CS 4700 / CS 5700
Network Fundamentals

Lecture 11: Transport

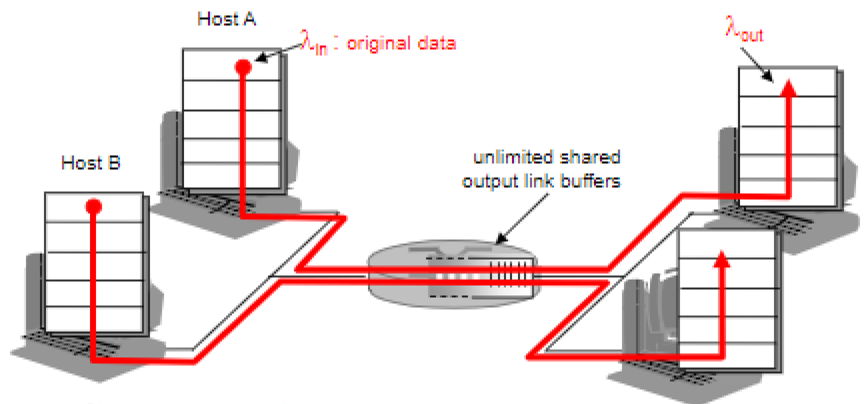
(UDP, but mostly TCP)

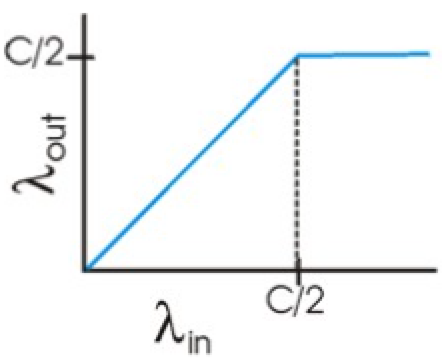
Revised 7/27/2013

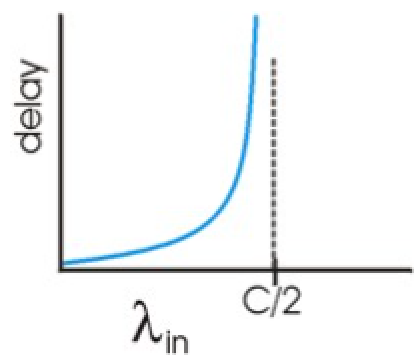
Causes/Costs of Congestion

Scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



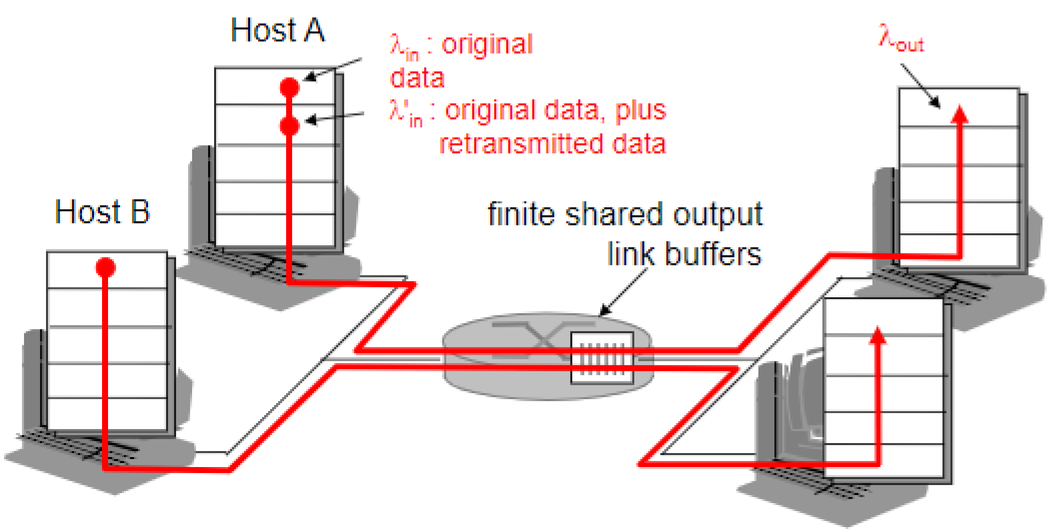




- large delays when congested
- maximum achievable throughput

Causes/Costs of Congestion Scenario 2

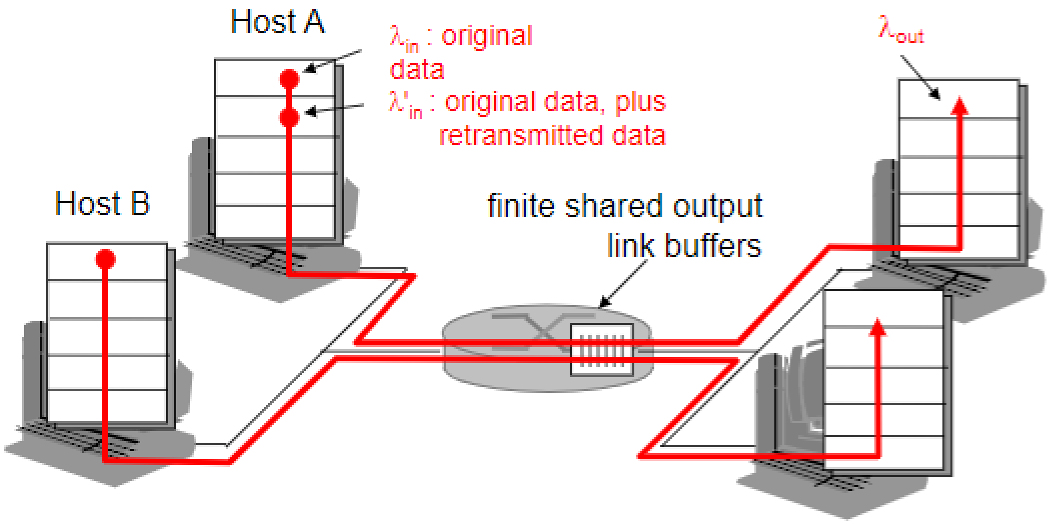
- one router, **finite** buffers
- sender retransmits lost packets



Causes/Costs of Congestion

Scenario 2

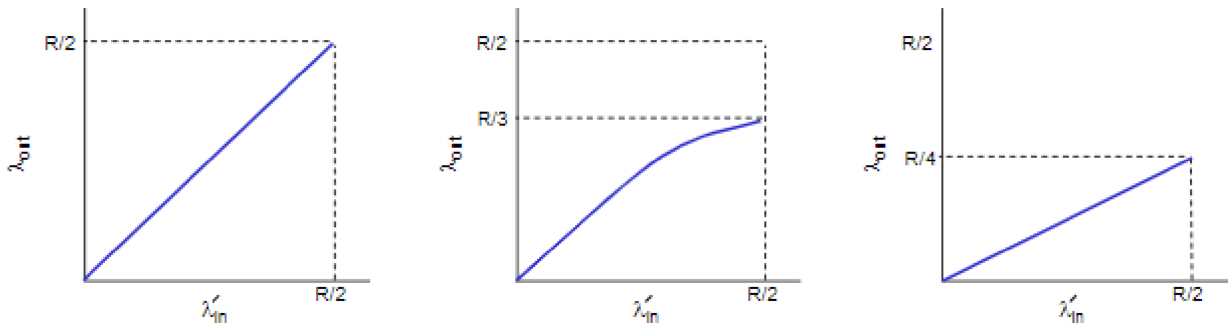
- one router, finite buffers
- sender retransmits lost packets



Causes/Costs of Congestion

Scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a. b. c.

"costs" of congestion:

- more work (retransmissions) for a given "goodput"
- unneeded retransmissions: link carries multiple copies of packet

TCP Congestion Control Summary

- Important TCP Congestion Control ideas include: **AIMD, Slow Start, Fast Retransmit and Fast Recovery** .
- Know the differences between **TCP Tahoe, TCP Reno and TCP New Reno** .
- Currently, the two most common versions of TCP are Compound (Windows) and Cubic (Linux).
- TCP needs rules and an algorithm to determine **RTO and RTO**.

Approaches towards Congestion Control

Two broad approaches towards congestion control:

end-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should use for sending.

TCP Congestion Control

- Essential strategy :: The TCP host sends packets into the network without a reservation and then the host reacts to observable events.
- Originally TCP assumed FIFO queuing.
- Basic idea :: each source determines how much capacity is available to a given flow in the network.
- ACKs are used to *'pace'* the transmission of packets such that TCP is "self-clocking".

TCP Congestion Control K & R

- **Goal:** TCP sender should transmit as fast as possible, but without congesting network.
  - issue - how to find rate **just below** congestion level?
- Each TCP sender sets its window size, based on implicit feedback:
  - **ACK** segment received → network is not congested, so increase sending rate.
  - **lost segment -** assume loss due to congestion, so decrease sending rate.

TCP Congestion Control [K & R]

- **Goal:** TCP sender should transmit as fast as possible, but without congesting network.
  - issue - how to find rate **just below** congestion level?
- Each TCP sender sets its window size, based on implicit feedback:
  - **ACK** segment received ➔ network is not congested, so increase sending rate.
  - **lost segment -** assume loss due to congestion, so decrease sending rate.

TCP Congestion Control

K & R

- **"probing for bandwidth":** increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network).

AIMD (Additive Increase / Multiplicative Decrease)

- CongestionWindow (cwnd) is a variable held by the TCP source for each connection.

MaxWindow :: min (CongestionWindow , AdvertisedWindow)

EffectiveWindow = MaxWindow – (LastByteSent -LastByteAcked)

- cwnd is set based on the perceived level of congestion. The Host receives implicit (packet drop) or explicit (packet mark) as an indication of internal congestion.

Additive Increase (AI)

- Additive Increase is a reaction to perceived available capacity (referred to as *congestion avoidance stage*).
- Frequently in the literature, additive increase is defined by parameter α (where the default is $\alpha = 1$).
- Linear Increase :: For each "cwnd's worth" of packets sent, increase cwnd by 1 packet.
- In practice, **cwnd** is incremented **fractionally** for each arriving ACK.

increment = MSS x (MSS /cwnd)
cwnd = cwnd + increment

Multiplicative Decrease (MD)

* Key assumption :: a dropped packet and resultant timeout are due to congestion at a router.
- Frequently in the literature, multiplicative decrease is defined by parameter β (where the default is β = 0.5)

Multiplicate Decrease:: TCP reacts to a timeout by halving **cwnd**.

- Although defined in bytes, the literature often discusses **cwnd** in terms of packets (or more formally in MSS == Maximum Segment Size).
- **cwnd** is not allowed below the size of a single packet.

AIMD (Additive Increase / Multiplicative Decrease)

- It has been shown that AIMD is a necessary condition for TCP congestion control to be stable.
- Because the simple CC mechanism involves timeouts that cause retransmissions, it is important that hosts have an accurate timeout mechanism.
- Timeouts set as a function of average RTT and standard deviation of RTT.
- However, TCP hosts only sample round-trip time once per RTT using coarse-grained clock.

Slow Start

- Linear additive increase takes **too long** to ramp up a new TCP connection from cold start.
- Beginning with TCP Tahoe, the slow start mechanism was added to provide an initial exponential increase in the size of **cwnd**.

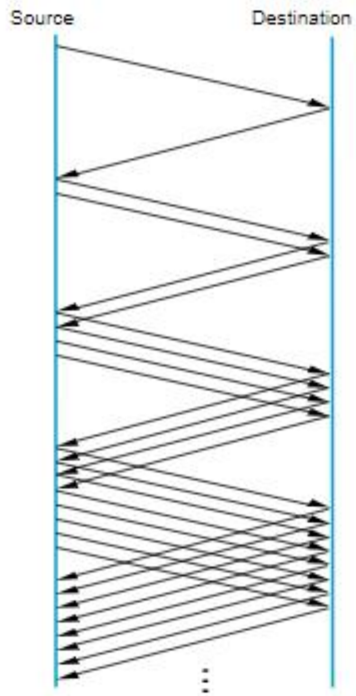
Remember mechanism by: **slow start <u>prevents</u> a slow start. Moreover, slow start is slower than sending a full advertised window's worth of packets all at once.**

Slow Start

- The source starts with cwnd = 1.
- Every time an ACK arrives, cwnd is incremented.

→ cwnd is effectively doubled per RTT "epoch".

- Two **slow start** situations:
  - At the very beginning of a connection {cold start} .
  - When the connection goes **dead** waiting for a timeout to occur (i.e, when the **advertized window** goes to zero!)

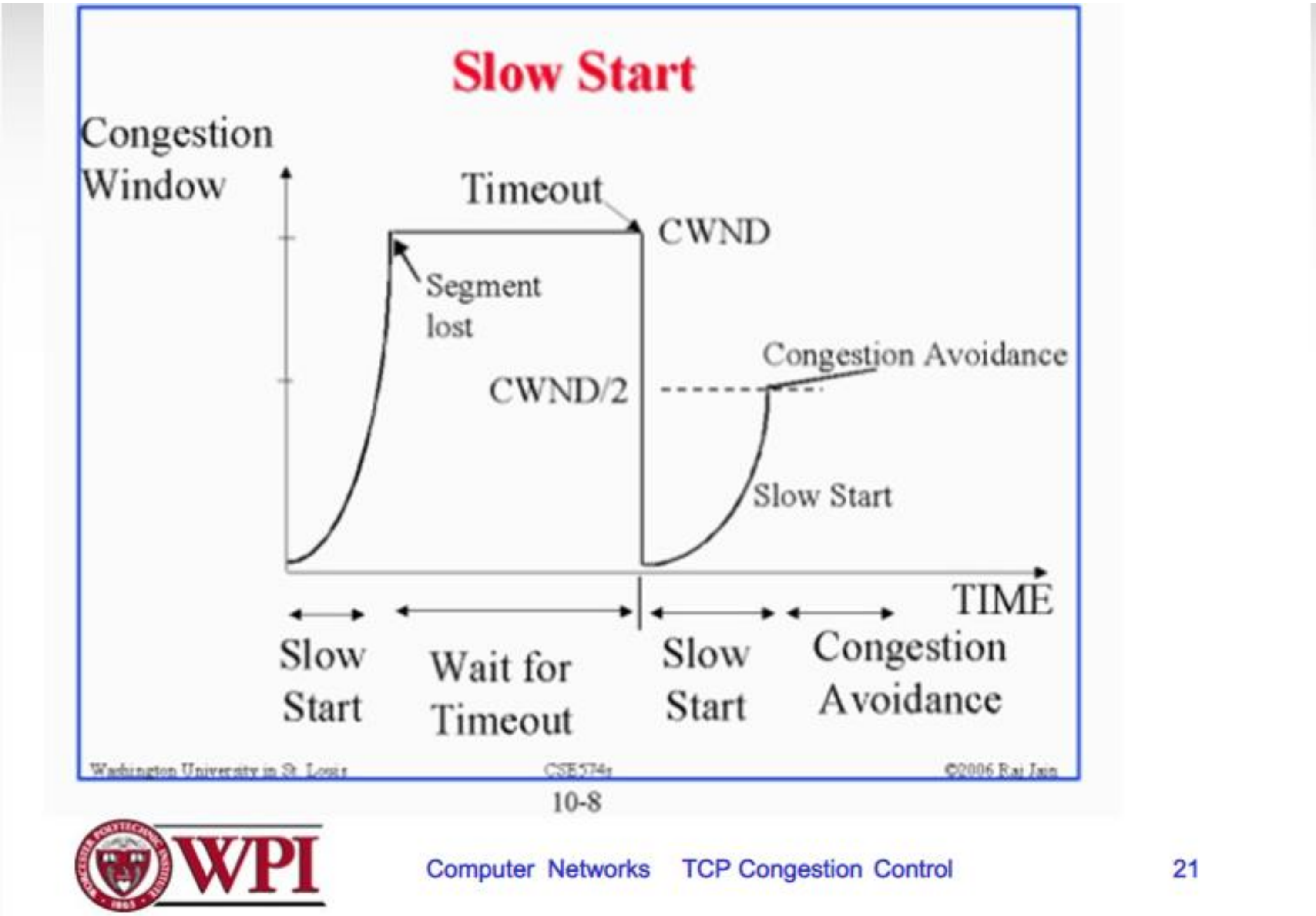


Slow Start
Add one packet per ACK

Figure 6.10 Slow Start

Slow Start

- However, in the second case the source has more information. The current value of cwnd can be saved as a congestion threshold .
- This is also known as the "slow start threshold" *ssthresh*.


Slow Start
Congestion Window
Timeout
CWND
Segment lost
CWND/2
Congestion Avoidance
Slow Start
TIME
Slow Start
Wait for Timeout
Slow Start
Congestion Avoidance
Washington University in St. Louis
CSE574s
©2006 Raj Jain
10-8
WPI
Computer Networks TCP Congestion Control
21

Fast Retransmit

- Coarse timeouts remained a problem, and Fast retransmit was added with **TCP Tahoe** .
- Since the receiver responds every time a packet arrives, this implies the sender will see duplicate ACKs.

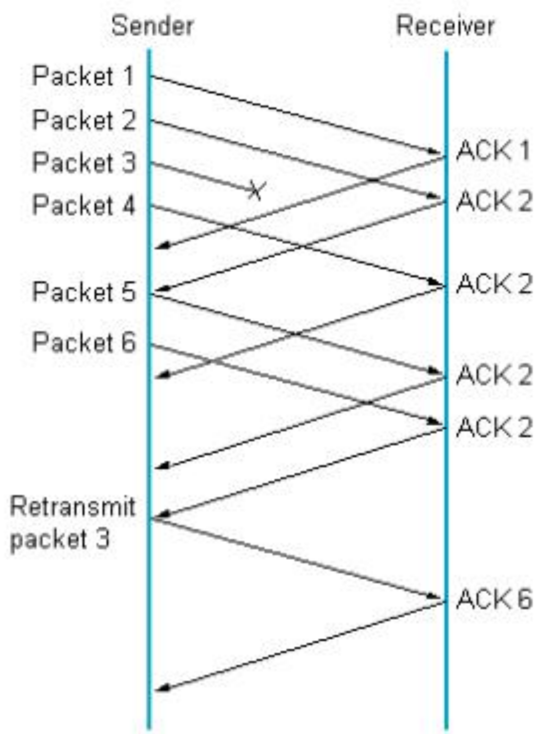
Basic Idea:: *use **duplicate ACKs** to signal lost packet.*

Fast Retransmit

Upon receipt of ***three*** duplicate ACKs, the TCP Sender retransmits the lost packet.

Fast Retransmit

- Generally, fast retransmit eliminates about **half** the coarse-grain timeouts.
- This yields roughly a 20% improvement in throughput.
- Note – fast retransmit does not eliminate all the timeouts due to small window sizes at the source.

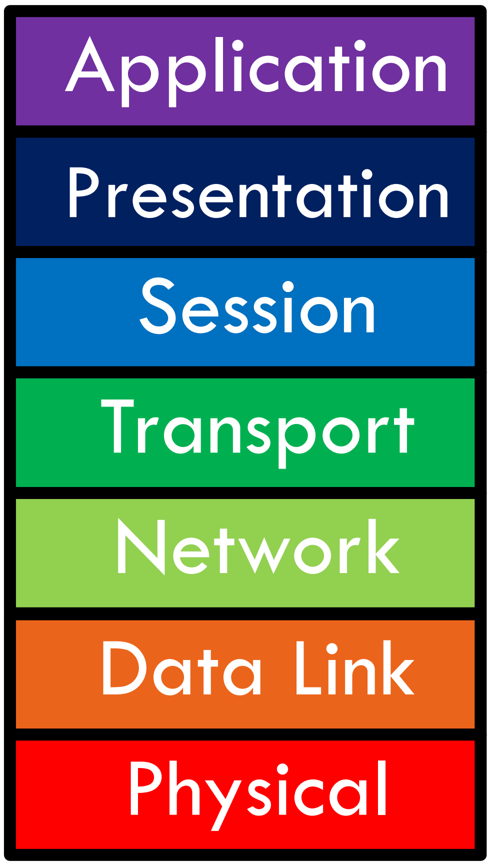


Fast Retransmit

Based on three duplicate ACKs

Figure 6.12 Fast Retransmit

Transport Layer



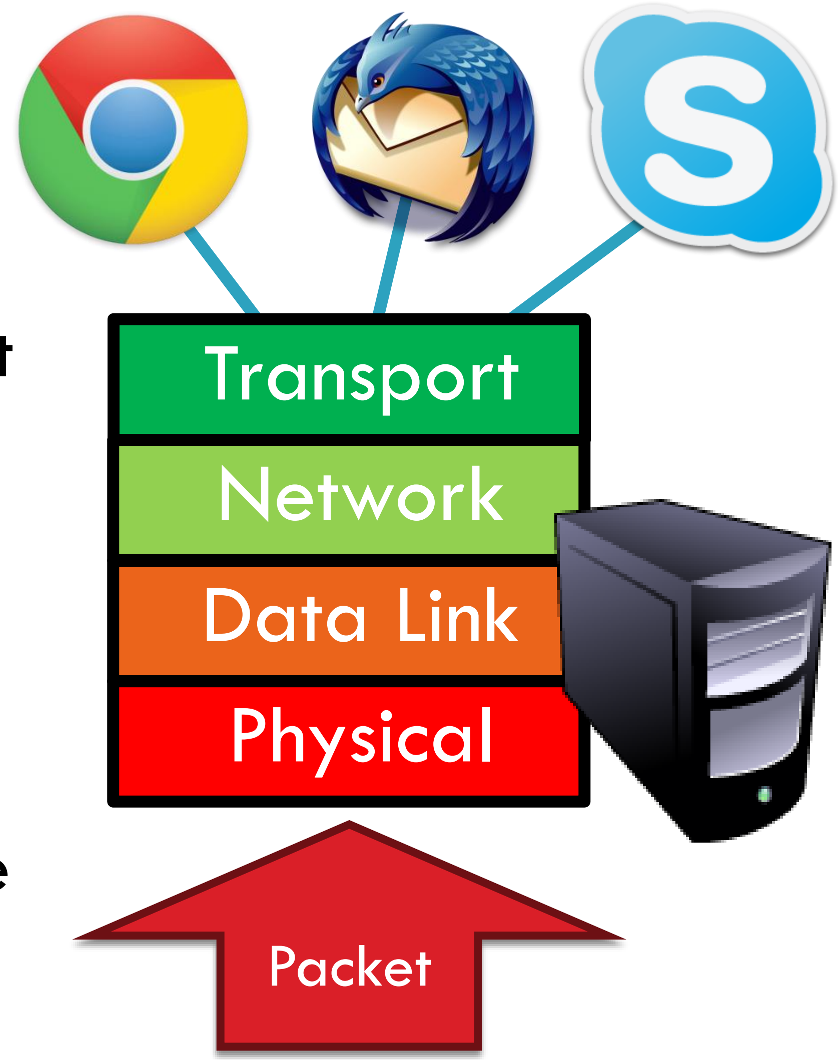
- Function:
  - Demultiplexing of data streams
- Optional functions:
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control
- Key challenges:
  - Detecting and responding to congestion
  - Balancing fairness against high utilization

Outline

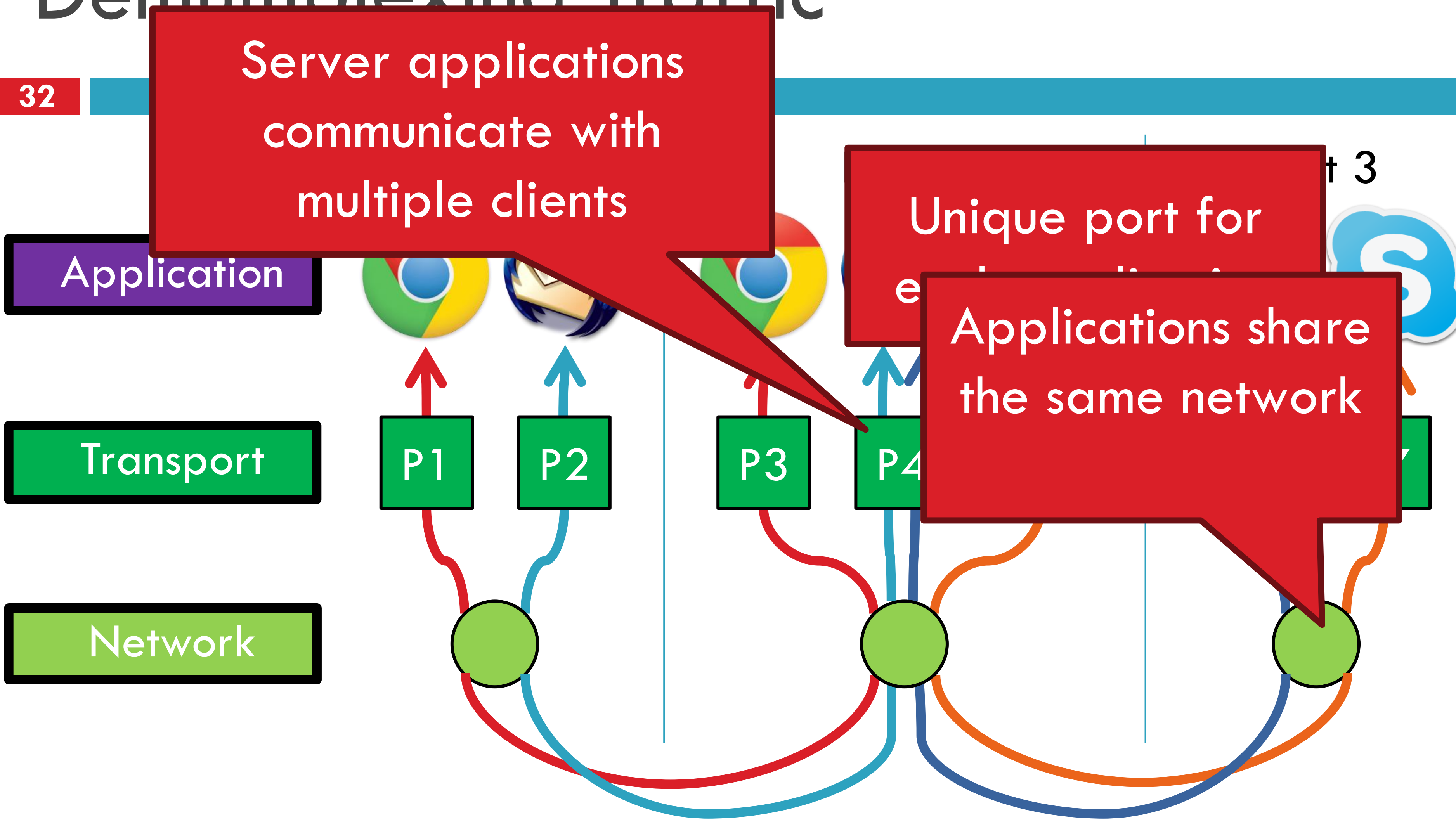
- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

The Case for Multiplexing

- Datagram network
  - No circuits
  - No connections
- Clients run many applications at the same time
  - Who to deliver packets to?
- IP header "protocol" field
  - 8 bits = 256 concurrent streams
- Insert Transport Layer to handle demultiplexing

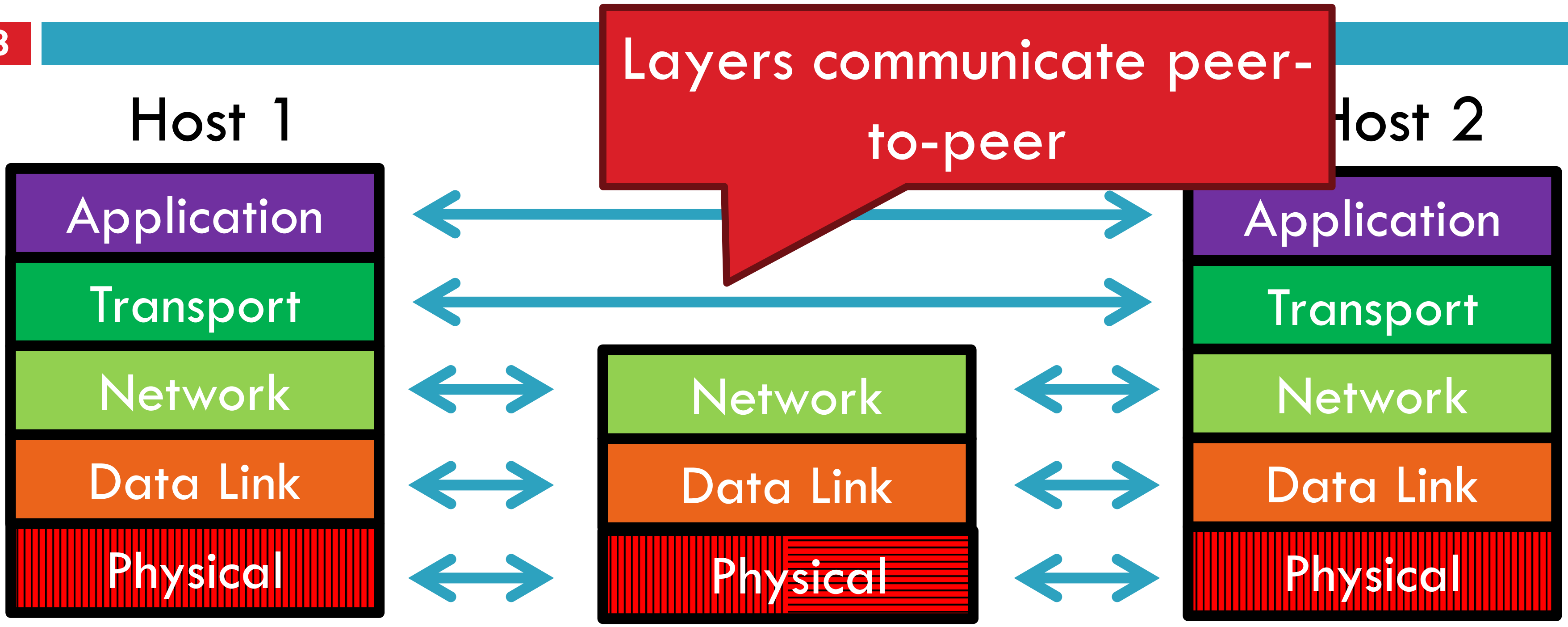


Demultiplexing Traffic



Endpoints identified by <*src_ip, src_port, dest_ip, dest_port*>

Layering, Revisited



- Lowest level end-to-end protocol (in theory)
  - Transport header only read by source and destination
  - Routers view transport header as payload

User Datagram Protocol (UDP)

0 16 31

Source Port	Destination Port
Payload Length	Checksum

- Simple, connectionless datagram
  - C sockets: SOCK_DGRAM
- Port numbers enable demultiplexing
  - 16 bits = 65535 possible ports
  - Port 0 is invalid
- Checksum for error detection
  - Detects (some) corrupt packets
  - Does not detect dropped, duplicated, or reordered packets

Uses for UDP

- Invented after TCP
  - Why?
- Not all applications can tolerate TCP
- Custom protocols can be built on top of UDP
  - Reliability? Strict ordering?
  - Flow control? Congestion control?
- Examples
  - RTMP, real-time media streaming (e.g. voice, video)
  - Facebook datacenter protocol

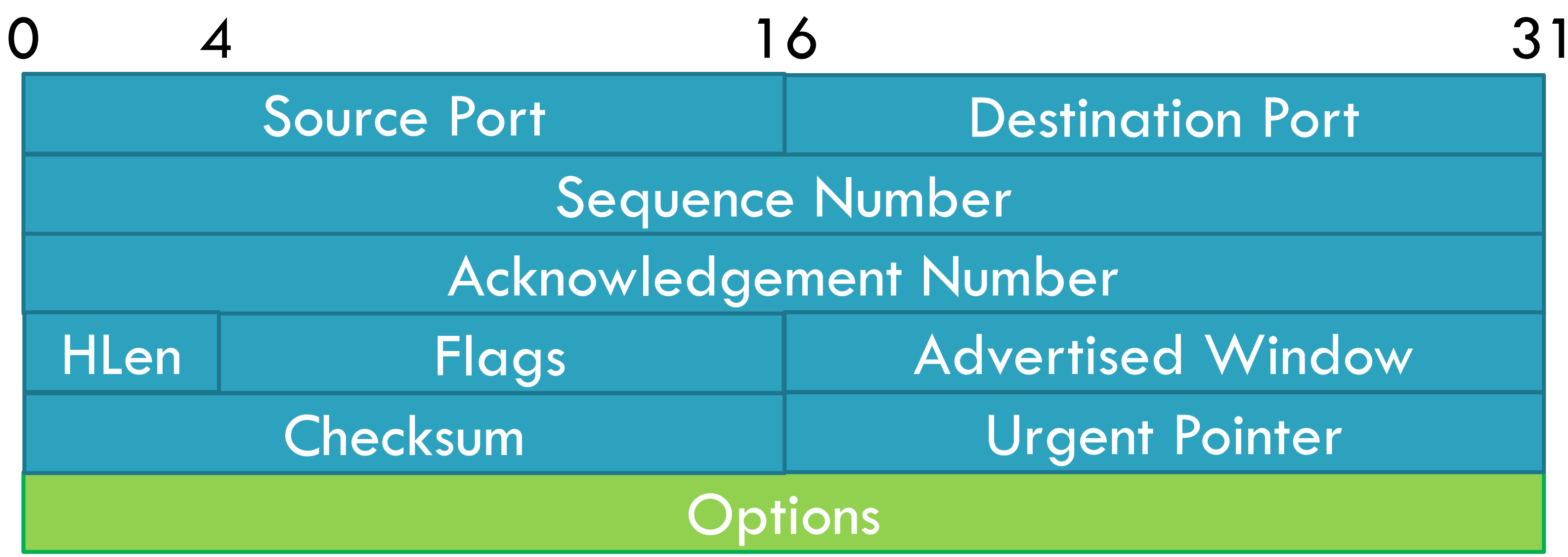
Outline

- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

Transmission Control Protocol

- Reliable, in-order, bi-directional byte streams
  - Port numbers for demultiplexing
  - Virtual circuits (connections)
  - Flow control
  - Congestion control, approximate fairness

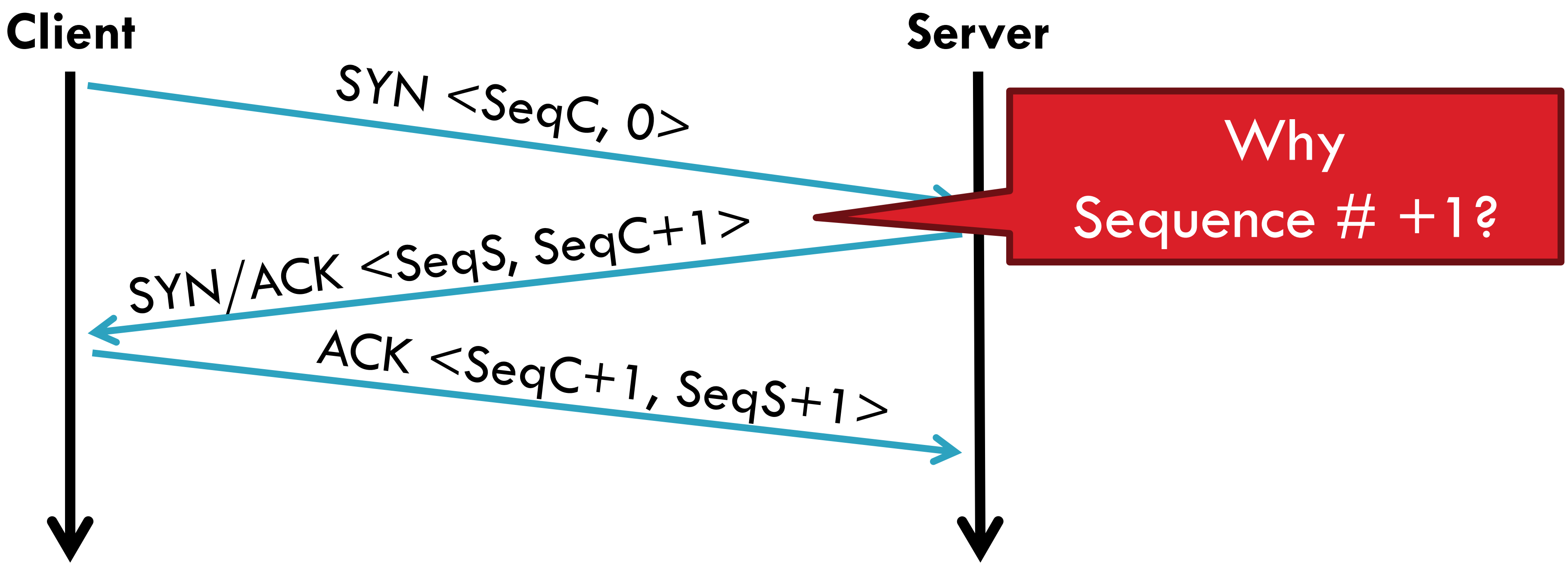
Why these features?



Connection Setup

- Why do we need connection setup?
  - To establish state on both hosts
  - Most important state: sequence numbers
    - Count the number of bytes that have been sent
    - Initial value chosen at random
    - Why?
- Important TCP flags (1 bit each)
  - SYN – synchronization, used for connection setup
  - ACK – acknowledge received data
  - FIN – finish, used to tear down connection

Three Way Handshake



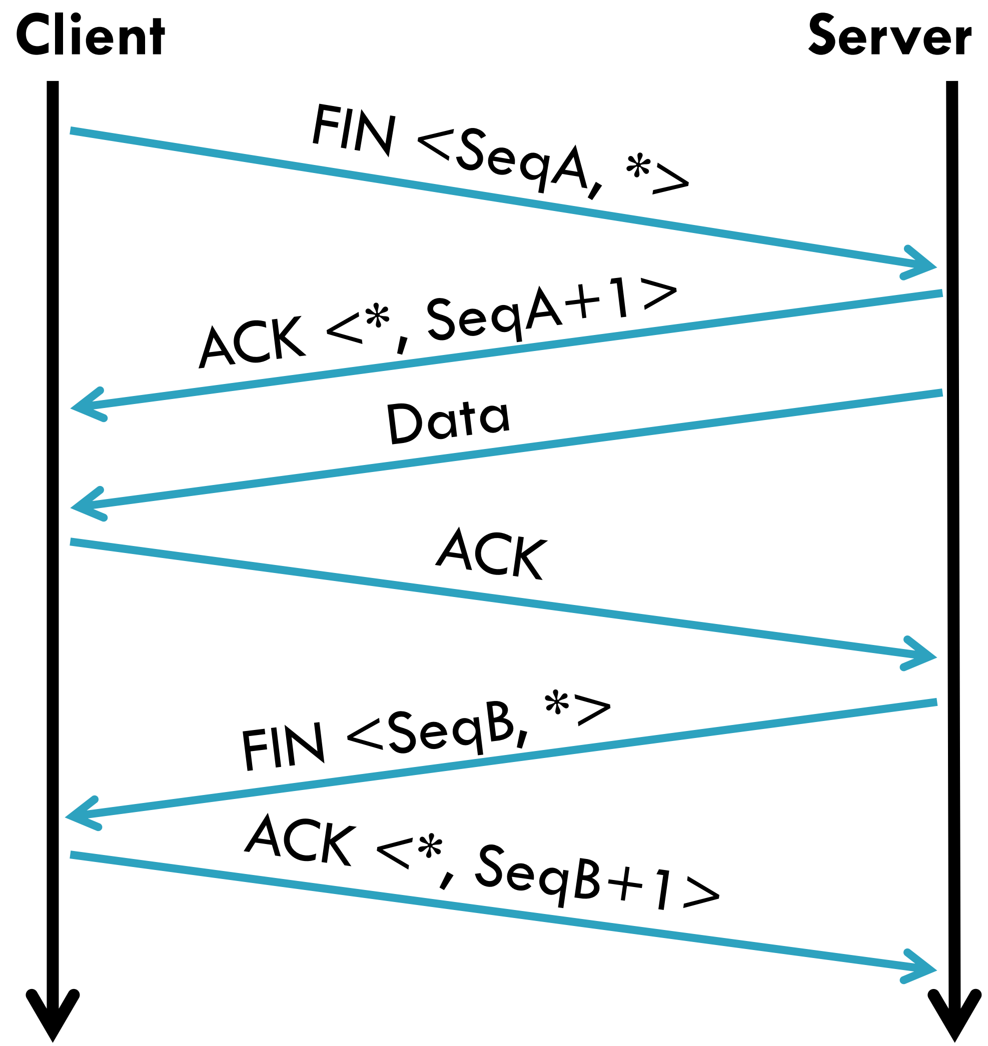
- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side's starting sequence number

Connection Setup Issues

- Connection confusion
  - How to disambiguate connections from the same host?
  - Random sequence numbers
- Source spoofing
  - Kevin Mitnick
  - Need good random number generators!
- Connection state management
  - Each SYN allocates state on the server
  - SYN flood = denial of service attack
  - Solution: SYN cookies

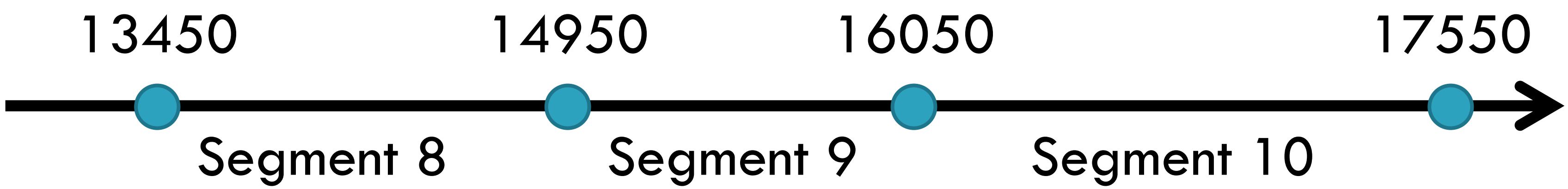
Connection Tear Down

- Either side can initiate tear down
- Other side may continue sending data
  - Half open connection
  - *shutdown()*
- Acknowledge the last FIN
  - Sequence number + 1

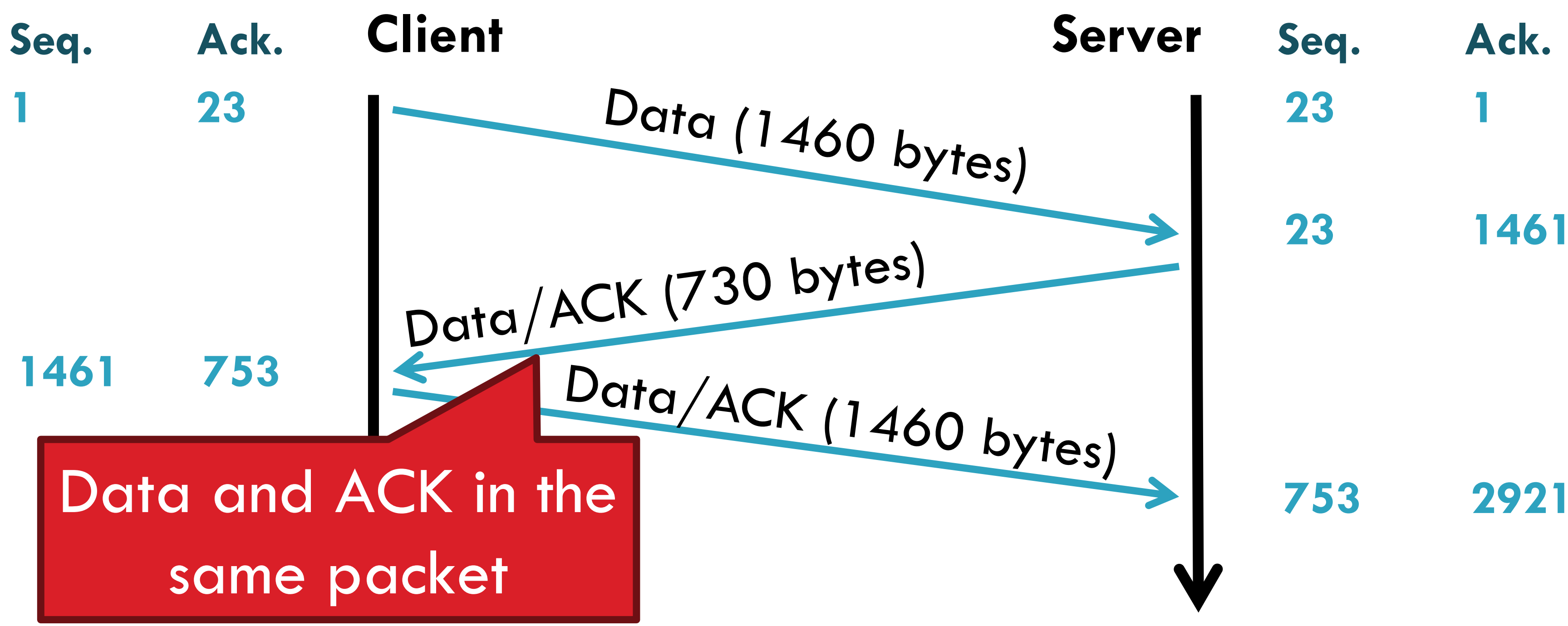


Sequence Number Space

- TCP uses a byte stream abstraction
  - Each byte in each stream is numbered
  - 32-bit value, wraps around
  - Initial, random values selected during setup
- Byte stream broken down into segments (packets)
  - Size limited by the Maximum Segment Size (MSS)
  - Set to limit fragmentation
- Each segment has a sequence number



Bidirectional Communication

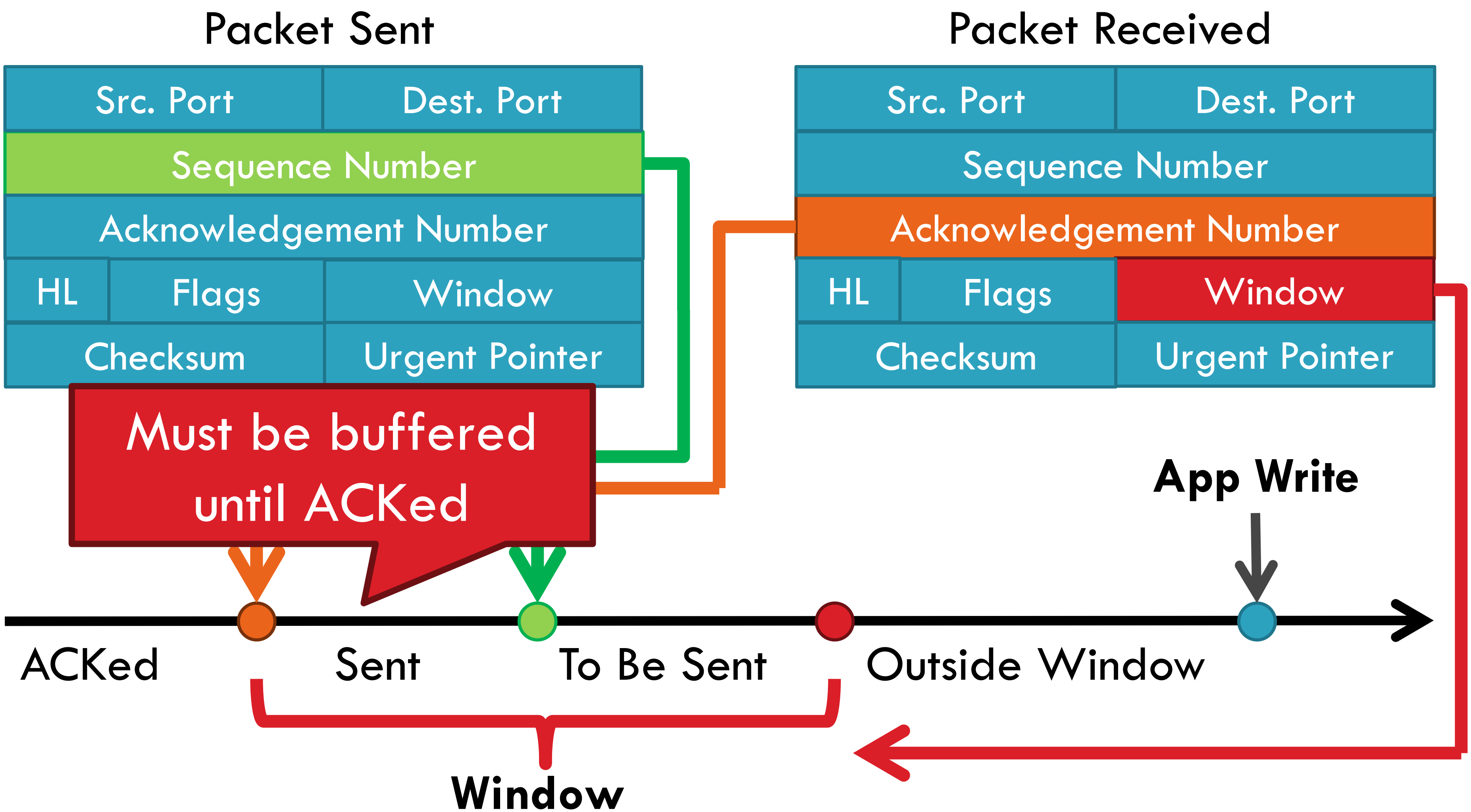


- Each side of the connection can send and receive
  - Different sequence numbers for each direction

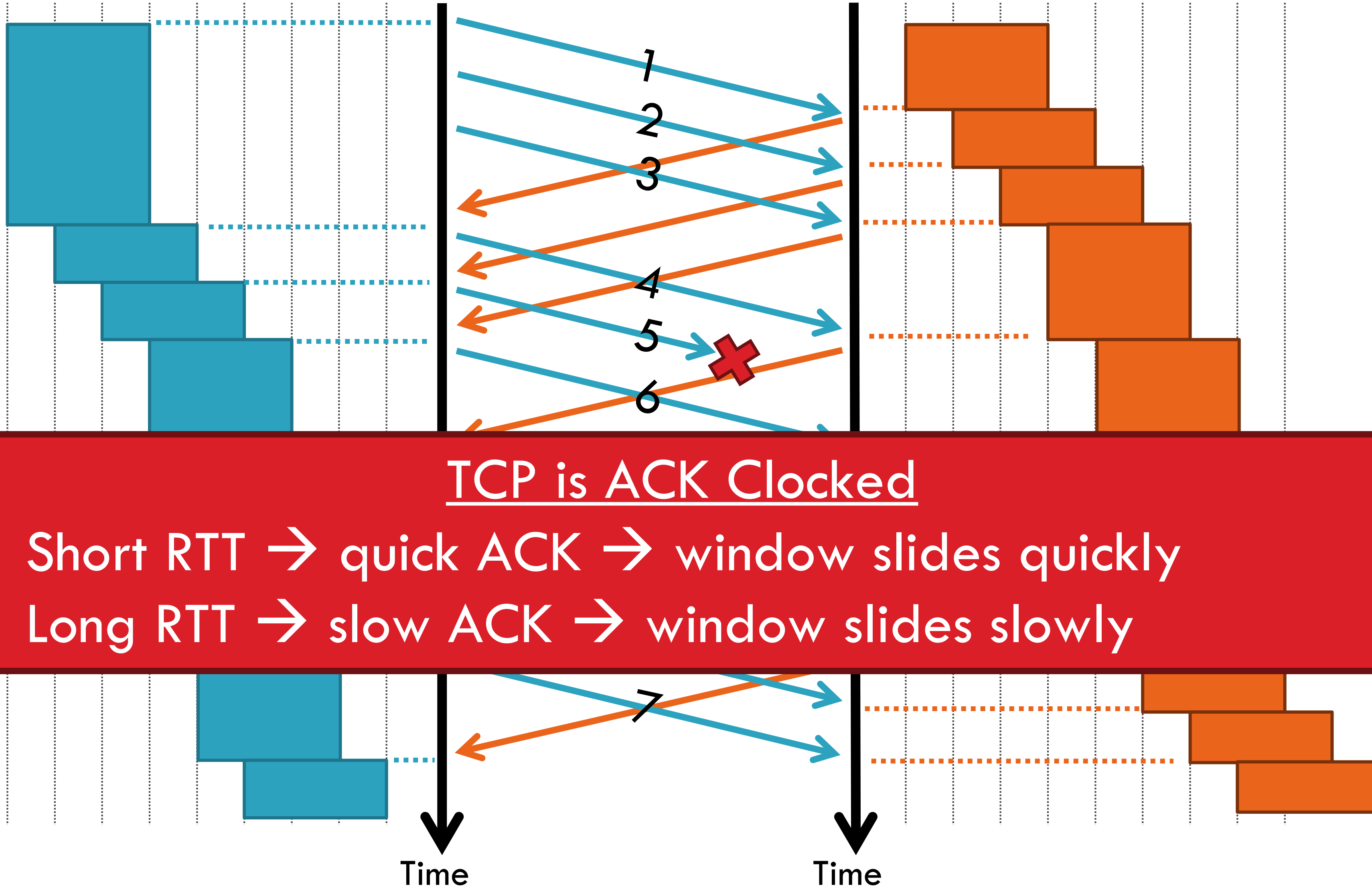
Flow Control

- Problem: how many packets should a sender transmit?
  - Too many packets may overwhelm the receiver
  - Size of the receivers buffers may change over time
- Solution: sliding window
  - Receiver tells the sender how big their buffer is
  - Called the advertised window
  - For window size *n*, sender may transmit *n* bytes without receiving an ACK
  - After each ACK, the window slides forward
- Window may go to zero!

Flow Control: Sender Side



Sliding Window Example



TCP is ACK Clocked

- Short RTT → quick ACK → window slides quickly
- Long RTT → slow ACK → window slides slowly

What Should the Receiver ACK?

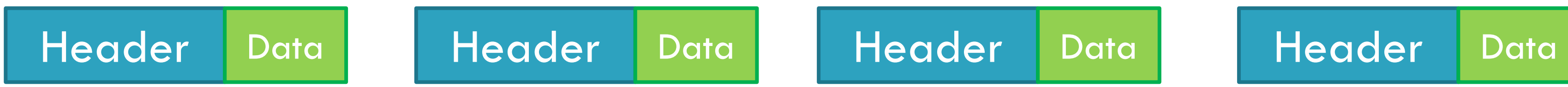
1. ACK every packet
2. Use *cumulative ACK*, where an ACK for sequence *n* implies ACKS for all $k < n$
3. Use *negative ACKs* (NACKs), indicating which packet did not arrive
4. Use *selective ACKs* (SACKs), indicating those that did arrive, even if not in order
   - SACK is an actual TCP extension

Sequence Numbers, Revisited

- 32 bits, unsigned
  - Why so big?
- For the sliding window you need…
  - |Sequence # Space| > 2 * |Sending Window Size|
  - $2^{32} > 2 * 2^{16}$
- Guard against stray packets
  - IP packets have a maximum segment lifetime (MSL) of 120 seconds
    - i.e. a packet can linger in the network for 3 minutes
  - Sequence number would wrap around at 286Mbps
    - What about GigE? PAWS algorithm + TCP options

Silly Window Syndrome

- Problem: what if the window size is very small?
  - Multiple, small packets, headers dominate data



- Equivalent problem: sender transmits packets one byte at a time

```
for (int x = 0; x < strlen(data); ++x)
  write(socket, data + x, 1);
```

Nagle's Algorithm

1. If the window >= MSS and available data >= MSS:
   Send the data
2. Elif there is unACKed data:
   Enqueue data in a buffer (send after a timeout)
3. Else: send the data

Send a full packet

Send a non-full packet if nothing else is happening

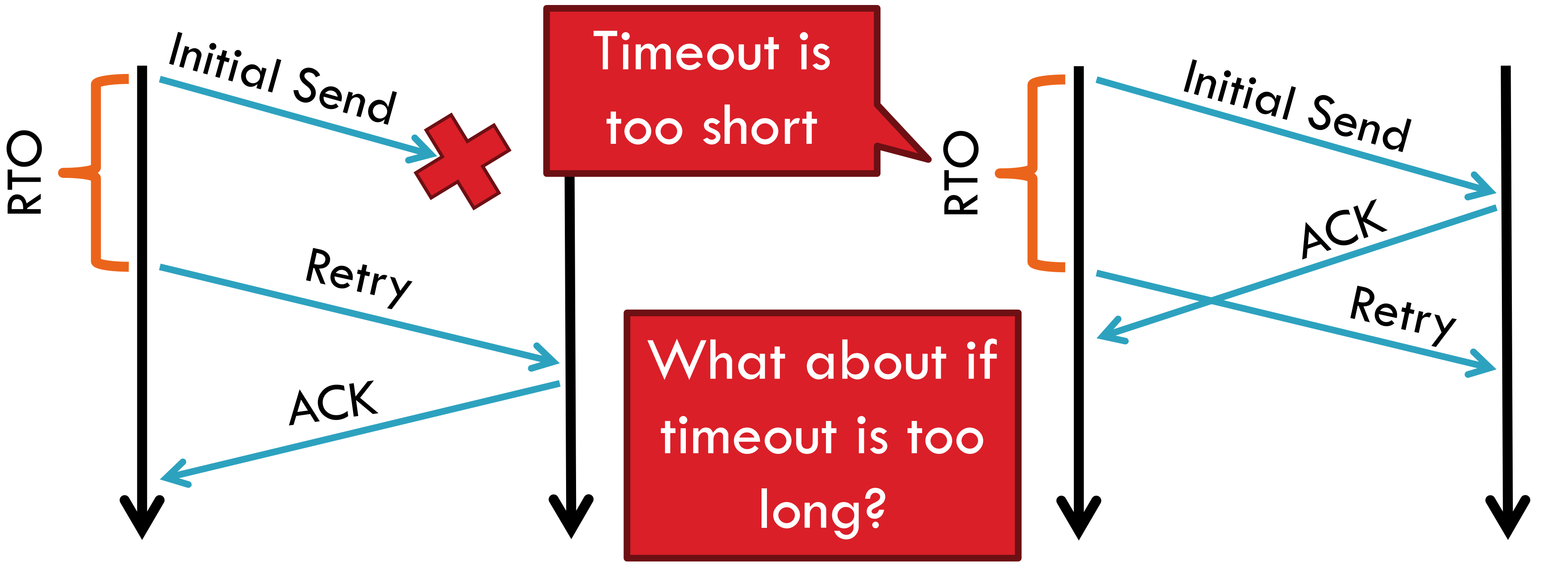
- Problem: Nagle's Algorithm delays transmissions
  - What if you need to send a packet immediately?
  1. int flag = 1;
  2. setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *) &flag, sizeof(int));

Error Detection

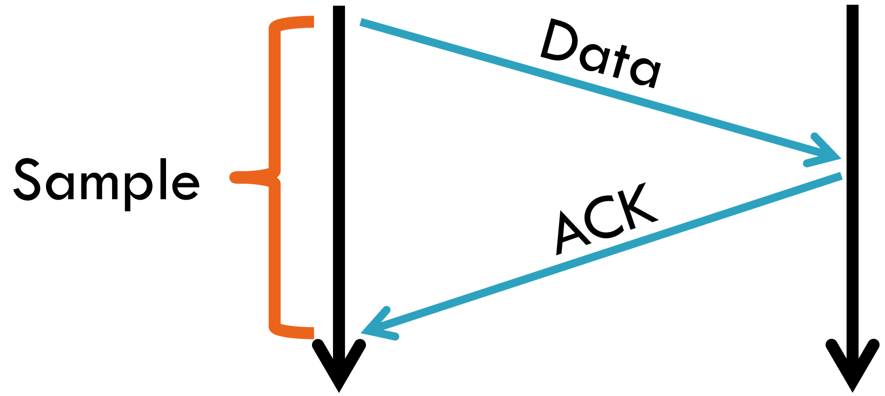
- Checksum detects (some) packet corruption
  - Computed over IP header, TCP header, and data
- Sequence numbers catch sequence problems
  - Duplicates are ignored
  - Out-of-order packets are reordered or dropped
  - Missing sequence numbers indicate lost packets
- Lost segments detected by sender
  - Use timeout to detect missing ACKs
  - Need to estimate RTT to calibrate the timeout
  - Sender must keep copies of all data until ACK

Retransmission Time Outs (RTO)

- Problem: time-out is linked to round trip time

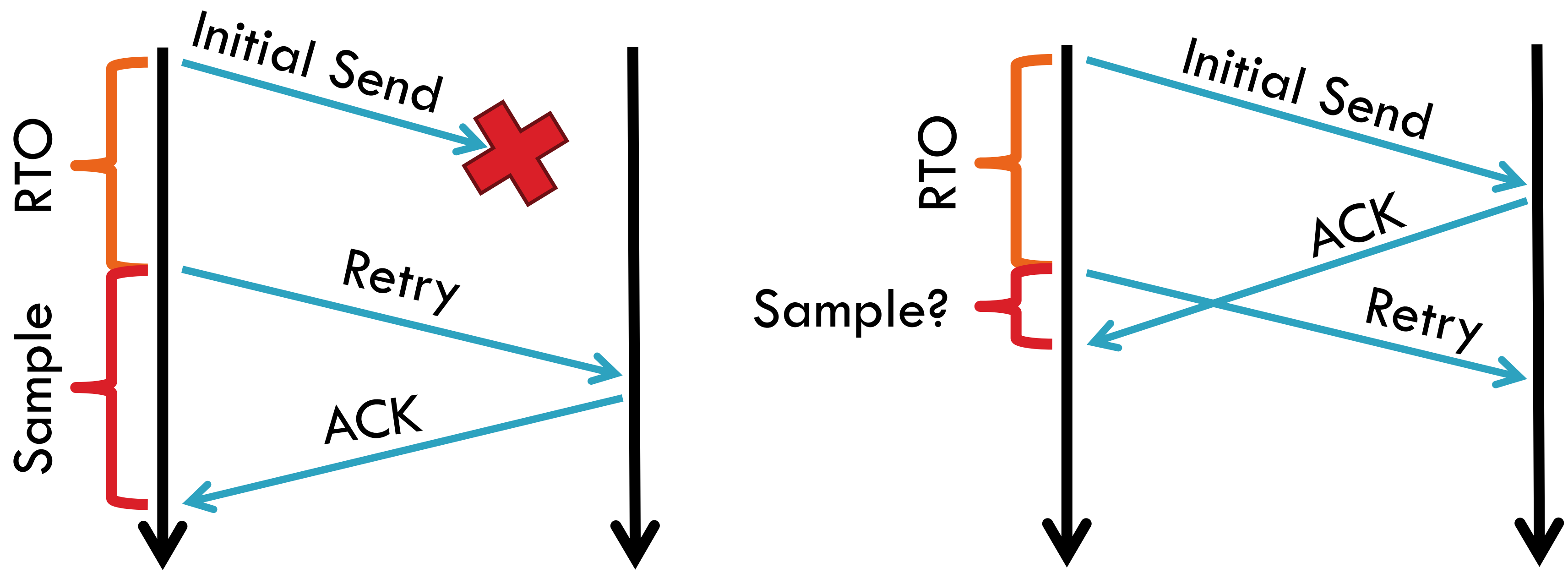


Round Trip Time Estimation



- Original TCP round-trip estimator
  - RTT estimated as a moving average
  - new_rtt = α (old_rtt) + (1 – α)(new_sample)
  - Recommended α: 0.8-0.9 (0.875 for most TCPs)
- RTO = 2 * new_rtt (i.e. TCP is conservative)

RTT Sample Ambiguity



- Karn's algorithm: ignore samples for retransmitted segments

Outline

- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

What is Congestion?

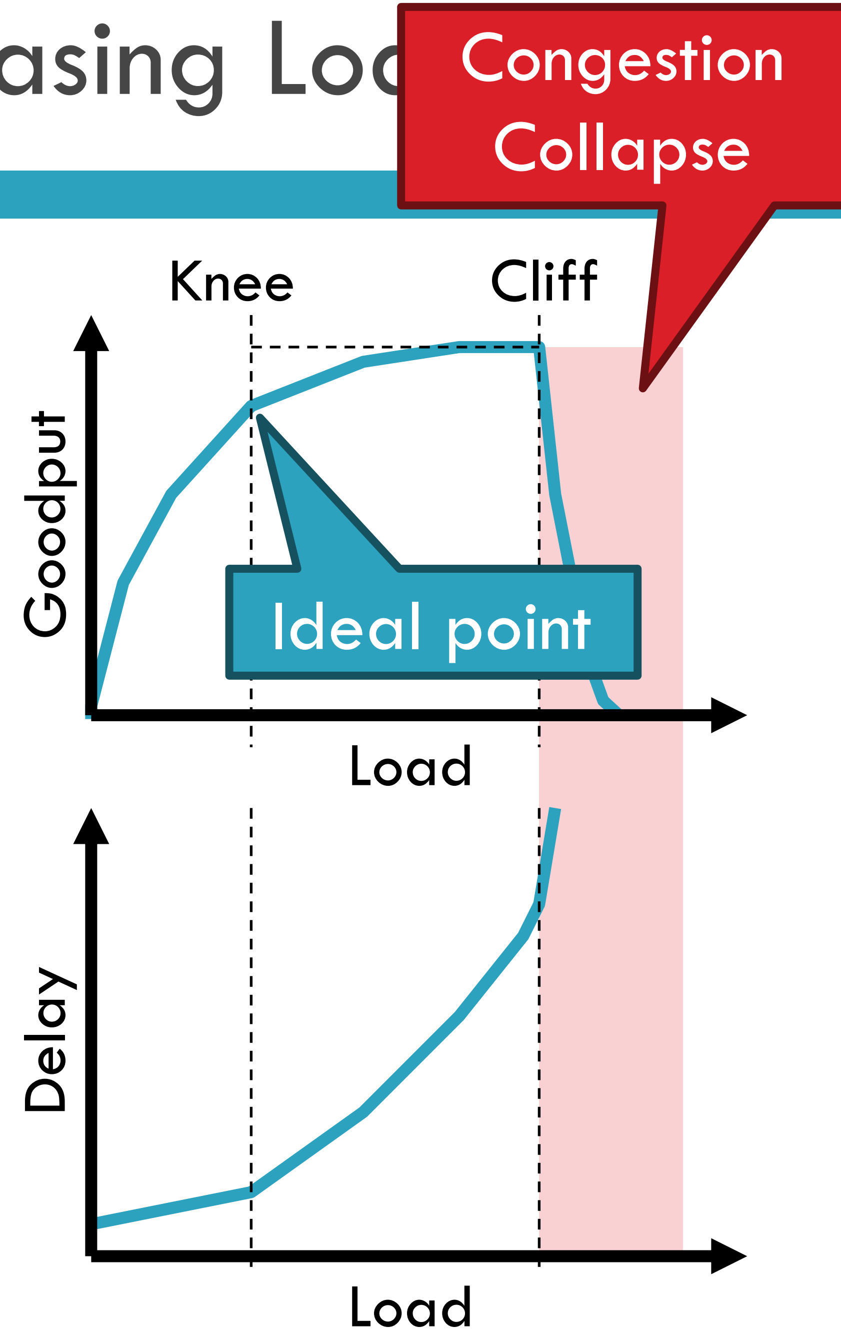
- Load on the network is higher than capacity
  - Capacity is not uniform across networks
    - Modem vs. Cellular vs. Cable vs. Fiber Optics
  - There are multiple flows competing for bandwidth
    - Residential cable modem vs. corporate datacenter
  - Load is not uniform over time
    - 10pm, Sunday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

Why is Congestion Bad?

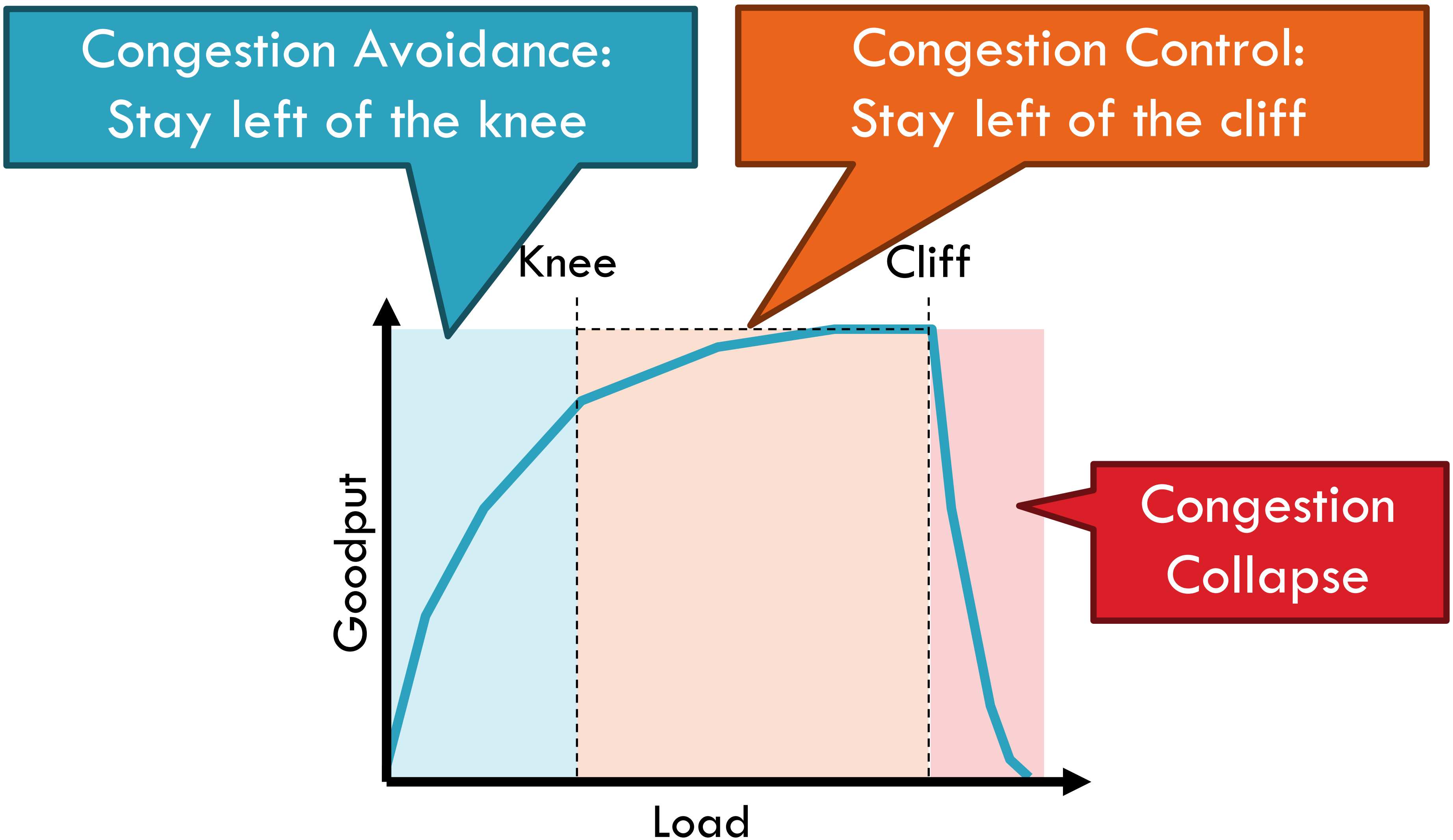
- Results in packet loss
  - Routers have finite buffers, packets must be dropped
- Practical consequences
  - Router queues build up, delay increases
  - Wasted bandwidth from retransmissions
  - Low network goodput

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 → ∞



Cong. Control vs. Cong. Avoidance



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

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
- 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 Summary

- Important TCP Congestion Control ideas include: **AIMD, Slow Start, Fast Retransmit and Fast Recovery** .
- Know the differences between **TCP Tahoe, TCP Reno and TCP New Reno** .
- Currently, the two most common versions of TCP are Compound (Windows) and Cubic (Linux).
- TCP needs rules and an algorithm to determine **RTO and RTO**.

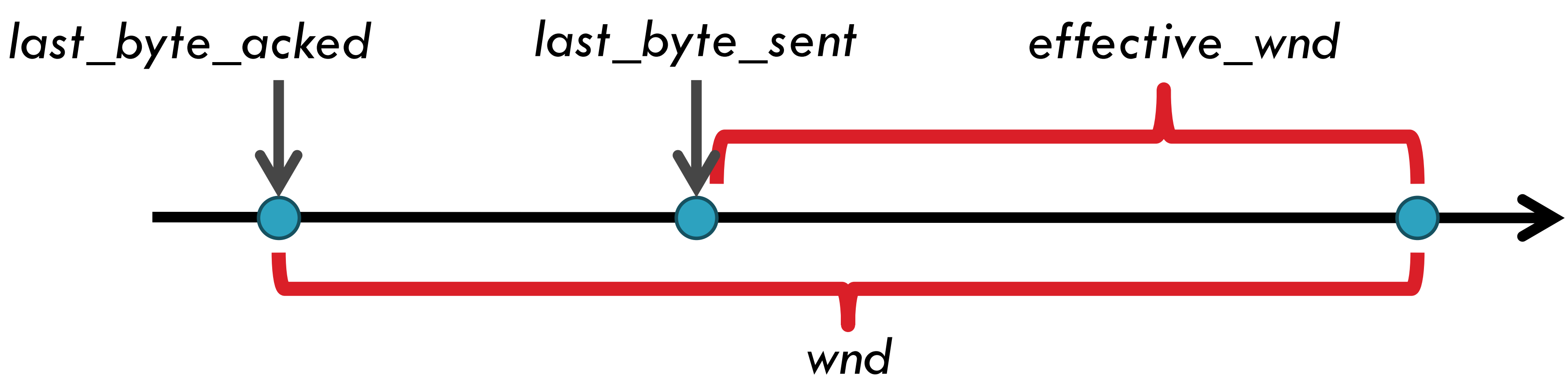
TCP Congestion Control

- 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*);



Two Basic Components

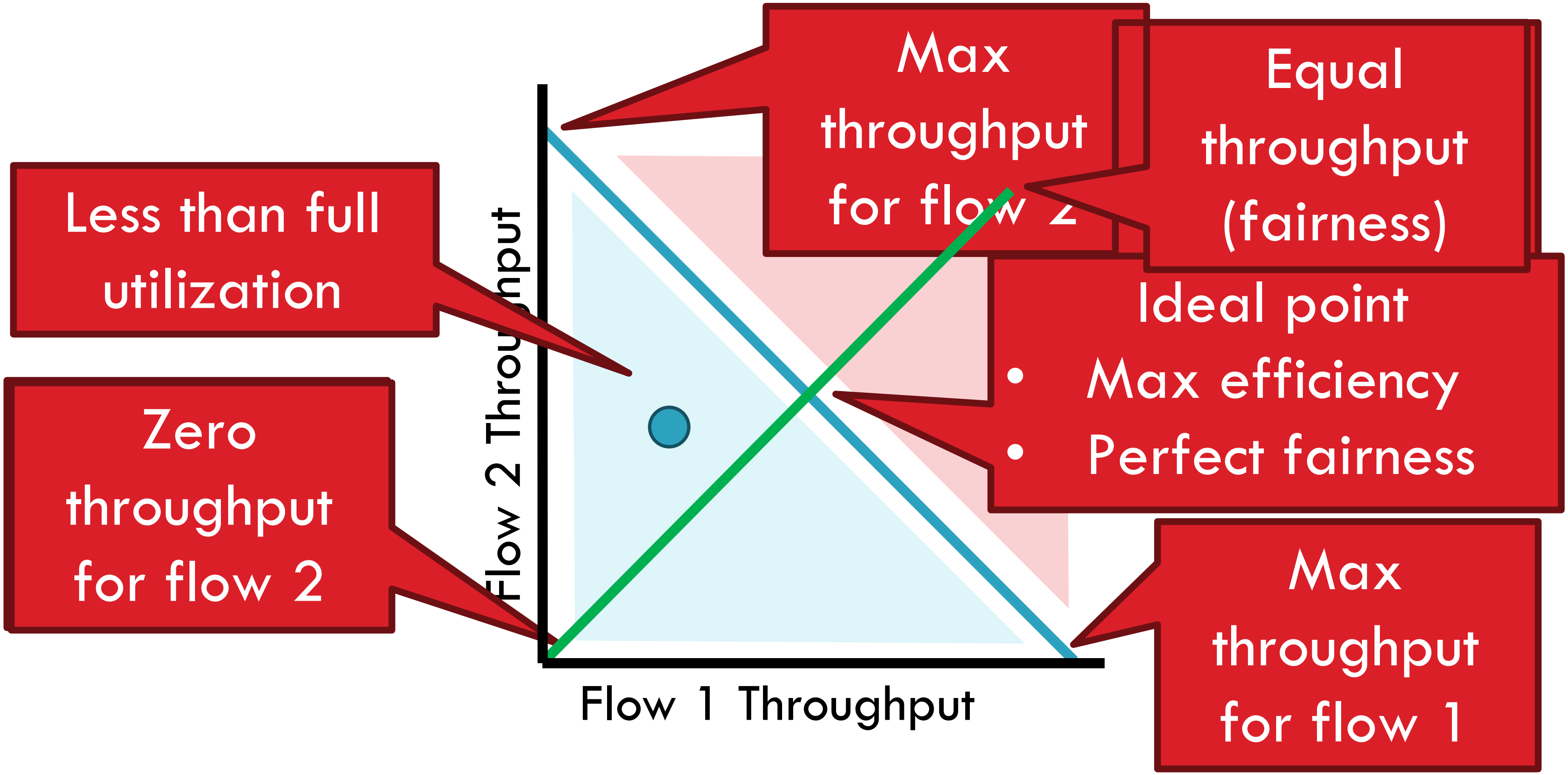
1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)
2. Rate adjustment algorithm
   - Modify *cwnd*
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

Except on wireless networks

Rate Adjustment

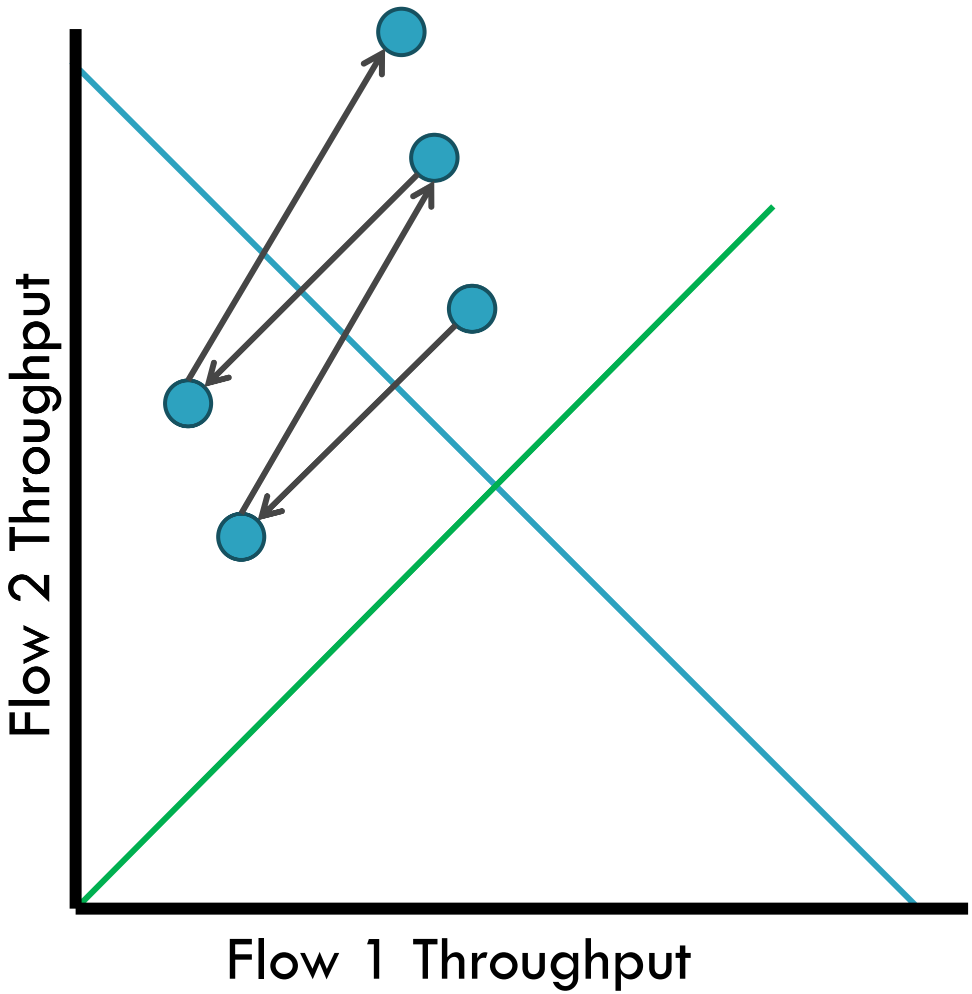
- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly
- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - *cwnd* growth is proportional to RTT
  - On loss: decrease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

Utilization and Fairness



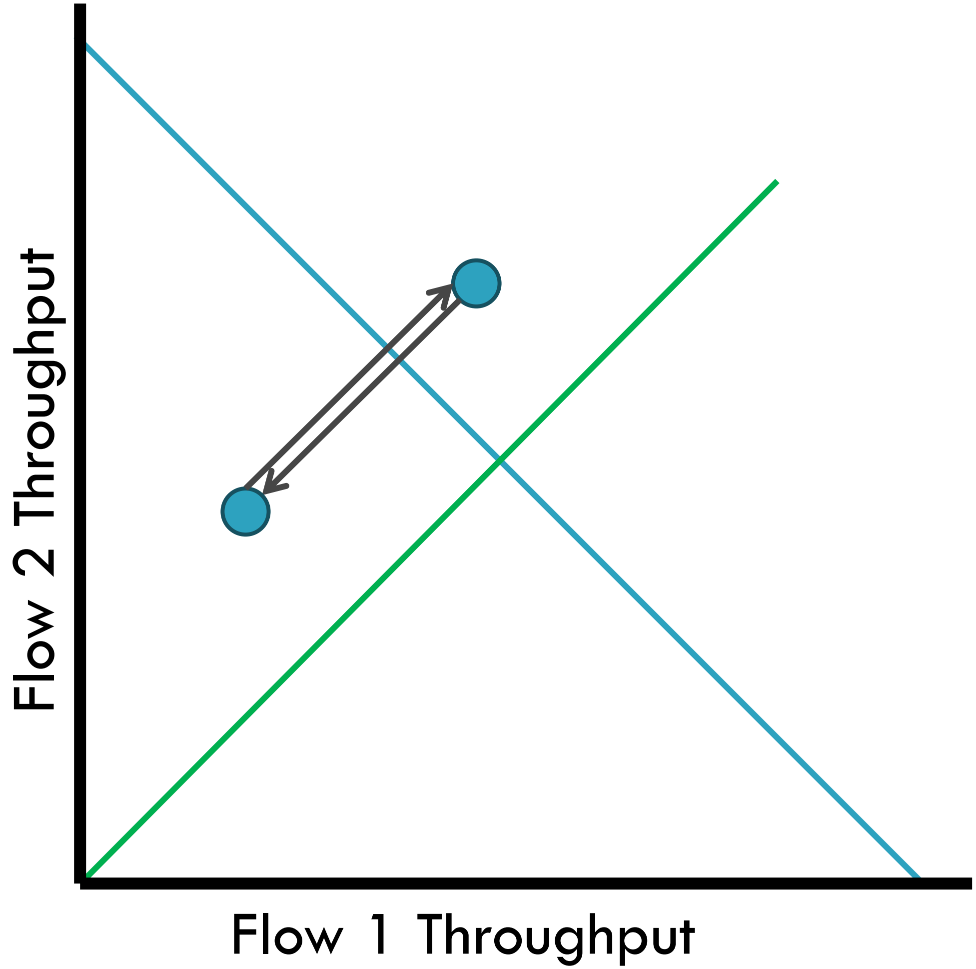
Multiplicative Increase, Additive Decrease

- Not stable!
- Veers away from fairness



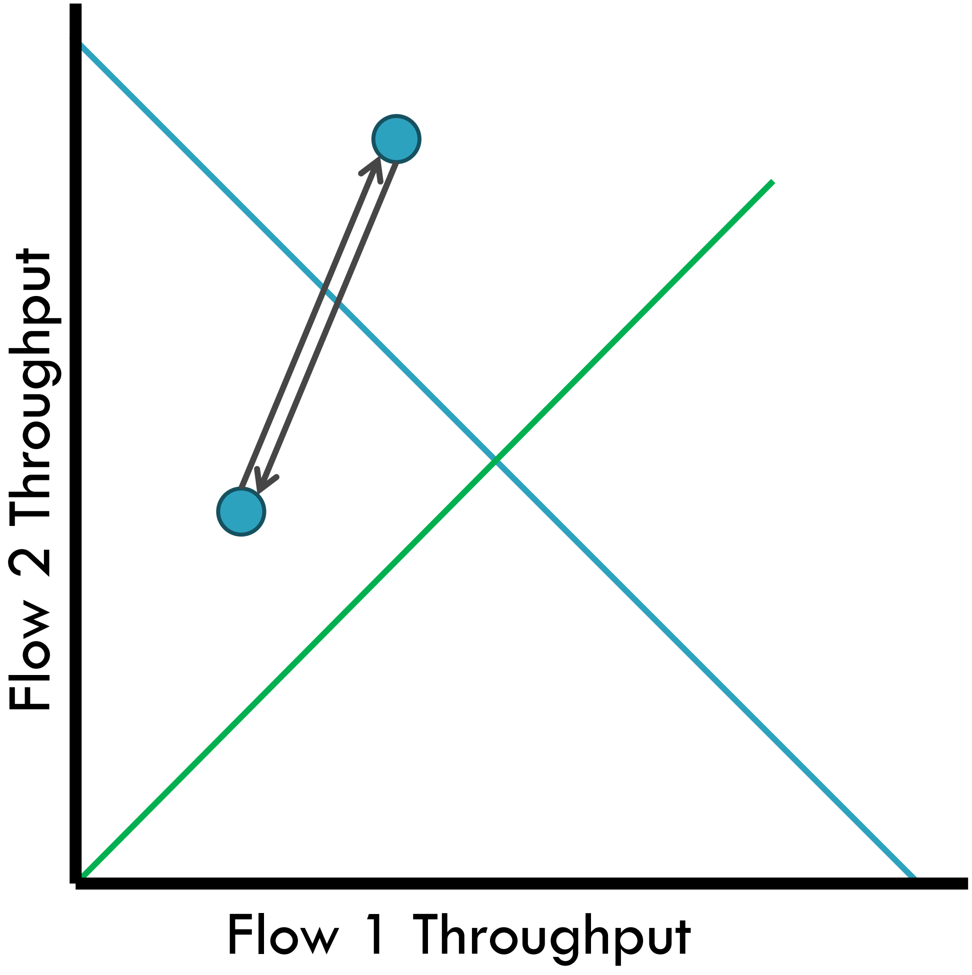
Additive Increase, Additive Decrease

- Stable
- But does not converge to fairness



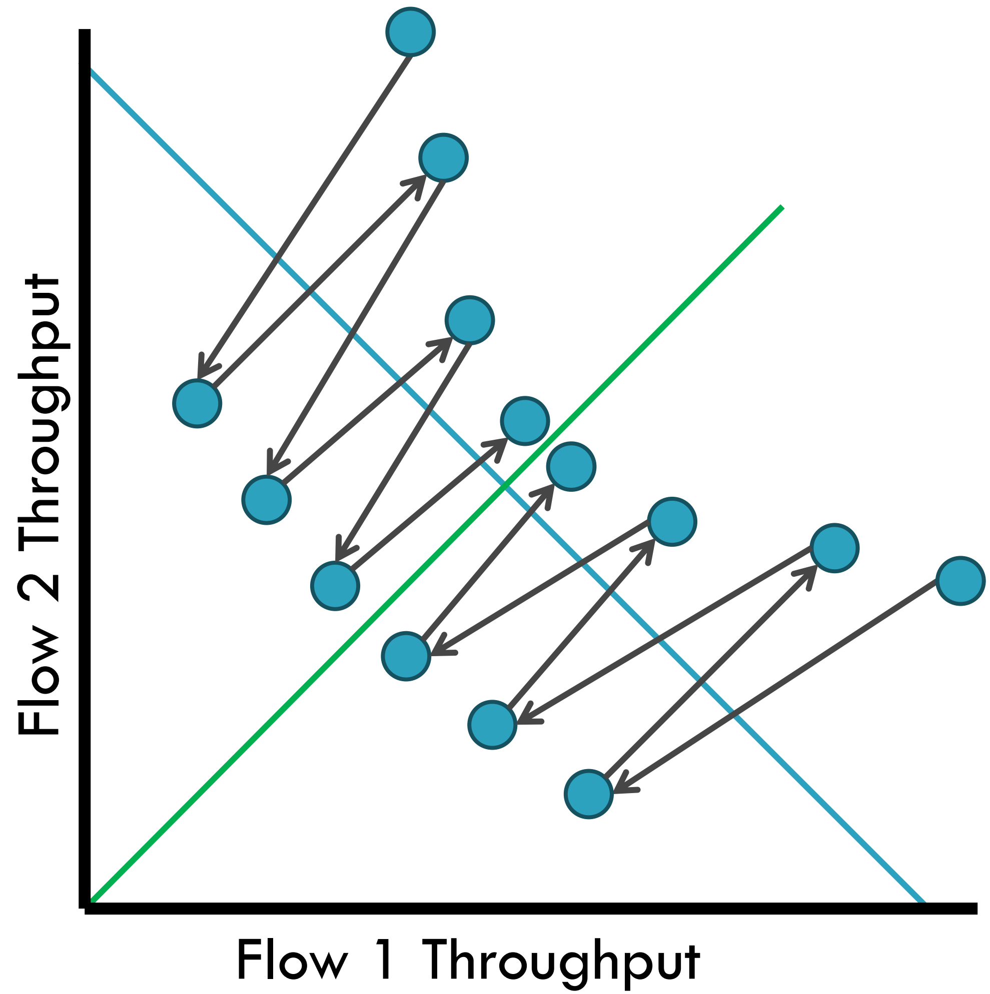
Multiplicative Increase, Multiplicative Decrease

- Stable
- But does not converge to fairness



Additive Increase, Multiplicative Decrease

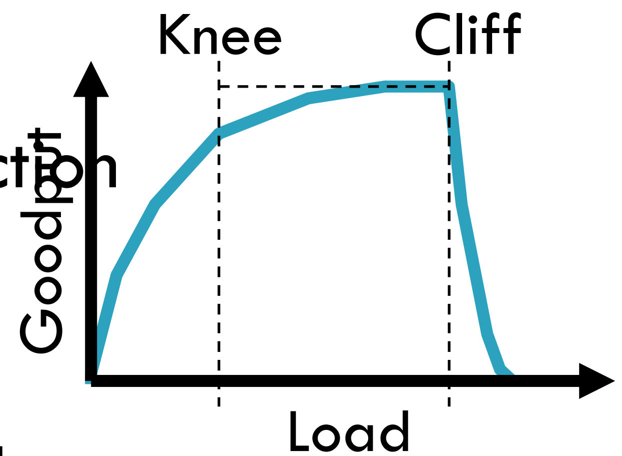
- Converges to stable and fair cycle
- Symmetric around $y=x$



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

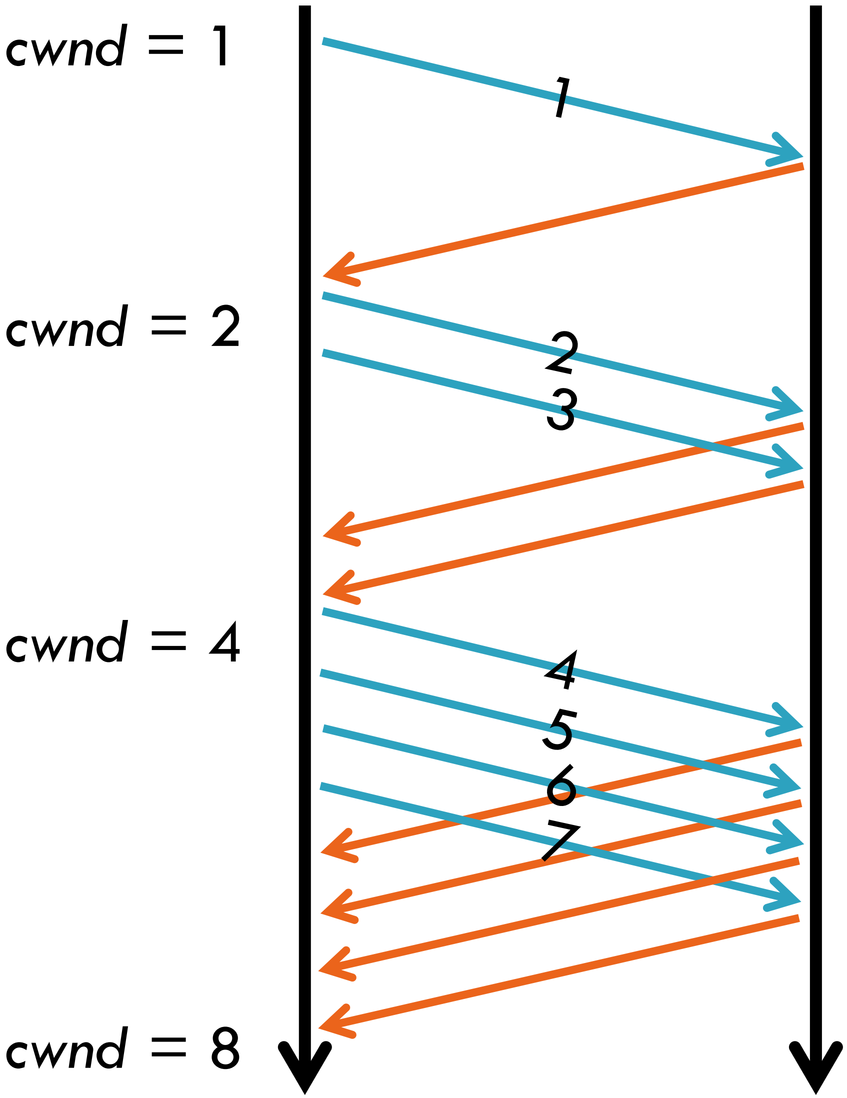
Slow Start



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

Slow Start Example

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



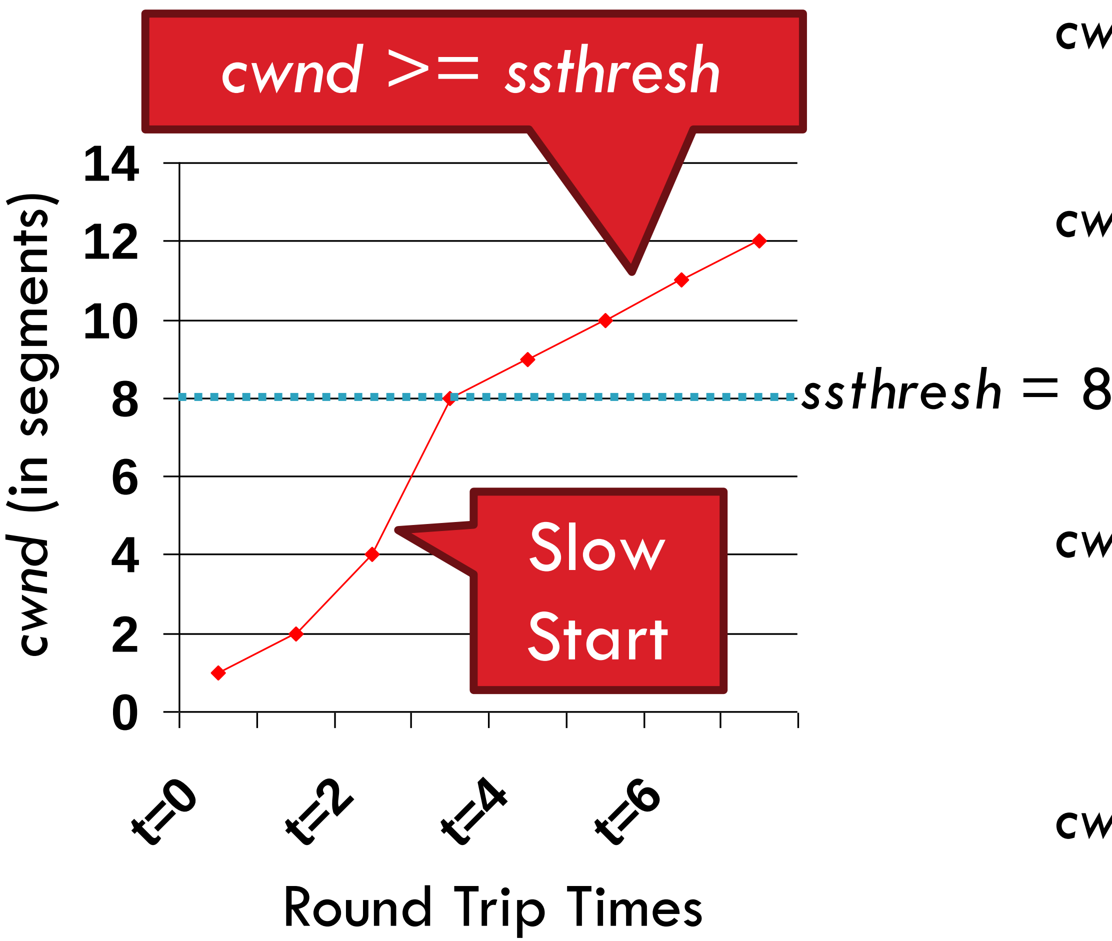
Congestion Avoidance

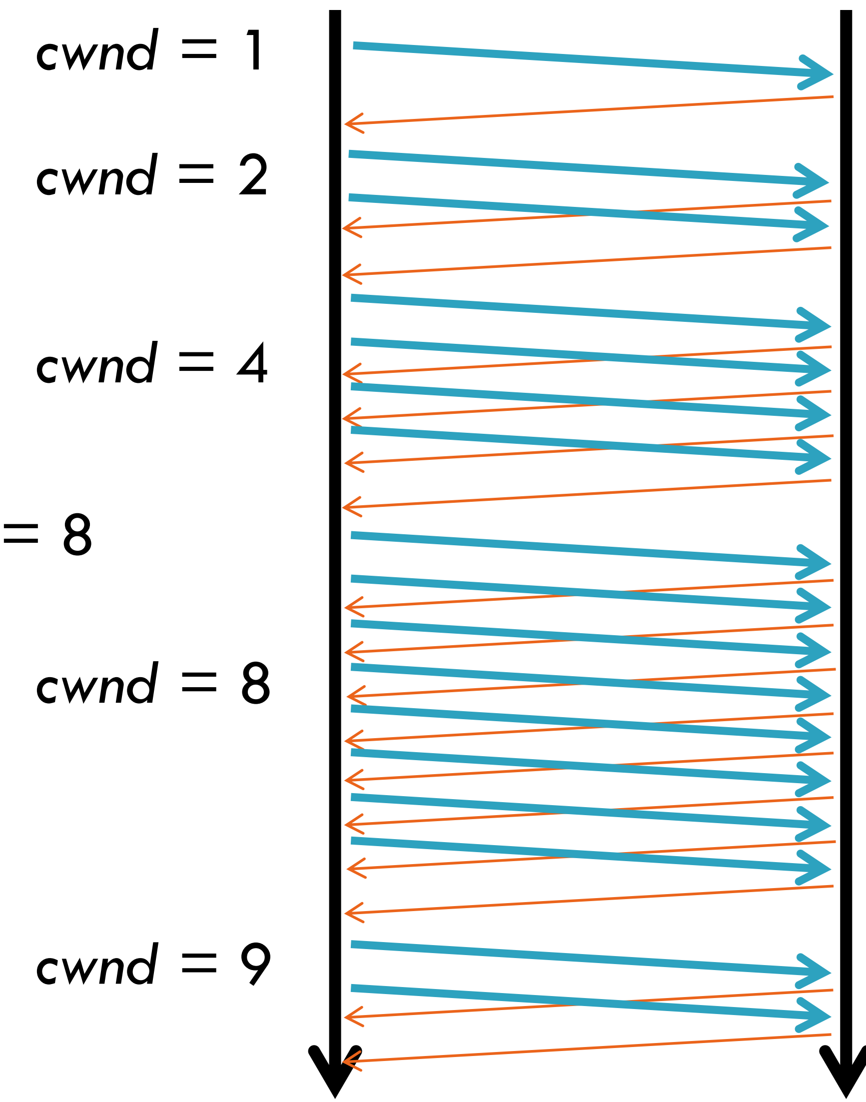
- AIMD mode
- *ssthresh* is lower-bound guess about location of the knee
- **If** *cwnd* >= *ssthresh* **then**

  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

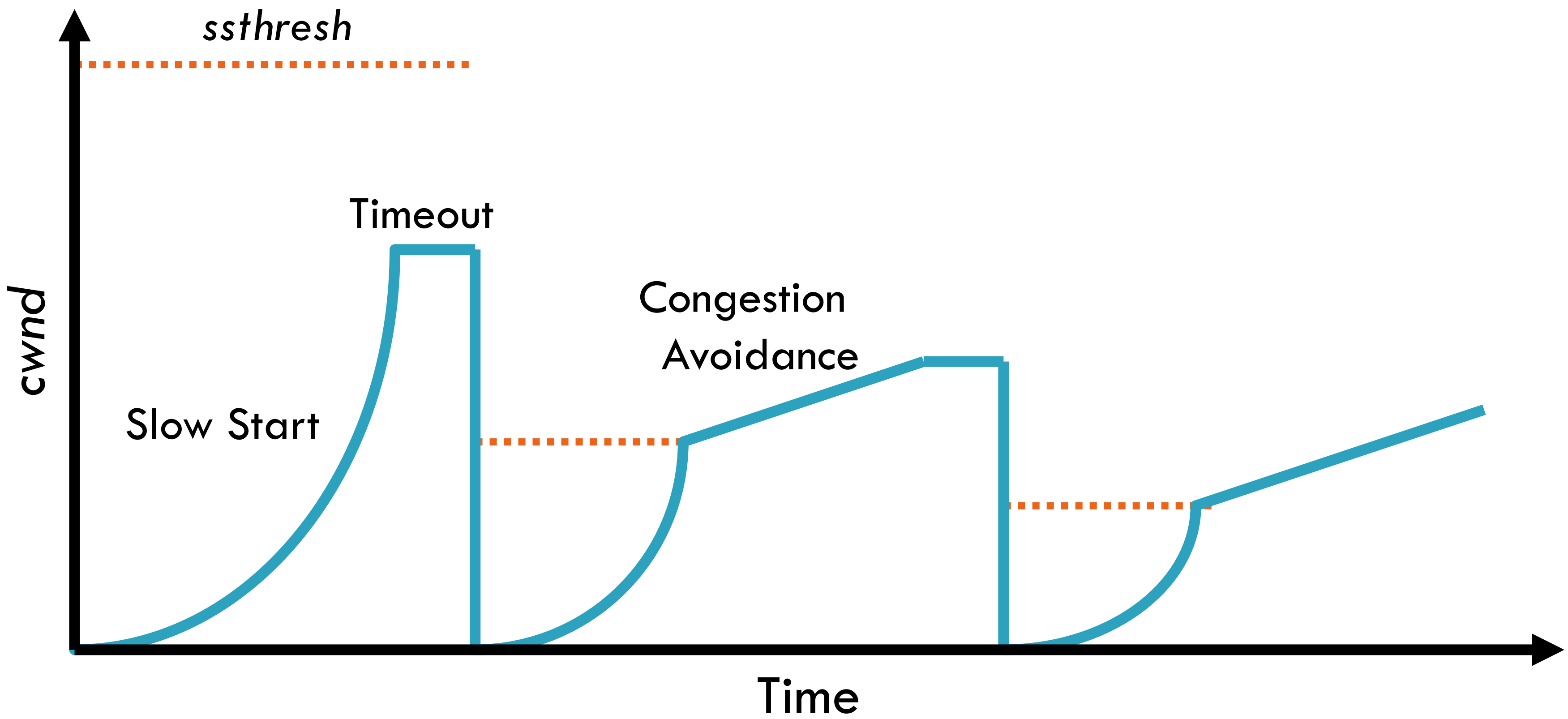




TCP Pseudocode


```
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



Outline

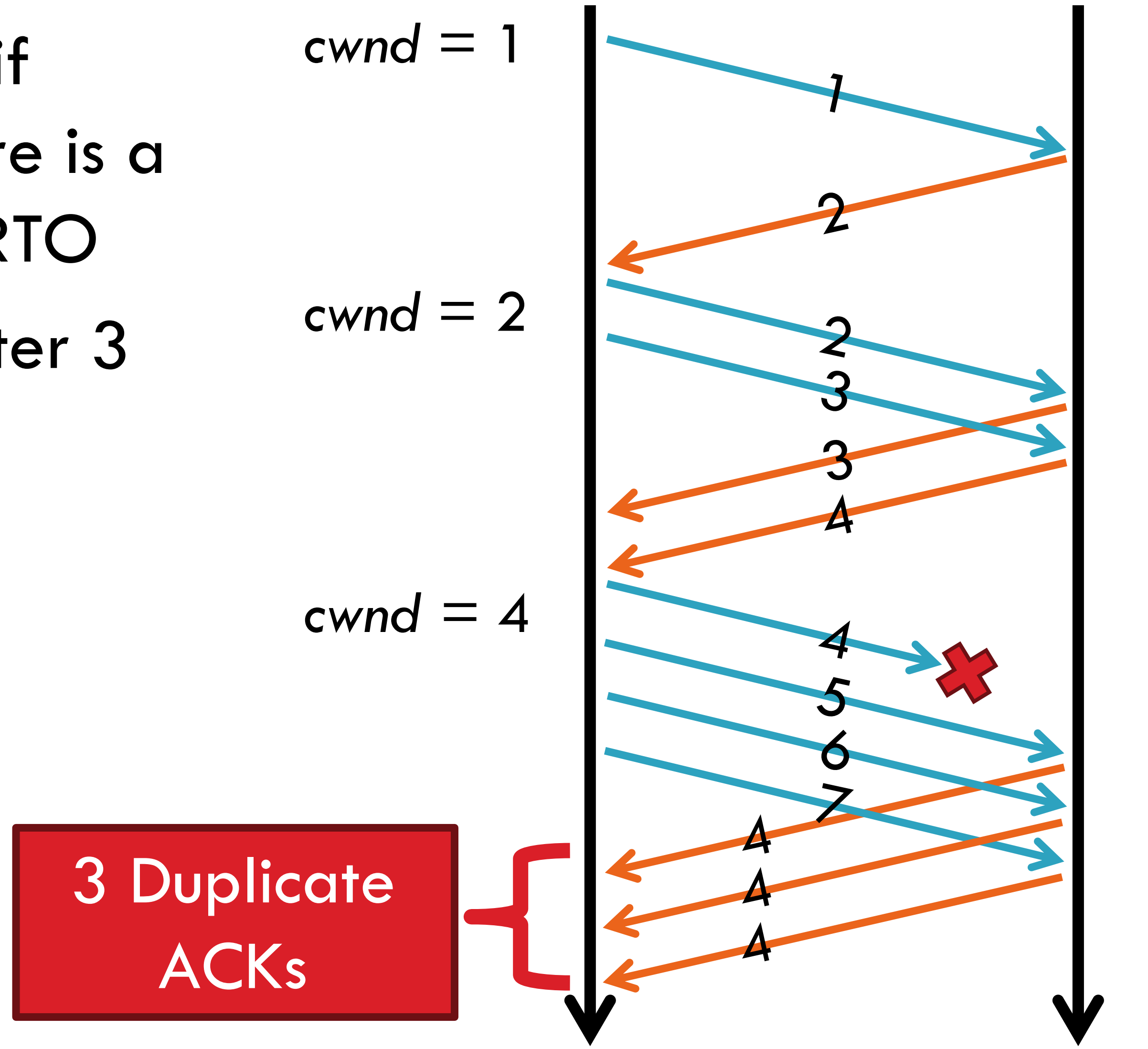
- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

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
- Early, popular variant: TCP Reno
  - Tahoe features, plus…
  - Fast retransmit
  - Fast recovery

TCP Reno: Fast Retransmit

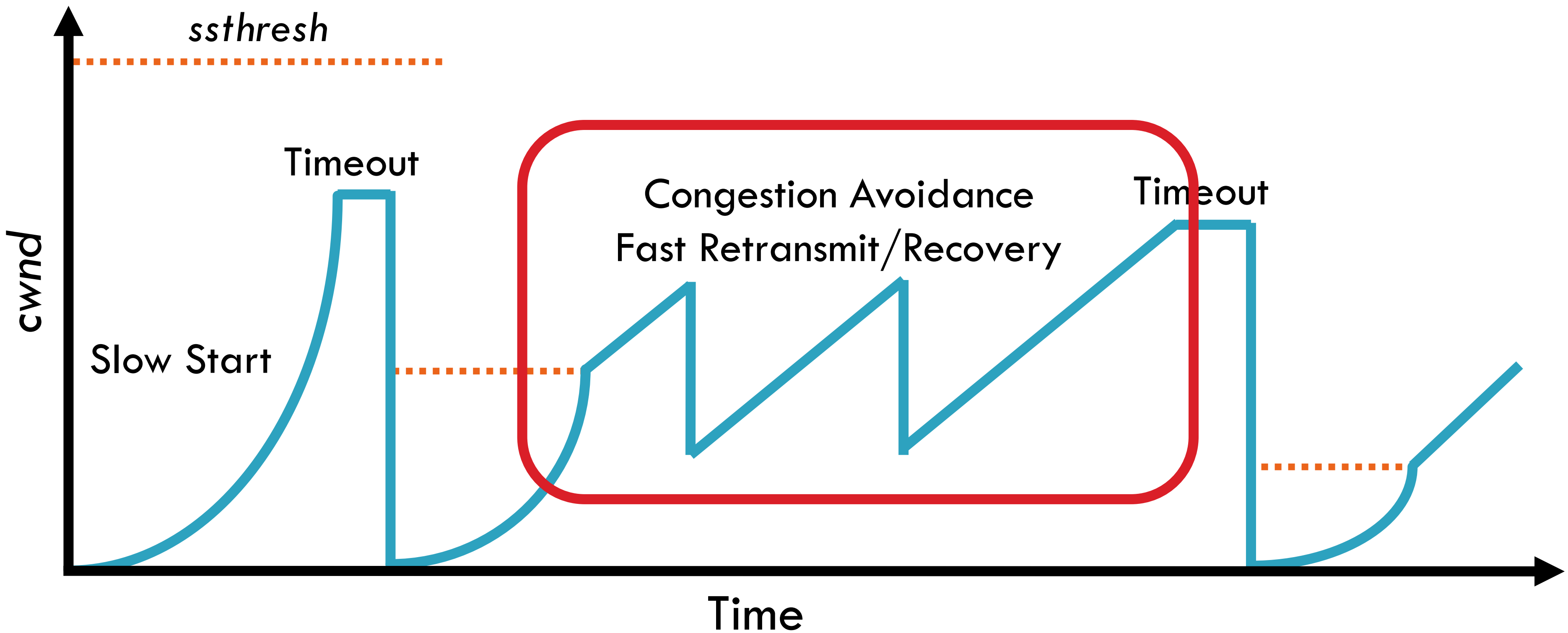
- Problem: in Tahoe, if segment is lost, there is a long wait until the RTO
- Reno: retransmit after 3 duplicate ACKs



TCP Reno: Fast 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



- 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
- 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…

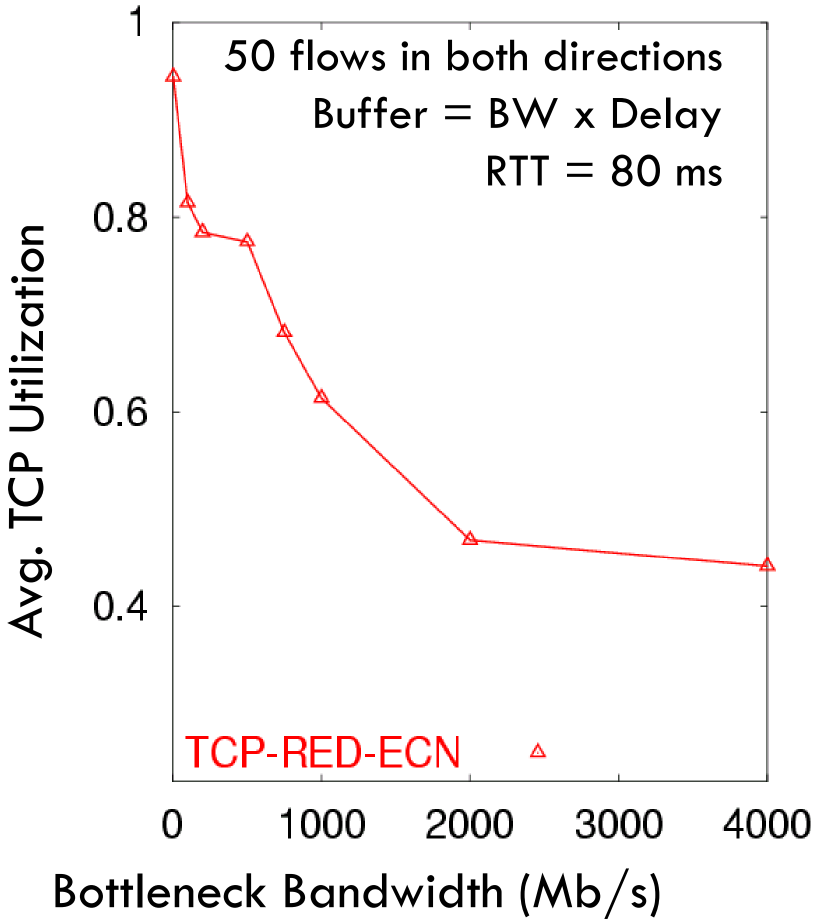
TCP in the Real World

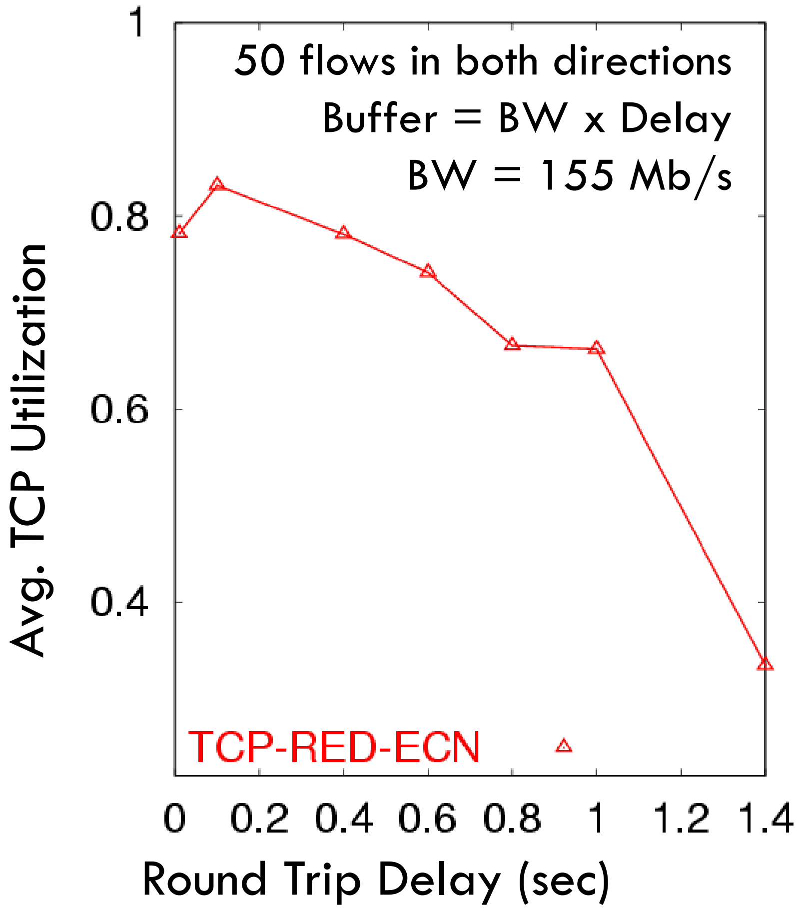
- What are the most popular variants today?
  - Key problem: TCP performs poorly on high bandwidth-delay product networks (like the modern Internet)
  - Compound TCP (Windows)
    - Based on Reno
    - Uses two congestion windows: delay based and loss based
    - Thus, it uses a *compound* congestion 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
- 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





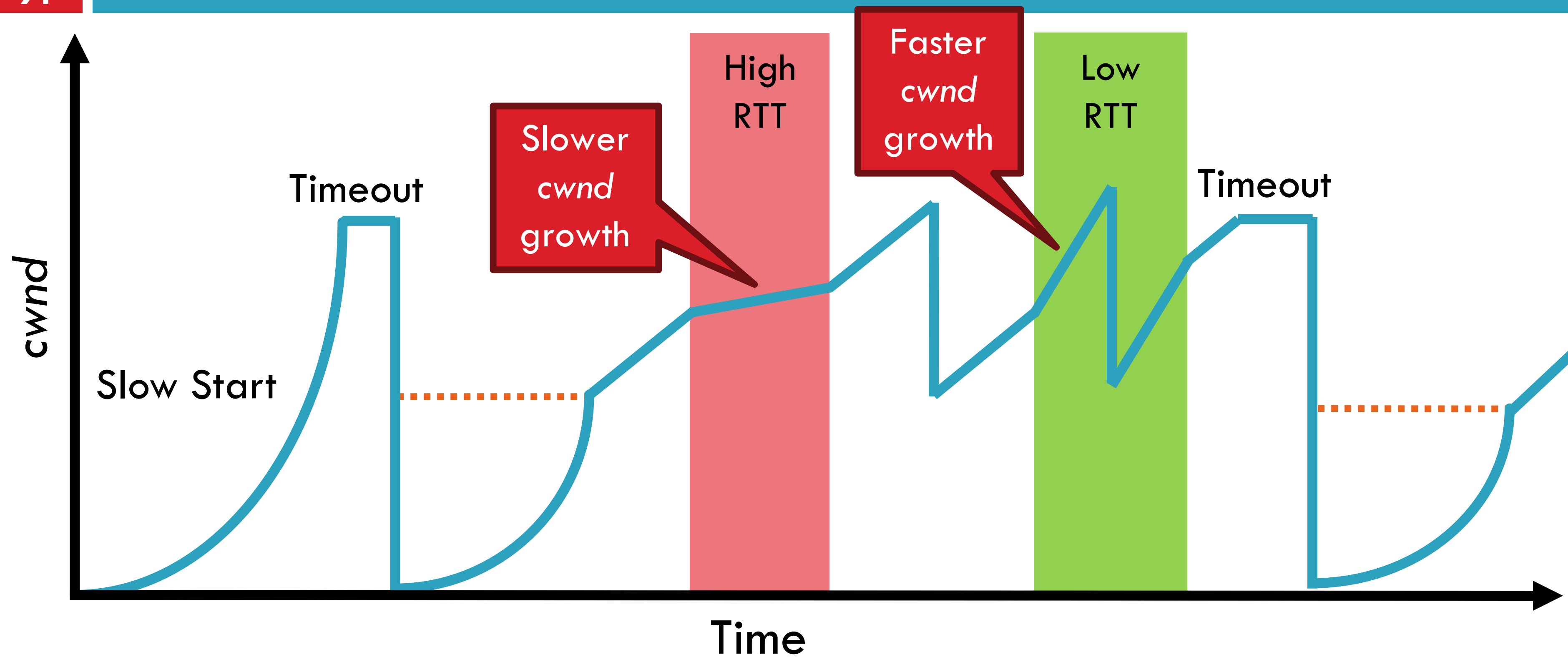
Goals

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



- 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

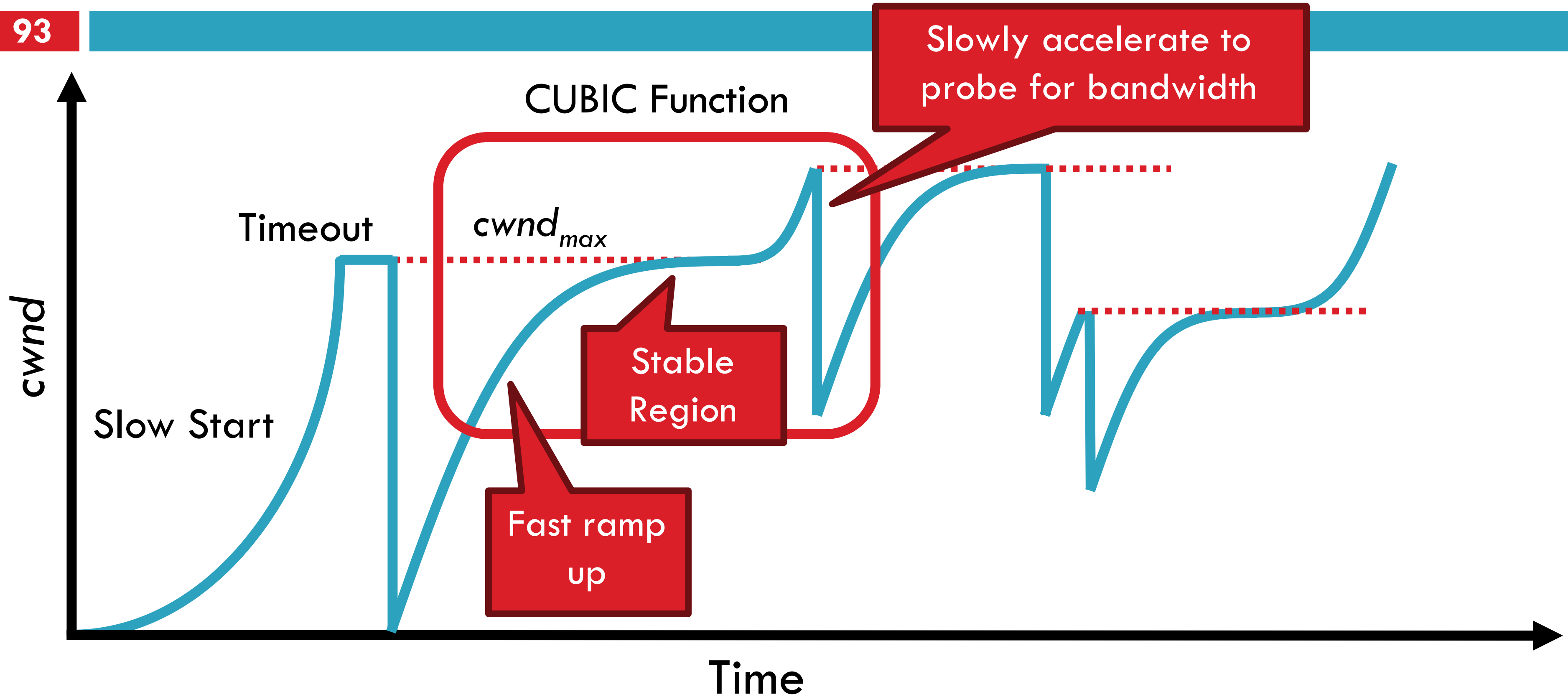
TCP CUBIC Implementation

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

$$cwnd = C * \left(T - \sqrt[3]{\frac{cwnd_{max} * \ \beta}{C}}\right)^3 + cwnd_{max}$$

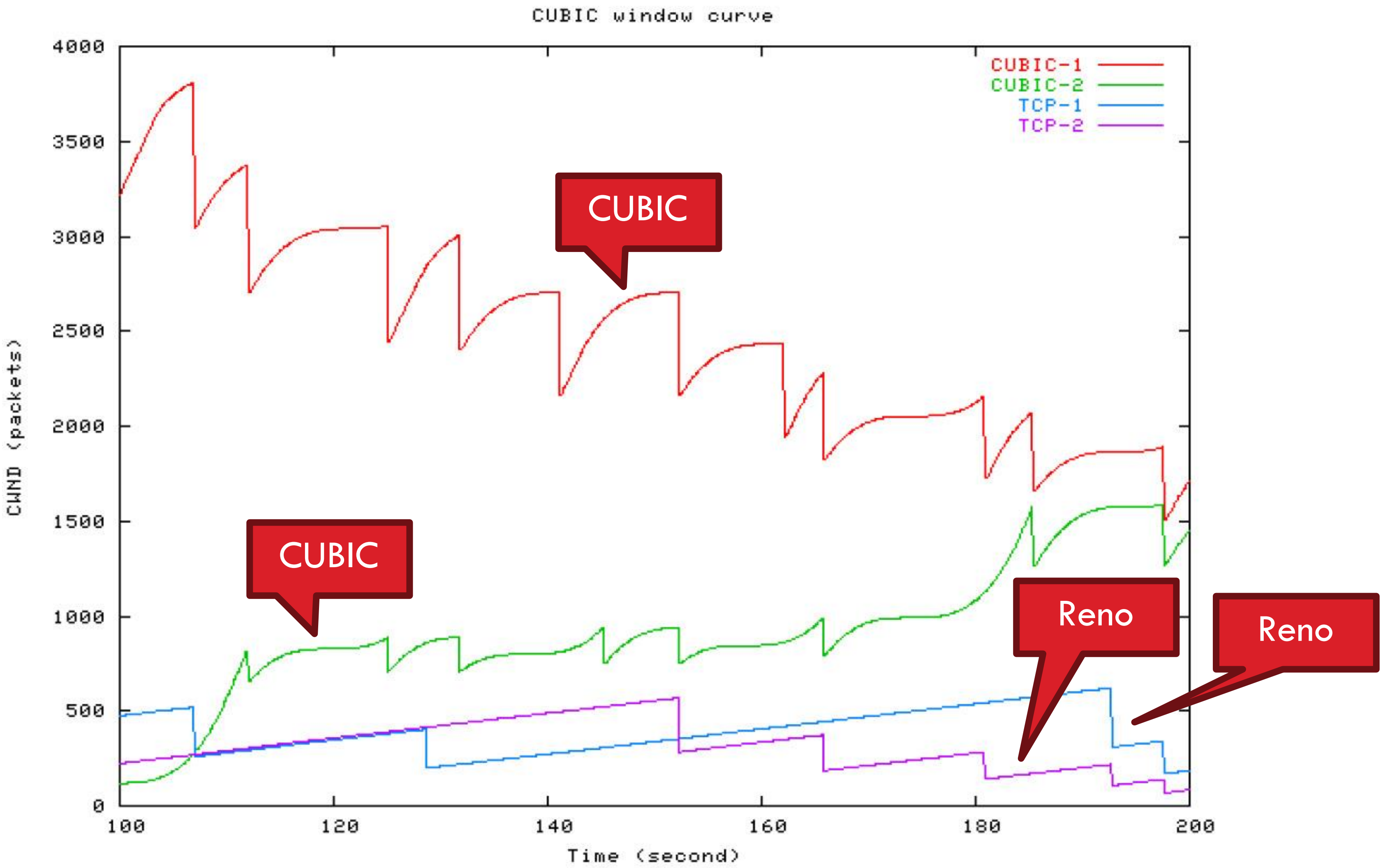
  - C → a constant scaling factor
  - β → a constant fraction for multiplicative decrease
  - T → time since last packet drop
  - $cwnd_{max}$ → cwnd when last packet dropped

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



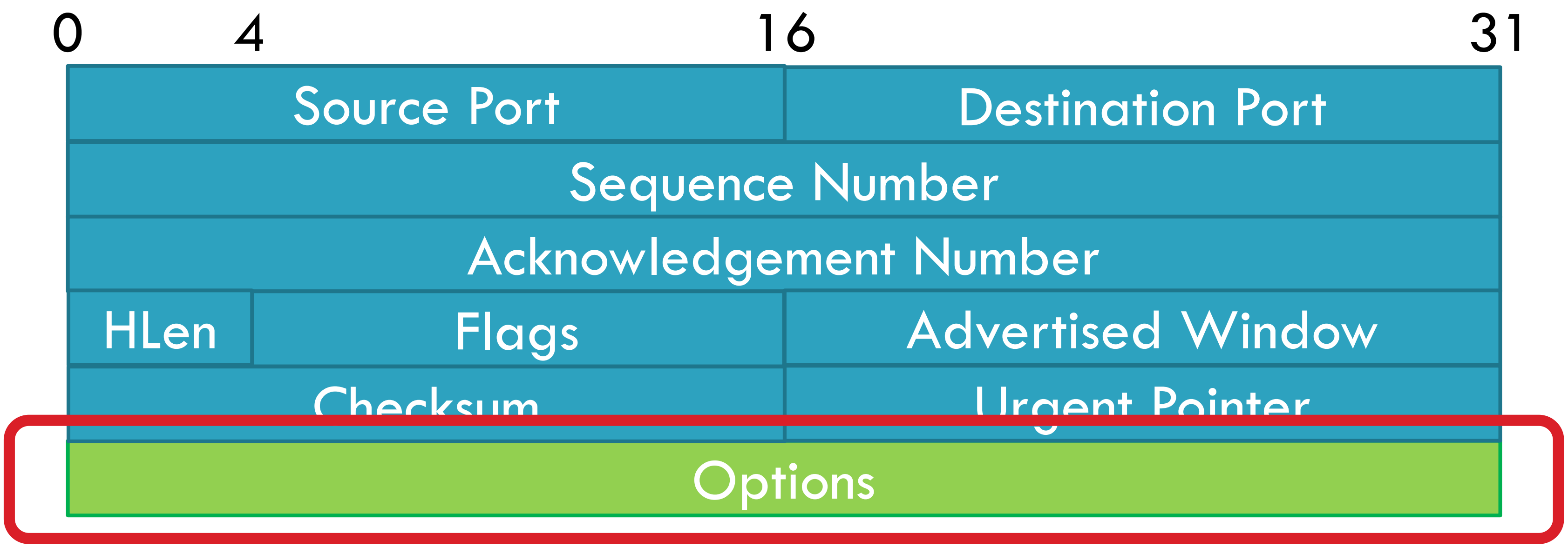
Deploying TCP Variants

- 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

Outline

- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

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

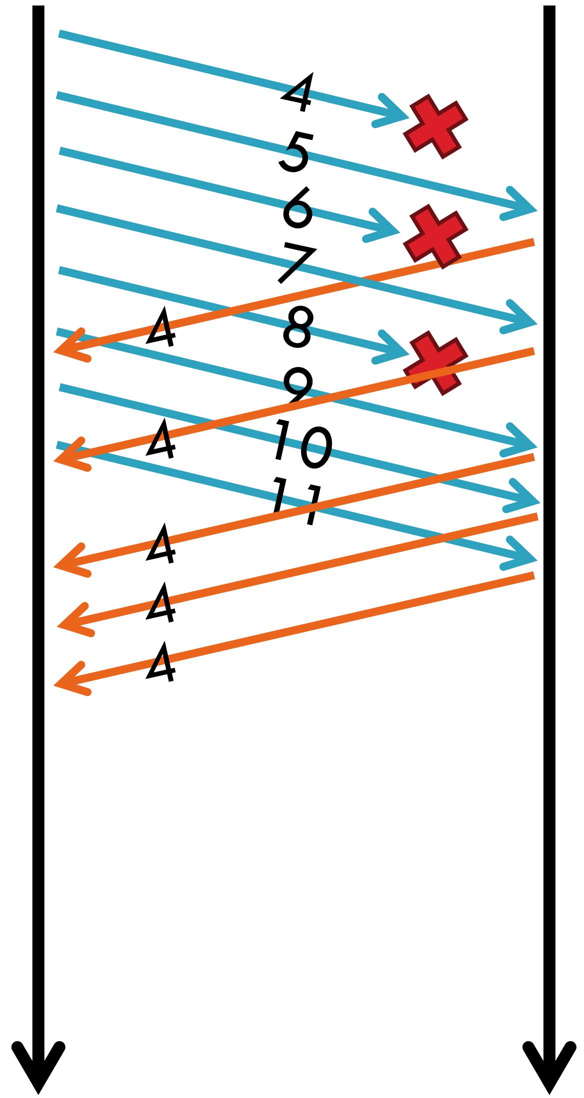
(1.5Mbps * 0.513s) = 94KB

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

- 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

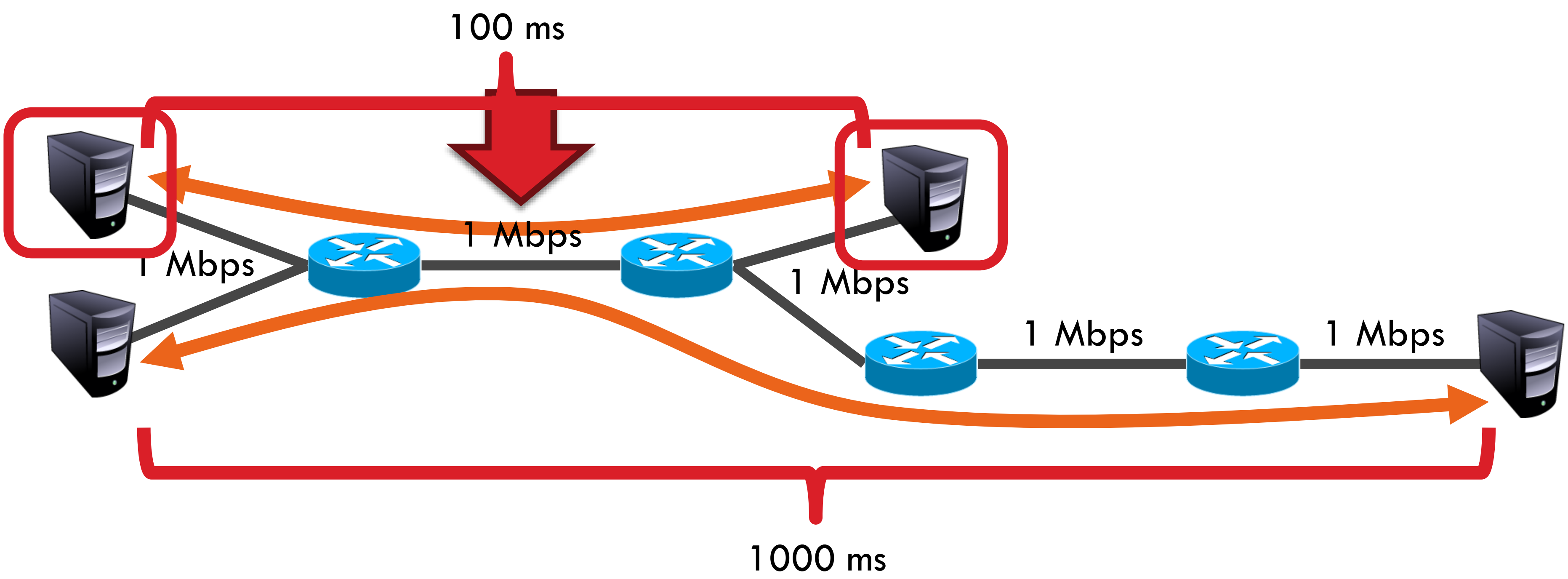
- Maximum segment size (MSS)
  - Essentially, what is the hosts MTU
  - Saves on path discovery overhead
- Timestamp
  - When was the packet sent (approximately)?
  - Used to prevent sequence number wraparound
  - PAWS algorithm

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

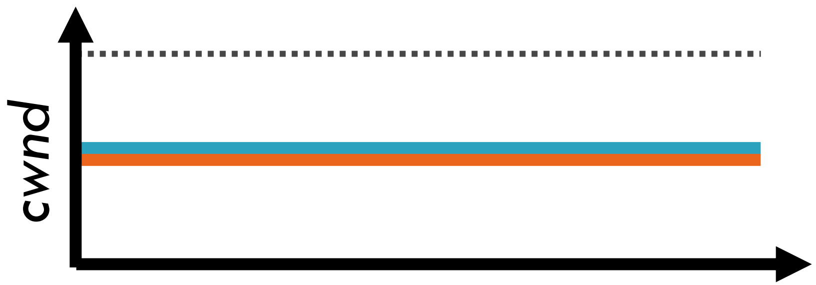
- Problem: TCP throughput depends on RTT



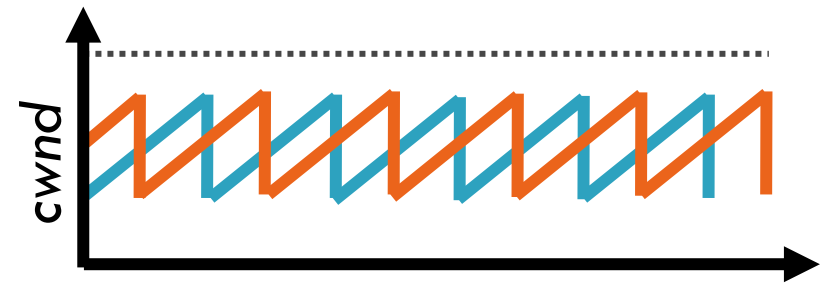
- ACK clocking makes TCP inherently unfair
- Possible solution: maintain a separate delay window
  - Implemented by Microsoft's Compound TCP

Synchronization of Flows

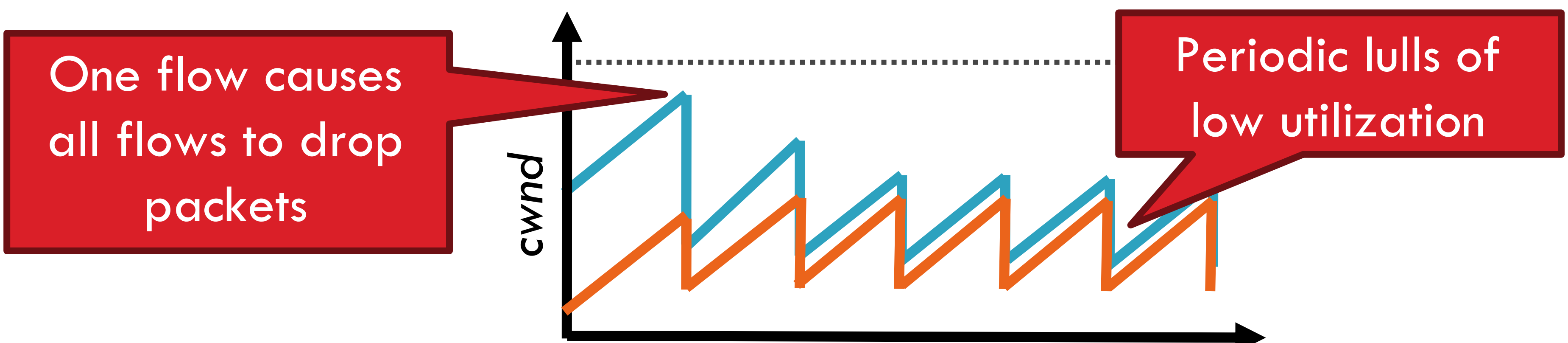
- Ideal bandwidth sharing



- 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
- 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
- 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)
    - More on this next week…

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

- 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
- 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.