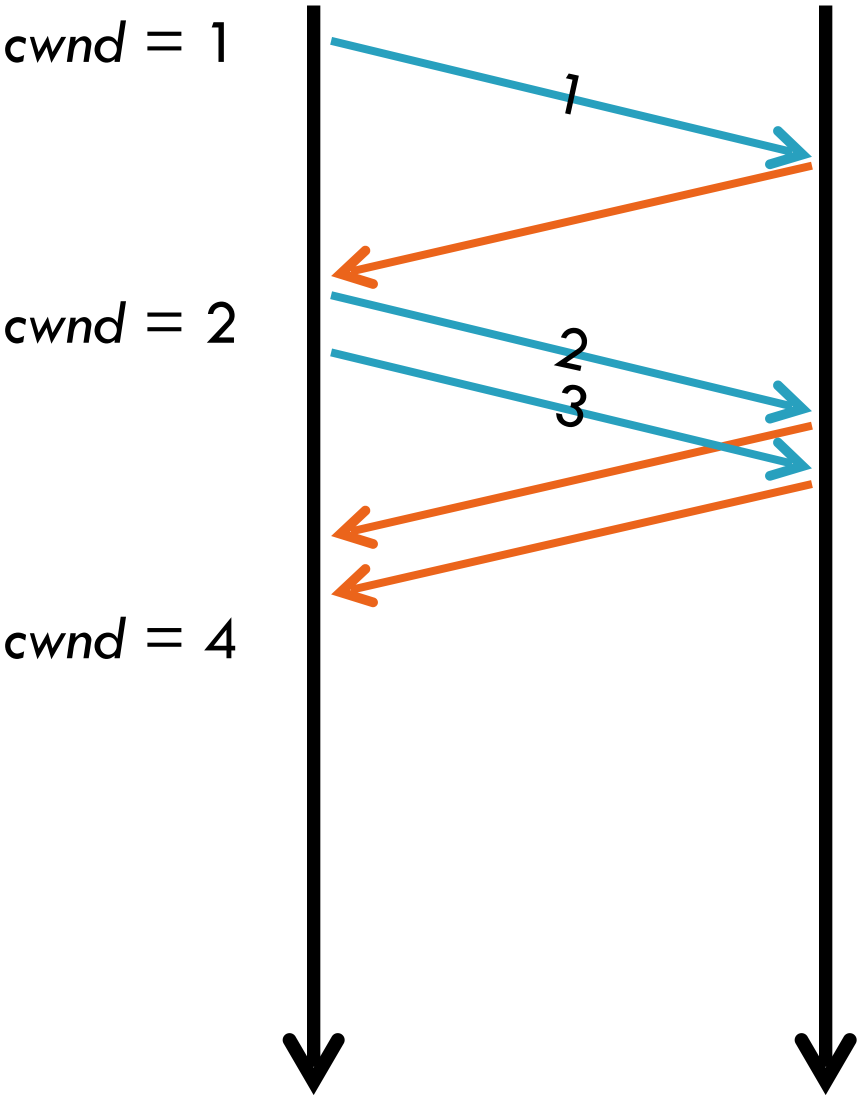
Slow Start Example



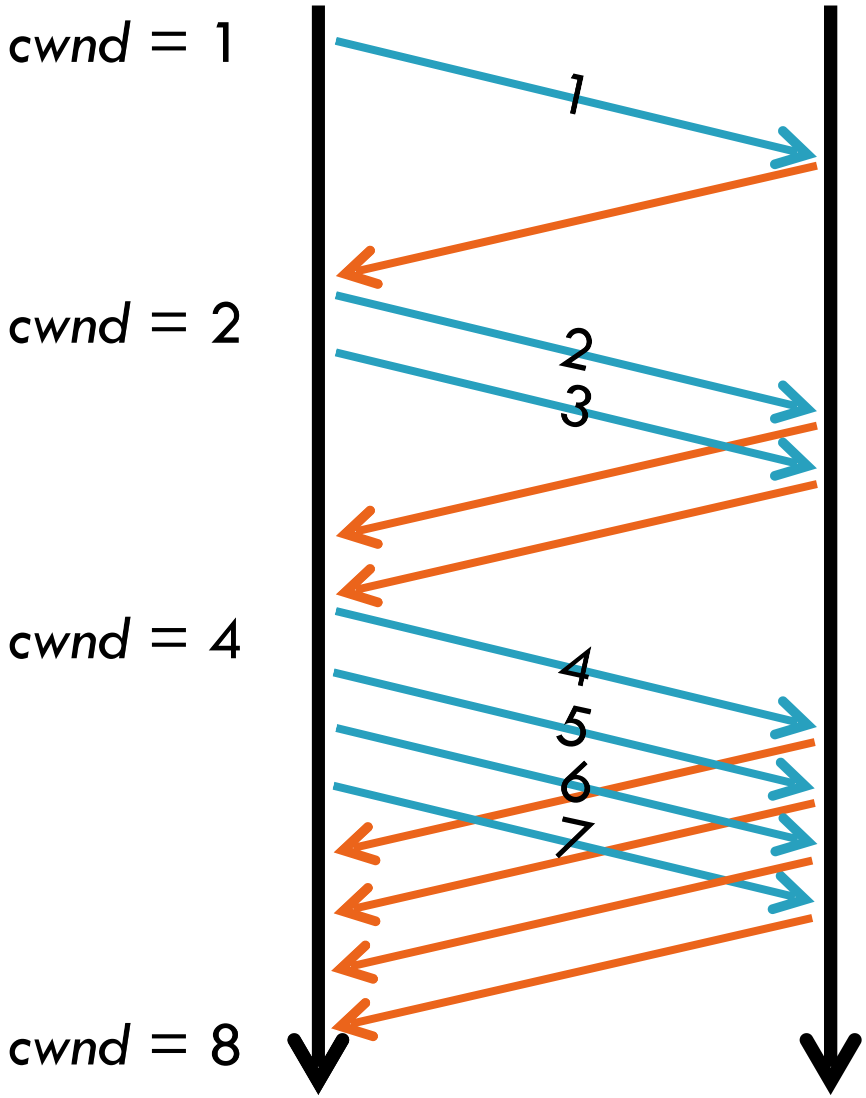
Slow Start Example



Slow Start Example

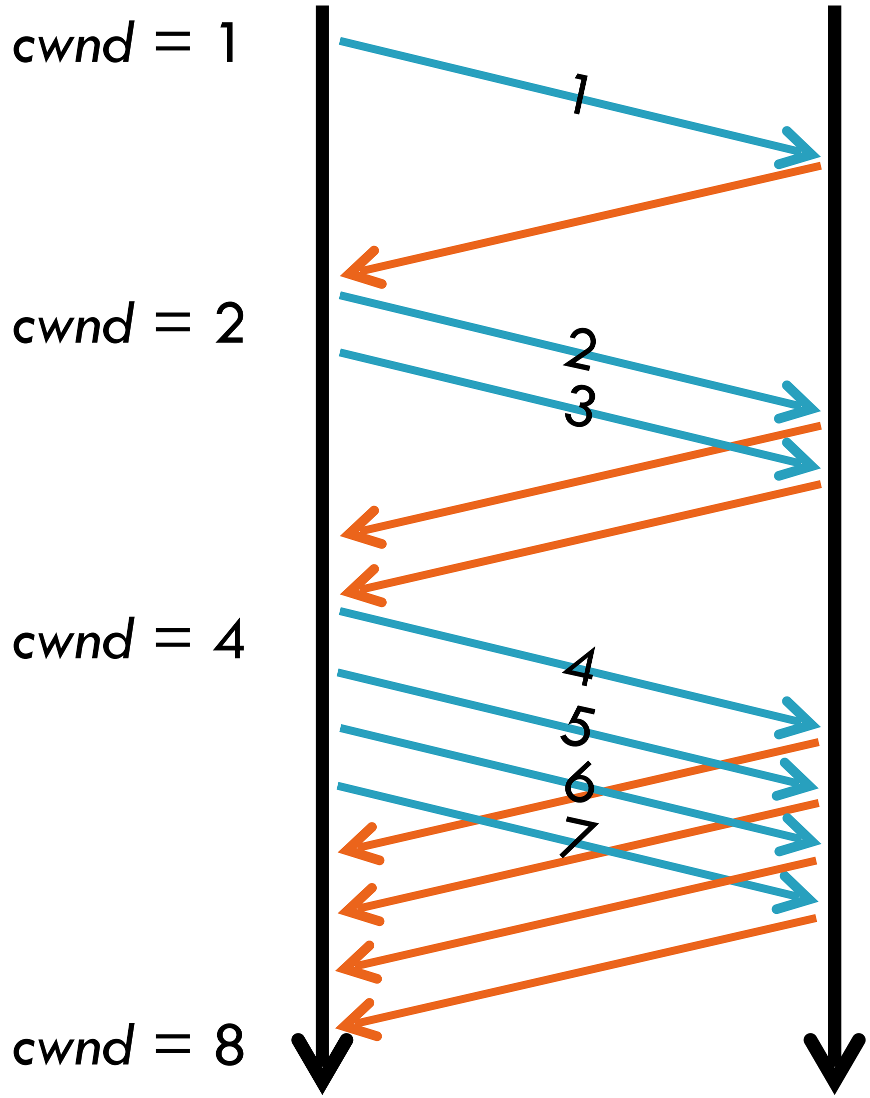


Slow Start Example



Slow Start Example

- *cwnd* grows rapidly
- Slows down when…
  - *cwnd* >= *ssthresh*
  - Or a packet drops



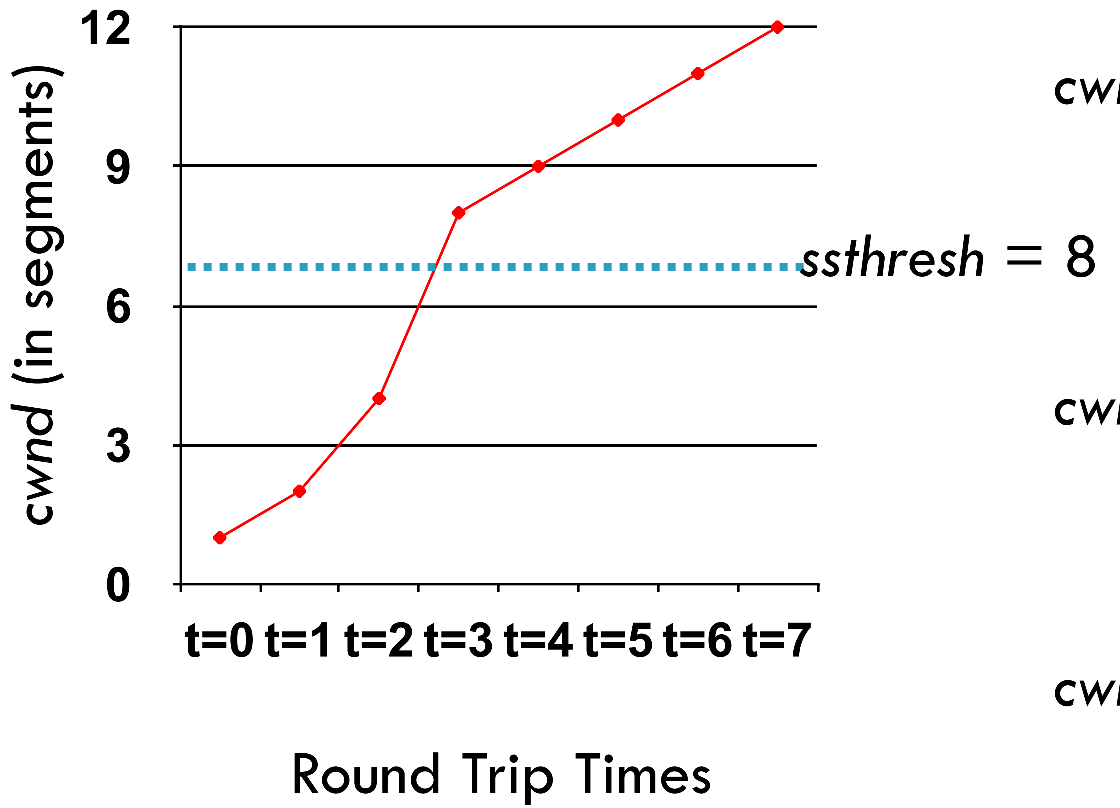
Congestion Avoidance

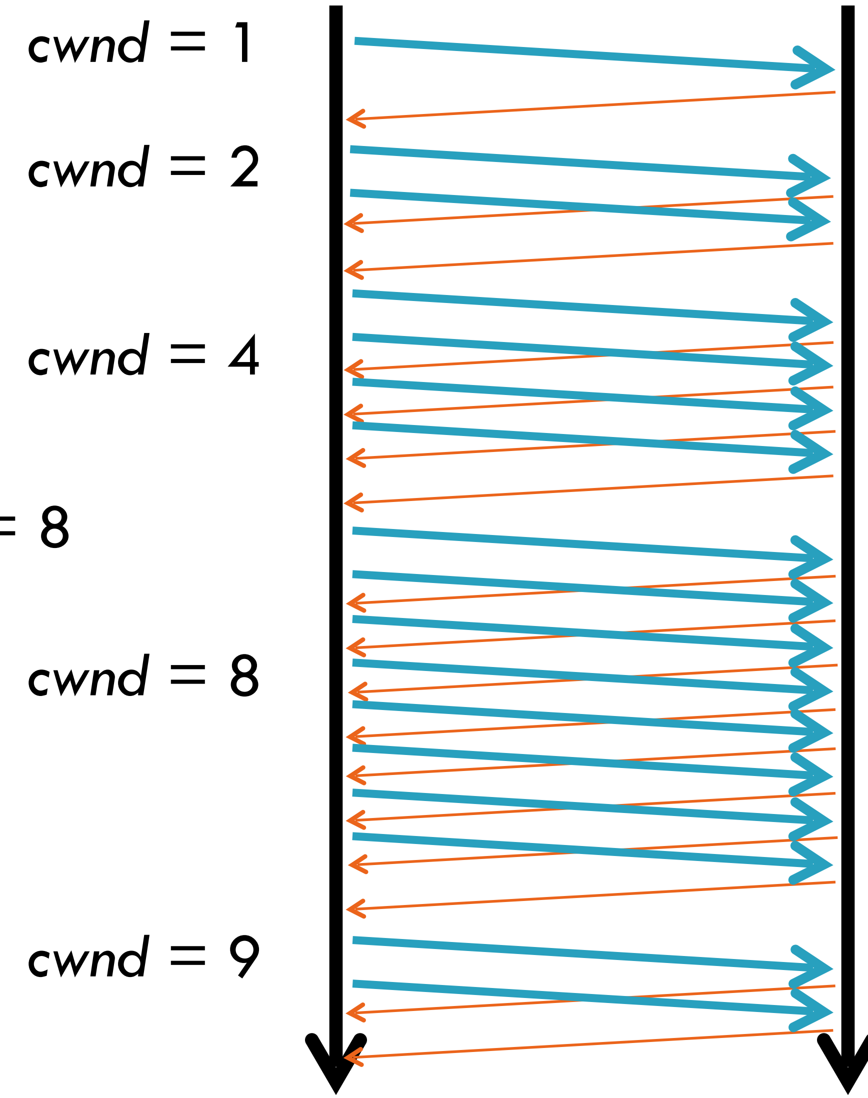
- AIMD mode
- *ssthresh* is lower-bound guess about location of the knee
- **If** *cwnd* >= *ssthresh* **then**

  each time a segment is ACKed

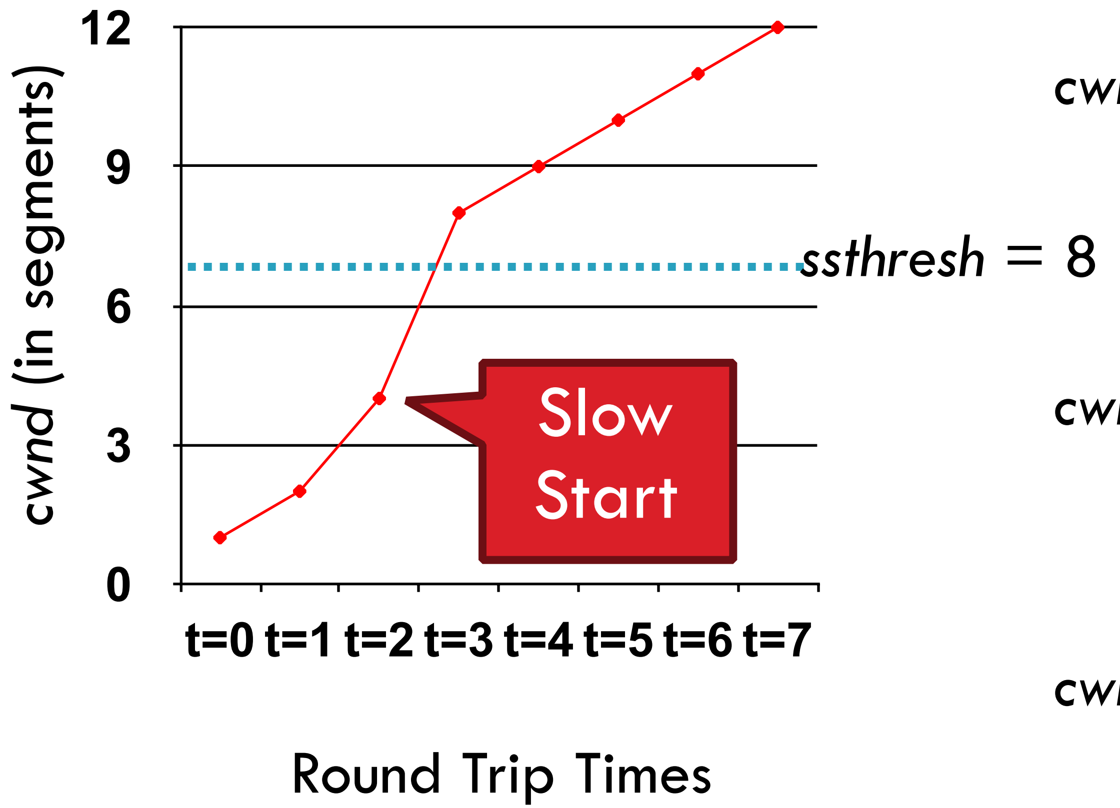
  increment *cwnd by 1/cwnd* (*cwnd* += *1/cwnd*).
- So *cwnd* is increased by one only if all segments have been acknowledged

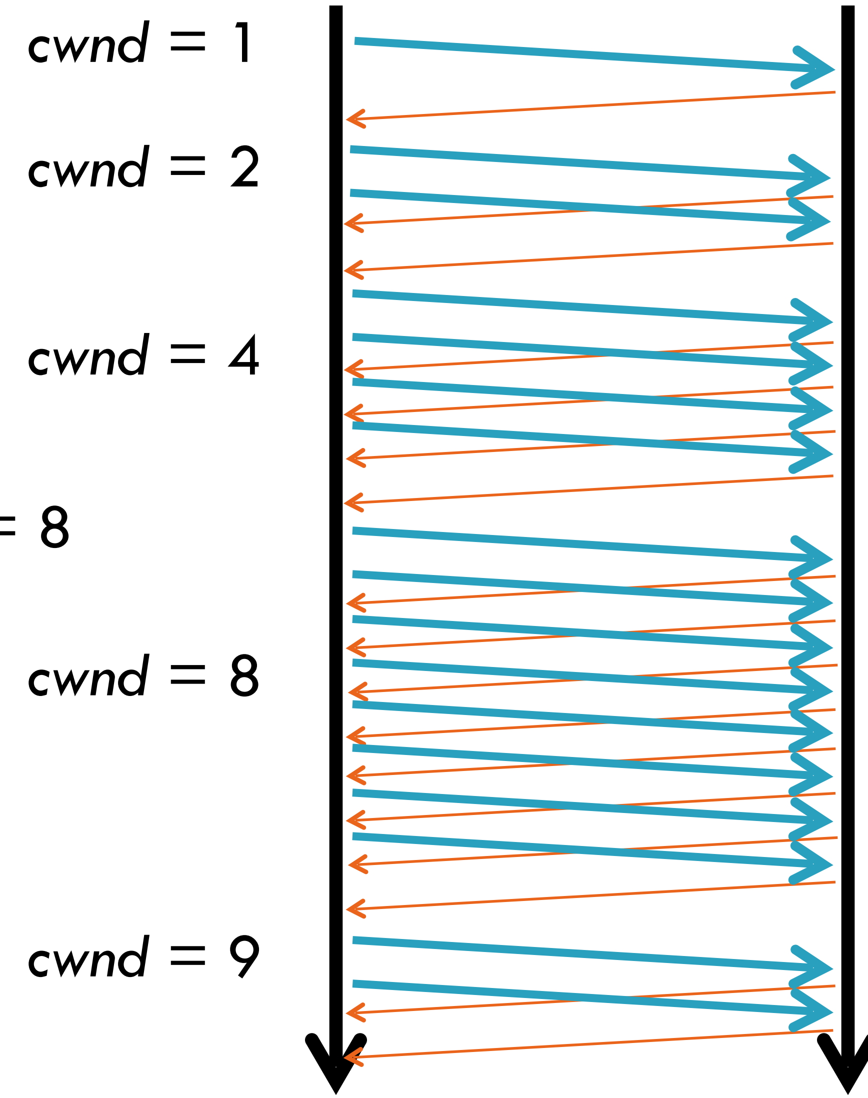
Congestion Avoidance Example



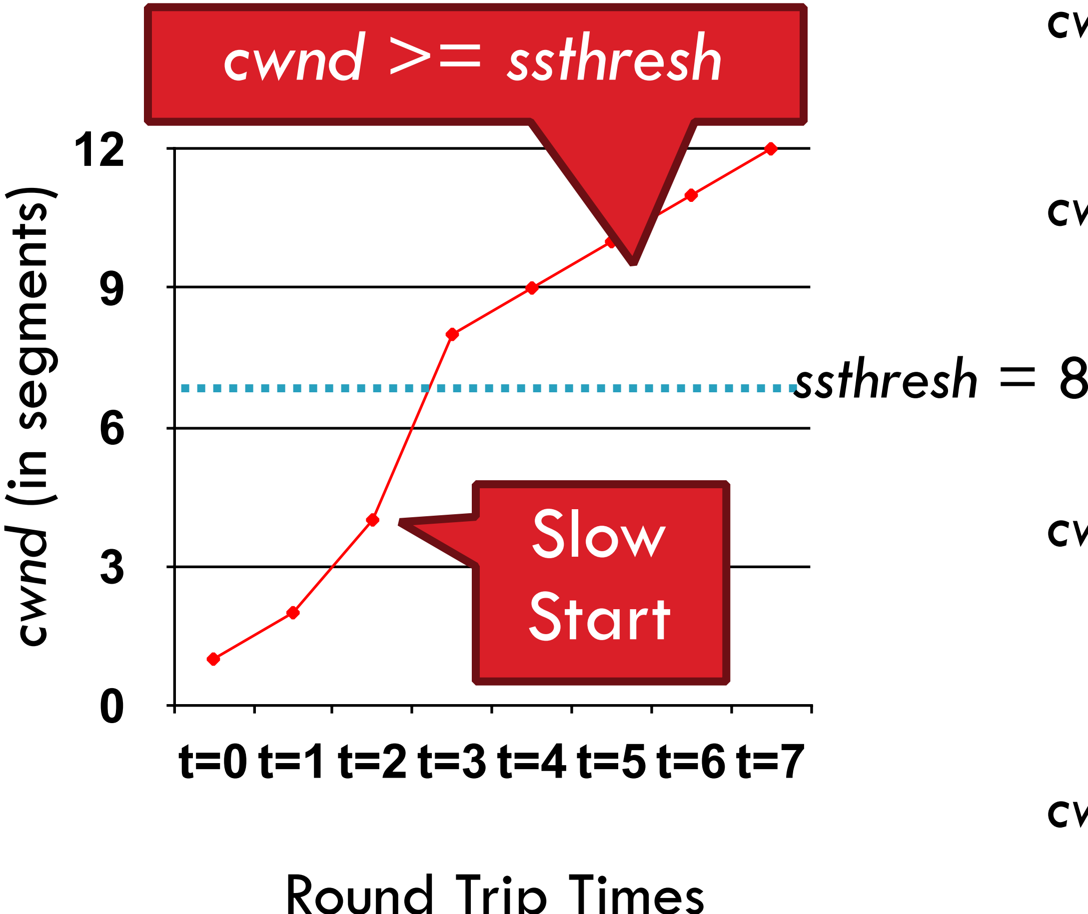


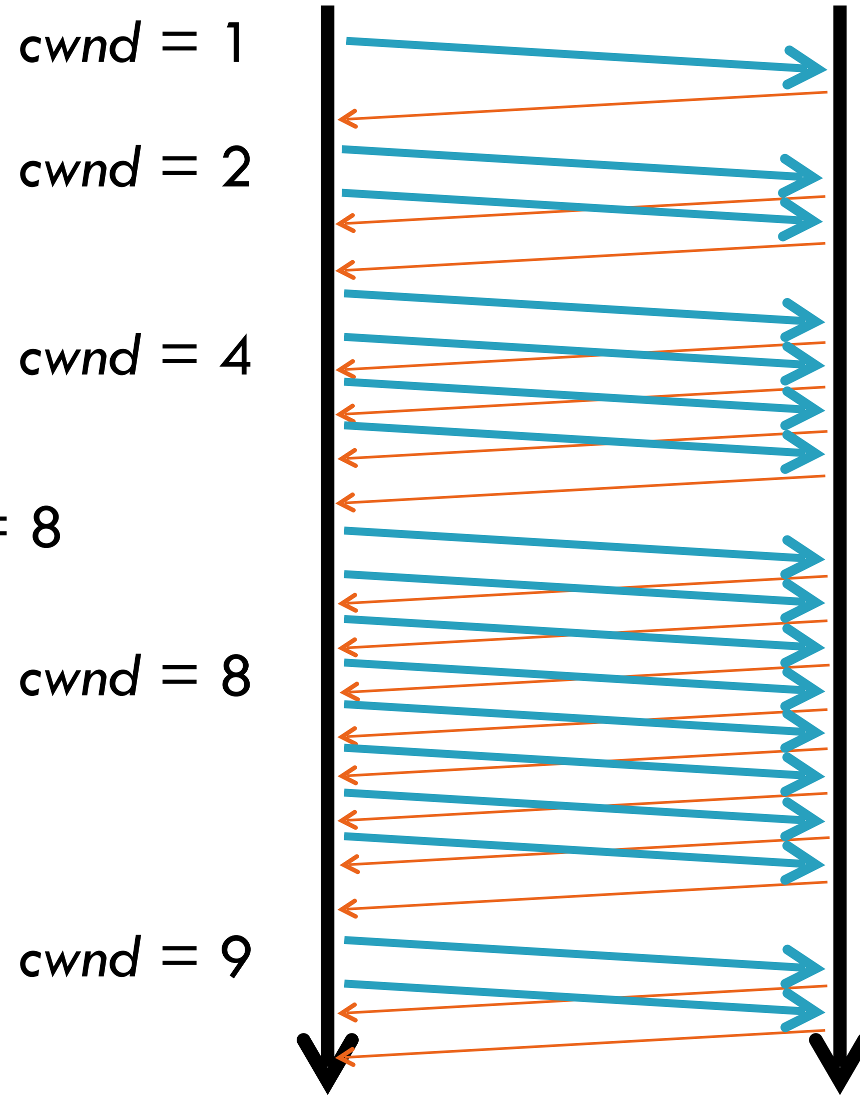
Congestion Avoidance Example





Congestion Avoidance Example

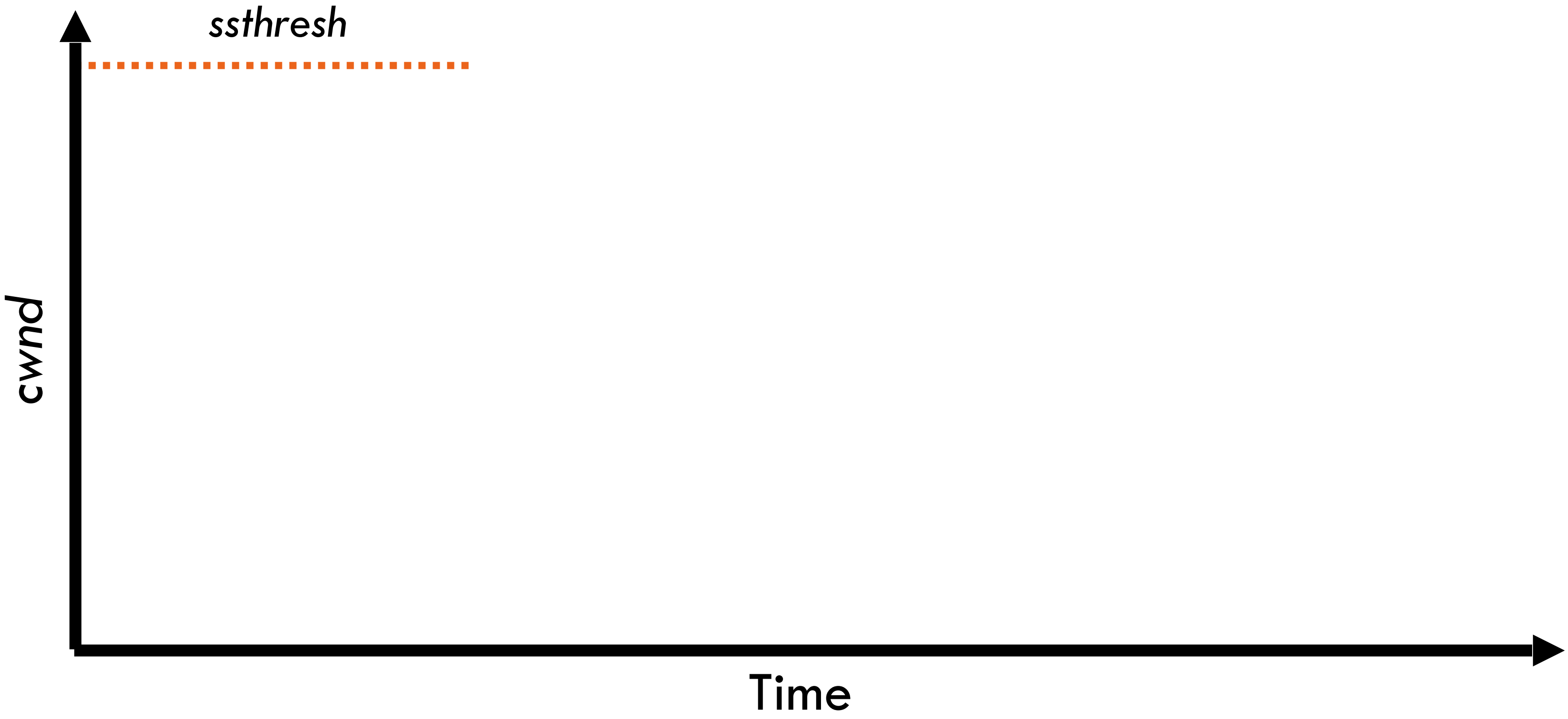




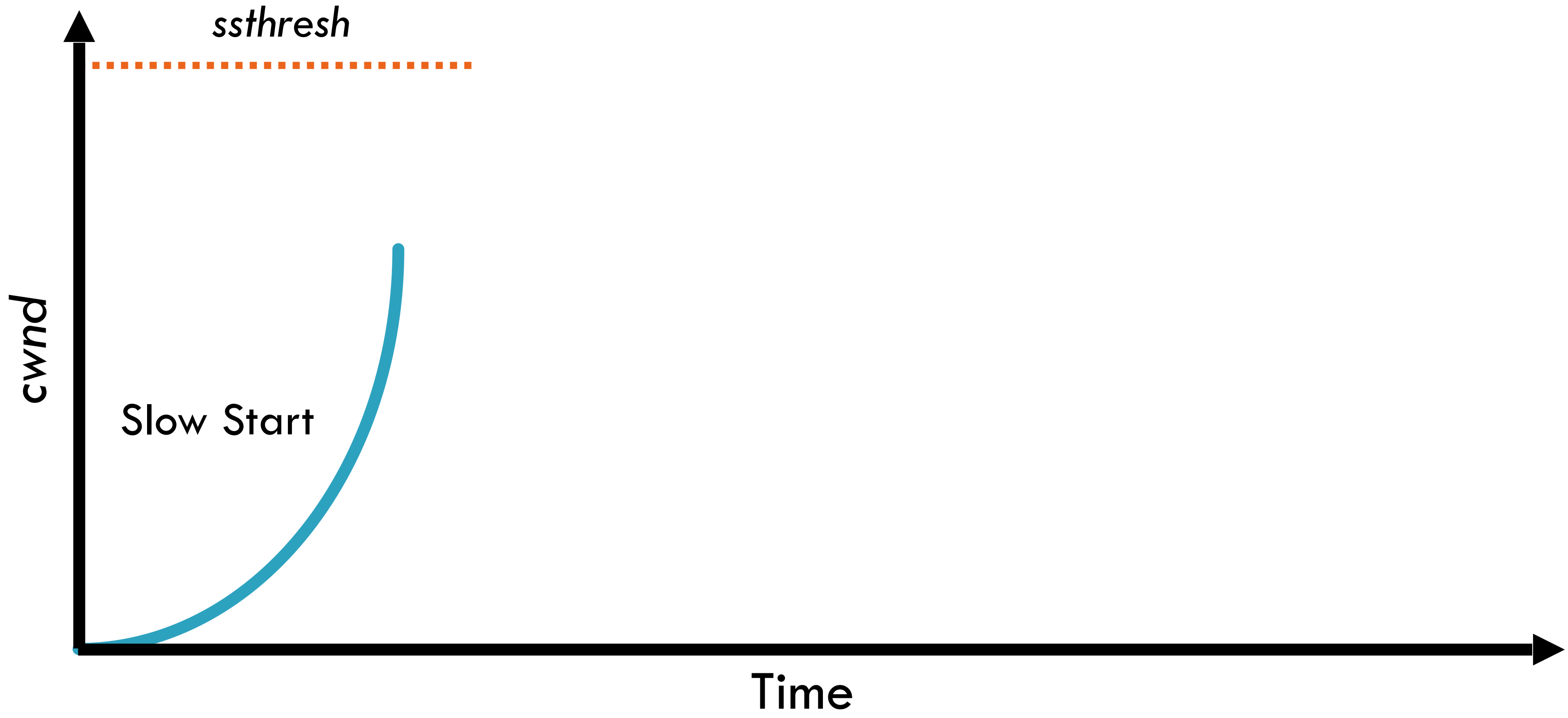
TCP Pseudocode


```
Initially:
        cwnd = 1;
        ssthresh = adv_wnd;
New ack received:
        if (cwnd < ssthresh)
            /* Slow Start*/
            cwnd = cwnd + 1;
        else
            /* Congestion Avoidance */
            cwnd = cwnd + 1/cwnd;
Timeout:
        /* Multiplicative decrease */
        ssthresh = cwnd/2;
        cwnd = 1;
```

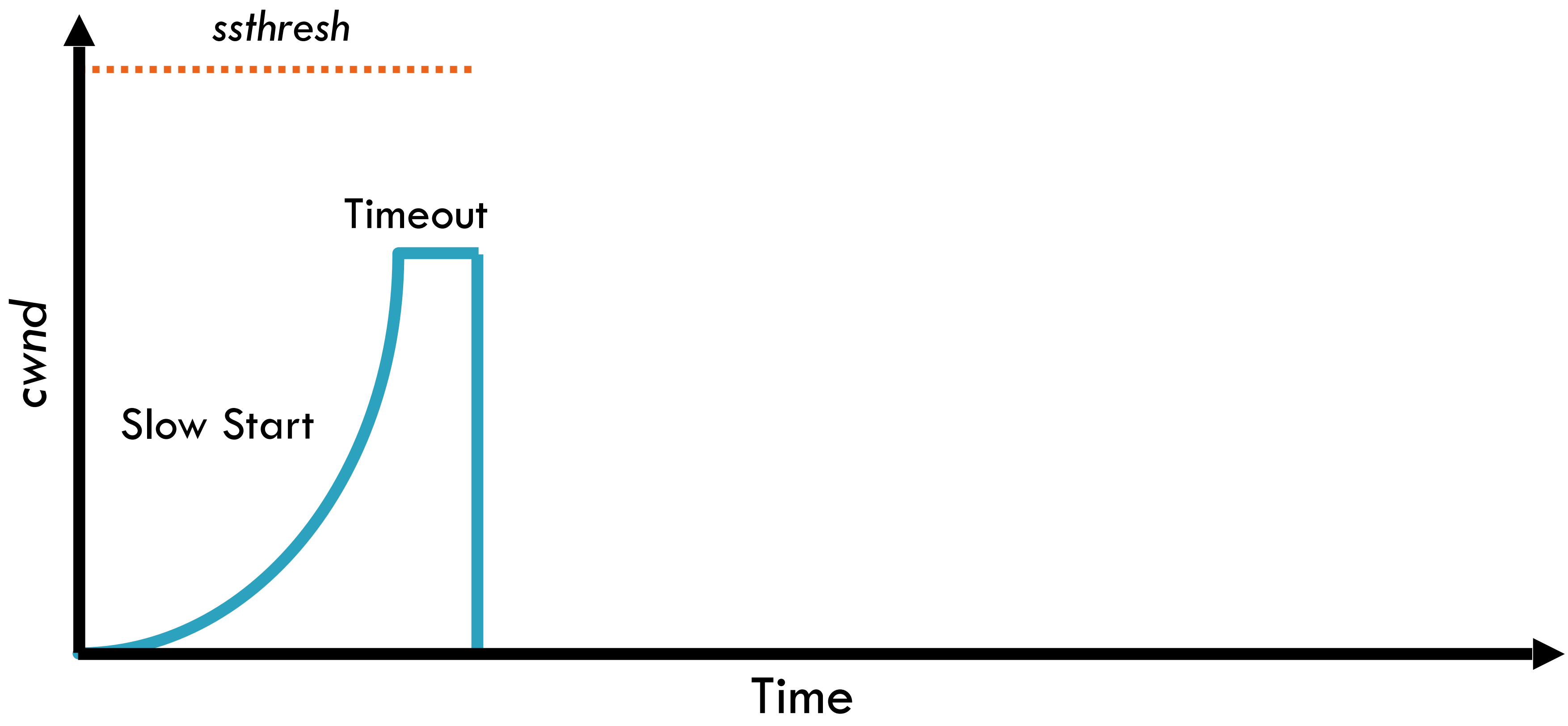
The Big Picture



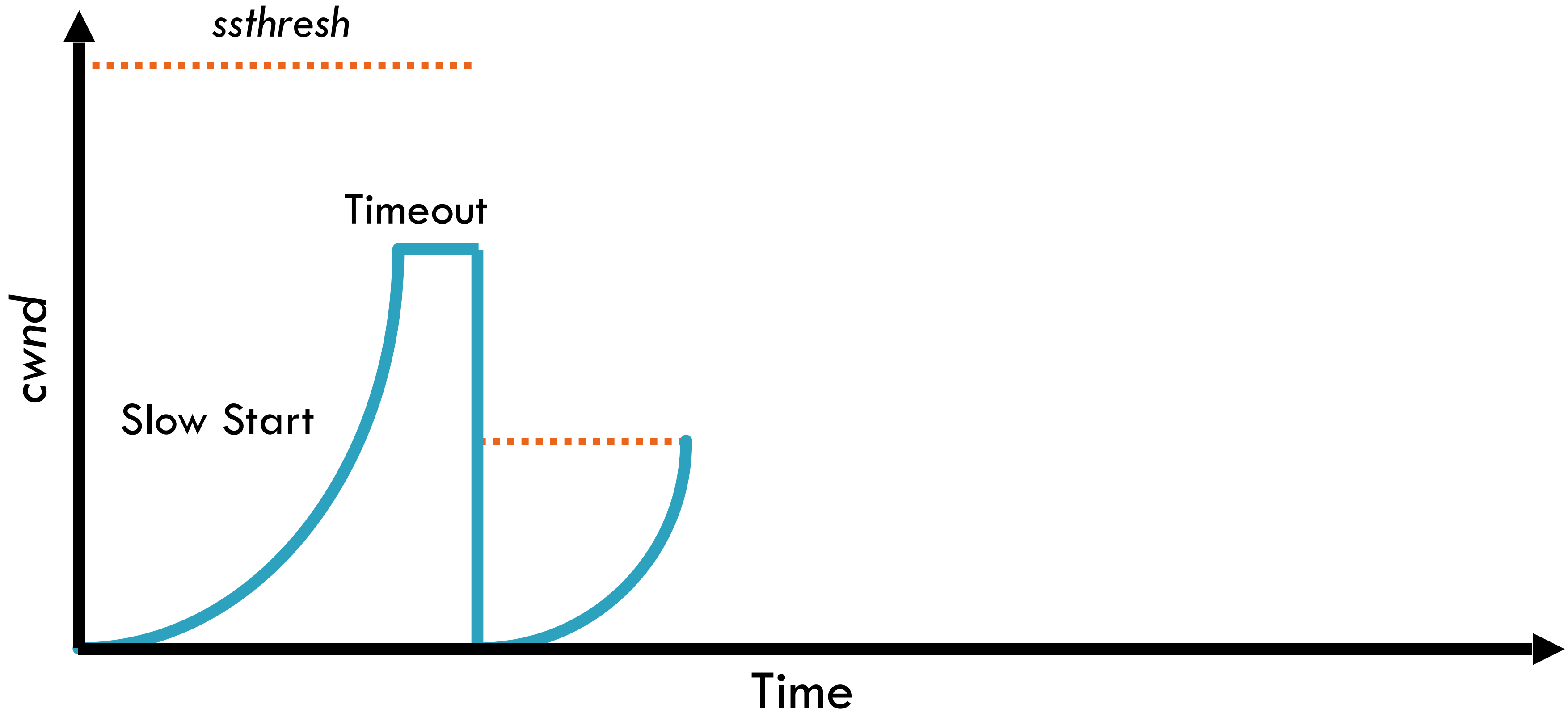
The Big Picture



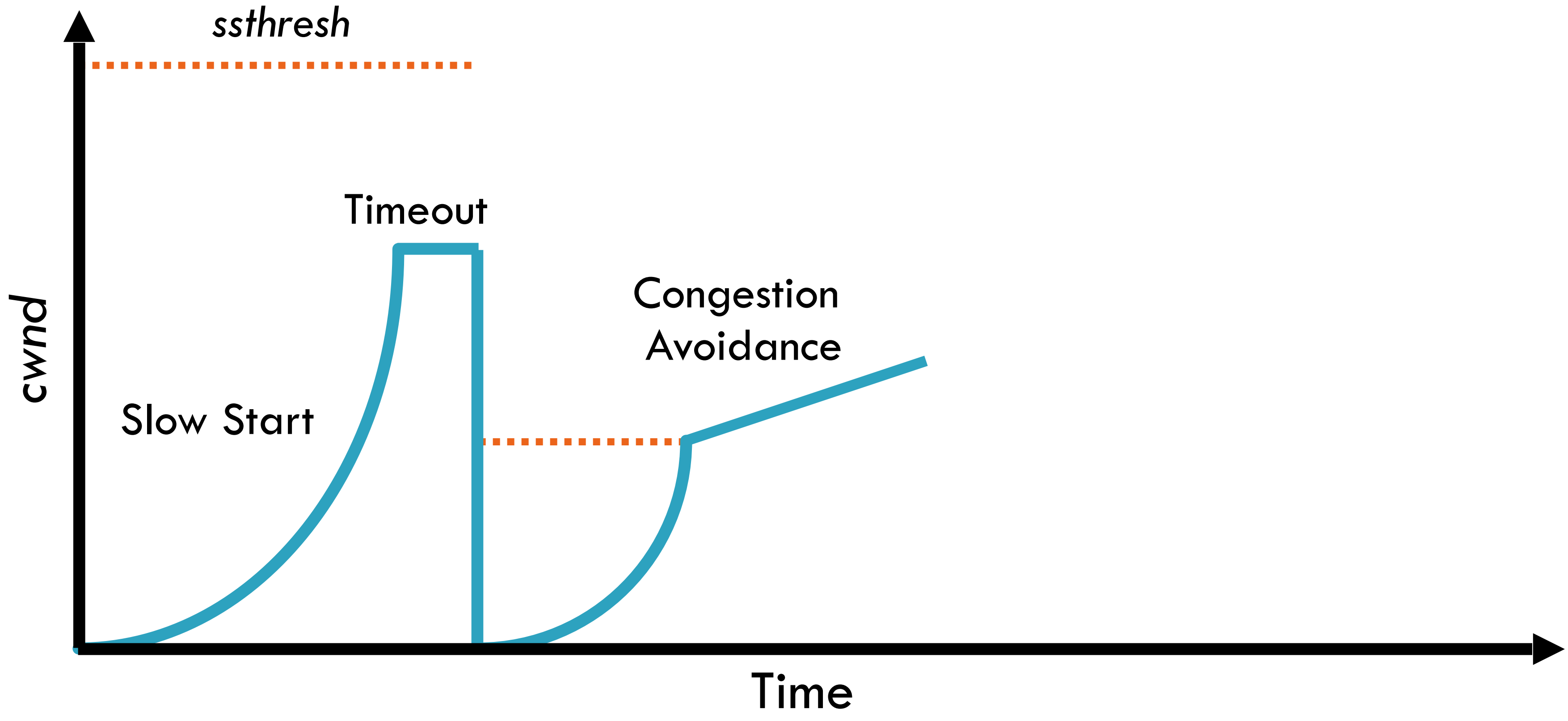
The Big Picture



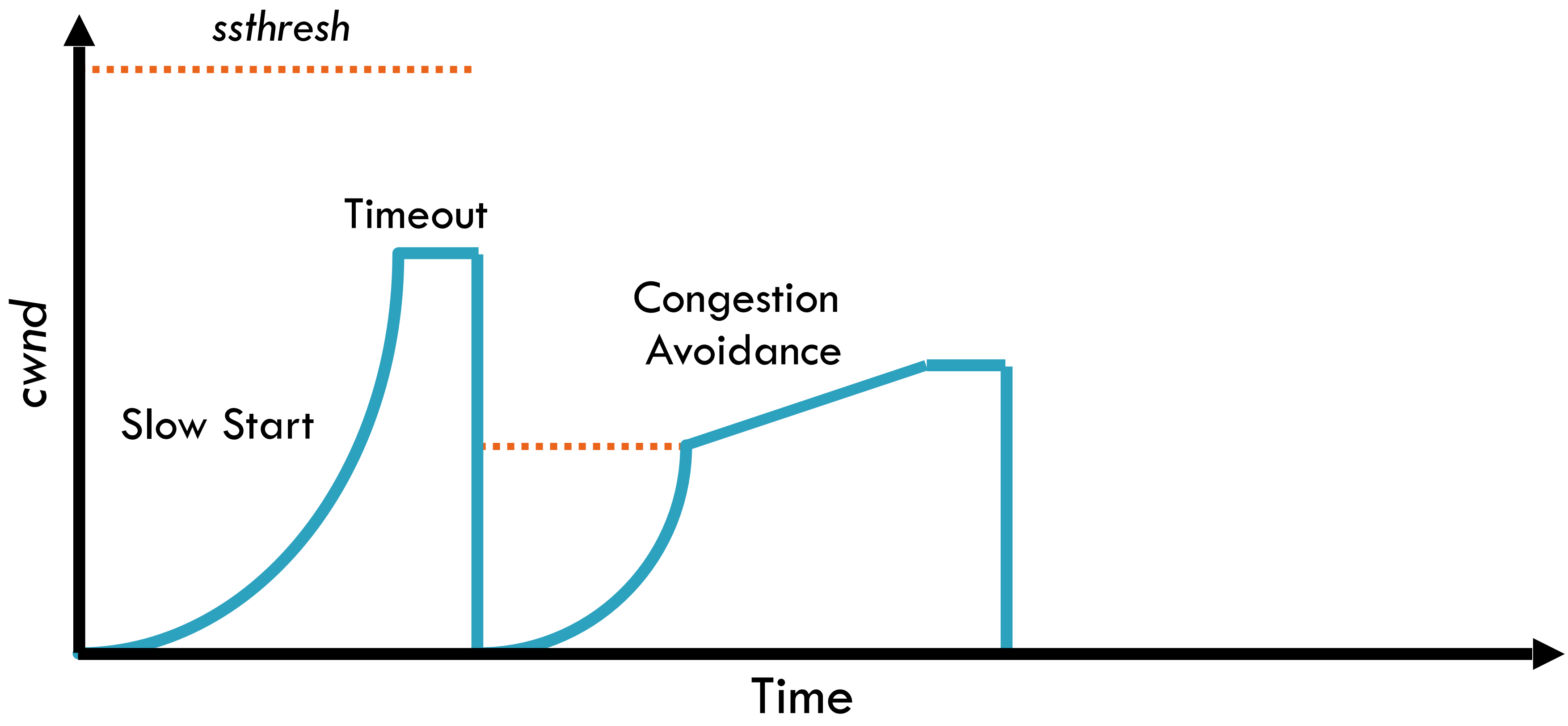
The Big Picture



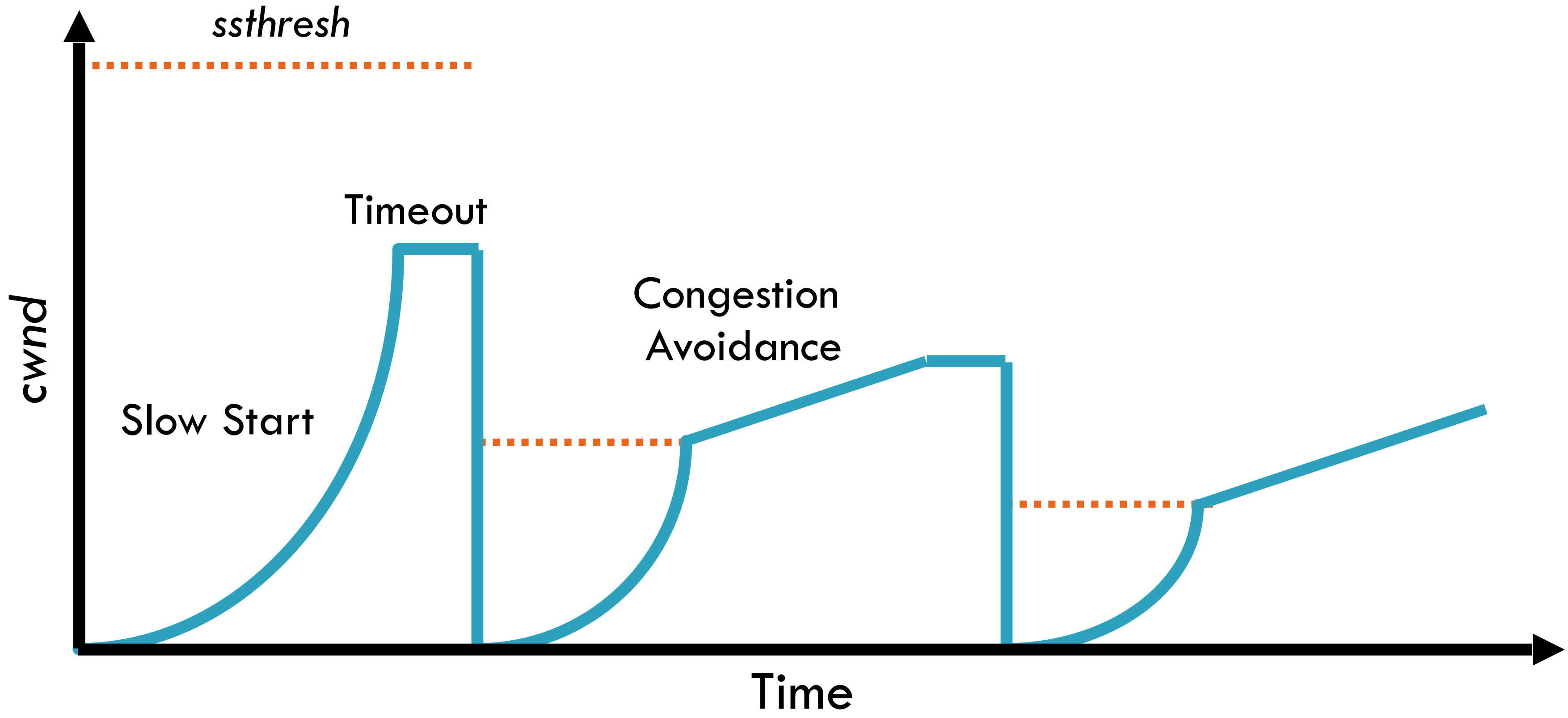
The Big Picture



The Big Picture



The Big Picture



Outline

- Congestion Control
- Evolution of TCP
- Problems with TCP

The Evolution of TCP

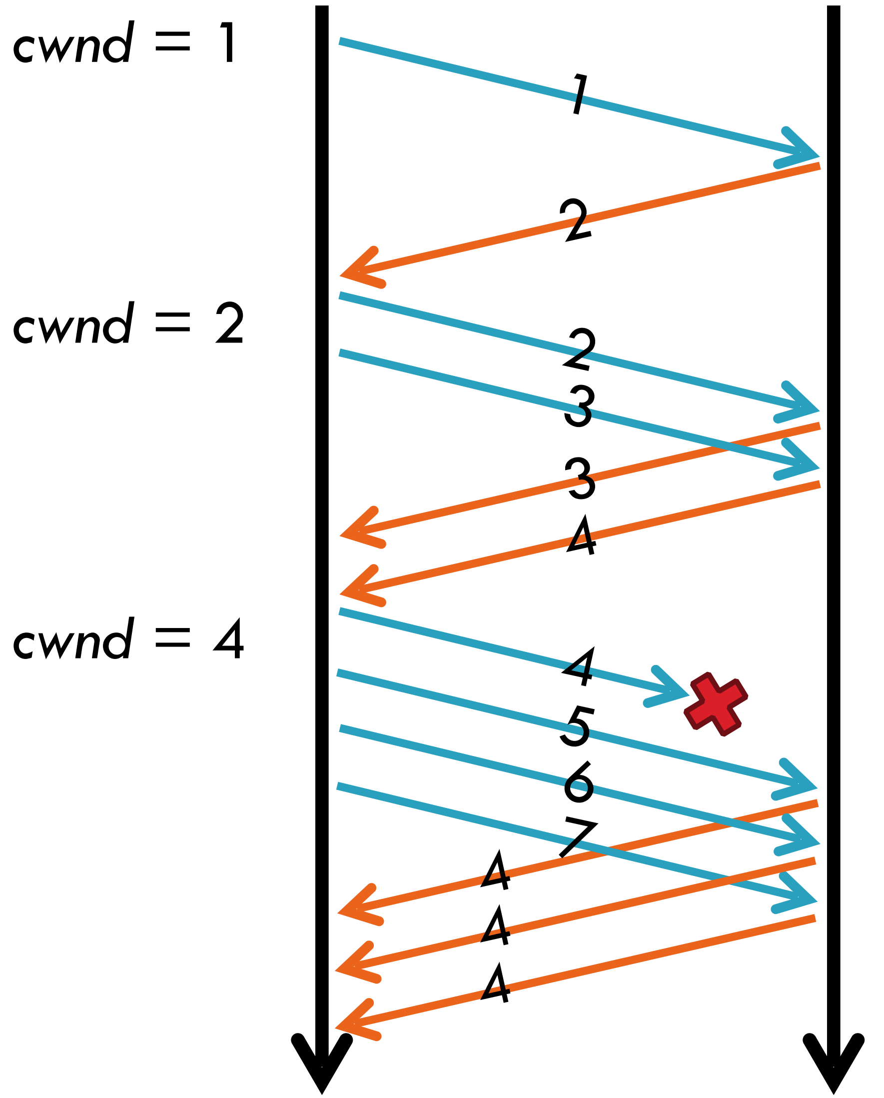
- Thus far, we have discussed TCP Tahoe
  - Original version of TCP
- However, TCP was invented in 1974!
  - Today, there are many variants of TCP

The Evolution of TCP

- Thus far, we have discussed TCP Tahoe
  - Original version of TCP
- However, TCP was invented in 1974!
  - Today, there are many variants of TCP
- Early, popular variant: TCP Reno
  - Tahoe features, plus…
  - Fast retransmit
  - Fast recovery

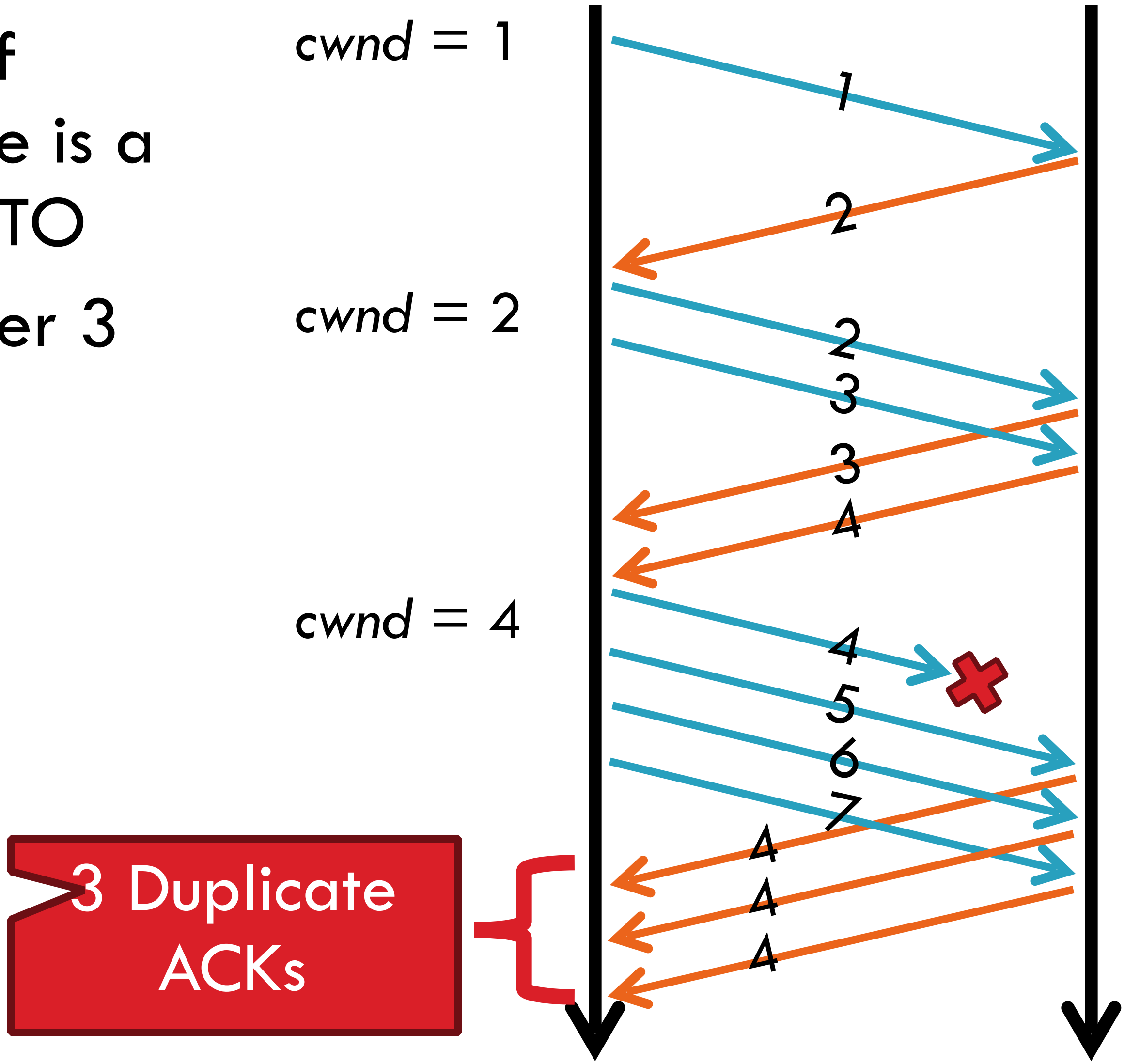
TCP Reno: Fast Retransmit

- Problem: in Tahoe, if segment is lost, there is a long wait until the RTO
- Reno: retransmit after 3 duplicate ACKs



TCP Reno: Fast Retransmit

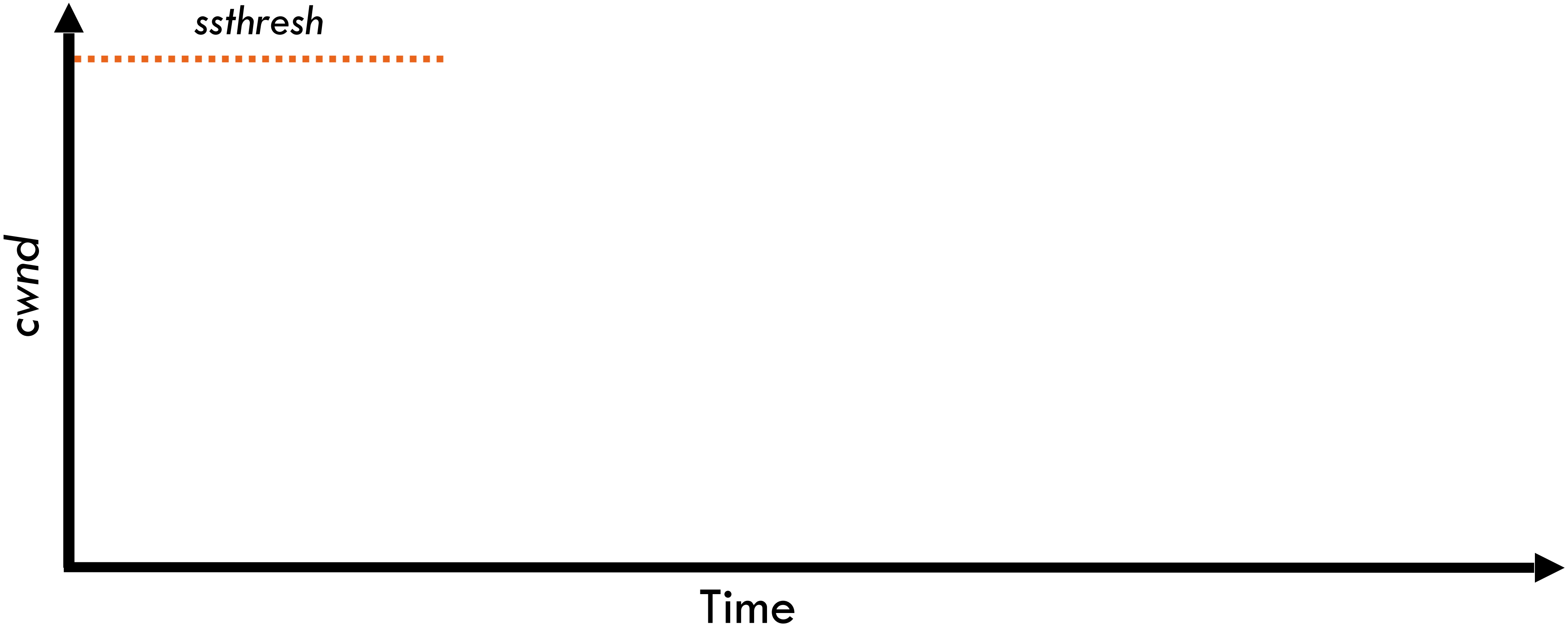
- Problem: in Tahoe, if segment is lost, there is a long wait until the RTO
- Reno: retransmit after 3 duplicate ACKs



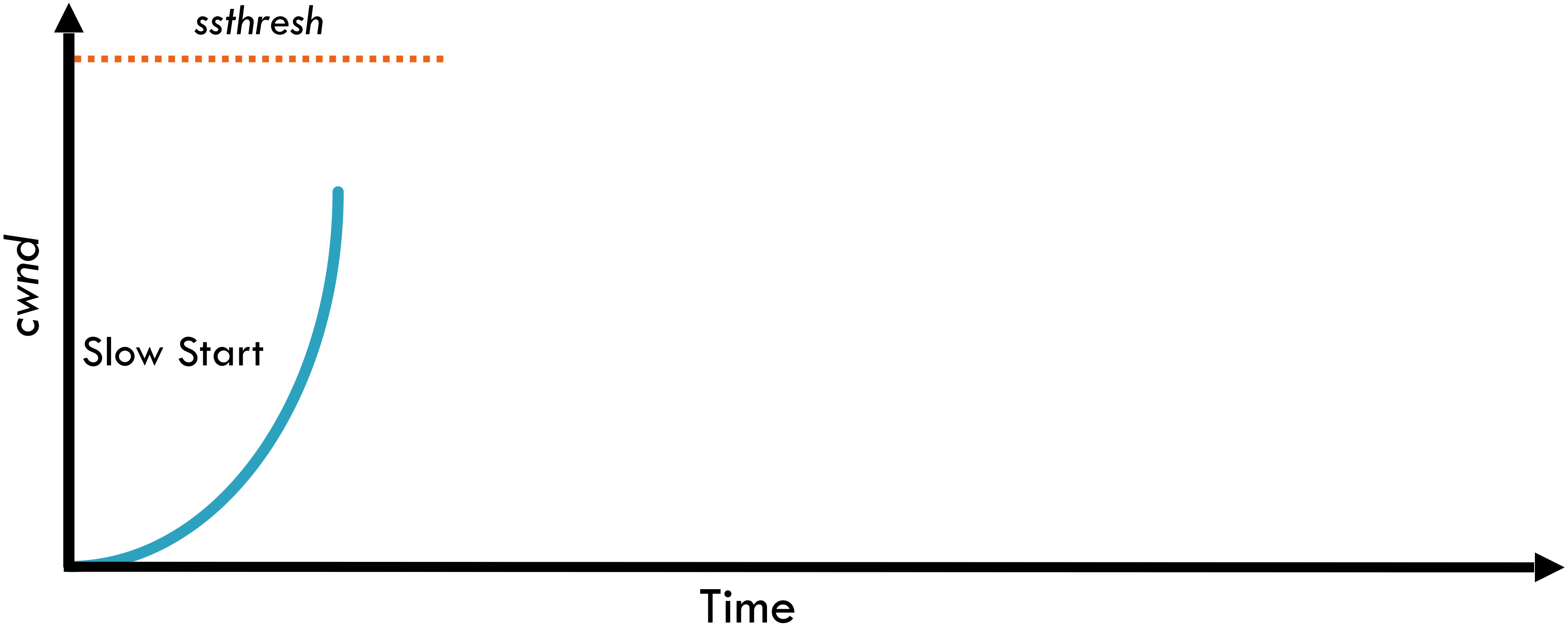
TCP Reno: Fast Recovery

- After a fast-retransmit set *cwnd* to *ssthresh/2*
  - i.e. don't reset *cwnd* to 1
  - Avoid unnecessary return to slow start
  - Prevents expensive timeouts
- But when RTO expires still do *cwnd* = 1
  - Return to slow start, same as Tahoe
  - Indicates packets aren't being delivered at all
  - i.e. congestion must be really bad

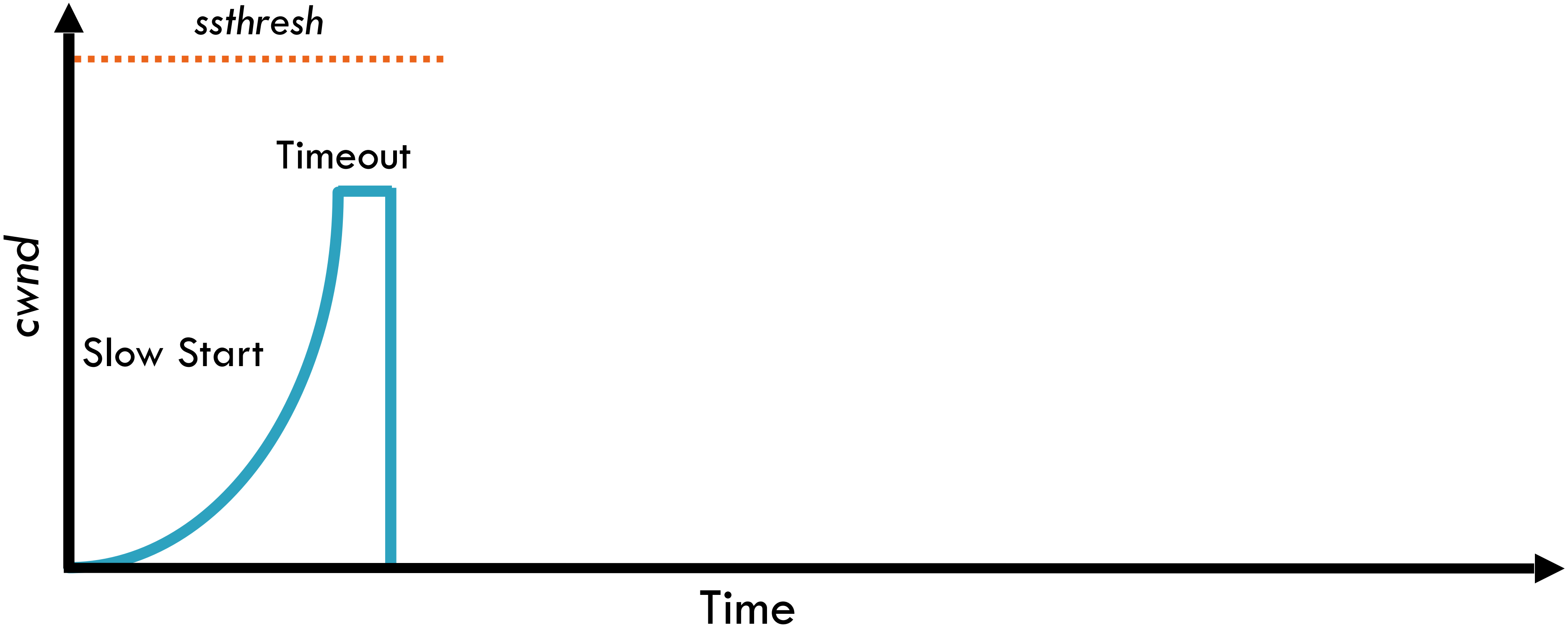
Fast Retransmit and Fast Recovery



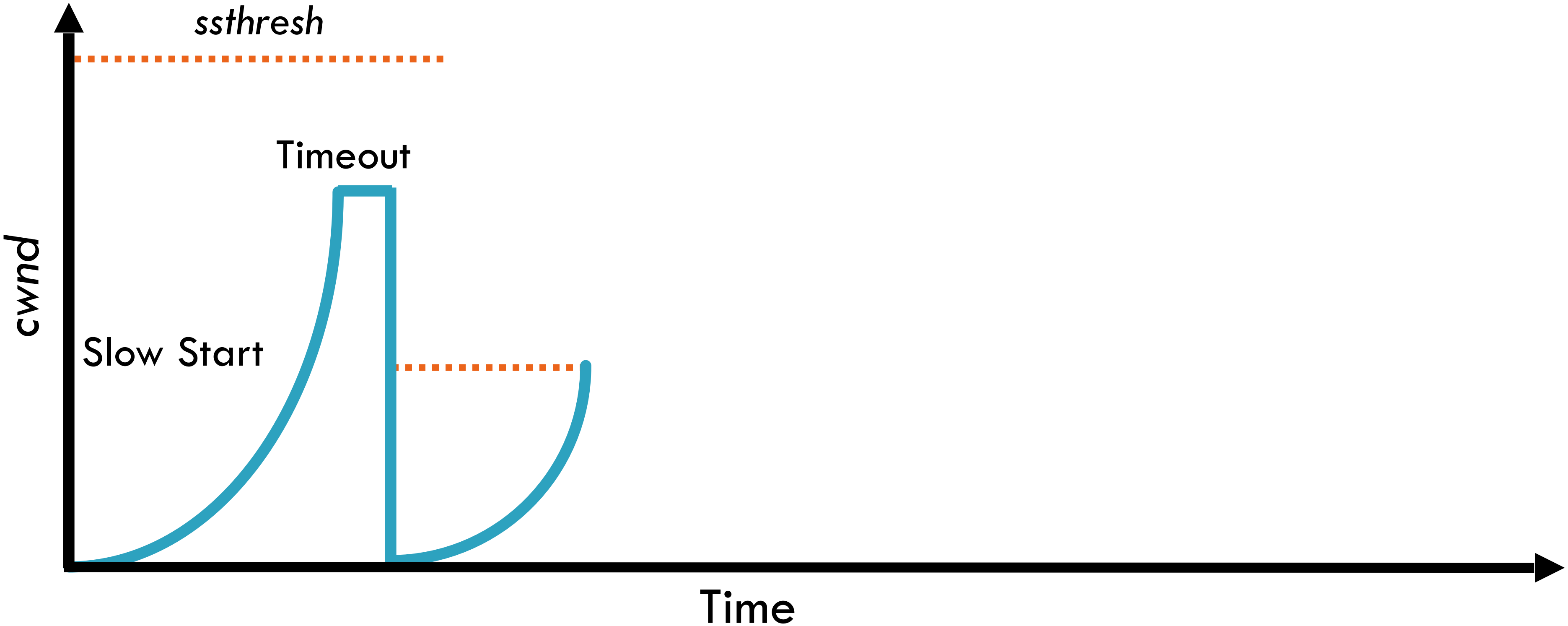
Fast Retransmit and Fast Recovery



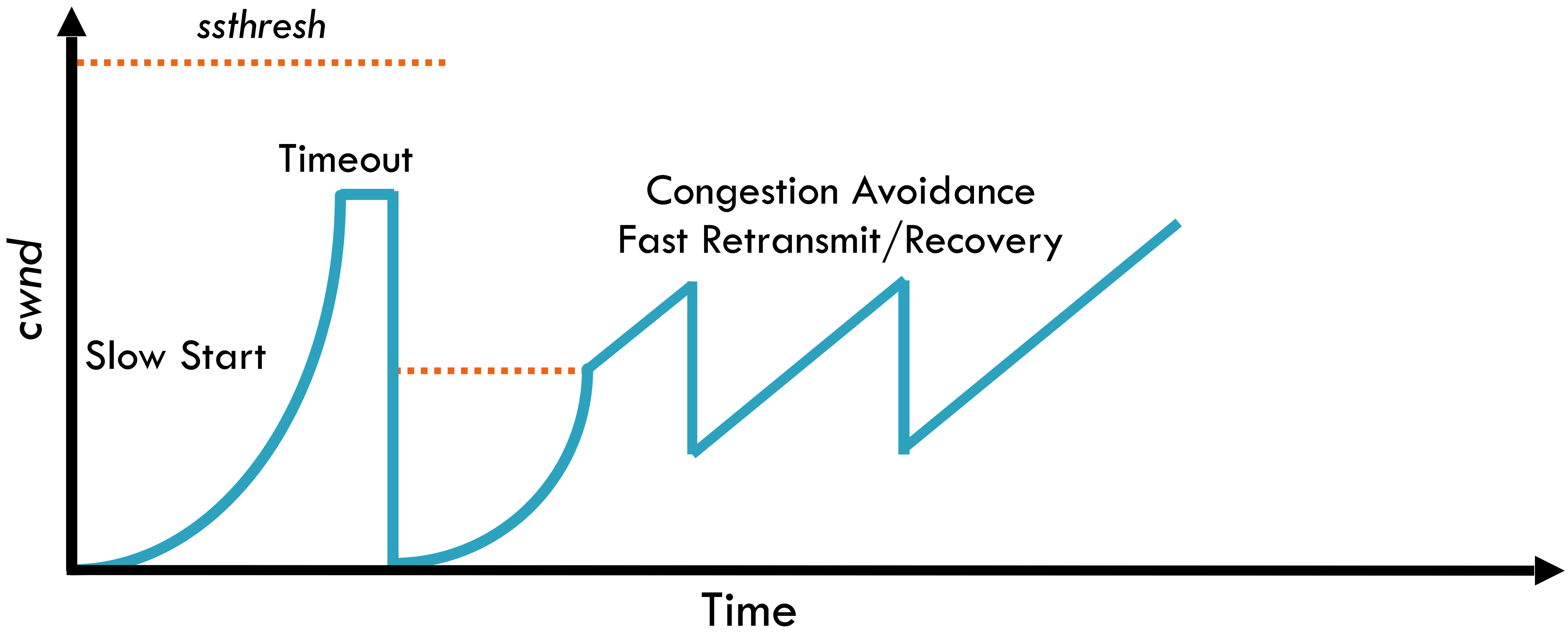
Fast Retransmit and Fast Recovery



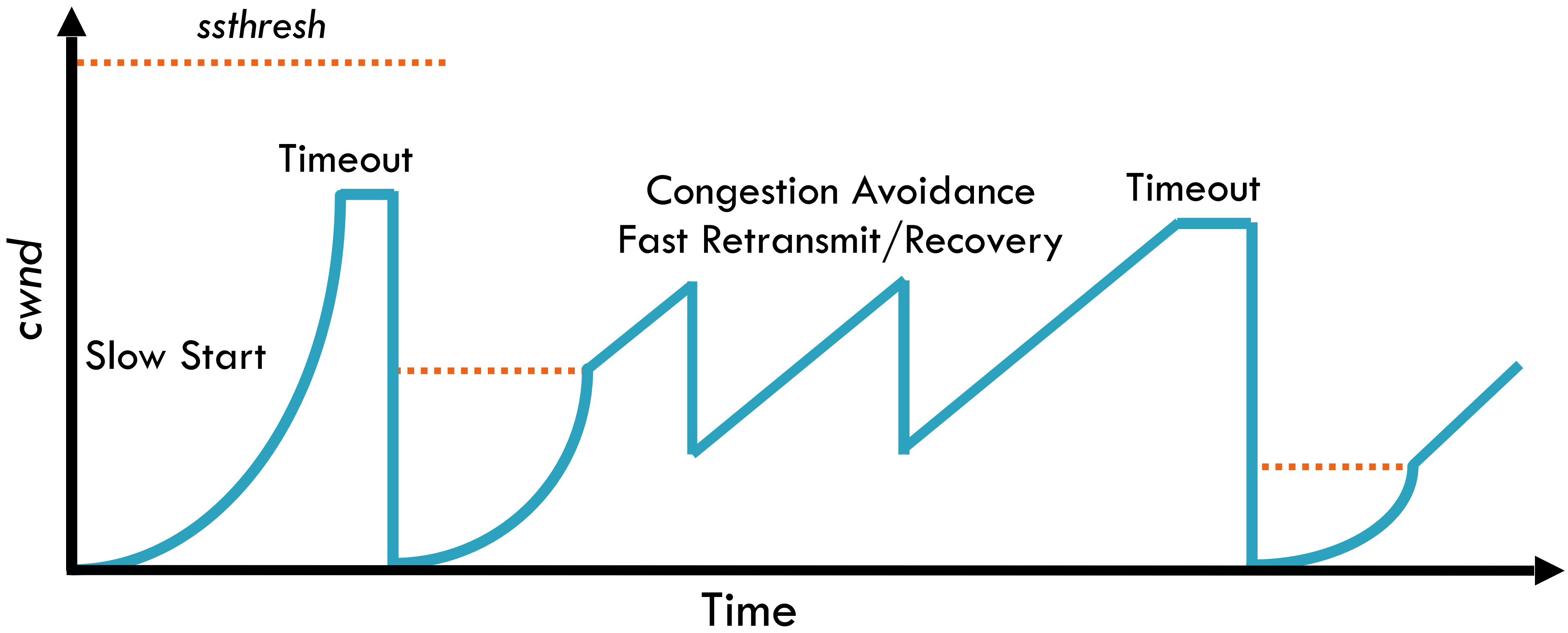
Fast Retransmit and Fast Recovery



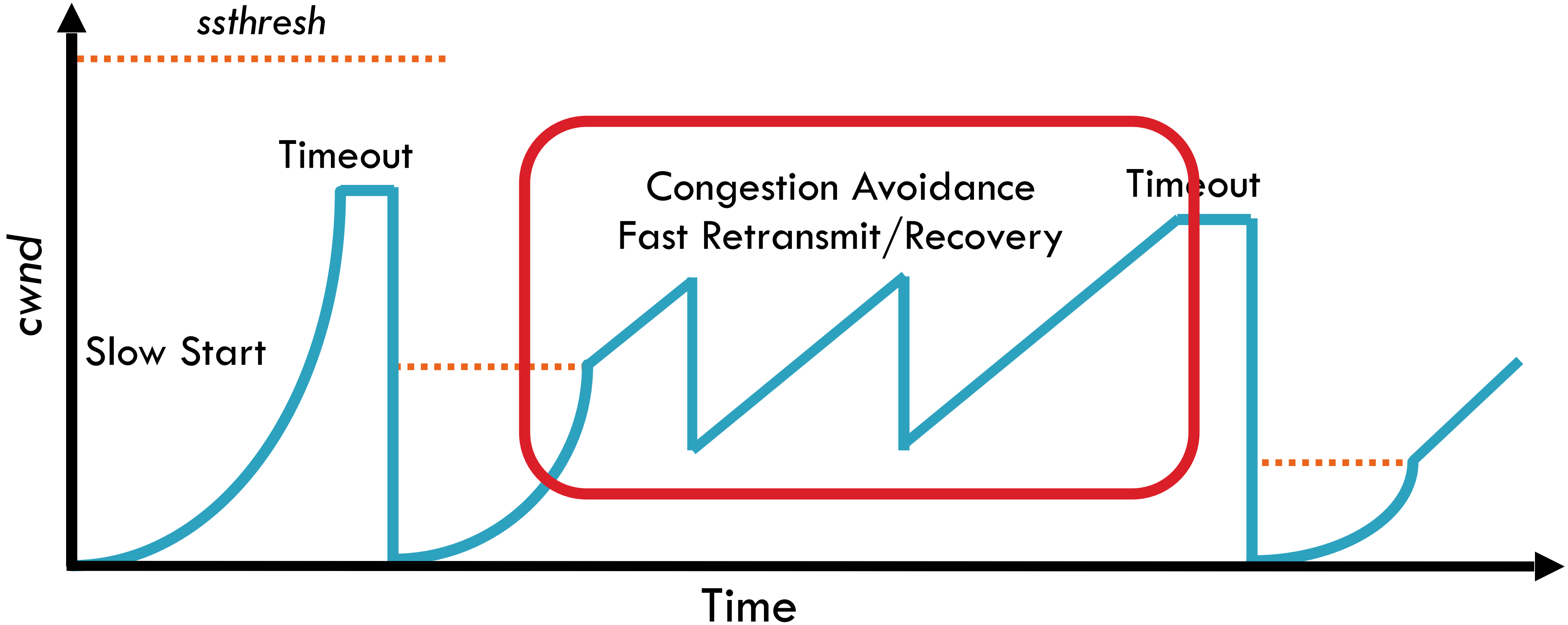
Fast Retransmit and Fast Recovery



Fast Retransmit and Fast Recovery

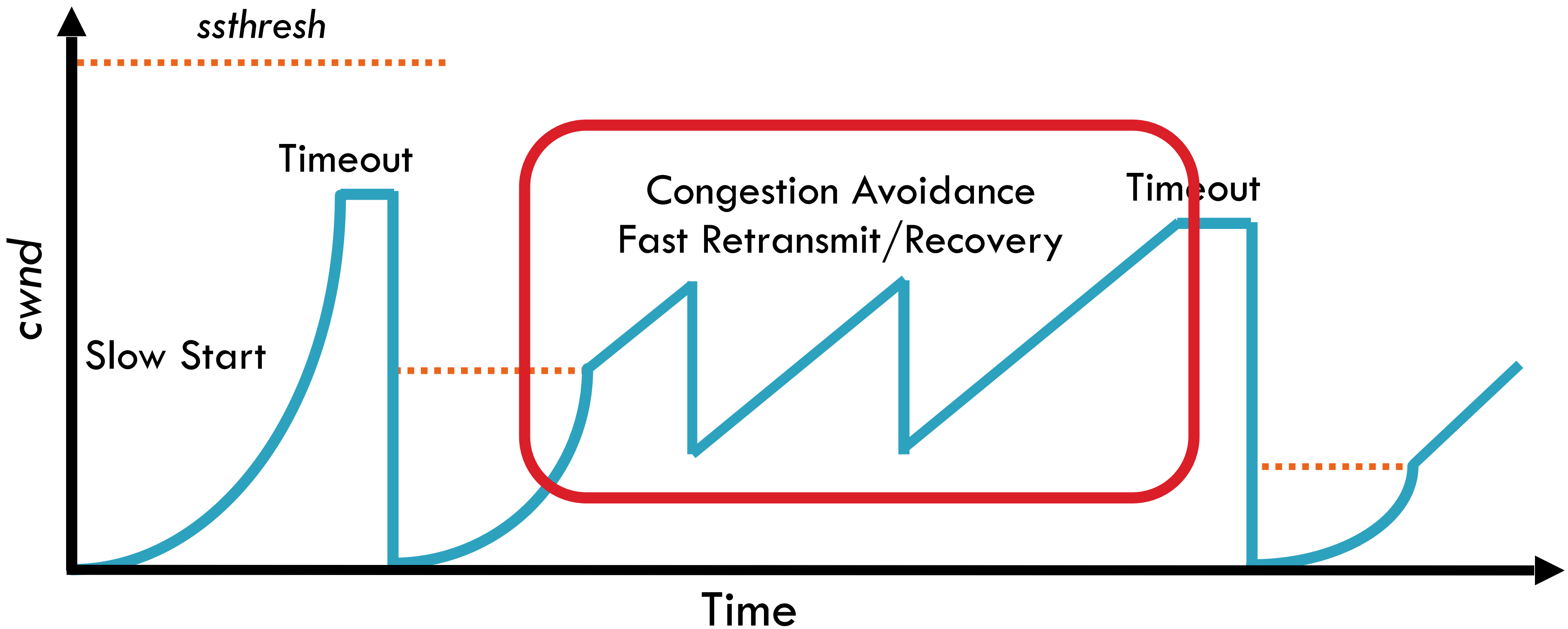


Fast Retransmit and Fast Recovery



- At steady state, *cwnd* oscillates around the optimal window size

Fast Retransmit and Fast Recovery



- At steady state, *cwnd* oscillates around the optimal window size
- TCP always forces packet drops

Many TCP Variants…

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno: fast retransmit and fast recovery

Many TCP Variants…

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno: fast retransmit and fast recovery
- NewReno: improved fast retransmit
  - Each duplicate ACK triggers a retransmission
  - Problem: >3 out-of-order packets causes pathological retransmissions

Many TCP Variants…

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno: fast retransmit and fast recovery
- NewReno: improved fast retransmit
  - Each duplicate ACK triggers a retransmission
  - Problem: >3 out-of-order packets causes pathological retransmissions
- Vegas: delay-based congestion avoidance

Many TCP Variants…

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno: fast retransmit and fast recovery
- NewReno: improved fast retransmit
  - Each duplicate ACK triggers a retransmission
  - Problem: >3 out-of-order packets causes pathological retransmissions
- Vegas: delay-based congestion avoidance
- And many, many, many more…

TCP in the Real World

- What are the most popular variants today?
  - Key problem: TCP performs poorly on high bandwidth-delay product networks (like the modern Internet)
  - Compound TCP (Windows)
    - Based on Reno
    - Uses two congestion windows: delay based and loss based
    - Thus, it uses a *compound* congestion controller
  - TCP CUBIC (Linux)
    - Enhancement of BIC (Binary Increase Congestion Control)
    - Window size controlled by cubic function
    - Parameterized by the time *T* since the last dropped packet

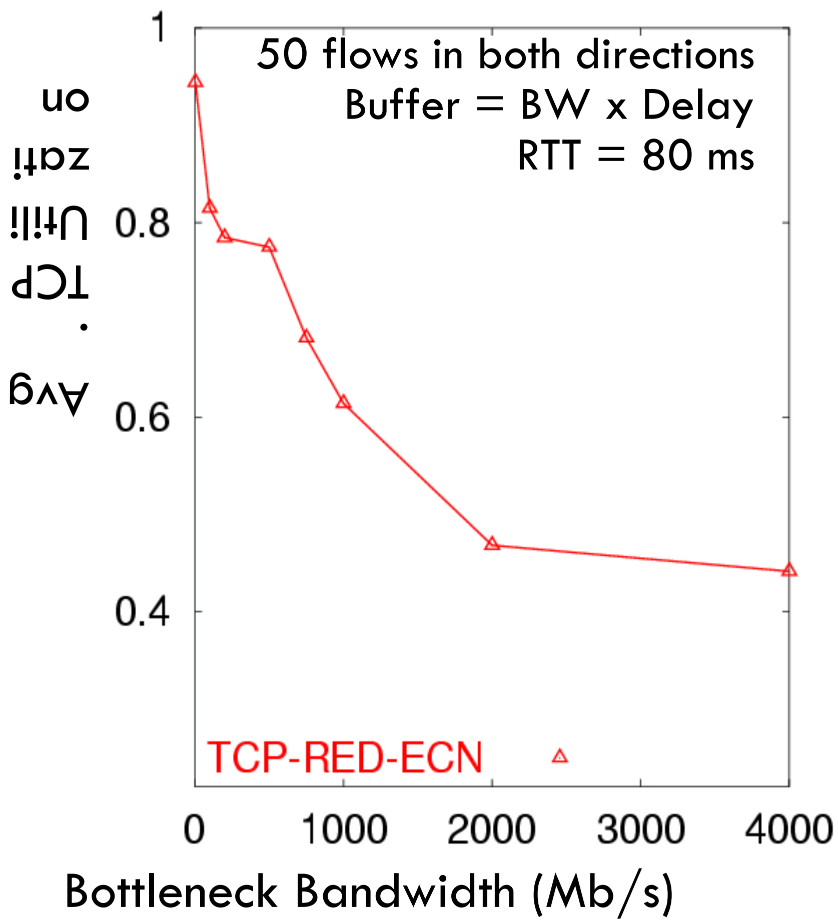
High Bandwidth-Delay Product

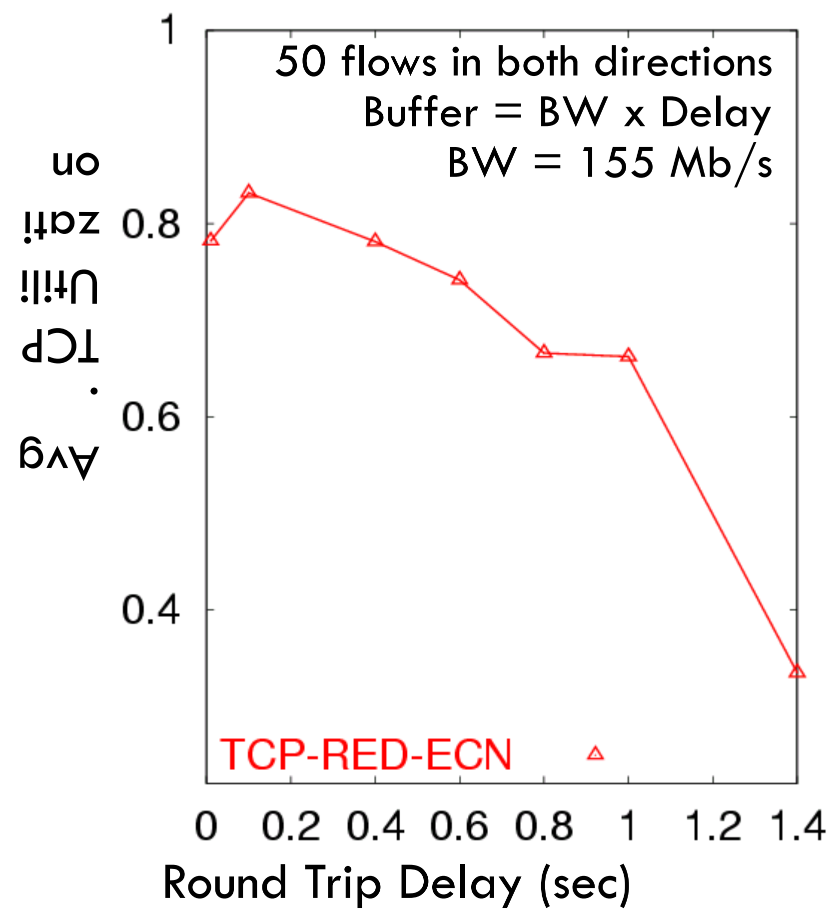
- Key Problem: TCP performs poorly when
  - The capacity of the network (bandwidth) is large
  - The delay (RTT) of the network is large
  - Or, when bandwidth * delay is large
    - b * d = maximum amount of in-flight data in the network
    - a.k.a. the bandwidth-delay product

High Bandwidth-Delay Product

- Key Problem: TCP performs poorly when
  - The capacity of the network (bandwidth) is large
  - The delay (RTT) of the network is large
  - Or, when bandwidth * delay is large
    - b * d = maximum amount of in-flight data in the network
    - a.k.a. the bandwidth-delay product
- Why does TCP perform poorly?
  - Slow start and additive increase are slow to converge
  - TCP is ACK clocked
    - i.e. TCP can only react as quickly as ACKs are received
    - Large RTT → ACKs are delayed → TCP is slow to react

Poor Performance of TCP Reno CC





Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly

Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
- Maintain fairness with other TCP varients
  - Window growth cannot be too aggressive

Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
- Maintain fairness with other TCP varients
  - Window growth cannot be too aggressive
- Improve RTT fairness
  - TCP Tahoe/Reno flows are not fair when RTTs vary widely

Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
- Maintain fairness with other TCP varients
  - Window growth cannot be too aggressive
- Improve RTT fairness
  - TCP Tahoe/Reno flows are not fair when RTTs vary widely
- Simple implementation

Compound TCP Implementation

- Default TCP implementation in Windows
- Key idea: split *cwnd* into two separate windows
  - Traditional, loss-based window
  - New, delay-based window

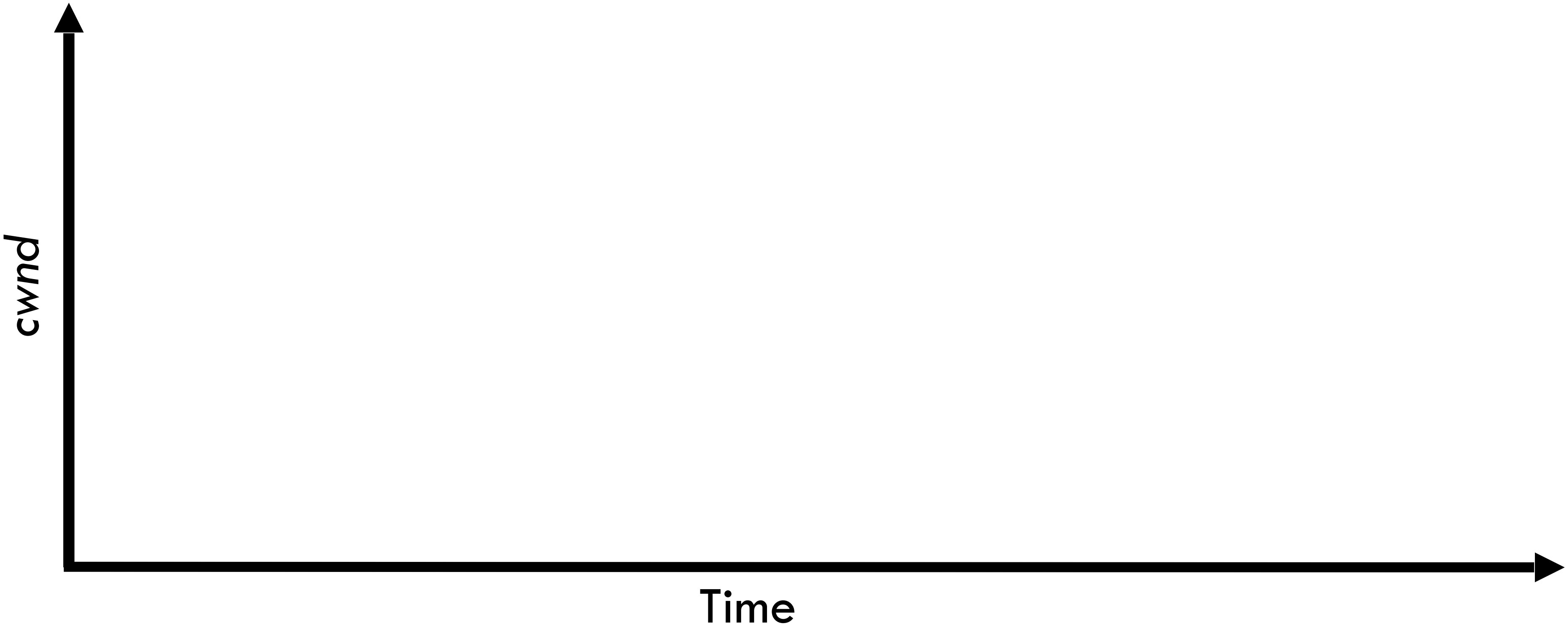
Compound TCP Implementation

- Default TCP implementation in Windows
- Key idea: split *cwnd* into two separate windows
  - Traditional, loss-based window
  - New, delay-based window
- *wnd* = min(*cwnd* + *dwnd*, *adv_wnd*)
  - *cwnd* is controlled by AIMD
  - *dwnd* is the delay window

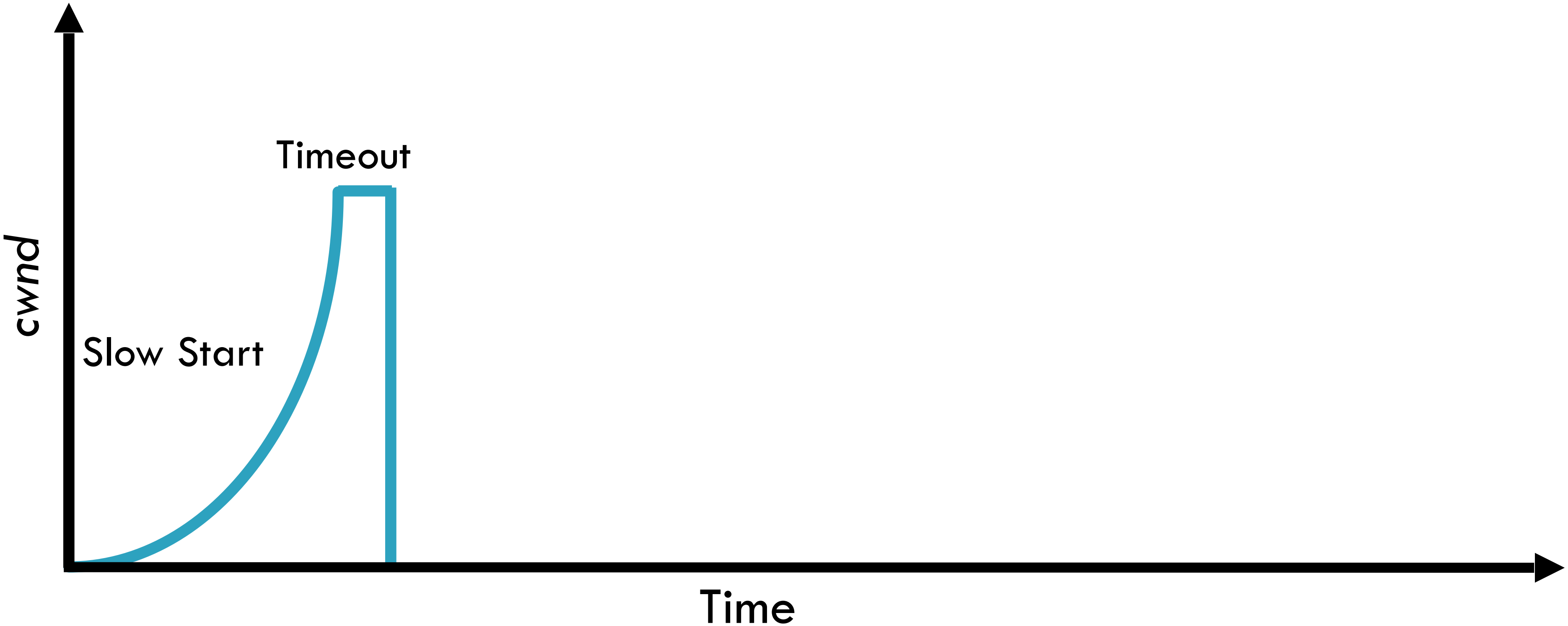
Compound TCP Implementation

- Default TCP implementation in Windows
- Key idea: split *cwnd* into two separate windows
  - Traditional, loss-based window
  - New, delay-based window
- *wnd* = min(*cwnd* + *dwnd*, *adv_wnd*)
  - *cwnd* is controlled by AIMD
  - *dwnd* is the delay window
- Rules for adjusting *dwnd:*
  - If RTT is increasing, decrease *dwnd* (*dwnd* >= 0)
  - If RTT is decreasing, increase *dwnd*
  - Increase/decrease are proportional to the rate of change

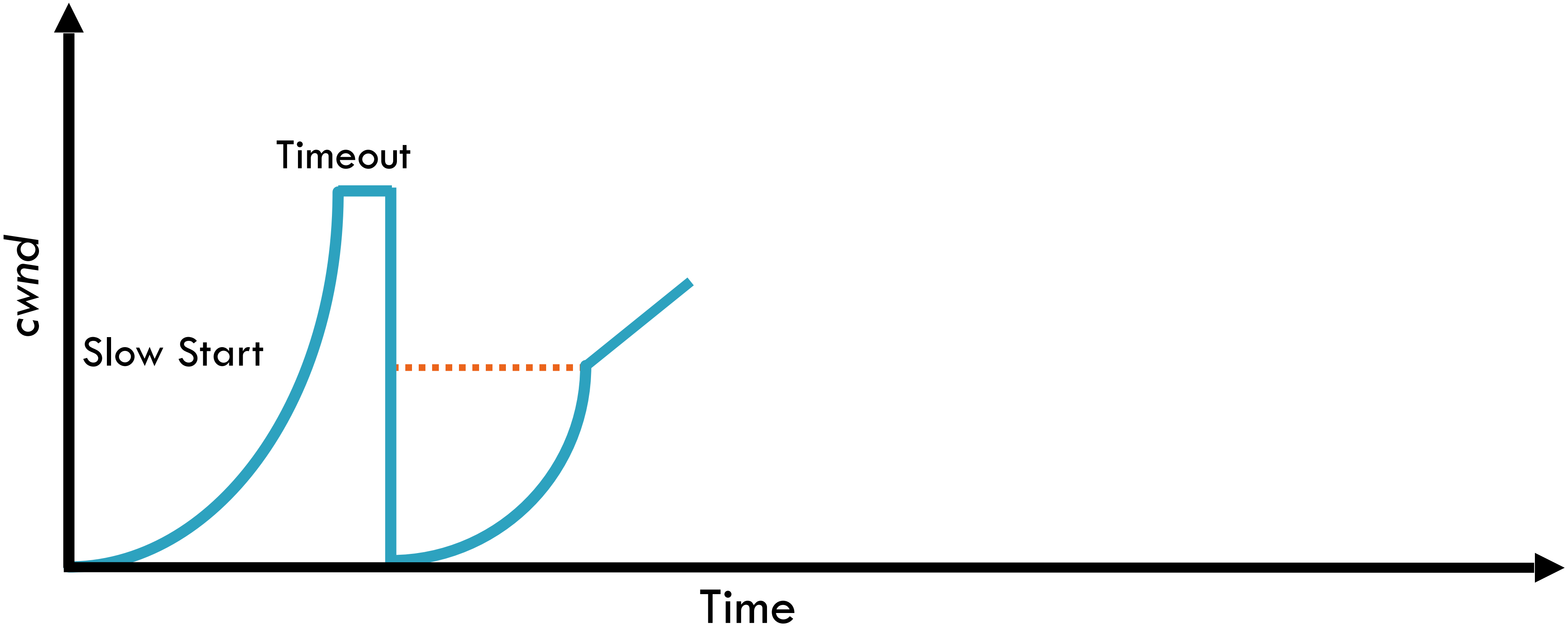
Compound TCP Example



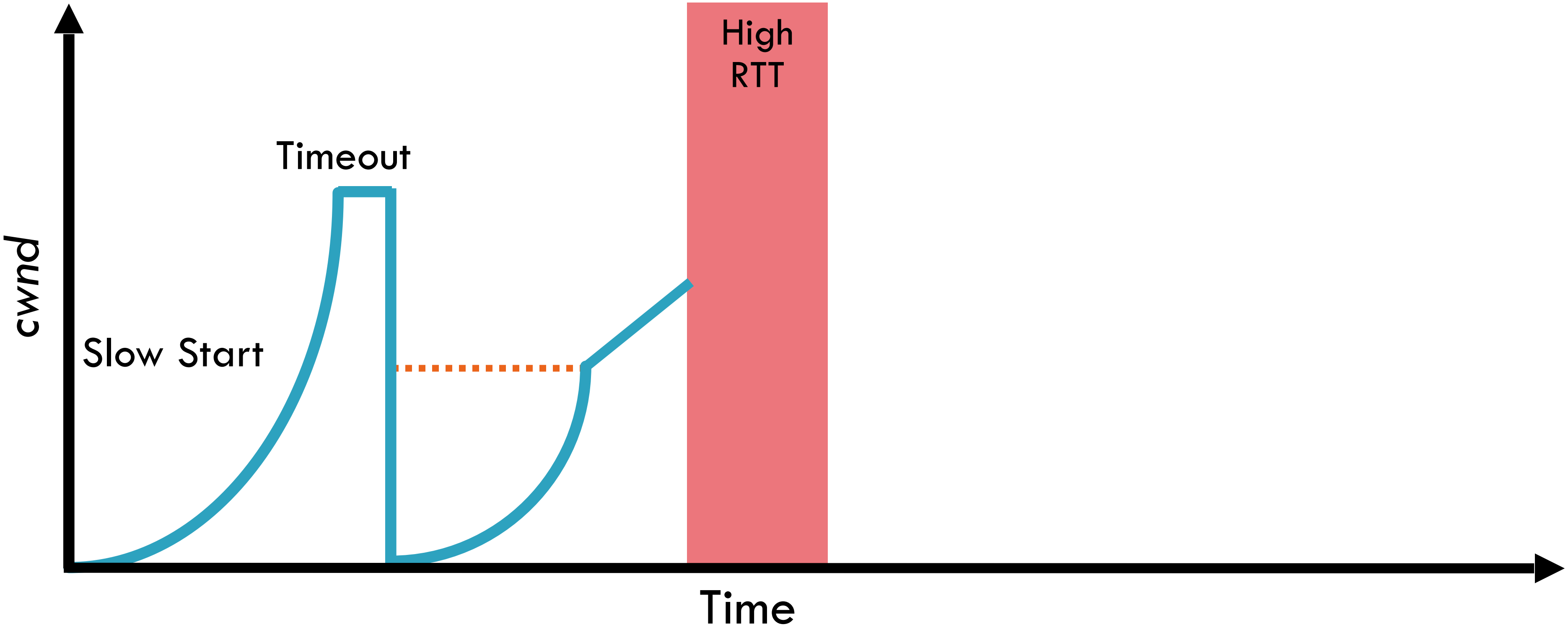
Compound TCP Example



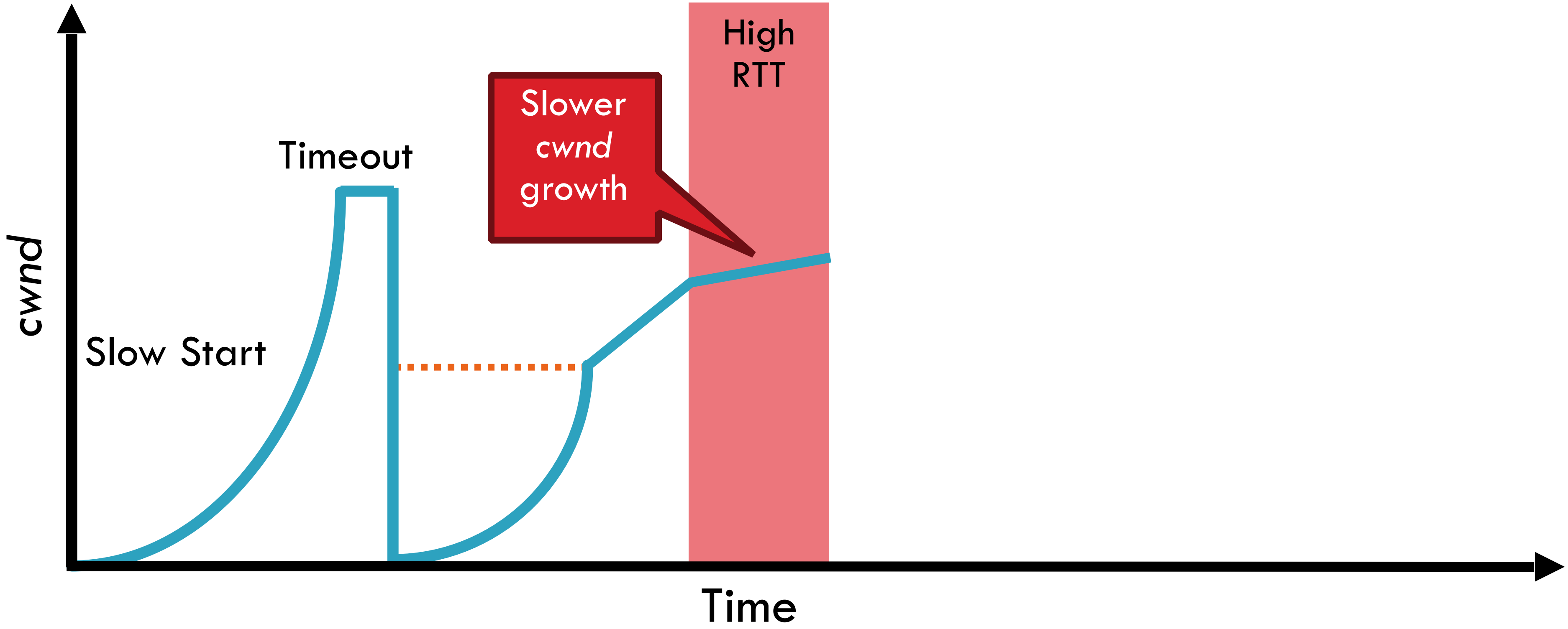
Compound TCP Example



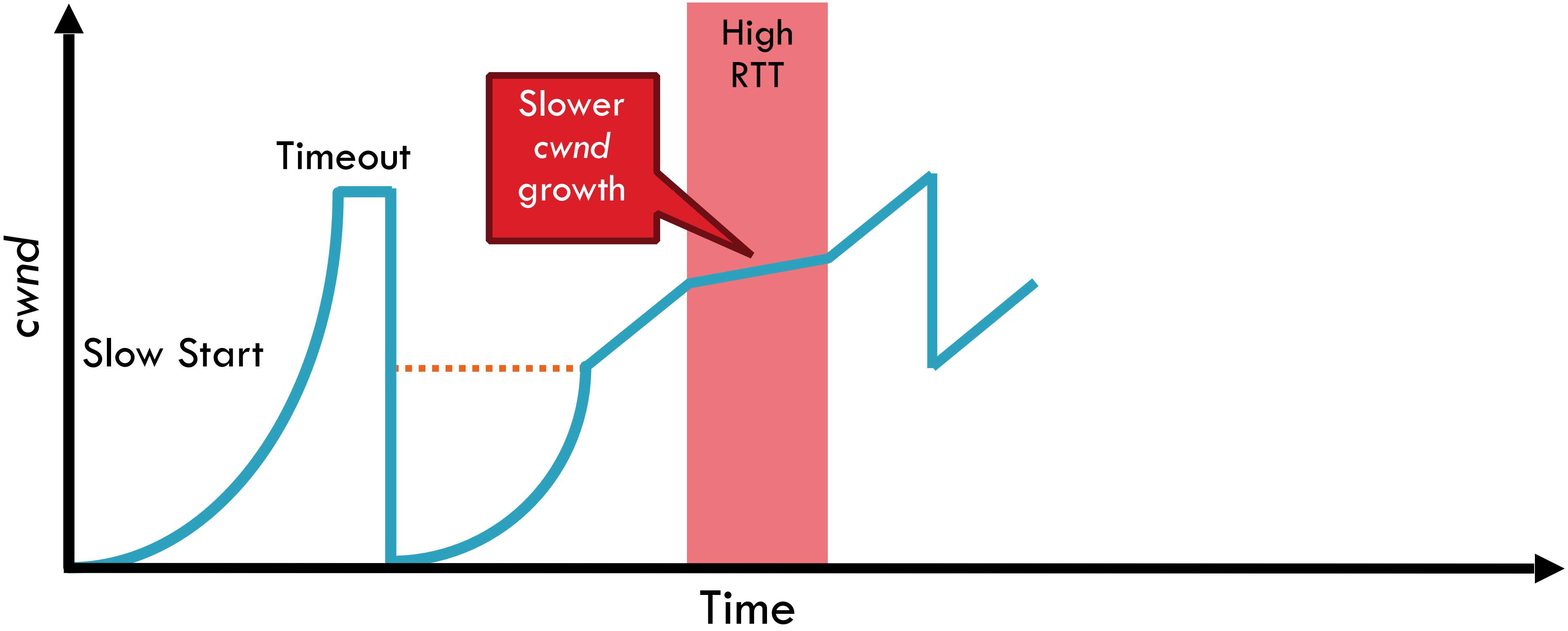
Compound TCP Example



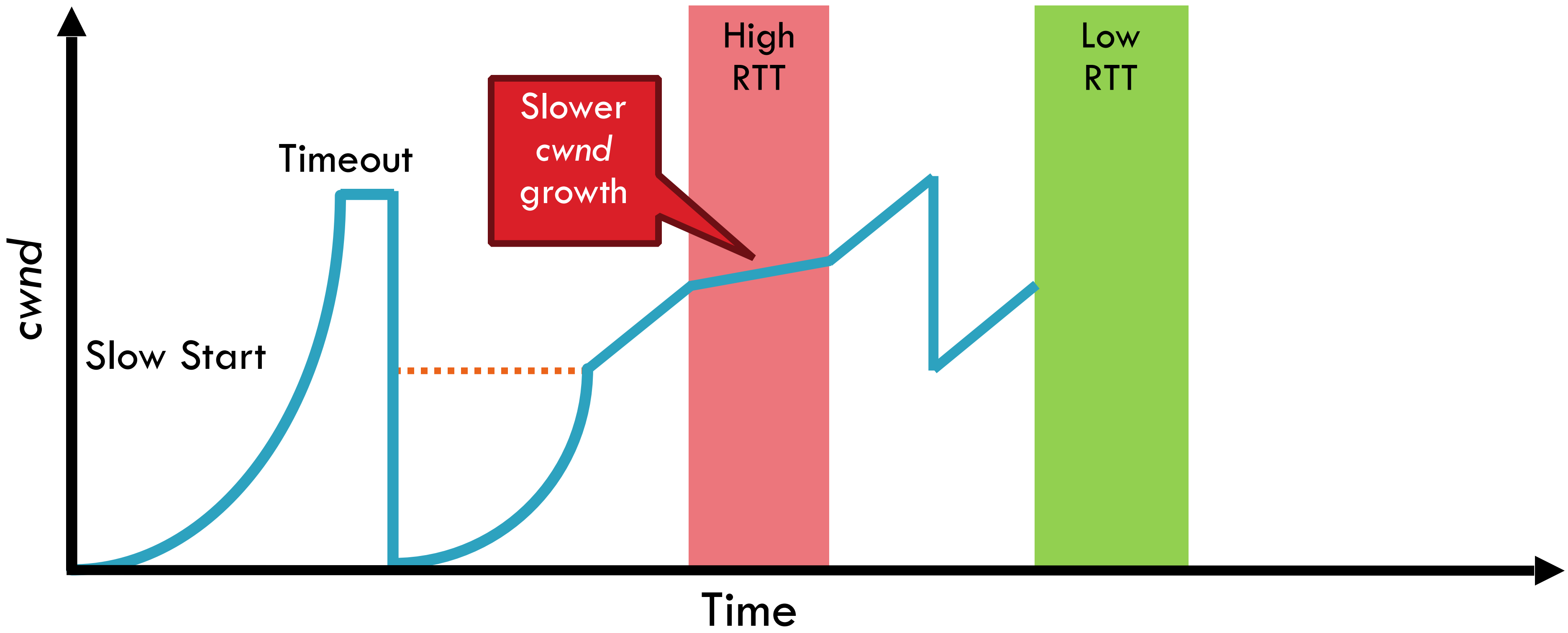
Compound TCP Example



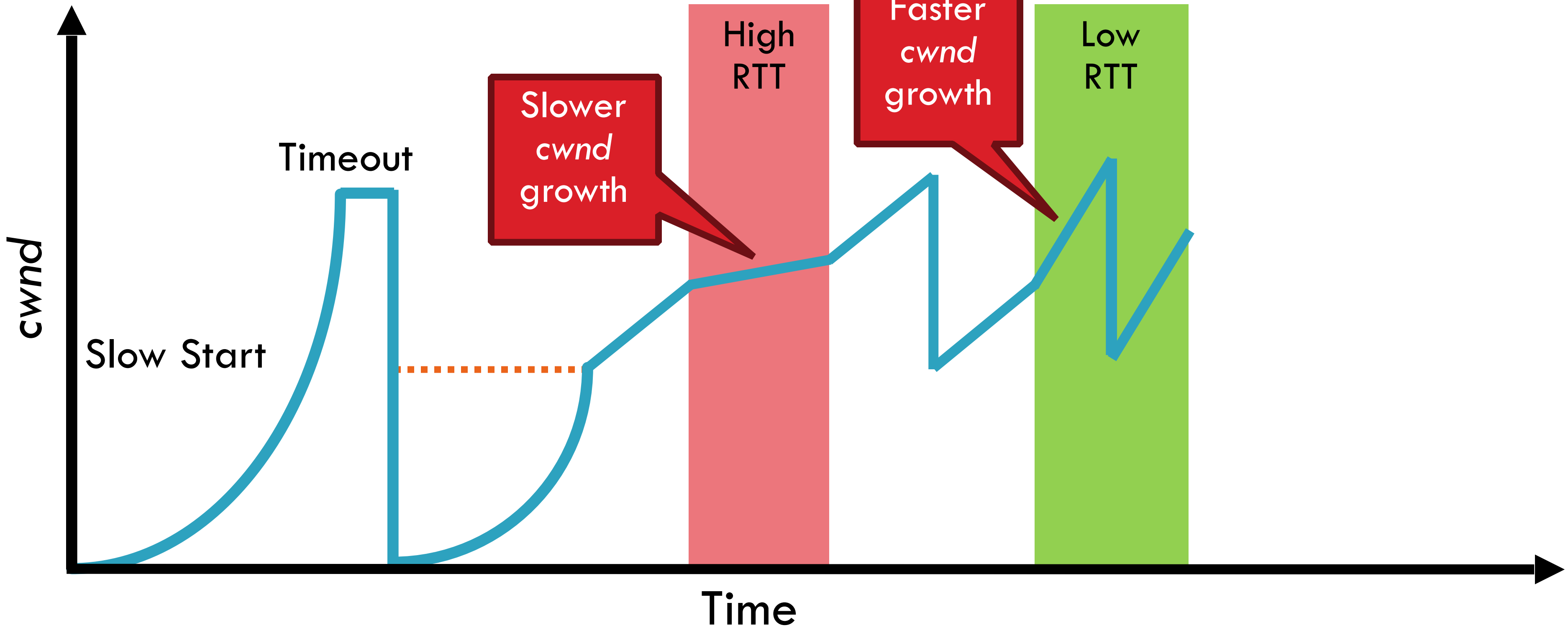
Compound TCP Example



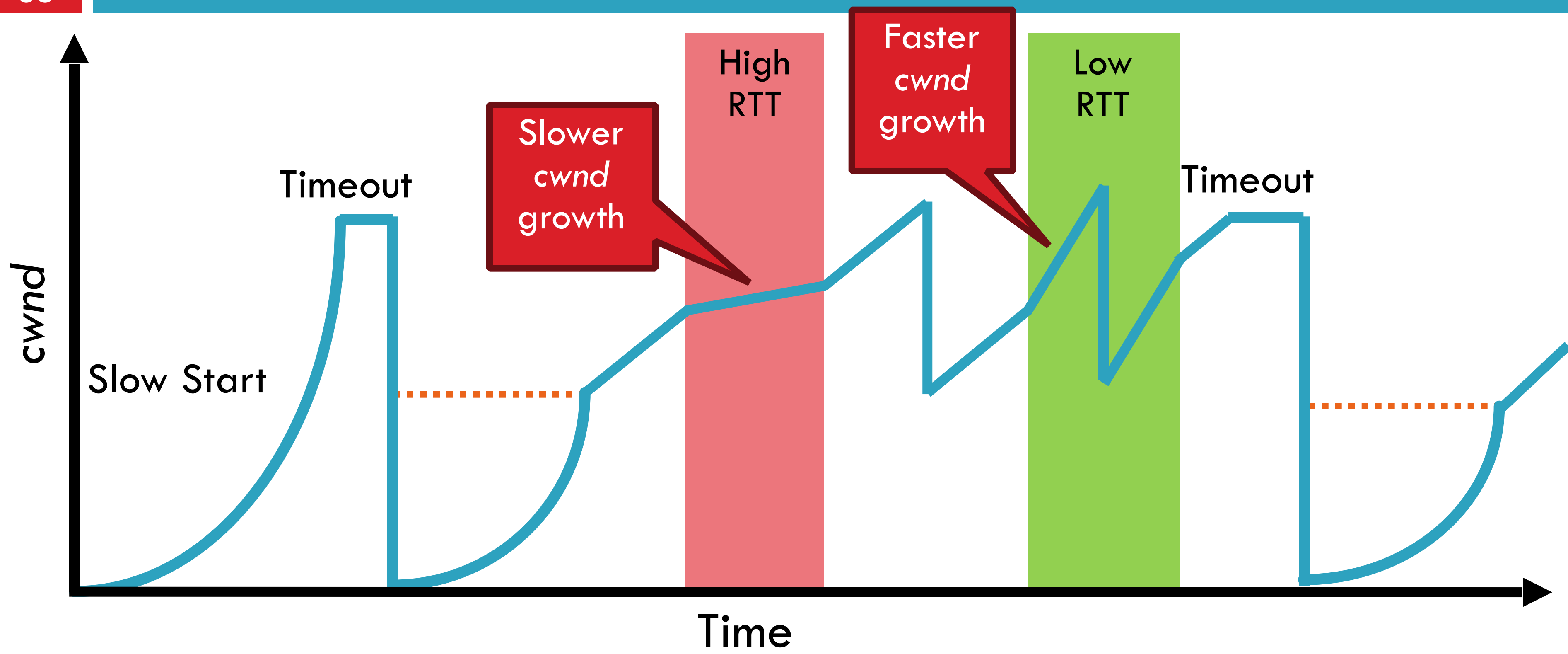
Compound TCP Example



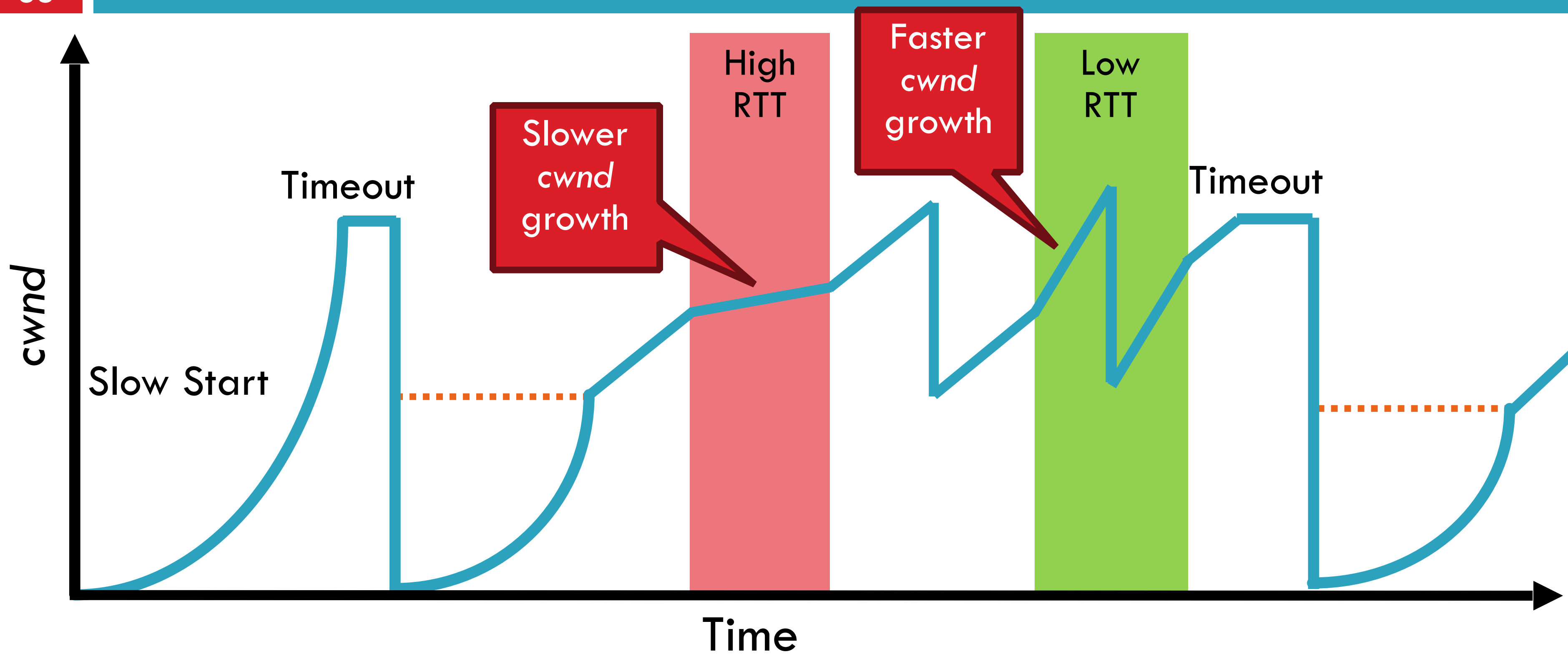
Compound TCP Example



Compound TCP Example

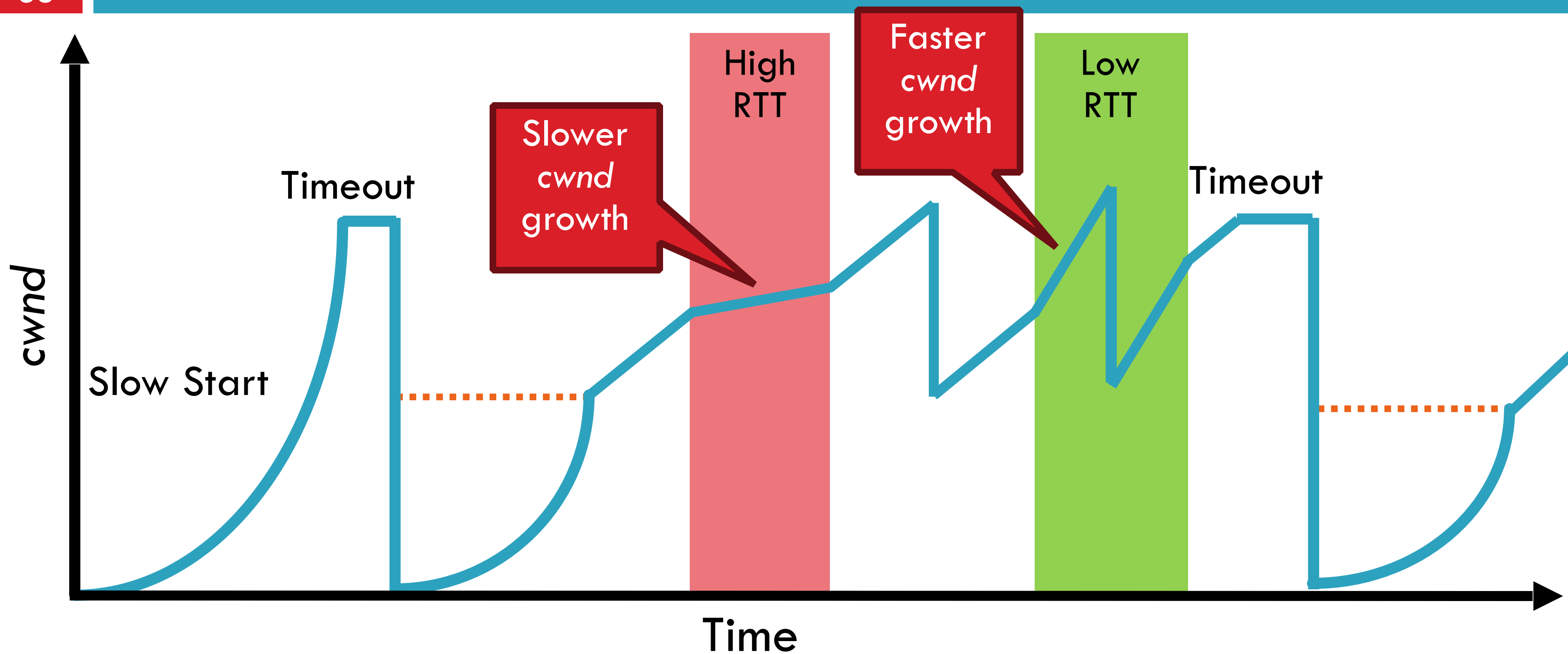


Compound TCP Example



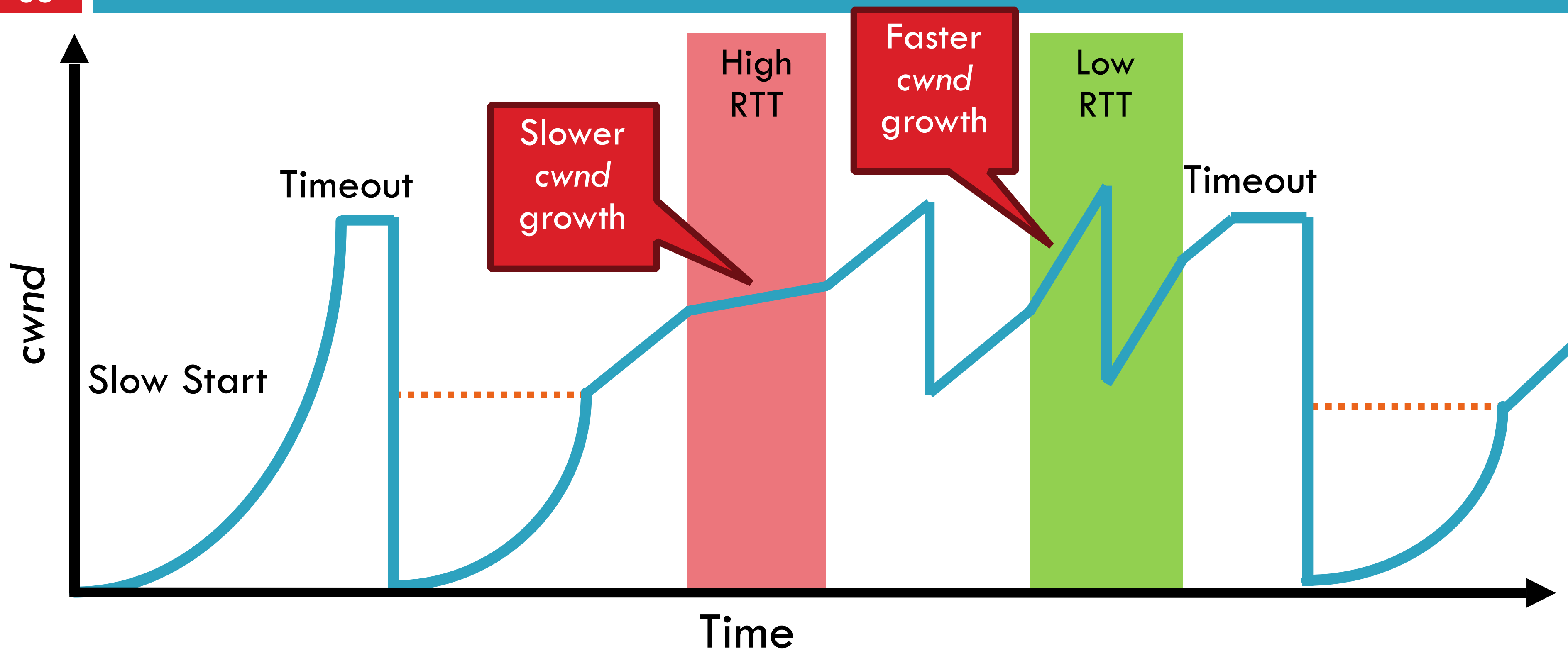
- Aggressiveness corresponds to changes in RTT

Compound TCP Example



- Aggressiveness corresponds to changes in RTT
- Advantages: fast ramp up, more fair to flows with different RTTs

Compound TCP Example



- Aggressiveness corresponds to changes in RTT
- Advantages: fast ramp up, more fair to flows with different RTTs
- Disadvantage: must estimate RTT, which is very challenging

TCP CUBIC Implementation

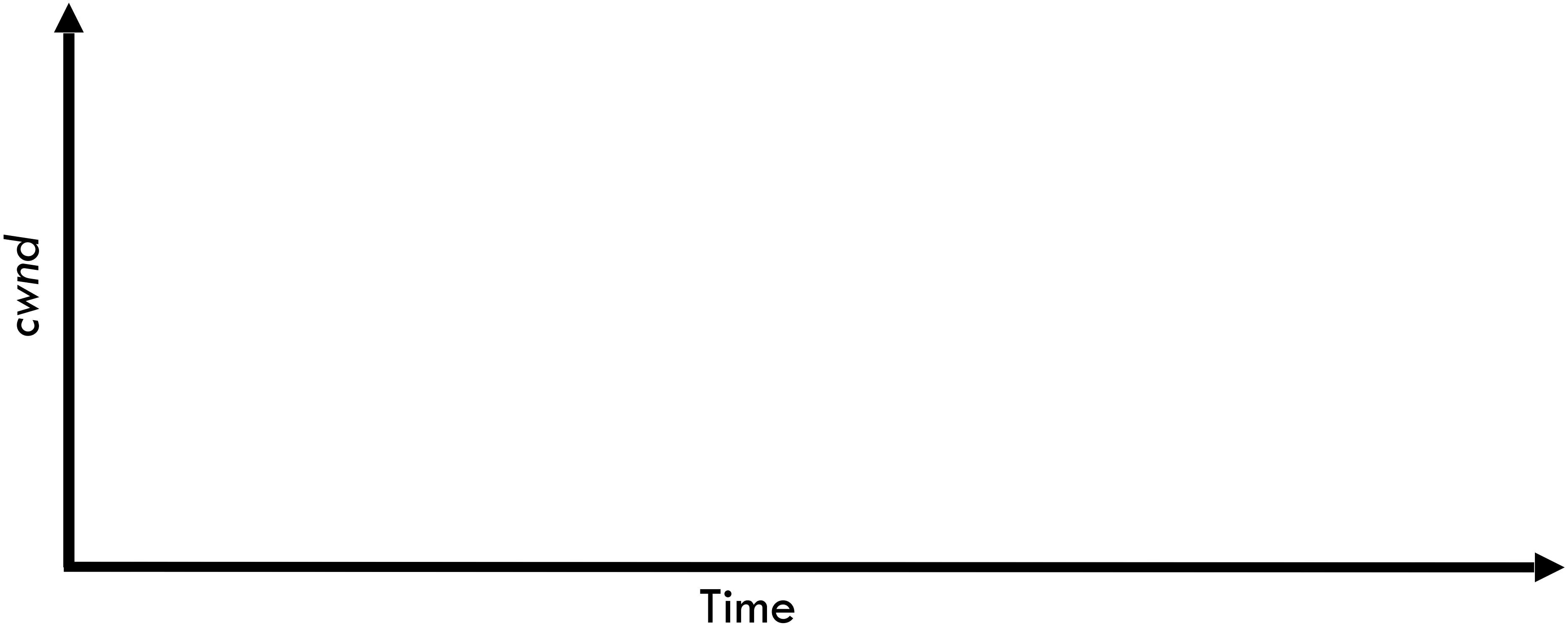
- Default TCP implementation in Linux
- Replace AIMD with cubic function

$$W_{cubic} = C(T - K)^3 + W_{max} \quad (1)$$

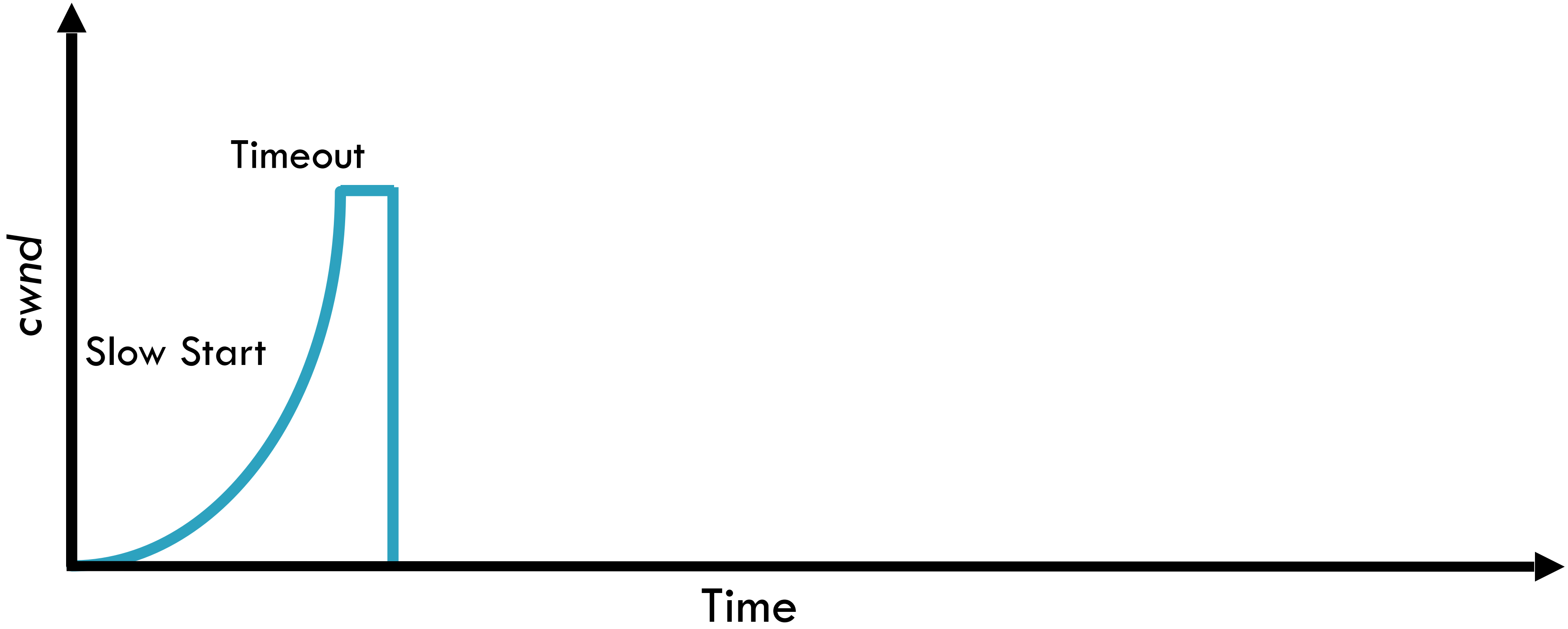
C is a scaling constant, and $K = \sqrt[3]{\frac{W_{max}\beta}{C}}$

  - B → a constant fraction for multiplicative increase
  - T → time since last packet drop
  - W_max → cwnd when last packet dropped

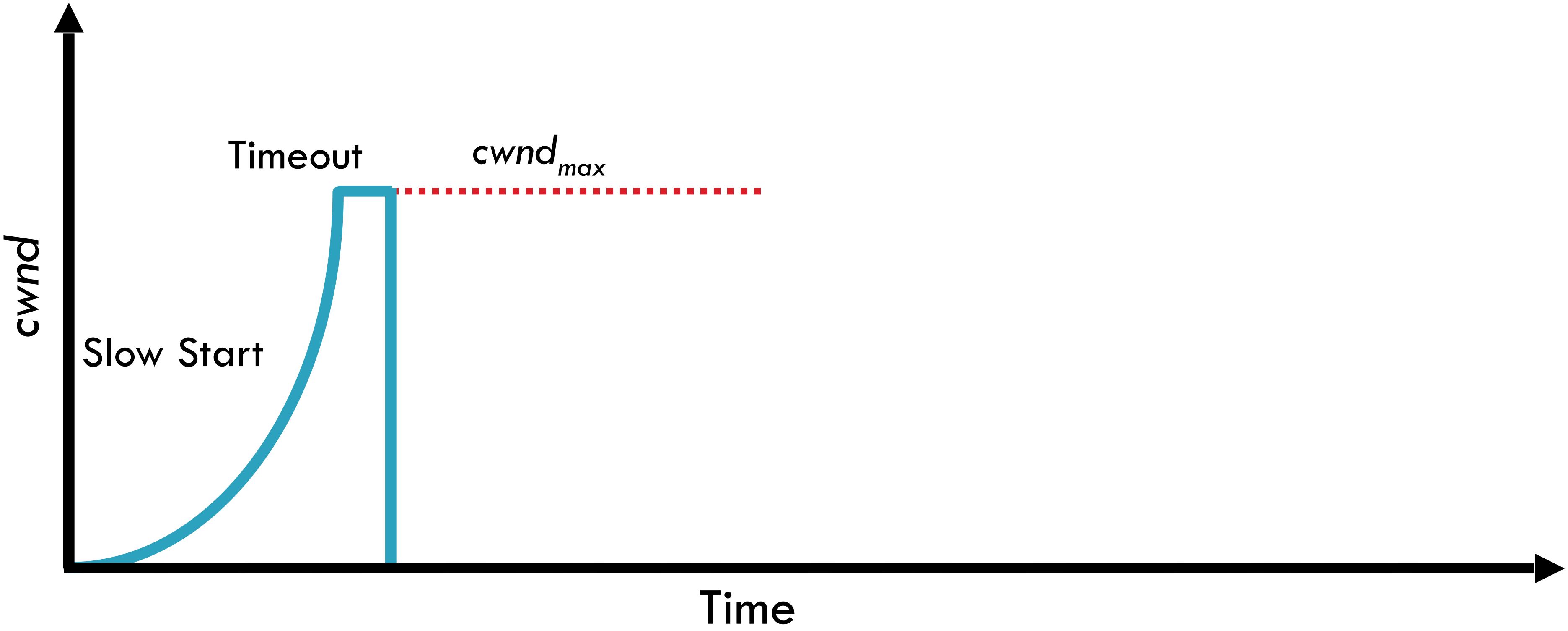
TCP CUBIC Example



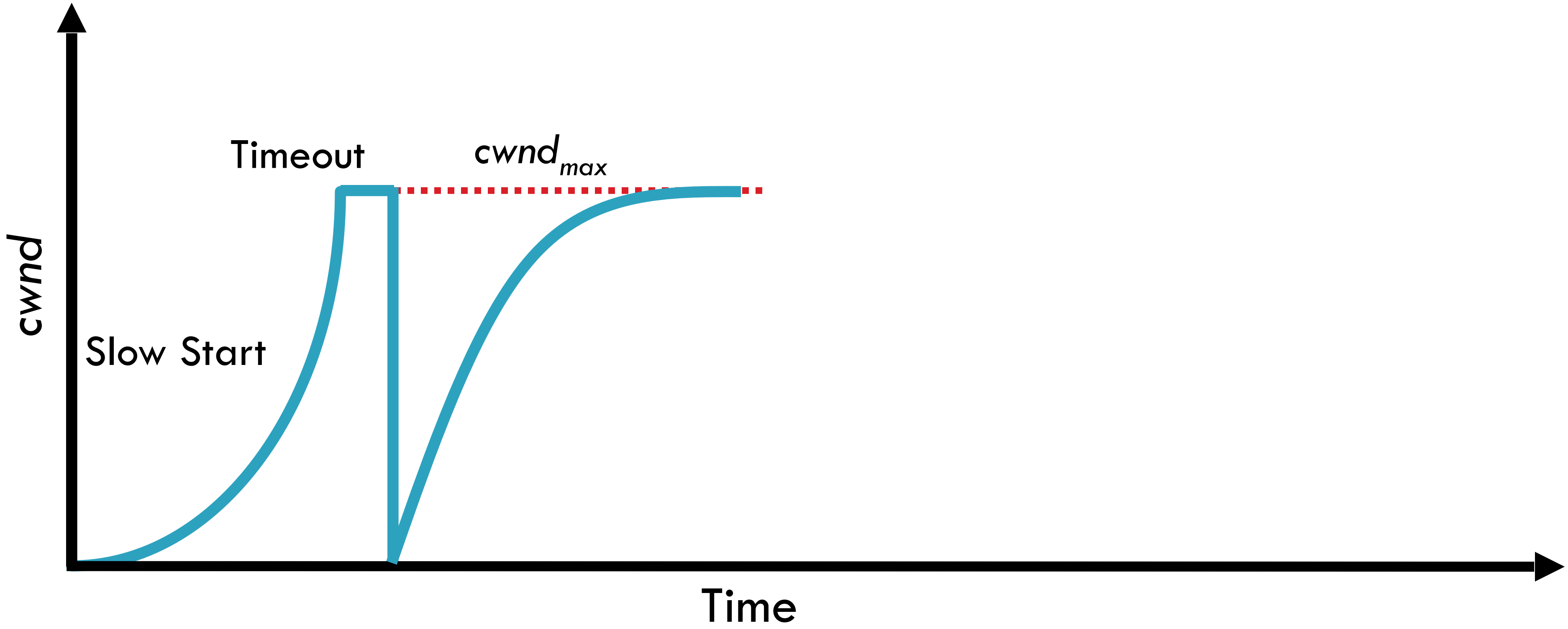
TCP CUBIC Example



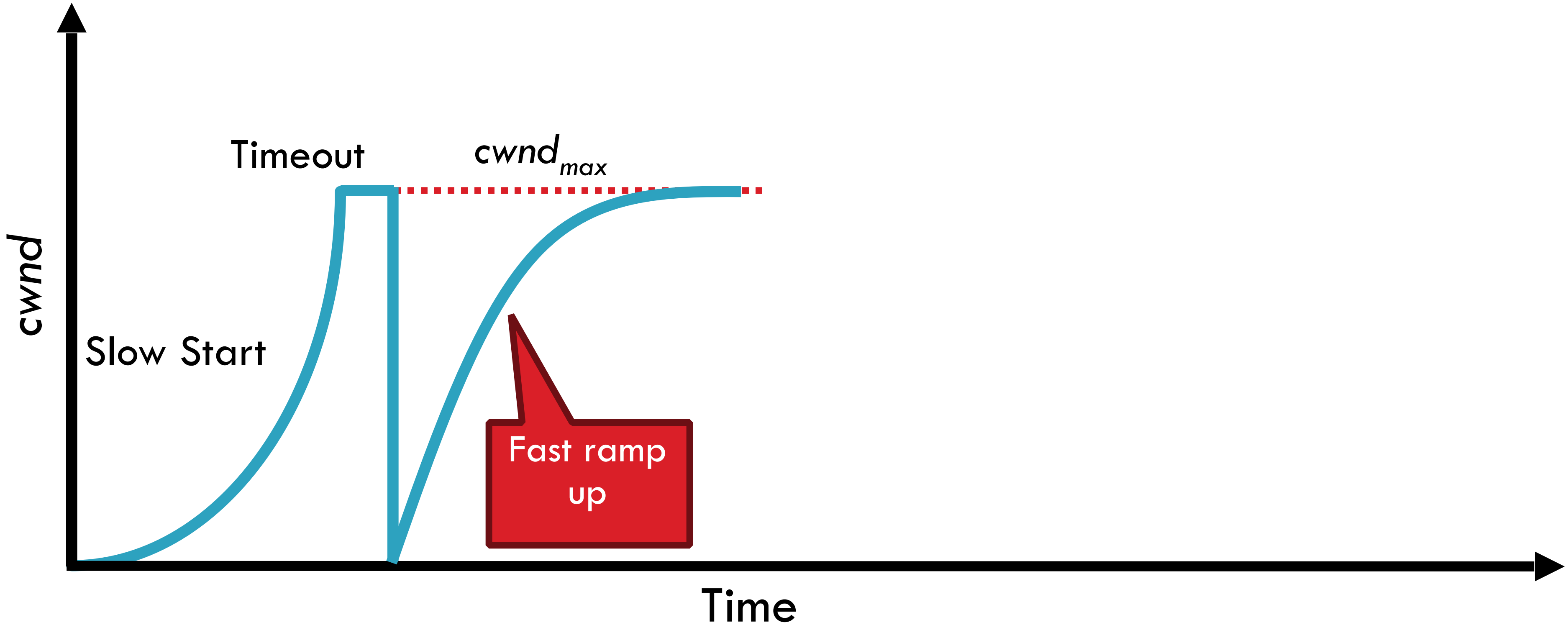
TCP CUBIC Example



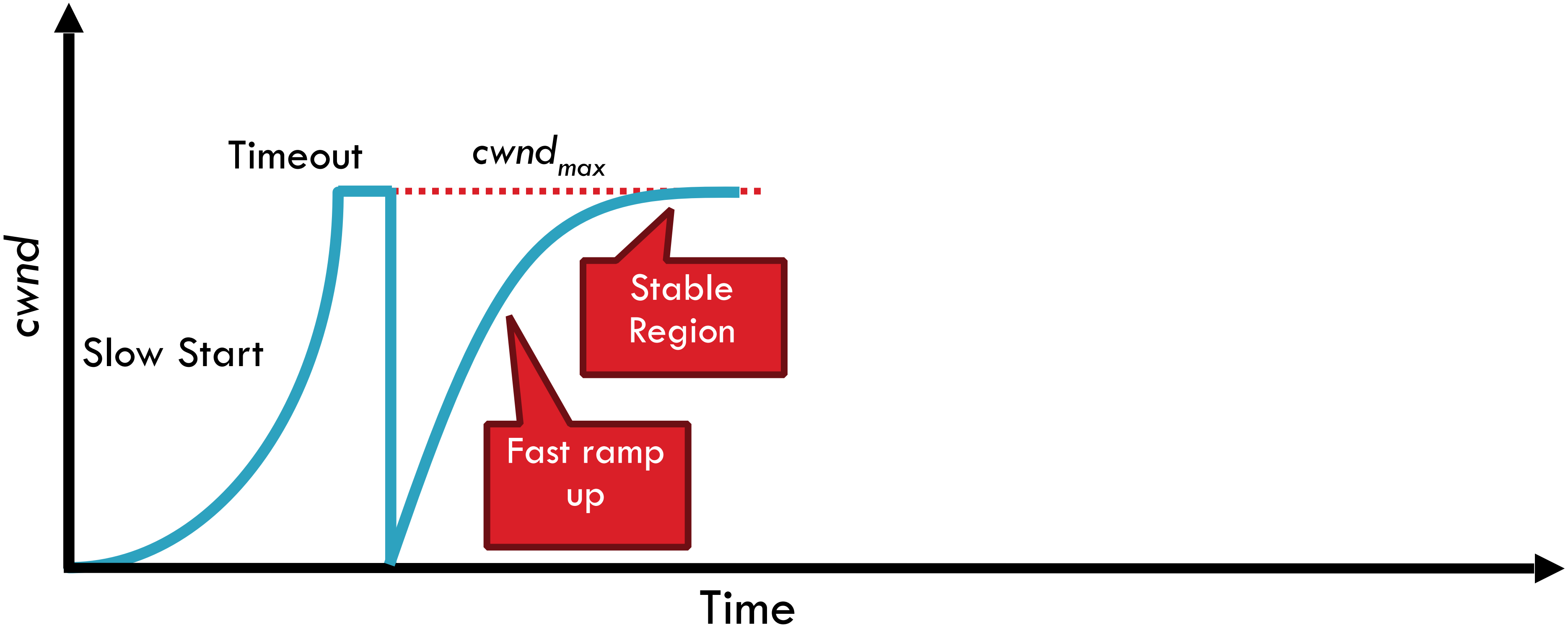
TCP CUBIC Example



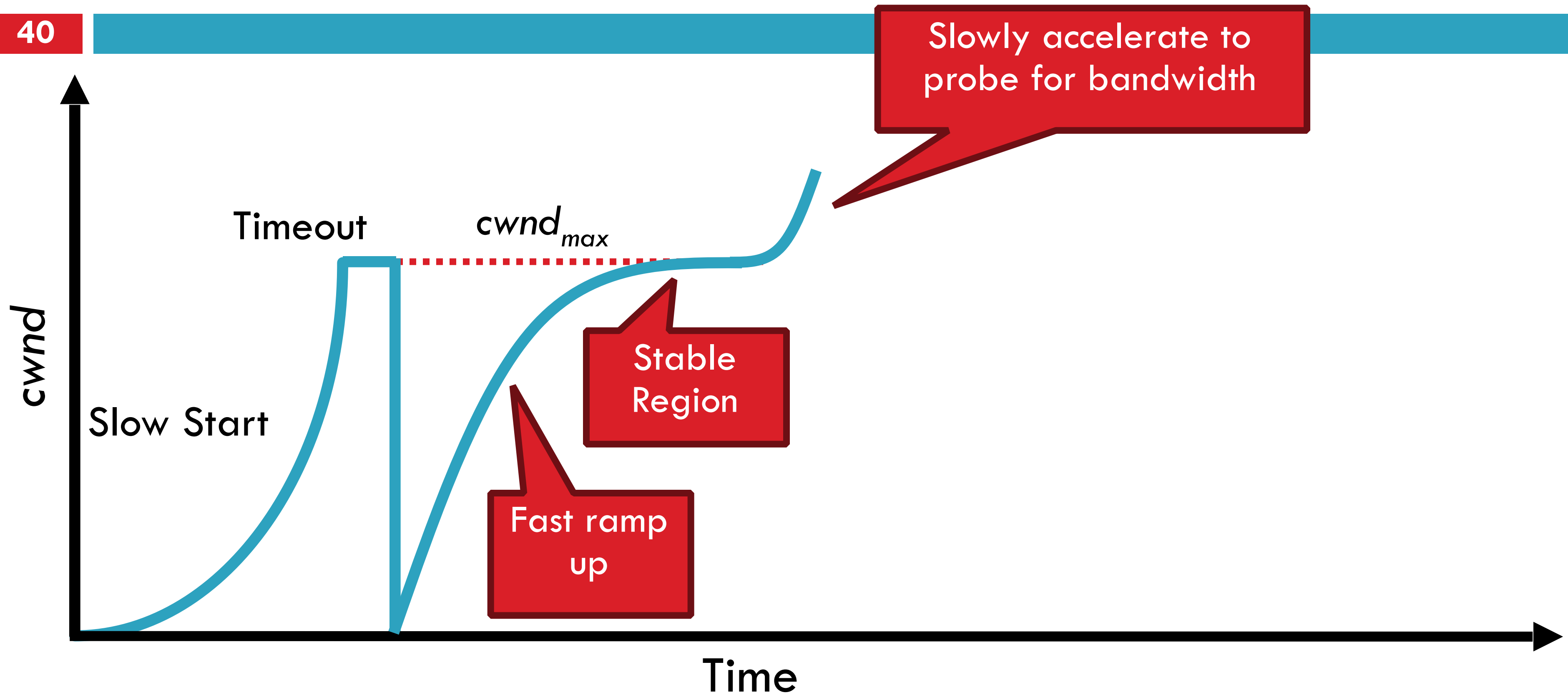
TCP CUBIC Example



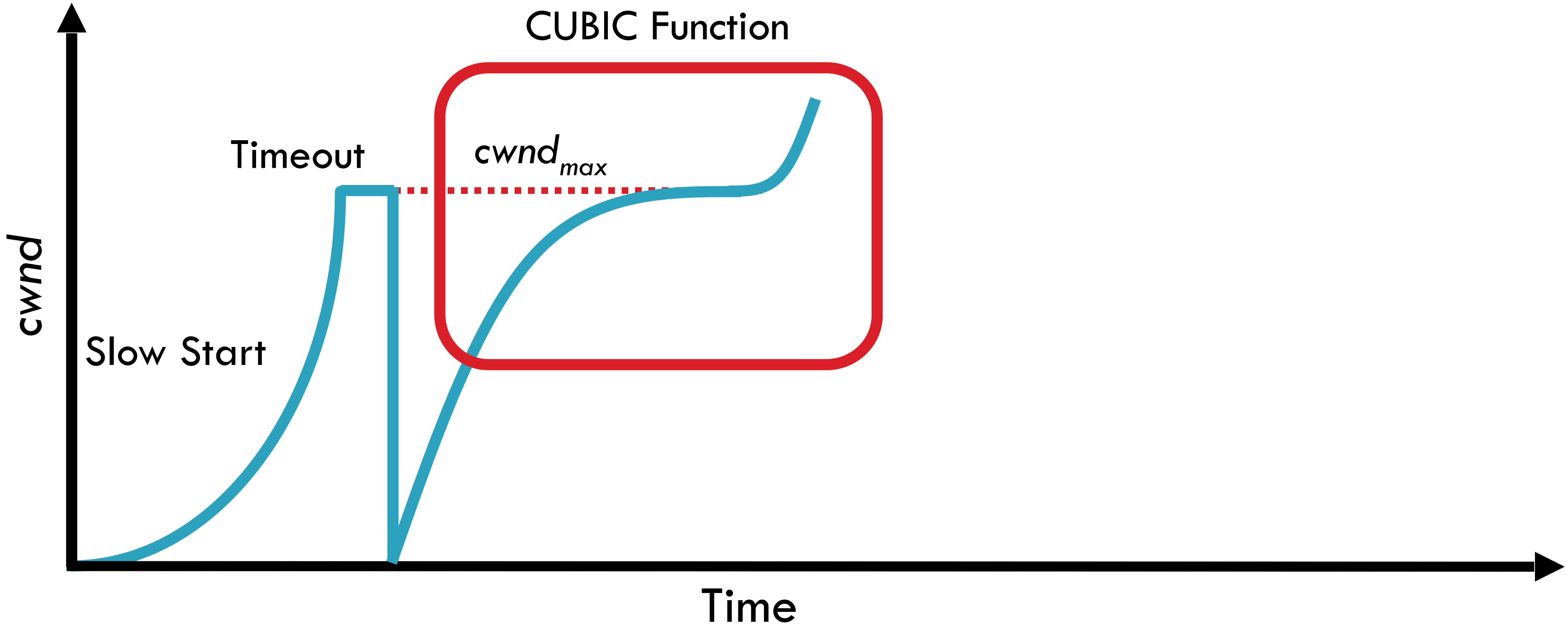
TCP CUBIC Example



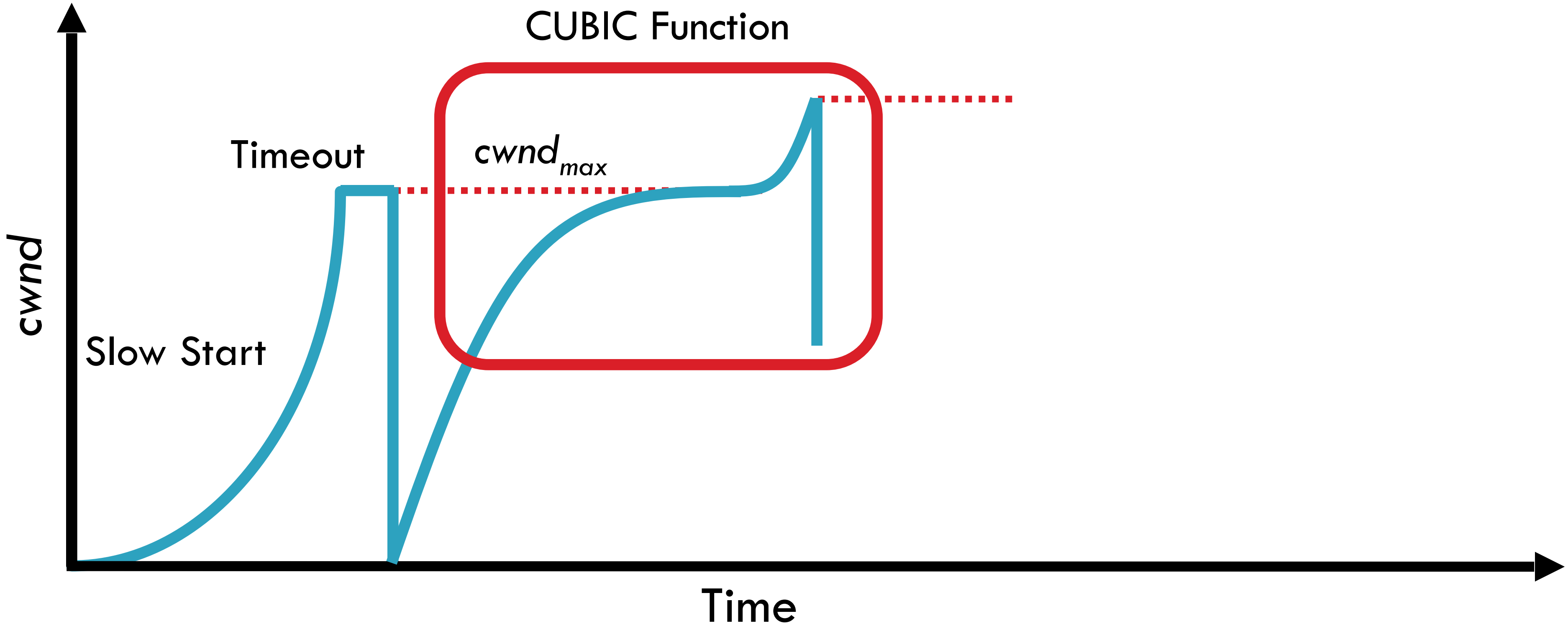
TCP CUBIC Example



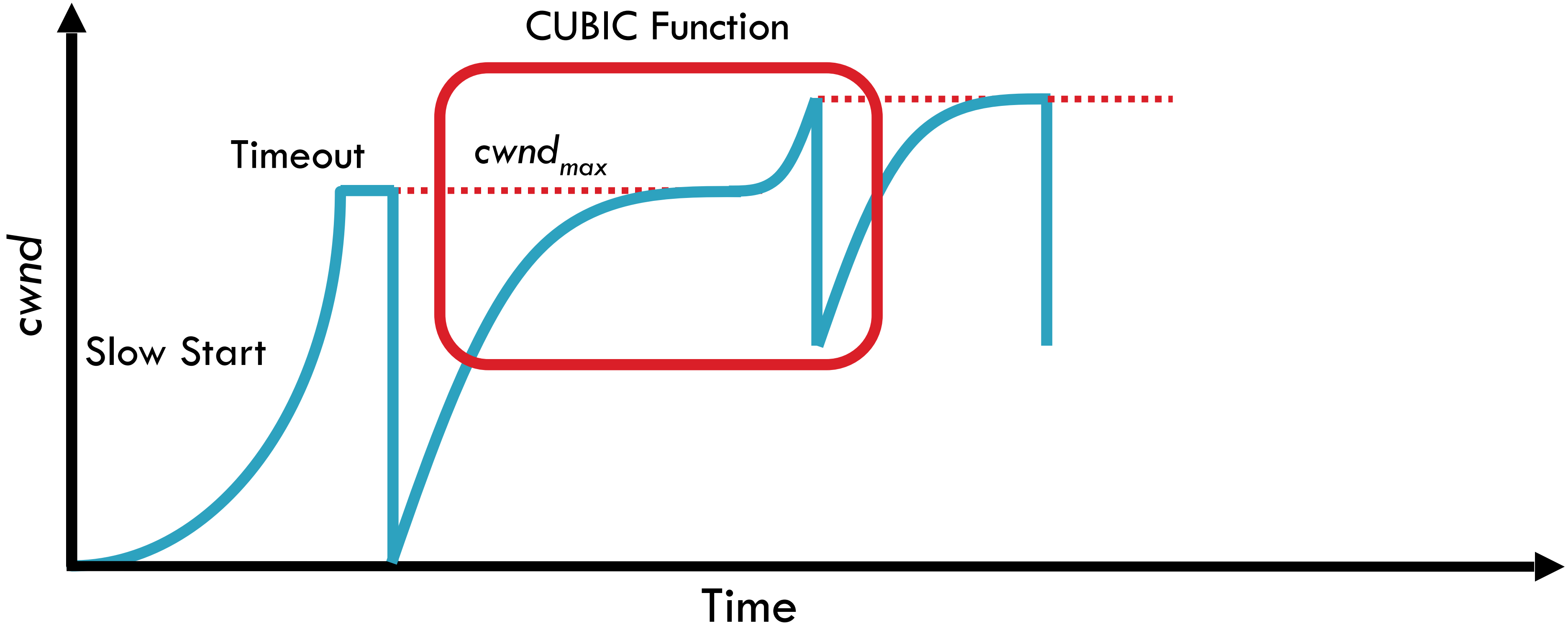
TCP CUBIC Example



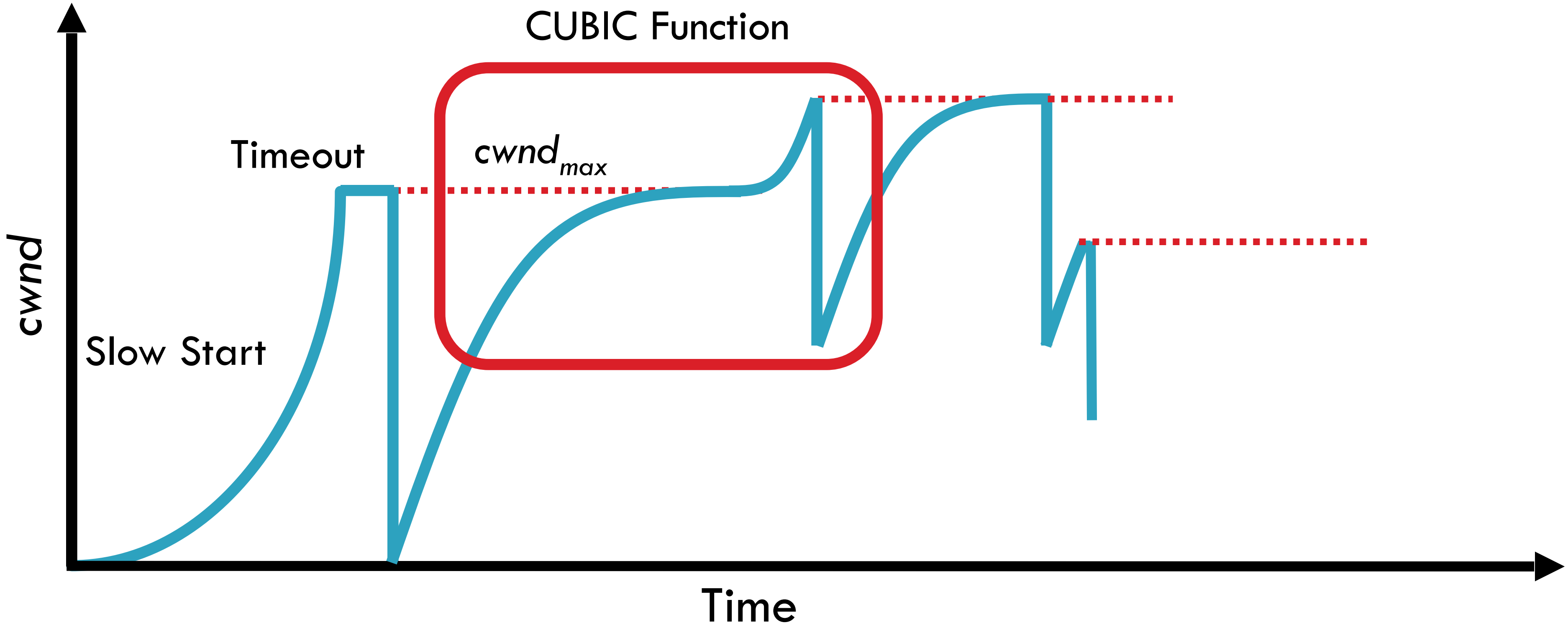
TCP CUBIC Example



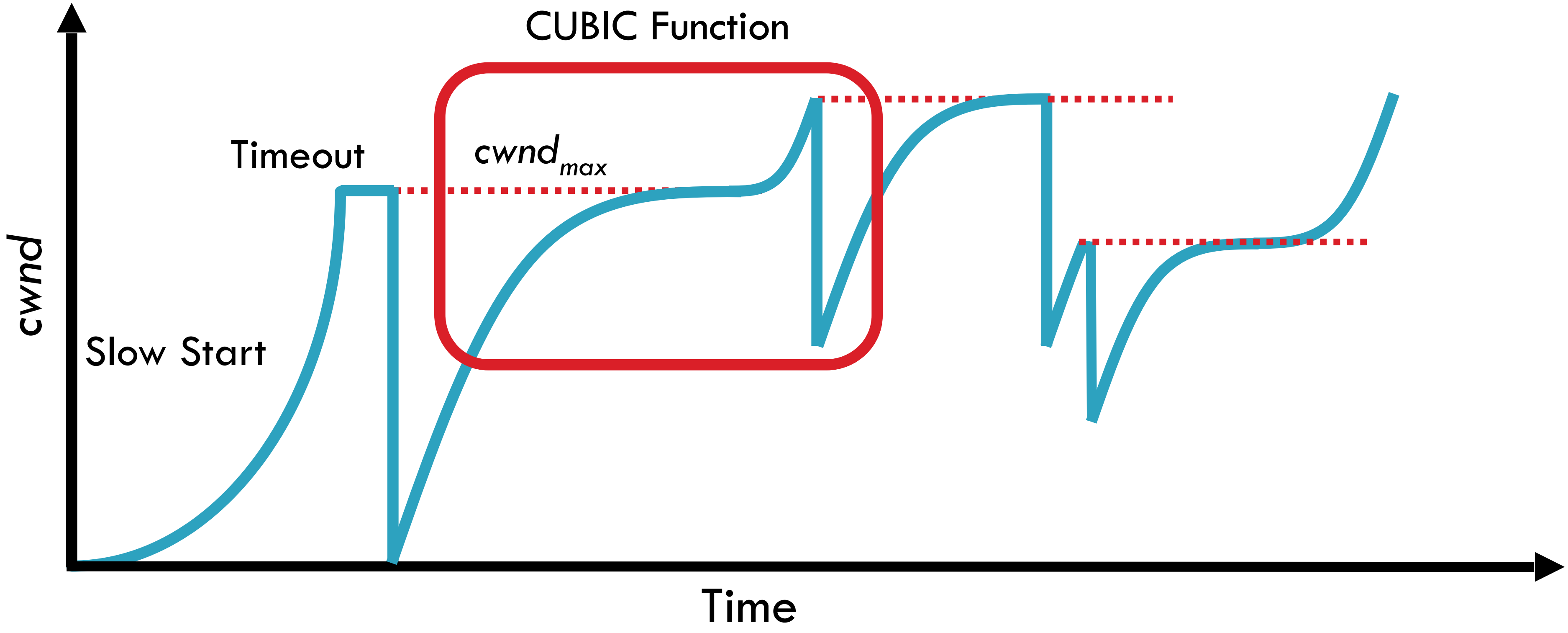
TCP CUBIC Example



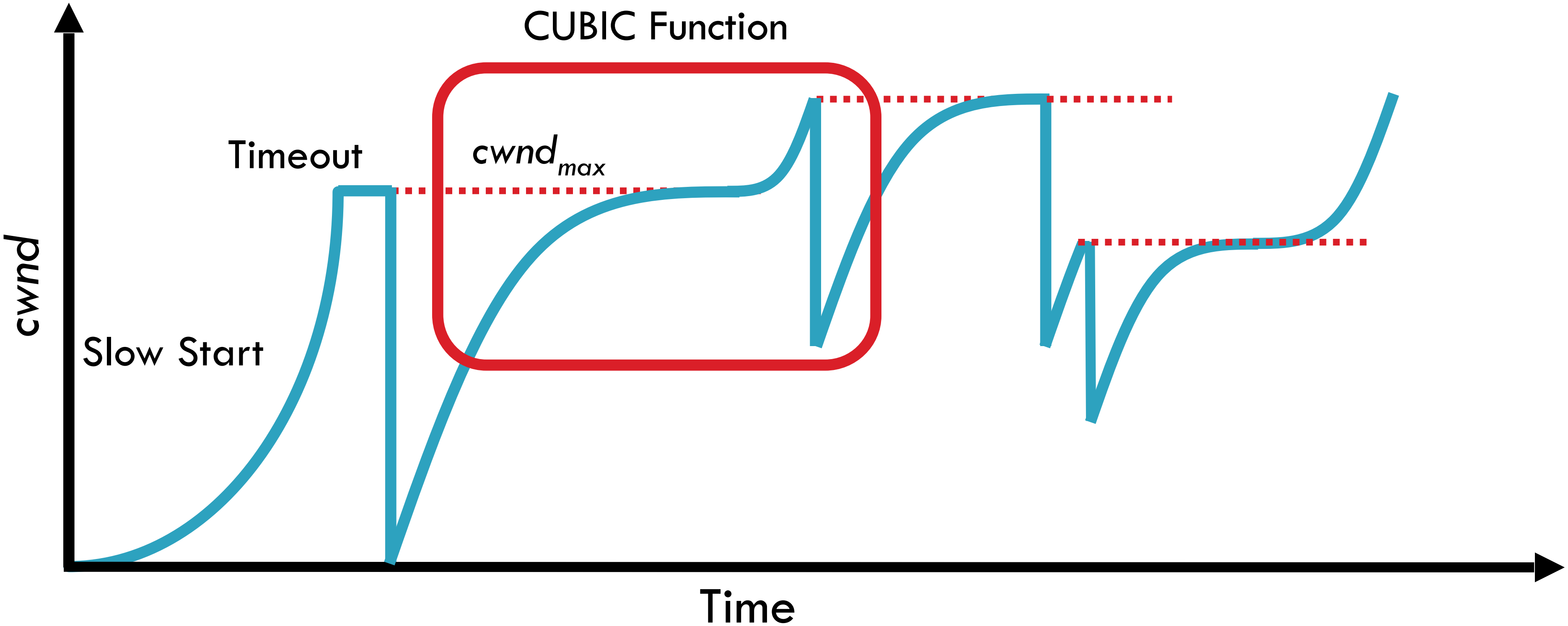
TCP CUBIC Example



TCP CUBIC Example

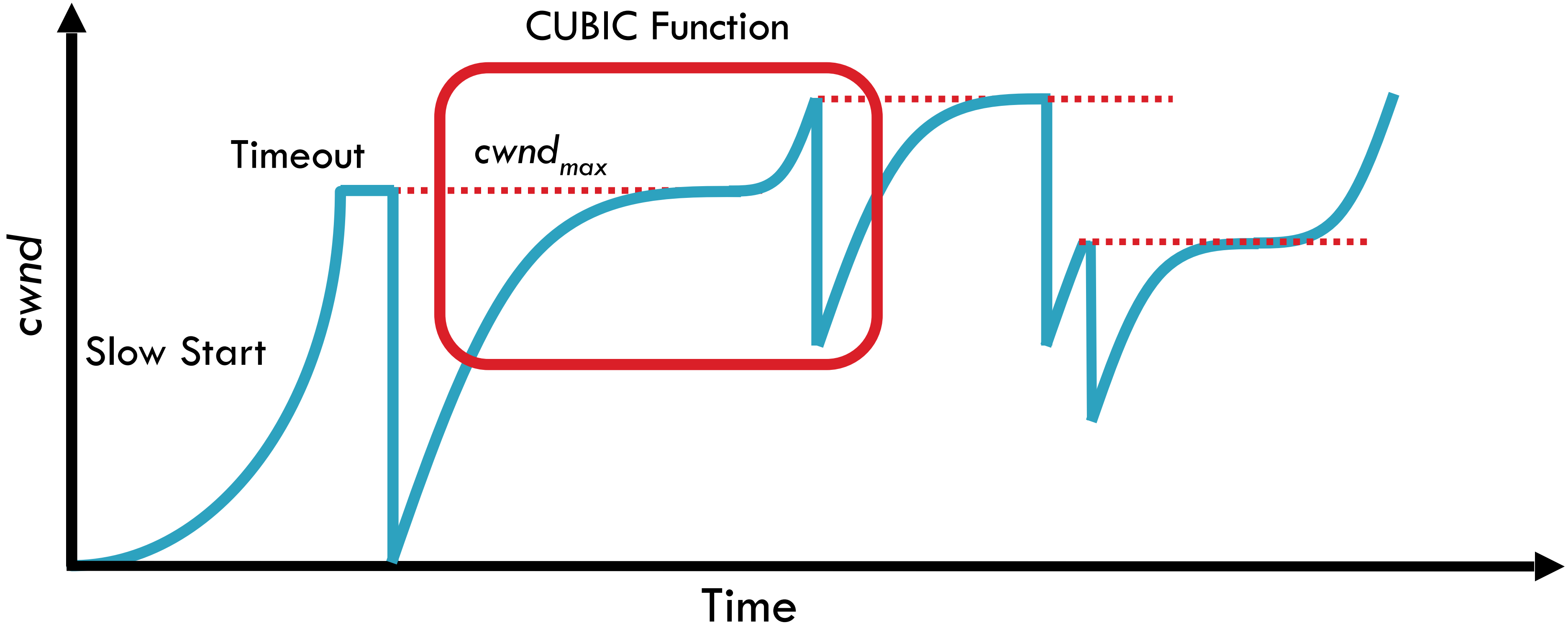


TCP CUBIC Example



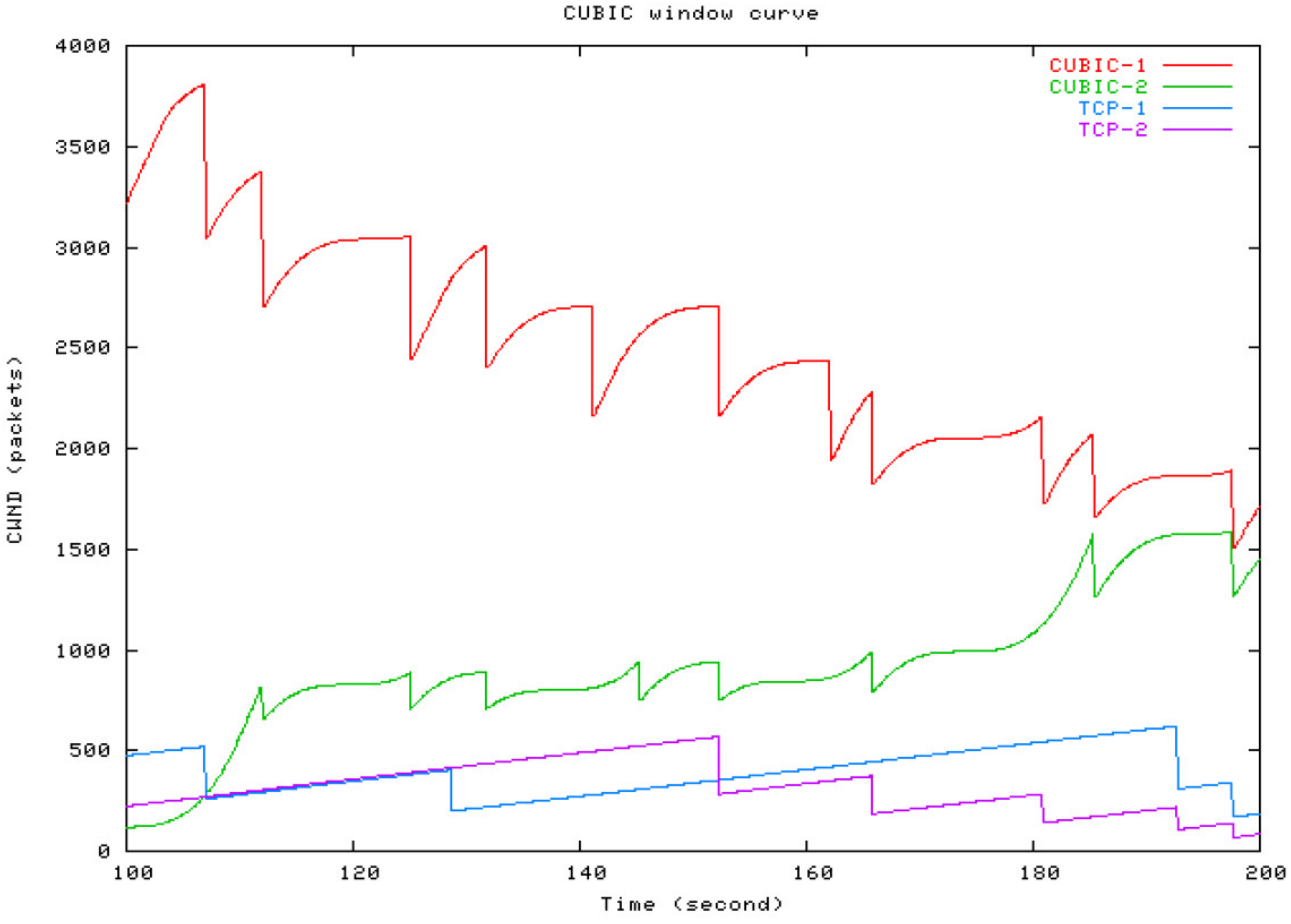
- Less wasted bandwidth due to fast ramp up

TCP CUBIC Example

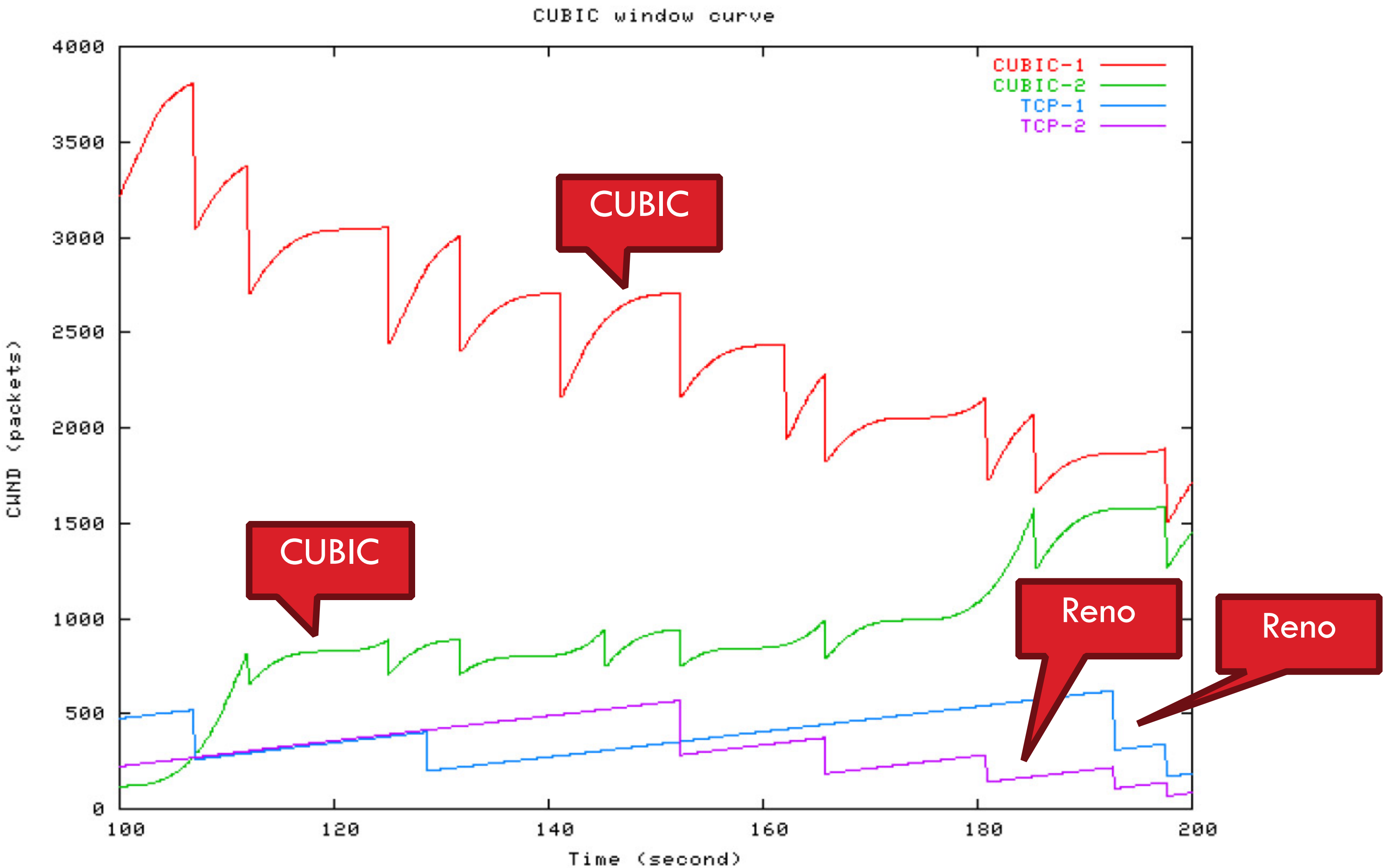


- Less wasted bandwidth due to fast ramp up
- Stable region and slow acceleration help maintain fairness
  - Fast ramp up is more aggressive than additive increase
  - To be fair to Tahoe/Reno, CUBIC needs to be less aggressive

Simulations of CUBIC Flows



Simulations of CUBIC Flows



Deploying TCP Variants

- TCP assumes all flows employ TCP-like congestion control
  - TCP-friendly or TCP-compatible
  - Violated by UDP :(

Deploying TCP Variants

- TCP assumes all flows employ TCP-like congestion control
  - TCP-friendly or TCP-compatible
  - Violated by UDP :(
- If new congestion control algorithms are developed, they must be TCP-friendly

Deploying TCP Variants

- TCP assumes all flows employ TCP-like congestion control
  - TCP-friendly or TCP-compatible
  - Violated by UDP :(
- If new congestion control algorithms are developed, they must be TCP-friendly
- Be wary of unforeseen interactions
  - Variants work well with others like themselves
  - Different variants competing for resources may trigger unfair, pathological behavior

TCP Perspectives

- Cerf/Kahn
  - Provide flow control
  - Congestion handled by retransmission

TCP Perspectives

- Cerf/Kahn
  - Provide flow control
  - Congestion handled by retransmission
- Jacobson / Karels
  - Need to avoid congestion
  - RTT estimates critical
  - Queuing theory can help

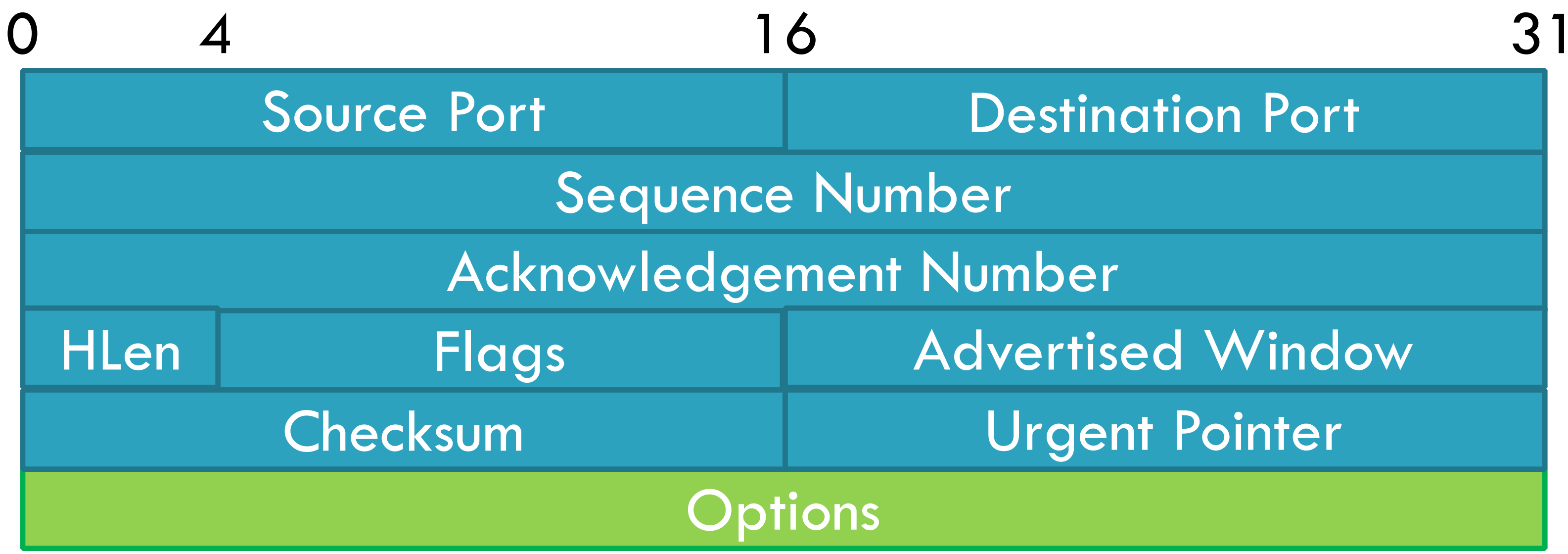
TCP Perspectives

- Cerf/Kahn
  - Provide flow control
  - Congestion handled by retransmission
- Jacobson / Karels
  - Need to avoid congestion
  - RTT estimates critical
  - Queuing theory can help
- Winstein/Balakrishnan
  - TCP is maximizing an objective function
    - Fairness/efficiency
    - Throughput/delay
  - Let a machine pick the best fit for your environment

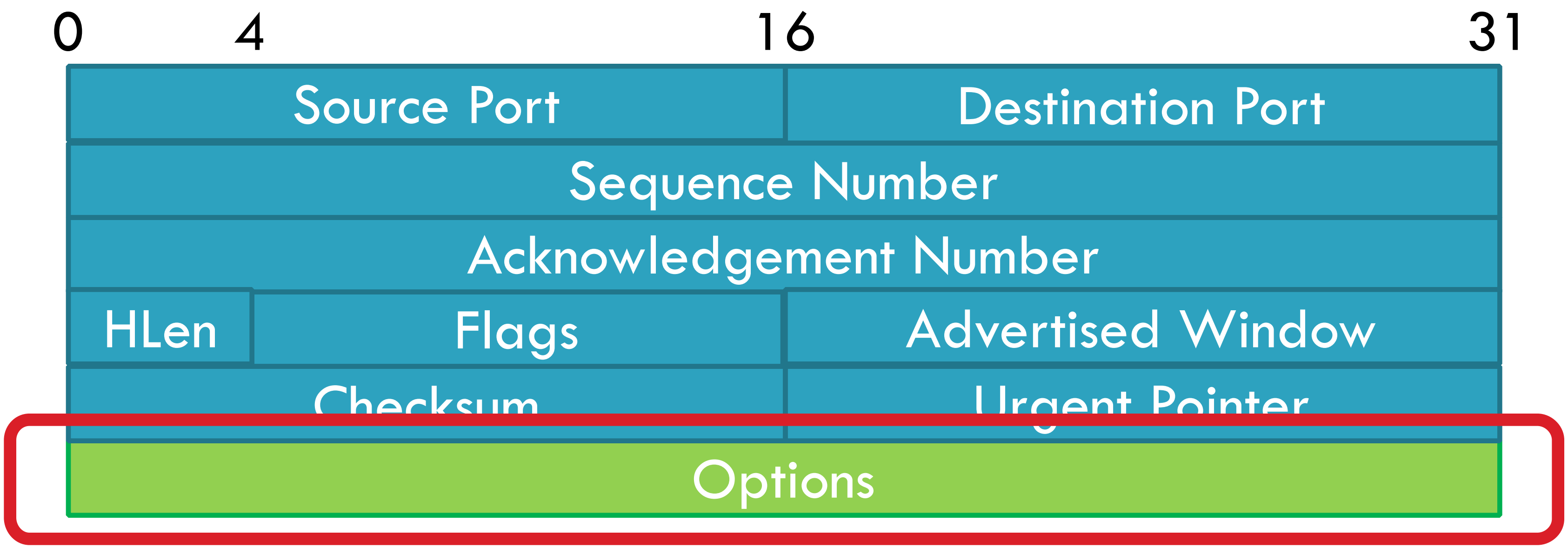
Outline

- Congestion Control
- Evolution of TCP
- Problems with TCP

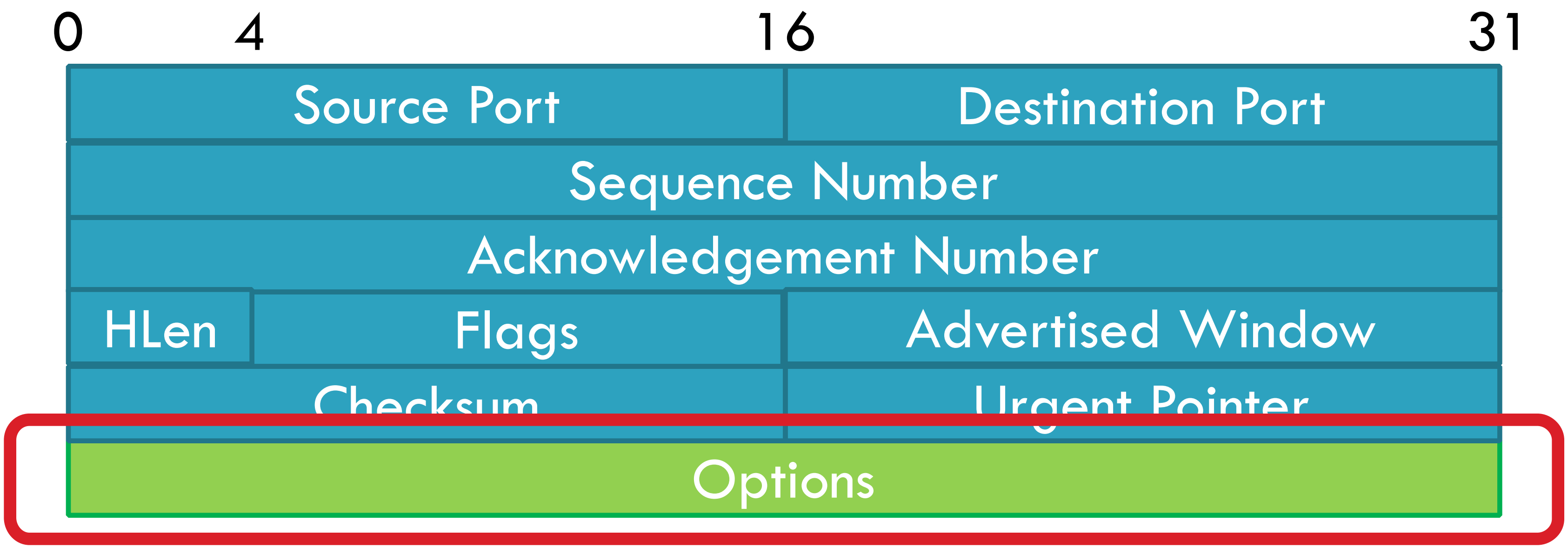
Common TCP Options



Common TCP Options

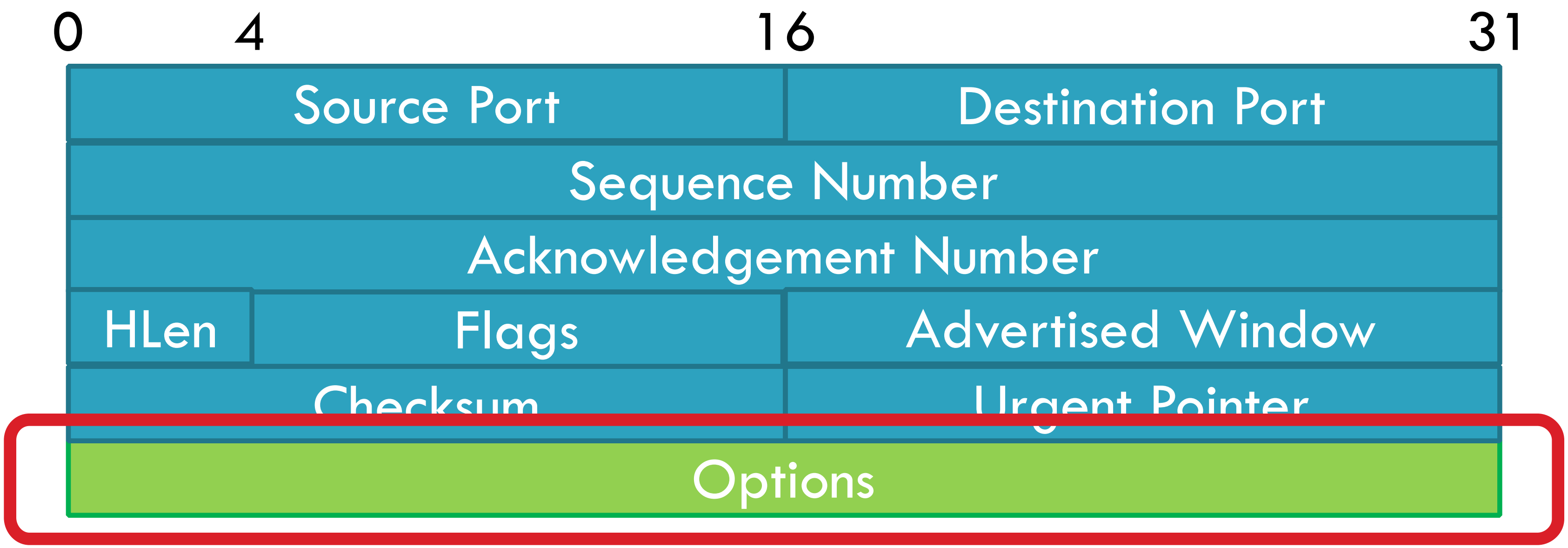


Common TCP Options



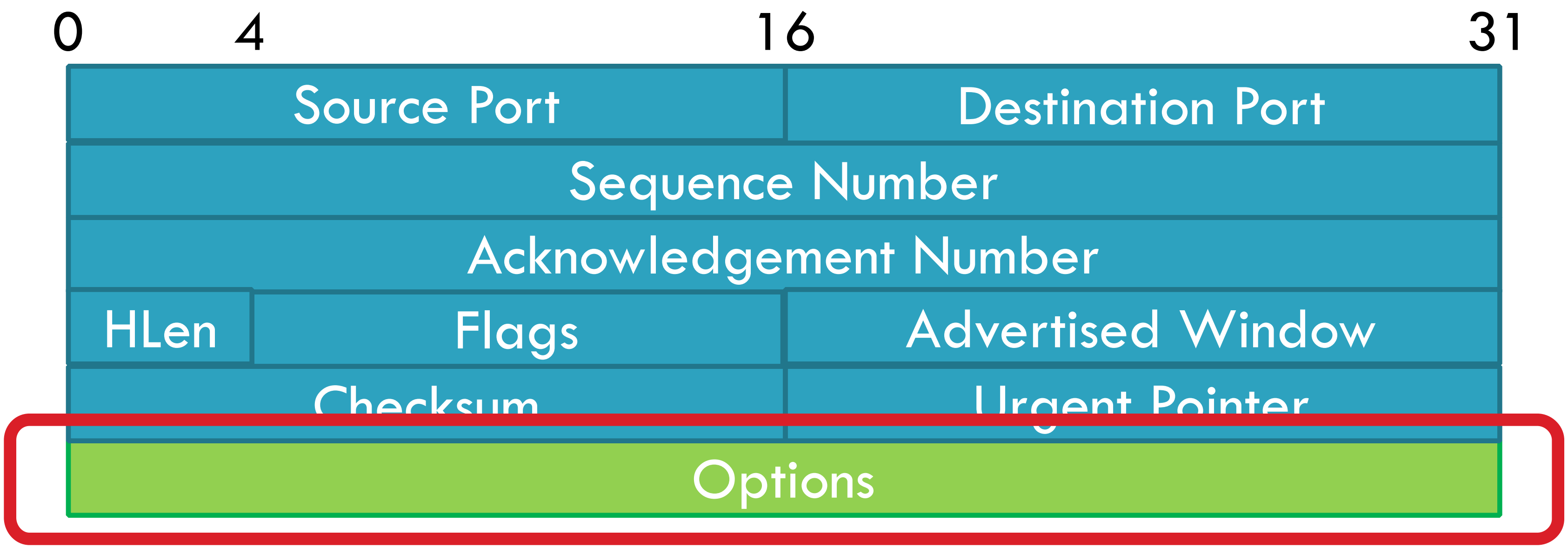
- Window scaling

Common TCP Options



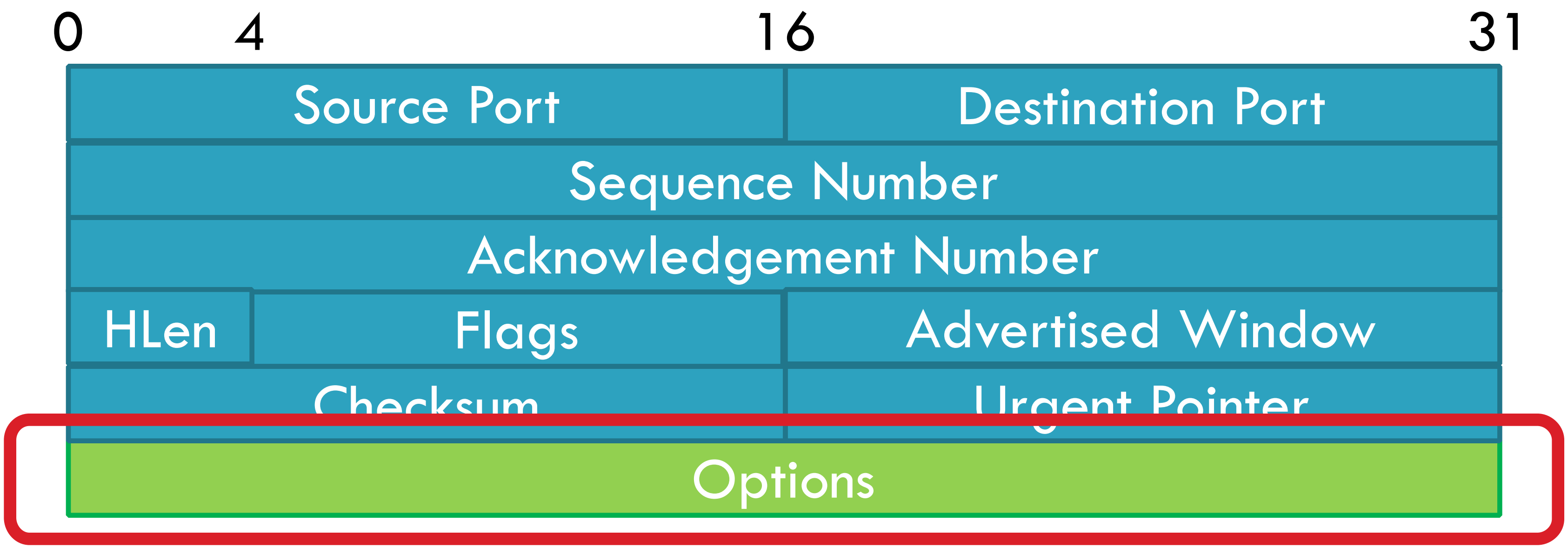
- Window scaling
- SACK: selective acknowledgement

Common TCP Options



- Window scaling
- SACK: selective acknowledgement
- Maximum segment size (MSS)

Common TCP Options



- Window scaling
- SACK: selective acknowledgement
- Maximum segment size (MSS)
- Timestamp

Window Scaling

- Problem: the advertised window is only 16-bits
  - Effectively caps the window at 65536B, 64KB
  - Example: 1.5Mbps link, 513ms RTT

Window Scaling

- Problem: the advertised window is only 16-bits
  - Effectively caps the window at 65536B, 64KB
  - Example: 1.5Mbps link, 513ms RTT

(1.5Mbps * 0.513s) = 94KB

64KB / 94KB = 68% of maximum possible speed

Window Scaling

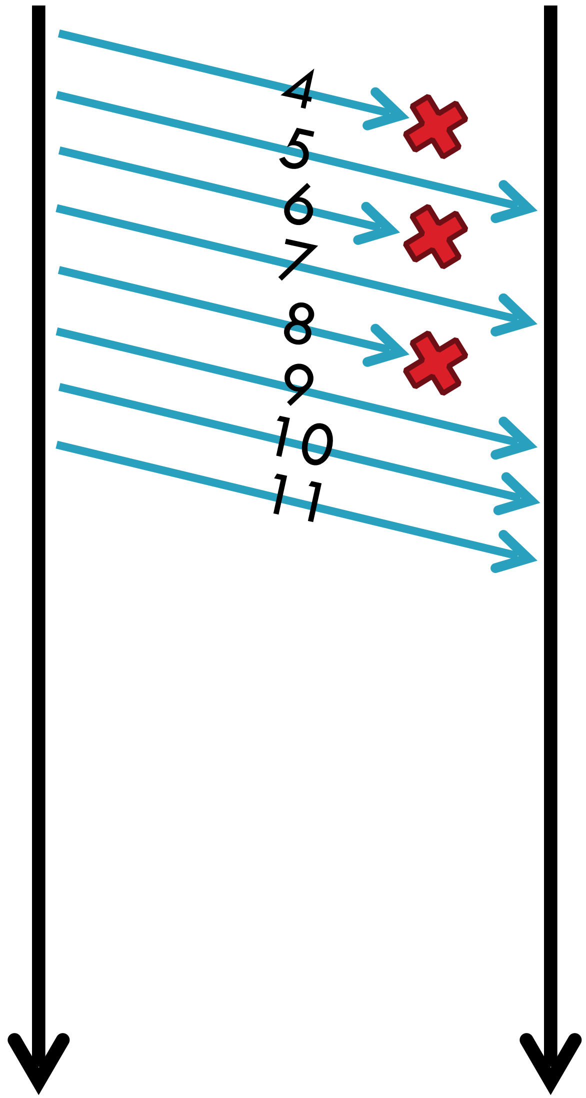
- Problem: the advertised window is only 16-bits
  - Effectively caps the window at 65536B, 64KB
  - Example: 1.5Mbps link, 513ms RTT

(1.5Mbps * 0.513s) = 94KB

64KB / 94KB = 68% of maximum possible speed

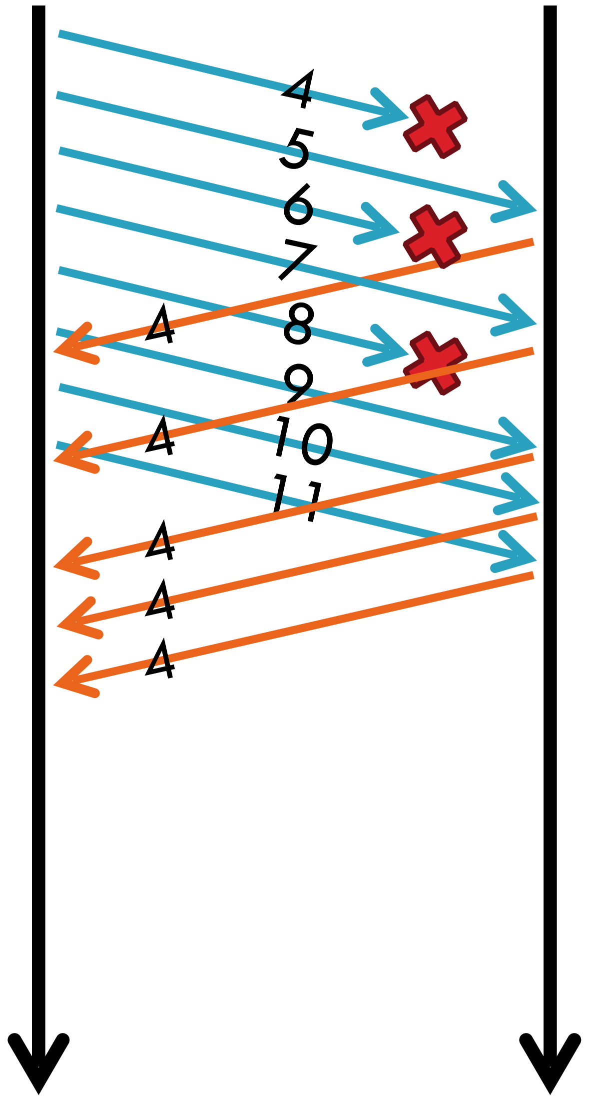
- Solution: introduce a window scaling value
  - *wnd = adv_wnd << wnd_scale;*
  - Maximum shift is 14 bits, 1GB maximum window

SACK: Selective Acknowledgment



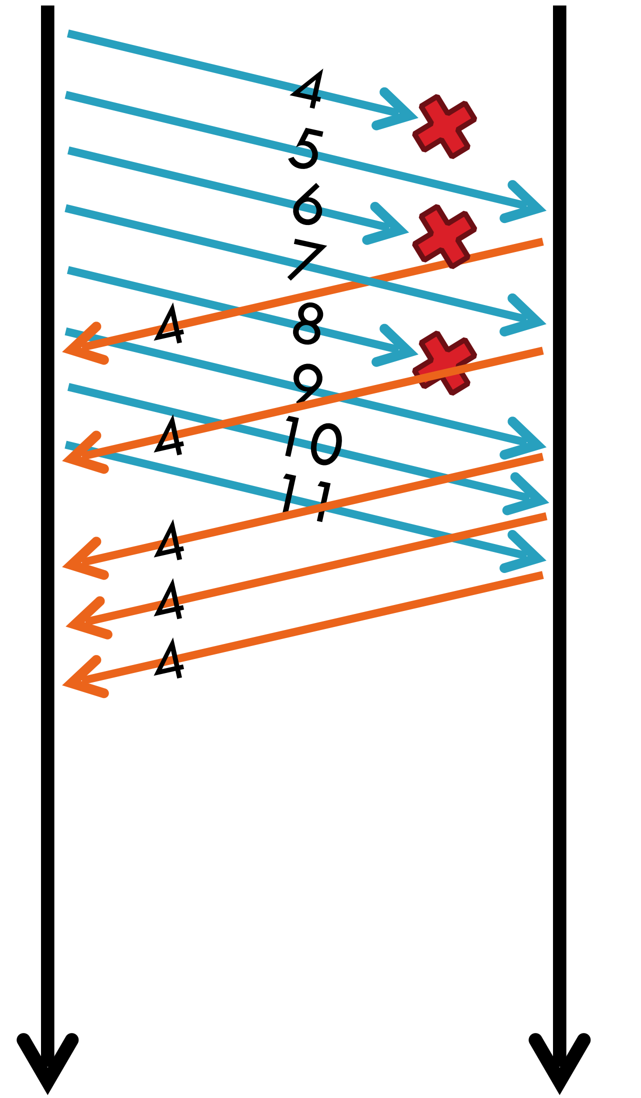
SACK: Selective Acknowledgment

- Problem: duplicate ACKs only tell us about 1 missing packet
  - Multiple rounds of dup ACKs needed to fill all holes



SACK: Selective Acknowledgment

- Problem: duplicate ACKs only tell us about 1 missing packet
  - Multiple rounds of dup ACKs needed to fill all holes
- Solution: selective ACK
  - Include received, out-of-order sequence numbers in TCP header
  - Explicitly tells the sender about holes in the sequence



Other Common Options

- Maximum segment size (MSS)
  - Essentially, what is the hosts MTU
  - Saves on path discovery overhead

Other Common Options

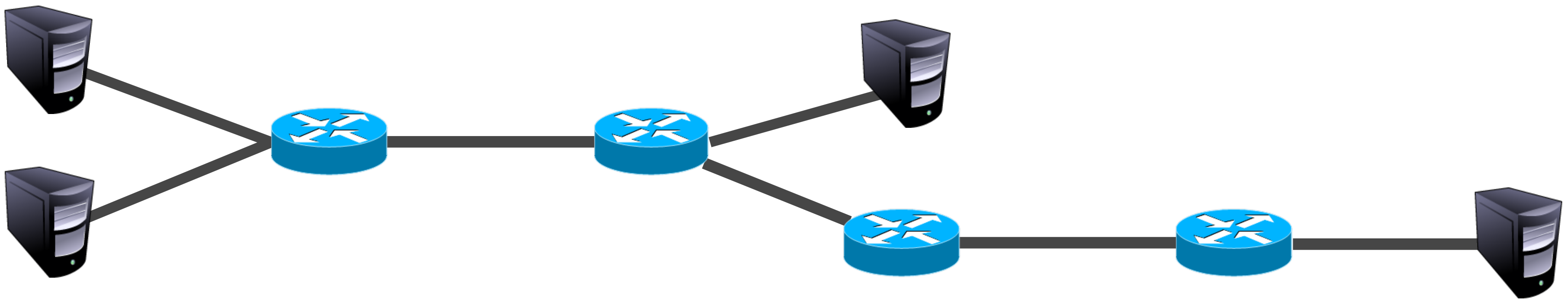
- Maximum segment size (MSS)
  - Essentially, what is the hosts MTU
  - Saves on path discovery overhead
- Timestamp
  - When was the packet sent (approximately)?
  - Used to prevent sequence number wraparound
  - PAWS algorithm

Issues with TCP

- The vast majority of Internet traffic is TCP
- However, many issues with the protocol
  - Lack of fairness
  - Synchronization of flows
  - Poor performance with small flows
  - Really poor performance on wireless networks
  - Susceptibility to denial of service

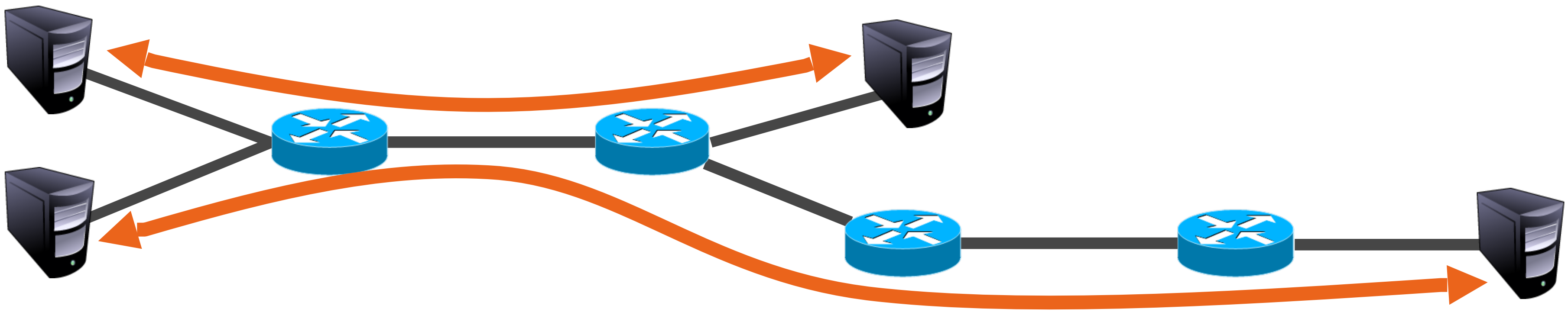
Fairness

- Problem: TCP throughput depends on RTT



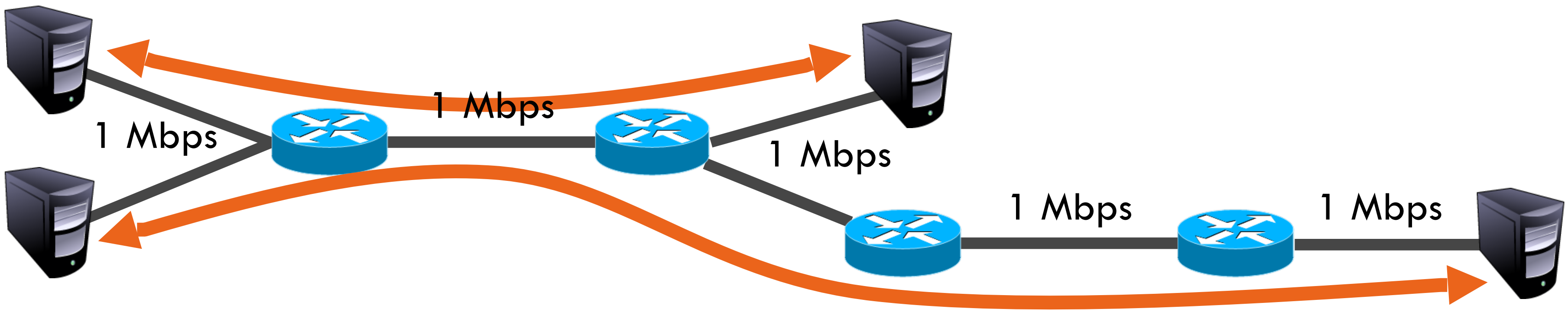
Fairness

- Problem: TCP throughput depends on RTT



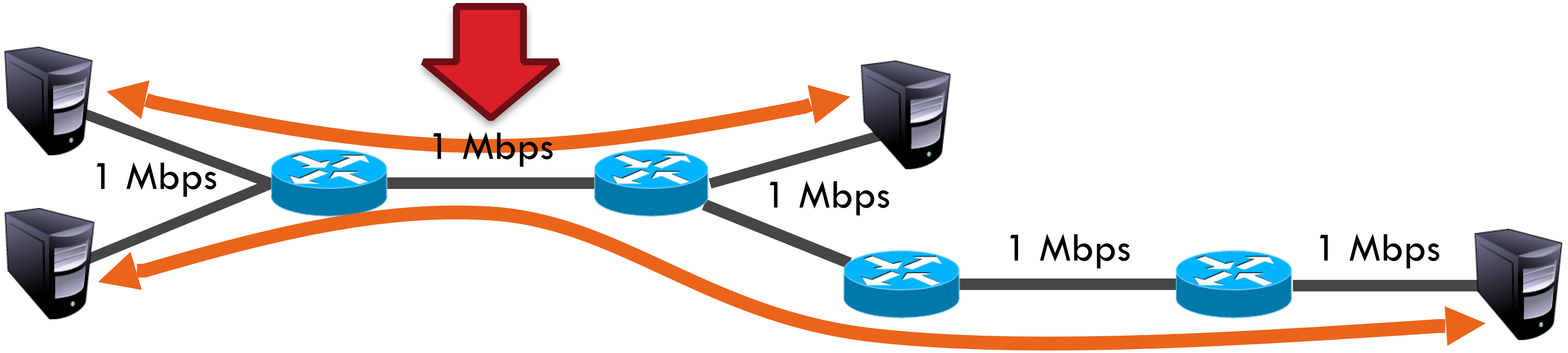
Fairness

- Problem: TCP throughput depends on RTT



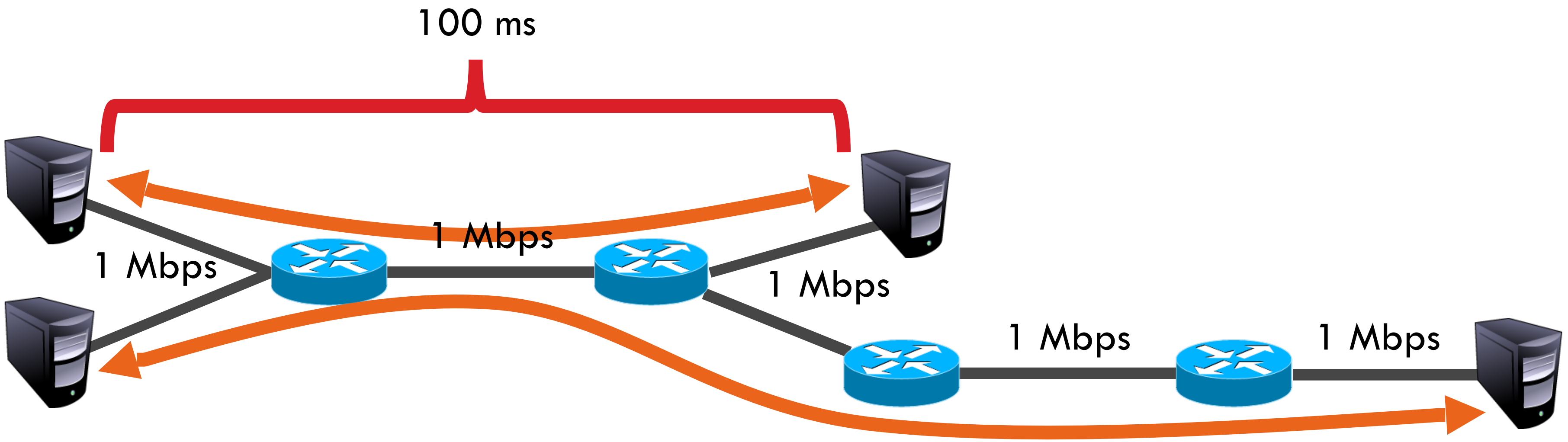
Fairness

- Problem: TCP throughput depends on RTT



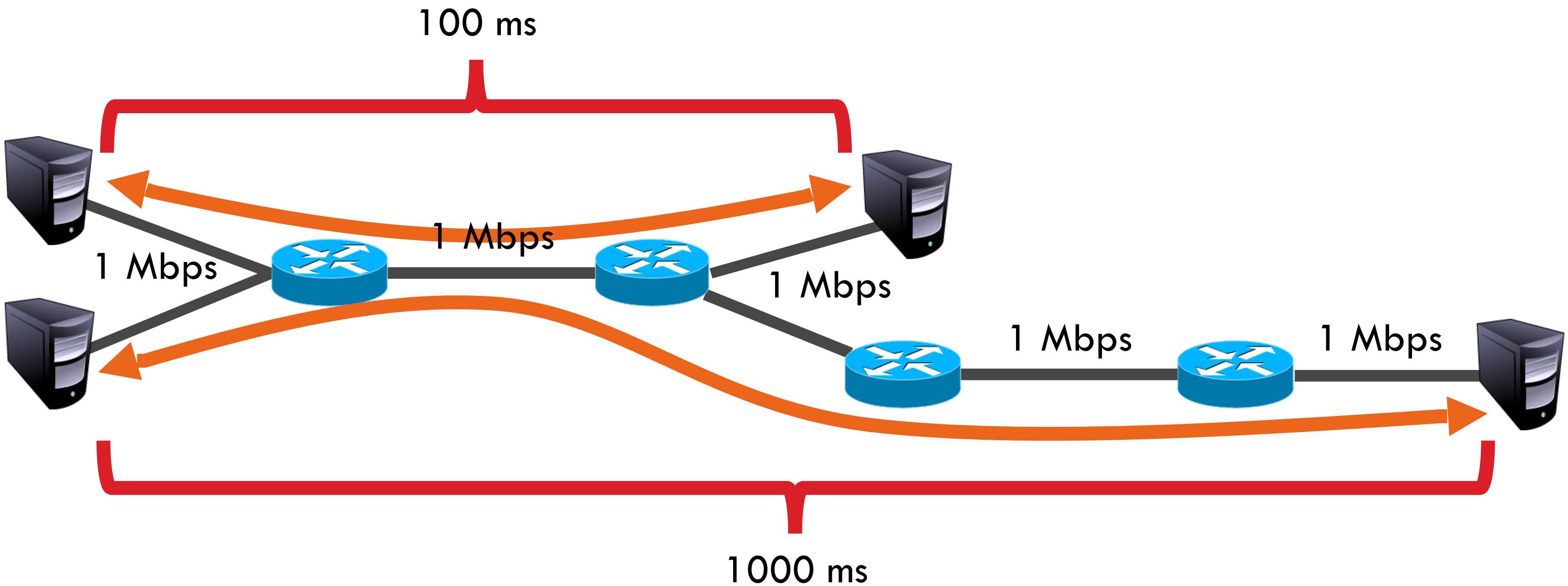
Fairness

- Problem: TCP throughput depends on RTT



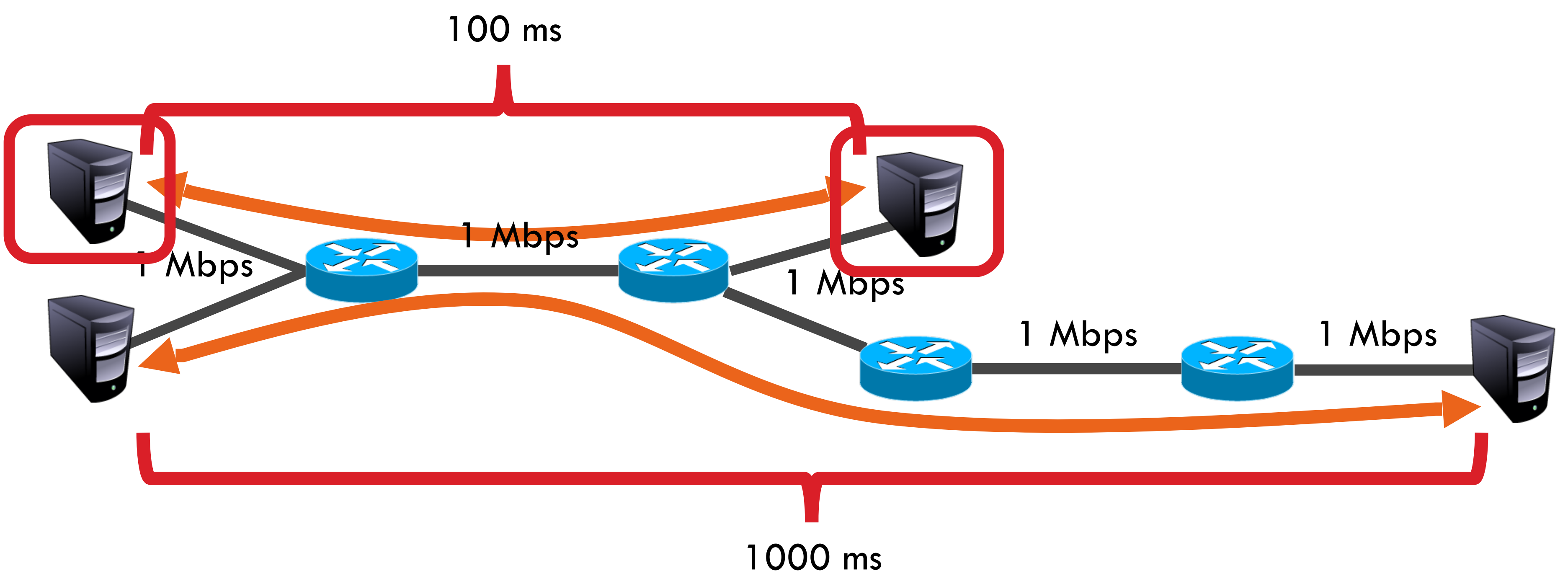
Fairness

- Problem: TCP throughput depends on RTT



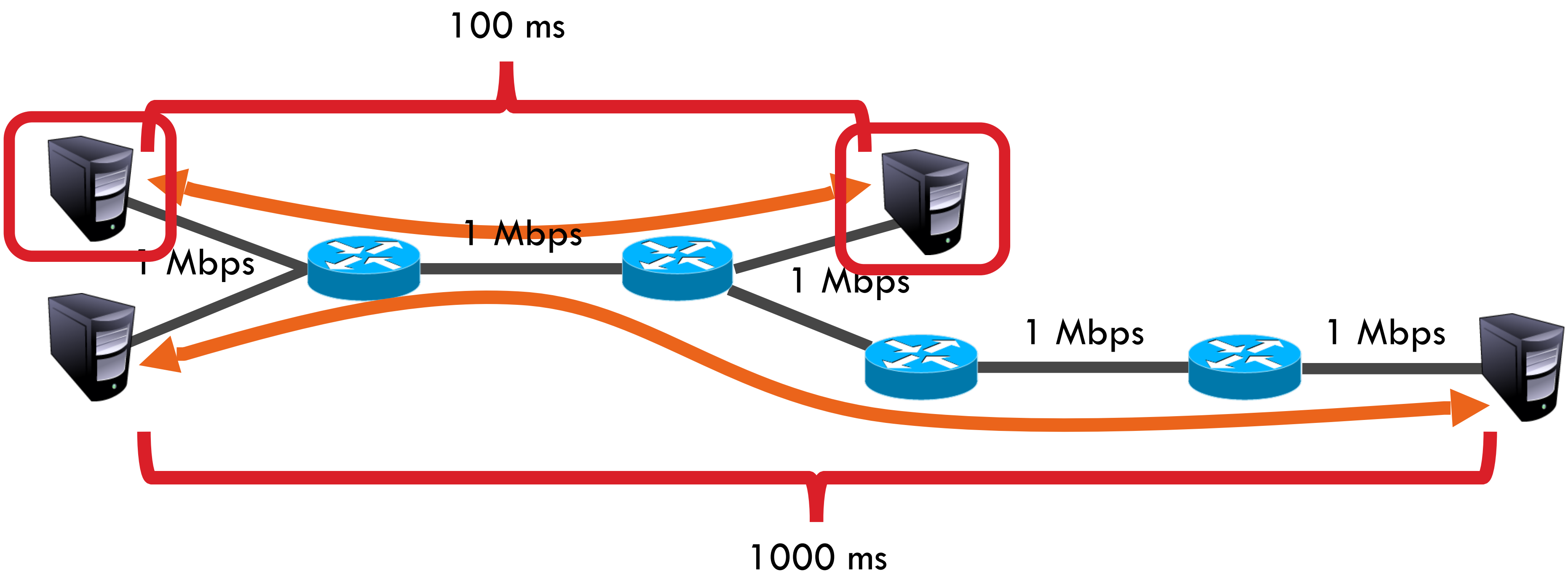
Fairness

- Problem: TCP throughput depends on RTT



Fairness

- Problem: TCP throughput depends on RTT



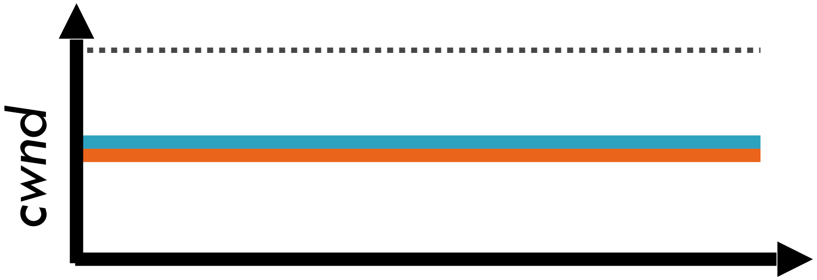
- ACK clocking makes TCP inherently unfair
- Possible solution: maintain a separate delay window
  - Implemented by Microsoft's Compound TCP

Synchronization of Flows

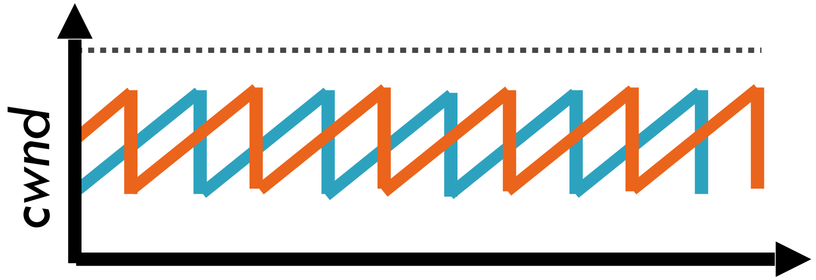
- Ideal bandwidth sharing

Synchronization of Flows

- Ideal bandwidth sharing



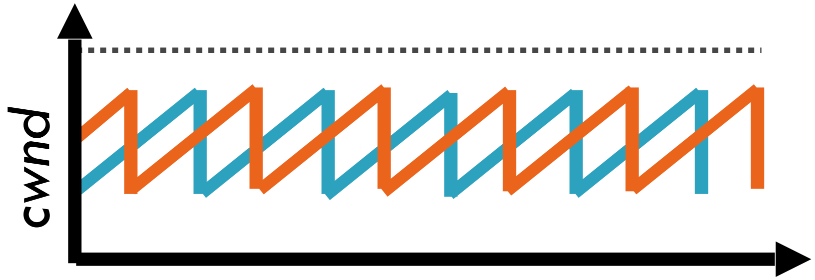
- Oscillating, but high overall utilization



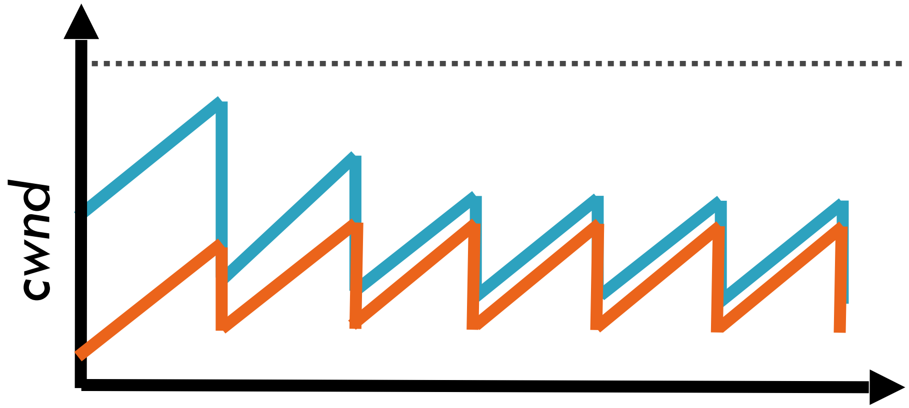
Synchronization of Flows

- Ideal bandwidth sharing

- Oscillating, but high overall utilization

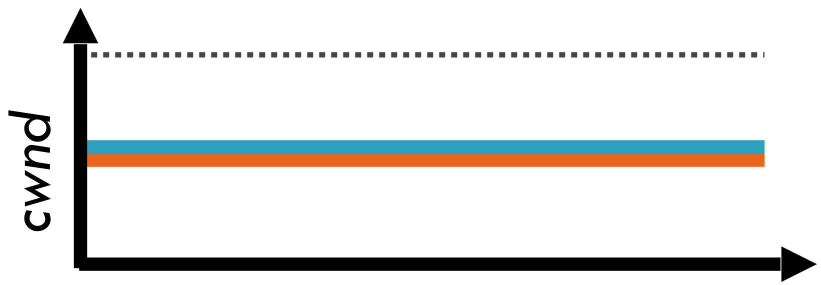


- In reality, flows synchronize

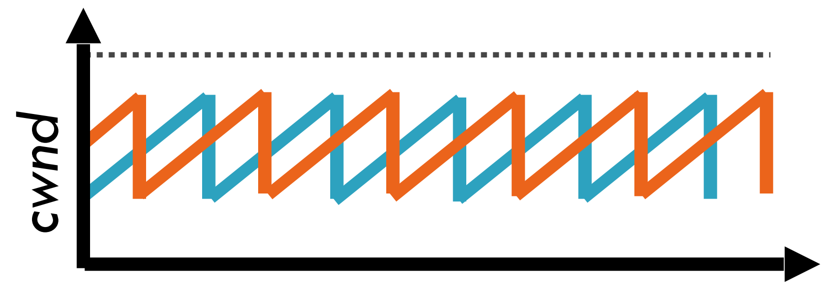


Synchronization of Flows

- Ideal bandwidth sharing

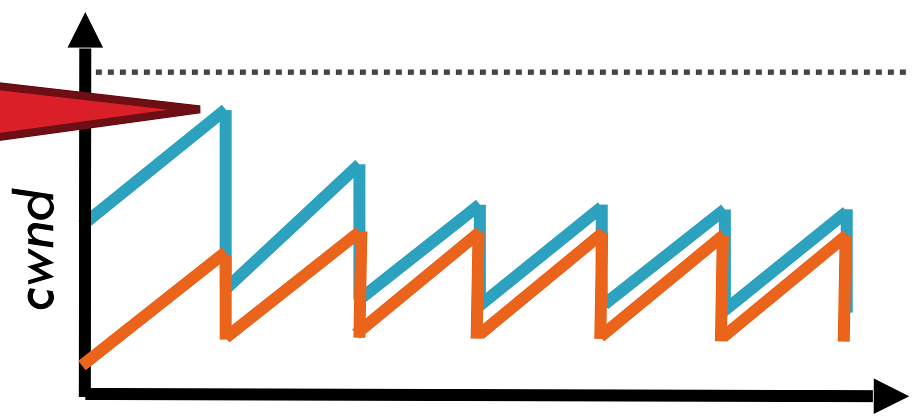


- Oscillating, but high overall utilization



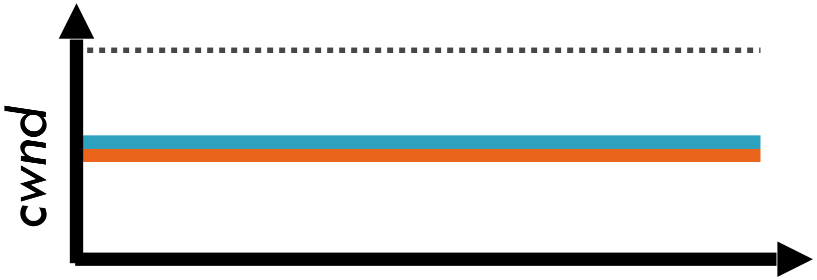
- In reality, flows synchronize

One flow causes all flows to drop packets

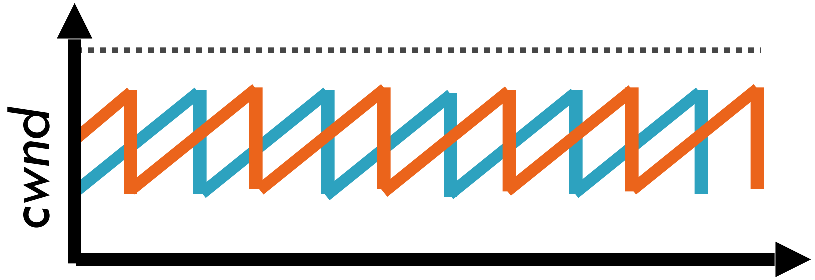


Synchronization of Flows

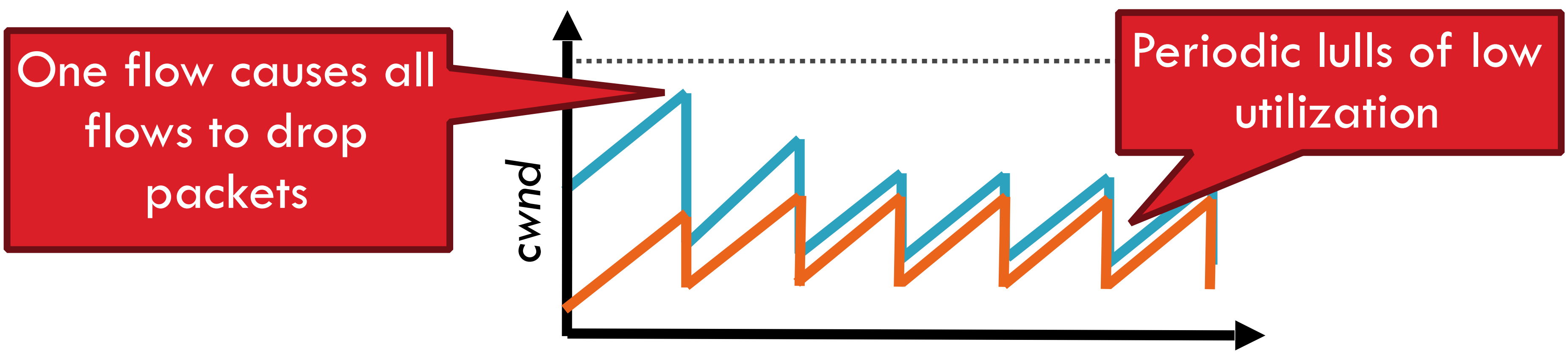
- Ideal bandwidth sharing



- Oscillating, but high overall utilization



- In reality, flows synchronize



Small Flows

- Problem: TCP is biased against short flows
  - 1 RTT wasted for connection setup (SYN, SYN/ACK)
  - *cwnd* always starts at 1

Small Flows

- Problem: TCP is biased against short flows
  - 1 RTT wasted for connection setup (SYN, SYN/ACK)
  - *cwnd* always starts at 1
- Vast majority of Internet traffic is short flows
  - Mostly HTTP transfers, <100KB
  - Most TCP flows never leave slow start!

Small Flows

- Problem: TCP is biased against short flows
  - 1 RTT wasted for connection setup (SYN, SYN/ACK)
  - *cwnd* always starts at 1
- Vast majority of Internet traffic is short flows
  - Mostly HTTP transfers, <100KB
  - Most TCP flows never leave slow start!
- Proposed solutions (driven by Google):
  - Increase initial *cwnd* to 10
  - TCP Fast Open: use cryptographic hashes to identify receivers, eliminate the need for three-way handshake

Wireless Networks

- Problem: Tahoe and Reno assume loss = congestion
  - True on the WAN, bit errors are very rare
  - False on wireless, interference is very common

Wireless Networks

- Problem: Tahoe and Reno assume loss = congestion
  - True on the WAN, bit errors are very rare
  - False on wireless, interference is very common
- TCP throughput ~ 1/sqrt(drop rate)
  - Even a few interference drops can kill performance

Wireless Networks

- Problem: Tahoe and Reno assume loss = congestion
  - True on the WAN, bit errors are very rare
  - False on wireless, interference is very common
- TCP throughput ~ 1/sqrt(drop rate)
  - Even a few interference drops can kill performance
- Possible solutions:
  - Break layering, push data link info up to TCP
  - Use delay-based congestion detection (TCP Vegas)
  - Explicit congestion notification (ECN)

Denial of Service

- Problem: TCP connections require state
  - Initial SYN allocates resources on the server
  - State must persist for several minutes (RTO)

Denial of Service

- Problem: TCP connections require state
  - Initial SYN allocates resources on the server
  - State must persist for several minutes (RTO)
- SYN flood: send enough SYNs to a server to allocate all memory/meltdown the kernel

Denial of Service

- Problem: TCP connections require state
  - Initial SYN allocates resources on the server
  - State must persist for several minutes (RTO)
- SYN flood: send enough SYNs to a server to allocate all memory/meltdown the kernel
- Solution: SYN cookies
  - Idea: don't store initial state on the server
  - Securely insert state into the SYN/ACK packet
  - Client will reflect the state back to the server

SYN Cookies

SYN Cookies

SYN Cookies

- Did the client really send me a SYN recently?
  - Timestamp: freshness check
  - Cryptographic hash: prevents spoofed packets

SYN Cookies

- Did the client really send me a SYN recently?
  - Timestamp: freshness check
  - Cryptographic hash: prevents spoofed packets
- Maximum segment size (MSS)
  - Usually stated by the client during initial SYN
  - Server should store this value…
  - Reflect the clients value back through them

SYN Cookies in Practice

- Advantages
  - Effective at mitigating SYN floods
  - Compatible with all TCP versions
  - Only need to modify the server
  - No need for client support

SYN Cookies in Practice

- Advantages
  - Effective at mitigating SYN floods
  - Compatible with all TCP versions
  - Only need to modify the server
  - No need for client support
- Disadvantages
  - MSS limited to 3 bits, may be smaller than clients actual MSS
  - Server forgets all other TCP options included with the client's SYN
    - SACK support, window scaling, etc.