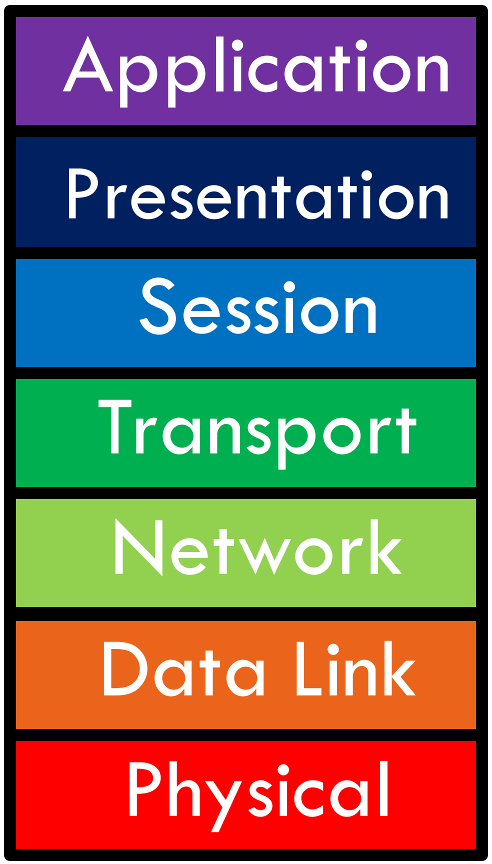
Computer Networks

Lecture 12: Transport Layer

Based on slides from D. Choffnes Northeastern U. and P. Gill from StonyBrook University
Revised Autumn 2015 by S. Laki

Transport Layer



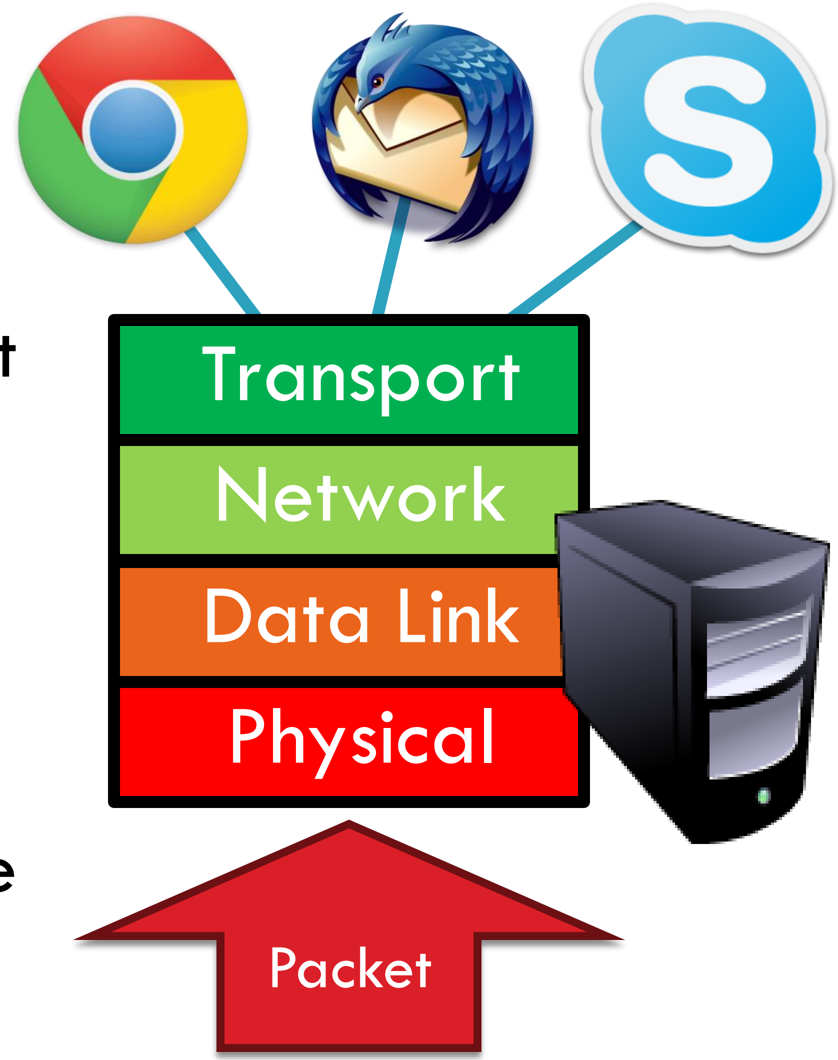
- Function:
  - Demultiplexing of data streams
- Optional functions:
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control
- Key challenges:
  - Detecting and responding to congestion
  - Balancing fairness against high utilization

Outline

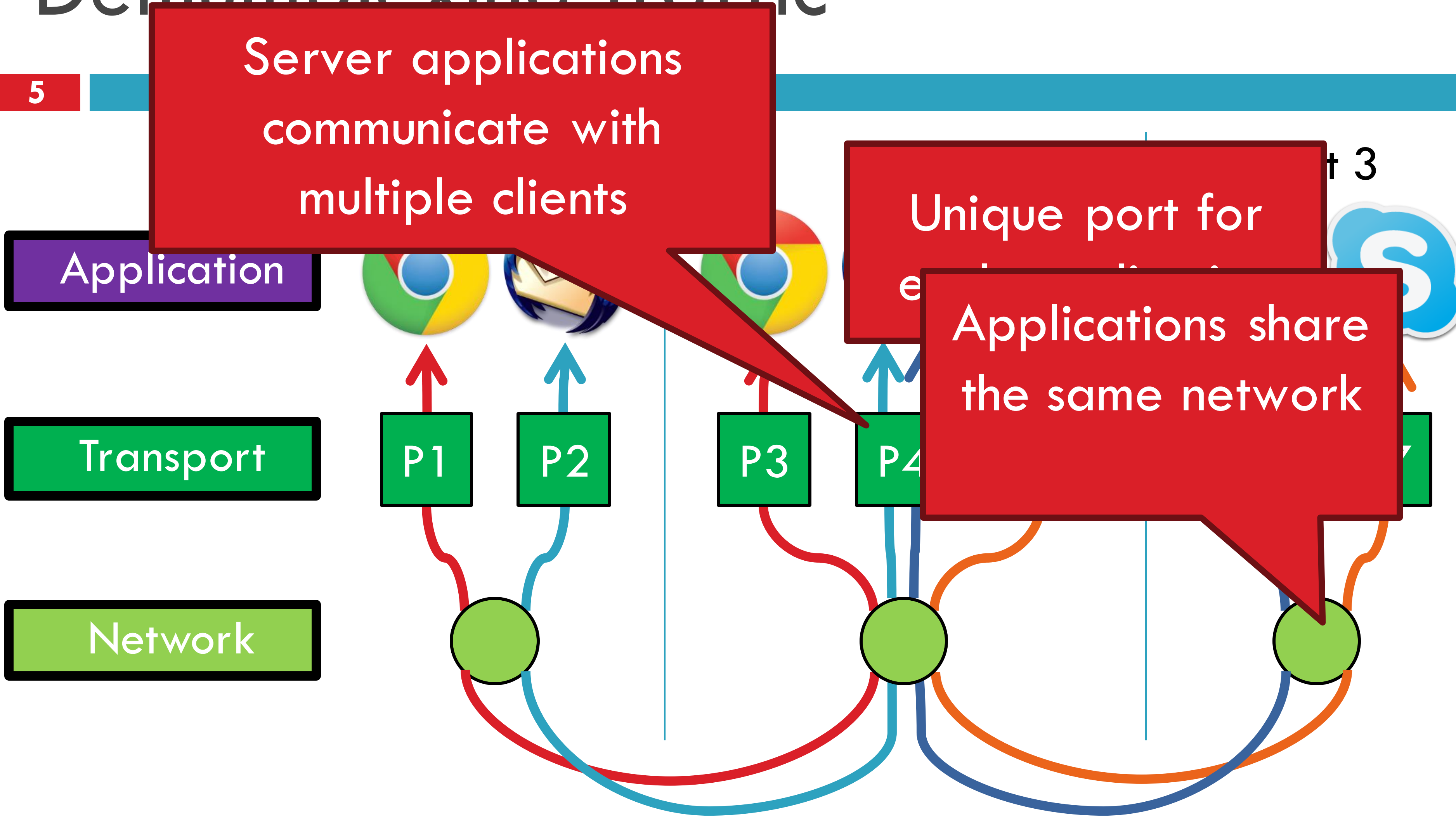
- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

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

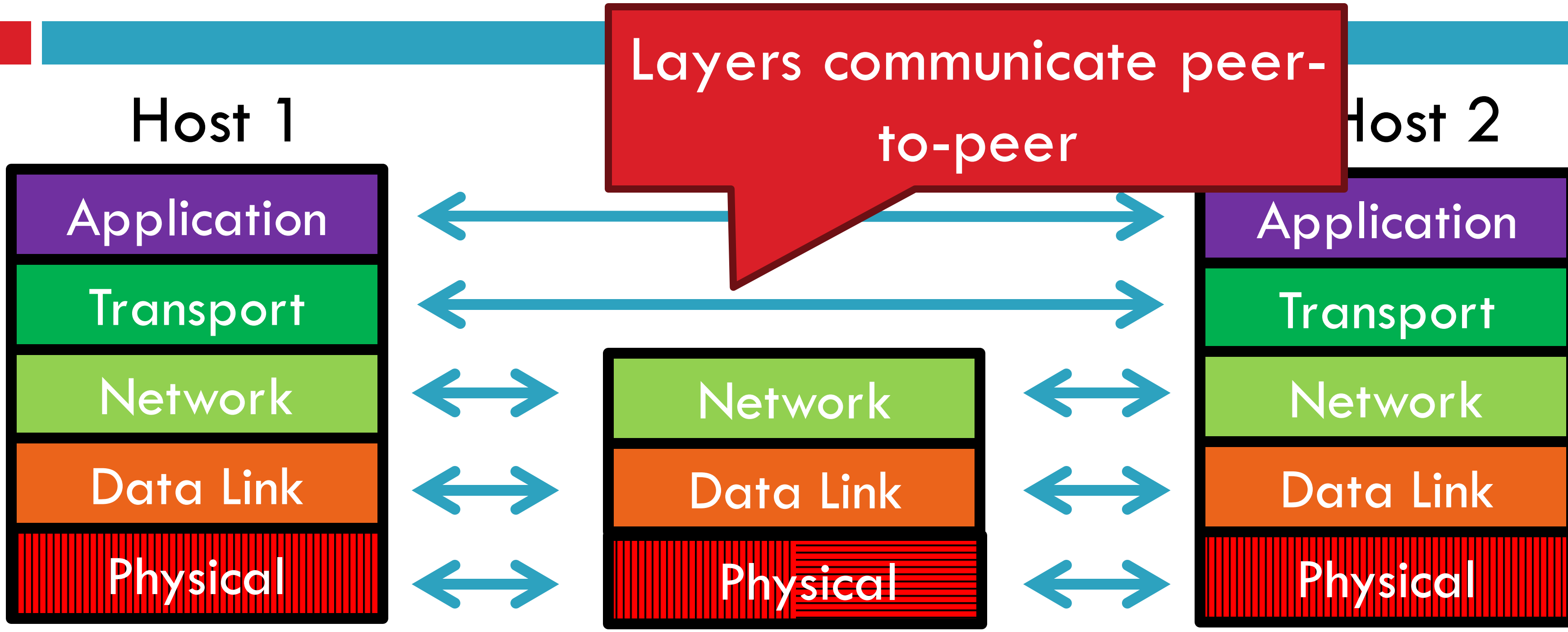


Demultiplexing Traffic



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

Layering, Revisited



- Lowest level end-to-end protocol
  - 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

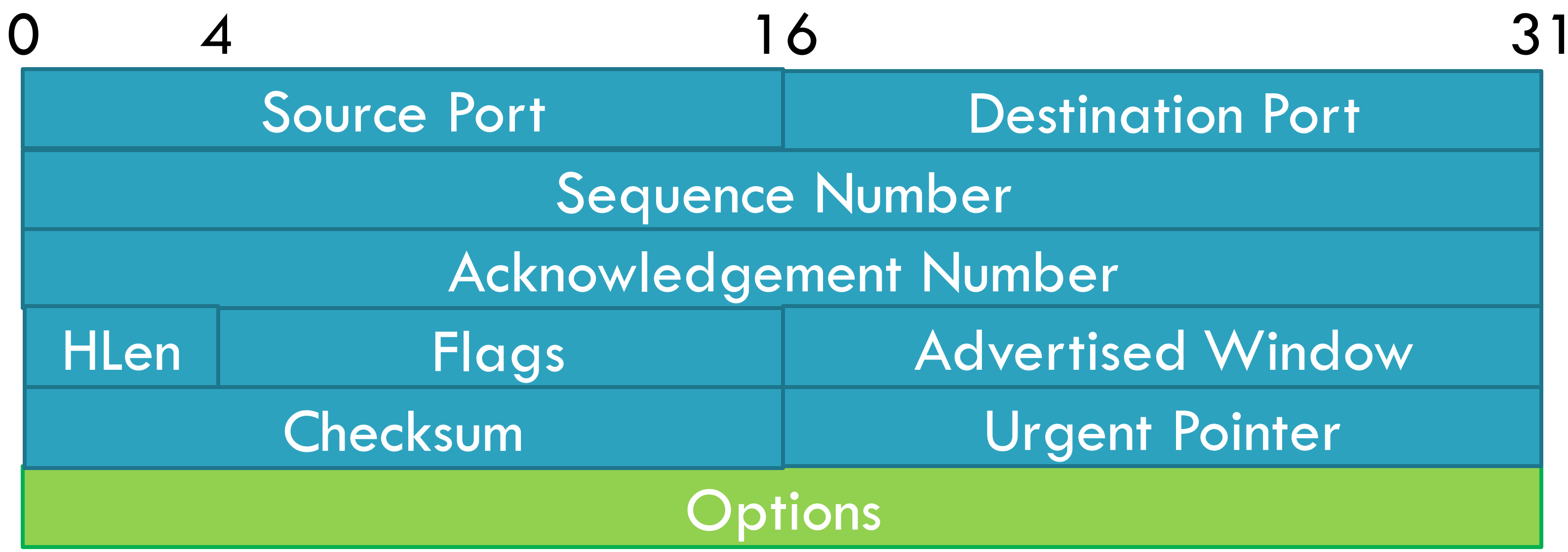
Outline

- UDP – already discussed
- 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

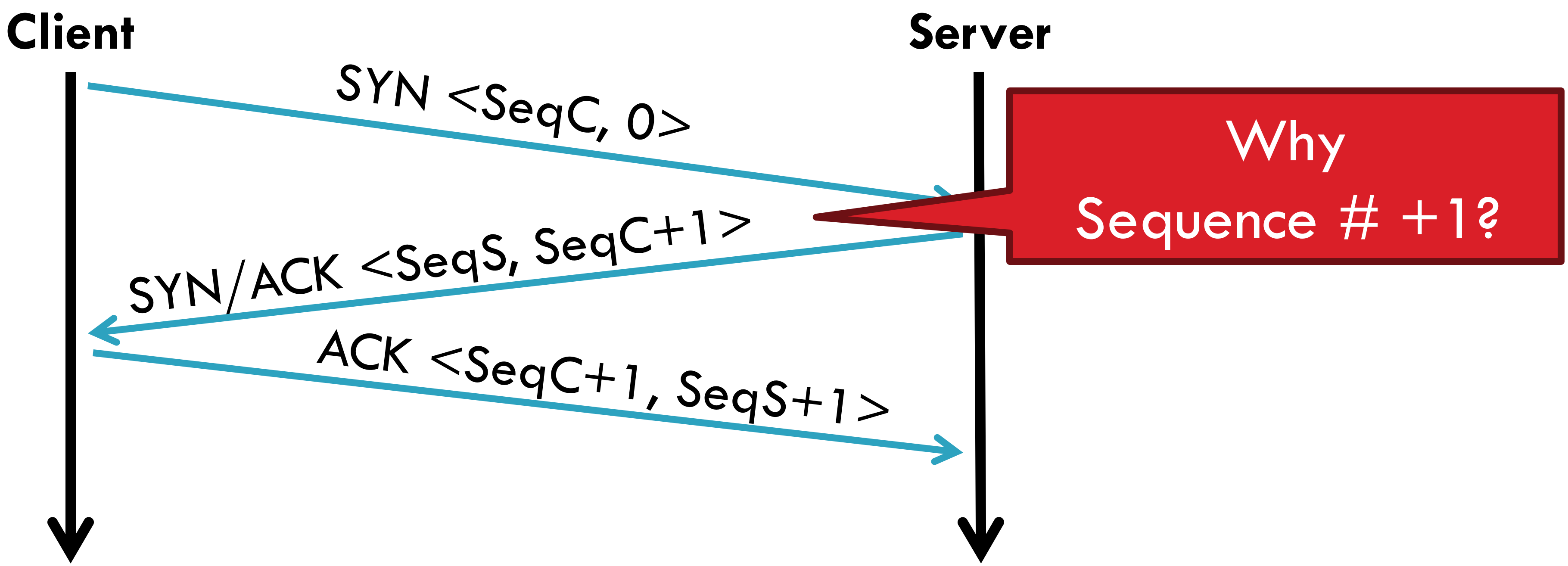
Why these features?



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

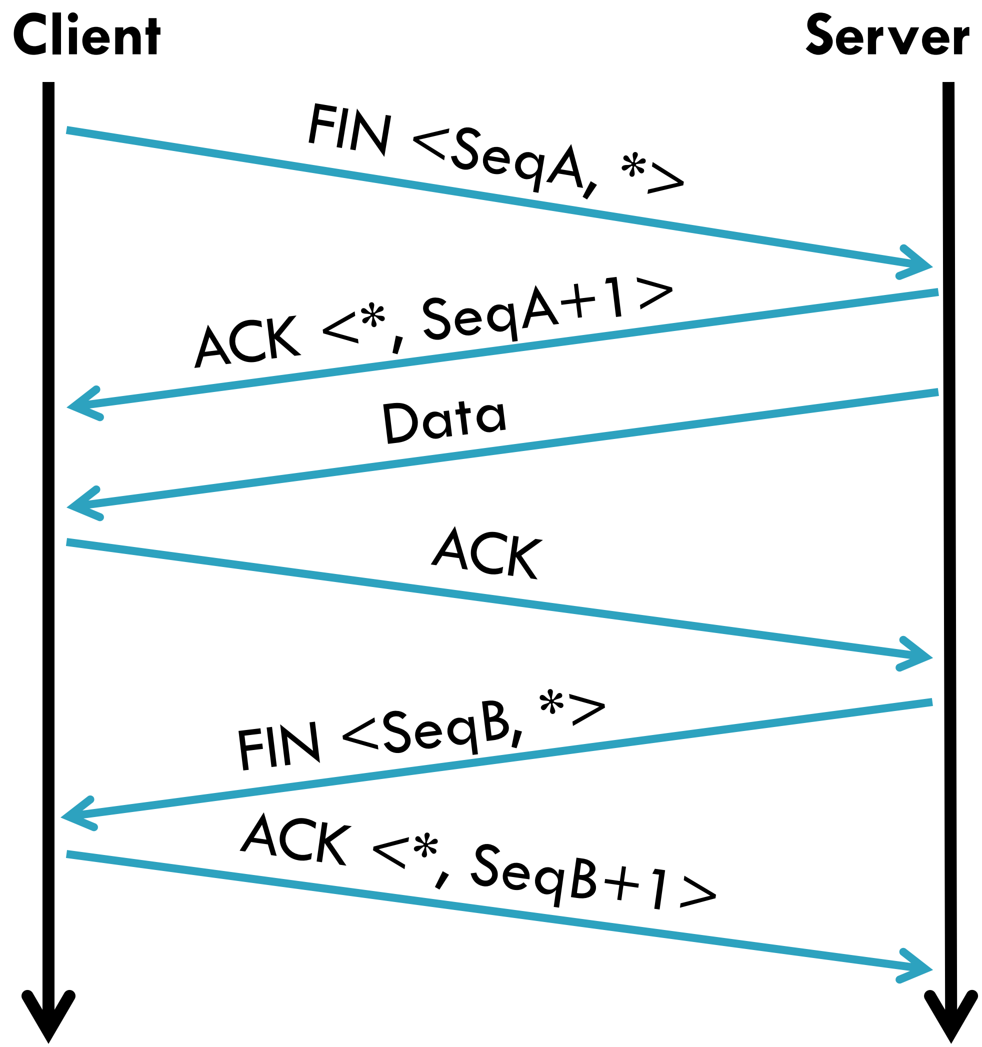
Three Way Handshake



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

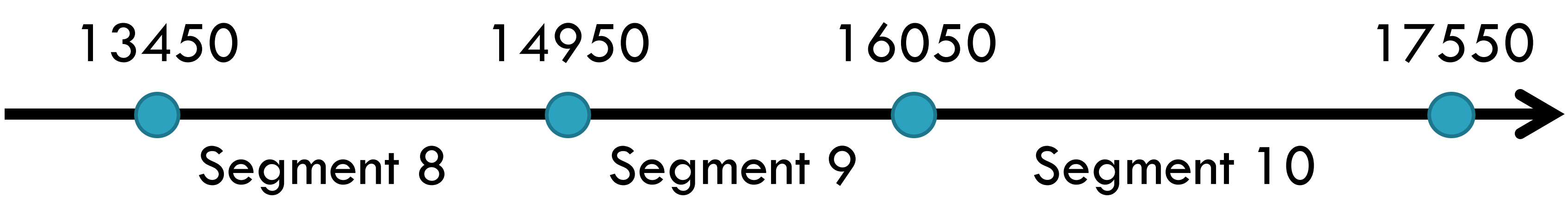
Connection Tear Down

- Either side can initiate tear down
- Other side may continue sending data
  - Half open connection
  - *shutdown()*
- Acknowledge the last FIN
  - Sequence number + 1
- What happens if 2nd FIN is lost?

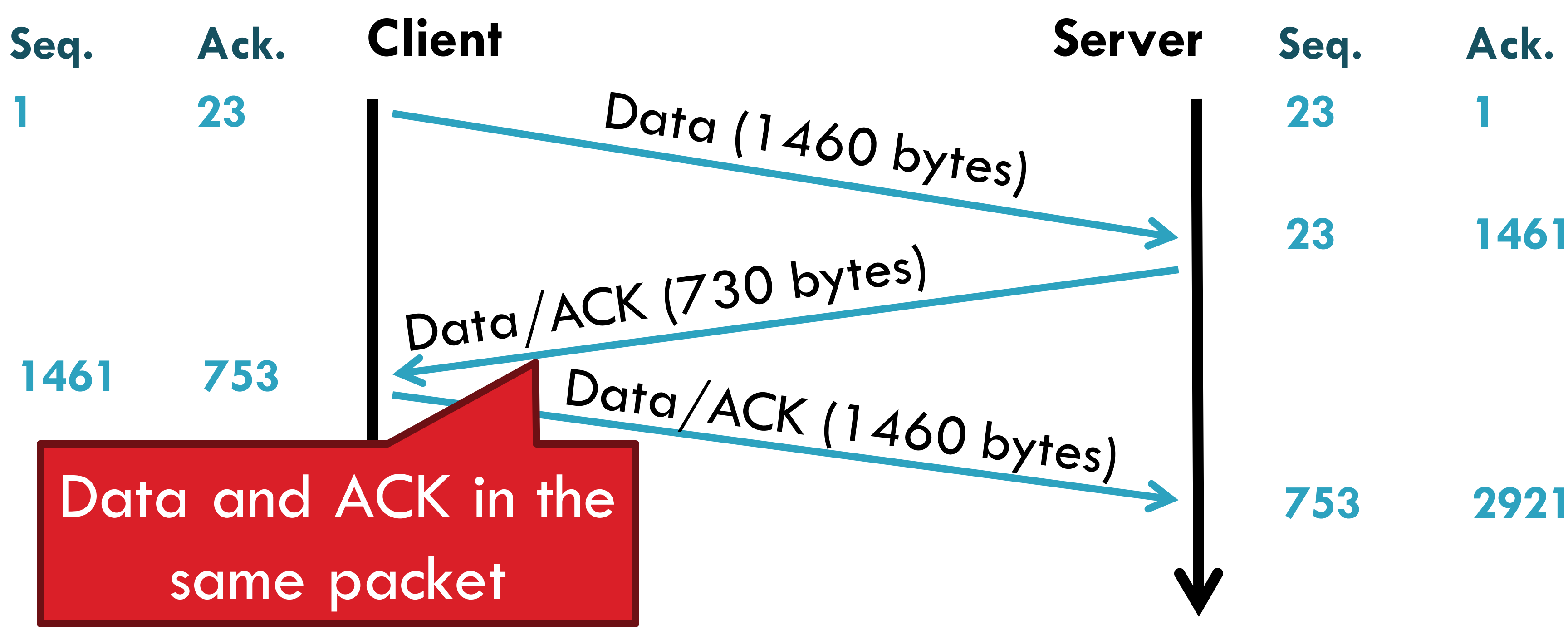


Sequence Number Space

- TCP uses a byte stream abstraction
  - Each byte in each stream is numbered
  - 32-bit value, wraps around
  - Initial, random values selected during setup. Why?
- 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

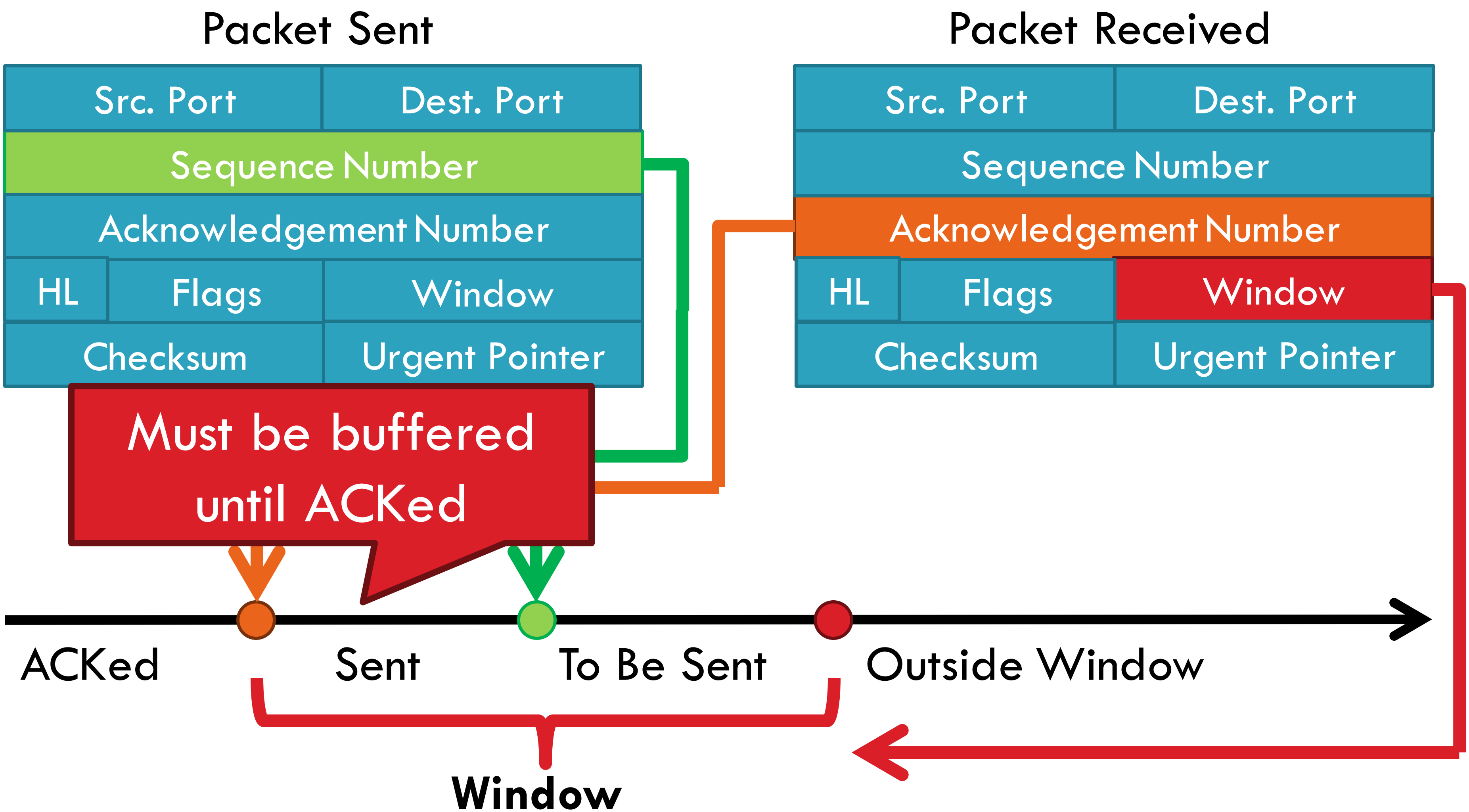


- Each side of the connection can send and receive
  - Different sequence numbers for each direction

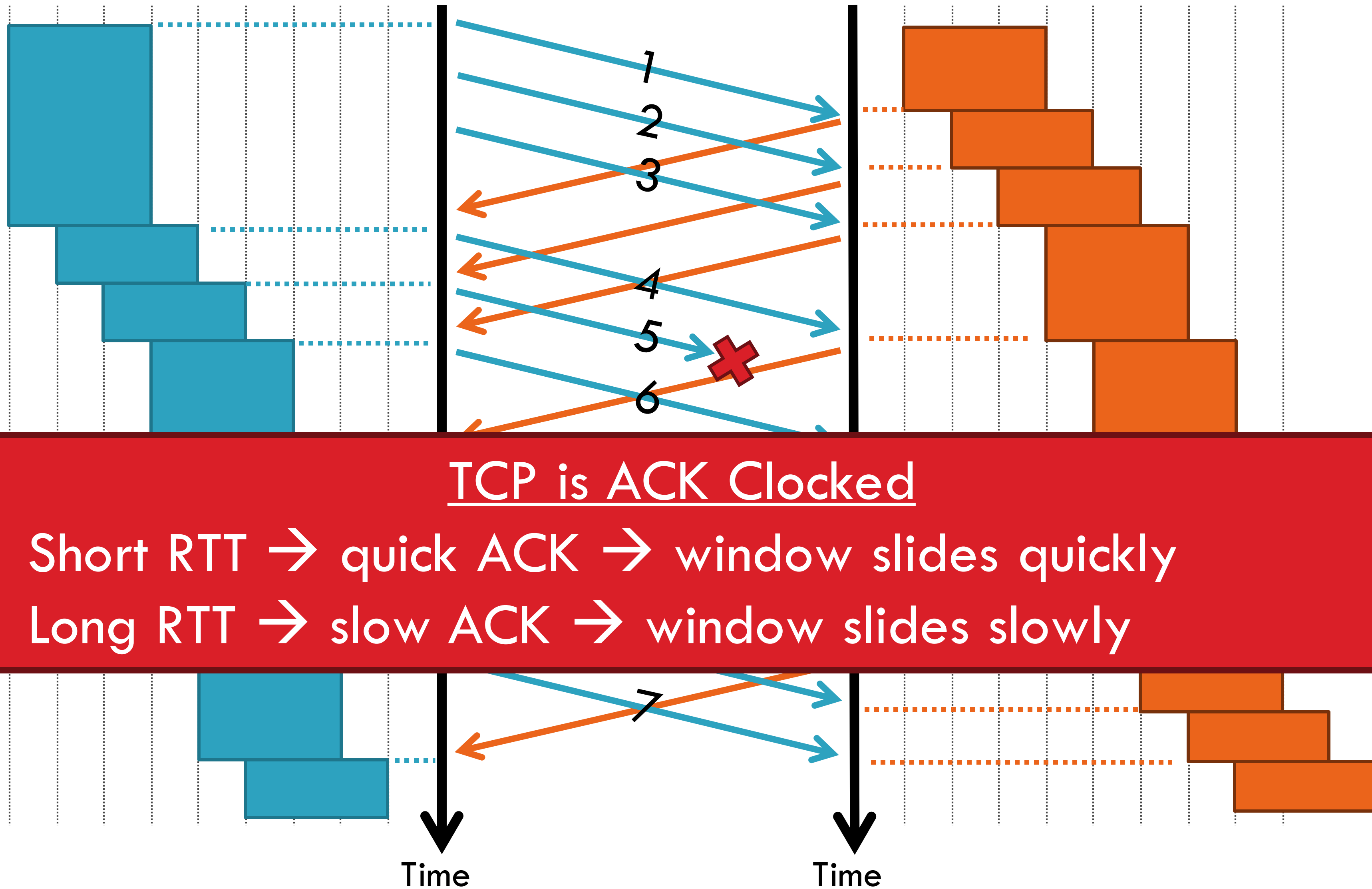
Flow Control

- 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



<u>TCP is ACK Clocked</u>

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

Observations

- Throughput is ~ w/RTT
- Sender has to buffer all unacknowledges packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to buffer limits

What Should the Receiver ACK?

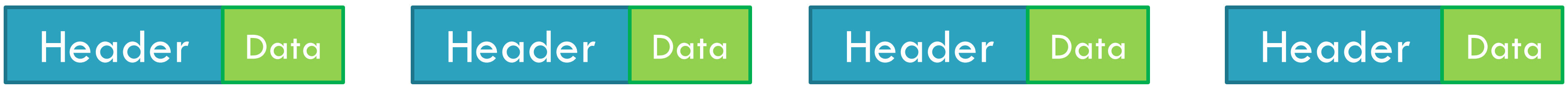
1. ACK every packet
2. Use *cumulative ACK*, where an ACK for sequence *n* implies ACKS for all $k < n$
3. Use *negative ACKs* (NACKs), indicating which packet did not arrive
4. Use *selective ACKs* (SACKs), indicating those that did arrive, even if not in order
   - SACK is an actual TCP extension

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 2 minutes

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

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

Nagle's Algorithm

1. If the window >= MSS and available data >= MSS:
   Send the data

   Send a full packet

2. Elif there is unACKed data:
   Enqueue data in a buffer until an ACK is received
3. Else: send the data

   Send a non-full packet if nothing else is happening

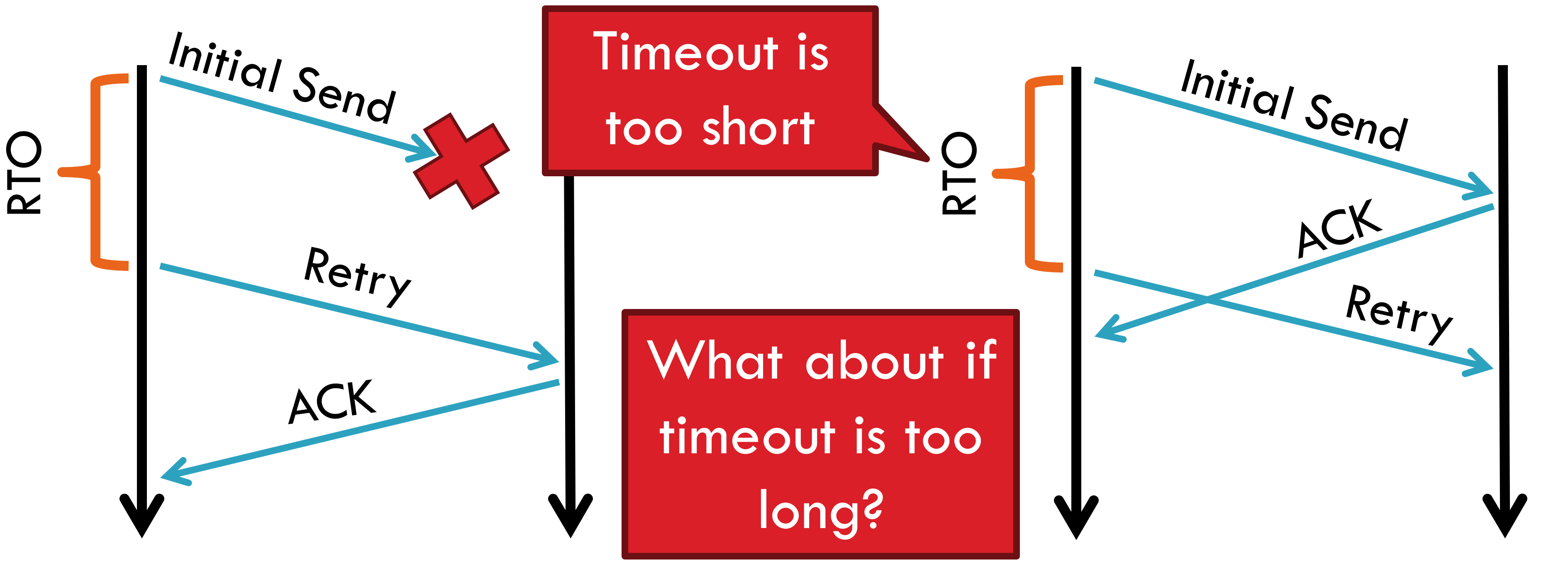
- Problem: Nagle's Algorithm delays transmissions
  - What if you need to send a packet immediately?
    1. int flag = 1;
    2. setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *) &flag, sizeof(int));

Error Detection

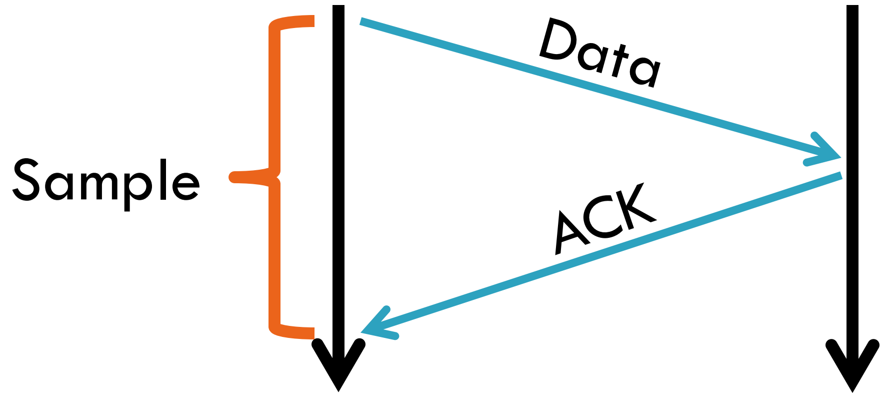
- Checksum detects (some) packet corruption
  - Computed over IP header, TCP header, and data
- Sequence numbers catch sequence problems
  - Duplicates are ignored
  - Out-of-order packets are reordered or dropped
  - Missing sequence numbers indicate lost packets
- Lost segments detected by sender
  - 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

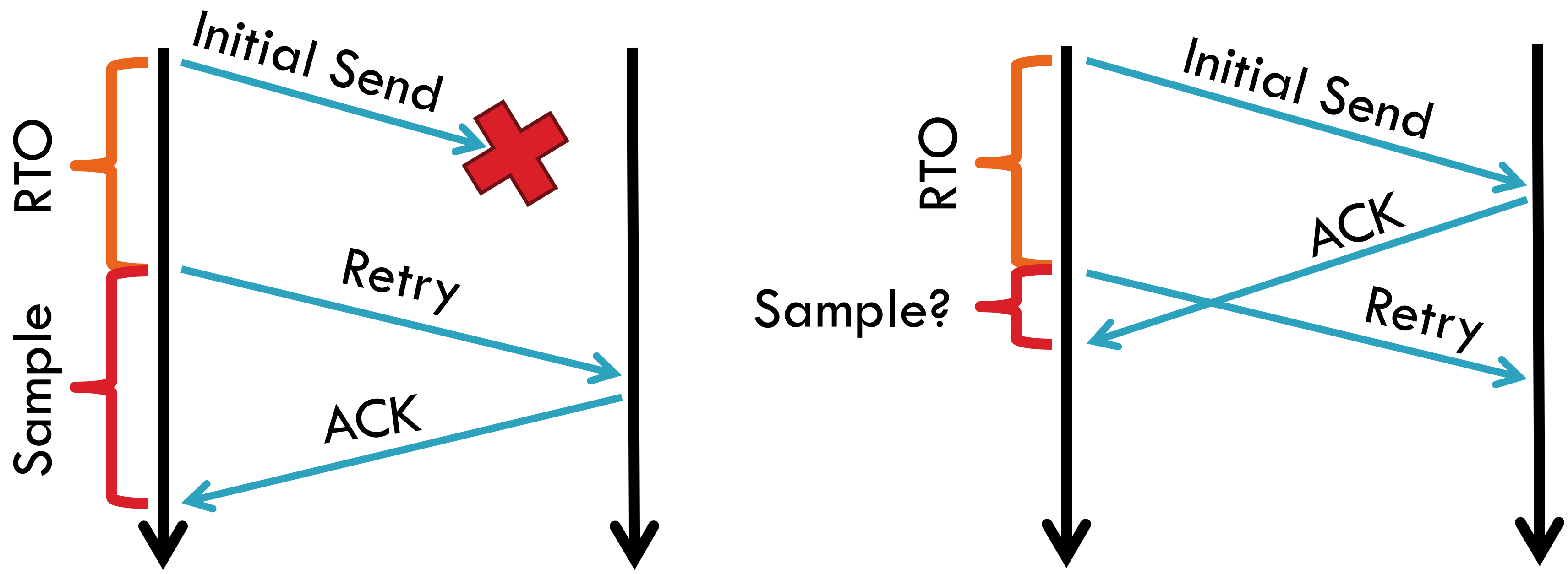


Round Trip Time Estimation



- 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

TCP Congestion Control

- **The network is congested if the load in the network is higher than its capacity.**
- 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

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

Rate Adjustment

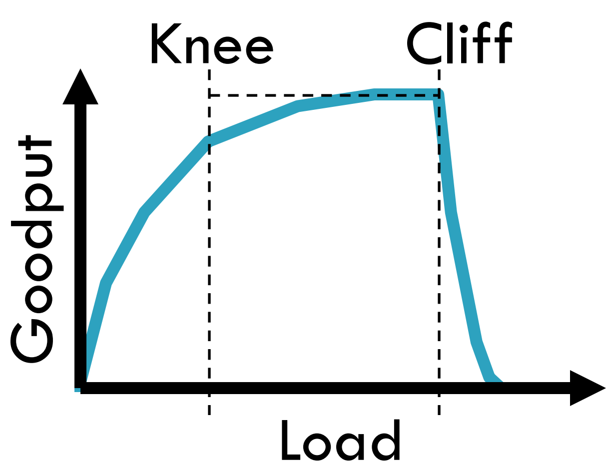
- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly
- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - *cwnd* growth is proportional to RTT
  - On loss: decrease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

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

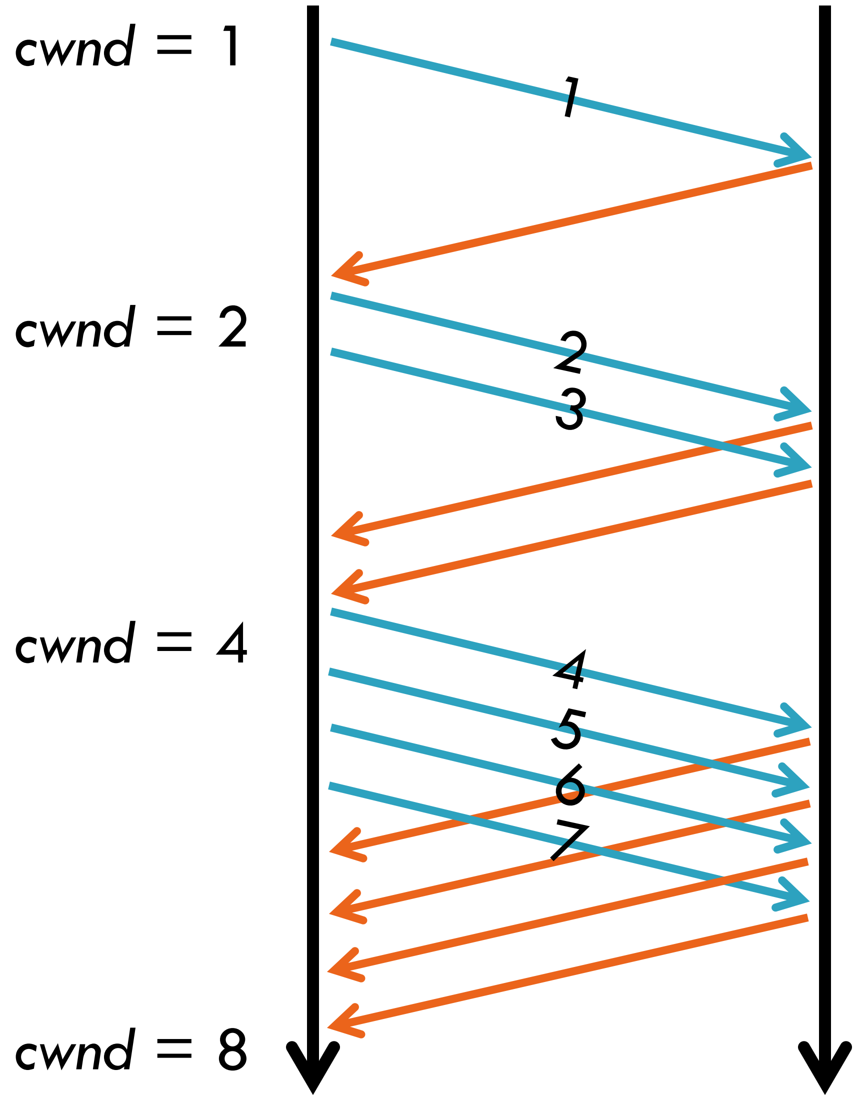
Slow Start

- 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

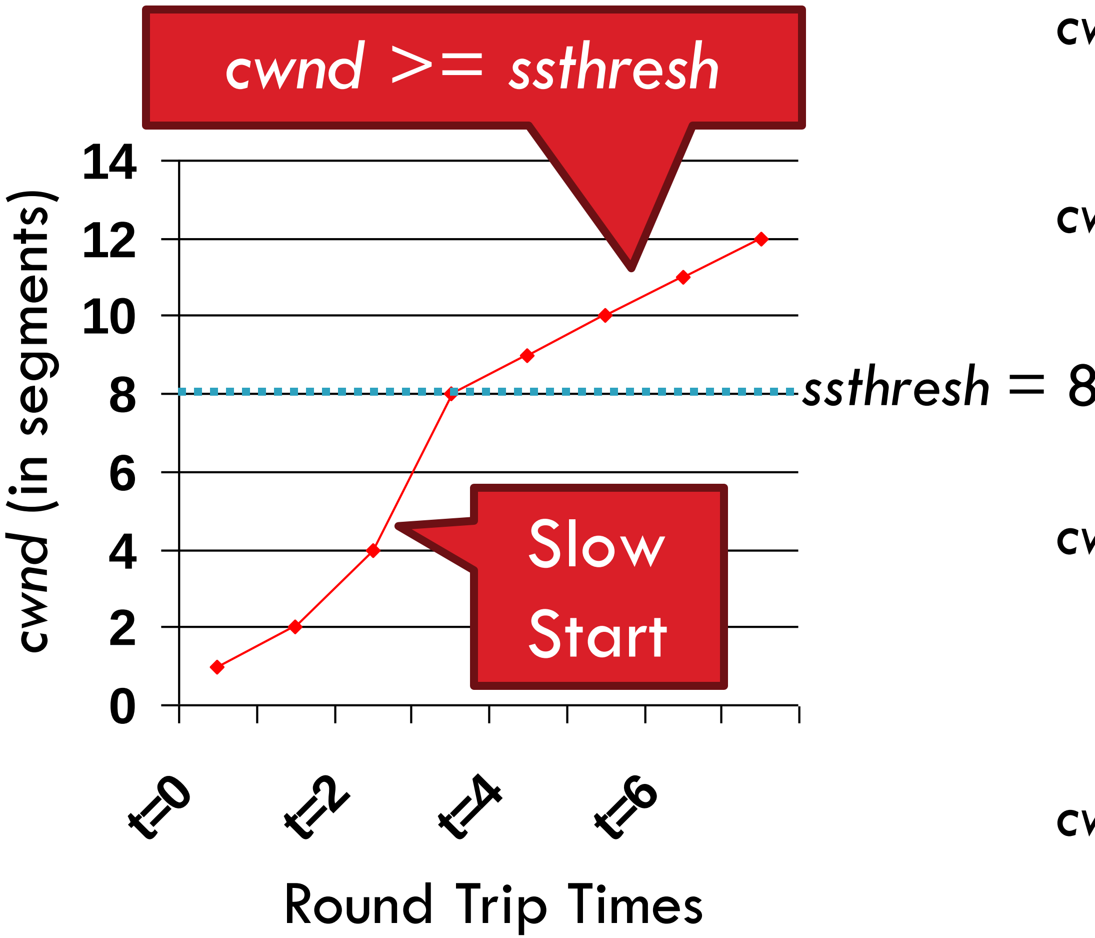
- *cwnd* grows rapidly
- Slows down when…
  - *cwnd* >= *ssthresh*
  - Or a packet drops

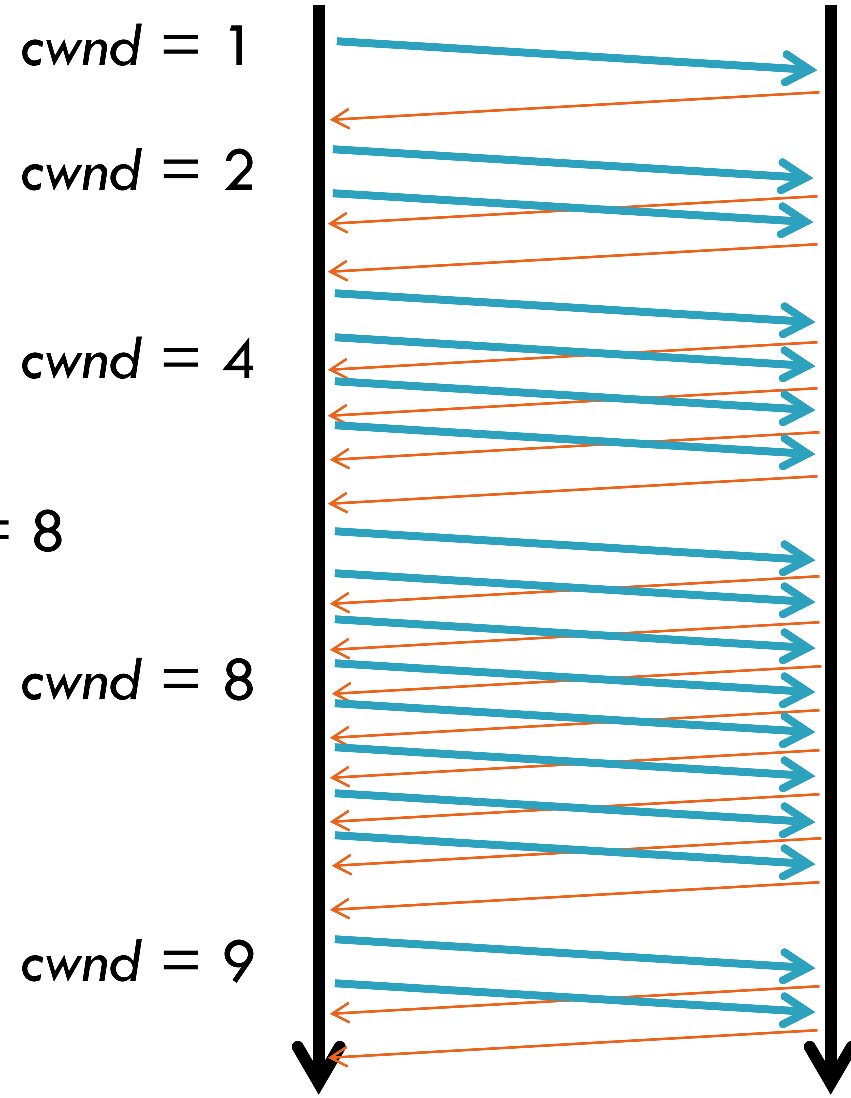


Congestion Avoidance

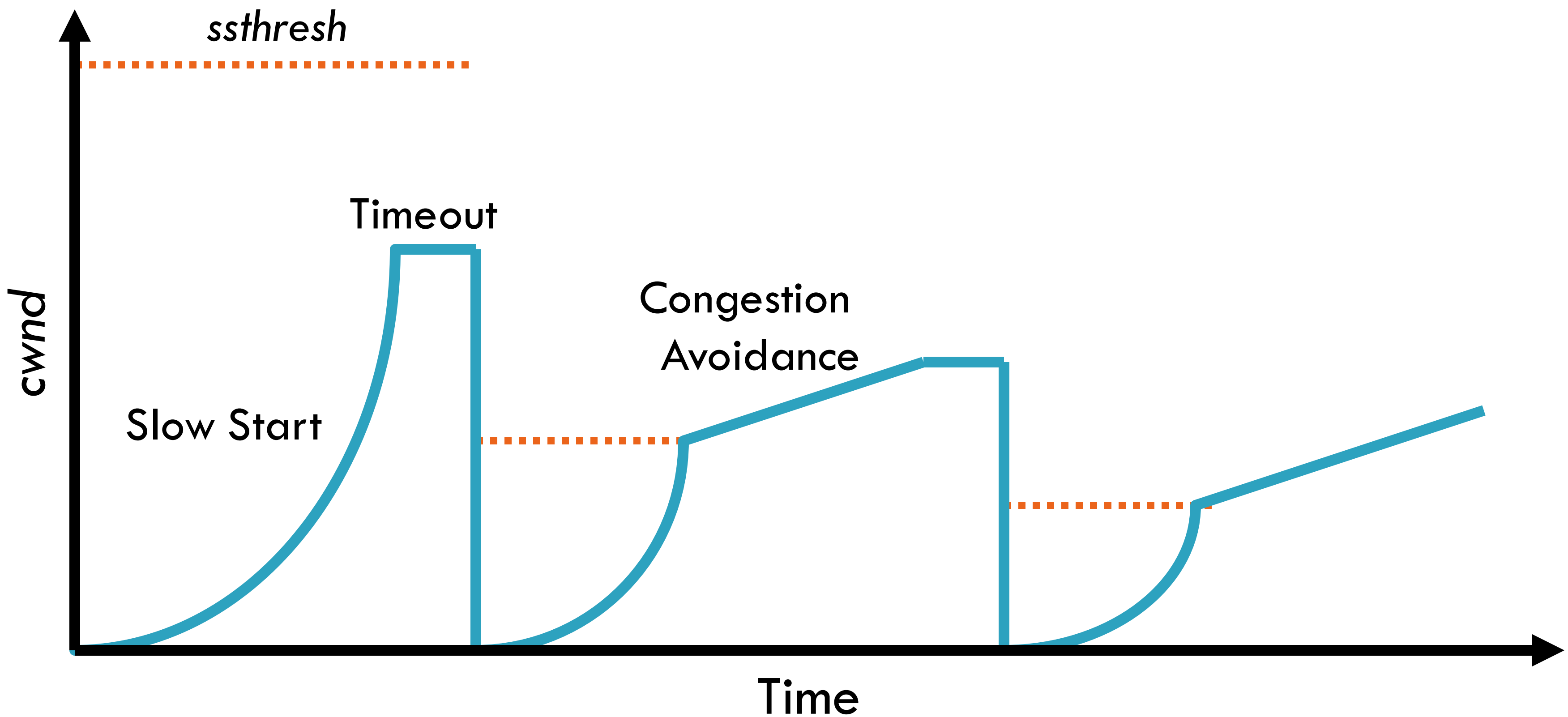
- Additive Increase Multiplicative Decrease (AIMD) mode
- *ssthresh* is lower-bound guess about location of the knee
- **If** *cwnd* >= *ssthresh* **then**
  each time a segment is ACKed
  increment *cwnd by 1/cwnd* (*cwnd* += 1/*cwnd*).
- So *cwnd* is increased by one only if all segments have been acknowledged

Congestion Avoidance Example





The Big Picture – TCP Tahoe (the original TCP)



Outline

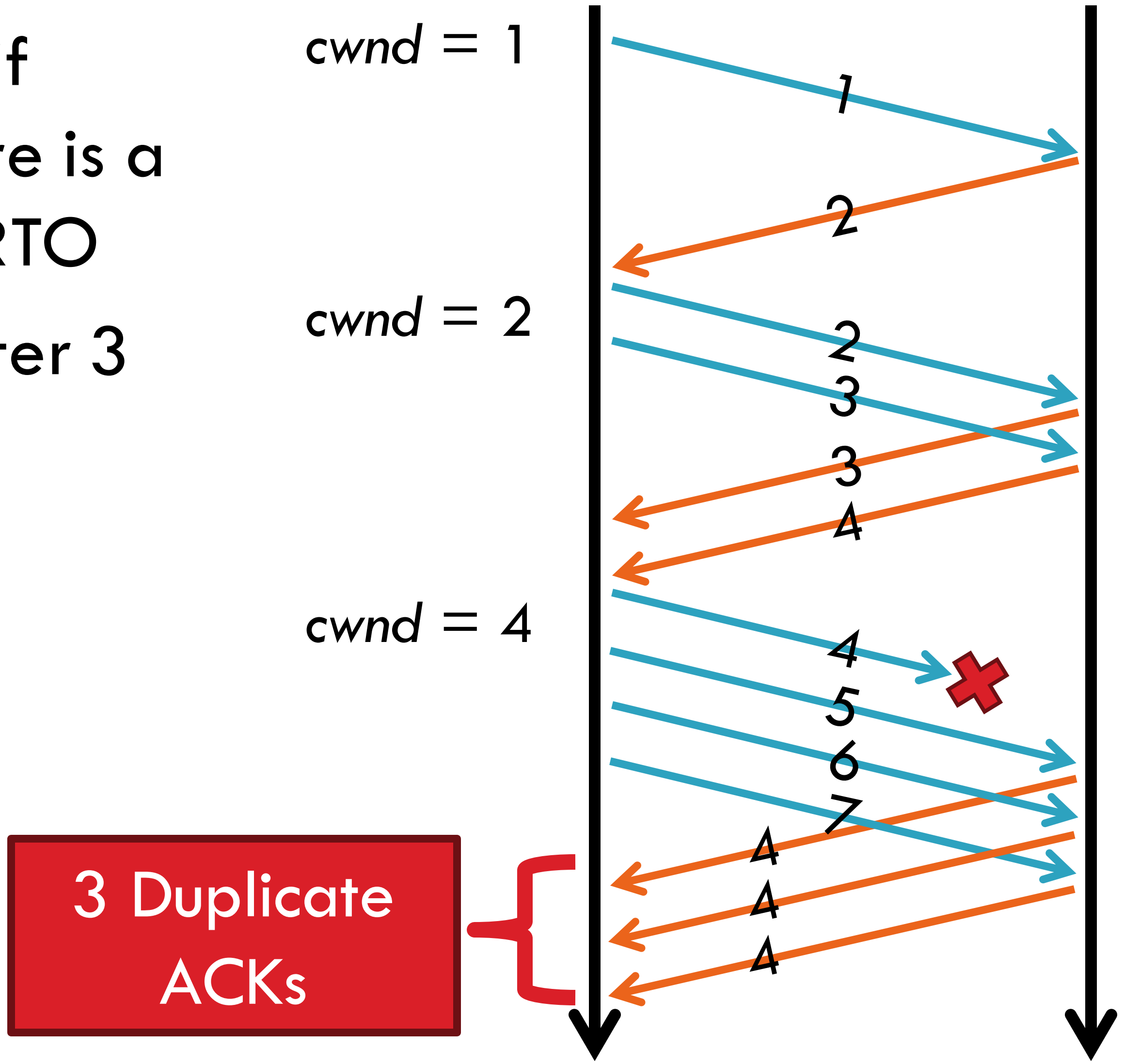
- UDP
- TCP
- Congestion Control
- **Evolution of TCP**
- Problems with TCP

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
    - 3 duplicate ACKs? -> retransmit (don't wait for RTO)
  - Fast recovery
    - On loss: set cwnd = cwnd/2 (ssthresh = new cwnd value)

TCP Reno: Fast Retransmit

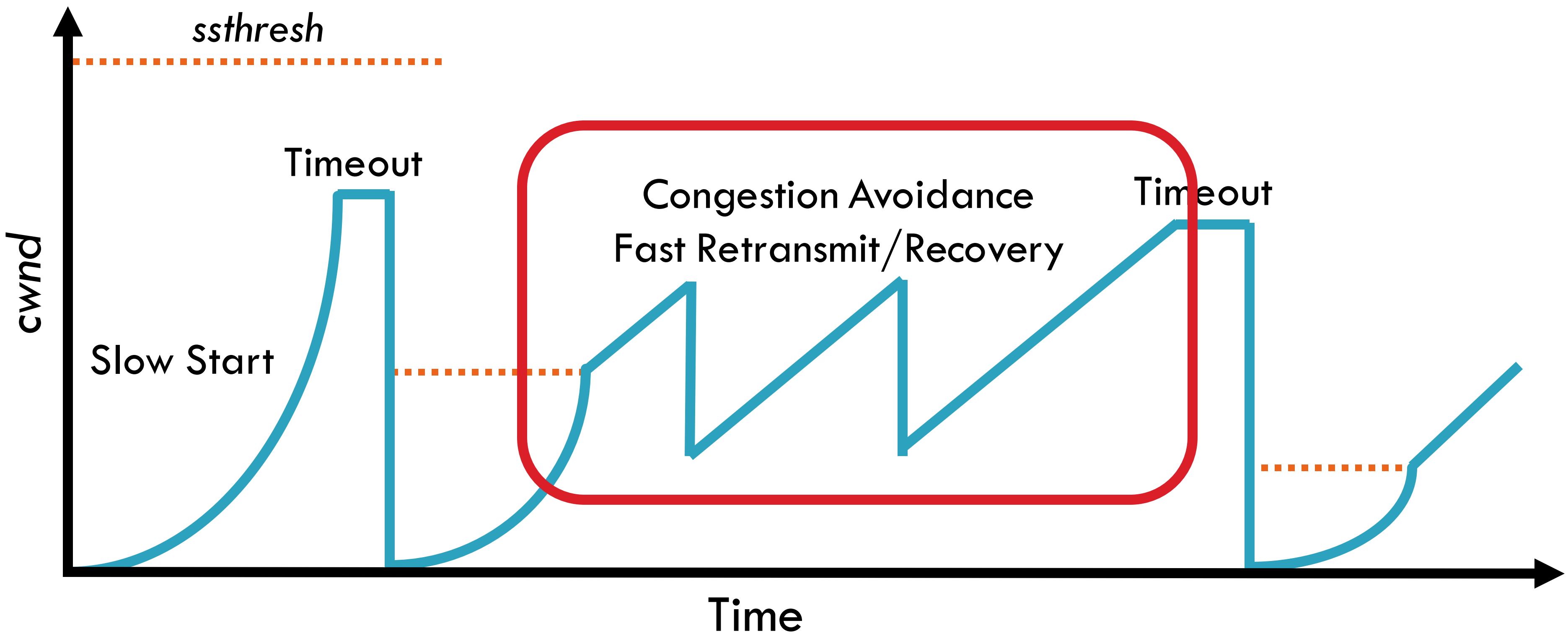
- Problem: in Tahoe, if segment is lost, there is a long wait until the RTO
- Reno: retransmit after 3 duplicate ACKs



TCP Reno: Fast Recovery

- After a fast-retransmit set *cwnd* to *cwnd*/2
  - Also reset ssthresh to the new halved cwnd value
  - 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

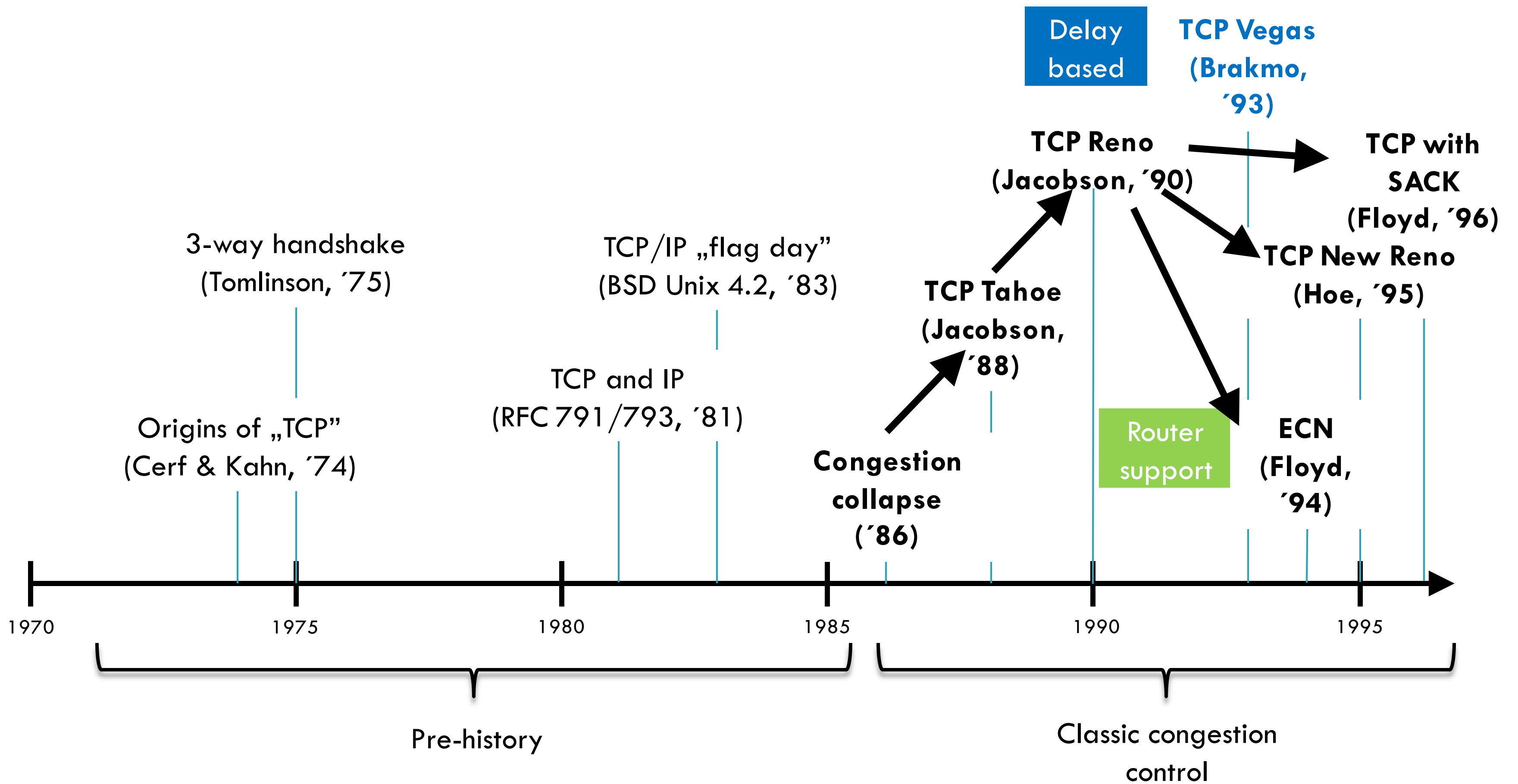


- At steady state, *cwnd* oscillates around the optimal window size
- TCP always forces packet drops

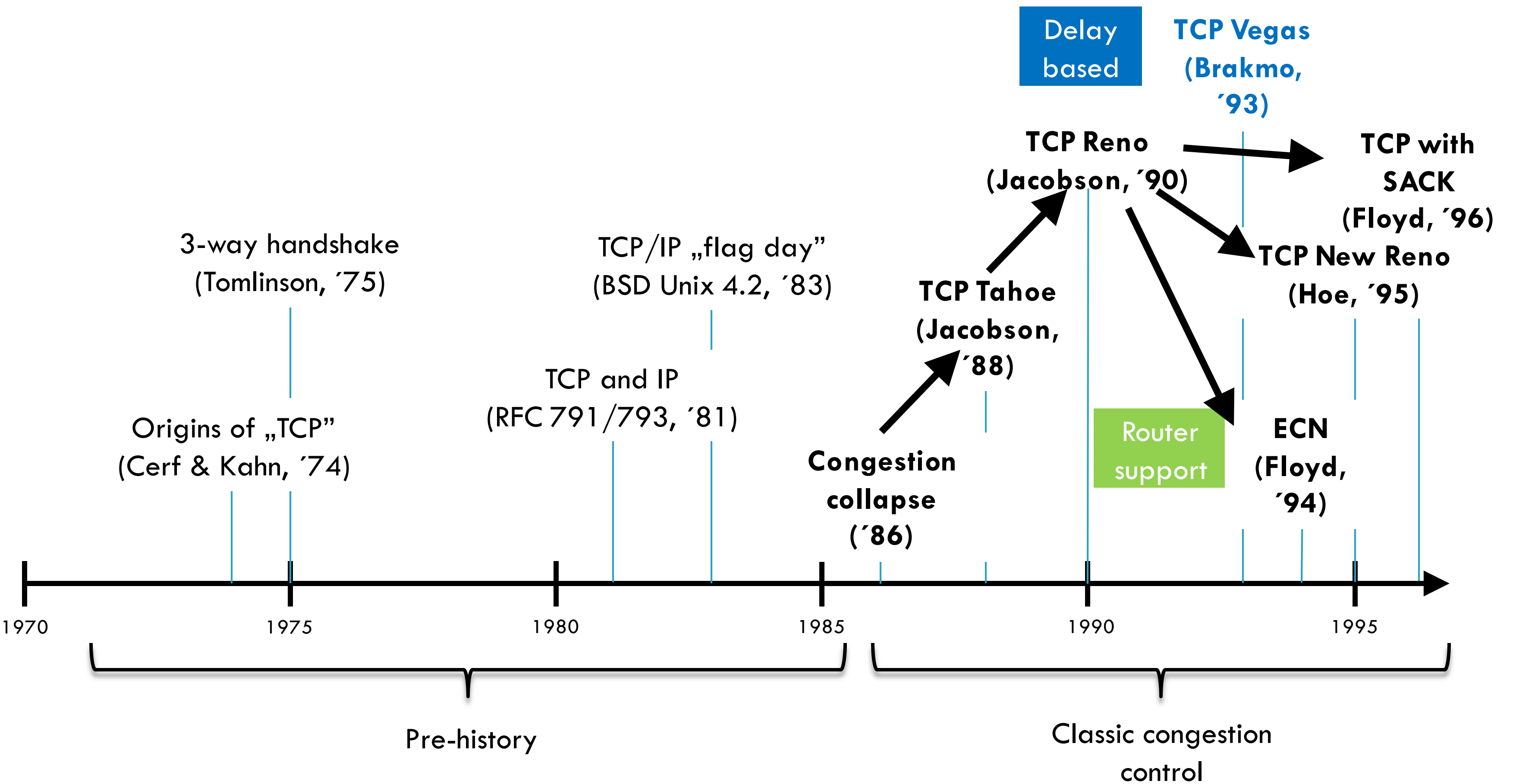
Many TCP Variants…

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno:
  - fast retransmit (3 dupACKs)
  - fast recovery (cwnd = cwnd/2 on loss)
- 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…

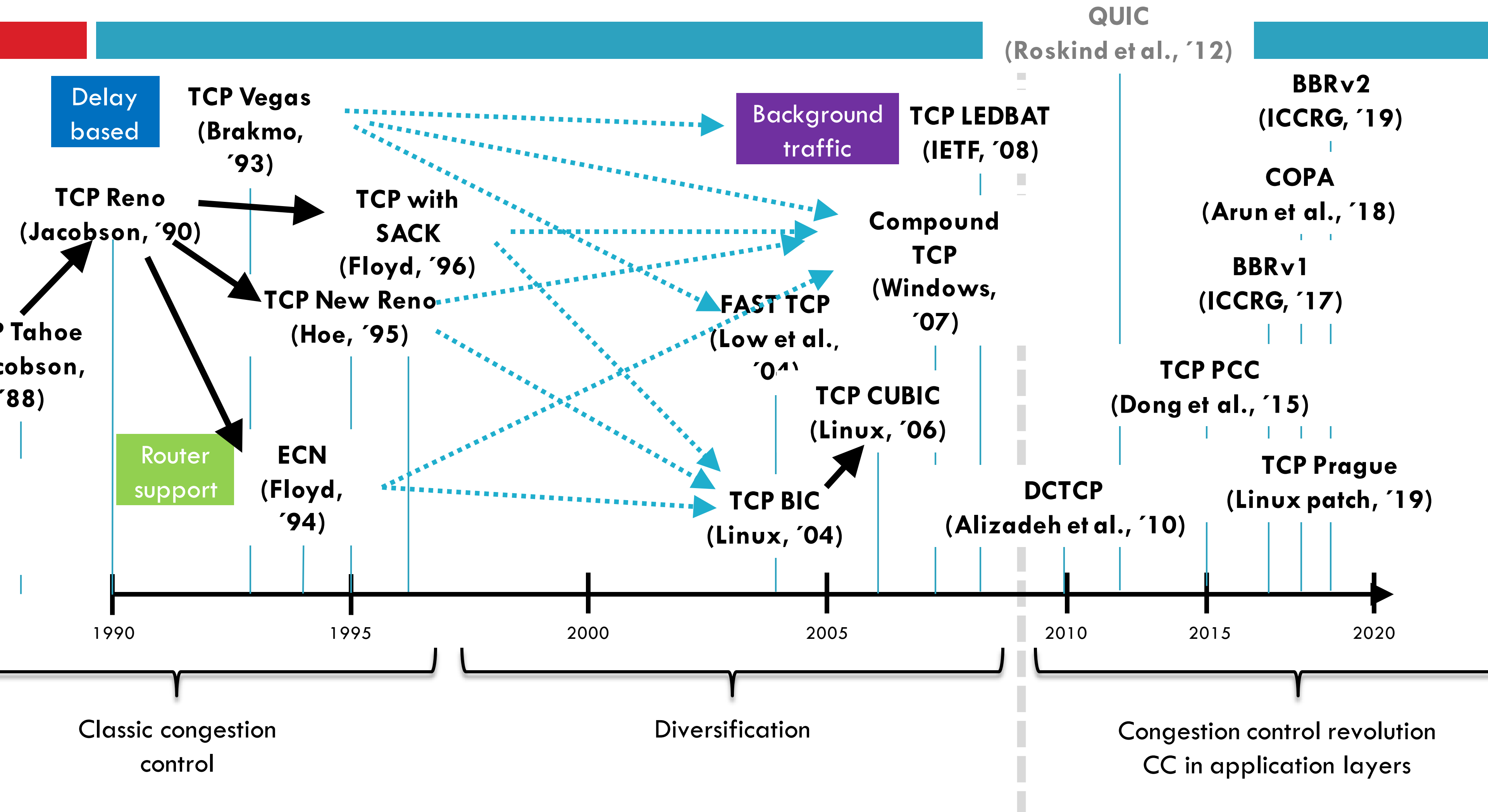
Transport layer evolution



Transport layer evolution



Transport layer (r)evolution



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

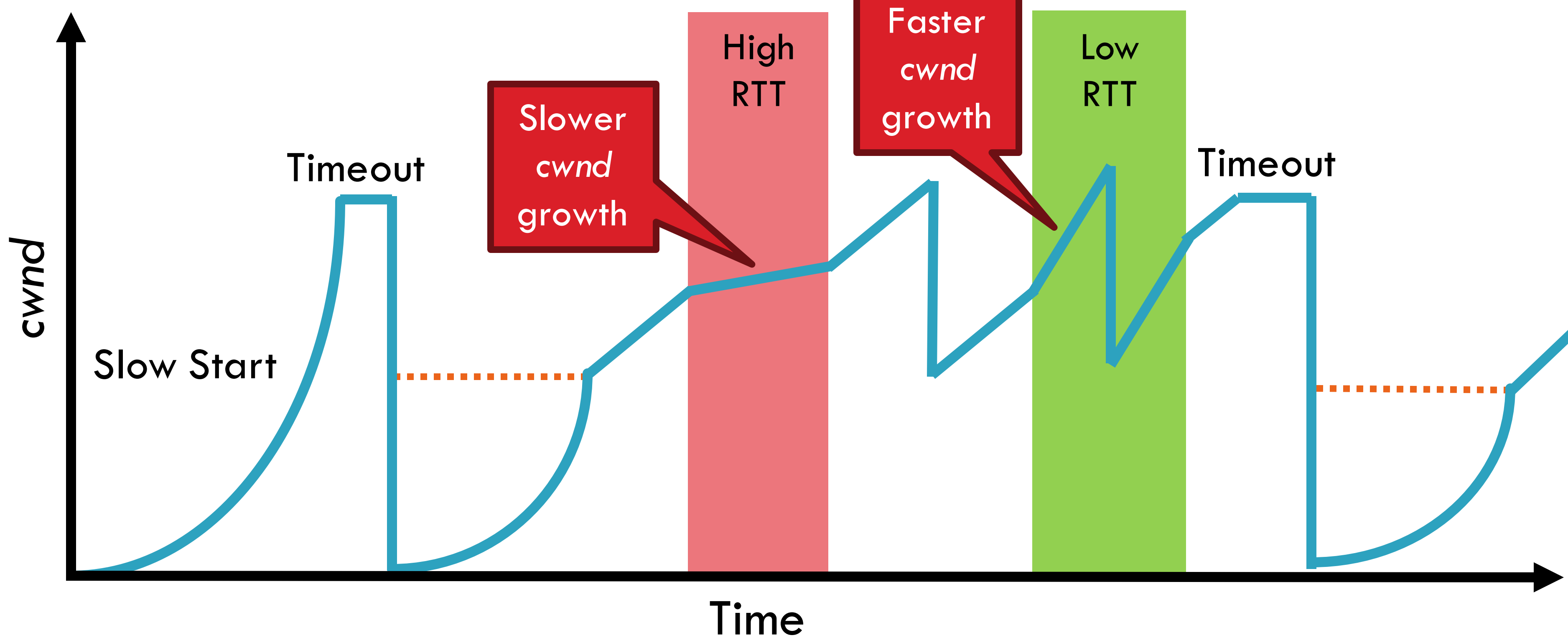
Goals

- 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 (before Win 10)
- 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

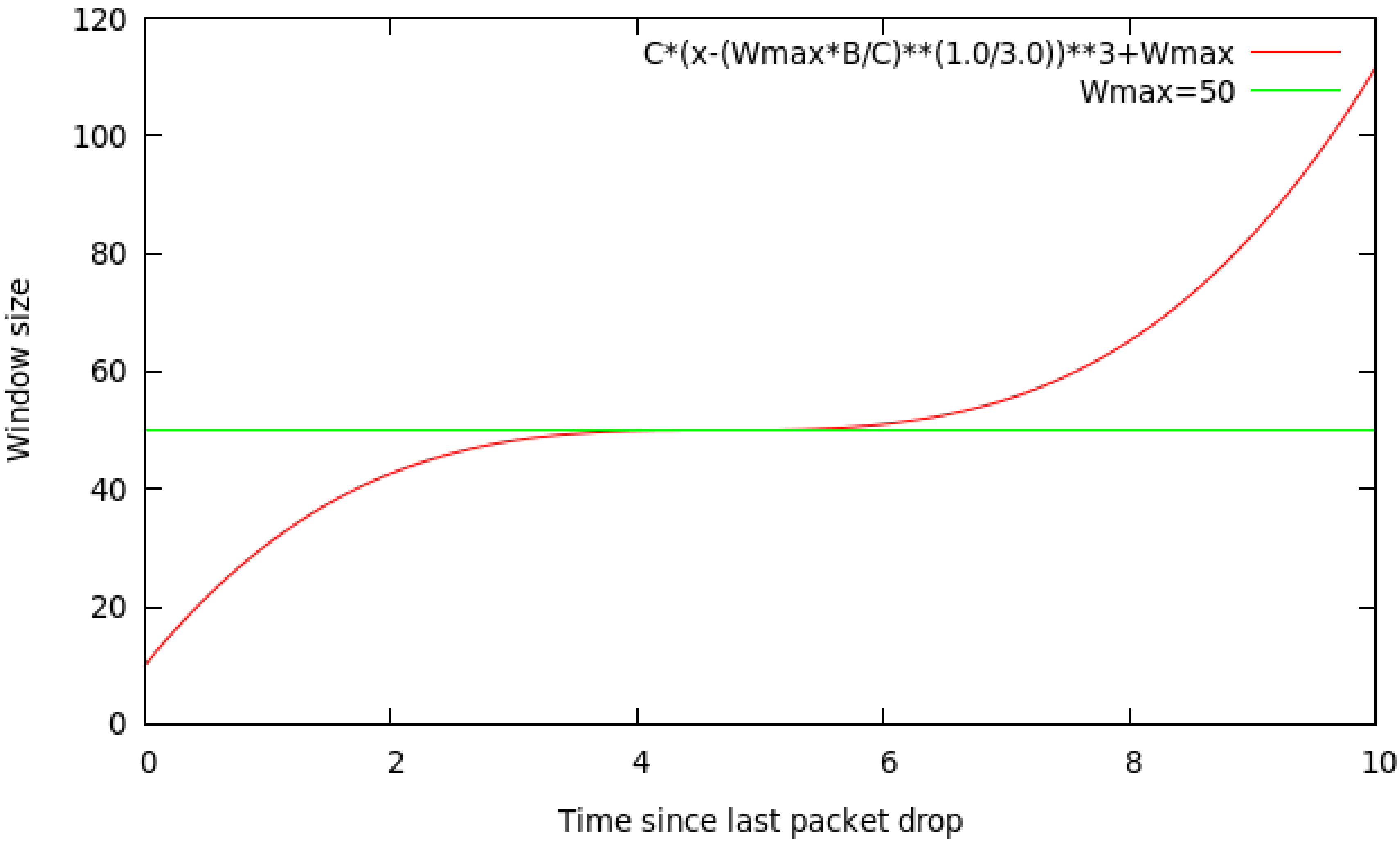
$$W_{cubic} = C(T - K)^3 + W_{max} \qquad (1)$$

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

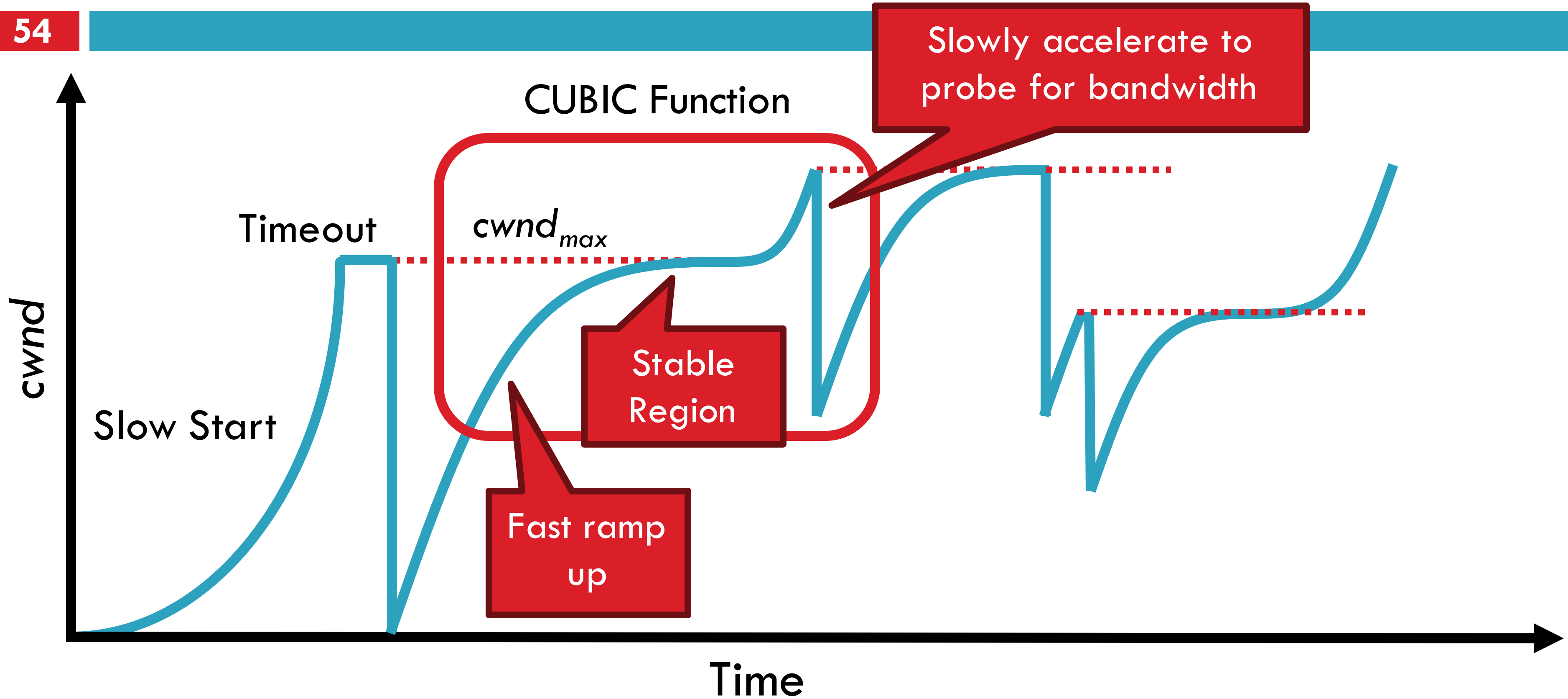
  - B → a constant fraction for multiplicative increase
  - T → time since last packet drop
  - W_max → cwnd when last packet dropped

TCP CUBIC Implementation

-
-



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

Outline

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

Issues with TCP

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

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)

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 (sequence number field)
  - Client will reflect the state back to the server

Further topics

Typical Internet Queuing

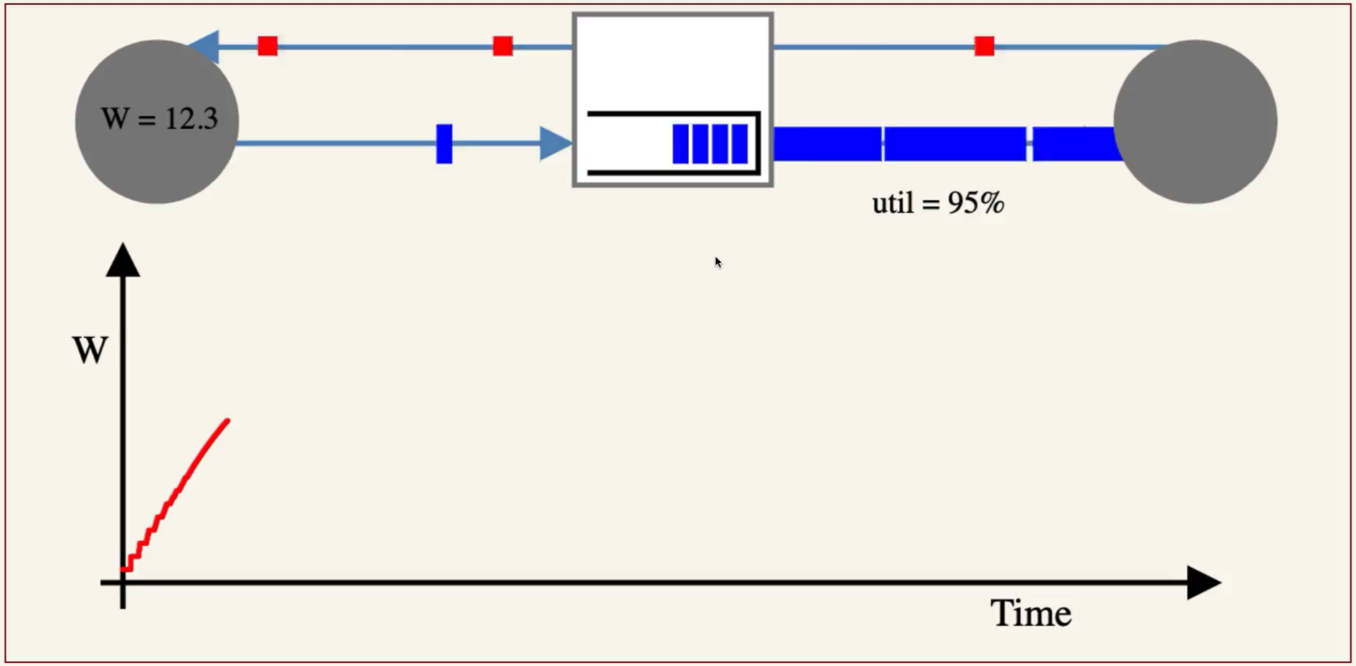
- FIFO + drop-tail
  - Simplest choice
  - Used widely in the Internet
- FIFO (first-in-first-out)
  - Implies single class of traffic
- Drop-tail
  - Arriving packets get dropped when queue is full regardless of flow or importance
- Important distinction:
  - FIFO: scheduling discipline
  - Drop-tail: drop policy

Buffer sizing

- Network is a shared resource
  - Many flows using the same bottleneck
- Temporal overloads should be
  - Buffers are needed
- Buffers are needed for good

- Drawbacks of large buffers
  - Increased end-to-end delay

Network Animations Home Animations GitHub



About This Animation

Controls: Left Arrow = Slow, Down Arrow = Medium, Right Arrow = Fast

1988 1994 2019

Congestion Avoidance and Control
VJ & MK

High Performance TCP in ANSNET
CV & CS


Congestion Avoidance and Control
Van Jacobson
Lawrence Berkeley Laboratory
Michael J. Karels
University of California at Berkeley
November, 1988

Introduction


High Performance TCP in ANSNET
Curtis Villamizar
Advanced Network & Services, Inc.
Cheng Song
Advantis
September 12, 1994

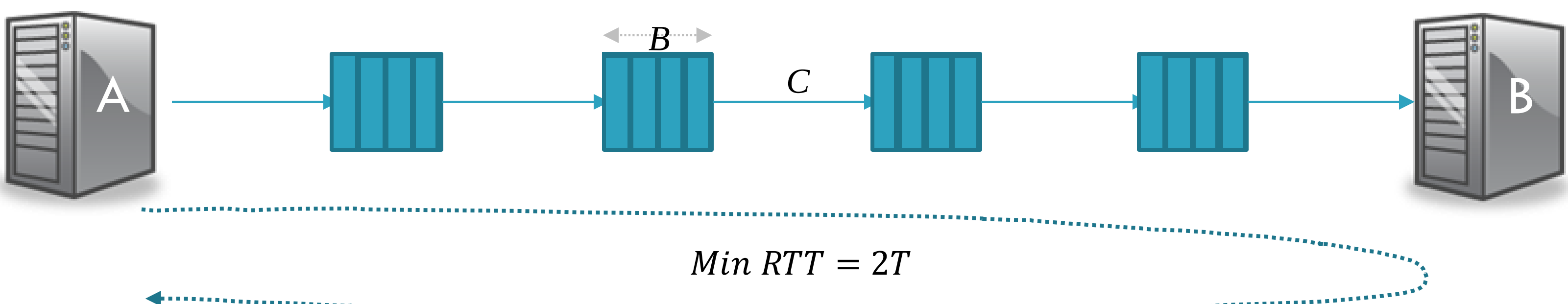
Abstract

$$B = 2T \times C$$

"Buffer size should equal the bandwidth delay product"

$$\cancel{Max\ RTT = 2T + B/C = 4T}$$

$$Max\ RTT = 2T + 4B/C = 10T$$



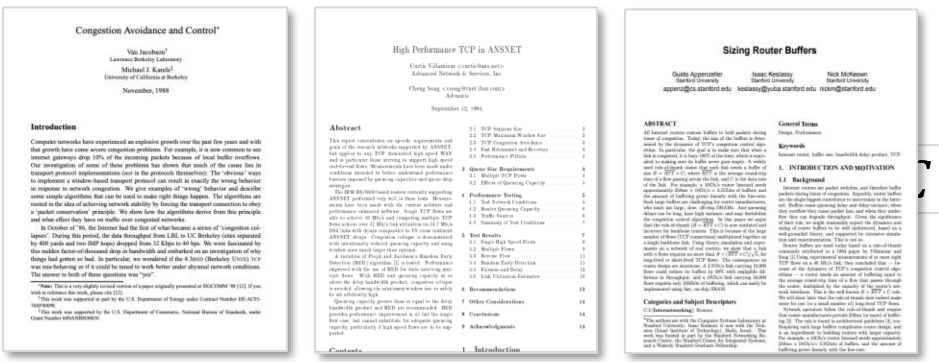
Slide from Nick McKeown's presentation at Stanford "pre-workshop" meeting on Buffer Sizing.

1988 | 1994 | 2004 | 2020

Congestion Avoidance and Control
VJ & MK

High Performance TCP in ANSNET
CV & CS

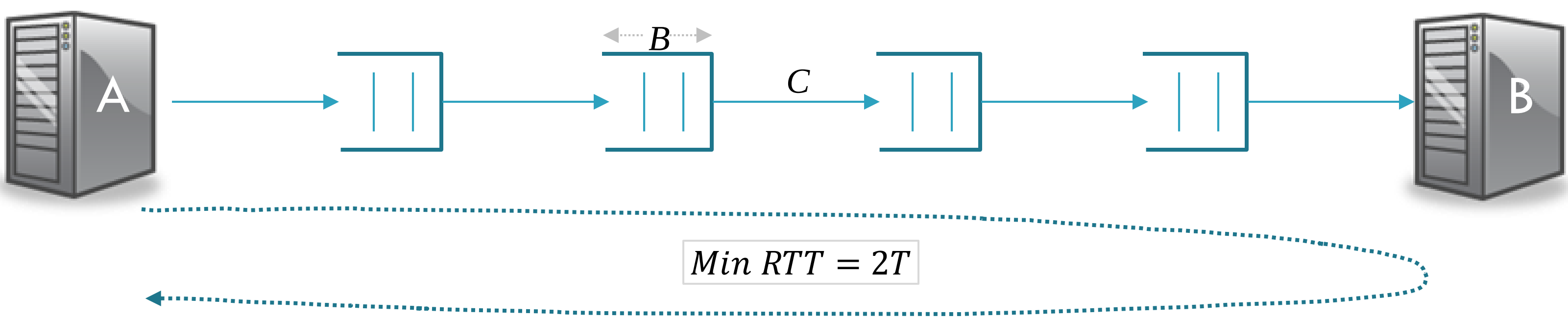
Sizing Router Buffers
GA, IK, NM


Congestion Avoidance and Control*

High Performance TCP in ANSNET

Sizing Router Buffers

$$B \geq \frac{2T \times C}{\sqrt{N}}$$ where N is the number of long-lived flows



Slide from Nick McKeown's presentation at Stanford "pre-workshop" meeting on Buffer Sizing.

1988 | 1994 | 2004 | 2006 | 2008 | 2020

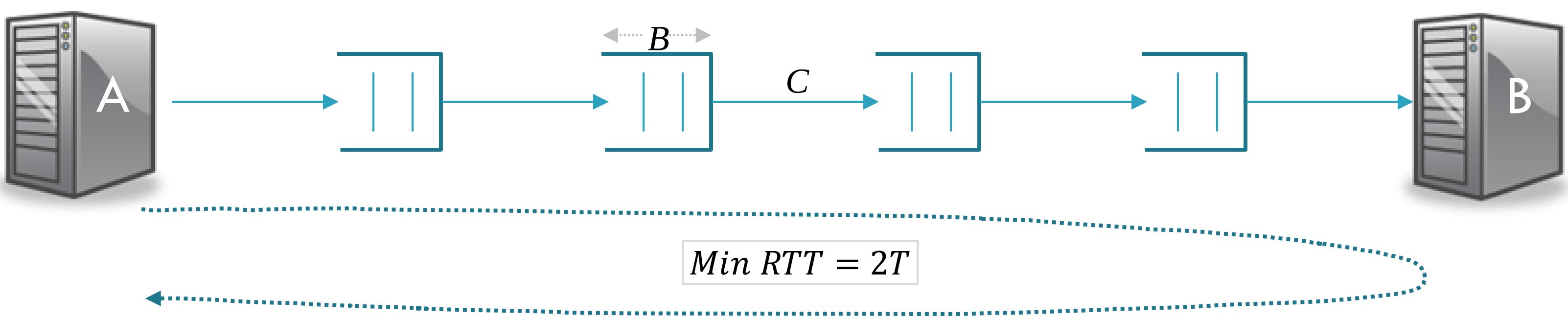
Congestion Avoidance and Control
VJ & MK

High Performance TCP in ANSNET
CV & CS

Sizing Router Buffers
GA, IK, NM

Routers with Very Small Buffers
ME, YG, AG, NM, TR

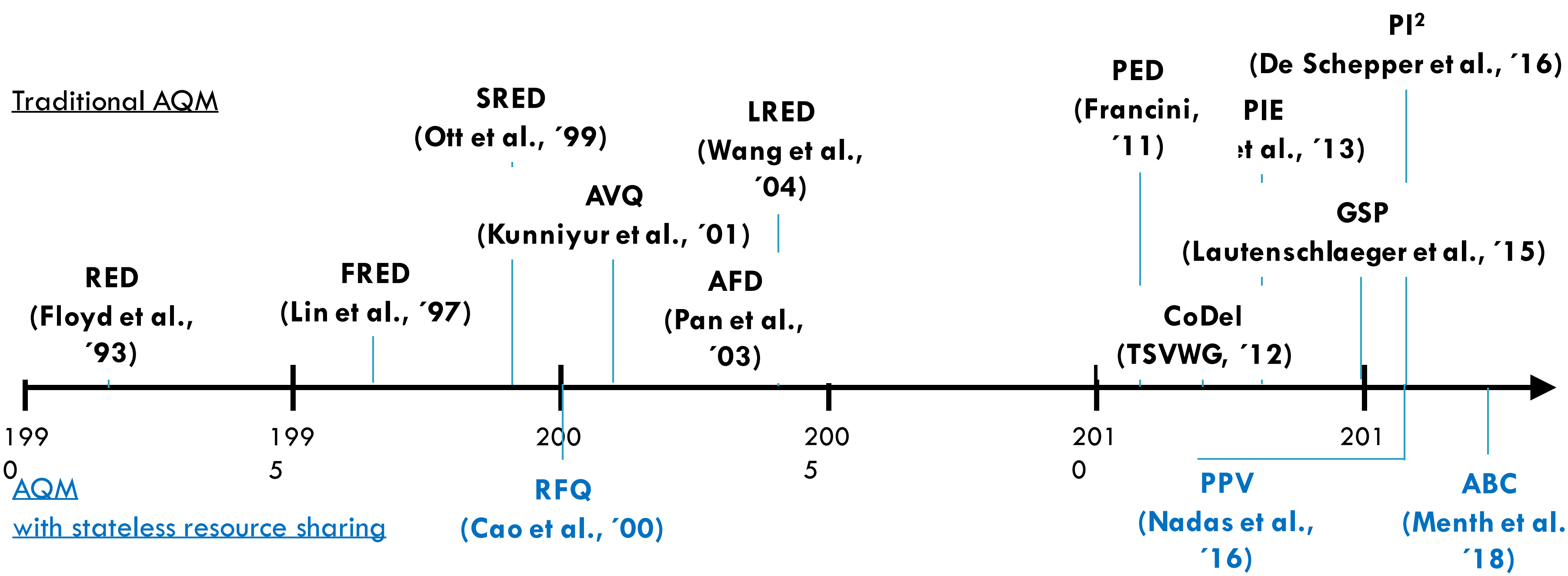
Experimental Study of Router Buffers
NB, YG, MG, NM, GS



Slide from Nick McKeown’s presentation at Stanford "pre-workshop" meeting on Buffer Sizing.

2010s – reducing queuing delay

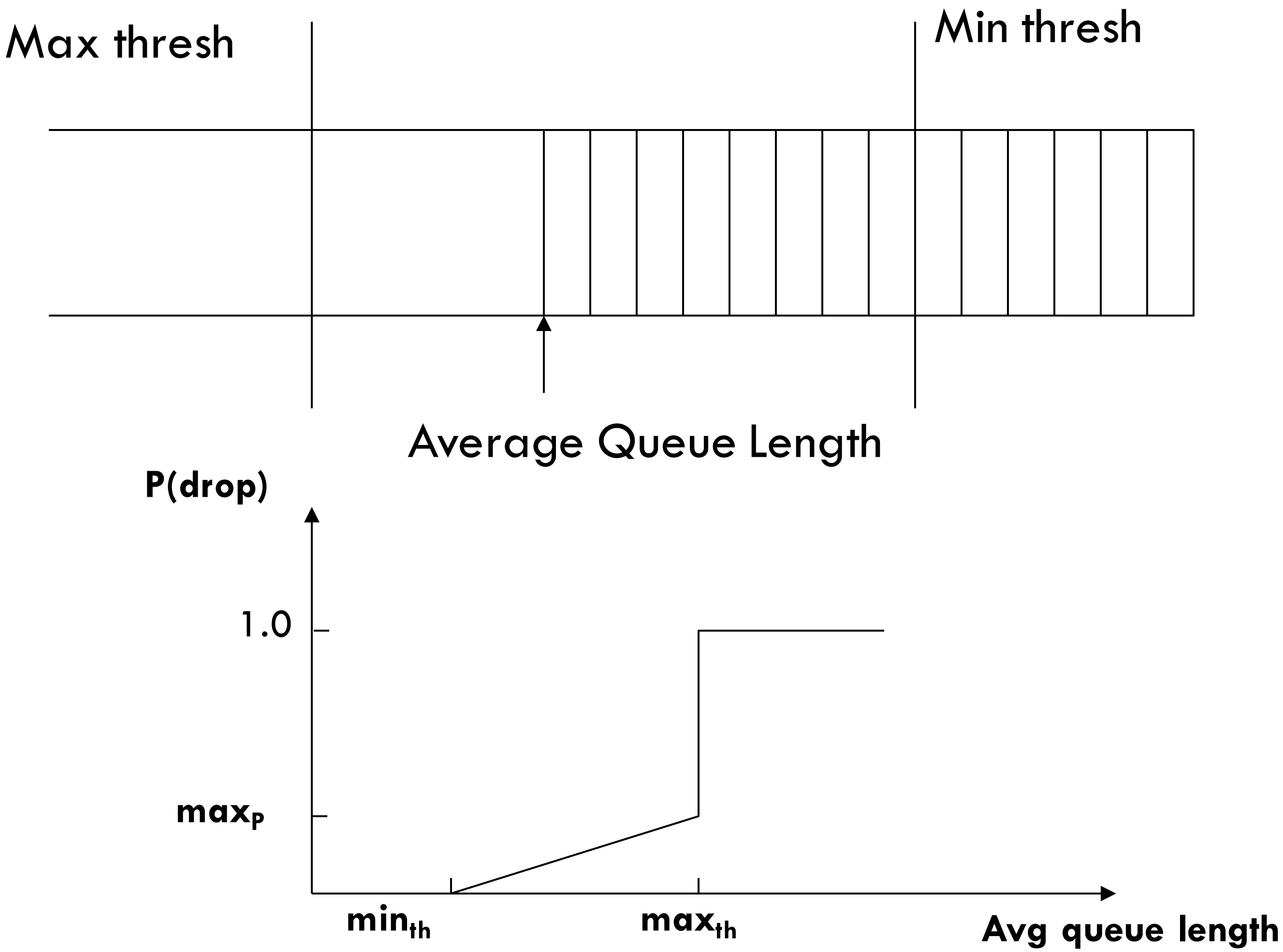
- Active Queue Management (AQM)
  - Goal is to reduce the average queuing delay, but allow temporal overshoots
  - Proactively starts dropping or marking packets to reduce queuing delay



RED Algorithm

- Maintain running average of queue length
- If avgq < min_{th} do nothing
  - Low queuing, send packets through
- If avgq > max_{th}, drop packet
  - Protection from misbehaving sources
- Else mark packet in a manner proportional to queue length
  - Notify sources of incipient congestion
  - E.g. by ECN IP field or dropping packets with a given probability

RED Operation



RED Algorithm

- Maintain running average of queue length
- For each packet arrival
  - Calculate average queue size (avg)
  - If $min_{th} \leq avgq < max_{th}$
    - Calculate probability P_a
    - With probability P_a
      - Mark the arriving packet: drop or set-up ECN
    - Else if $max_{th} \leq avg$
      - Mark the arriving packet: drop, ECN

Data Center TCP: DCTCP

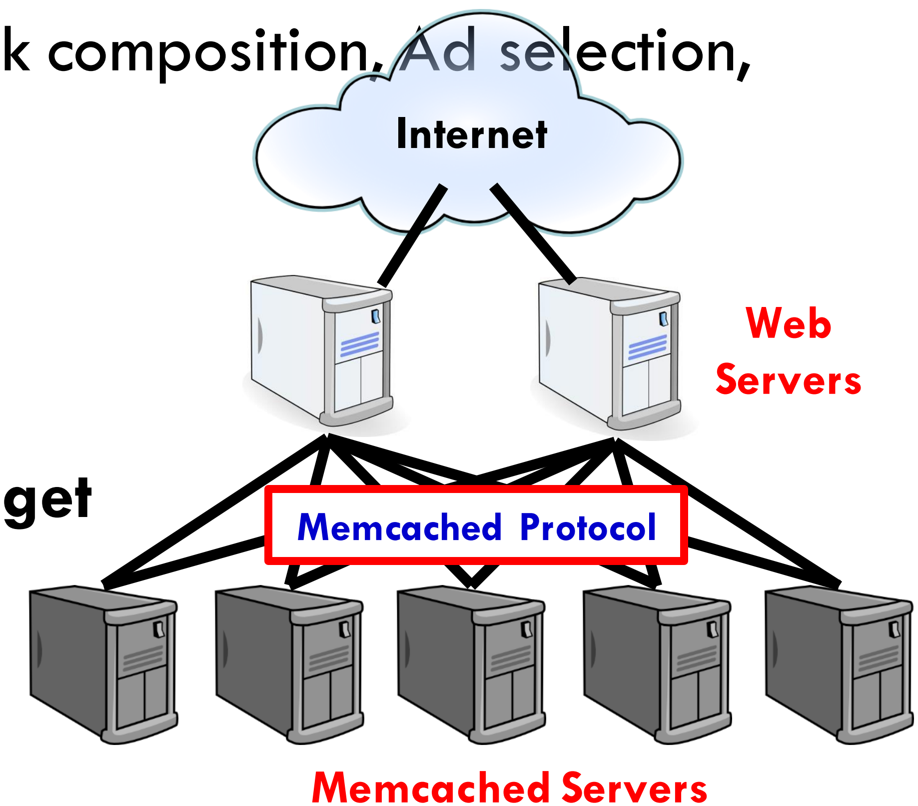
Generality of Partition/Aggregate

- The foundation for many large-scale web applications.
  - Web search, Social network composition, Ad selection, etc.

- Example: **Facebook**

Partition/Aggregate ~ Multiget

- Aggregators: **Web Servers**
- Workers: **Memcached Servers**



Workloads

- Partition/Aggregate

  (Query)

- Short messages [50KB-1MB]

  (Coordination, Control state)

- Large flows [1MB-50MB]

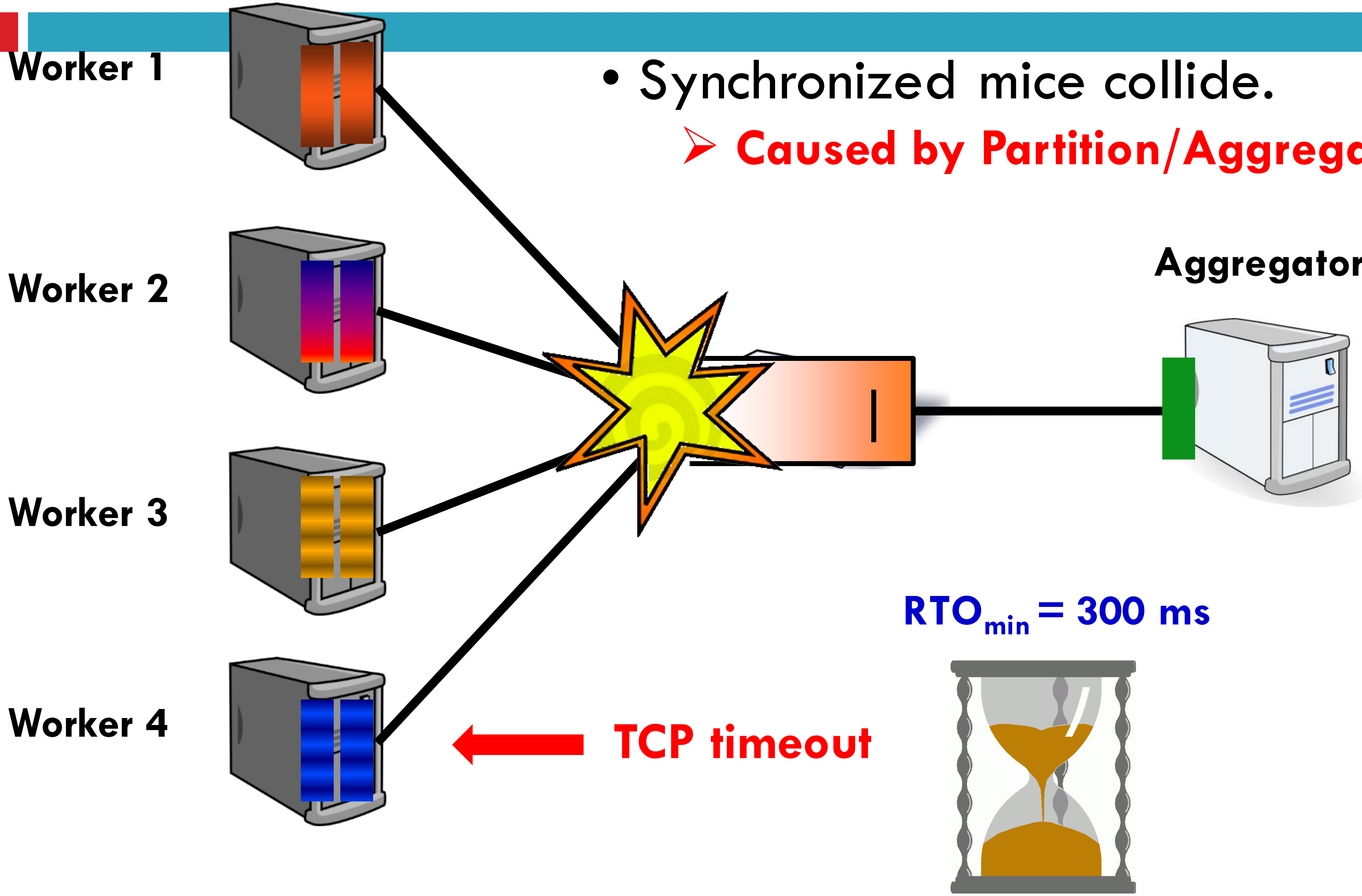
  (Data update)

Impairments

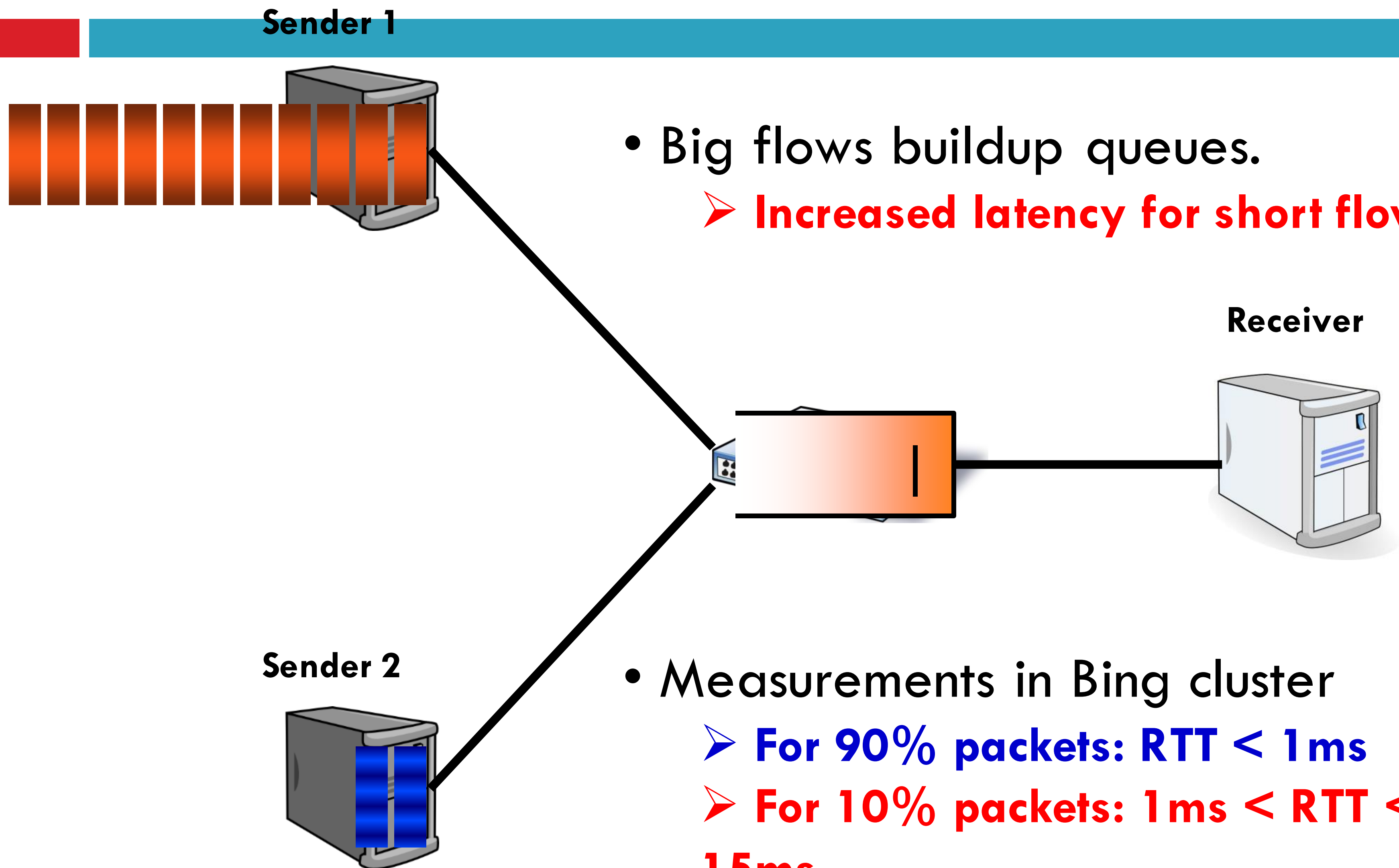
- Incast
- Queue Buildup
- Buffer Pressure

Incast

- Synchronized mice collide.
  - **Caused by Partition/Aggregate.**



Queue Buildup



- Big flows buildup queues.
  - ➢ **Increased latency for short flows.**
- Measurements in Bing cluster
  - ➢ **For 90% packets: RTT < 1ms**
  - ➢ **For 10% packets: 1ms < RTT < 15ms**

Data Center Transport Requirements

1. **High Burst Tolerance**
   – Incast due to Partition/Aggregate is common.

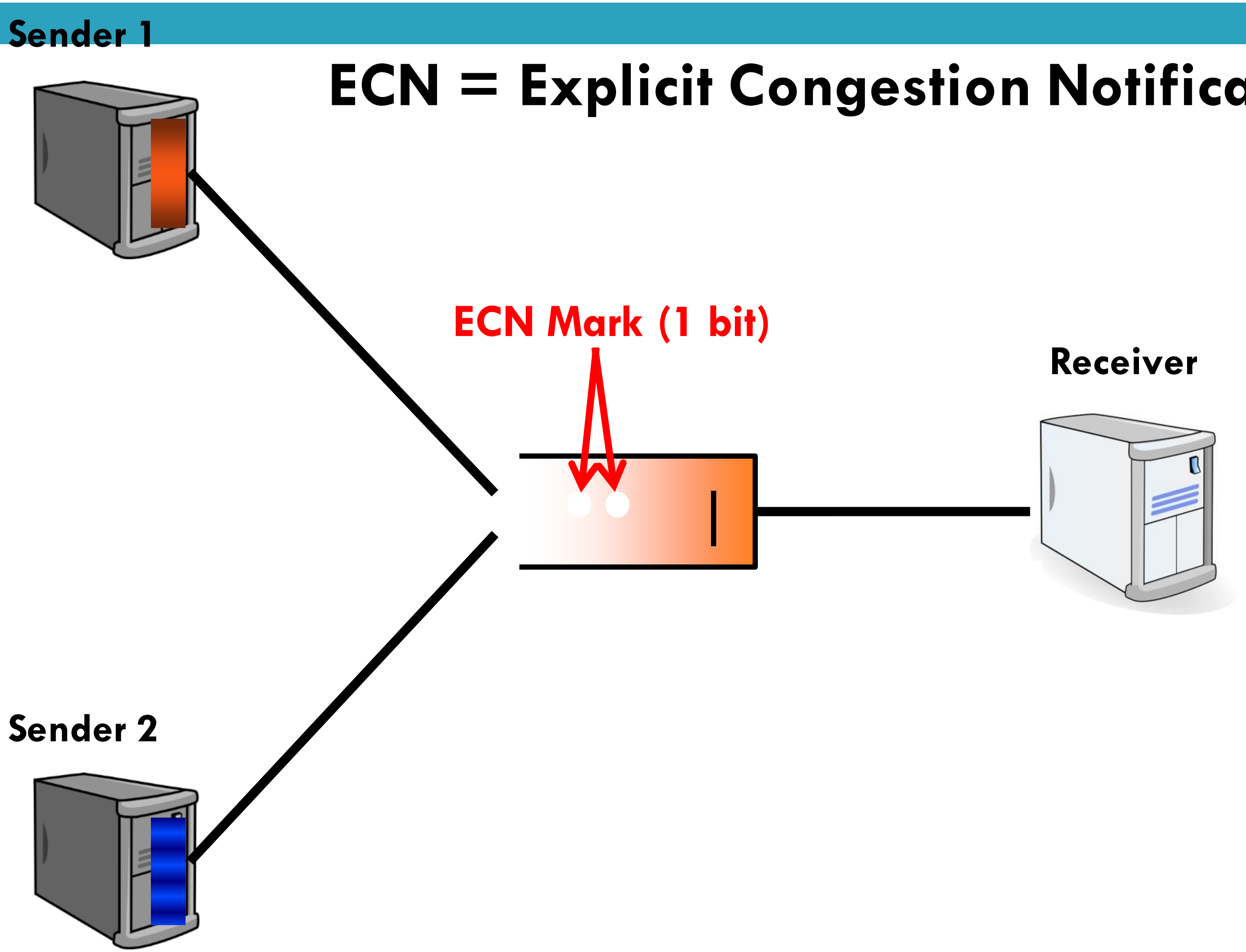
2. **Low Latency**
   – Short flows, queries

3. **High Throughput**
   – Continuous data updates, large file transfers

The challenge is to achieve these three together.

DCTCP: The TCP/ECN Control Loop

ECN = Explicit Congestion Notification



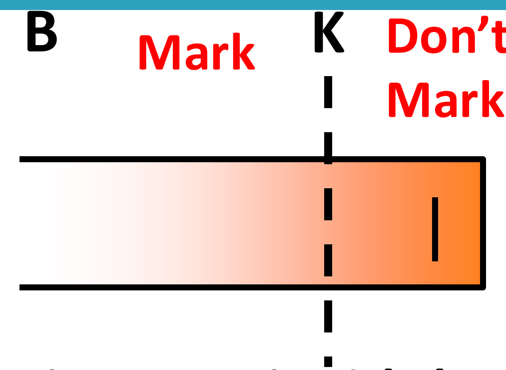
DCTCP: Two Key Ideas

1. React in proportion to the **extent** of congestion, not its **presence**.
   - ✓ Reduces **variance** in sending rates, lowering queuing requirements.

ECN Marks	TCP	DCTCP
1 0 1 1 1 1 0 1 1 1	Cut window by 50%	Cut window by 40%
0 0 0 0 0 0 0 0 0 1	Cut window by 50%	Cut window by 5%

2. Mark based on **instantaneous** queue length.
   - ✓ Fast feedback to better deal with bursts.

Data Center TCP Algorithm



Switch side:

- Mark packets when **Queue Length > K.**

Sender side:

– Maintain running average of ***fraction*** of packets marked **(α)**.

In each RTT:

$$F = \frac{\#\ of\ marked\ ACKs}{Total\ \#\ of\ ACKs} \qquad \alpha \leftarrow (1-g)\alpha + gF$$

➢ **Adaptive window decreases:** $Cwnd \leftarrow (1-\frac{\alpha}{2})Cwnd$

– Note: decrease factor between 1 and 2.