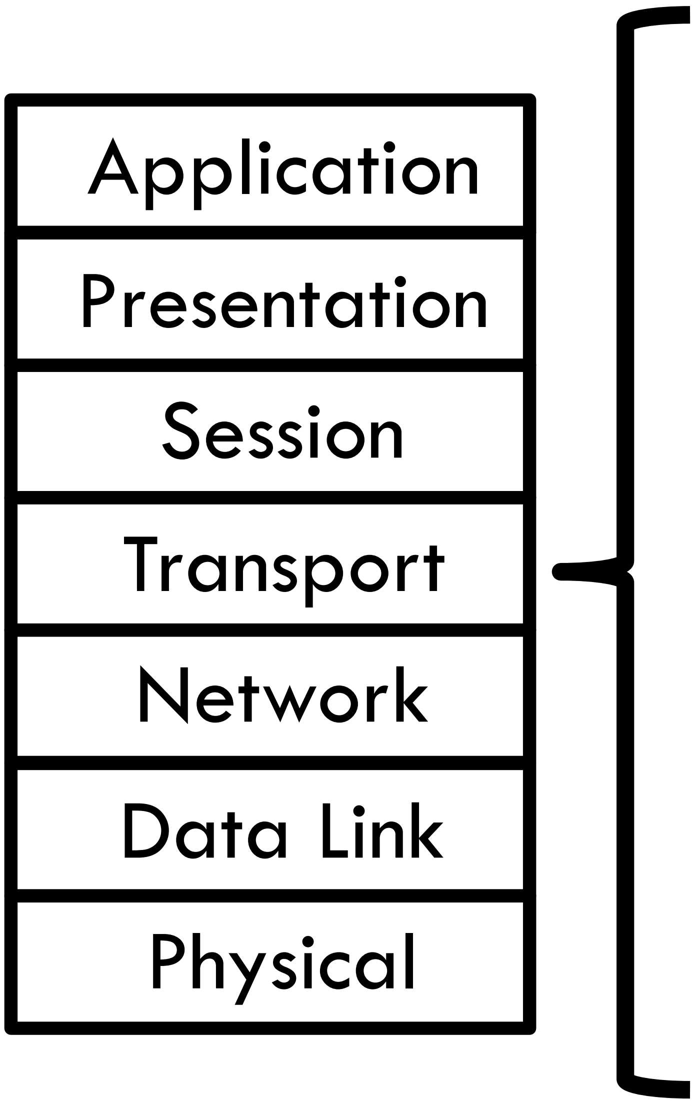
CSCI-351

Data communication and Networks

Lecture 11: Transport (UDP, but mostly TCP)

The slide is built with the help of Prof. Alan Mislove, Christo Wilson, and David Choffnes's class

Transport Layer



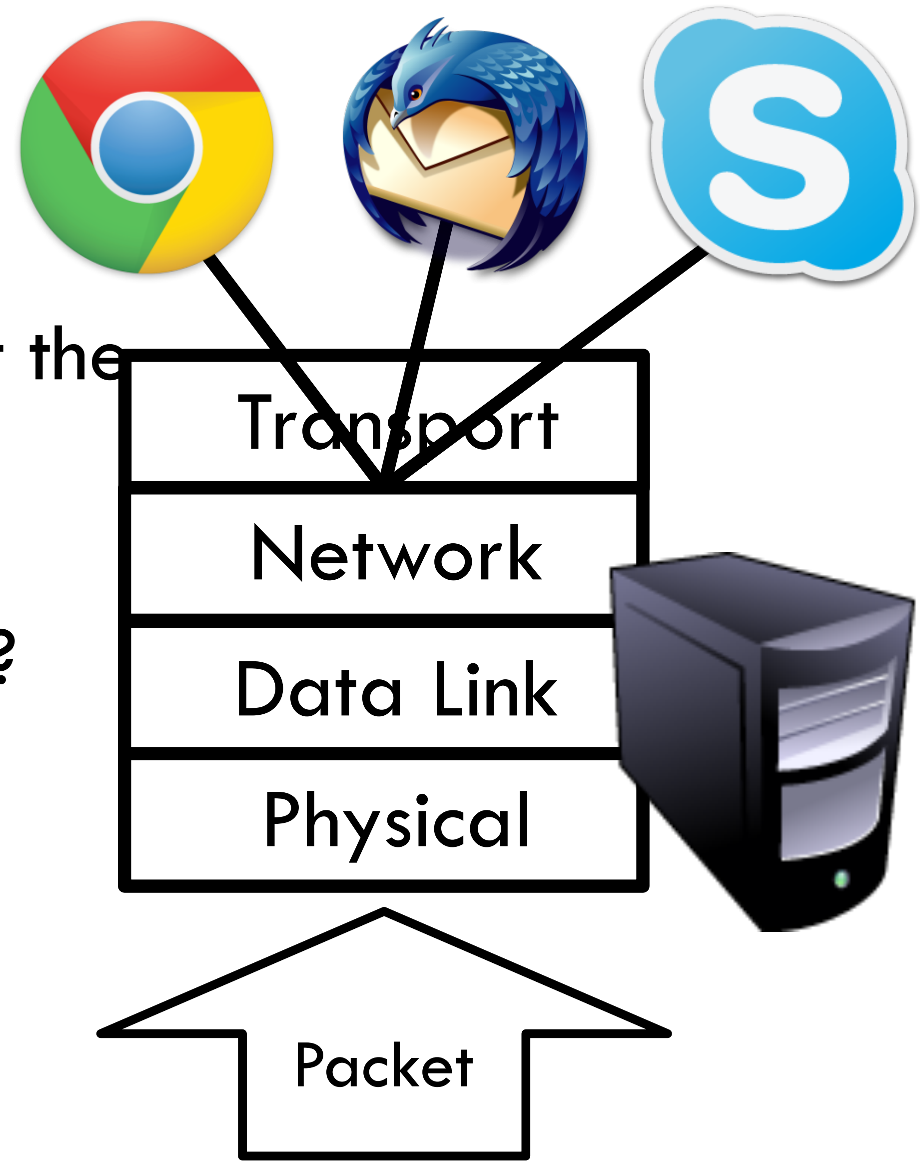
- Function:
  - Demultiplexing of data streams
- Optional functions:
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control
- Key challenges:
  - Detecting and responding to congestion
  - Balancing fairness against high utilization

Outline

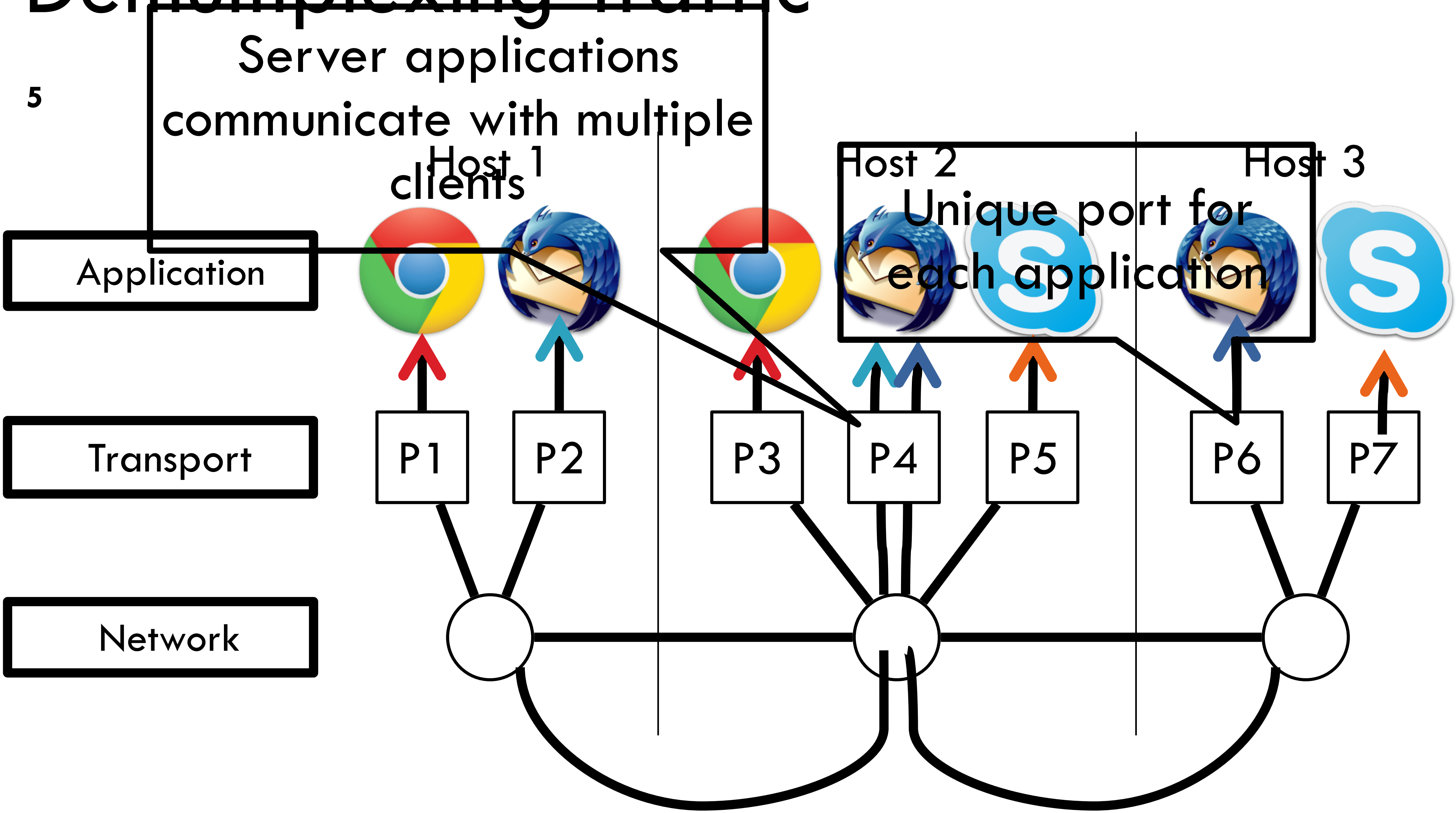
- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

The Case for Multiplexing

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

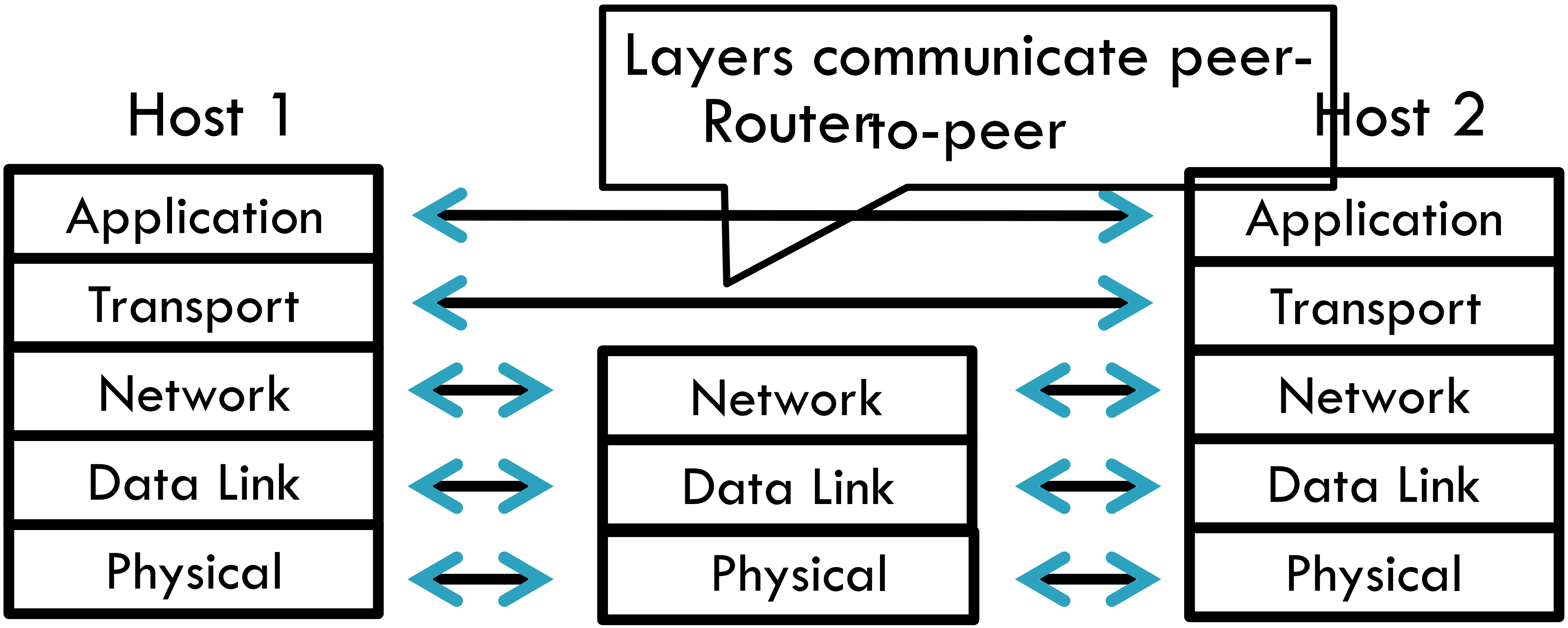


Demultiplexing Traffic



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

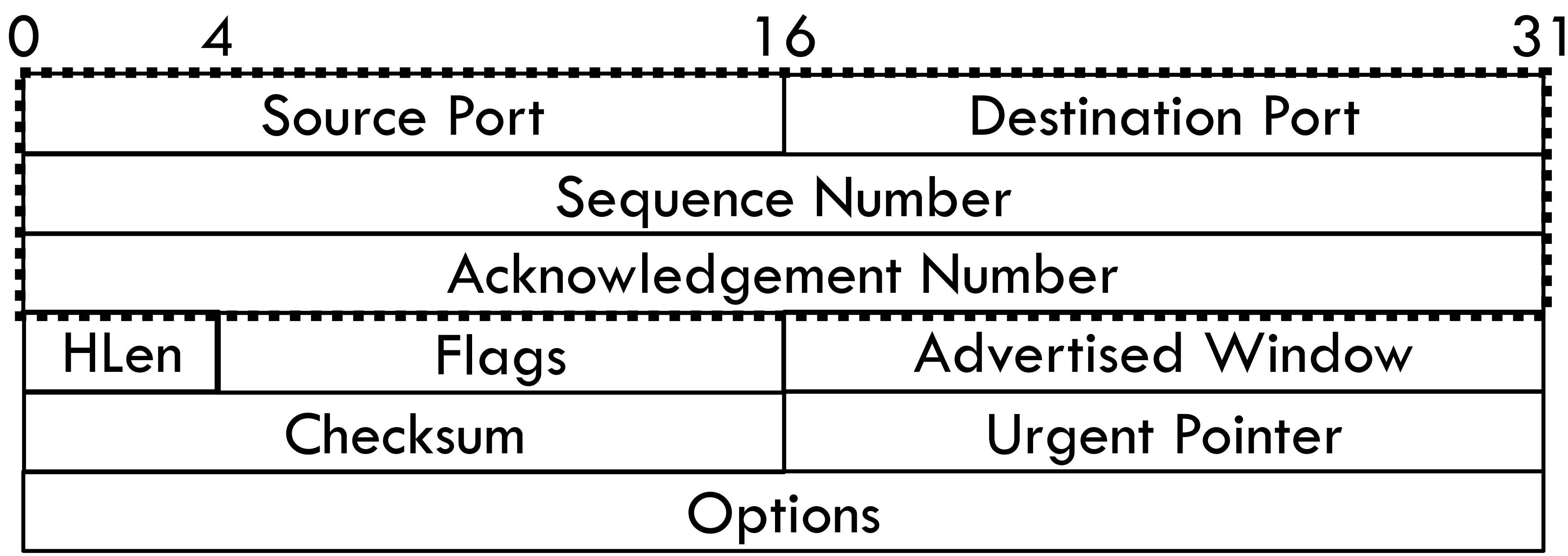
- Invented after TCP
  - Why?
- Not all applications can tolerate TCP
- Custom protocols can be built on top of UDP
  - Reliability? Strict ordering?
  - Flow control? Congestion control?
- Examples
  - RTMP, real-time media streaming (e.g. voice, video)
  - Facebook datacenter protocol
    - Why?

Outline

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

Transmission Control Protocol

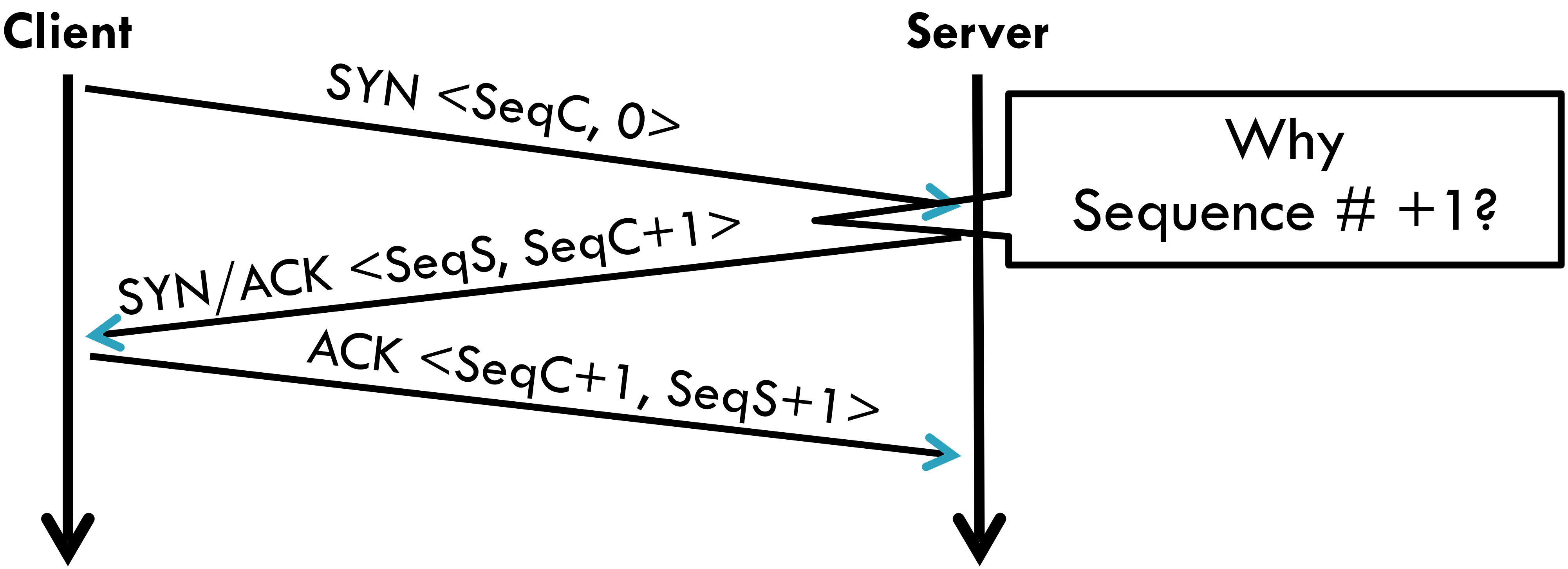
- Reliable, in-order, bi-directional byte streams
  - Port numbers for demultiplexing
  - Virtual circuits (connections)
  - Flow control
  - Congestion control, approximate fairness



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



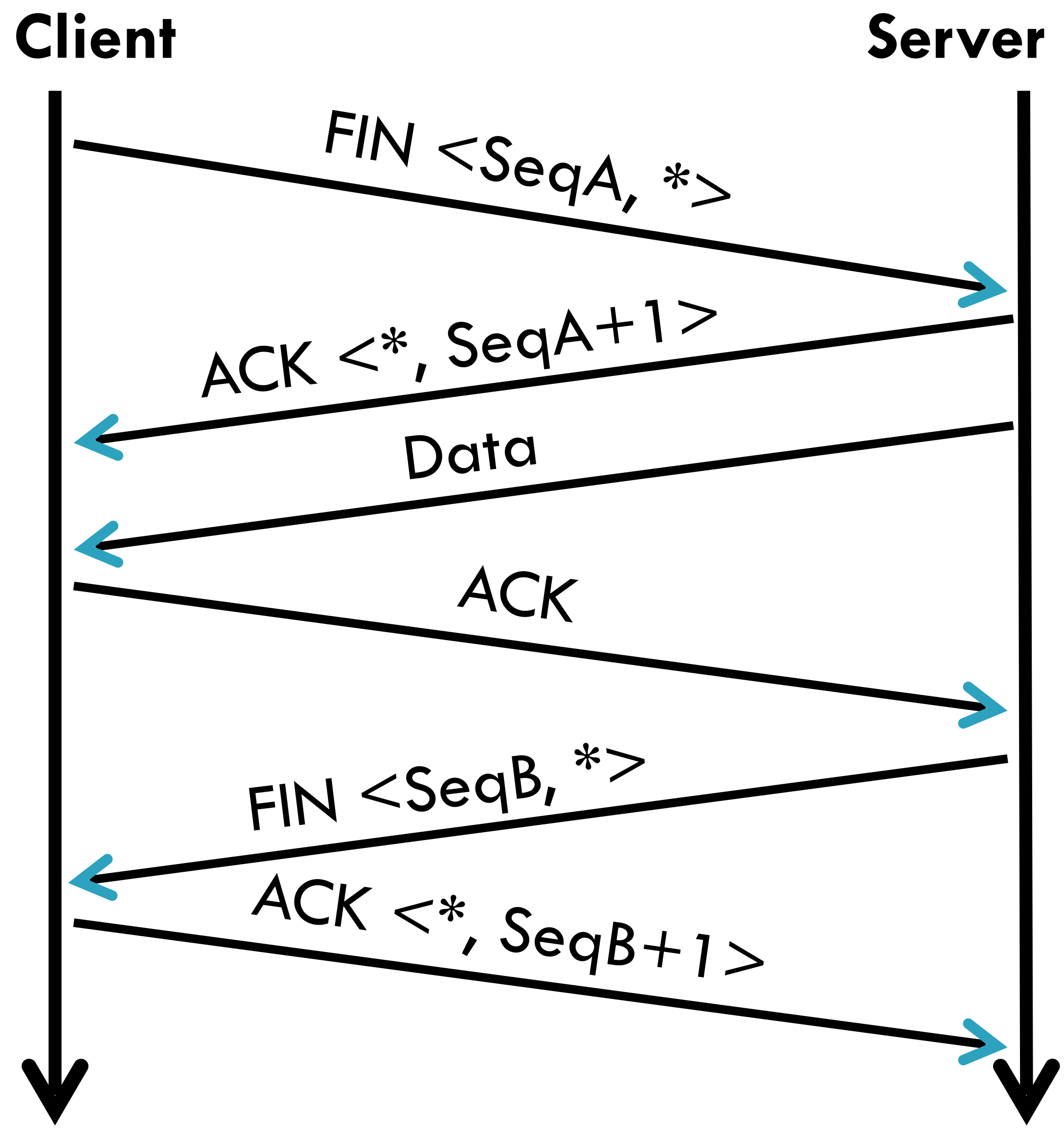
- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side's starting sequence number

Connection Setup Issues

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

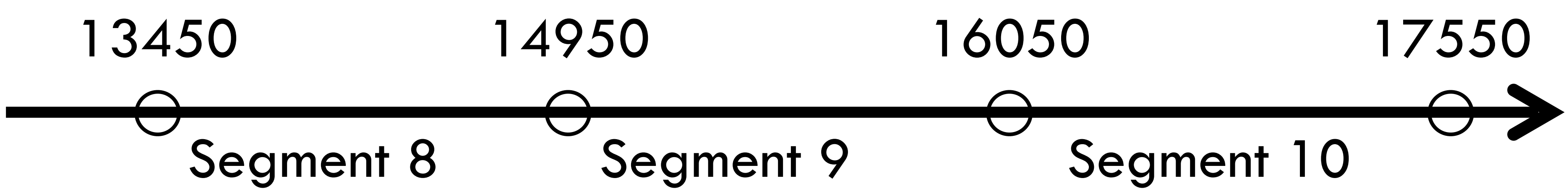
Connection Tear Down

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

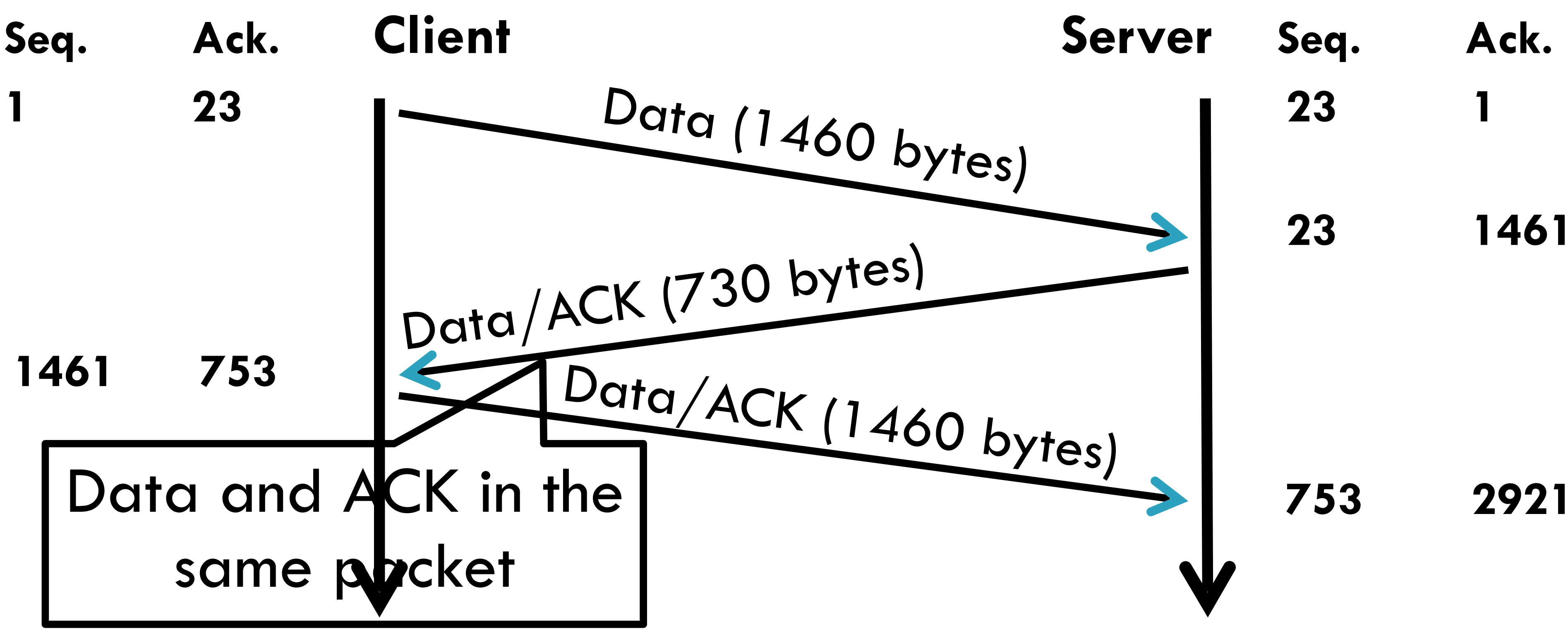


Sequence Number Space

- 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

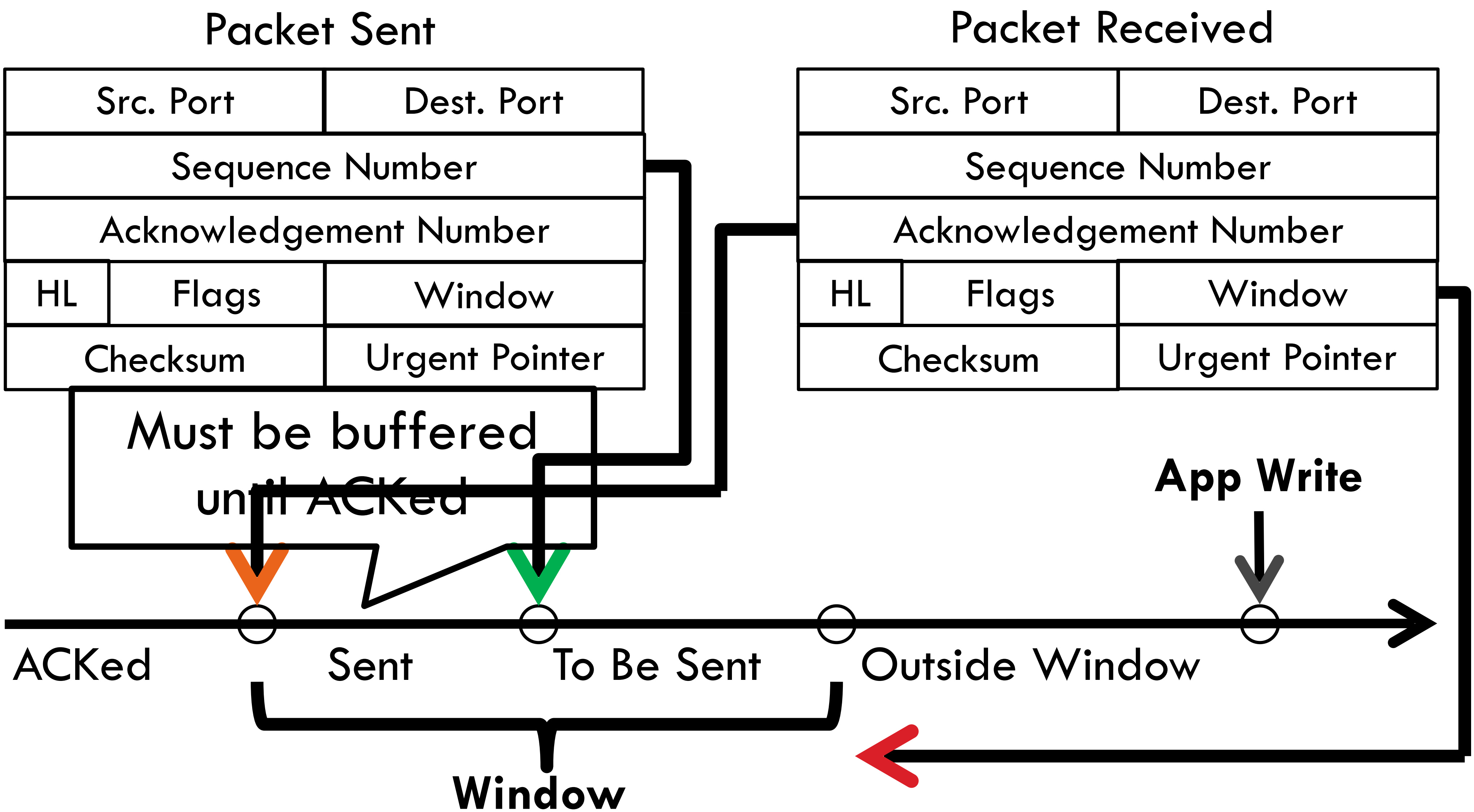


- Each side of the connection can send and receive
  - Different sequence numbers for each direction

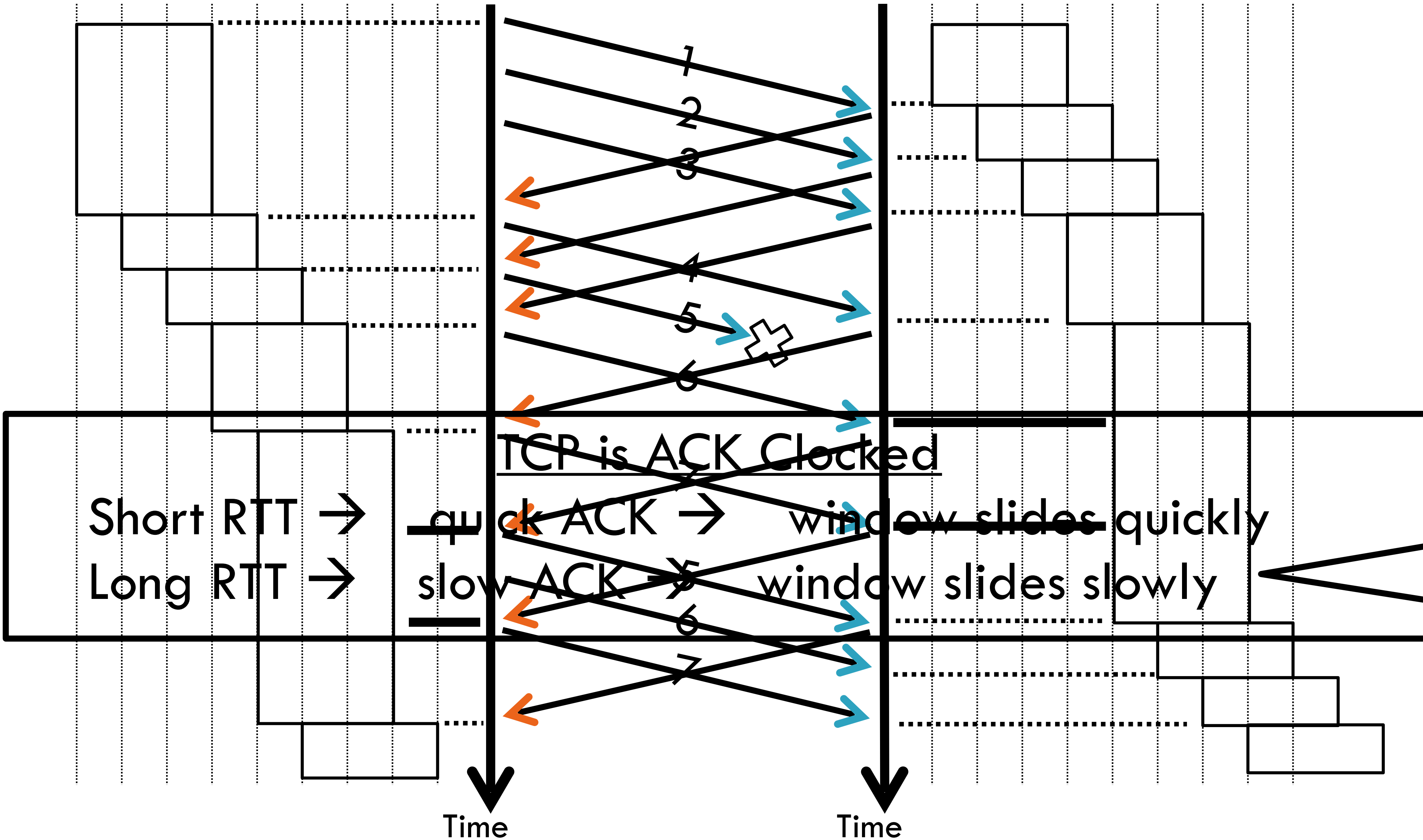
Flow Control

- 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?
- 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

Silly Window Syndrome

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

Header	Data	Header	Data	Header	Data	Header	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 (send after a timeout)

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?

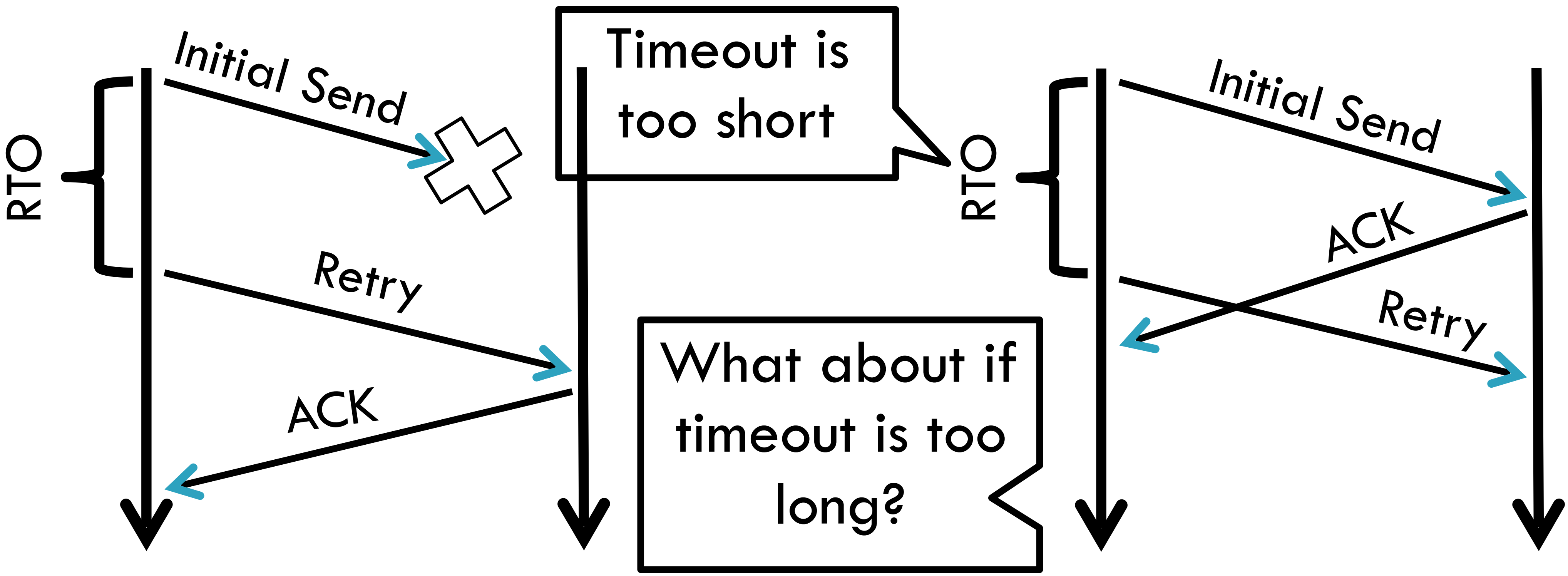
1. int flag = 1;
2. setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *) &flag, sizeof(int));

Error Detection

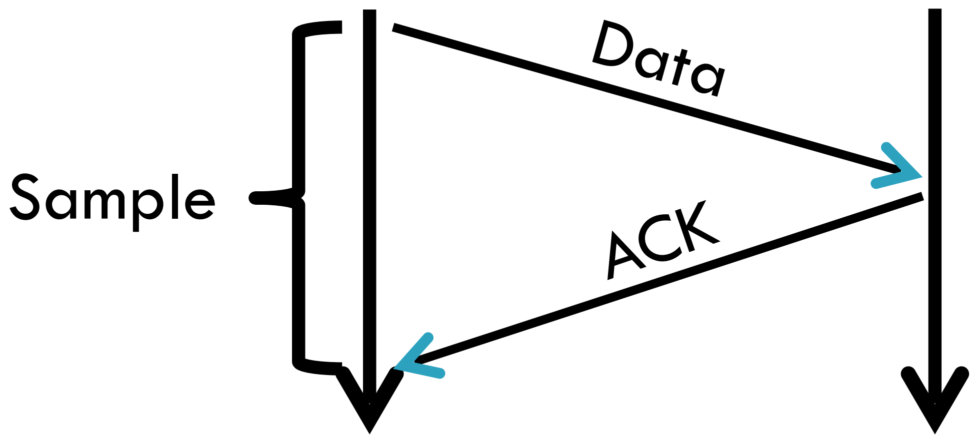
- Checksum detects (some) packet corruption
  - Computed over IP header, TCP header, and data
- Sequence numbers catch sequence problems
  - Duplicates are ignored
  - Out-of-order packets are reordered or dropped
  - Missing sequence numbers indicate lost packets
- Lost segments detected by sender
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

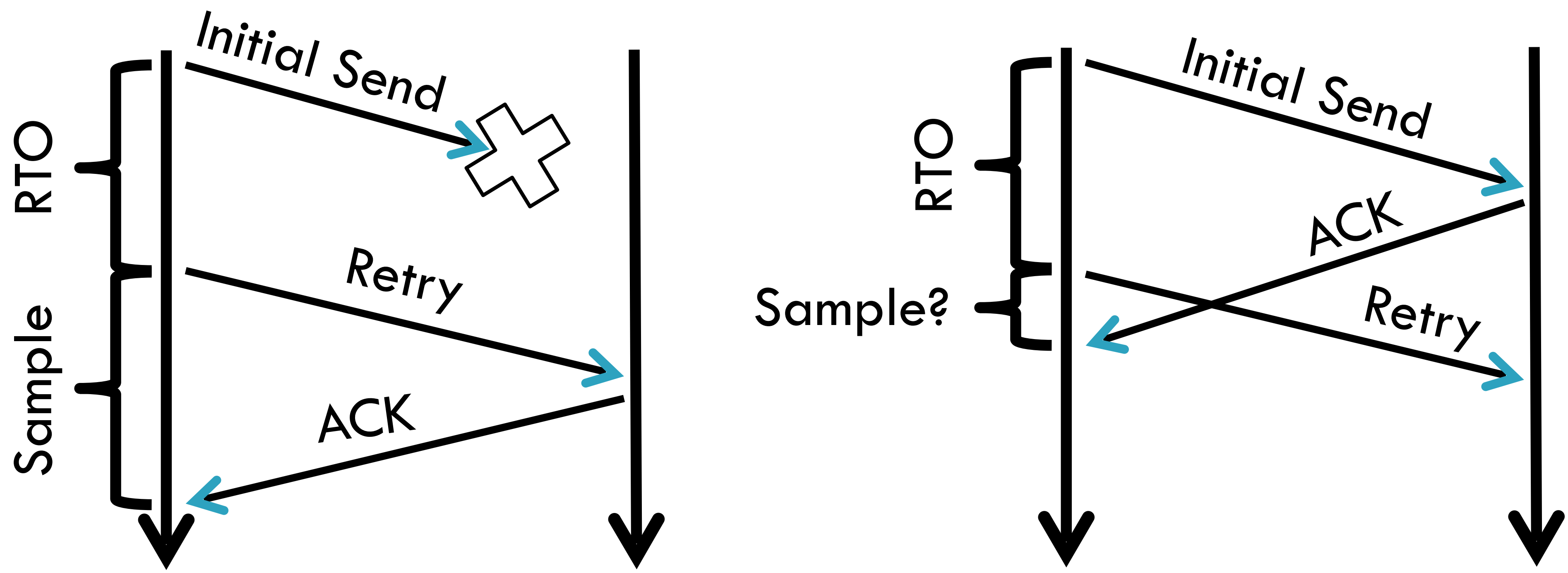


Round Trip Time Estimation



- 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

Outline

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

What is Congestion?

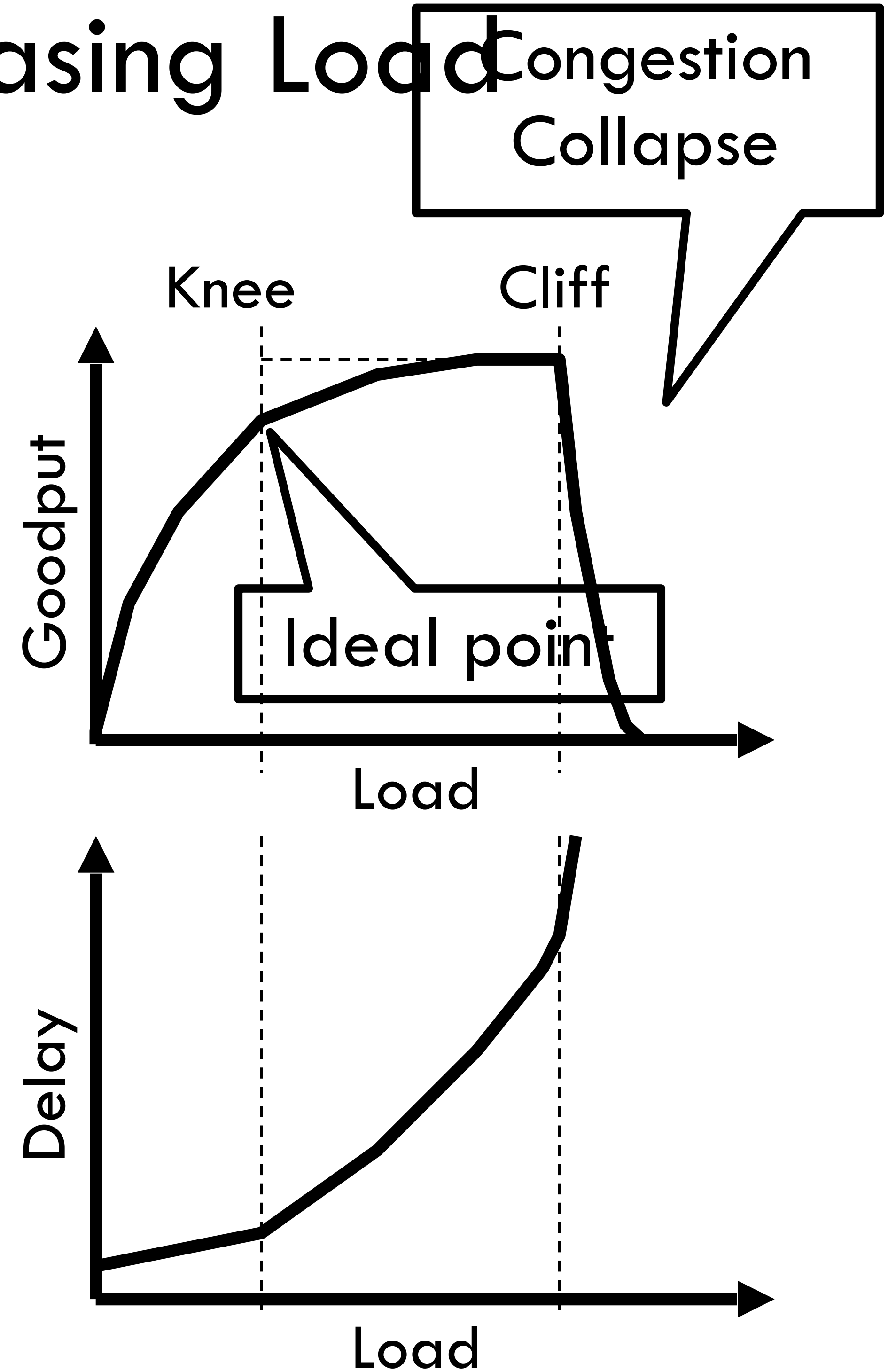
- Load on the network is higher than capacity
  - Capacity is not uniform across networks
    - Modem vs. Cellular vs. Cable vs. Fiber Optics
  - There are multiple flows competing for bandwidth
    - Residential cable modem vs. corporate datacenter
  - Load is not uniform over time
    - 10pm, Sunday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

Why is Congestion Bad?

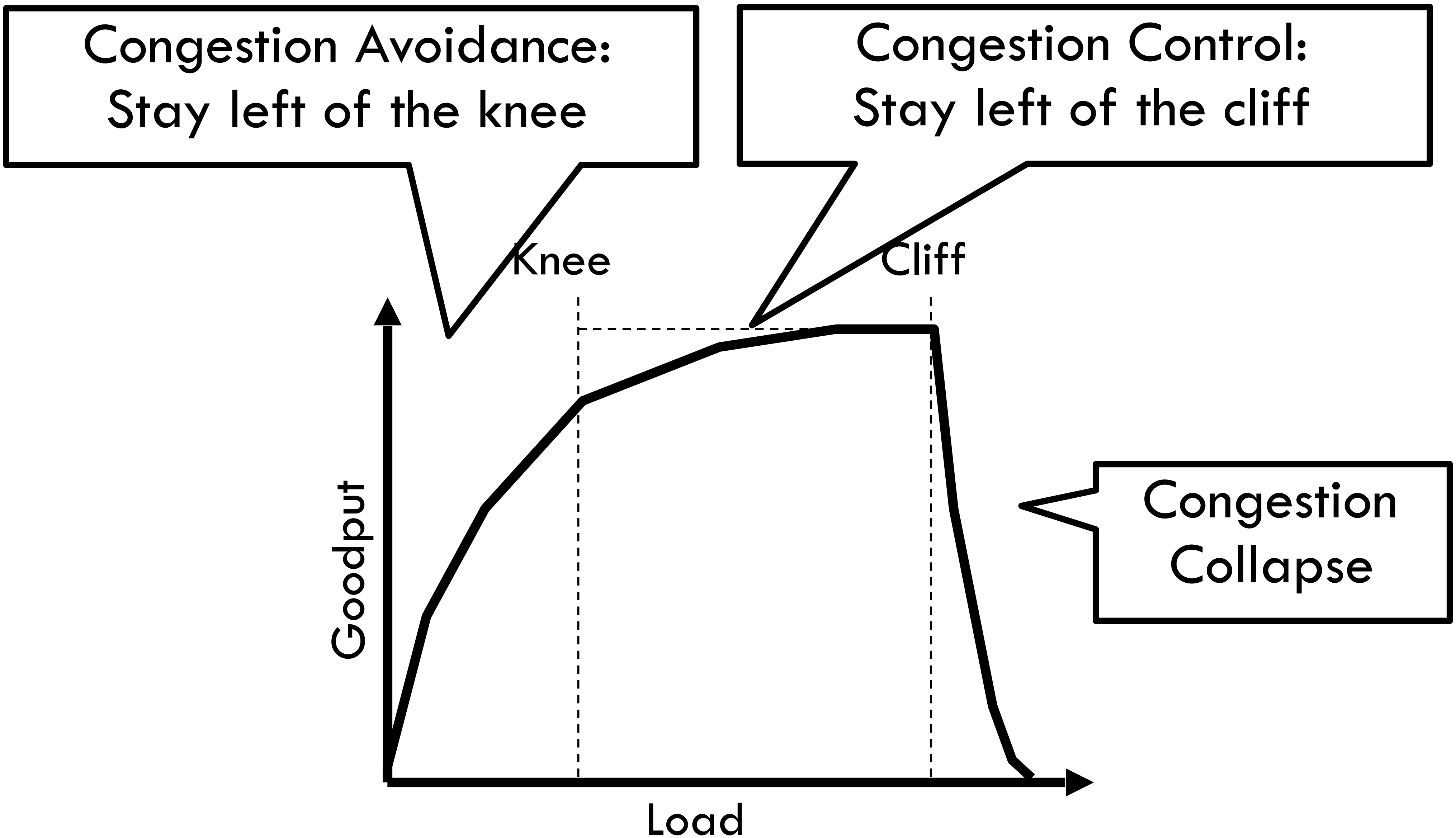
- Results in packet loss
  - Routers have finite buffers
  - Internet traffic is self similar, no buffer can prevent all drops
  - When routers get overloaded, packets will be dropped
- Practical consequences
  - Router queues build up, delay increases
  - Wasted bandwidth from retransmissions
  - Low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

The Danger of Increasing Load

- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- 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

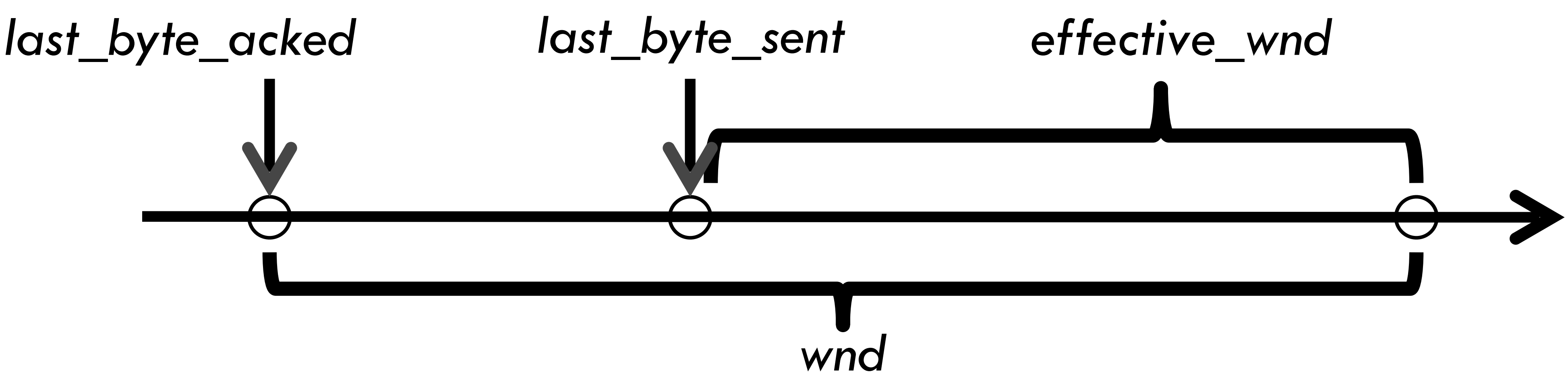
TCP Congestion Control

- 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

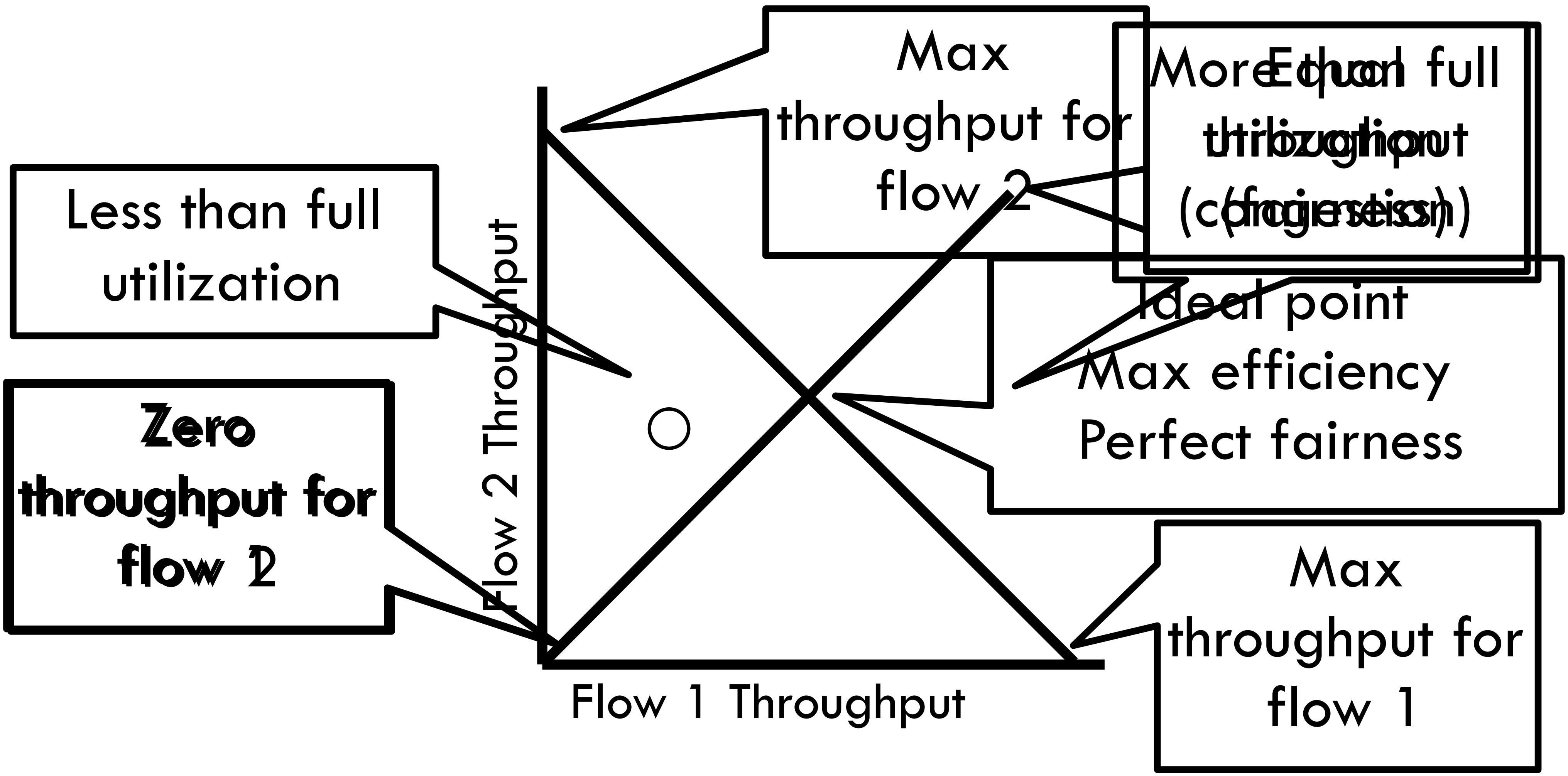
1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)
2. Rate adjustment algorithm
   - Modify *cwnd*
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

Except on wireless networks

Rate Adjustment

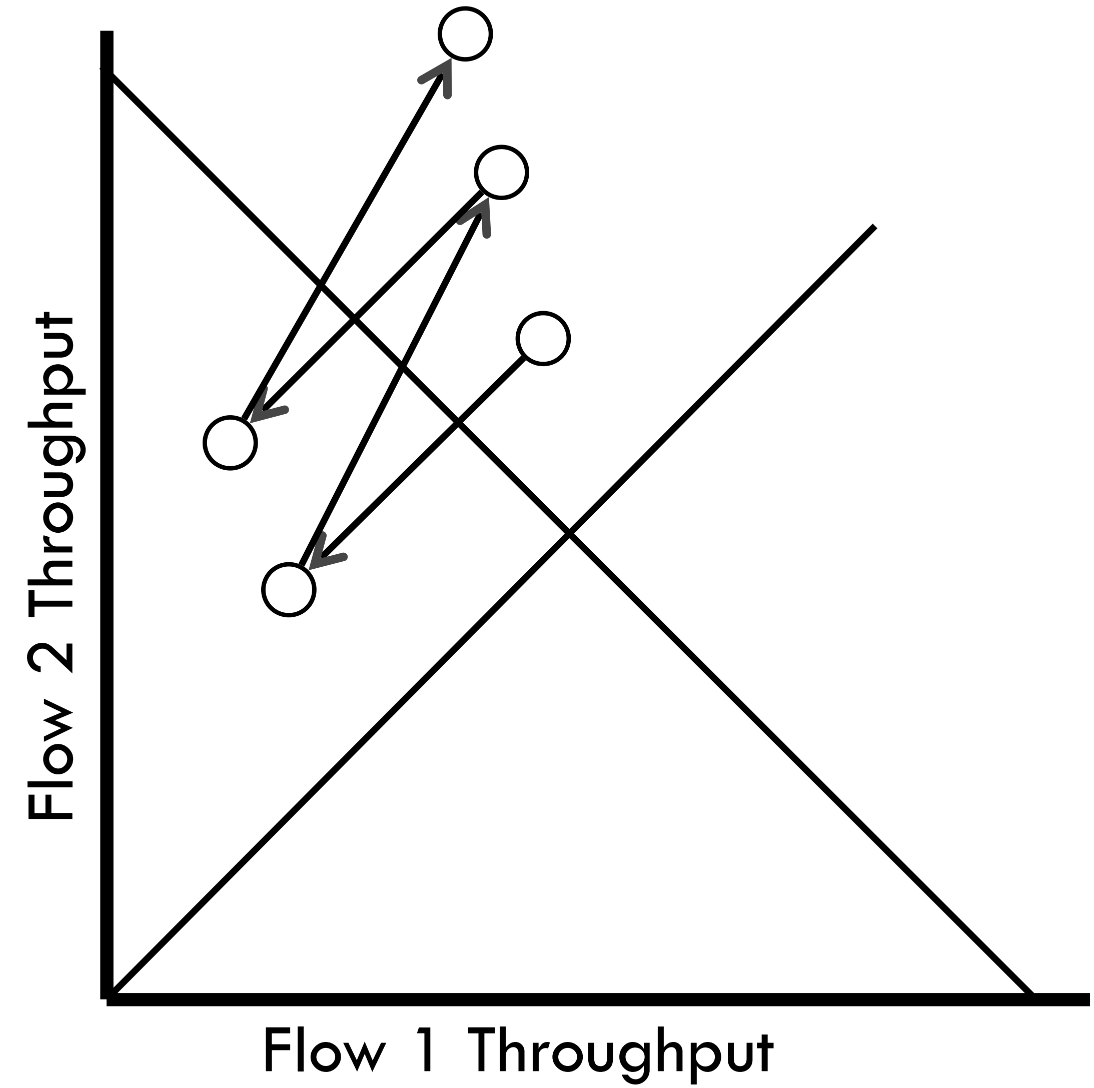
- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly
- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - *cwnd* growth is proportional to RTT
  - On loss: decrease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

Utilization and Fairness



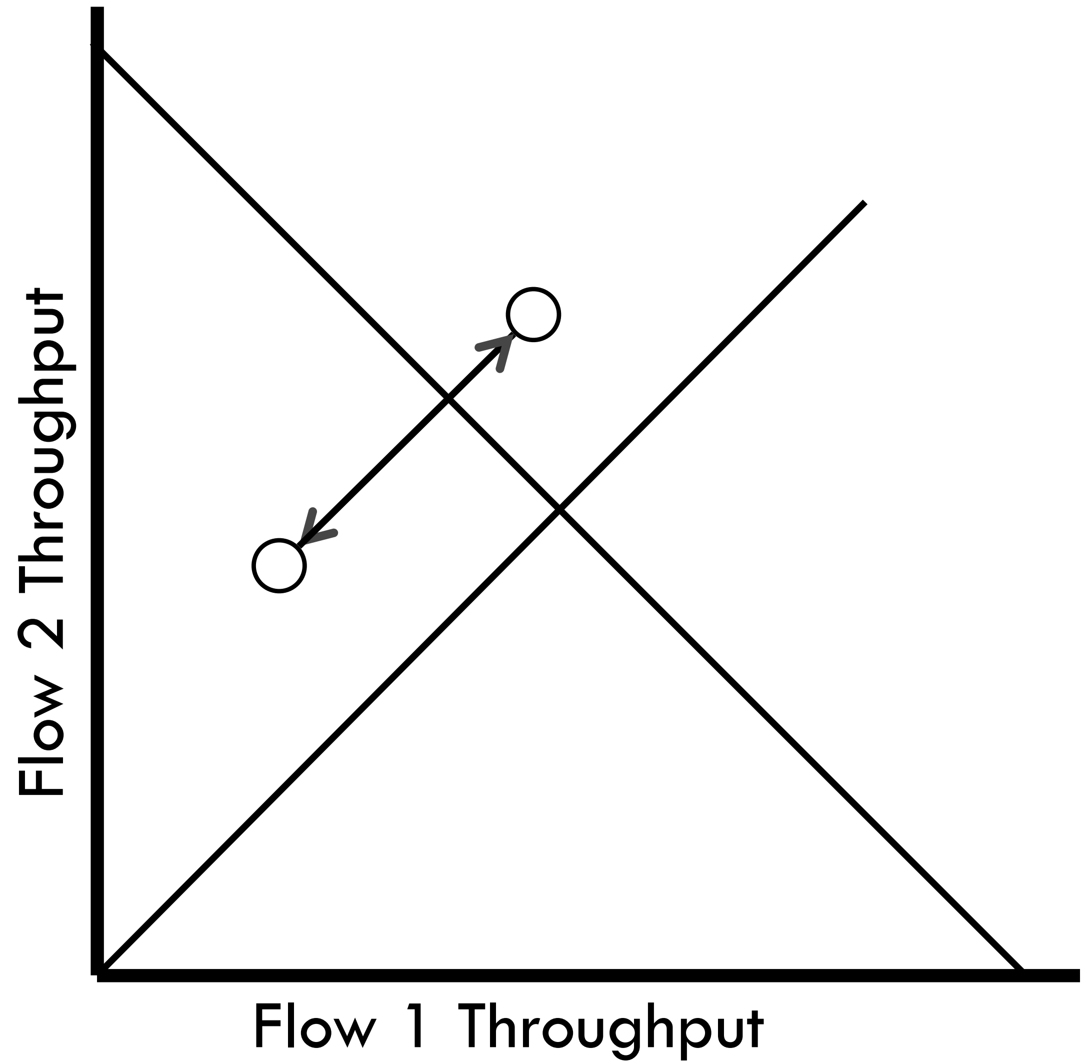
Multiplicative Increase, Additive Decrease

- Not stable!
- Veers away from fairness



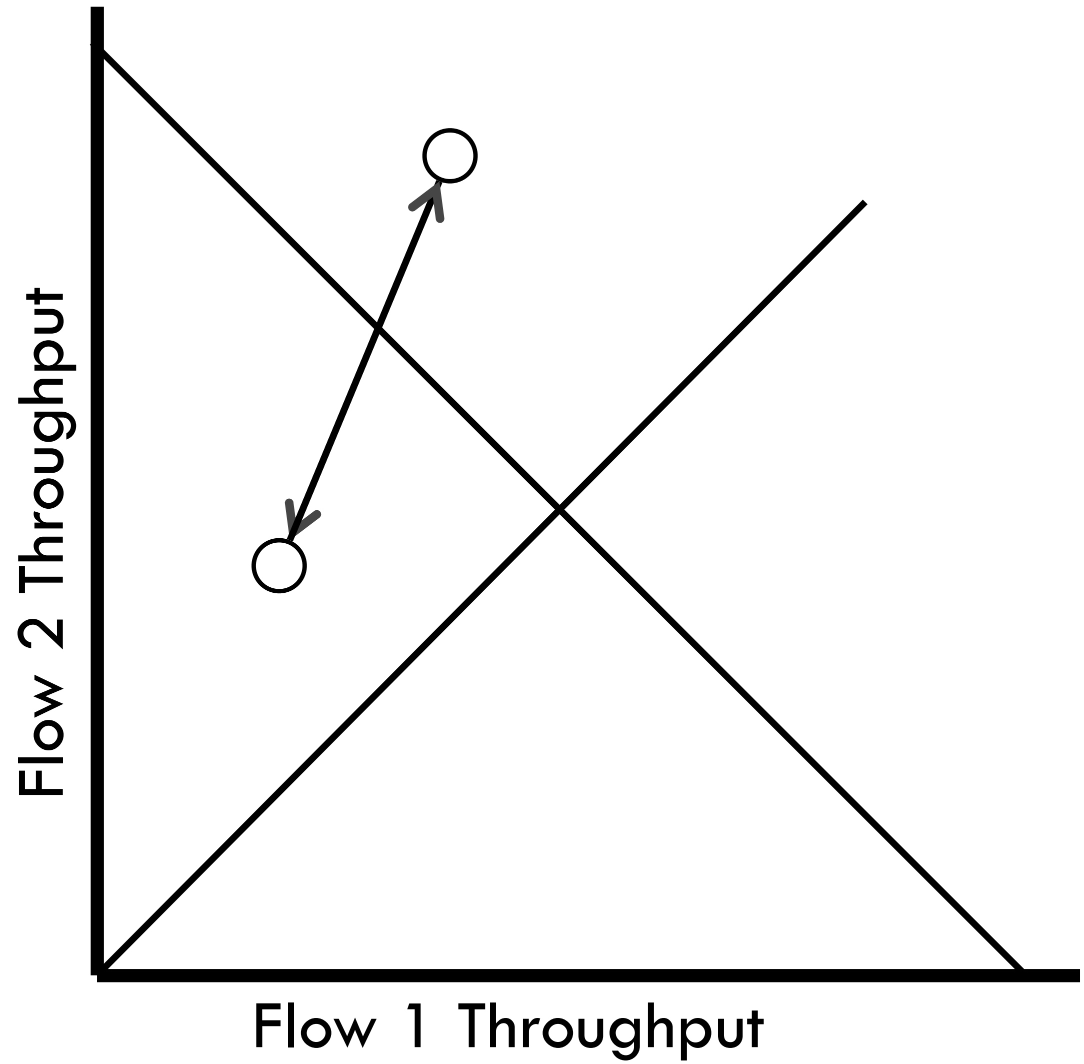
Additive Increase, Additive Decrease

- Stable
- But does not converge to fairness



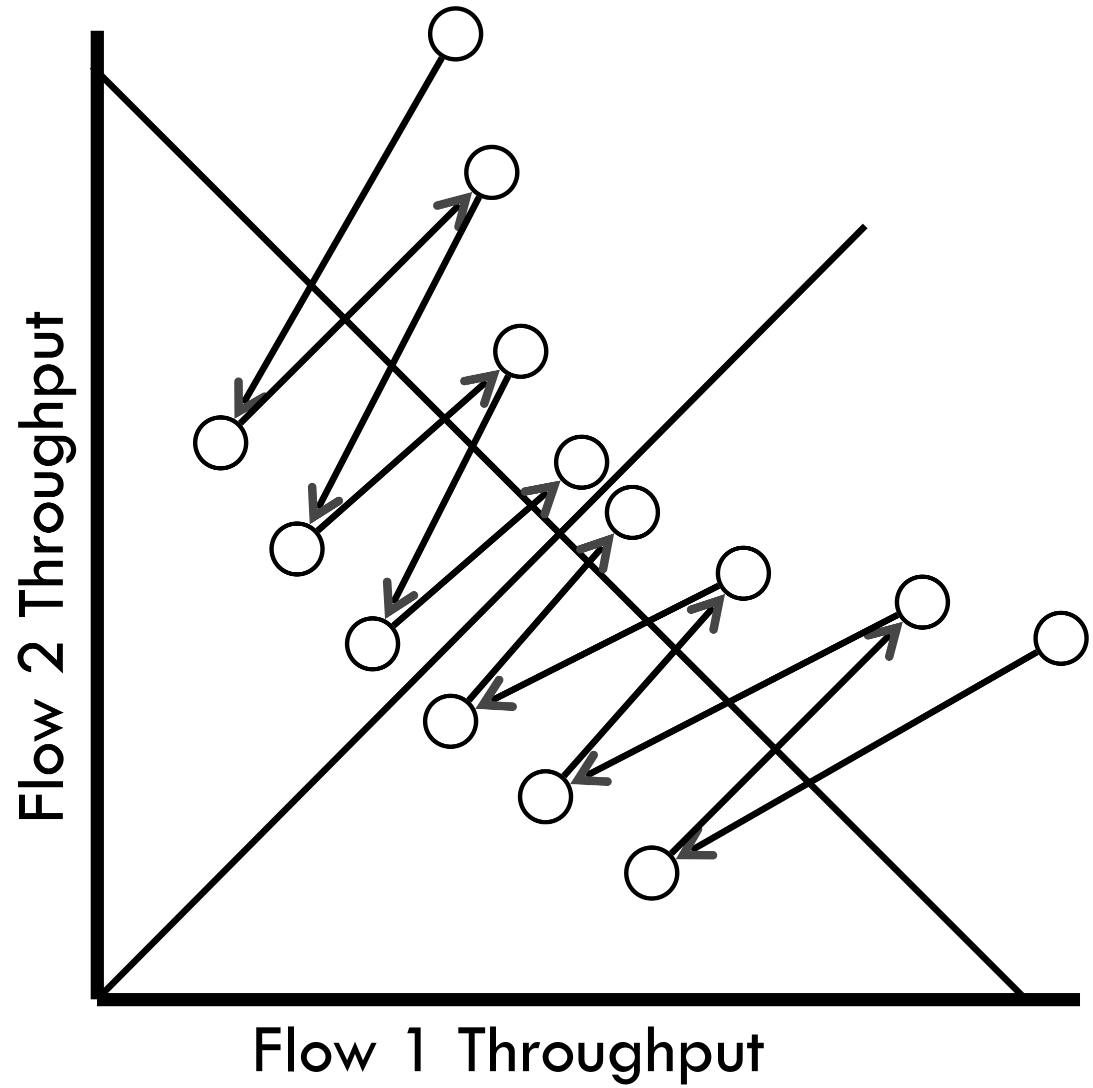
Multiplicative Increase, Multiplicative Decrease

- Stable
- But does not converge to fairness



Additive Increase, Multiplicative Decrease

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

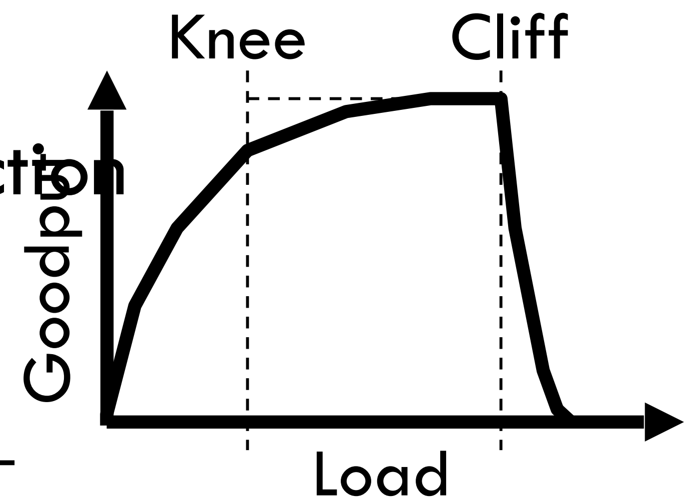


Implementing Congestion Control

- 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

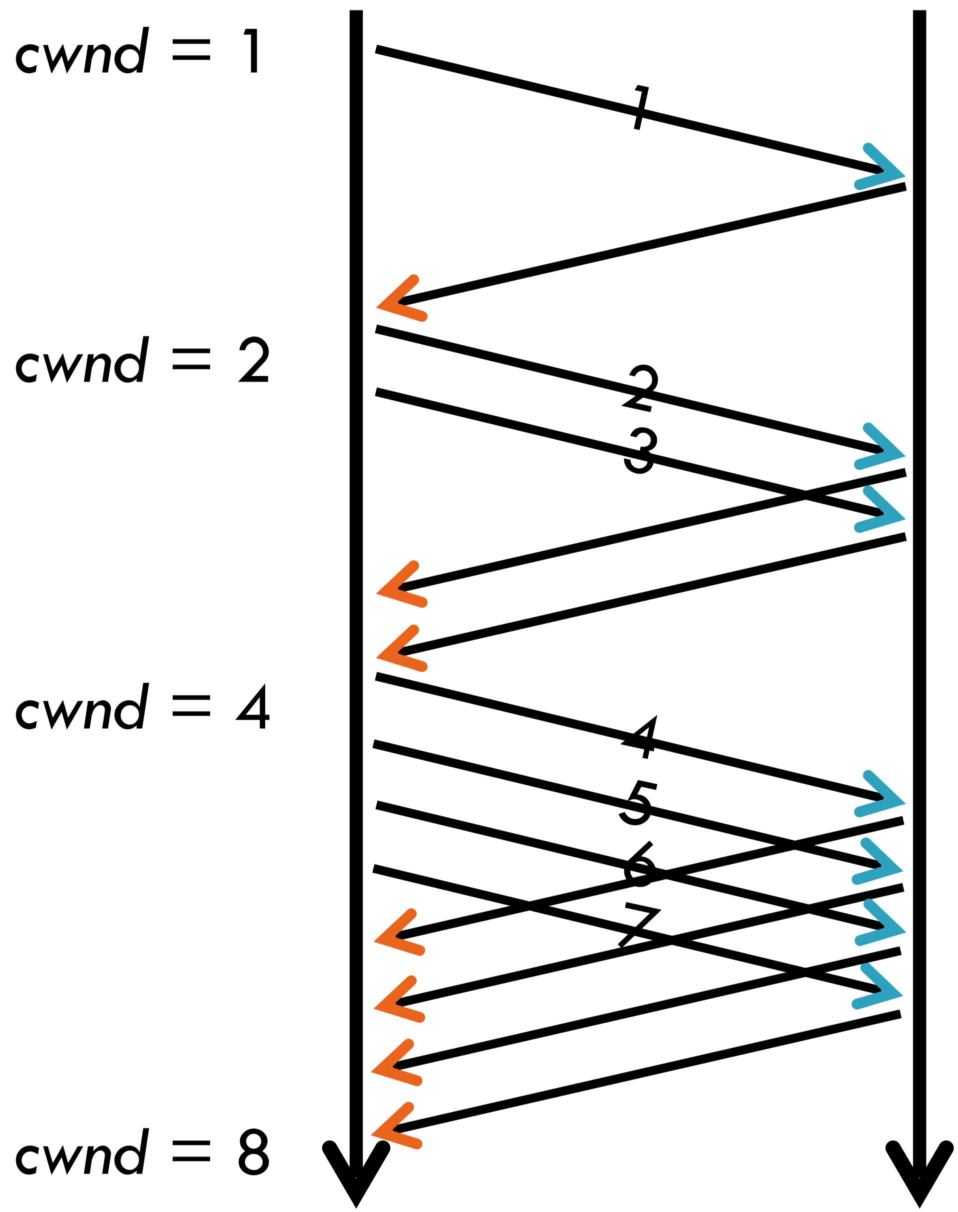
Slow Start

- 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

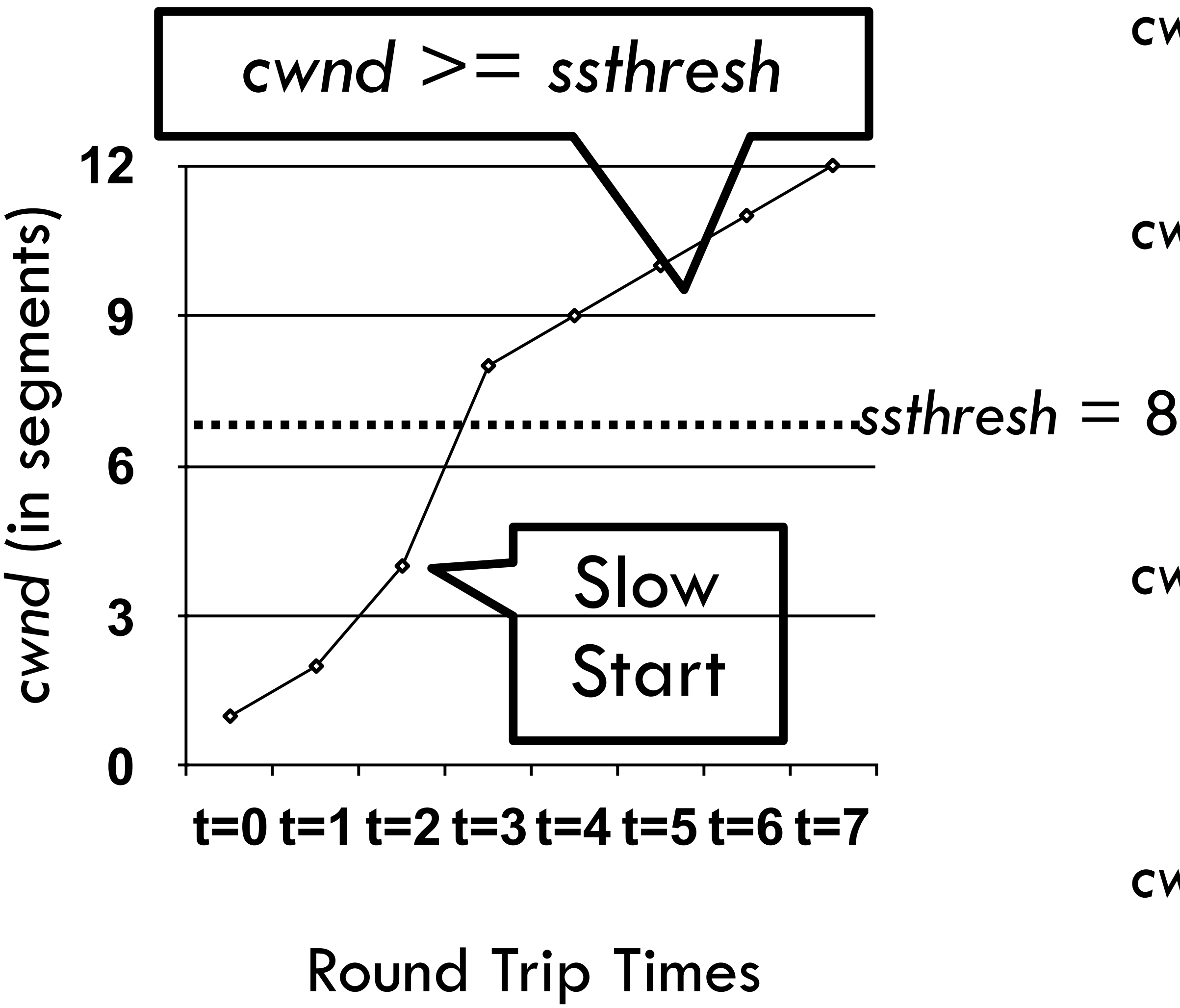
- *cwnd* grows rapidly
- Slows down when…
  - *cwnd* >= *ssthresh*
  - Or a packet drops

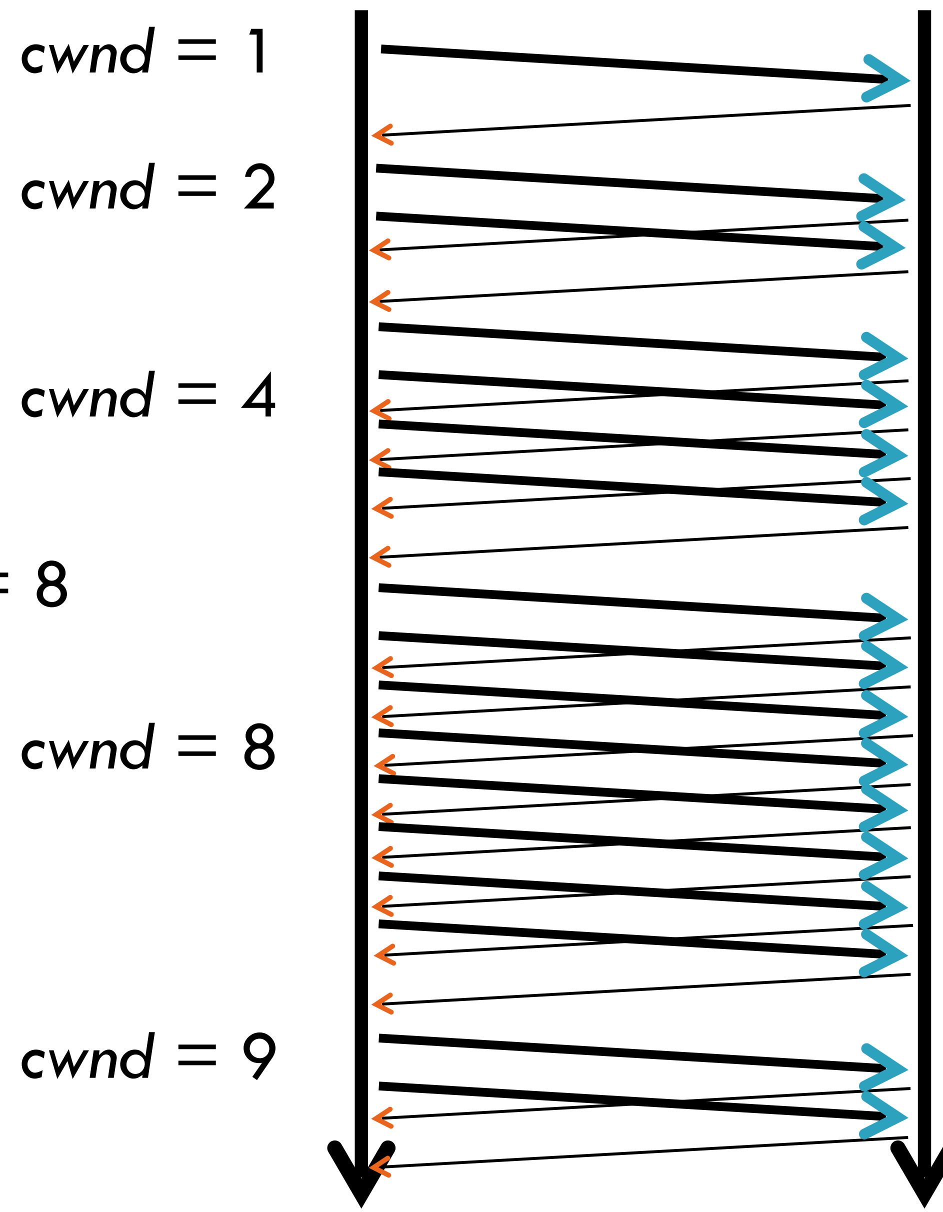


Congestion Avoidance

- 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:

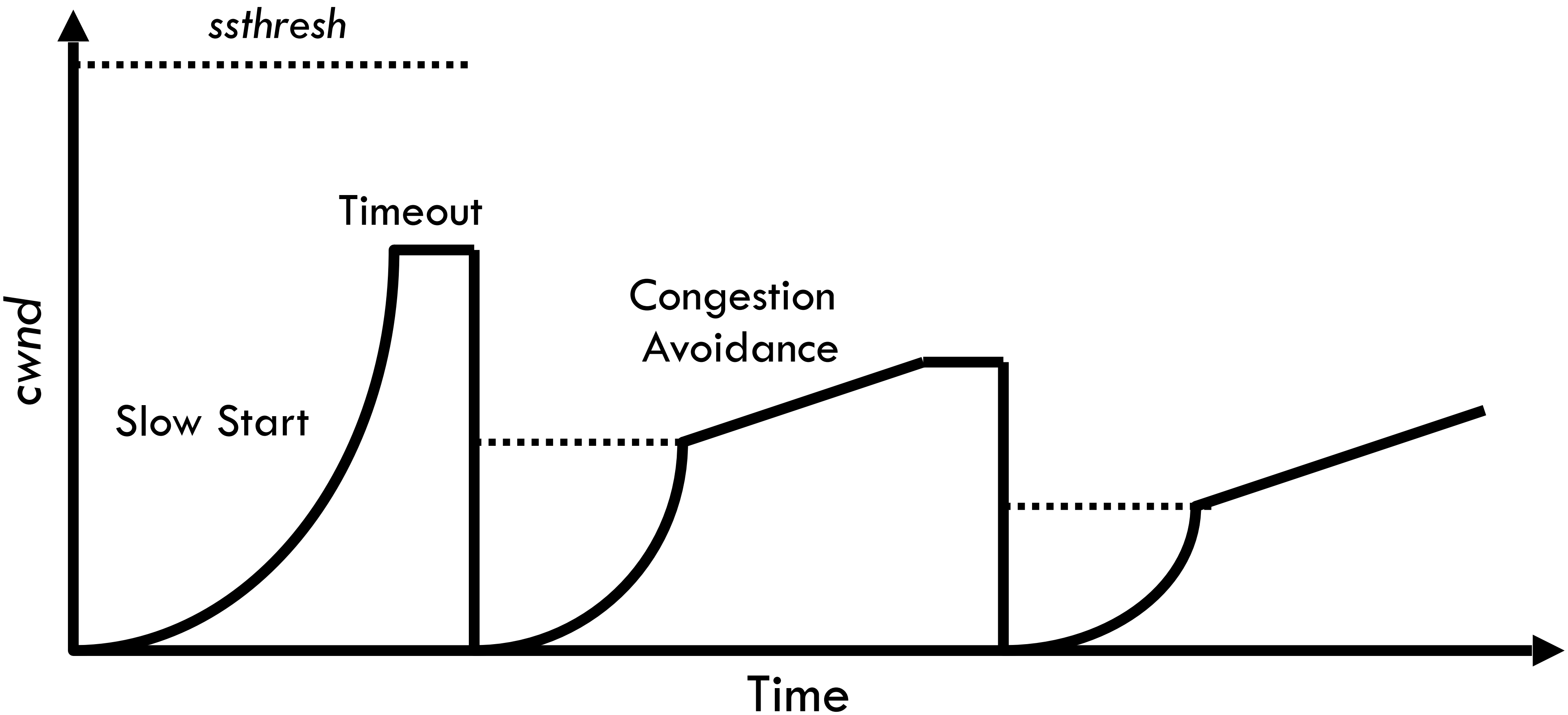
```
if (cwnd < ssthresh)
    /* Slow Start*/
    cwnd = cwnd + 1;
else
    /* Congestion Avoidance */
    cwnd = cwnd + 1/cwnd;
```

Timeout:

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

The Big Picture

51



Outline

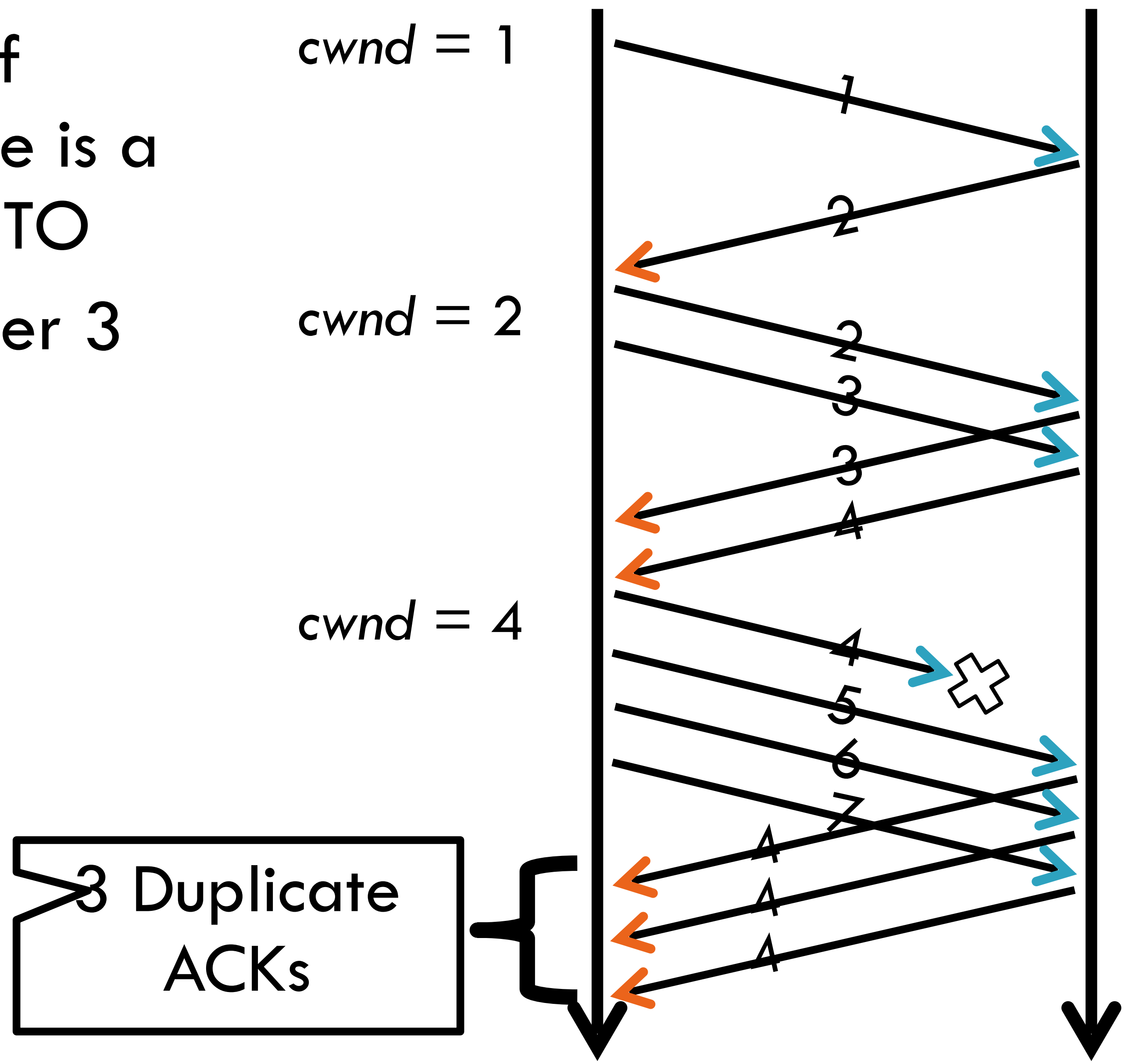
- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

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

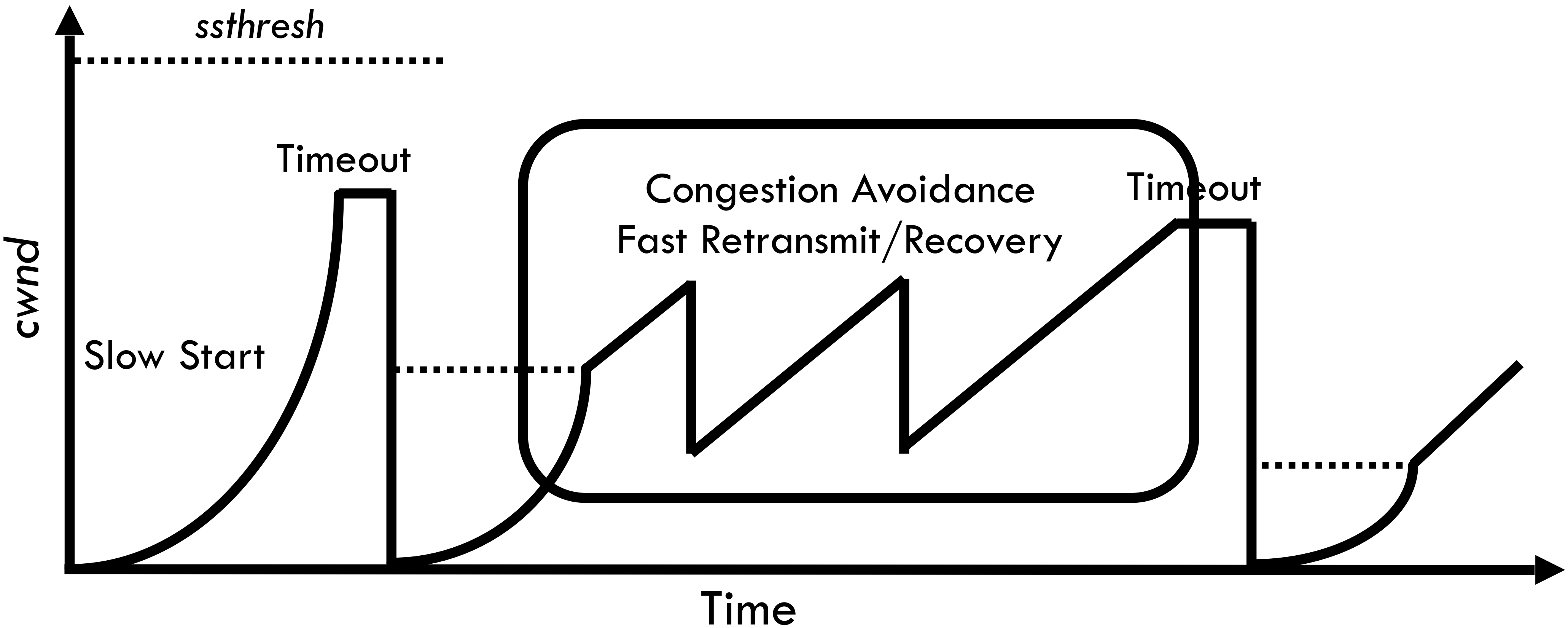
- Problem: in Tahoe, if segment is lost, there is a long wait until the RTO
- Reno: retransmit after 3 duplicate ACKs



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

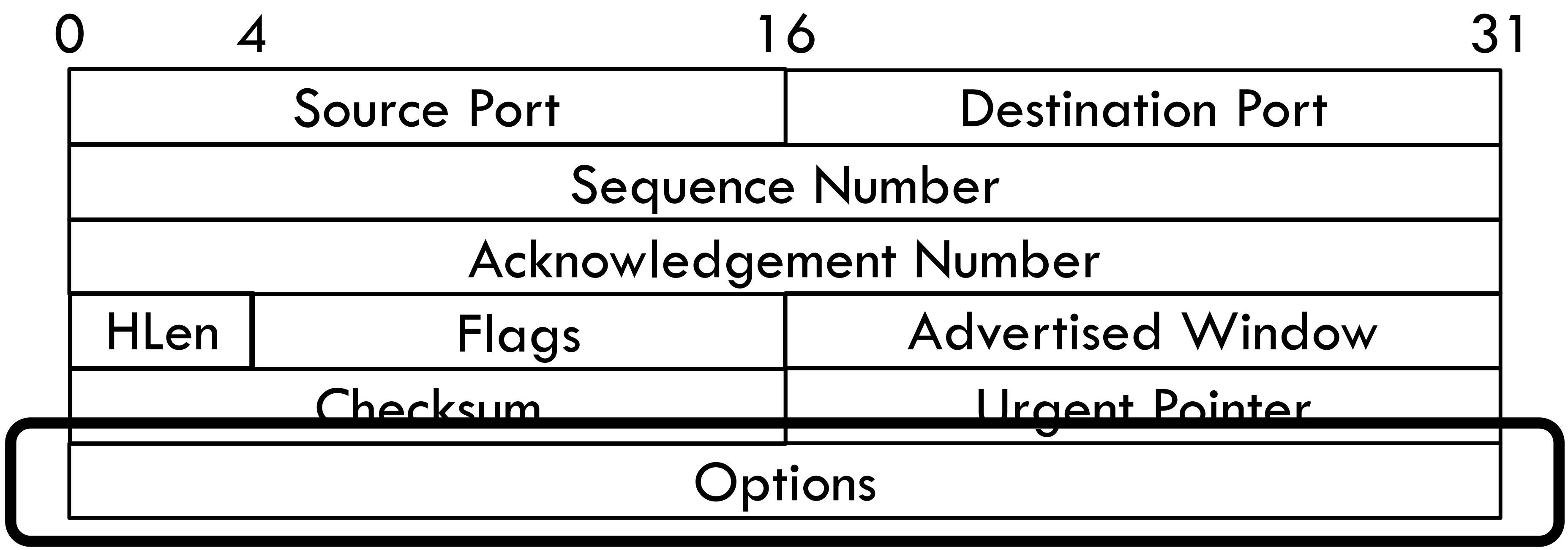
Many TCP Variants…

- 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

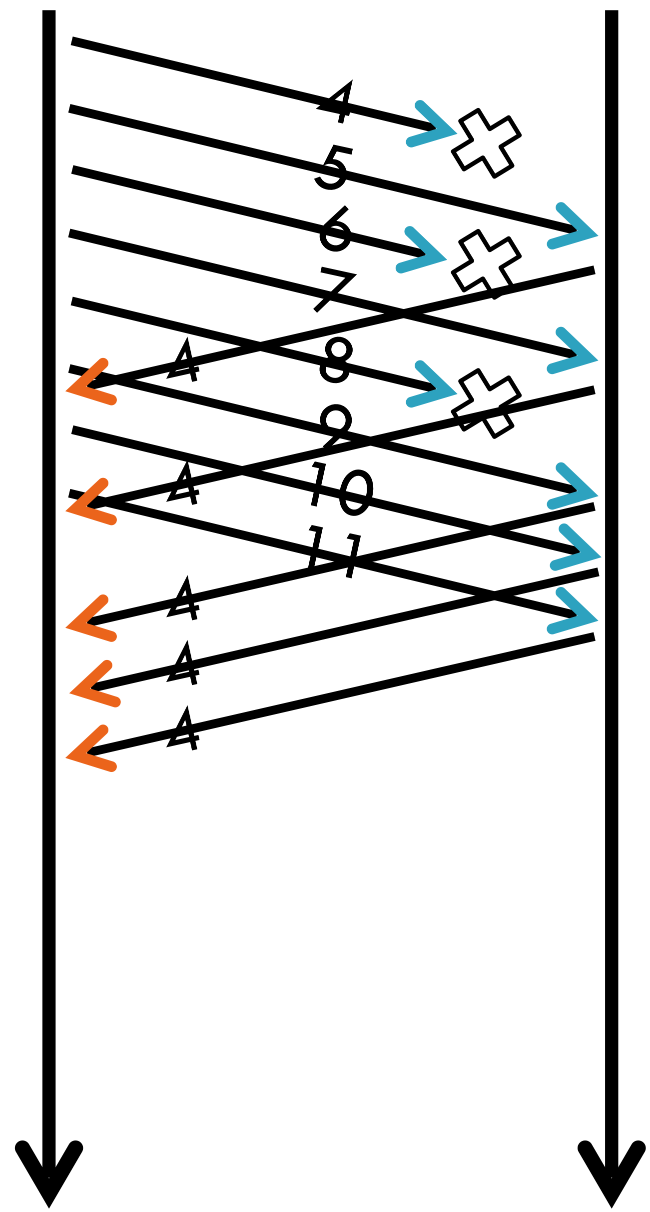
Common TCP Options



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

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

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

SYN Cookies

0	5	8	31
Timestamp	MSS	Crypto Hash of Client IP & Port	

- 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.