

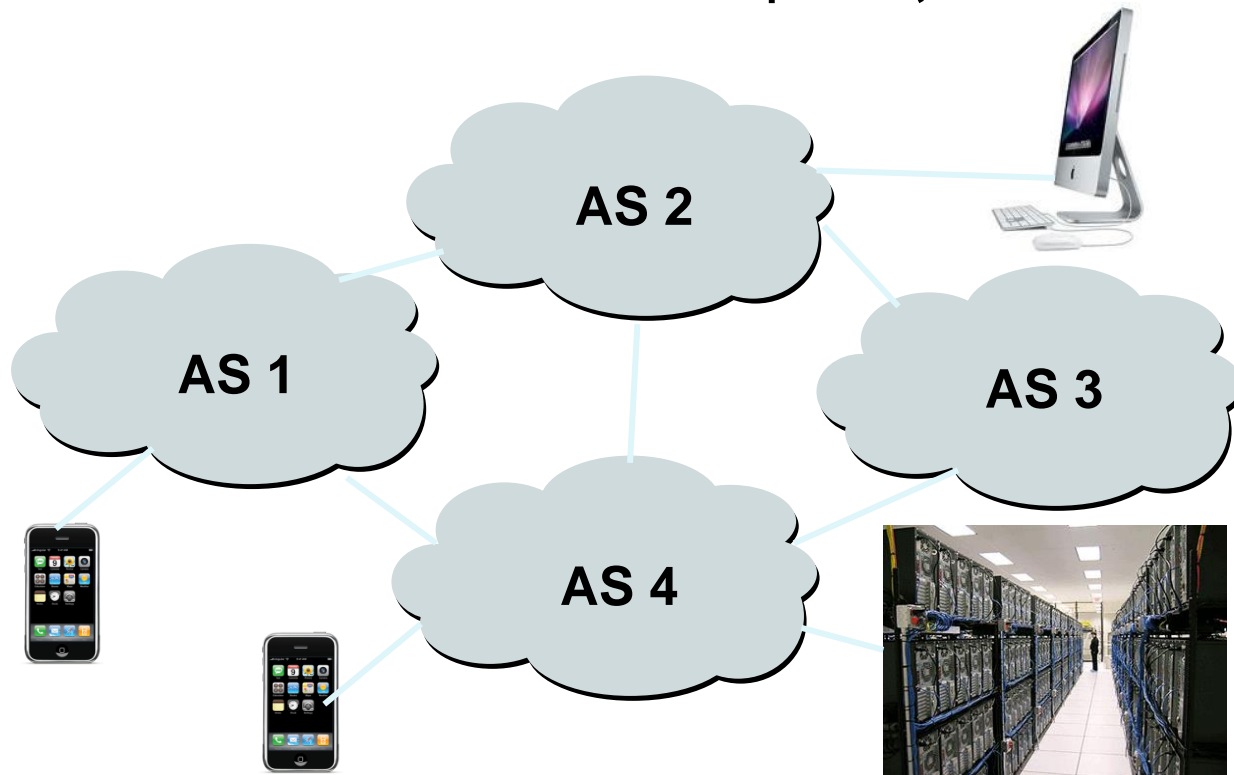
# TDTS21 Advanced Networking

## **Lecture 3: Transport, including TCP and congestion control ...**

Based on slides from D. Choffnes, P. Gill, and S. Katti  
Revised Spring 2015 by N. Carlsson

# Holding the Internet Together

- Distributed cooperation for resource allocation
  - ▣ BGP: what end-to-end *paths* to take (for ~50K ASes)
  - ▣ TCP: what *rate* to send over each path (for ~3B hosts)



# What Problem Does a Protocol Solve?

- BGP path selection
  - ▣ Select a path that each AS on the path is willing to use
  - ▣ Adapt path selection in the presence of failures
- TCP congestion control
  - ▣ Prevent congestion collapse of the Internet
  - ▣ Allocate bandwidth fairly and efficiently

# What Problem Does a Protocol Solve?

- BGP path selection

- Select a path that each AS on the path is willing to use
- Adapt path selection in the presence of failures

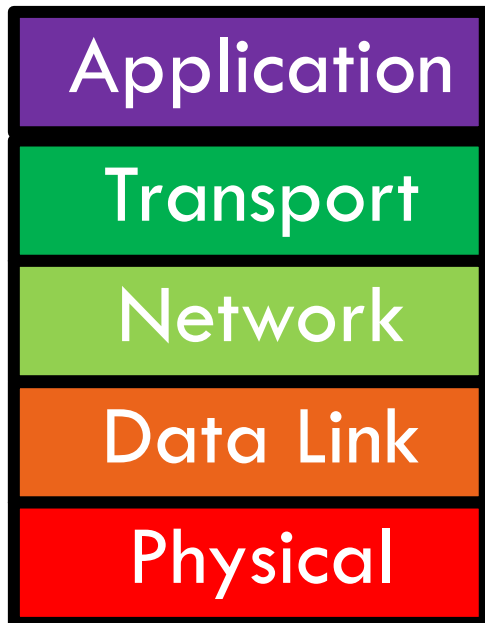
- TCP congestion control

- Prevent congestion collapse of the Internet
- Allocate bandwidth fairly and efficiently

Today, we will focus on TCP (and UDP) ...

# Transport Layer

5



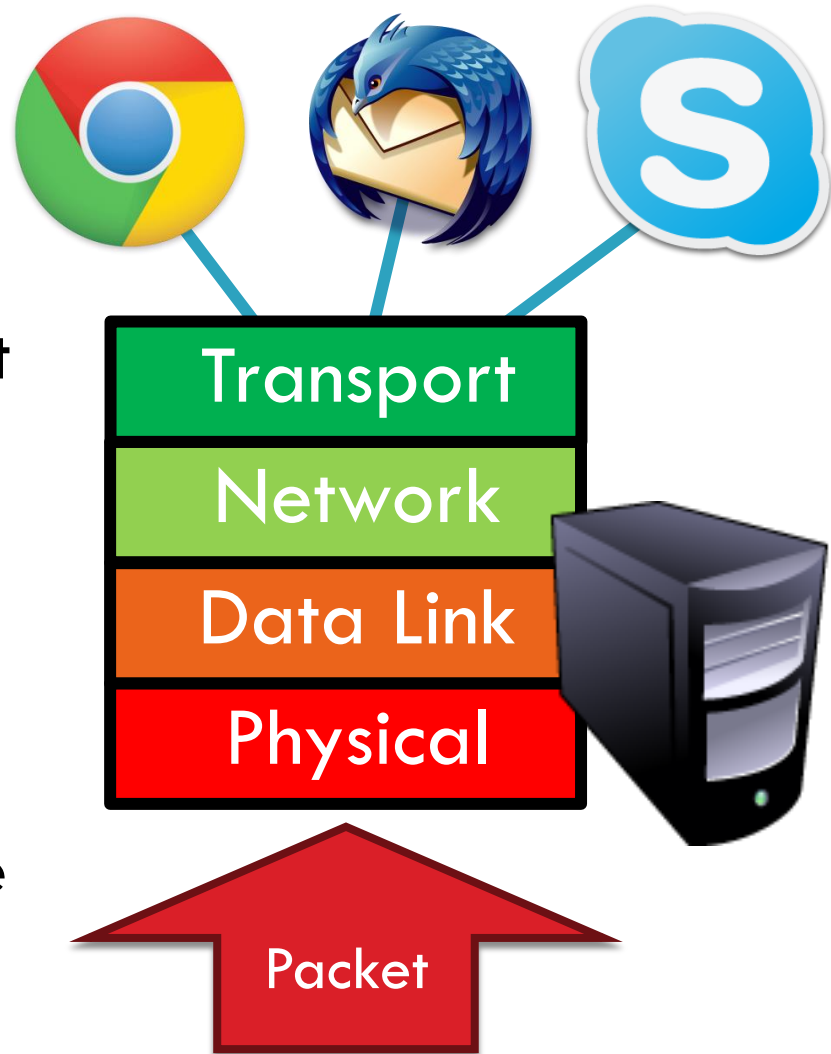
- Function:
  - ▣ Demultiplexing of data streams
- Optional functions:
  - ▣ Creating long lived connections
  - ▣ Reliable, in-order packet delivery
  - ▣ Error detection
  - ▣ Flow and congestion control
- Key challenges:
  - ▣ Detecting and responding to congestion
  - ▣ Balancing fairness against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

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

# The Case for Multiplexing

7

- 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

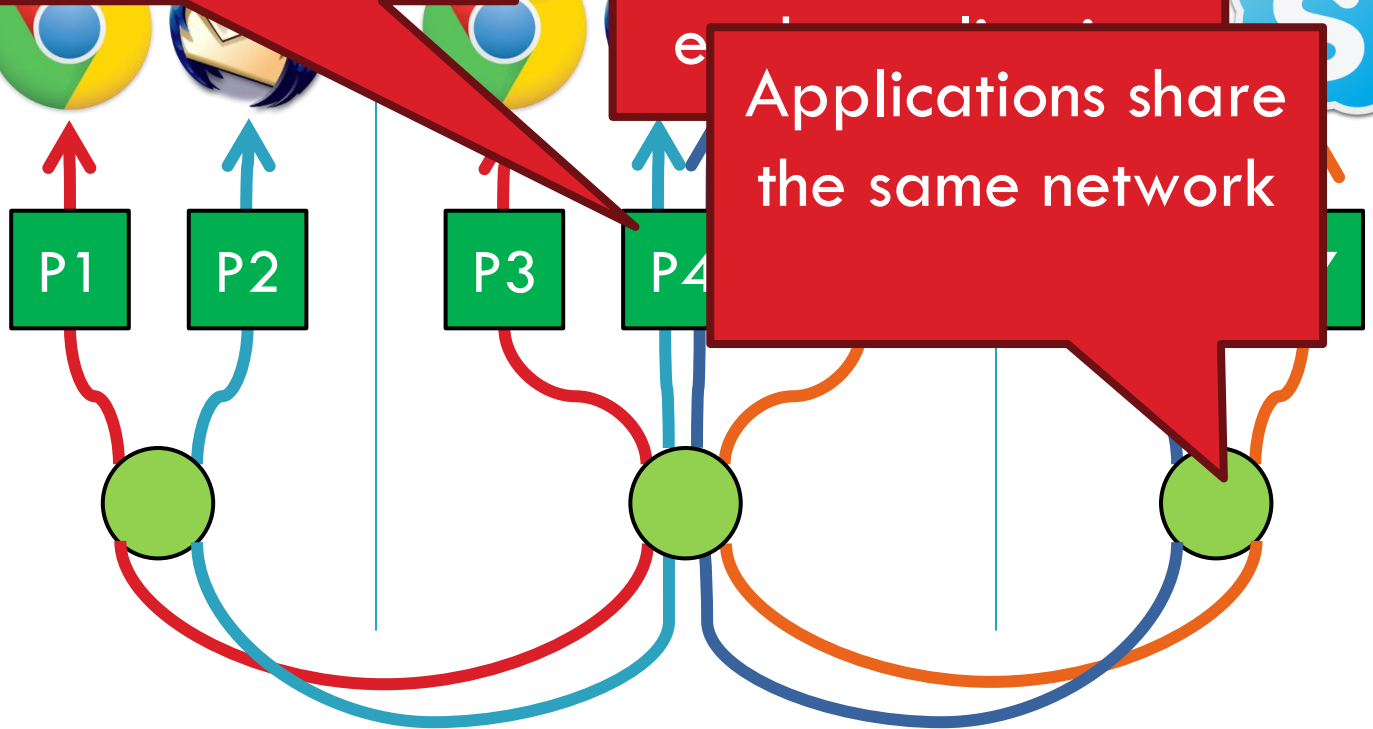
8

Server applications communicate with multiple clients

Application

Transport

Network



3

Unique port for each application

Applications share the same network

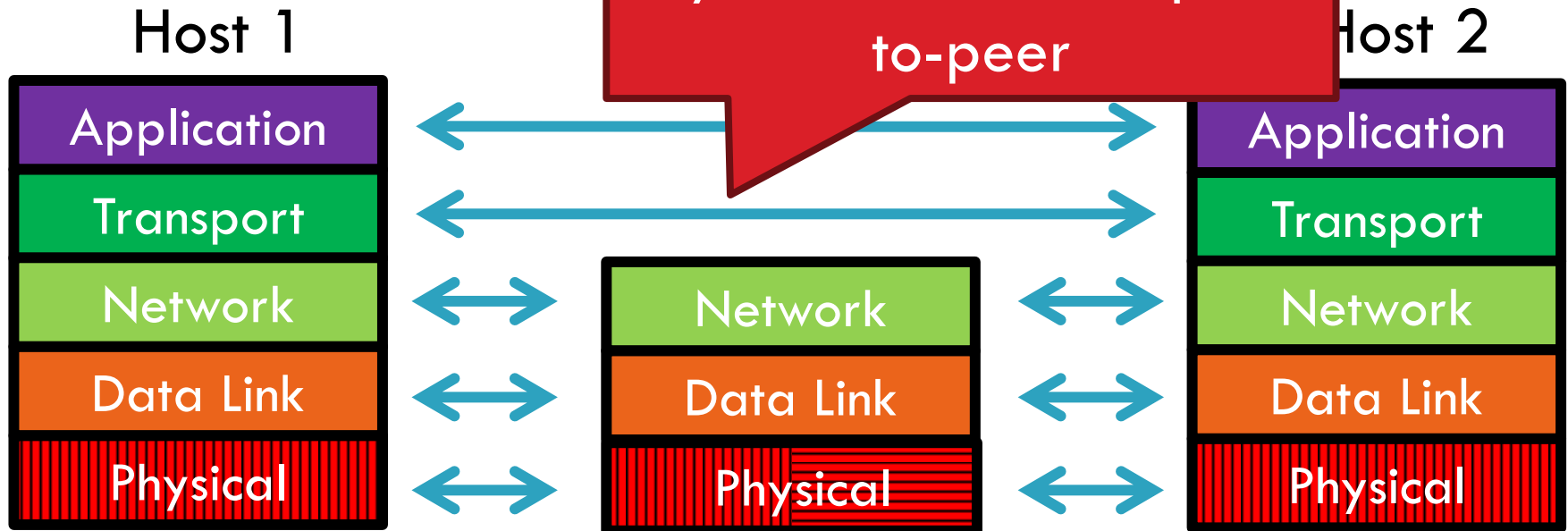
Endpoints identified by  $\langle src\_ip, src\_port, dest\_ip, dest\_port \rangle$



# Layering, Revisited

9

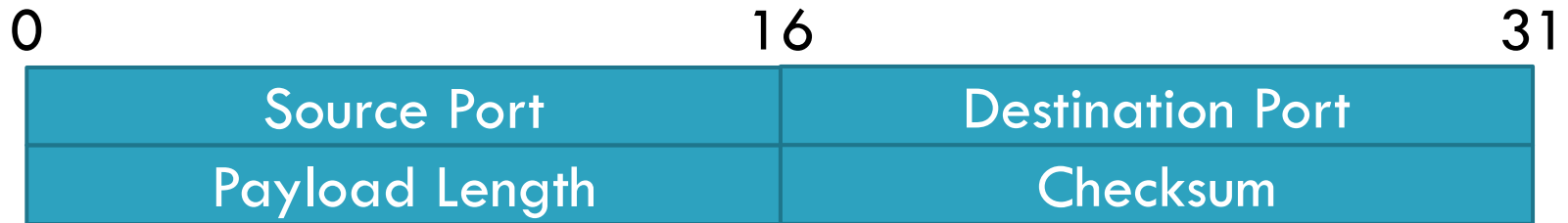
Layers communicate peer-to-peer



- Lowest level end-to-end protocol
  - ▣ Transport header only read by source and destination
  - ▣ Routers view transport header as payload

# User Datagram Protocol (UDP)

10



- ❑ Simple, connectionless datagram
- ❑ 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

11

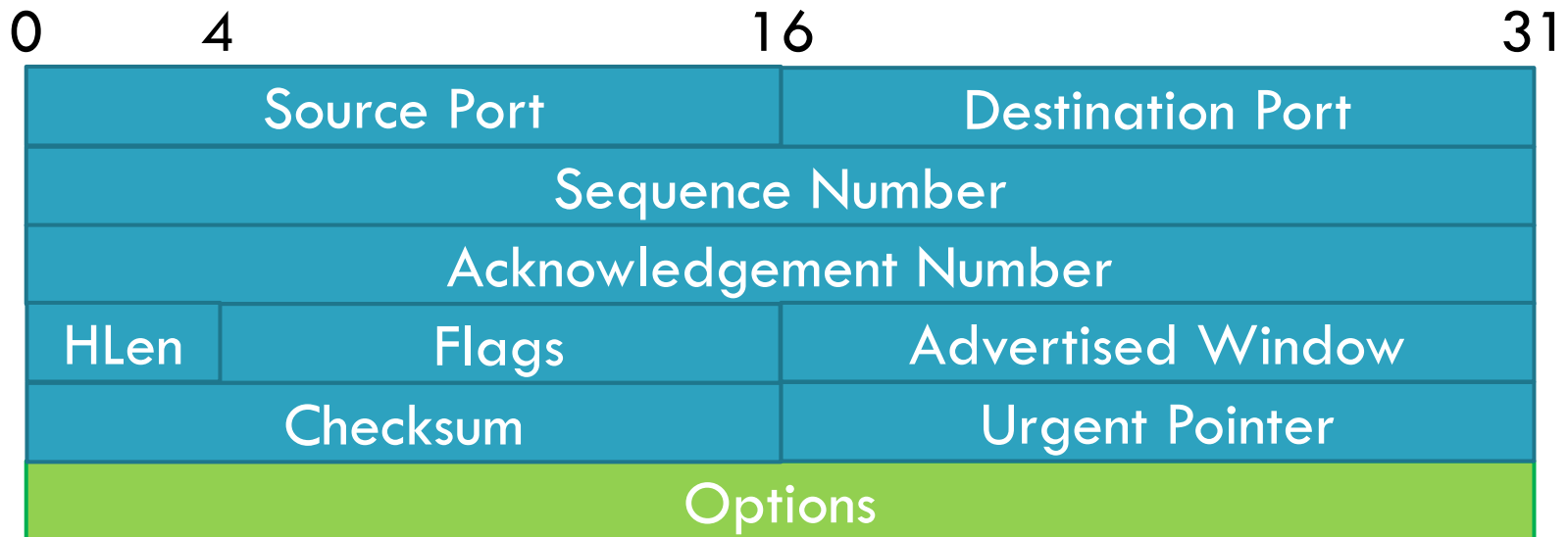
- Invented after TCP
  - ▣ Why?
- Not all applications can tolerate TCP
- Custom protocols can be built on top of UDP
  - ▣ Reliability? Strict ordering?
  - ▣ Flow control? Congestion control?
- Examples
  - ▣ RTMP, real-time media streaming (e.g. voice, video)
  - ▣ Facebook datacenter protocol

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

# Transmission Control Protocol

13

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



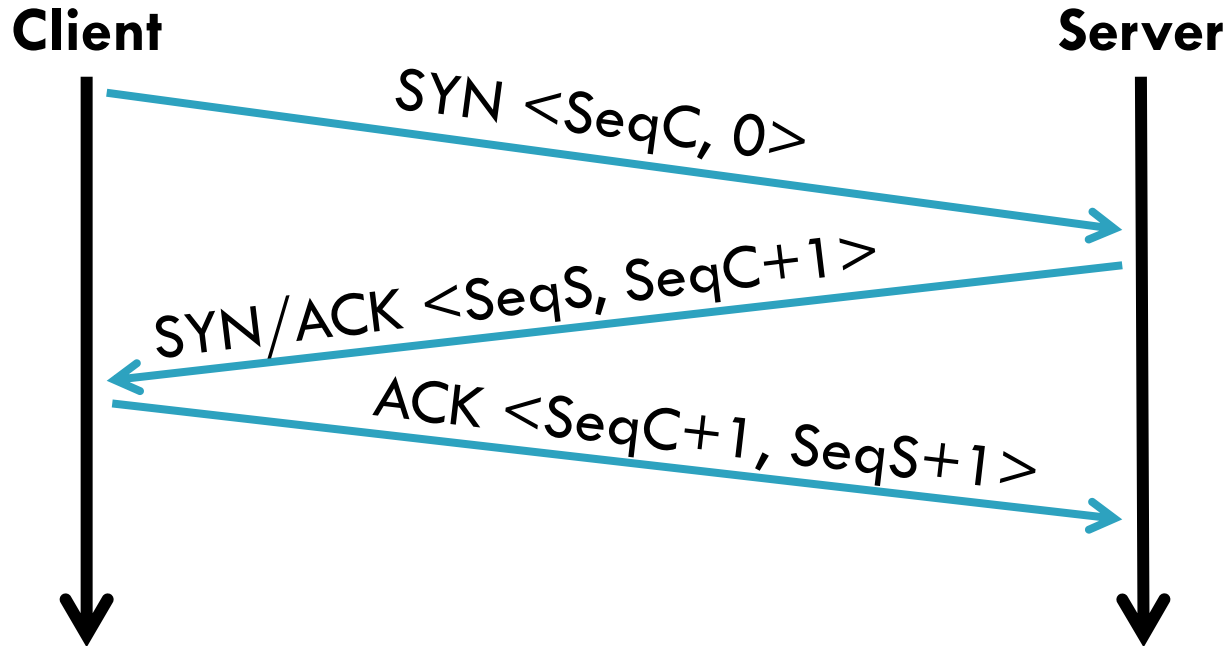
# Connection Setup

14

- 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

15



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

# Connection Setup Issues

16

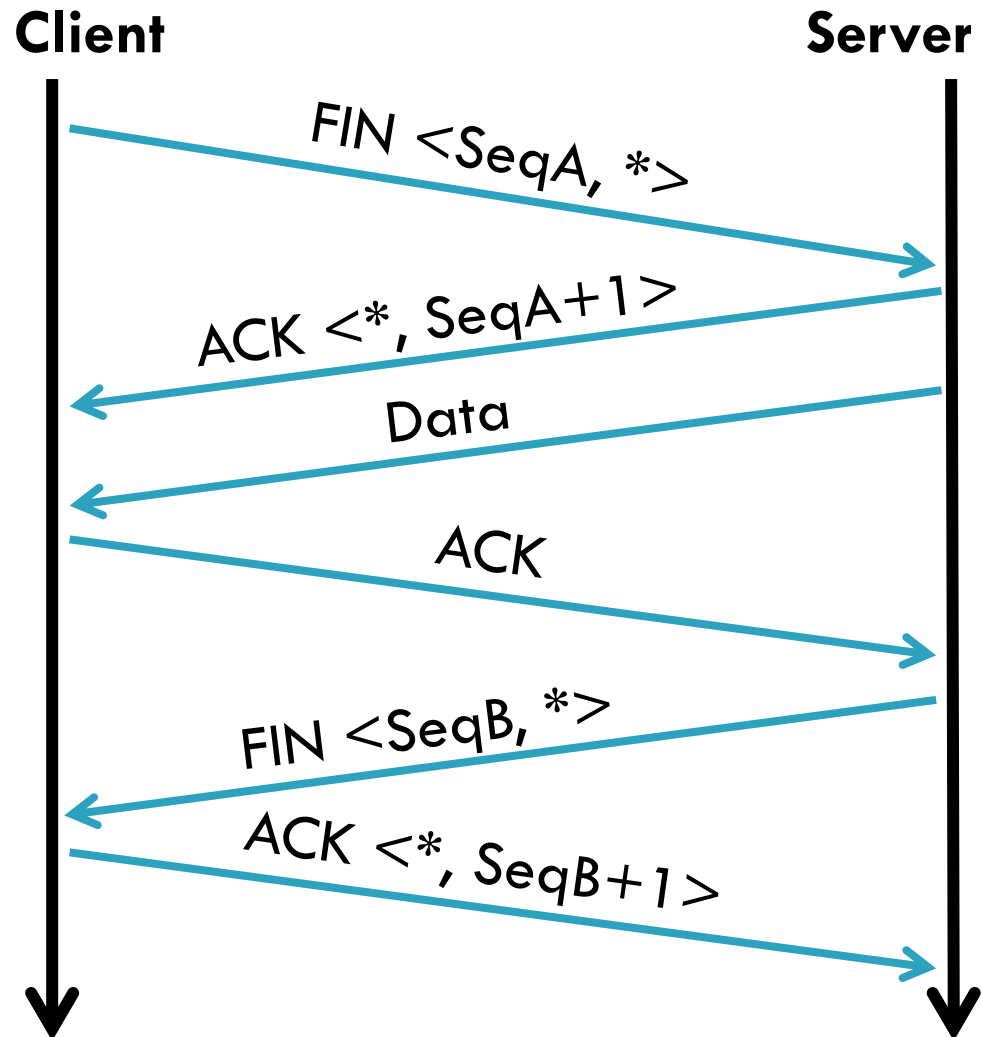
- ❑ Connection confusion
  - ▣ How to disambiguate connections from the same host?
  - ▣ Random sequence numbers
- ❑ Source spoofing
  - ▣ Kevin Mitnick
  - ▣ Need good random number generators!
- ❑ Connection state management
  - ▣ Each SYN allocates state on the server
  - ▣ SYN flood = denial of service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



# Connection Tear Down

17

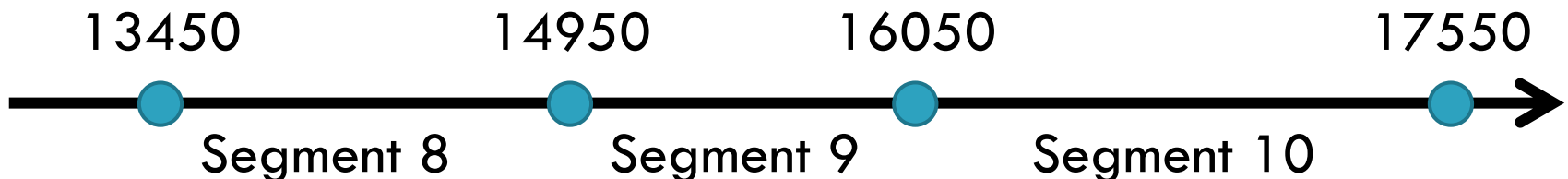
- Either side can initiate tear down
- Other side may continue sending data
  - ▣ Half open connection
  - ▣ *shutdown()*
- Acknowledge the last FIN
  - ▣ Sequence number + 1



# Sequence Number Space

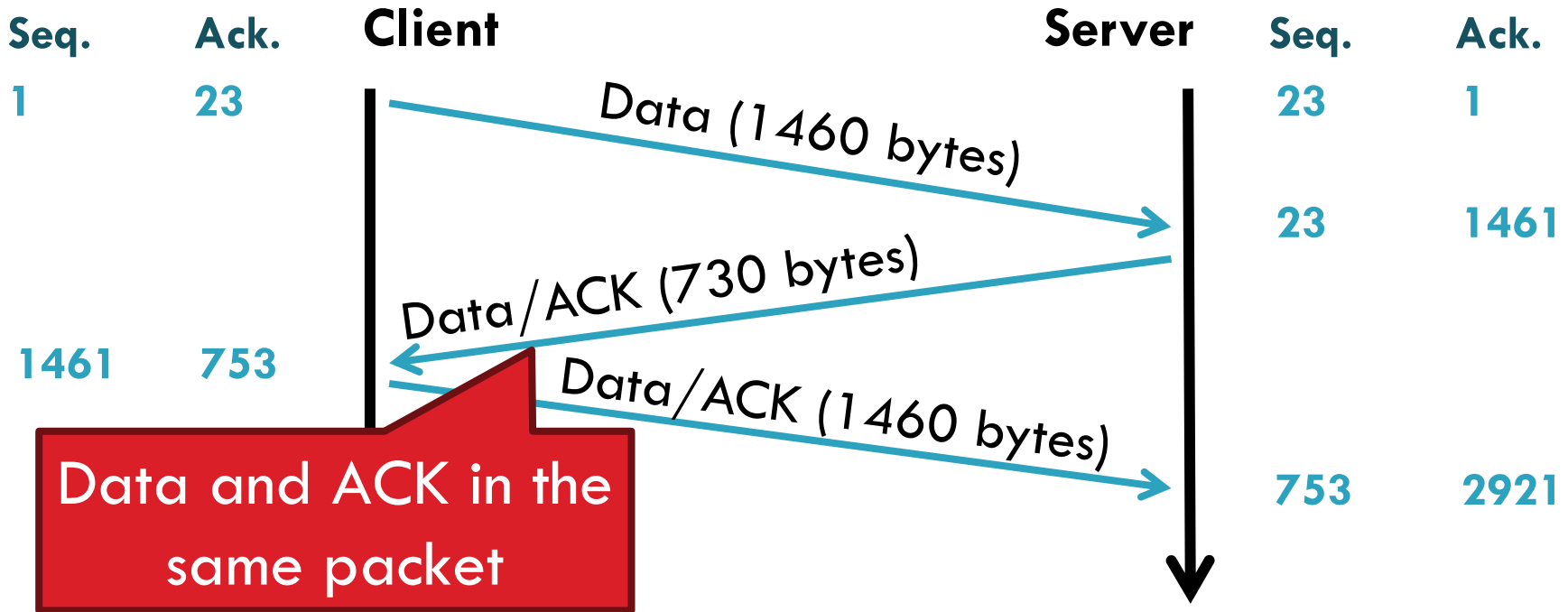
18

- ❑ 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

19



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

# Flow Control

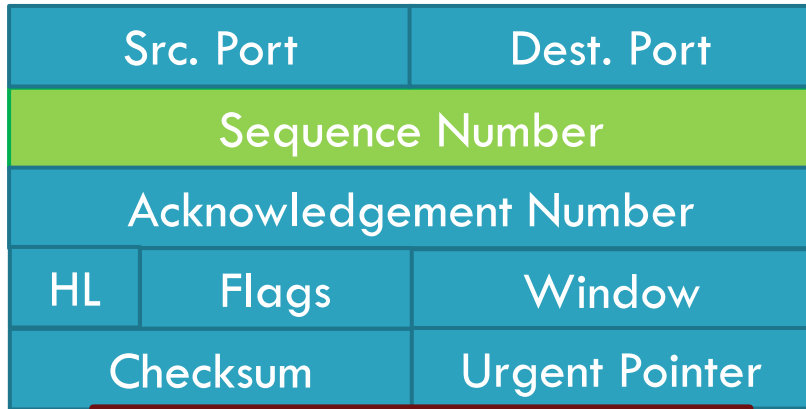
20

- 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

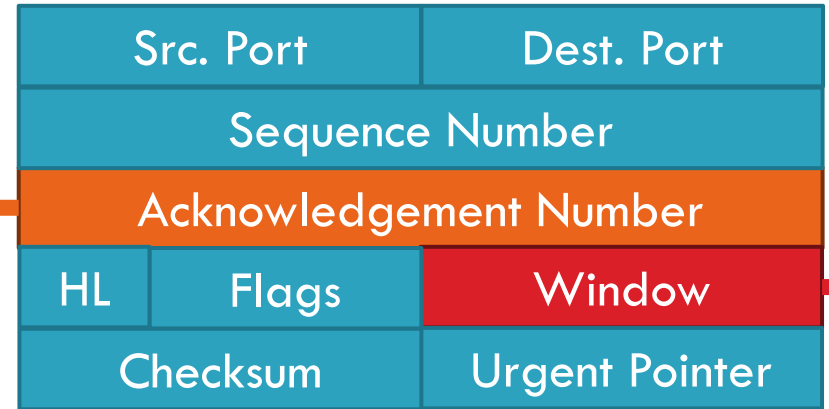
# Flow Control: Sender Side

21

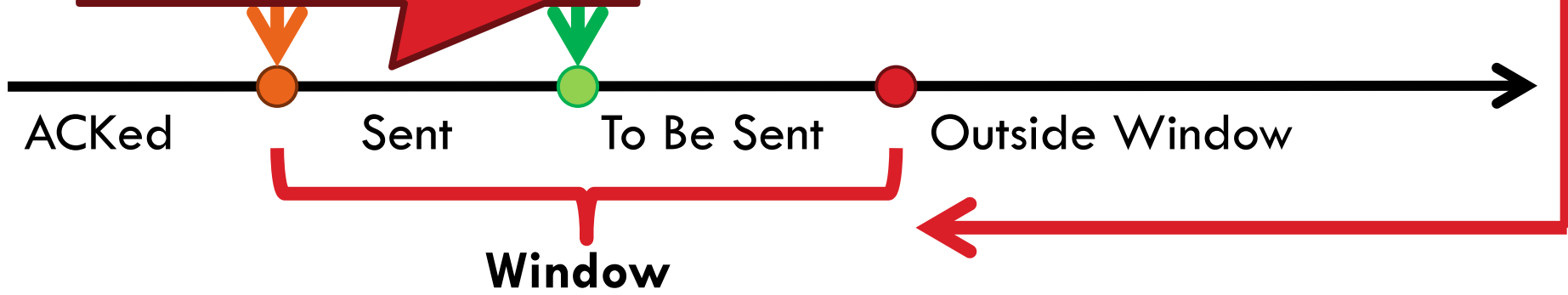
Packet Sent



Packet Received

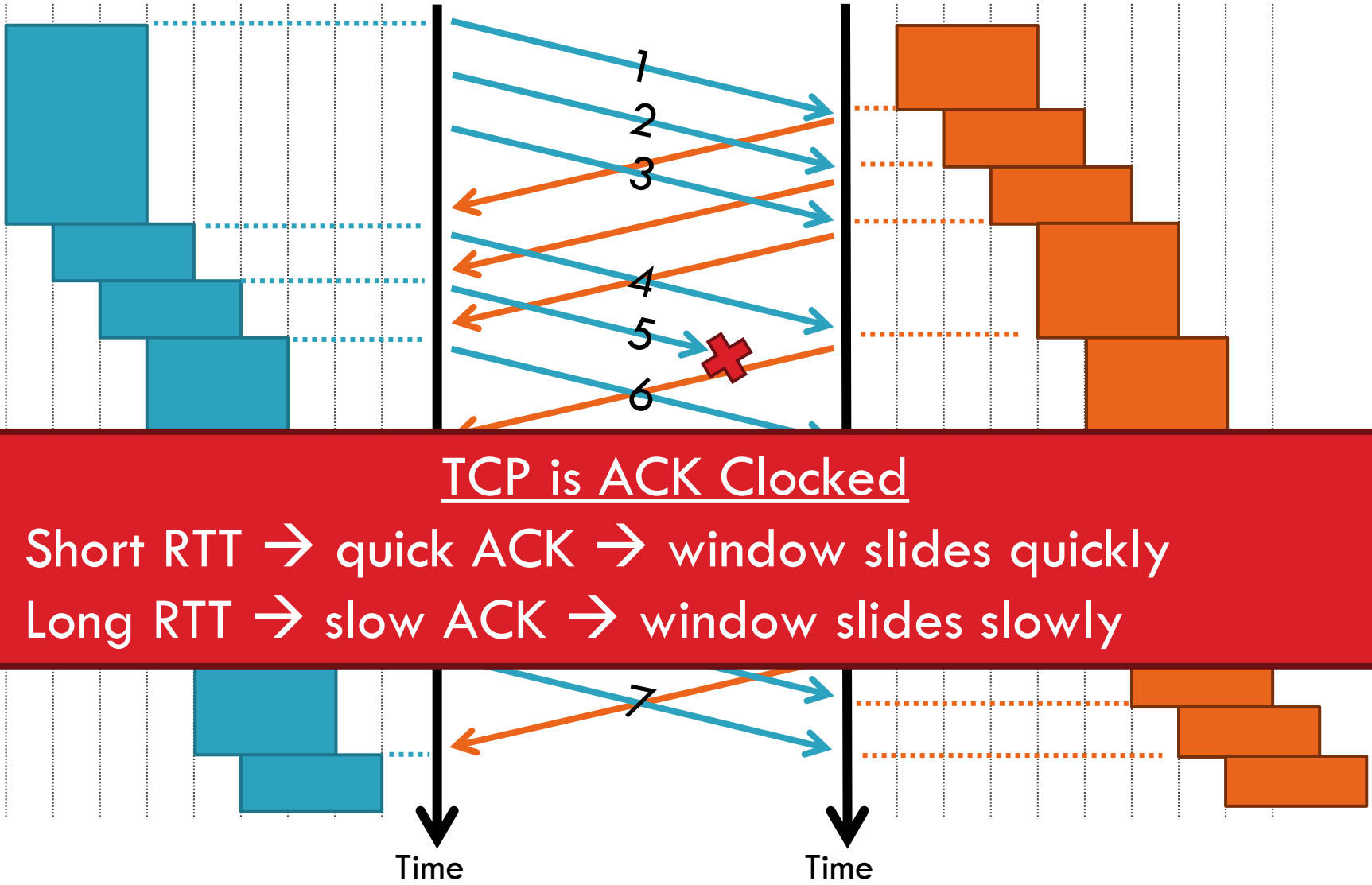


Must be buffered until ACKed



# Sliding Window Example

22



# Observations

23

- Throughput is  $\sim w/\text{RTT}$
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to buffer limits

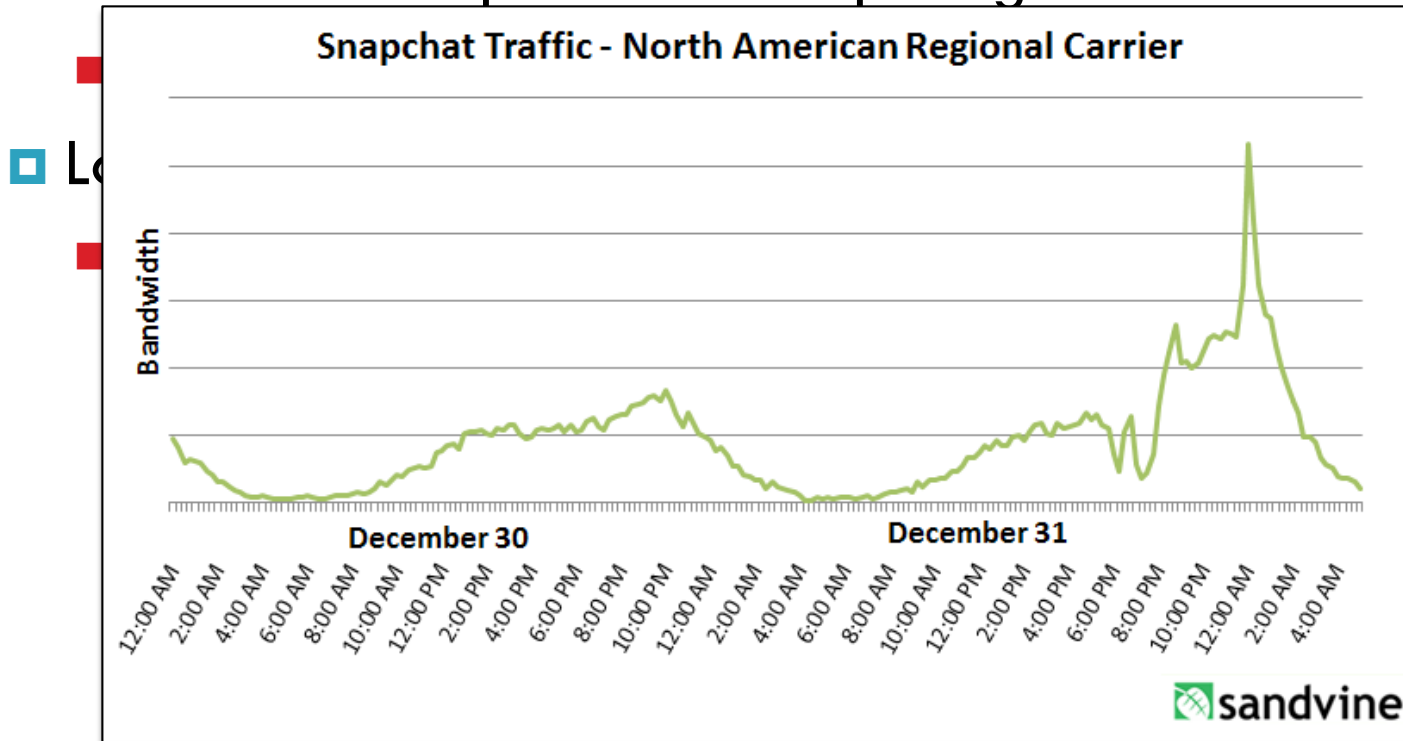
- ❑ UDP
- ❑ TCP
- ❑ Congestion Control
- ❑ Evolution of TCP
- ❑ Problems with TCP



# What is Congestion?

25

- 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



# Why is Congestion Bad?

26

- Results in packet **loss**
  - Routers have finite buffers
  - Internet traffic is bursty, 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”

# CONGESTION AVOIDANCE AND CONTROL

VAN JACOBSON '88

# Main contributions



- ???

# Main contributions



## Seven new algorithms:

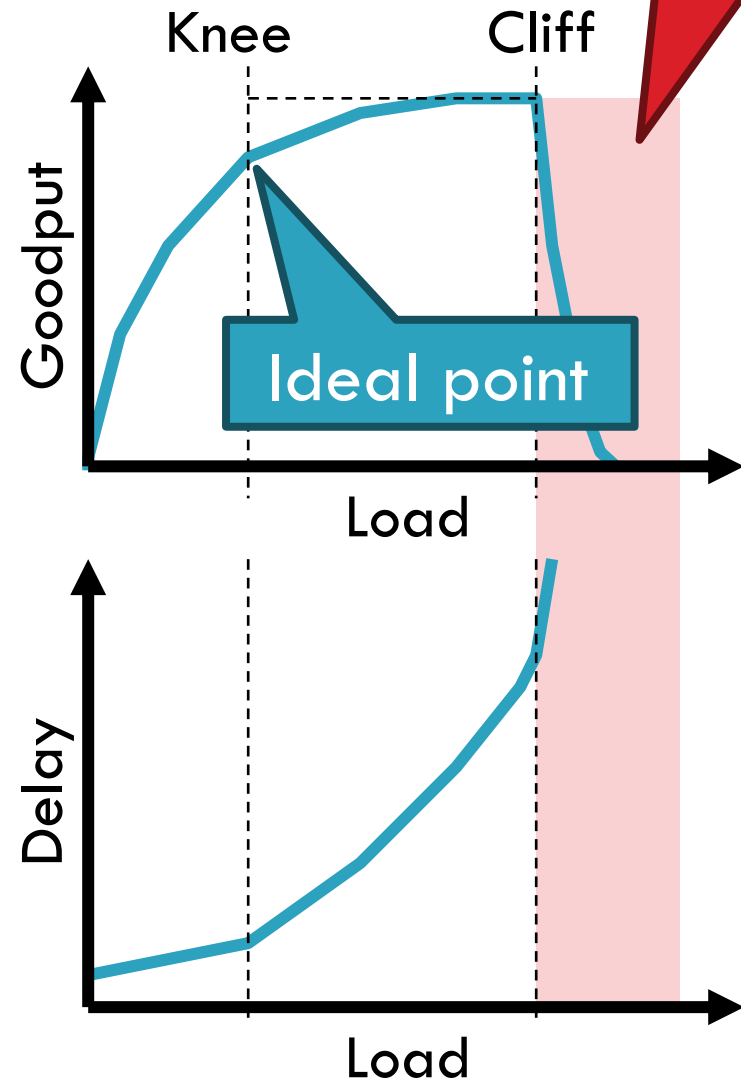
1. RTT Variance estimation
2. Exponential retransmit timer backoff
3. Slow-start
4. More aggressive receiver ack policy
5. Dynamic window sizing on congestion
6. Karn's algorithm
7. Fast retransmit

Paper explores the first 5.

# The Danger of Increasing Load

30

- Knee – point after which
  - ▣ Throughput increases very slow
  - ▣ Delay increases fast
- In an  $M/M/1$  queue
  - ▣ Delay =  $1 / (1 - \text{utilization})$
- Cliff – point after which
  - ▣ Throughput  $\rightarrow 0$
  - ▣ Delay  $\rightarrow \infty$

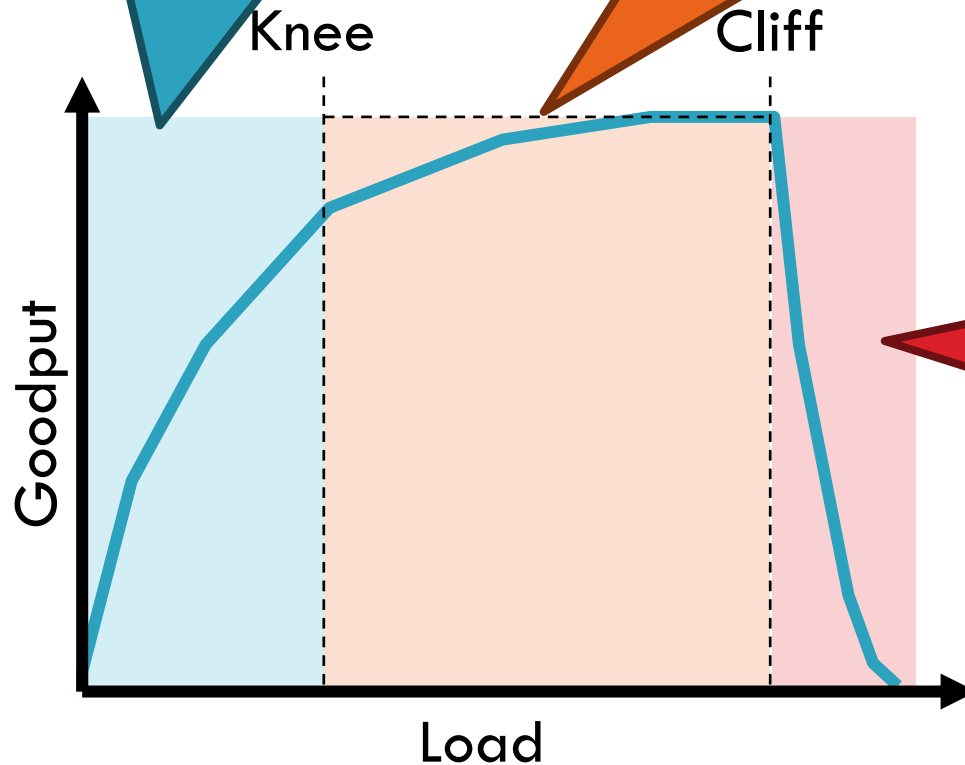


# Cong. Control vs. Cong. Avoidance

31

Congestion Avoidance:  
Stay left of the knee

Congestion Control:  
Stay left of the cliff



Congestion  
Collapse

# Advertised Window, Revisited

32

- 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

33

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

# General Approaches

34

- ❑ 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

35

- Each TCP connection has a window
  - ▣ Controls the number of unACKed packets
- Sending rate is  $\sim \text{window}/\text{RTT}$
- 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)

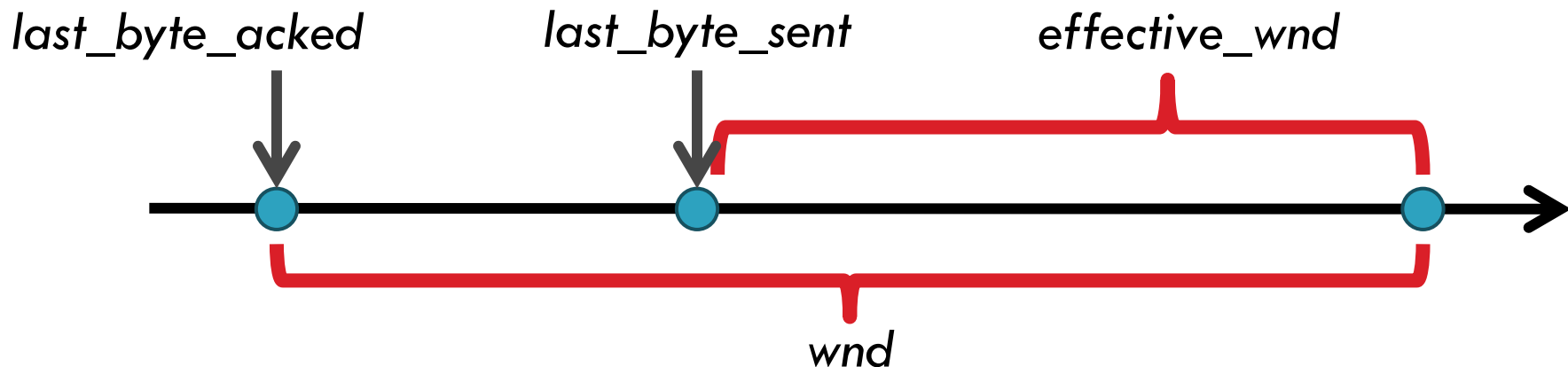
36

- 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

37

## 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

# Error Detection

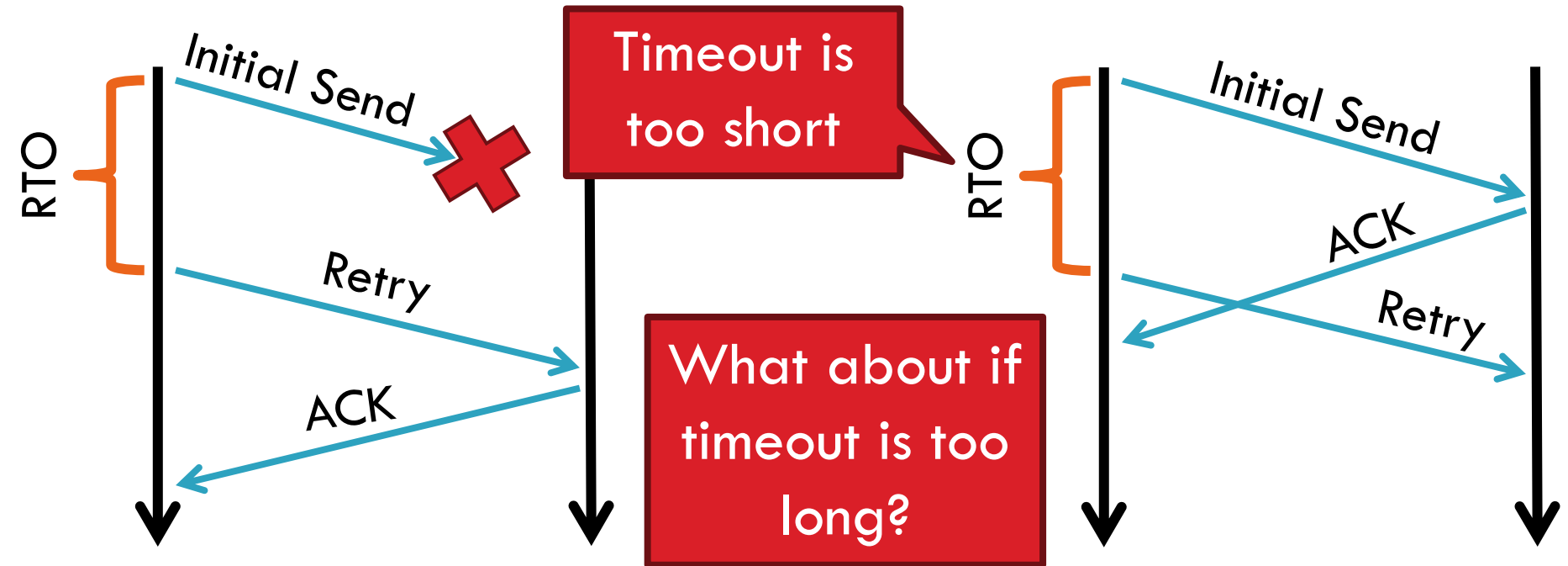
38

- ❑ 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)

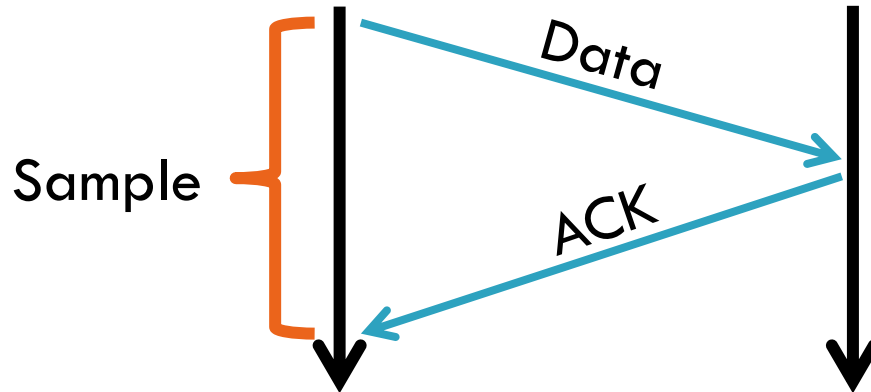
39

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



# Round Trip Time Estimation

40

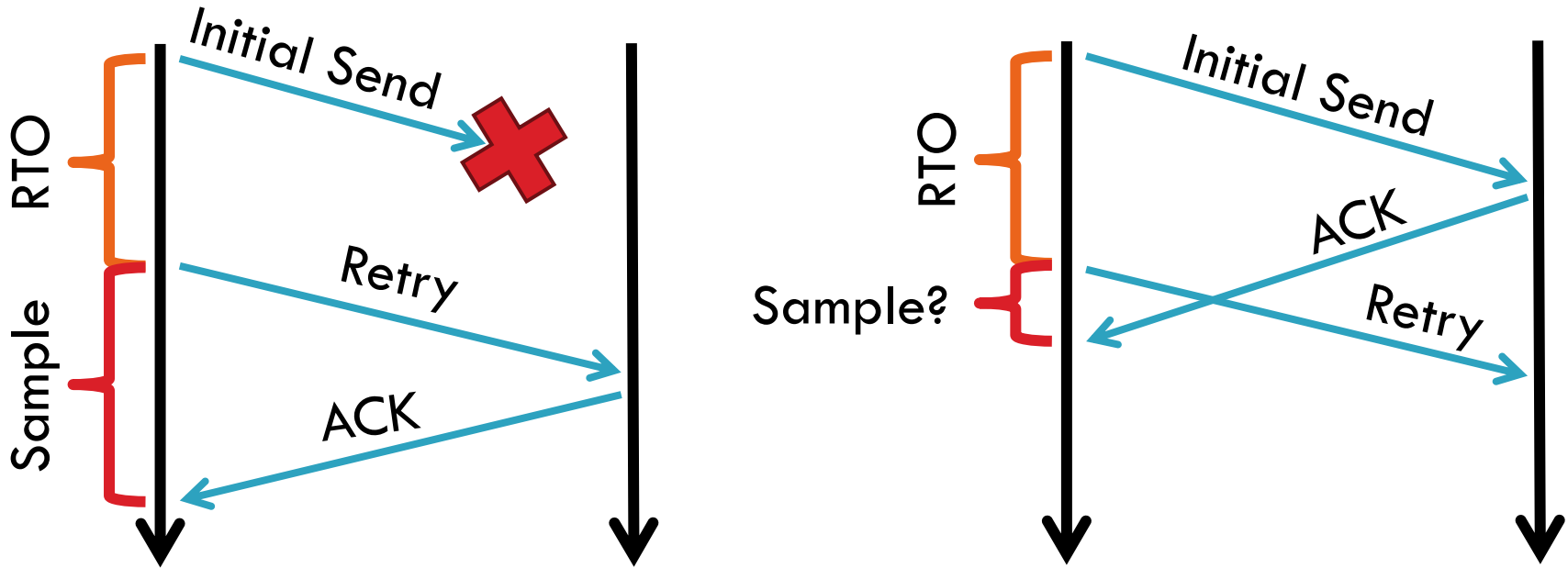


- Original TCP round-trip estimator
  - ▣ RTT estimated as a moving average
  - ▣  $\text{new\_rtt} = \alpha (\text{old\_rtt}) + (1 - \alpha)(\text{new\_sample})$
  - ▣ Recommended  $\alpha$ : 0.8-0.9 (0.875 for most TCPs)
- $\text{RTO} = \text{function of new\_rtt and new\_dev\_rtt}$



# RTT Sample Ambiguity

41



- Karn's algorithm: ignore samples for retransmitted segments

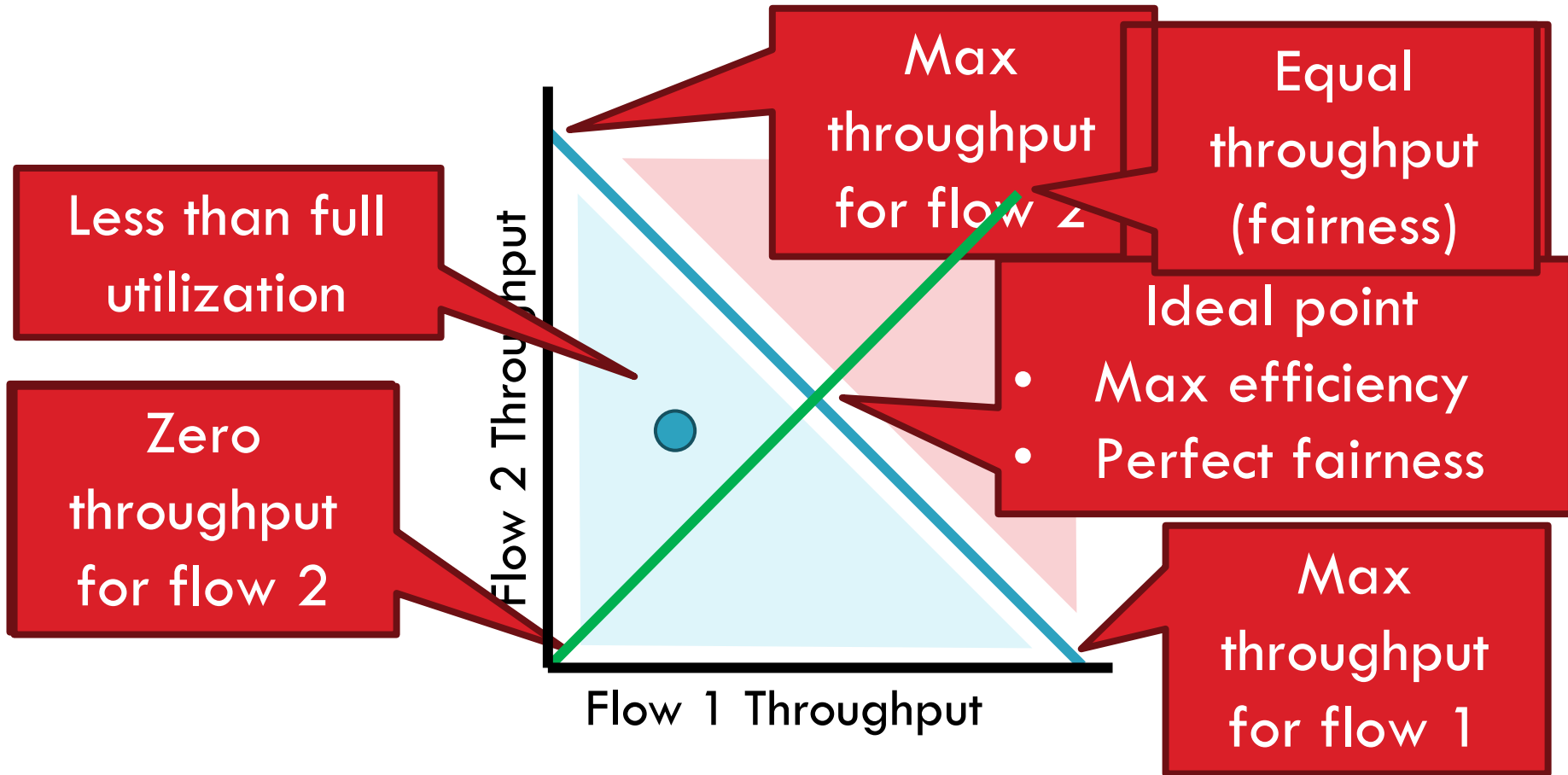
# Rate Adjustment

42

- 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

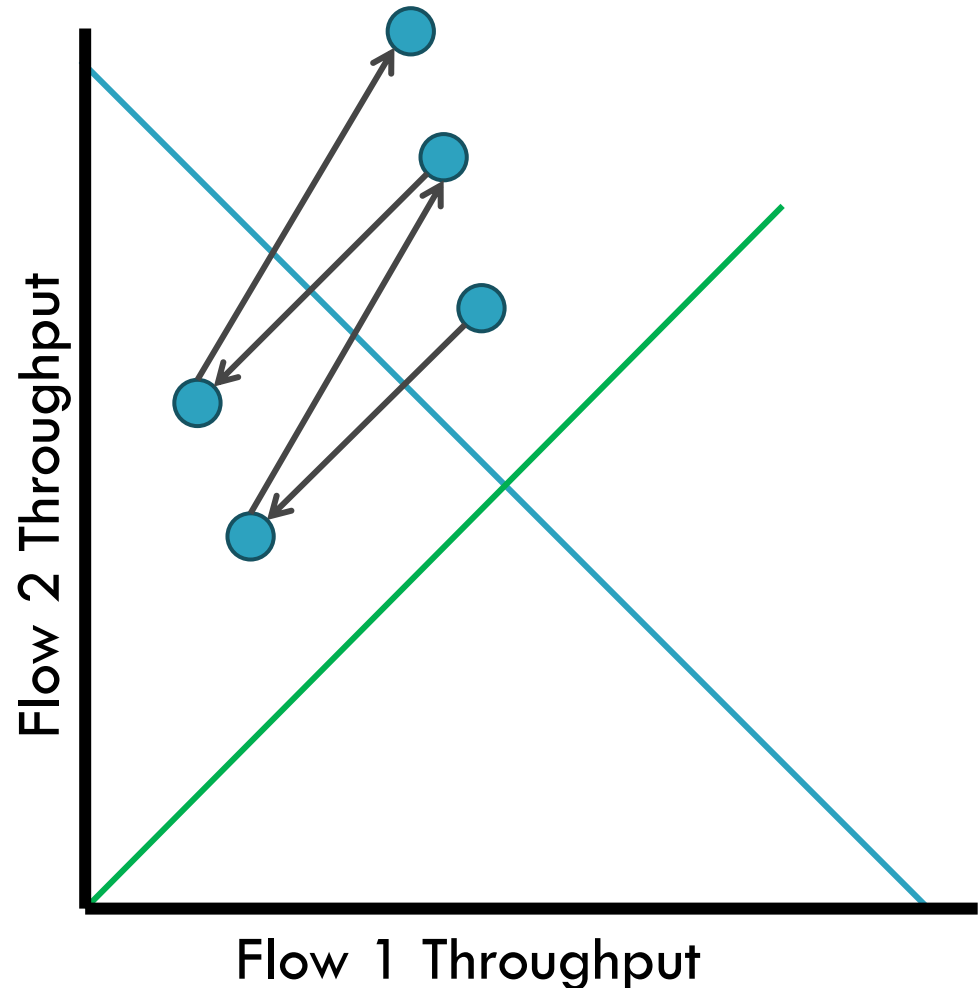
43



# Multiplicative Increase, Additive Decrease

44

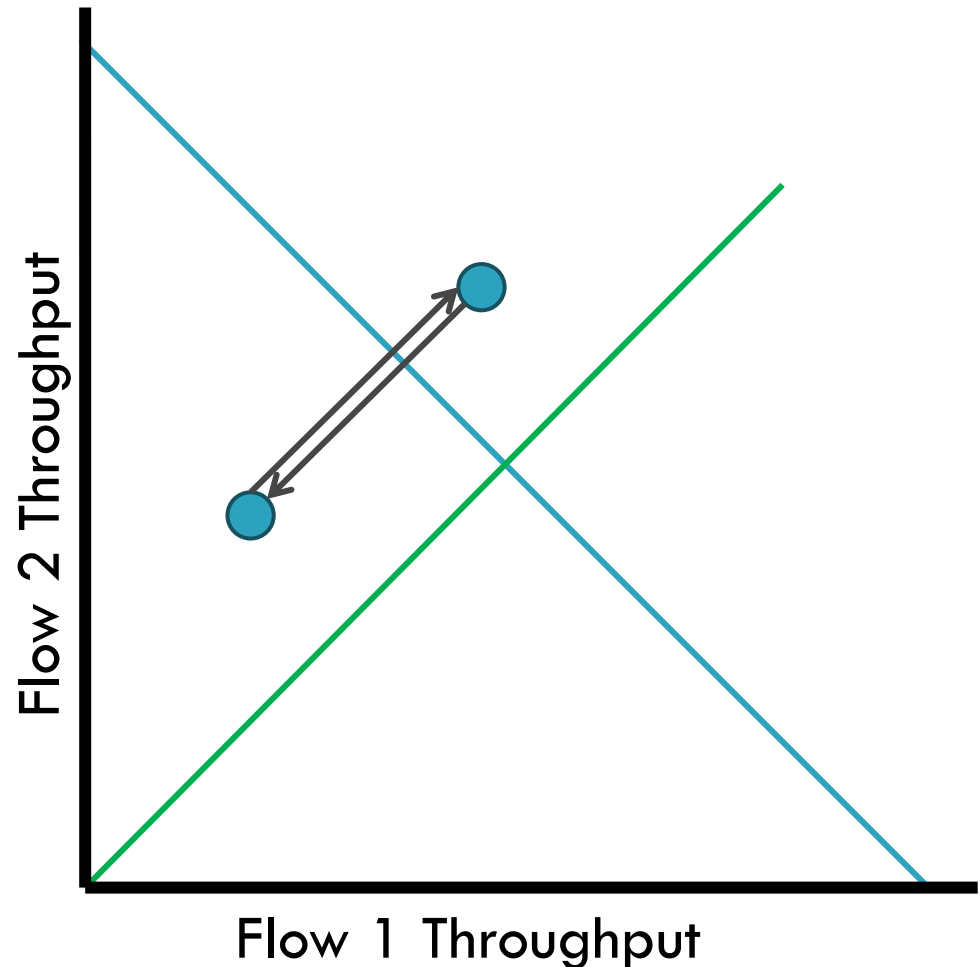
- ❑ Not stable!
- ❑ Veers away from fairness



# Additive Increase, Additive Decrease

45

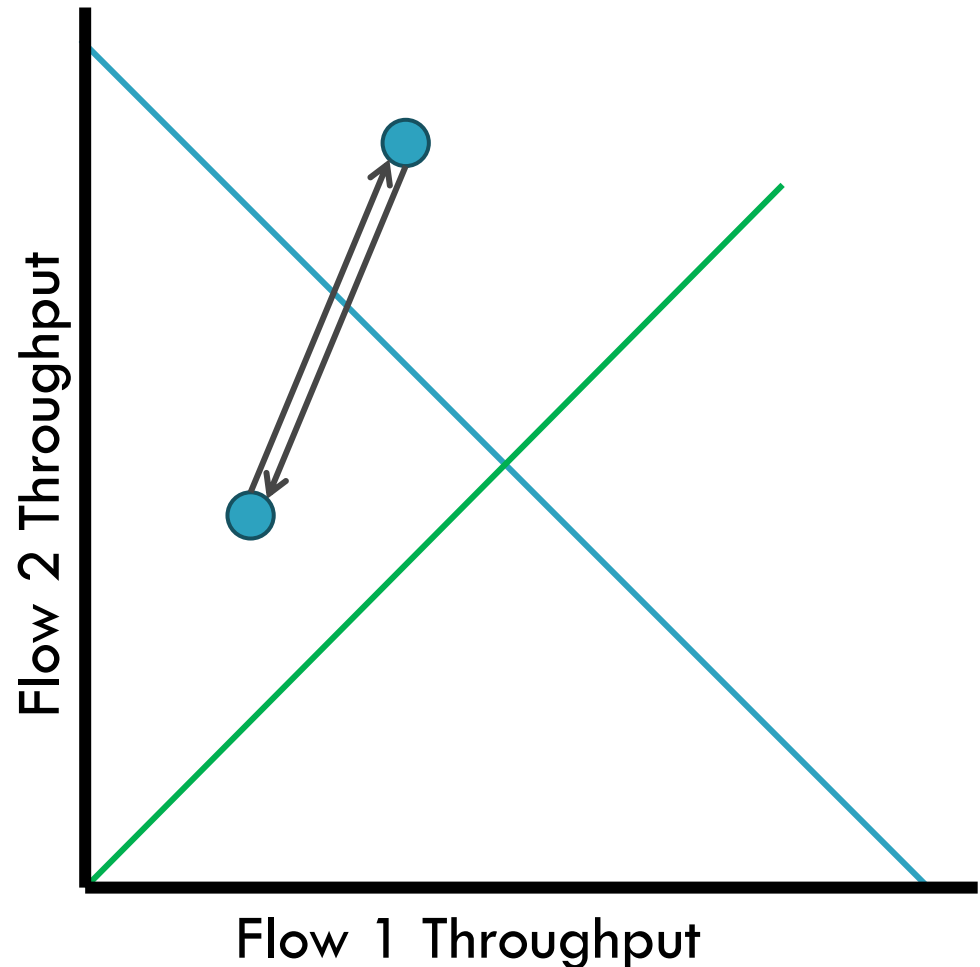
- Stable
- But does not converge to fairness



# Multiplicative Increase, Multiplicative Decrease

46

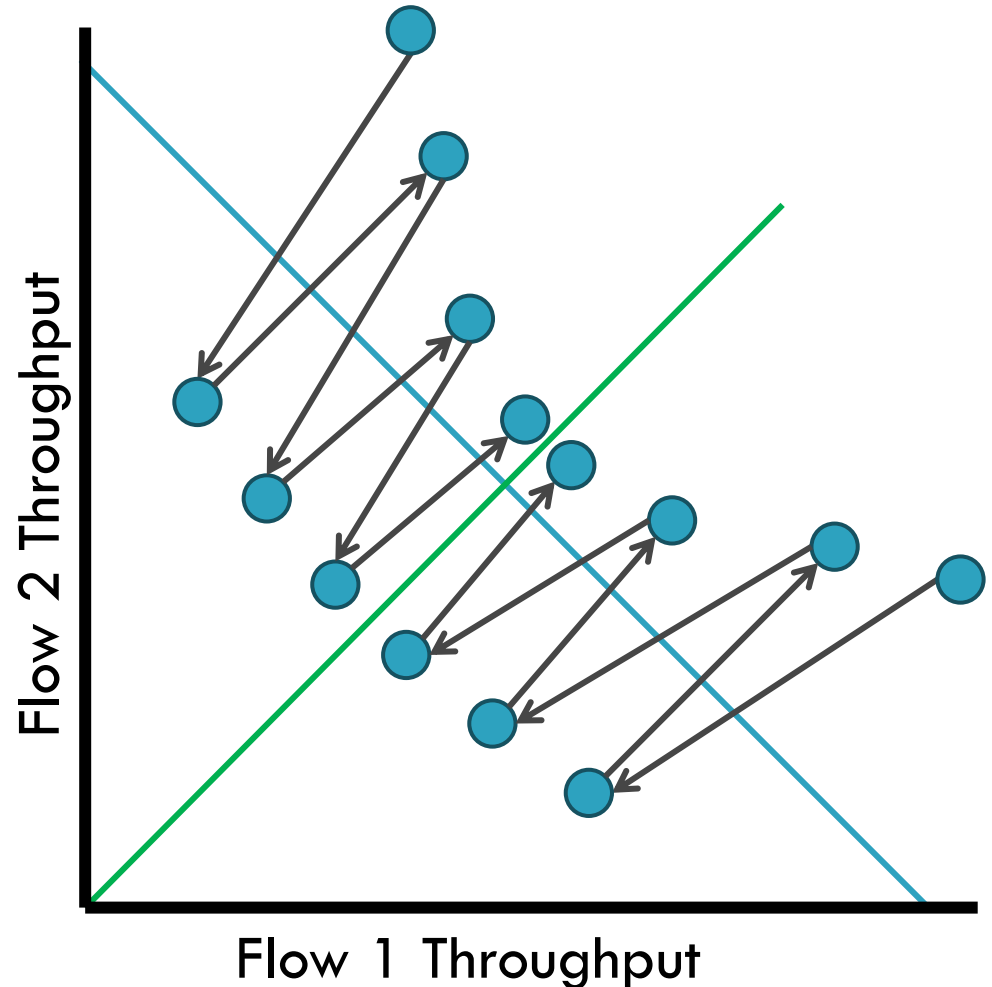
- Stable
- But does not converge to fairness



# Additive Increase, Multiplicative Decrease

47

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



# Implementing Congestion Control

48

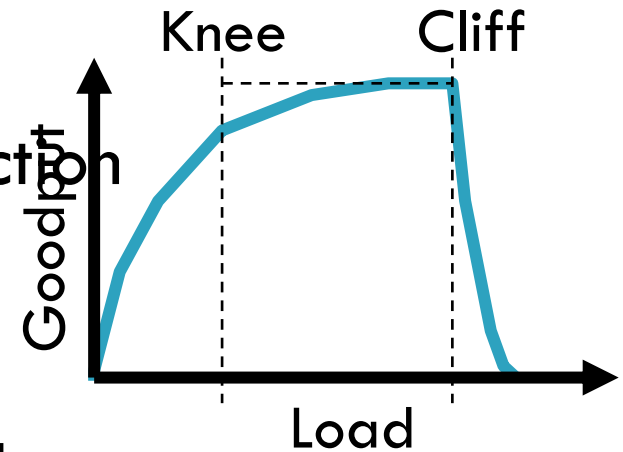
- Maintains three variables:
  - *cwnd*: congestion window
  - *adv\_wnd*: receiver advertised window
  - *ssthresh*: threshold size (used to update *cwnd*)
- For sending, use:  $wnd = \min(cwnd, adv\_wnd)$
- Two phases of congestion control
  1. Slow start ( $cwnd < ssthresh$ )
    - Probe for bottleneck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



# Slow Start

49

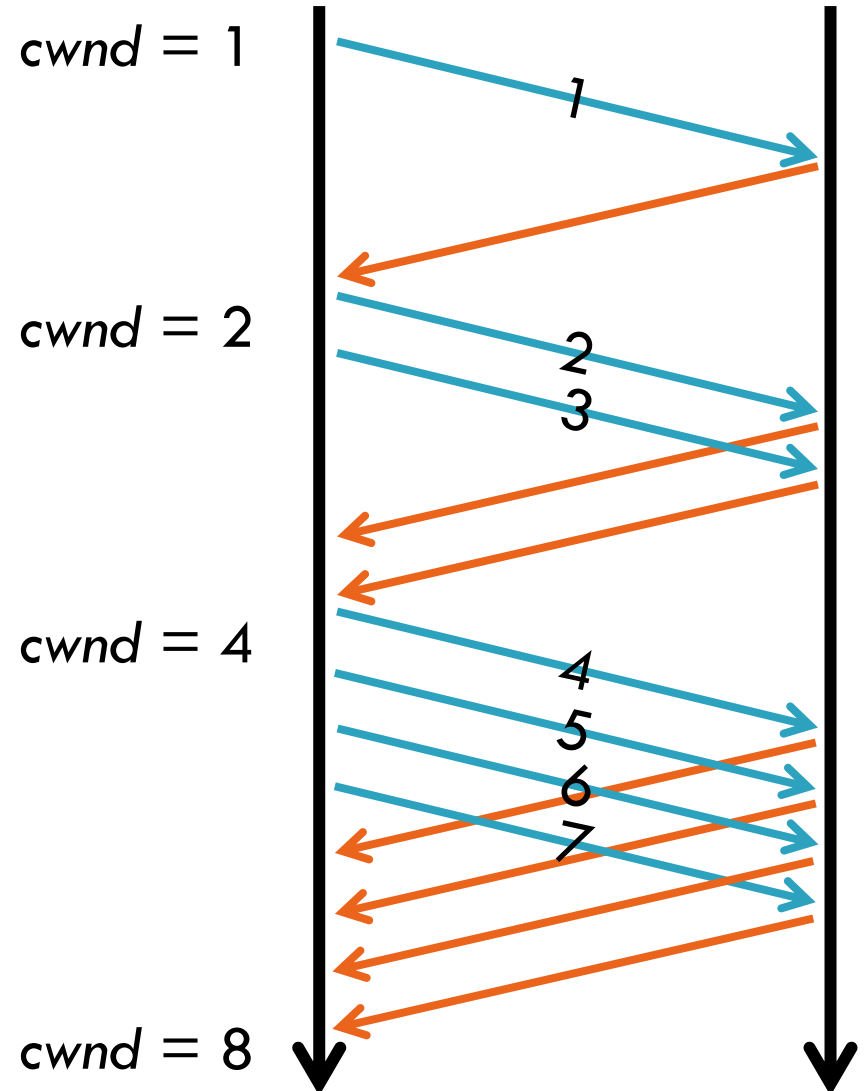
- Goal: reach knee quickly
- Upon starting (or restarting) a connection
  - $cwnd = 1$
  - $ssthresh = adv\_wnd$
  - Each time a segment is ACKed,  $cwnd++$
- Continues until...
  - $ssthresh$  is reached
  - Or a packet is lost
- Slow Start is not actually slow
  - $cwnd$  increases exponentially



# Slow Start Example

50

- $cwnd$  grows rapidly
- Slows down when...
  - ▣  $cwnd \geq ssthresh$
  - ▣ Or a packet drops



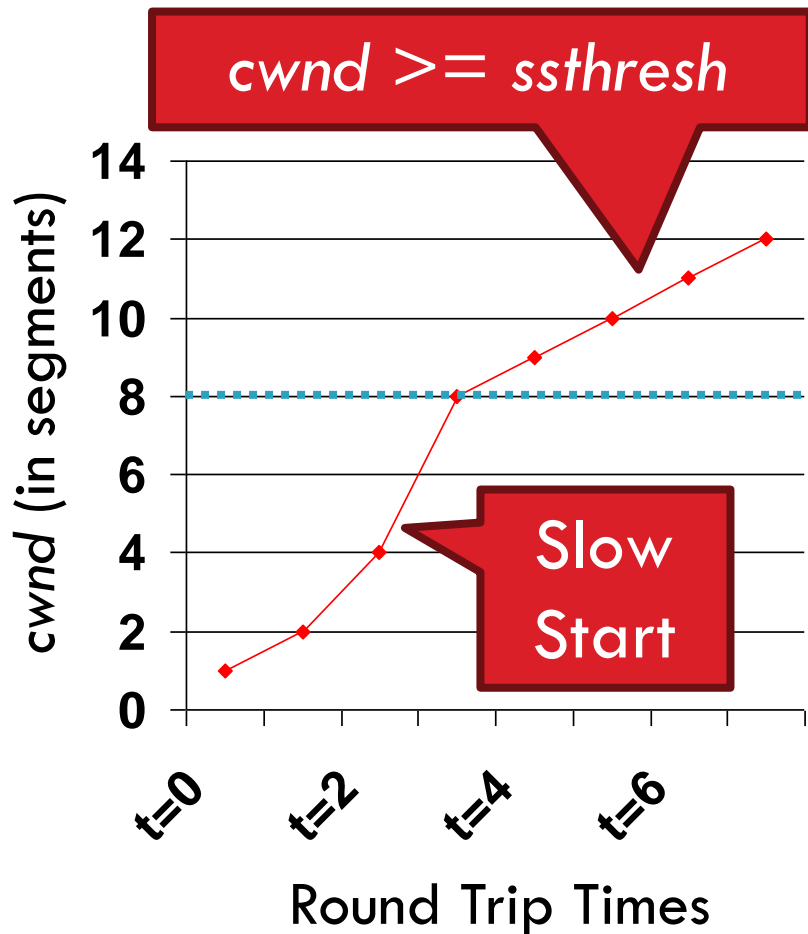
# Congestion Avoidance

51

- AIMD mode
- *ssthresh* is lower-bound guess about location of the knee
- **If** *cwnd*  $\geq$  *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

52



$cwnd = 1$

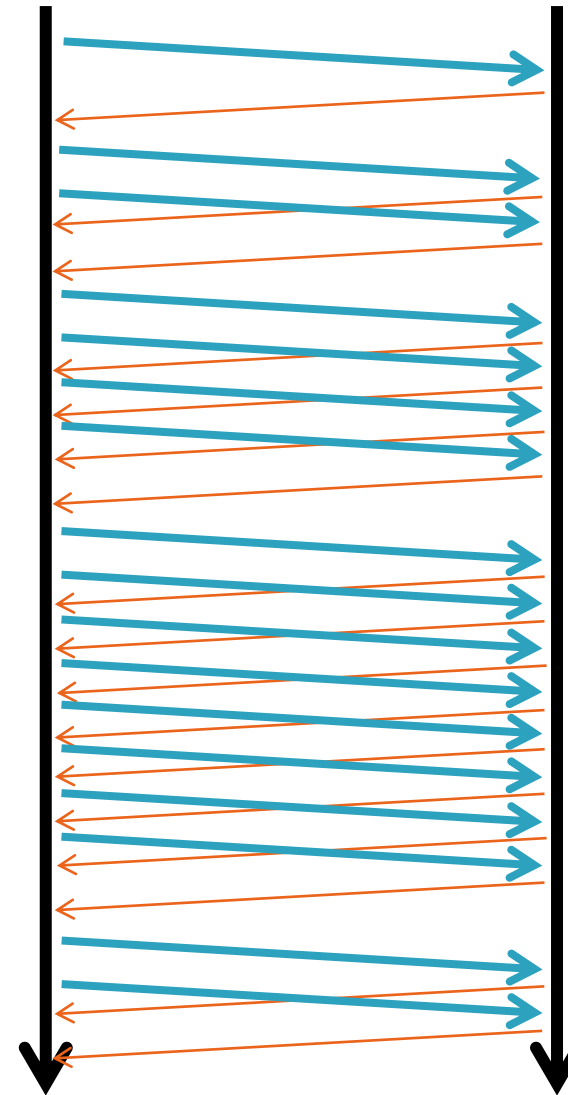
$cwnd = 2$

$cwnd = 4$

$ssthresh = 8$

$cwnd = 8$

$cwnd = 9$



# TCP Pseudocode

53

## **Initially:**

```
    cwnd = 1;  
    ssthresh = adv_wnd;
```

## **New ack received:**

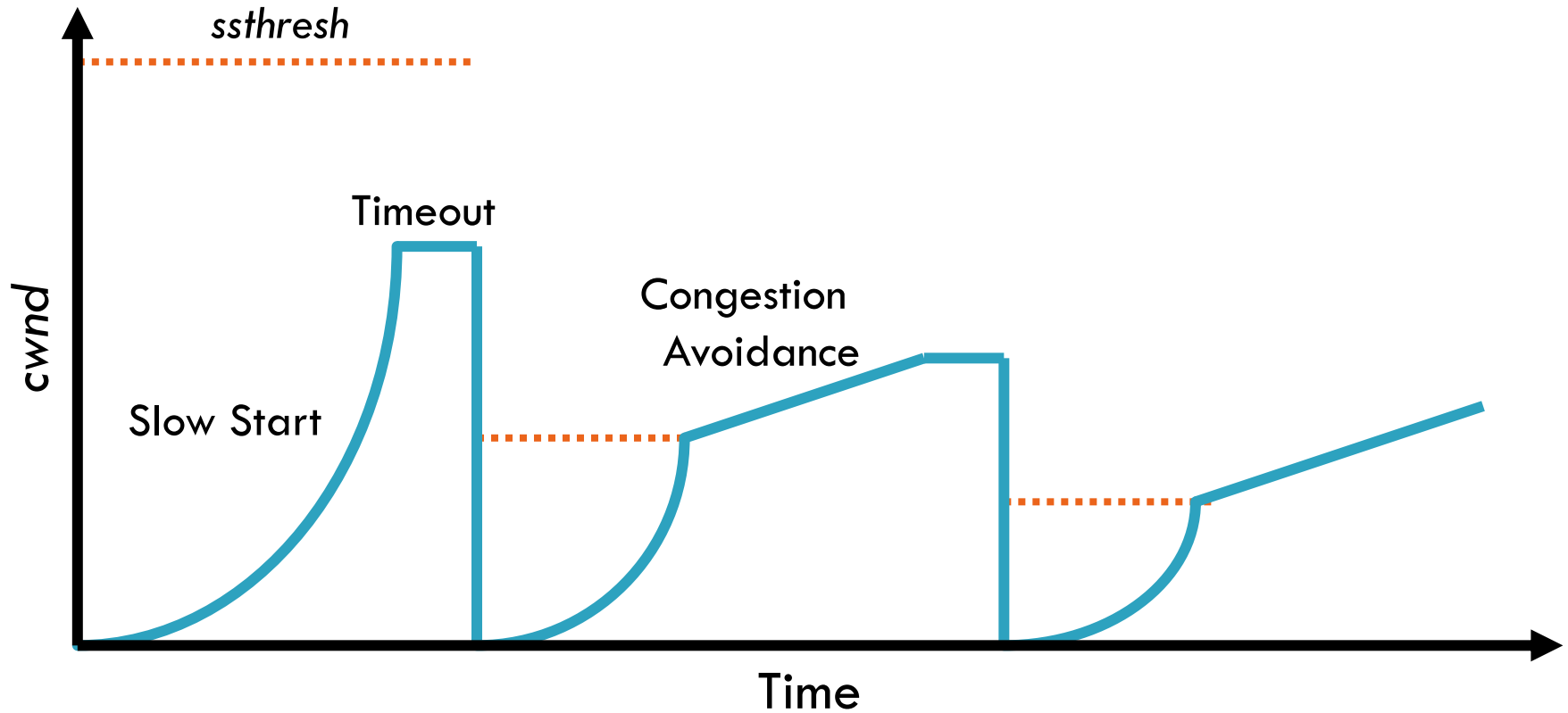
```
    if (cwnd < ssthresh)  
        /* Slow Start */  
        cwnd = cwnd + 1;  
    else  
        /* Congestion Avoidance */  
        cwnd = cwnd + 1/cwnd;
```

## **Timeout:**

```
    /* Multiplicative decrease */  
    ssthresh = cwnd/2;  
    cwnd = 1;
```

# The Big Picture

54



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

# The Evolution of TCP

56

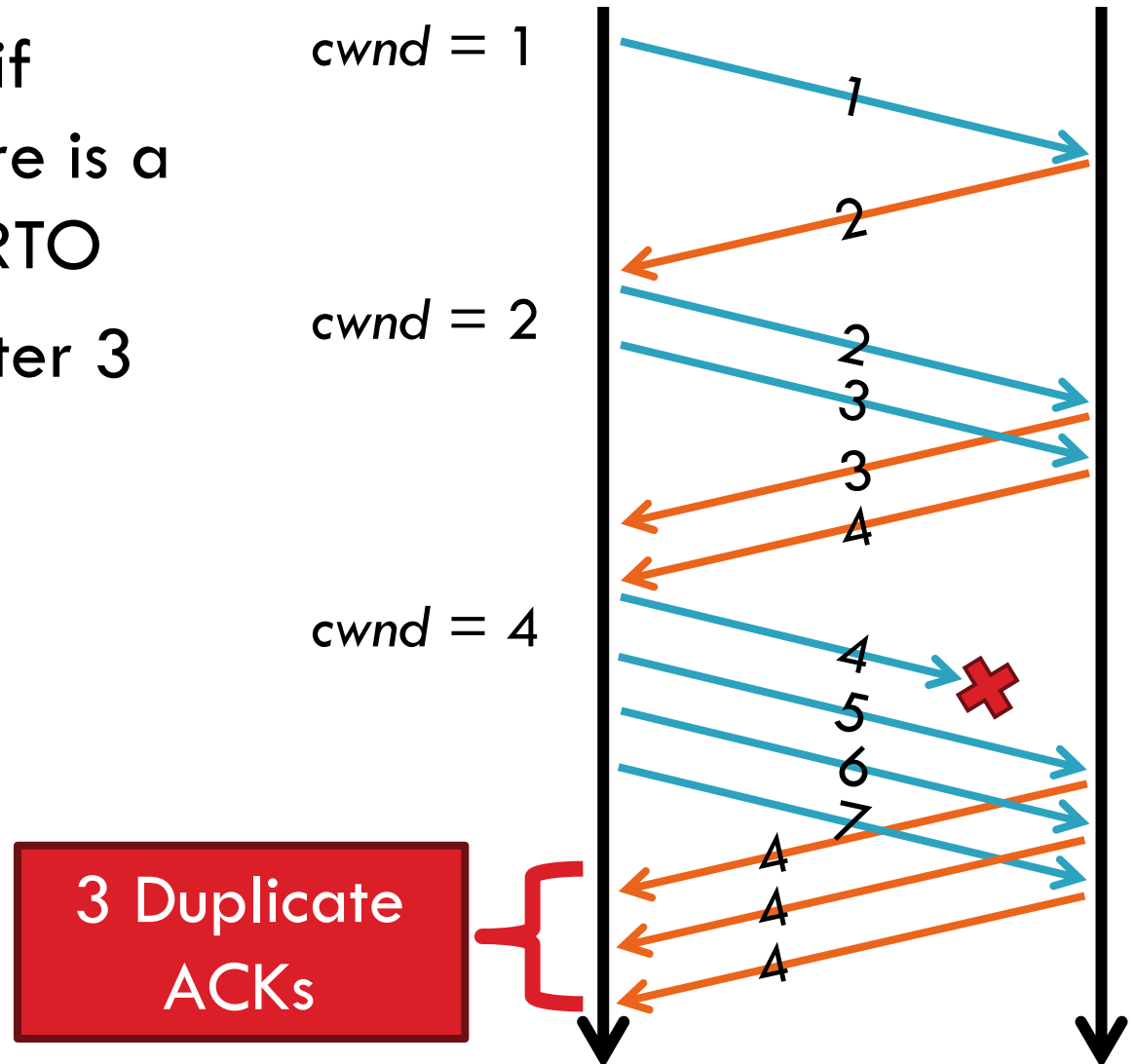
- Thus far, we have discussed TCP Tahoe
  - ▣ Original version of TCP
- However, TCP was invented in 1974!
  - ▣ Today, there are many variants of TCP
- Early, popular variant: TCP Reno
  - ▣ Tahoe features, plus...
  - ▣ Fast retransmit
  - ▣ Fast recovery



# TCP Reno: Fast Retransmit

57

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



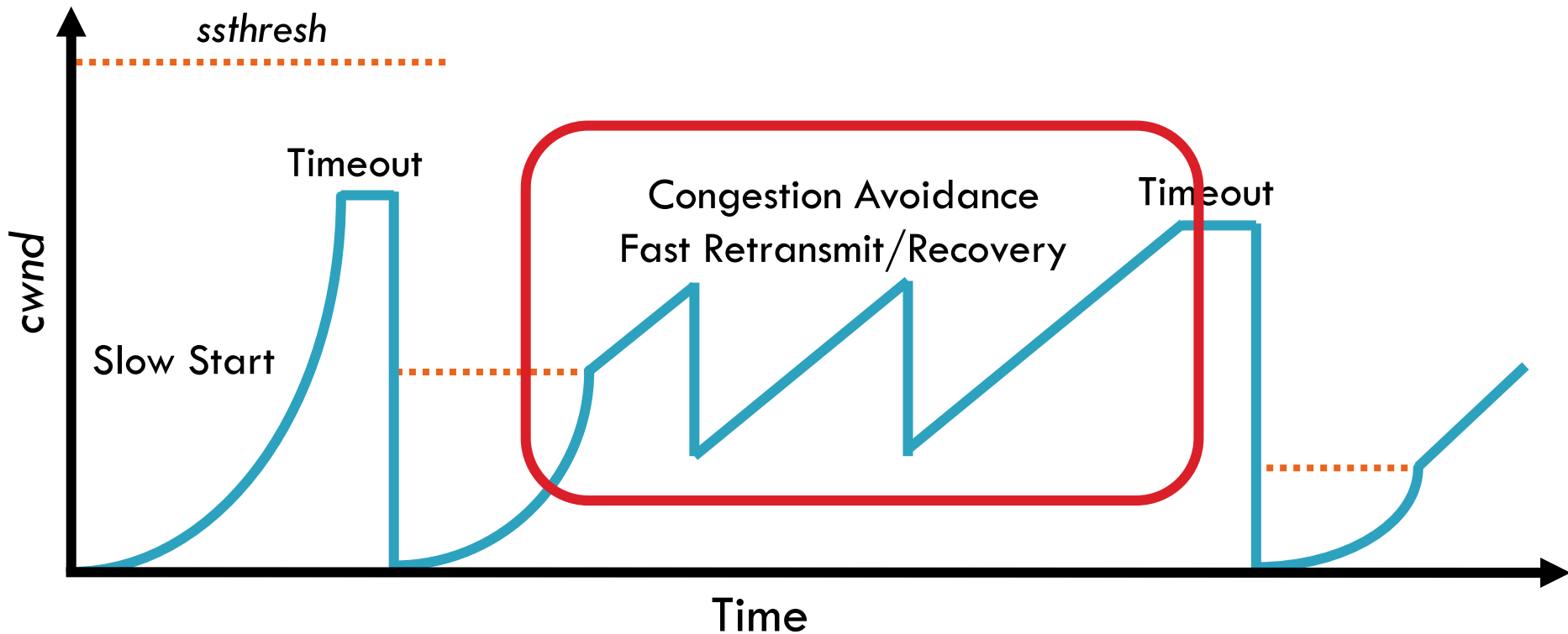
# TCP Reno: Fast Recovery

58

- 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

59



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

# Many TCP Variants...

60

- ❑ Tahoe: the original
  - ❑ Slow start with AIMD
  - ❑ Dynamic RTO based on RTT estimate
- ❑ Reno: fast retransmit and fast recovery
- ❑ NewReno: improved fast retransmit
  - ❑ Reduce number of retransmissions
  - ❑ Window inflation
- ❑ Vegas: delay-based congestion avoidance
- ❑ And many, many, many more...

# TCP in the Real World

61

- 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

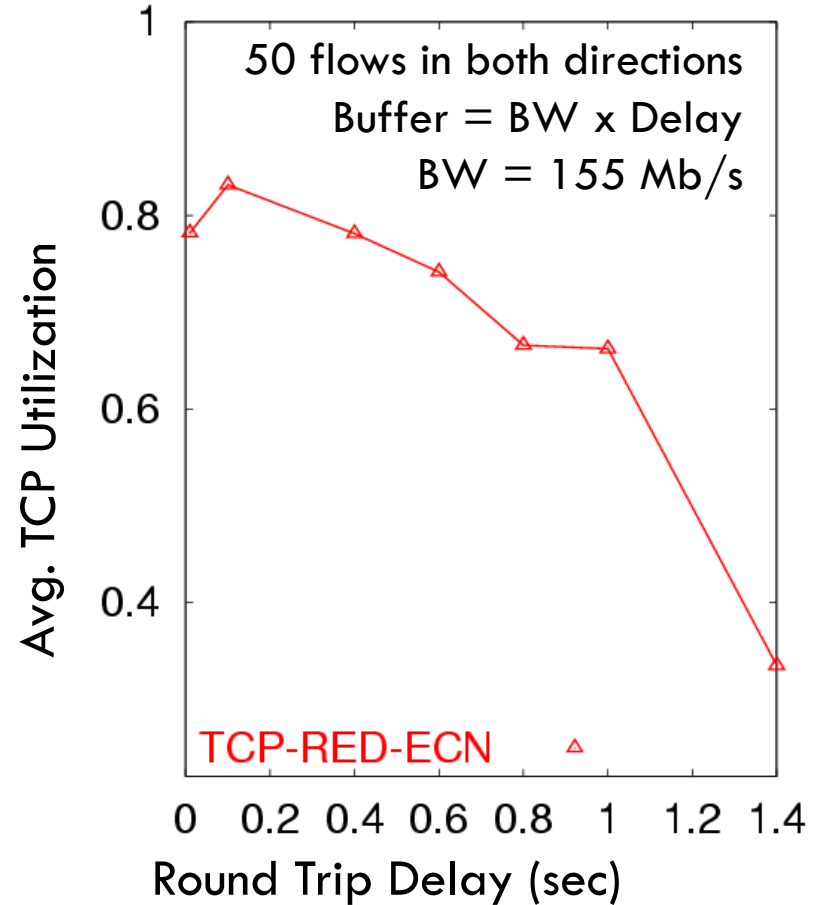
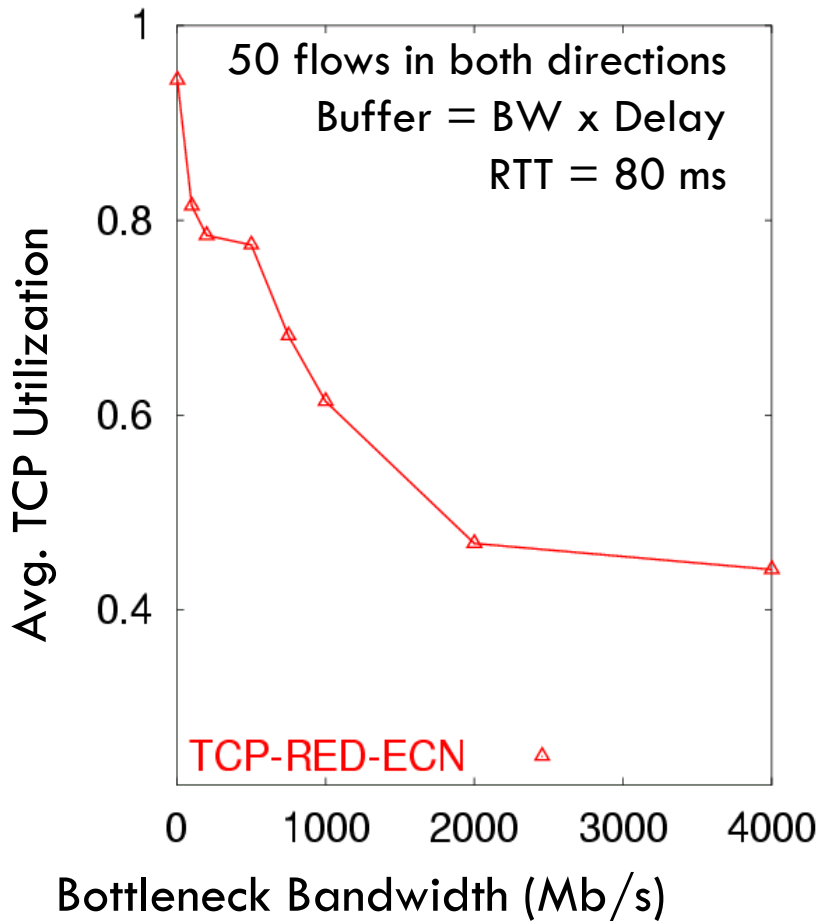
# High Bandwidth-Delay Product

62

- 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  $\rightarrow$  ACKs are delayed  $\rightarrow$  TCP is slow to react

# Poor Performance of TCP Reno CC

63



# Goals

64

- ❑ 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 variants
  - ❑ Window growth cannot be too aggressive
- ❑ Improve RTT fairness
  - ❑ TCP Tahoe/Reno flows are not fair when RTTs vary widely
- ❑ Simple implementation



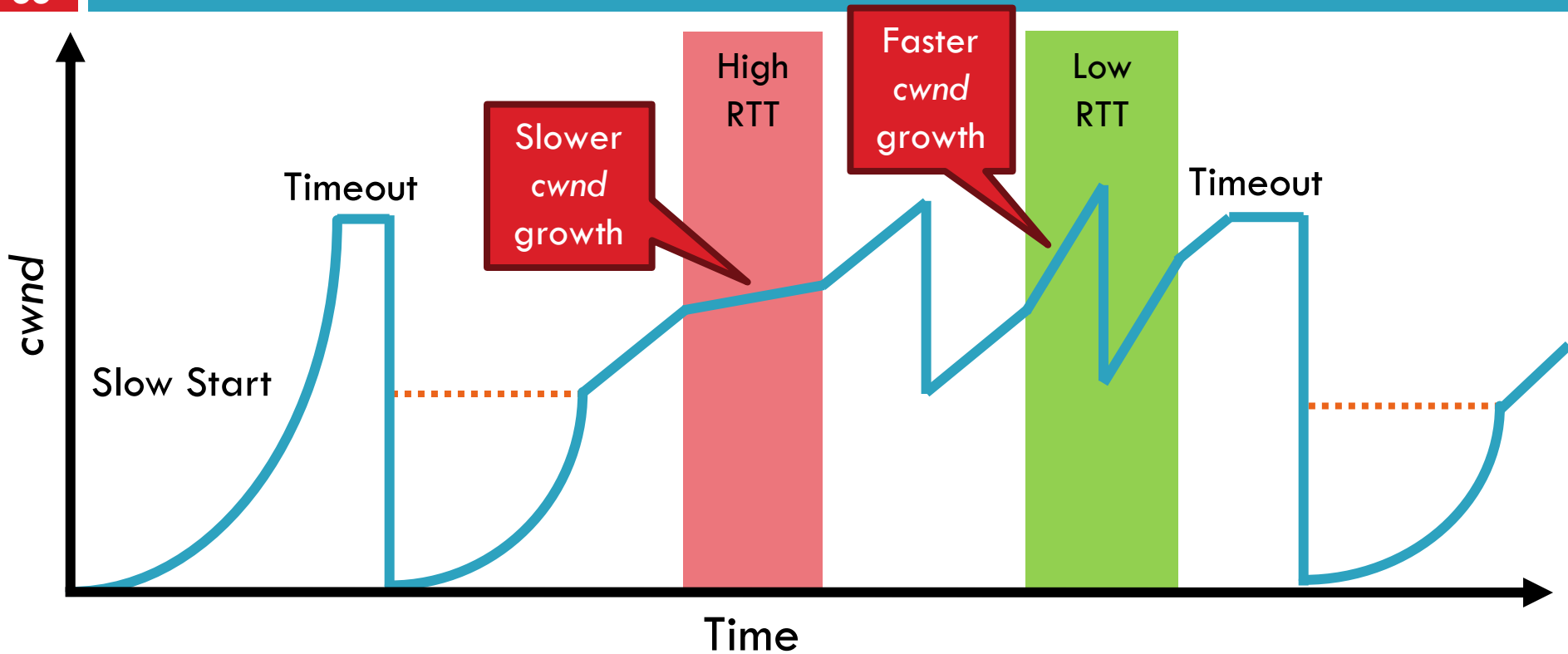
# Compound TCP Implementation

65

- 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* ( $dwnd \geq 0$ )
  - ▣ If RTT is decreasing, increase *dwnd*
  - ▣ Increase/decrease are proportional to the rate of change

# Compound TCP Example

66



- ❑ 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

67

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

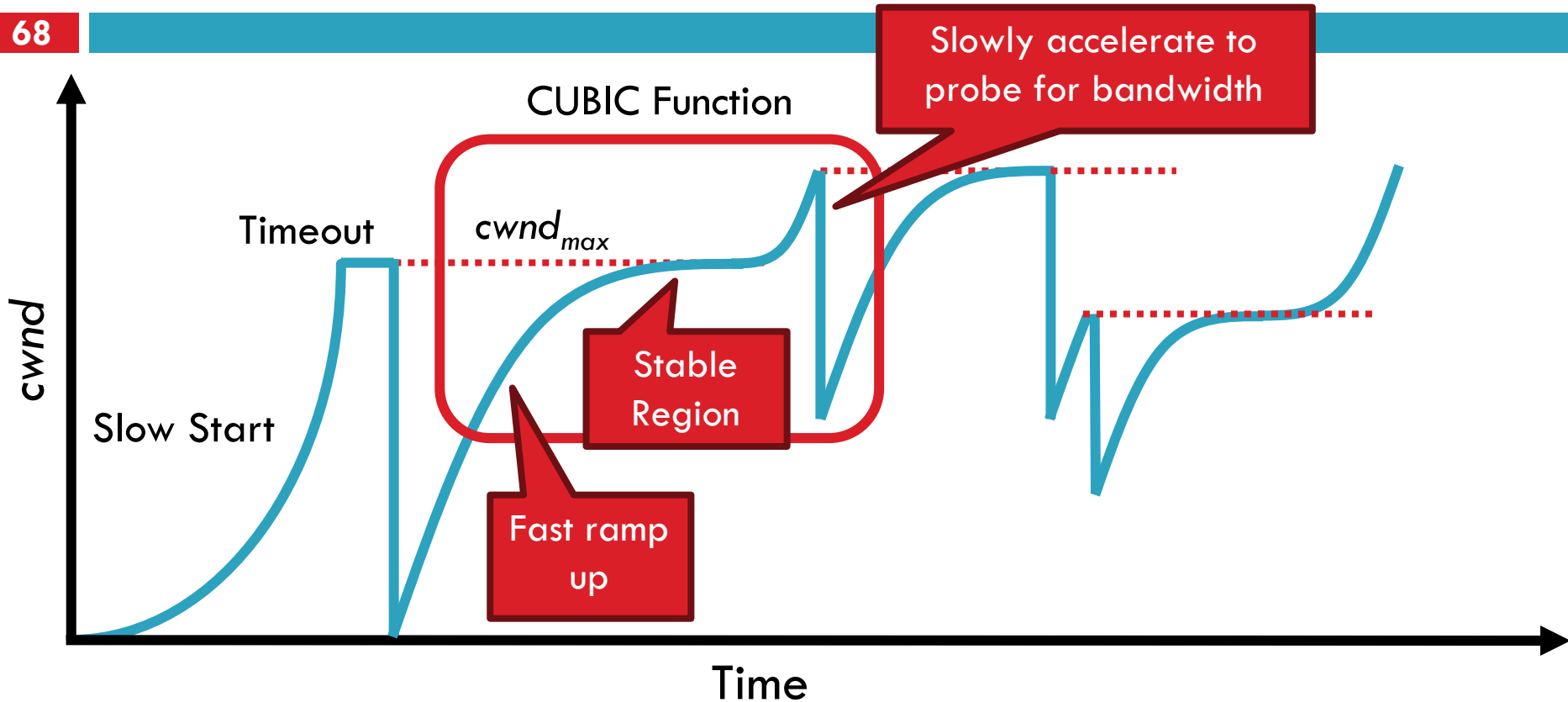
$$W_{cubic} = C(T - K)^3 + W_{max} \quad (1)$$

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

- B → a constant fraction for multiplicative increase
- T → time since last packet drop
- $W_{max}$  → cwnd when last packet dropped

# TCP CUBIC Example

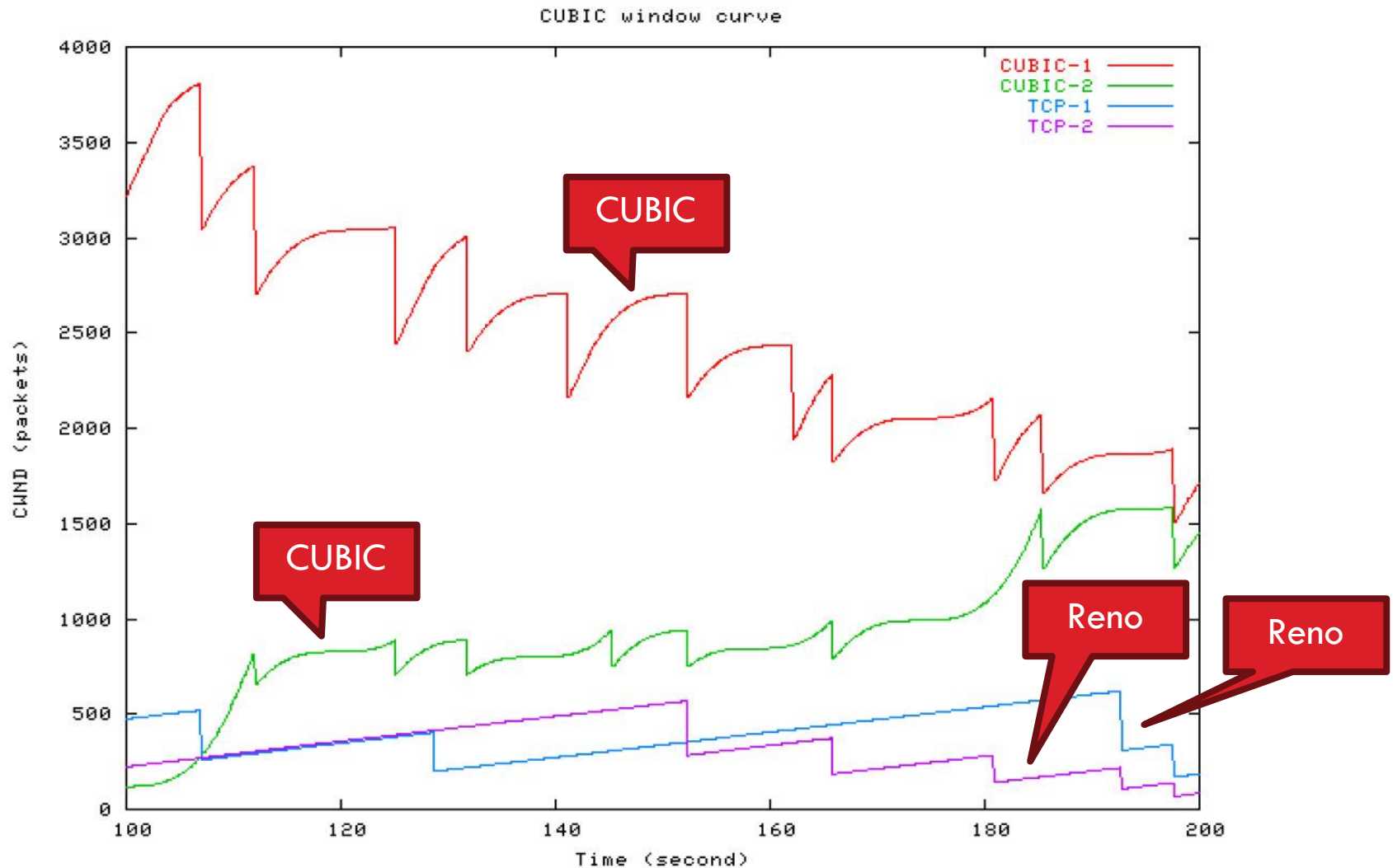
68



- ❑ 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

69



# 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



# Issues with TCP

72

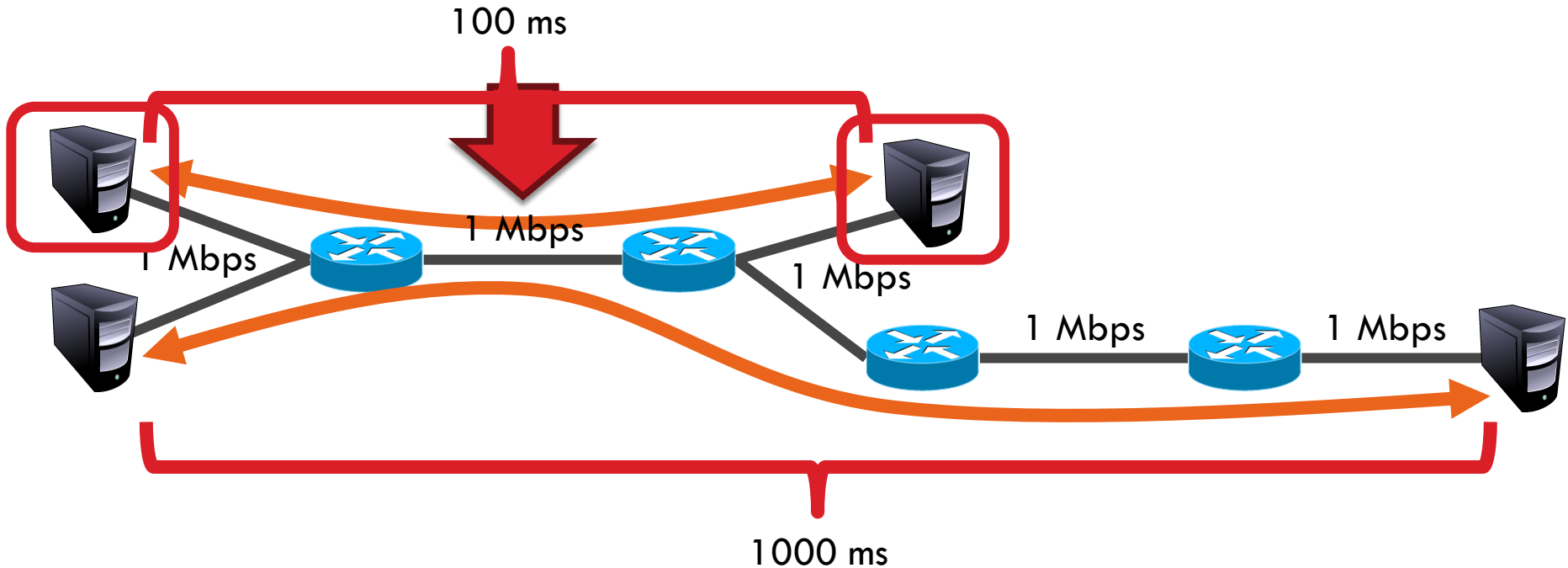
- The vast majority of Internet traffic is TCP
- However, many issues with the protocol
  - ▣ Lack of fairness
  - ▣ Synchronization of flows
  - ▣ Poor performance with small flows
  - ▣ Really poor performance on wireless networks
  - ▣ Susceptibility to denial of service



# Fairness

73

- Problem: TCP throughput depends on RTT

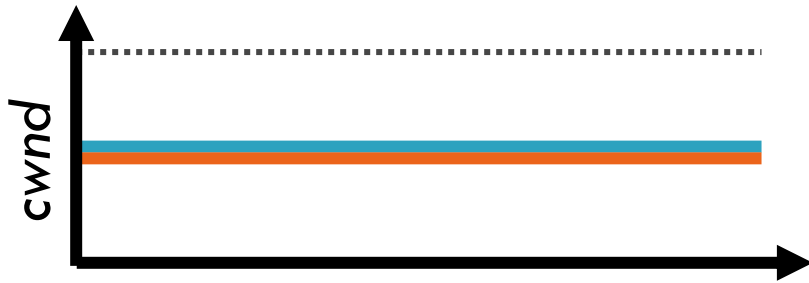


- ACK clocking makes TCP inherently unfair
- Possible solution: maintain a separate delay window
  - ▣ Implemented by Microsoft's Compound TCP

# Synchronization of Flows

74

- Ideal bandwidth sharing

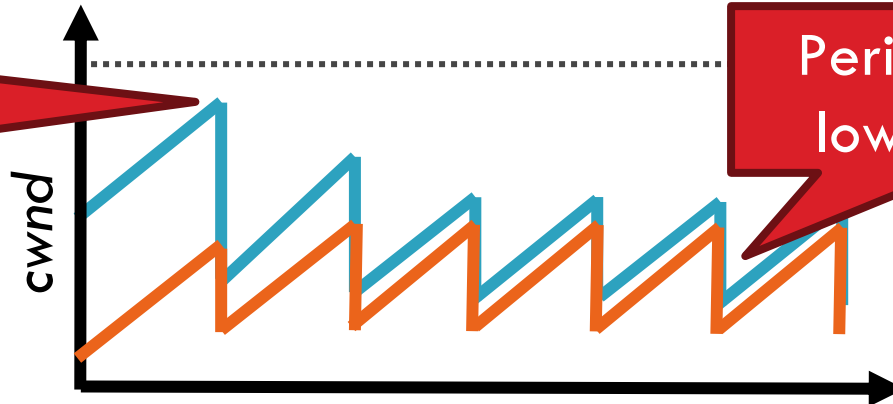


- Oscillating, but high overall utilization



- In reality, flows synchronize

One flow causes all flows to drop packets



Periodic lulls of low utilization

# Small Flows

75

- ❑ 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

76

- 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  $\sim 1/\sqrt{\text{drop rate}}$ 
  - ▣ 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

77

- ❑ 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

78



- ❑ 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

79

## □ 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.





# More slides ...

# What Should the Receiver ACK?

82

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

83

- 32 bits, unsigned
  - ▣ Why so big?
- For the sliding window you need...
  - ▣  $|\text{Sequence \# Space}| > 2 * |\text{Sending Window Size}|$
  - ▣  $2^{32} > 2 * 2^{16}$
- 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
  - ▣ Sequence number would wrap around at 286Mbps
    - What about GigE? PAWS algorithm + TCP options

# Silly Window Syndrome

84

□ 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

85

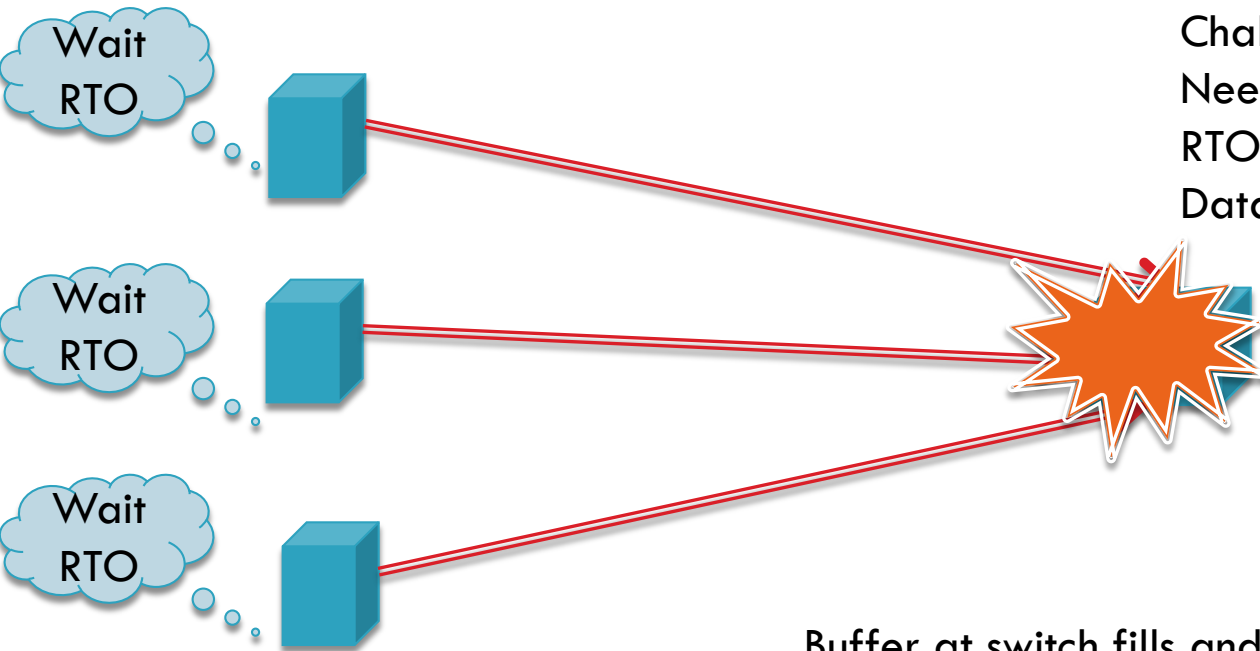
1. If the window  $\geq$  MSS and available data  $\geq$  MSS:  
Send the data  Send a full packet
  2. Elif there is unACKed data:  
Enqueue data in a buffer until an ACK is received
  3. Else: send the data  Send a non-full packet if nothing else is happening
- Problem: Nagle's Algorithm delays transmissions
- ▣ What if you need to send a packet immediately?
    1. `int flag = 1;`
    2. `setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *) &flag, sizeof(int));`

# Challenge of RTO in data centers

86

- TCP Incast problem – E.g. Hadoop, Map Reduce, HDFS, GFS

Many senders sending simultaneously to receiver



Challenges:

Need to break synchronization

RTO estimation designed for wide area

Data centers have much smaller RTT

Buffer at switch fills and packets are lost!

No ACKs will come back ☹️

# TCP Perspectives

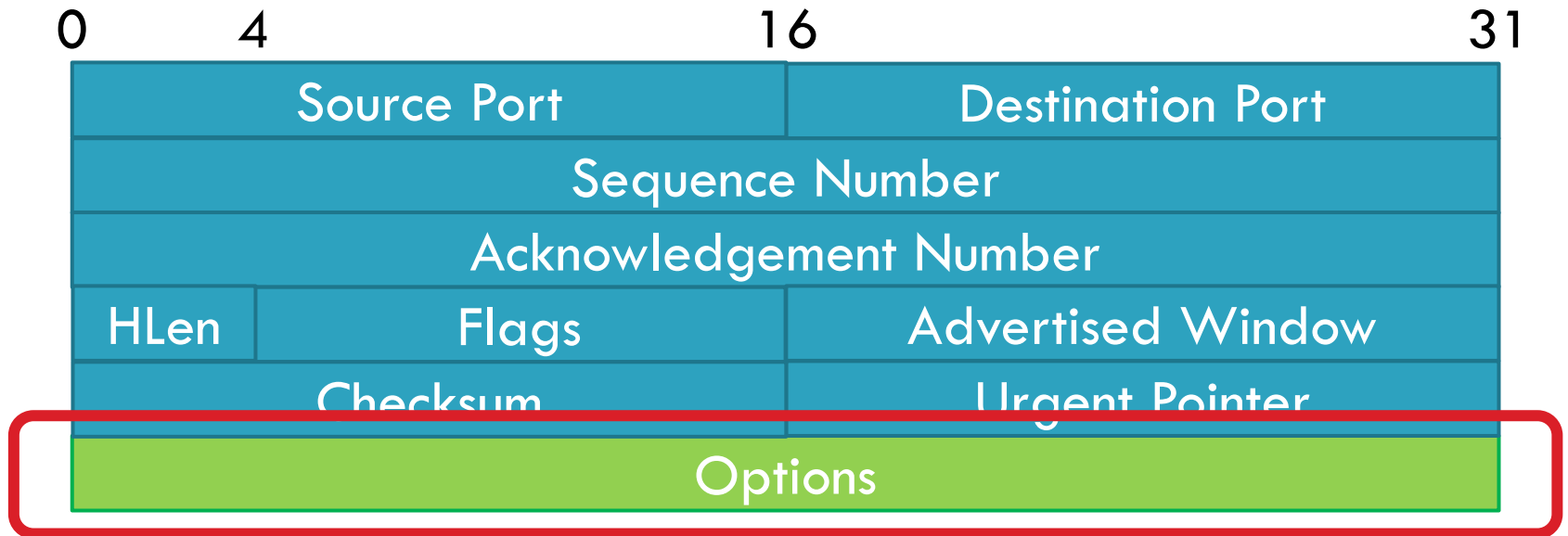
- Cerf/Kahn
  - ▣ Provide flow control
  - ▣ Congestion handled by retransmission
- Jacobson / Karels
  - ▣ Need to avoid congestion
  - ▣ RTT estimates critical
  - ▣ Queuing theory can help
- Winstein/Balakrishnan
  - ▣ TCP is maximizing an objective function
    - Fairness/efficiency
    - Throughput/delay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

- ❑ UDP
- ❑ TCP
- ❑ Congestion Control
- ❑ Evolution of TCP
- ❑ Common TCP options
- ❑ Problems with TCP



# Common TCP Options

89



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

# Window Scaling

90

- Problem: the advertised window is only 16-bits

- Effectively caps the window at 65536B, 64KB

- Example: 1.5Mbps link, 513ms RTT

$$(1.5\text{Mbps} * 0.513\text{s}) = 94\text{KB}$$

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

- Solution: introduce a window scaling value

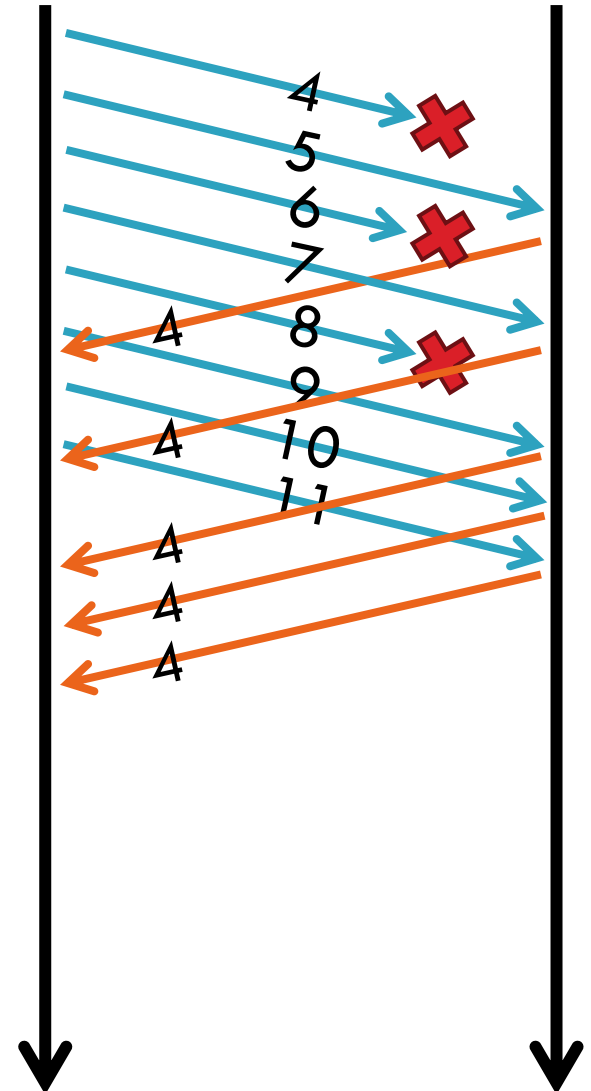
- $wnd = adv\_wnd \ll wnd\_scale;$

- Maximum shift is 14 bits, 1GB maximum window

# SACK: Selective Acknowledgment

91

- ❑ 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

92

- ❑ 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