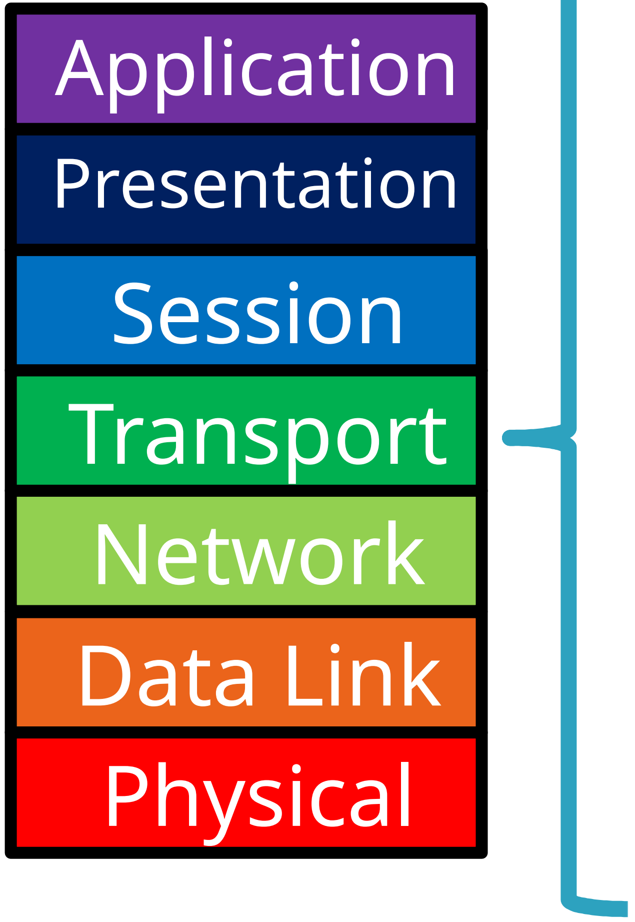
Computer Networks

Lecture 9: Transport layer Part I

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

Transport Layer



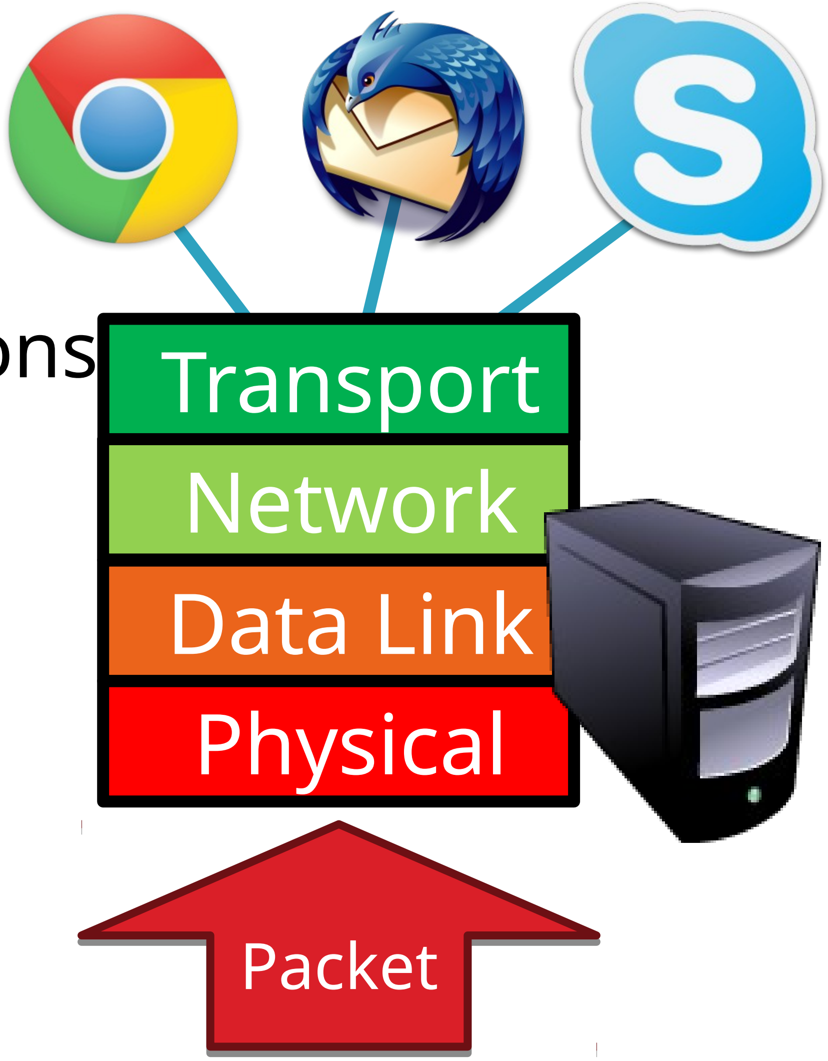
- Function:
  - Demultiplexing of data streams
- Optional functions:
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control
- Key challenges:
  - Detecting and responding to congestion
  - Balancing fairness against high utilization

Outline

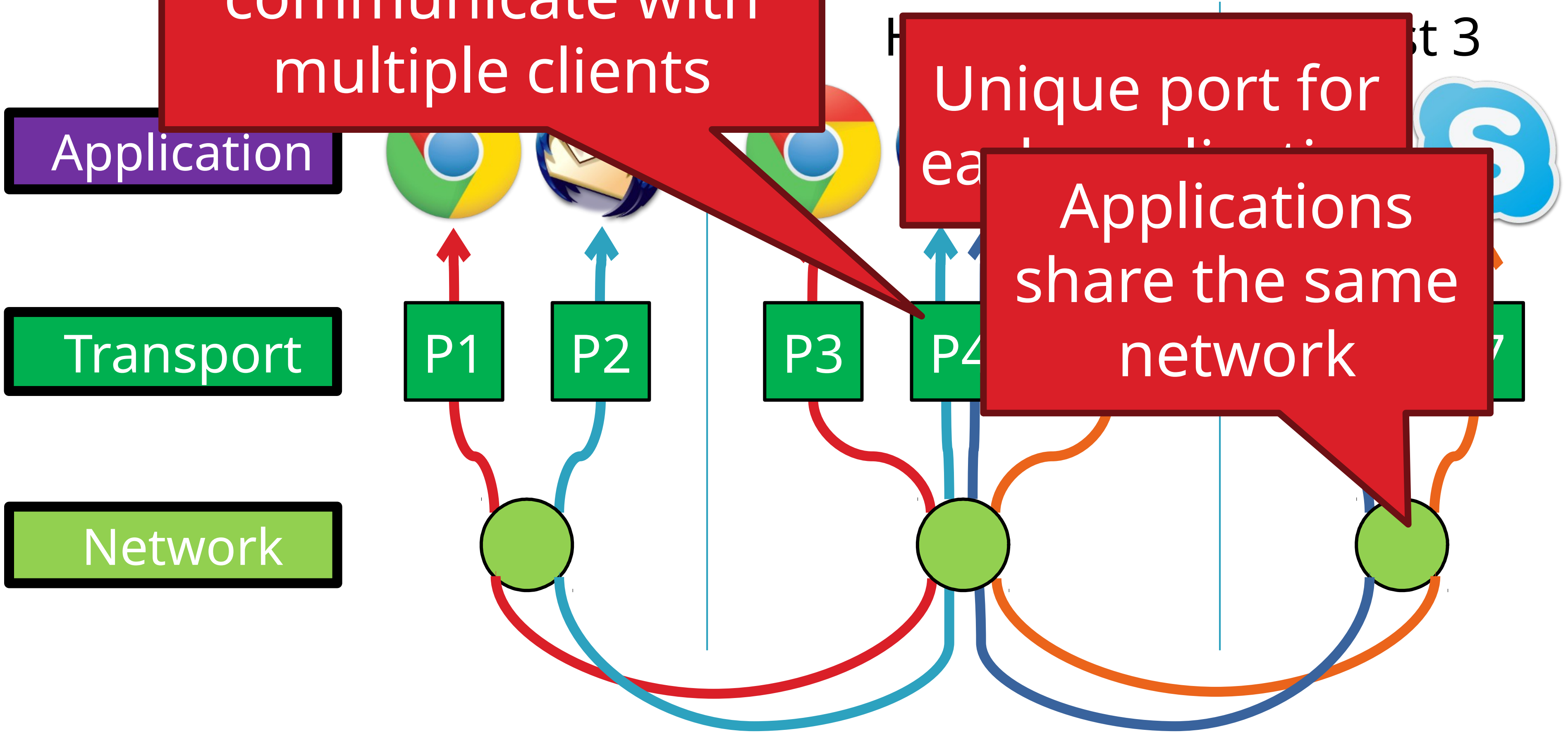
- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

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

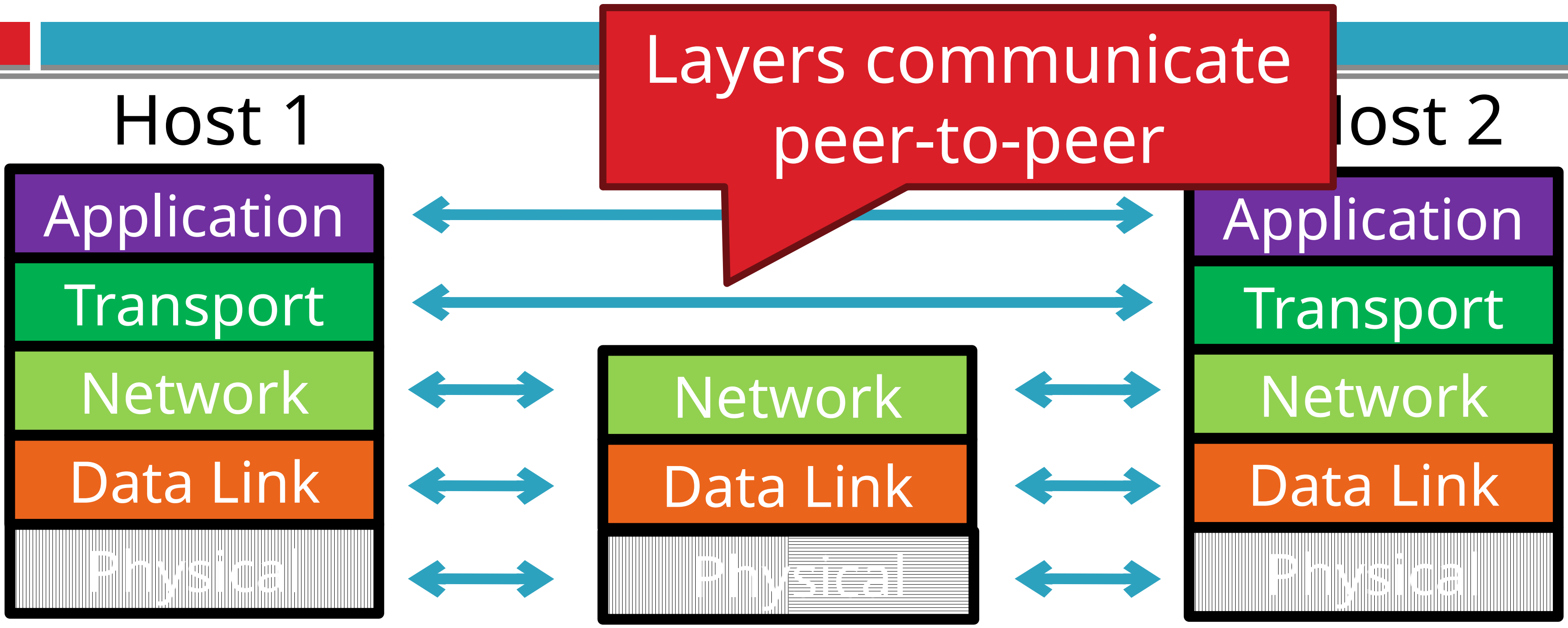


Demultiplexing Traffic


Application
Transport
Network
P1
P2
P3
P4
7
t 3
Server applications communicate with multiple clients
Unique port for ea
Applications share the same network

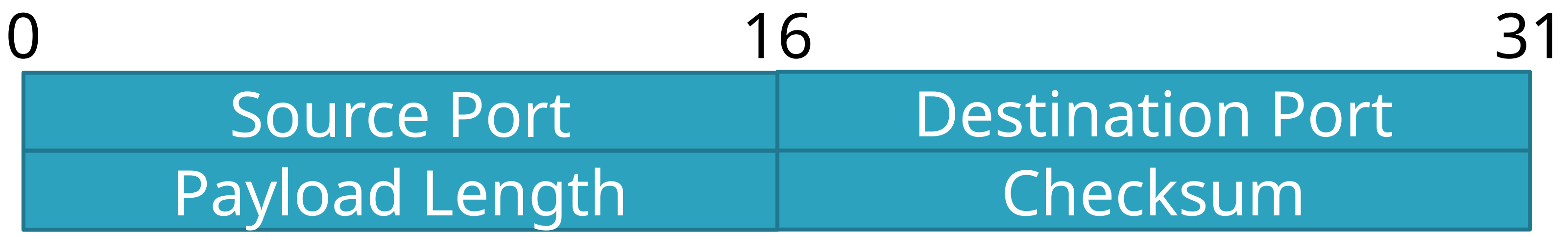
Endpoints identified by <*src_ip*, *src_port*, *dest_ip*, *dest_port*>

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)



- 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

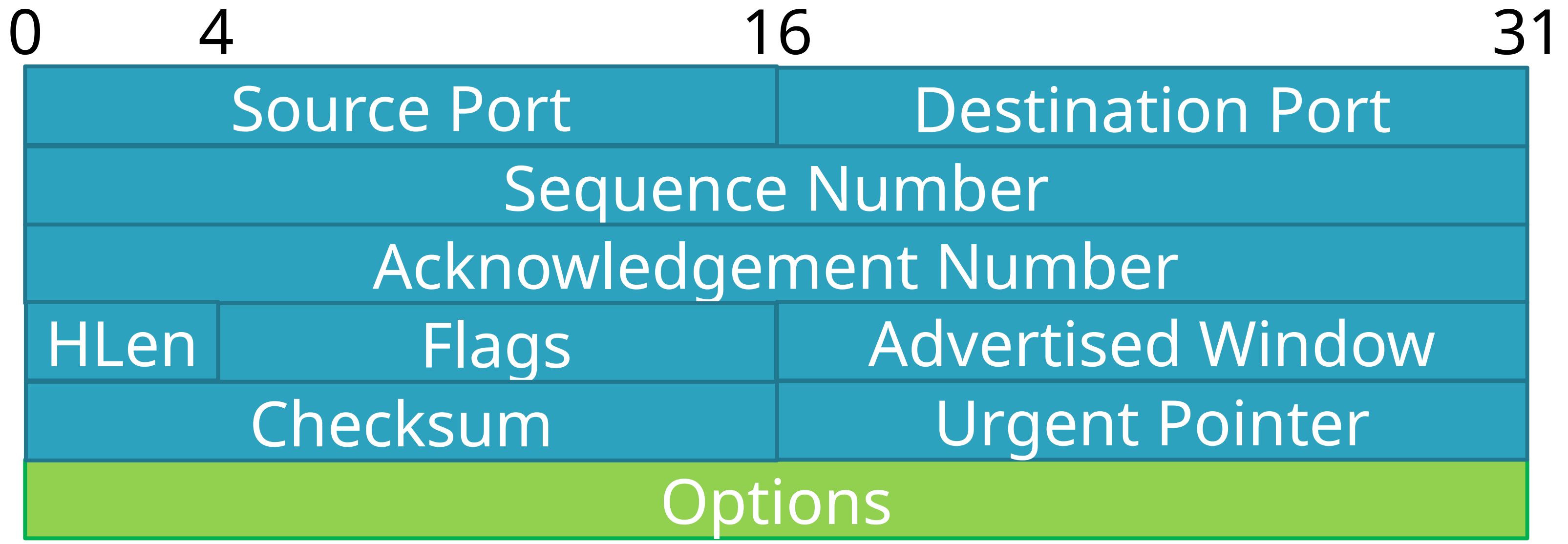
Outline

- 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

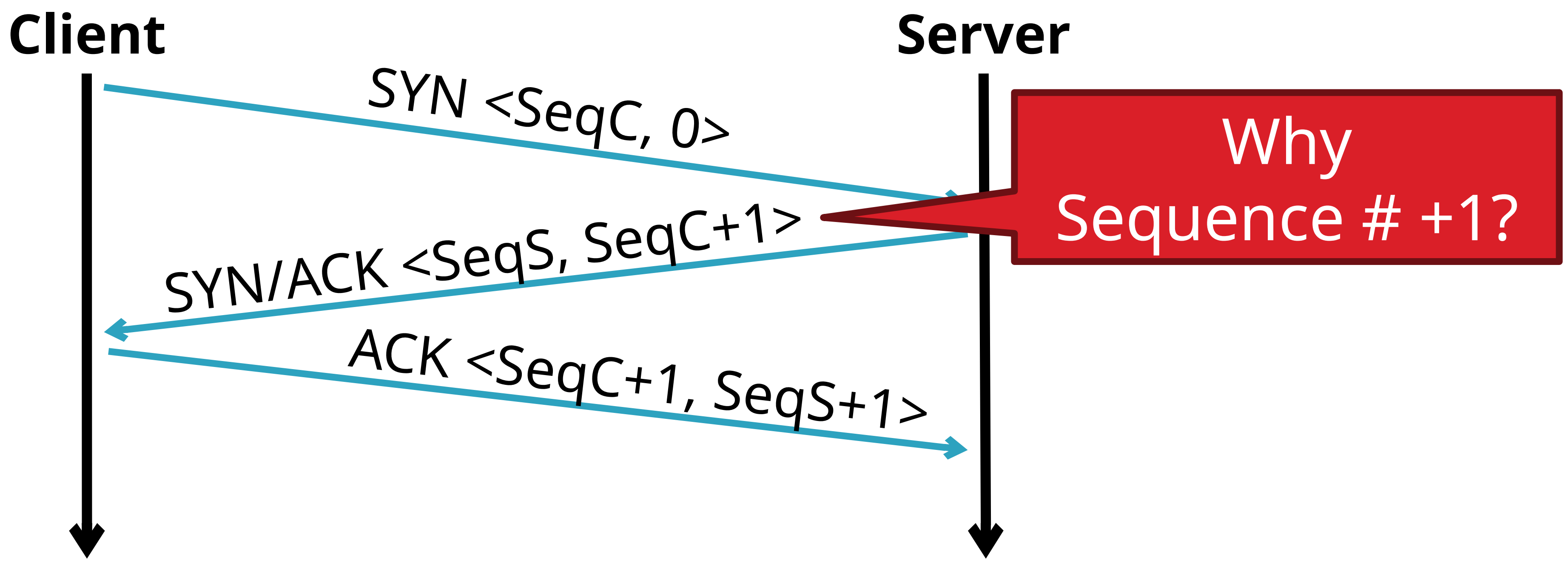
Why these features?



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

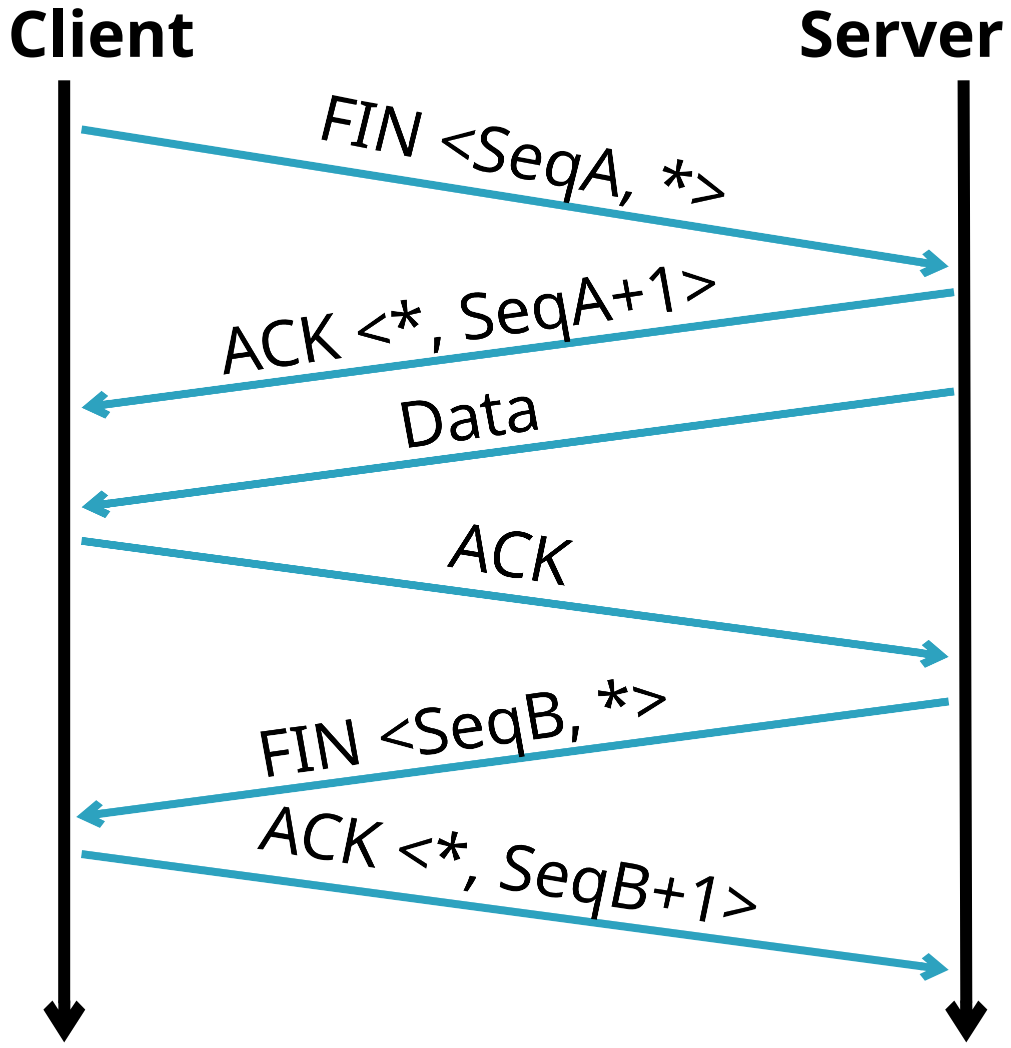
Three Way Handshake



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

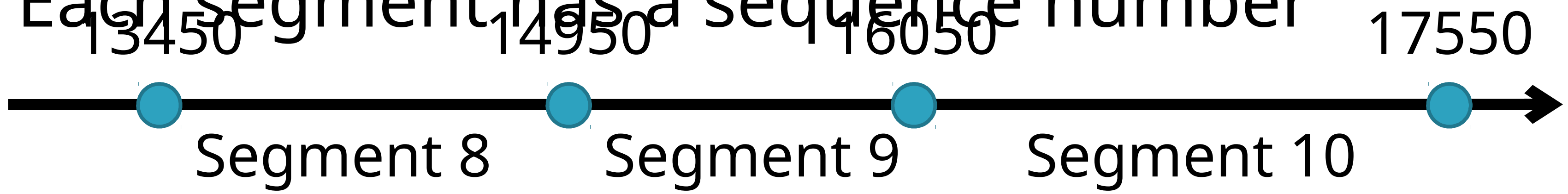
Connection Tear Down

- 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?

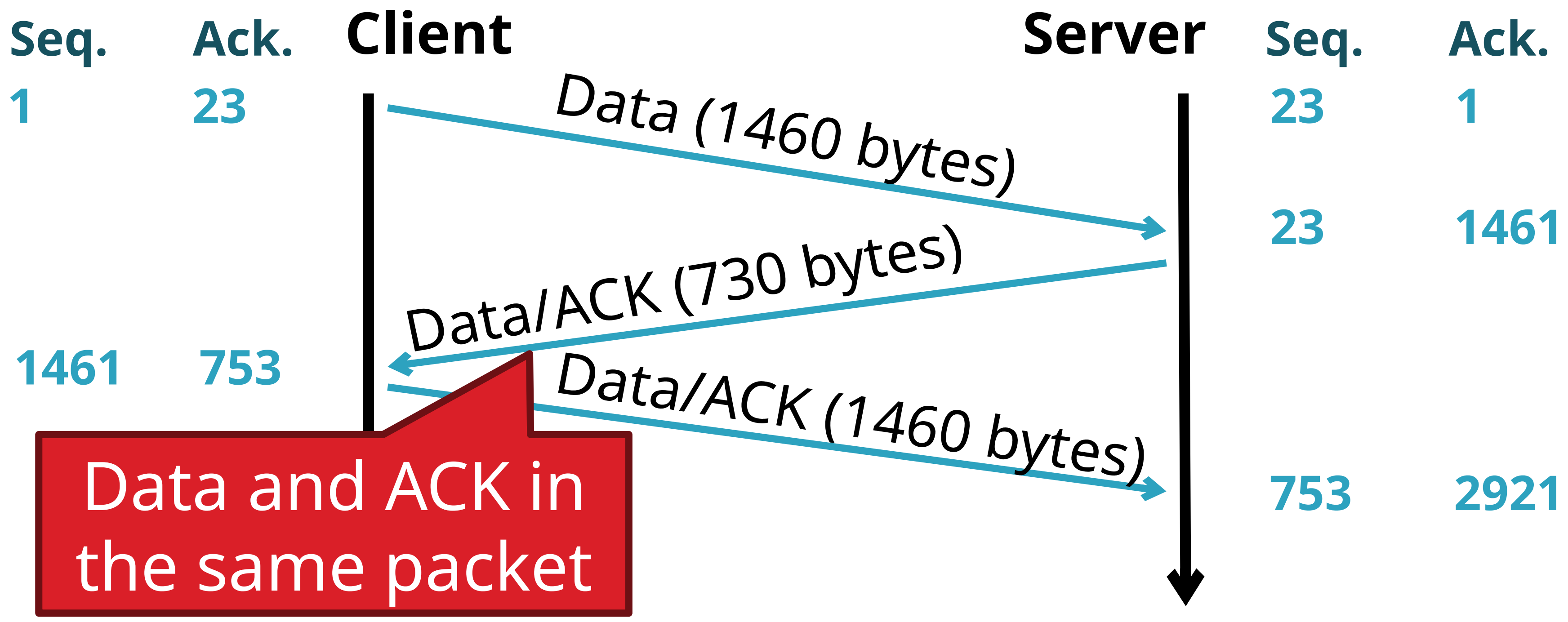


Sequence Number Space

- 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

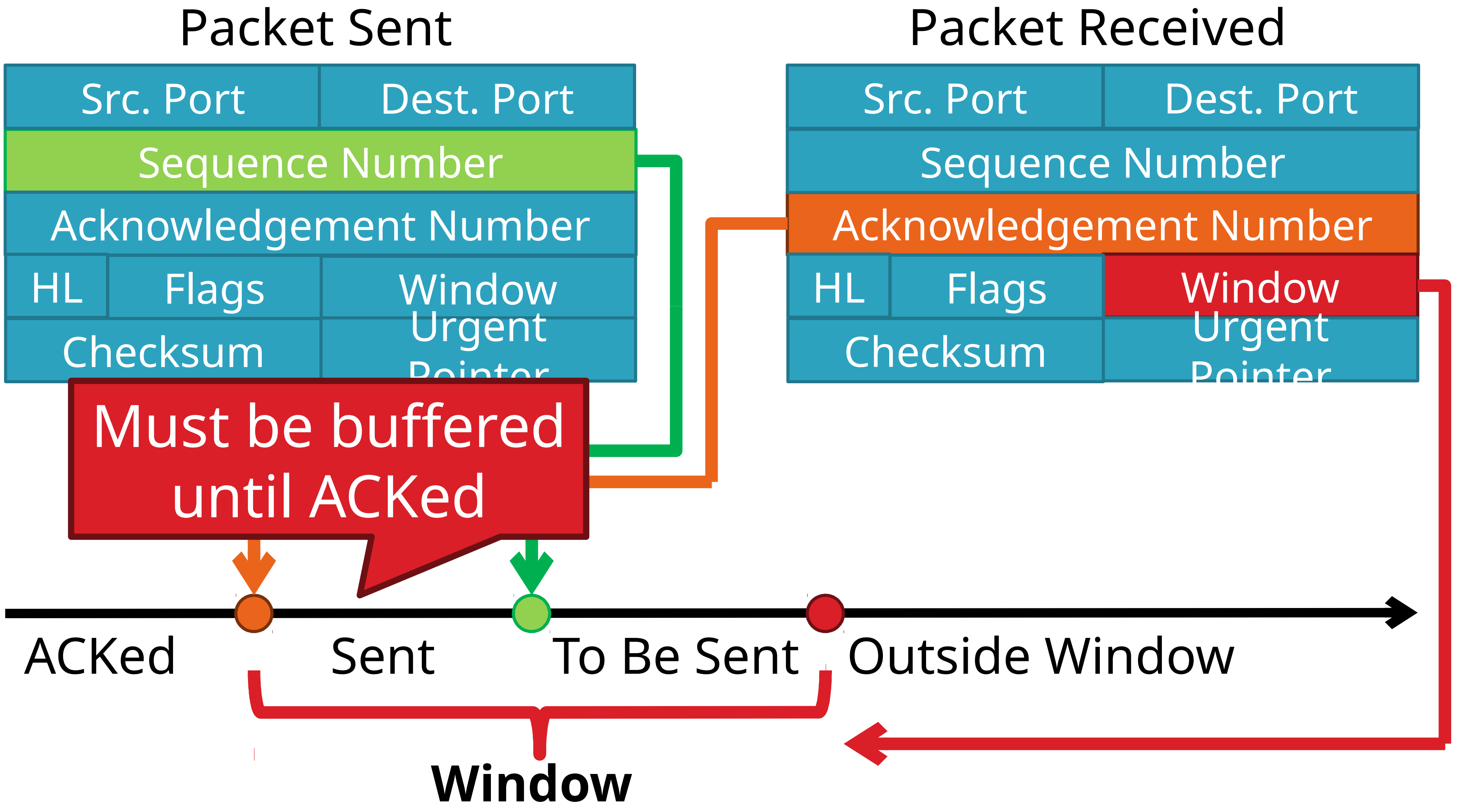


- Each side of the connection can send and receive
  - Different sequence numbers for each direction

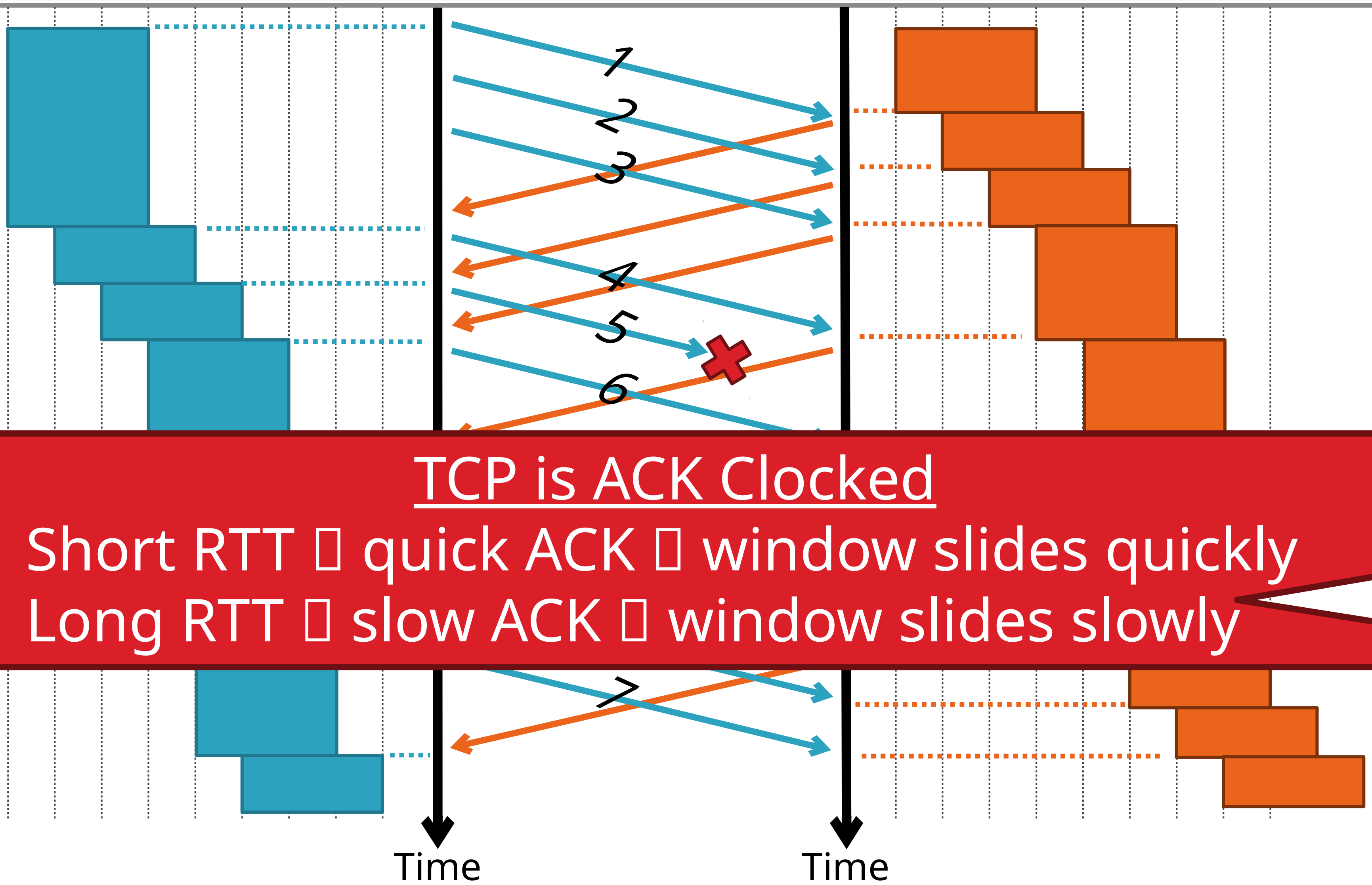
Flow Control

- 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?

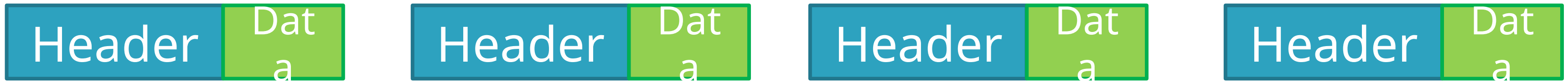
1. ACK every packet
2. Use *cumulative ACK*, where an ACK for sequence *n* implies ACKS for all *k* < *n*
3. Use *negative ACKs* (NACKs), indicating which packet did not arrive
4. Use *selective ACKs* (SACKs), indicating those that did arrive, even if not in order
   - SACK is an actual TCP extension

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

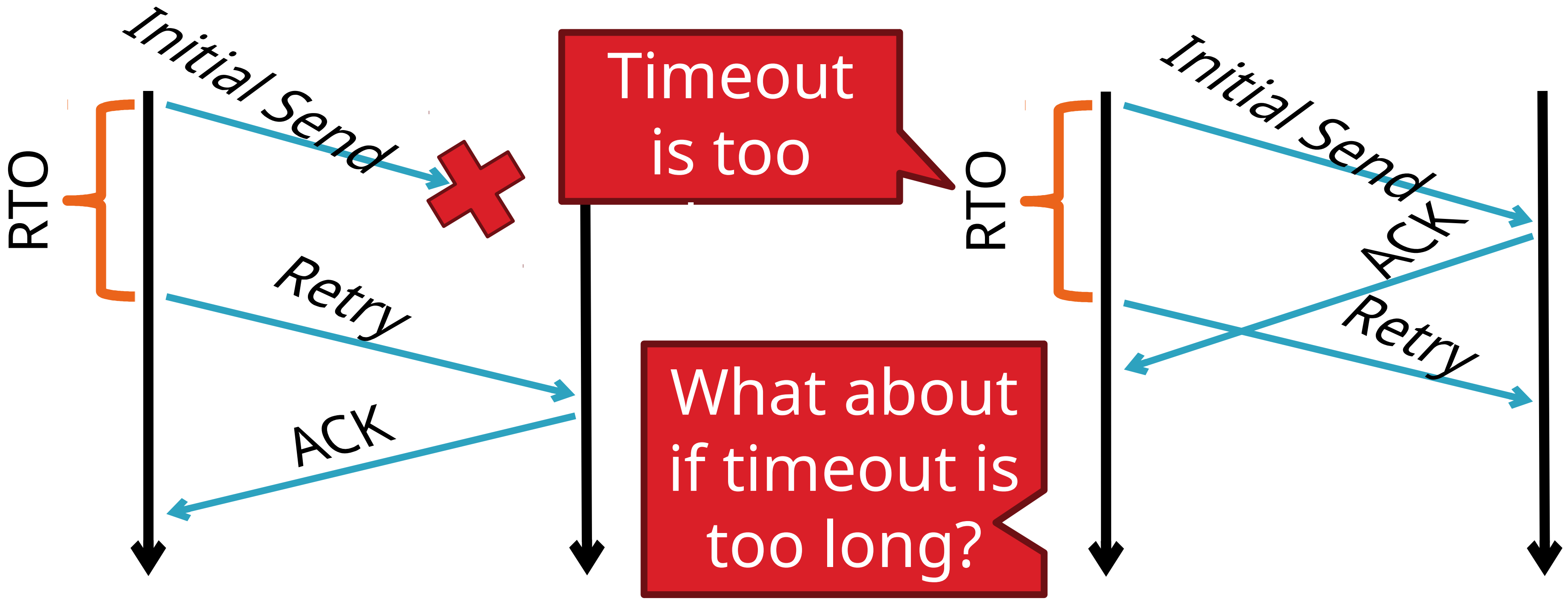
- Problem: Nagle's Algorithm delays transmissions
  - What if you need to send a packet immediately?
  1. int flag = 1;
  2. setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *) &flag, sizeof(int));

Error Detection

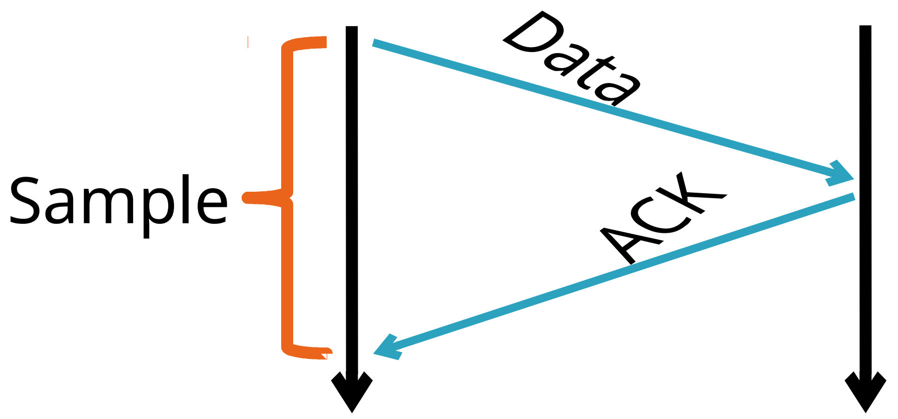
- Checksum detects (some) packet corruption
  - Computed over IP header, TCP header, and data
- Sequence numbers catch sequence problems
  - Duplicates are ignored
  - Out-of-order packets are reordered or dropped
  - Missing sequence numbers indicate lost packets
- Lost segments detected by sender
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

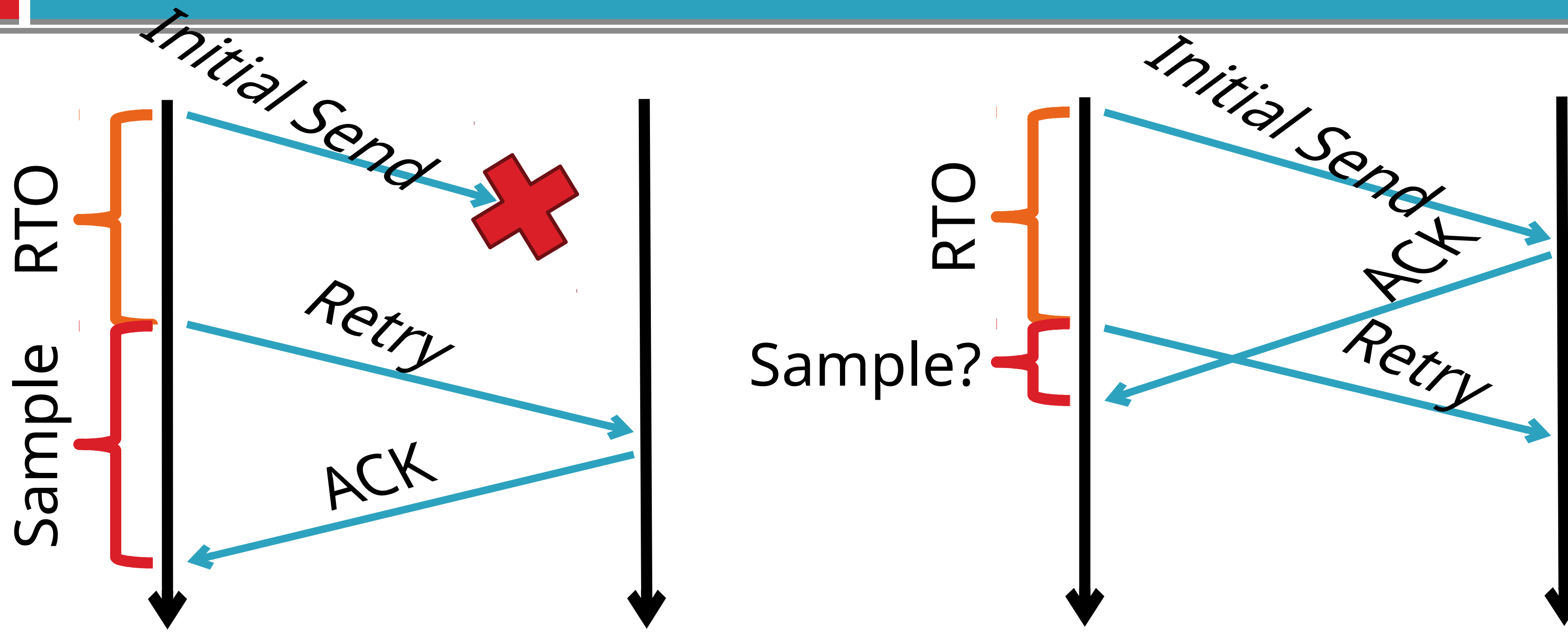


Round Trip Time Estimation



- 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



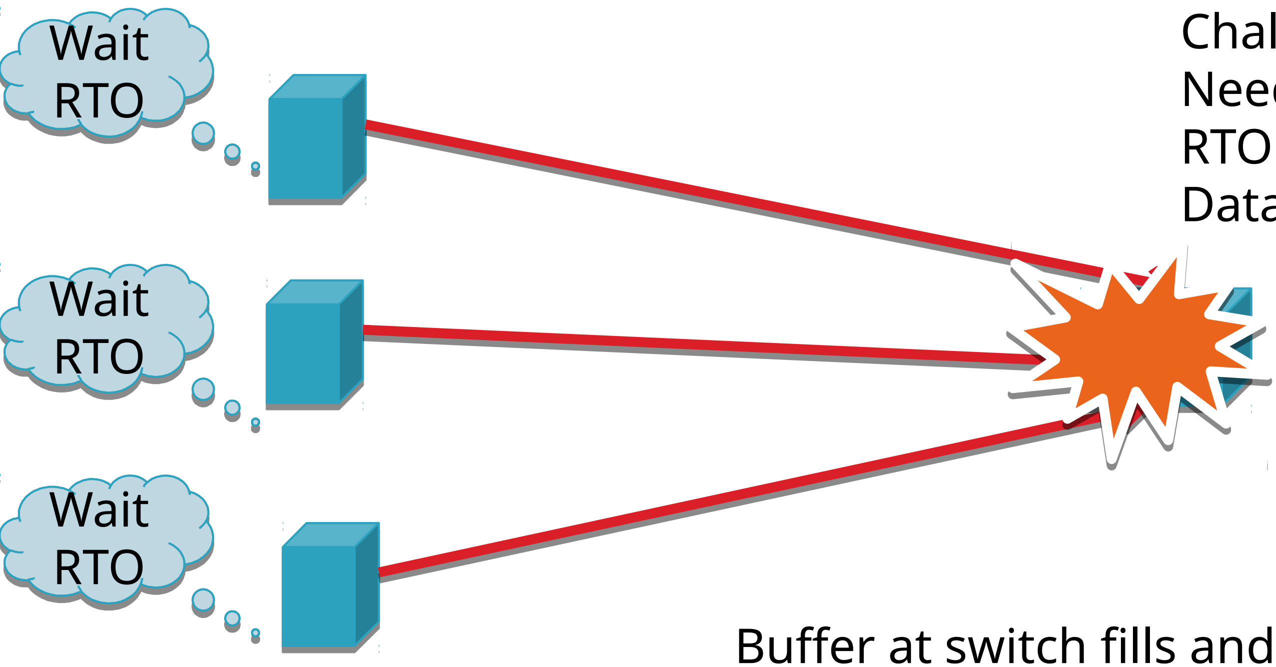
- Karn’s algorithm: ignore samples for retransmitted segments

Challenge of RTO in data centers

- 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 wid
Data centers have much smaller



Buffer at switch fills and packets are lost!
No ACKs will come back