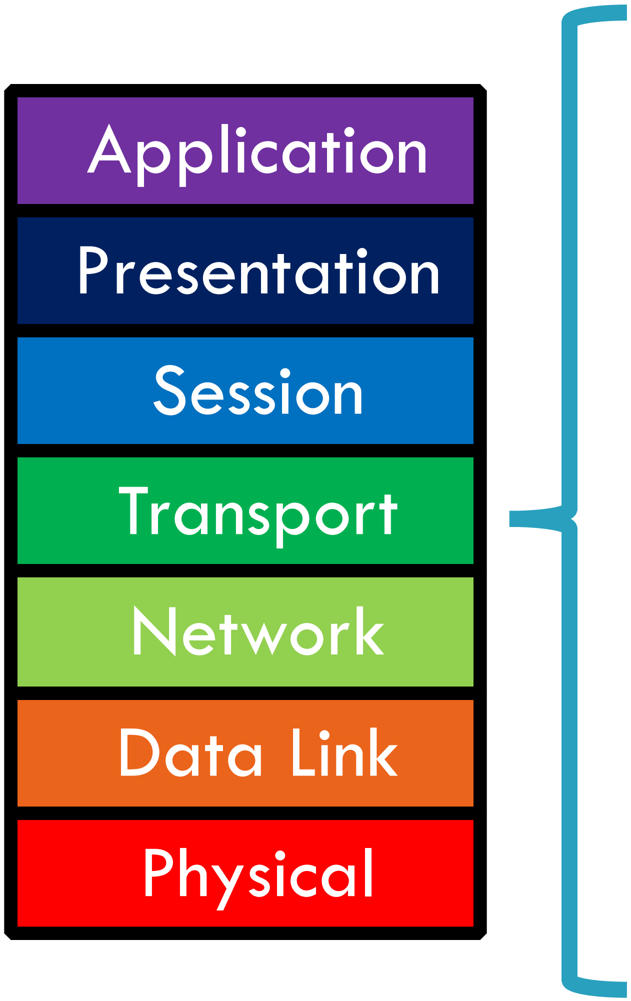
CS 3700
Networks and Distributed Systems

Lecture 9: UDP/TCP

Revised 2/9/2014

Transport Layer



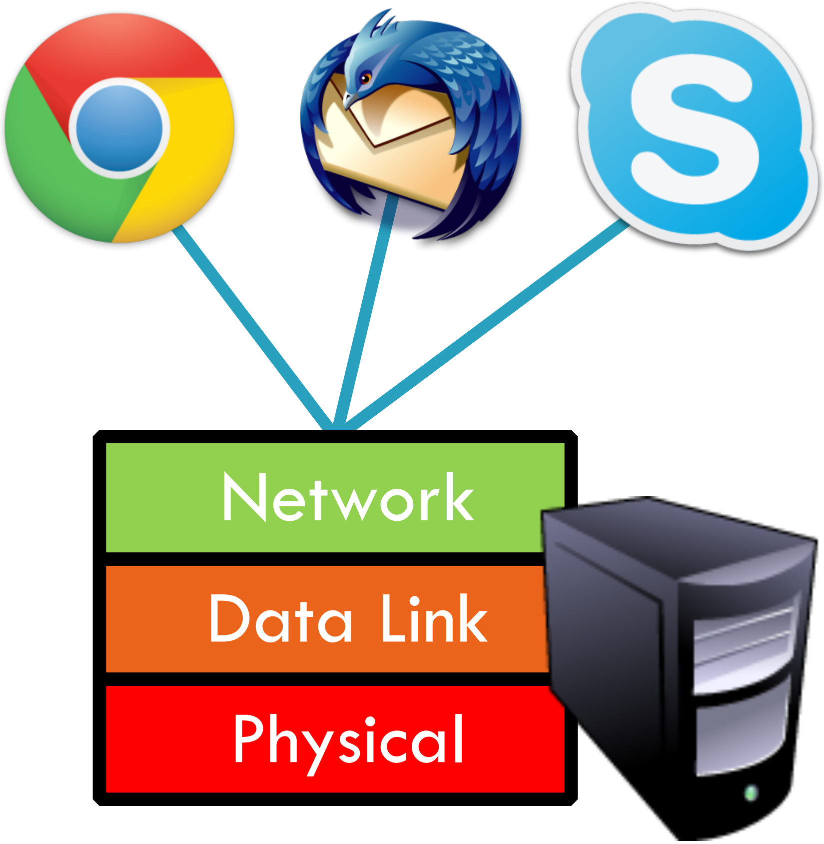
- Function:
  - Demultiplexing of data streams
- Optional functions:
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control
- Key challenges:
  - Detecting and responding to congestion
  - Balancing fairness against high utilization

Outline

- UDP
- TCP

The Case for Multiplexing

- Datagram network
  - No circuits
  - No connections



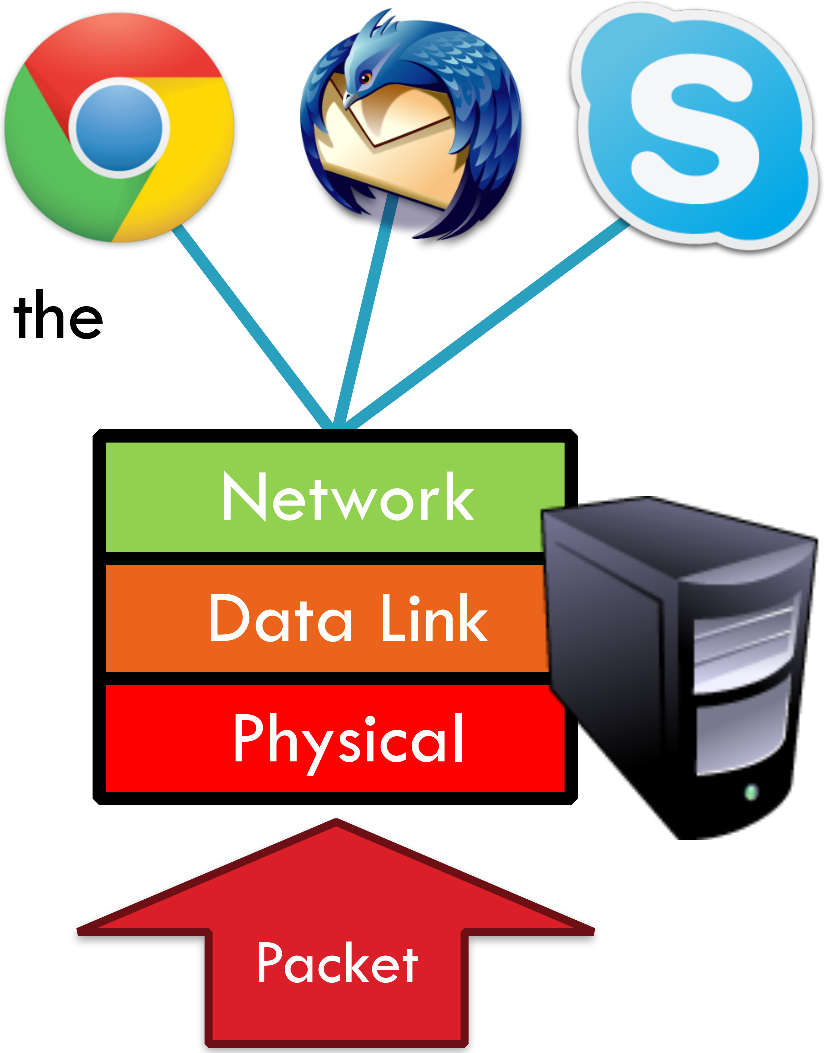
Network

Data Link

Physical

The Case for Multiplexing

- Datagram network
  - No circuits
  - No connections
- Clients run many applications at the same time
  - Who to deliver packets to?



Network

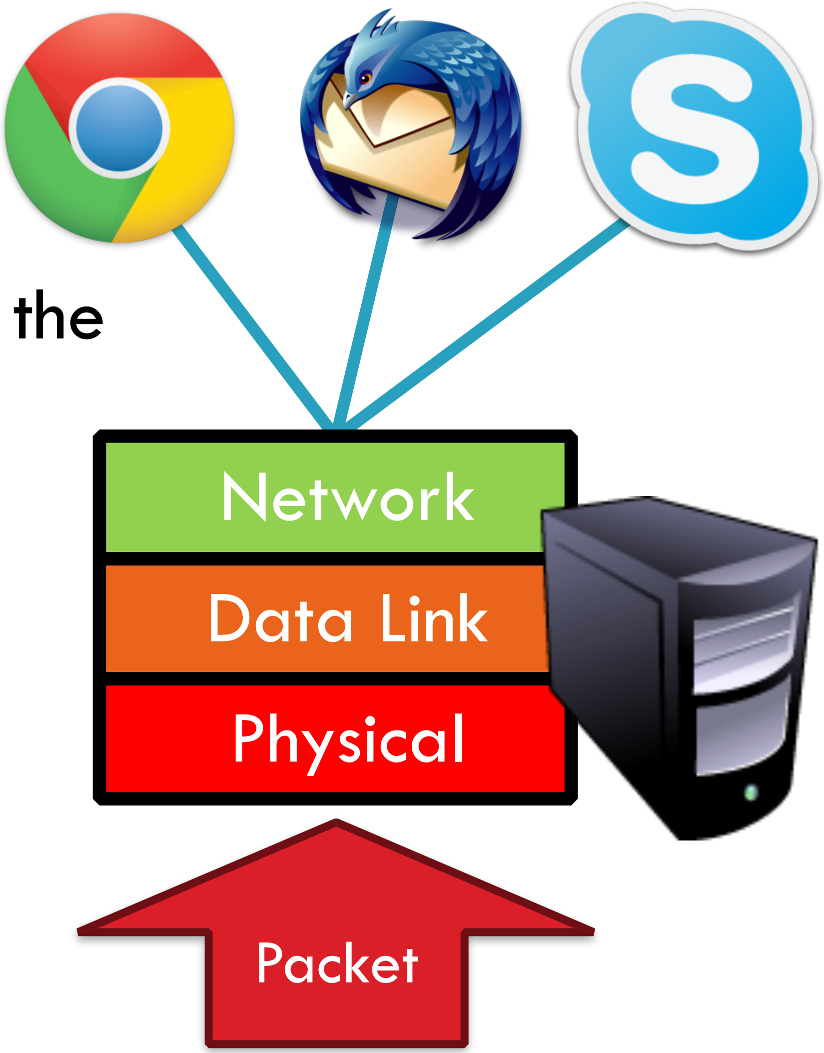
Data Link

Physical

Packet

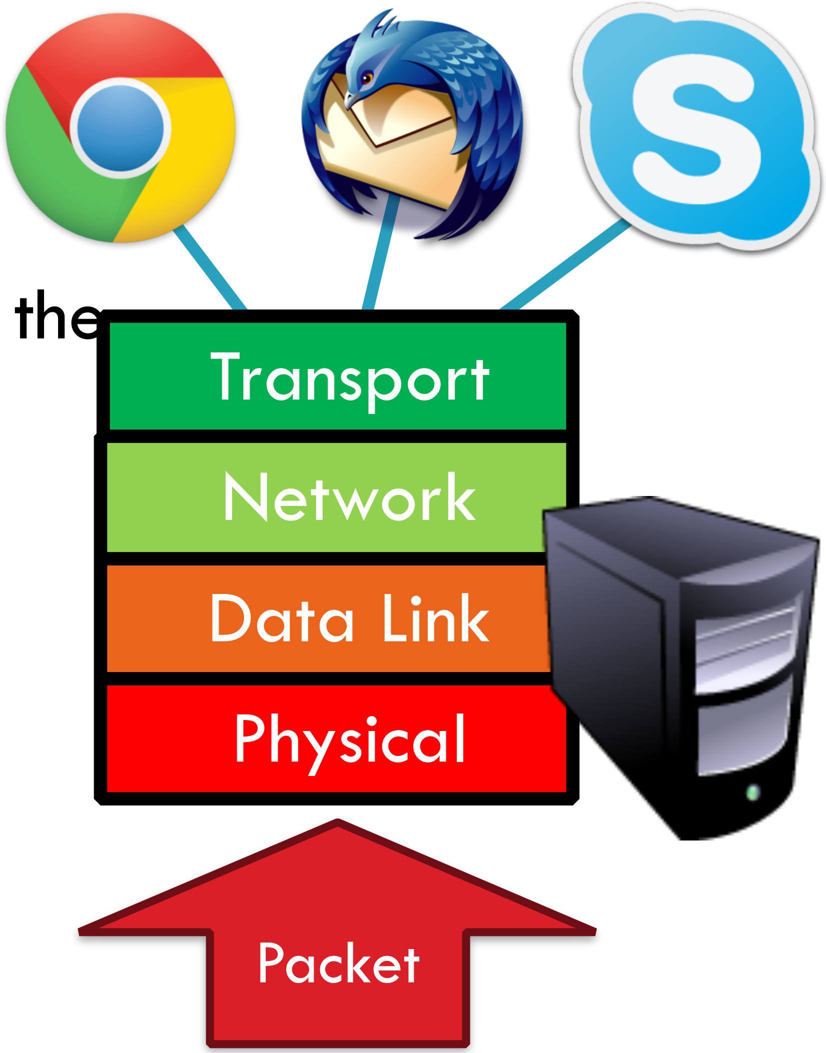
The Case for Multiplexing

- 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



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



Transport

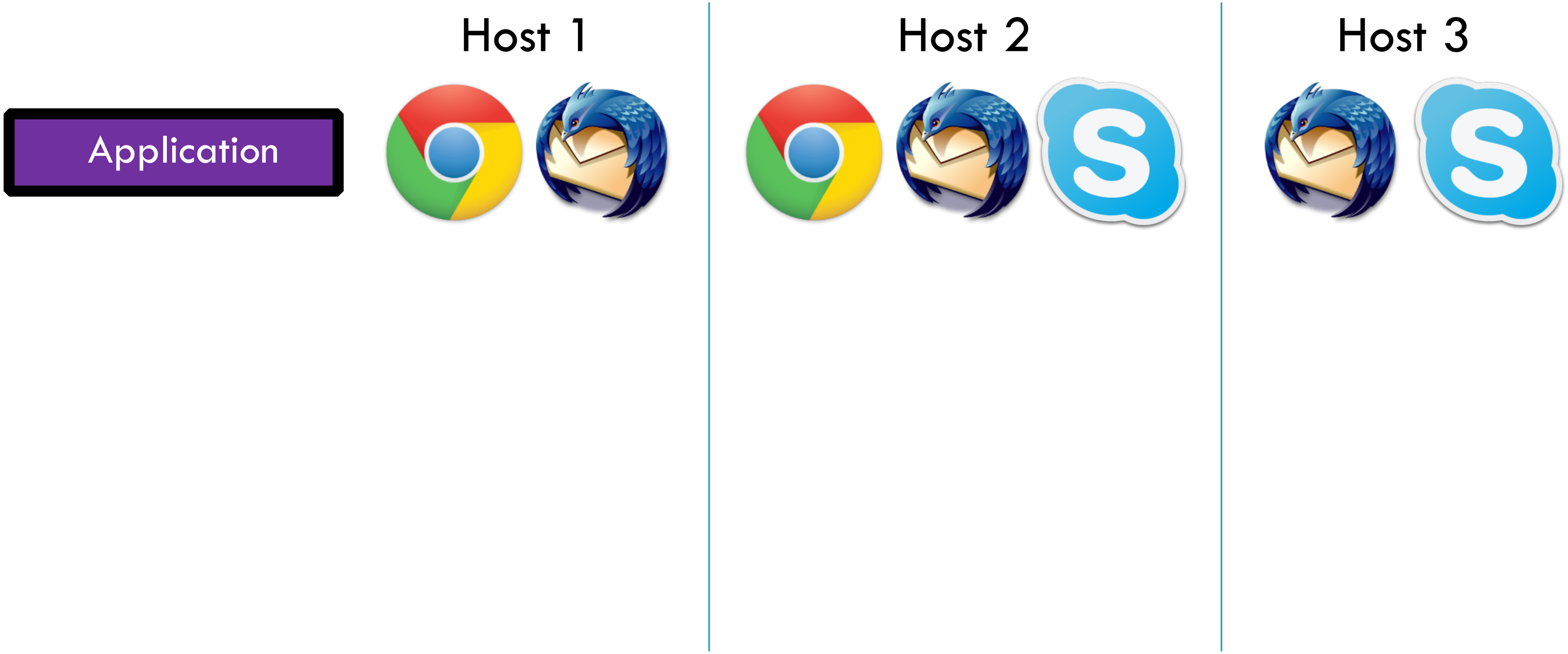
Network

Data Link

Physical

Packet

Demultiplexing Traffic



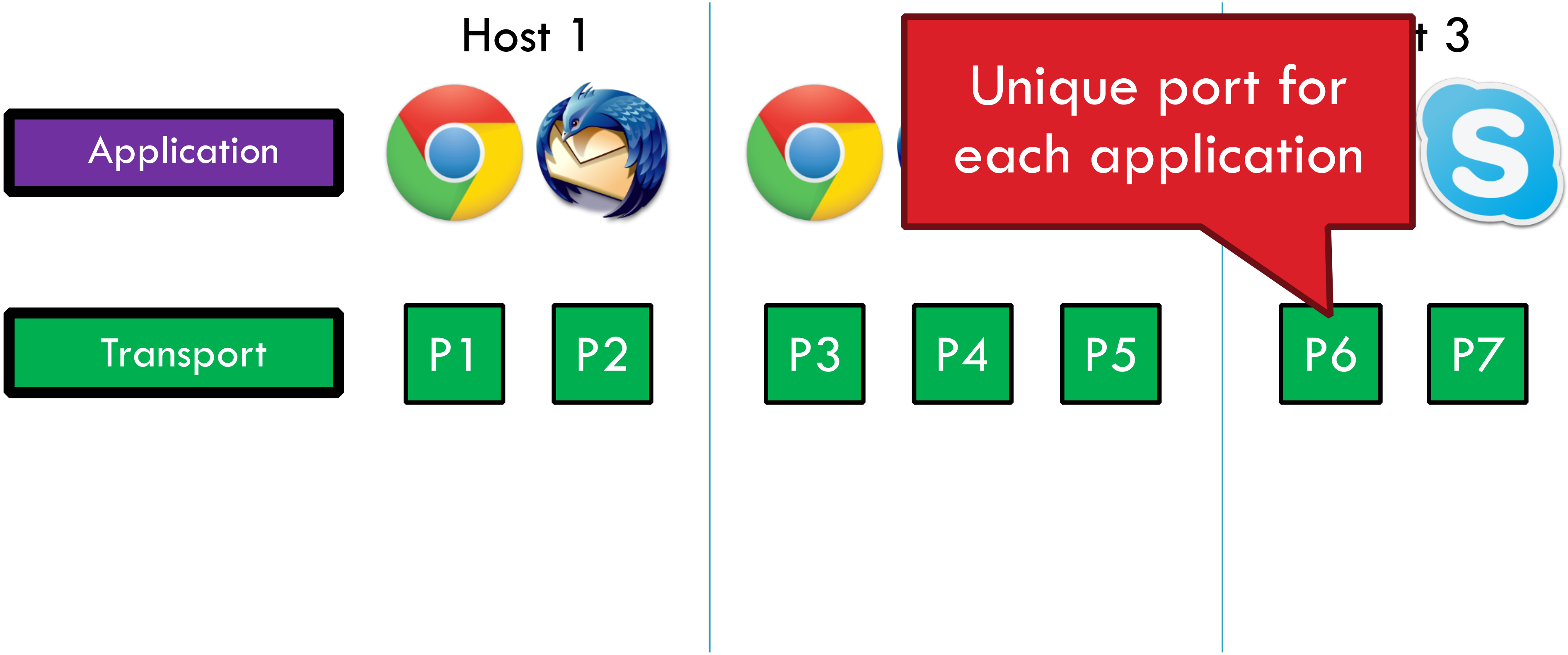
Application

Host 1

Host 2

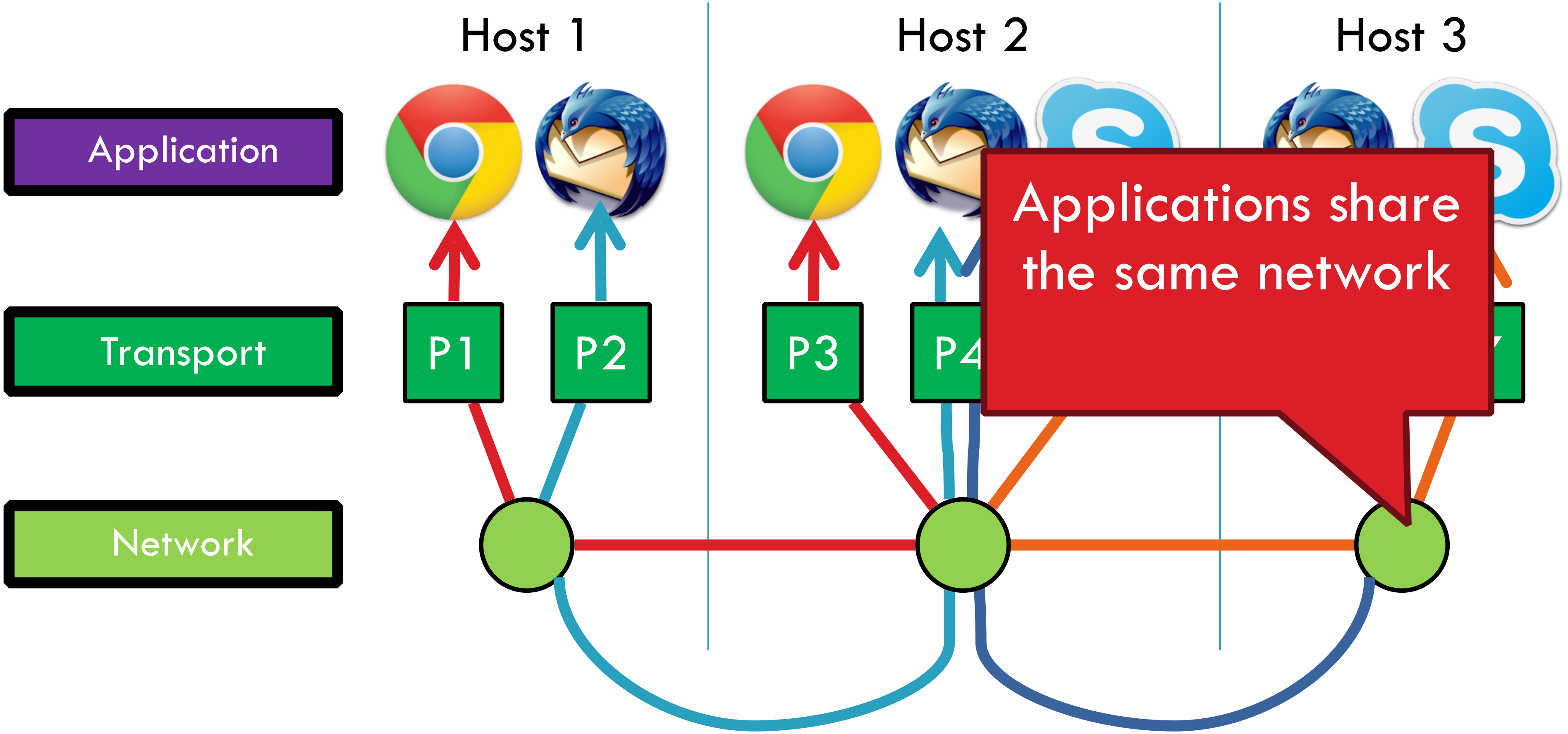
Host 3

Demultiplexing Traffic



Unique port for each application

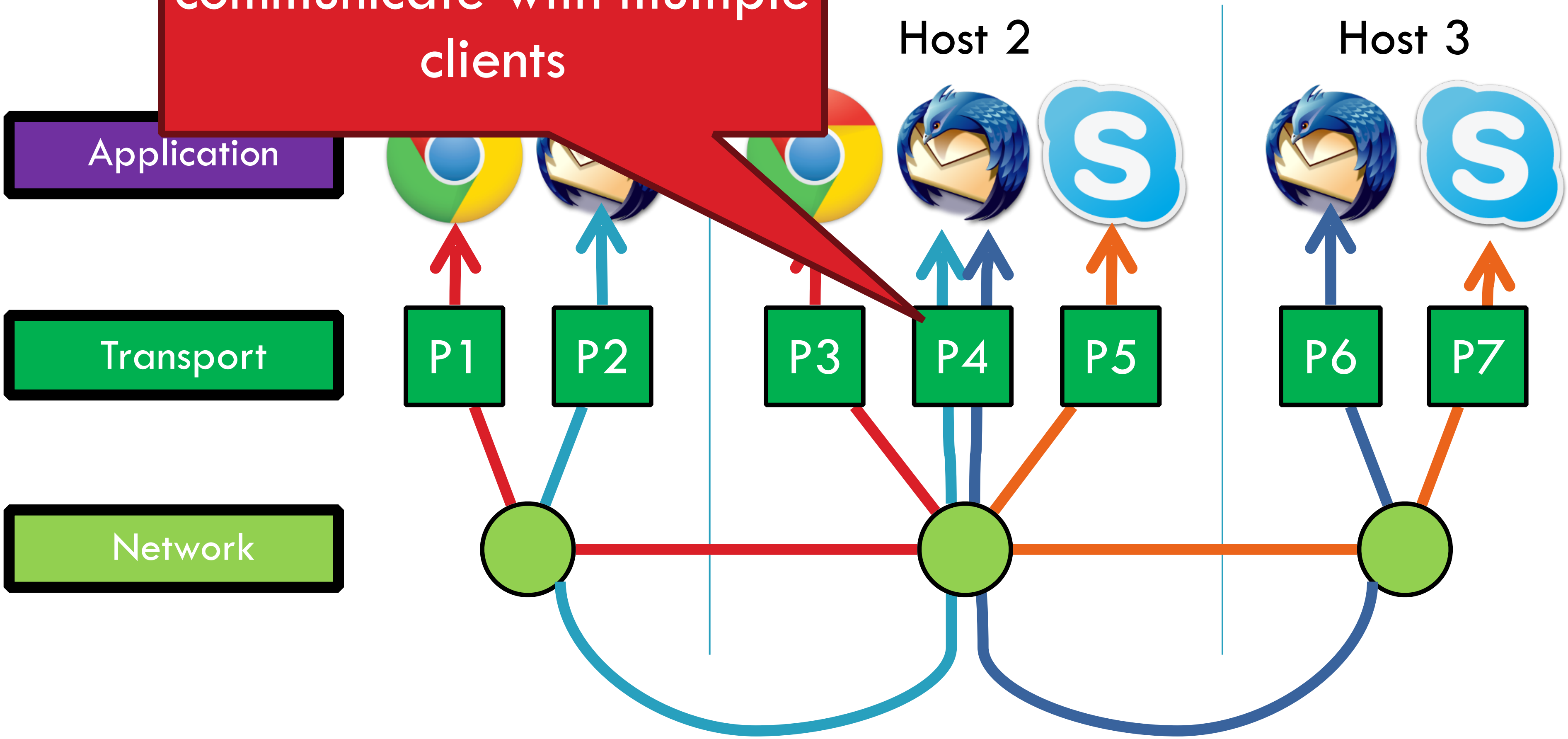
Demultiplexing Traffic



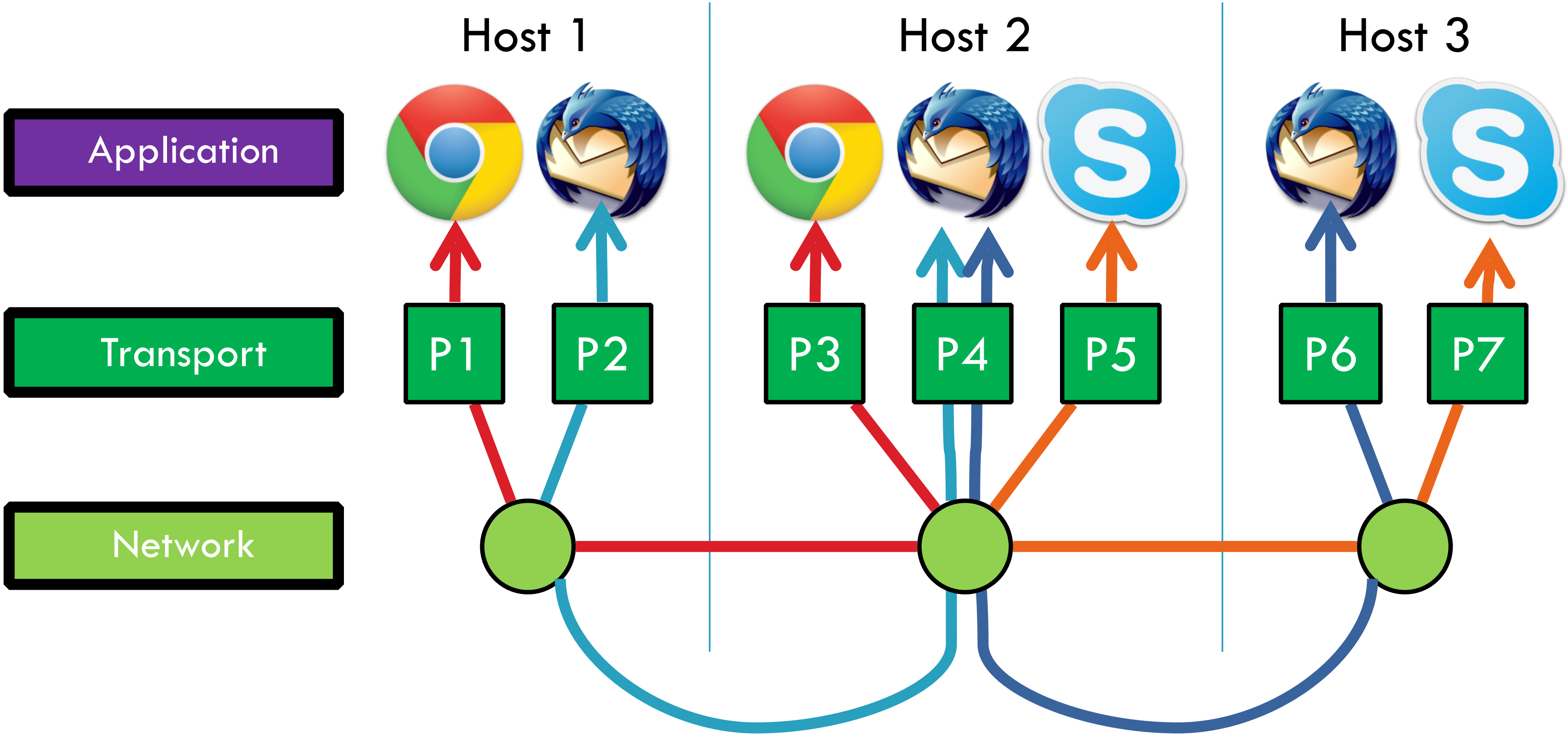
Applications share the same network

Demultiplexing Traffic

Server applications communicate with multiple clients

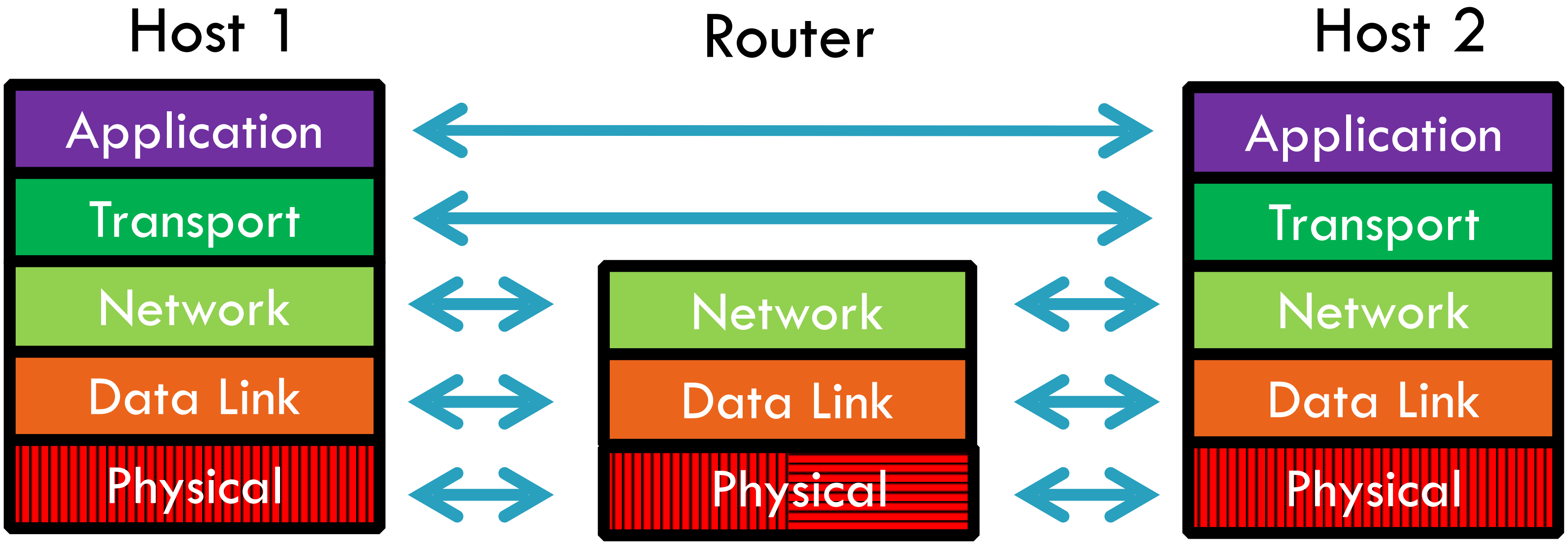


Demultiplexing Traffic



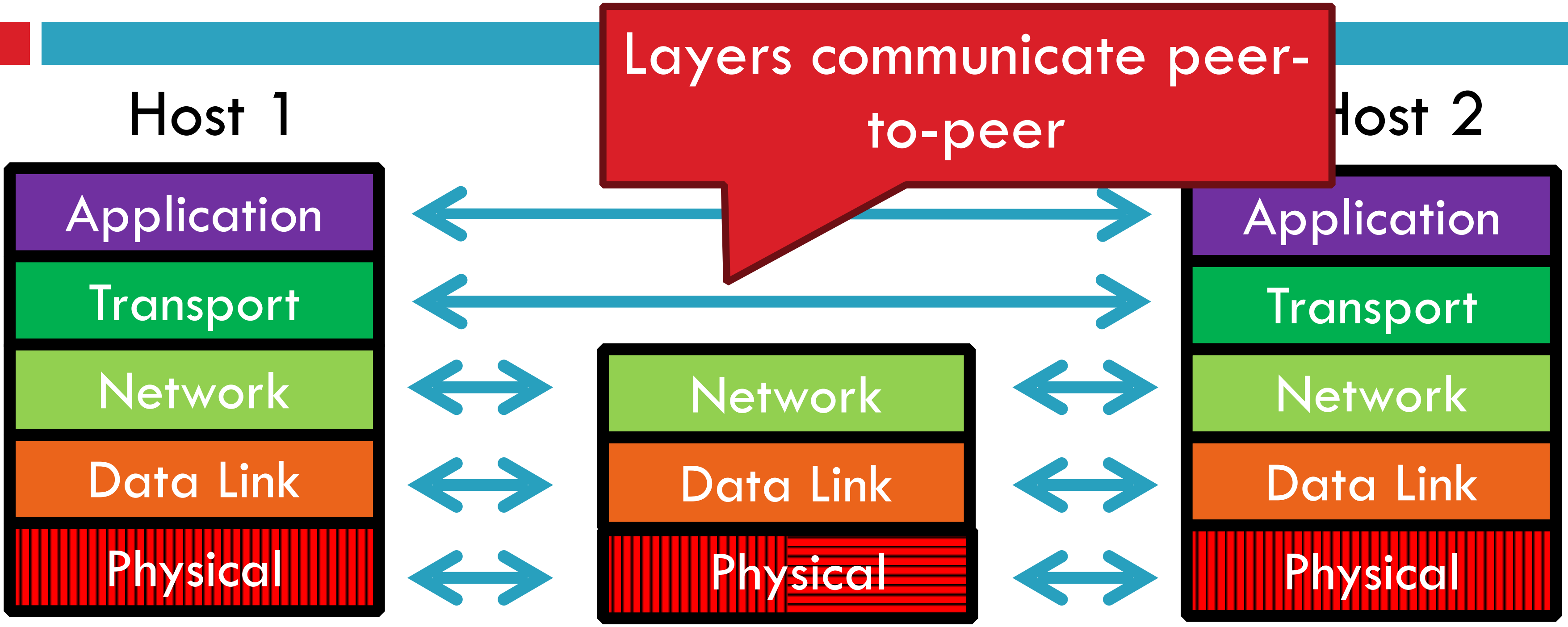
Endpoints identified by <*src_ip, src_port, dest_ip, dest_port*>

Layering, Revisited



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

Layering, Revisited



Layers communicate peer-to-peer

Host 1

Application

Transport

Network

Data Link

Physical

Network

Data Link

Physical

Host 2

Application

Transport

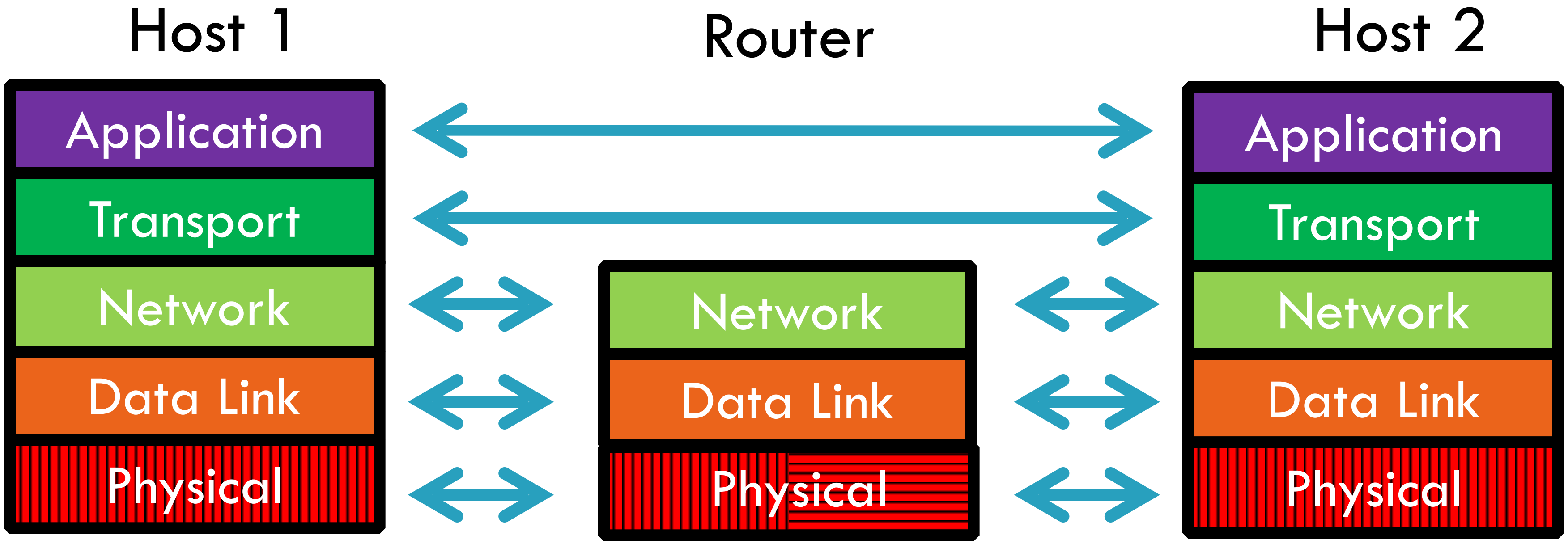
Network

Data Link

Physical

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

Layering, Revisited



Host 1

Application
Transport
Network
Data Link
Physical

Router

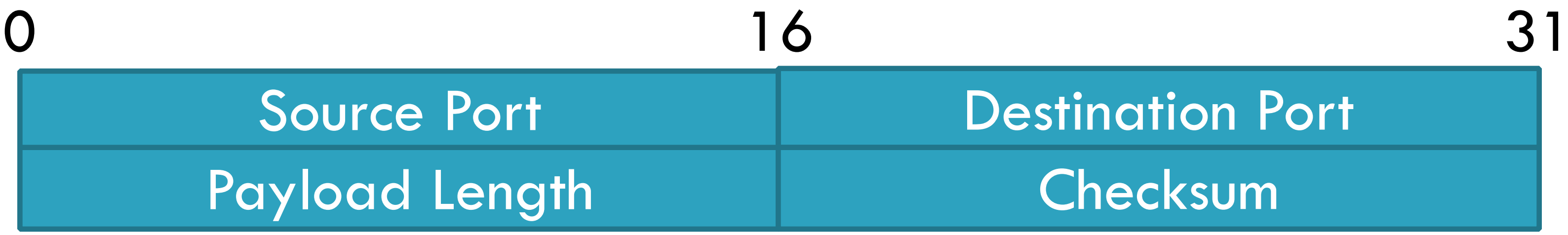
Network
Data Link
Physical

Host 2

Application
Transport
Network
Data Link
Physical

- 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?

Uses for UDP

- Invented after TCP
  - Why?
- Not all applications can tolerate TCP

Uses for UDP

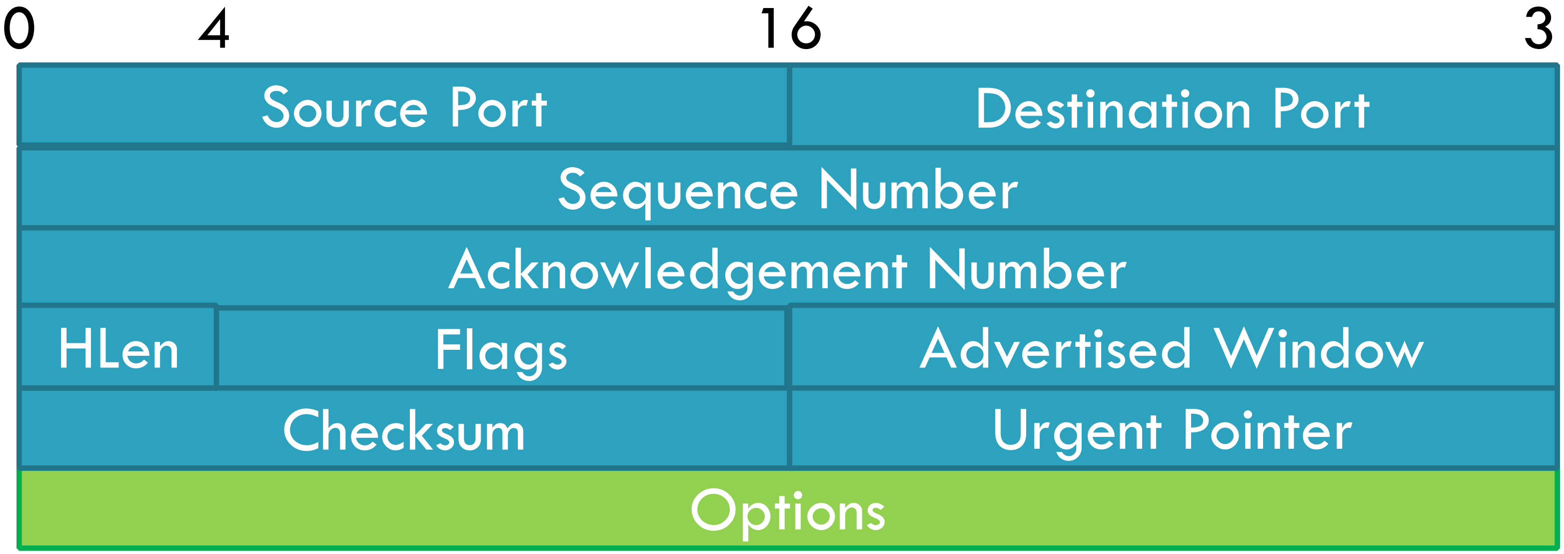
- Invented after TCP
  - Why?
- Not all applications can tolerate TCP
- Custom protocols can be built on top of UDP
  - Reliability? Strict ordering?
  - Flow control? Congestion control?
- Examples
  - RTMP, real-time media streaming (e.g. voice, video)
  - Facebook datacenter protocol

Outline

- UDP
- TCP

Transmission Control Protocol

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

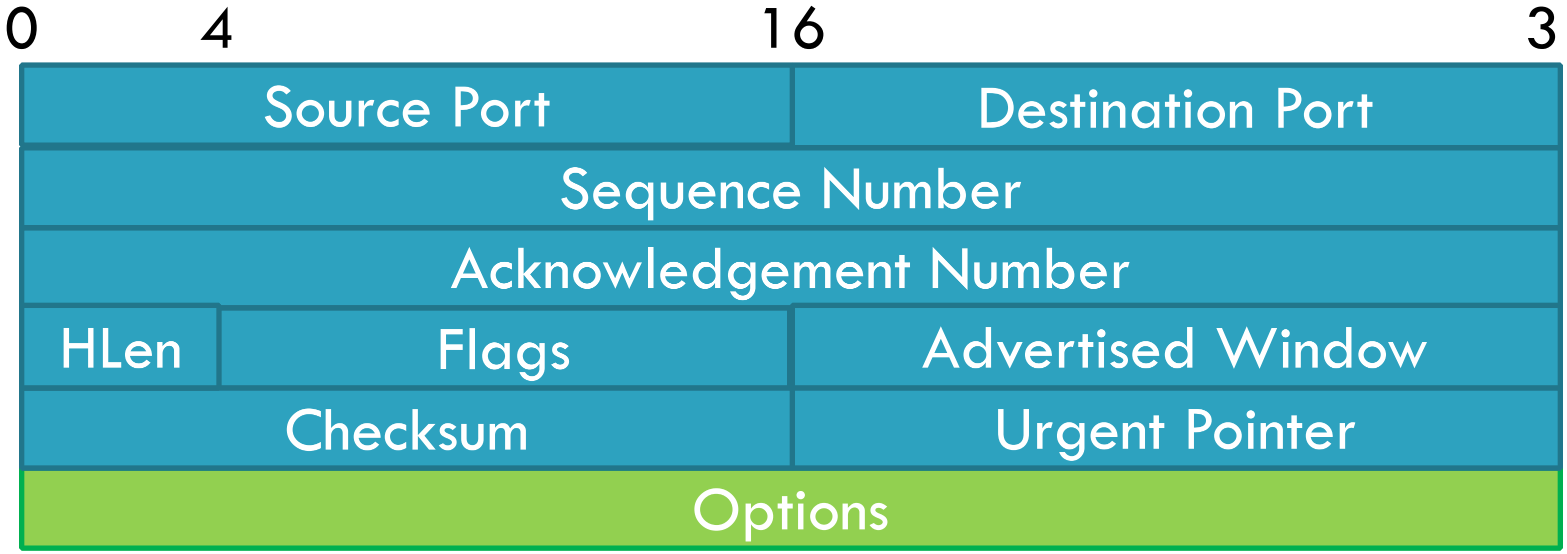


<table>
<tr><th>0</th><th>4</th><th>16</th><th>31</th></tr>
<tr><td colspan="2">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="4">Sequence Number</td></tr>
<tr><td colspan="4">Acknowledgement Number</td></tr>
<tr><td>HLen</td><td>Flags</td><td colspan="2">Advertised Window</td></tr>
<tr><td colspan="2">Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="4">Options</td></tr>
</table>

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?

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?

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

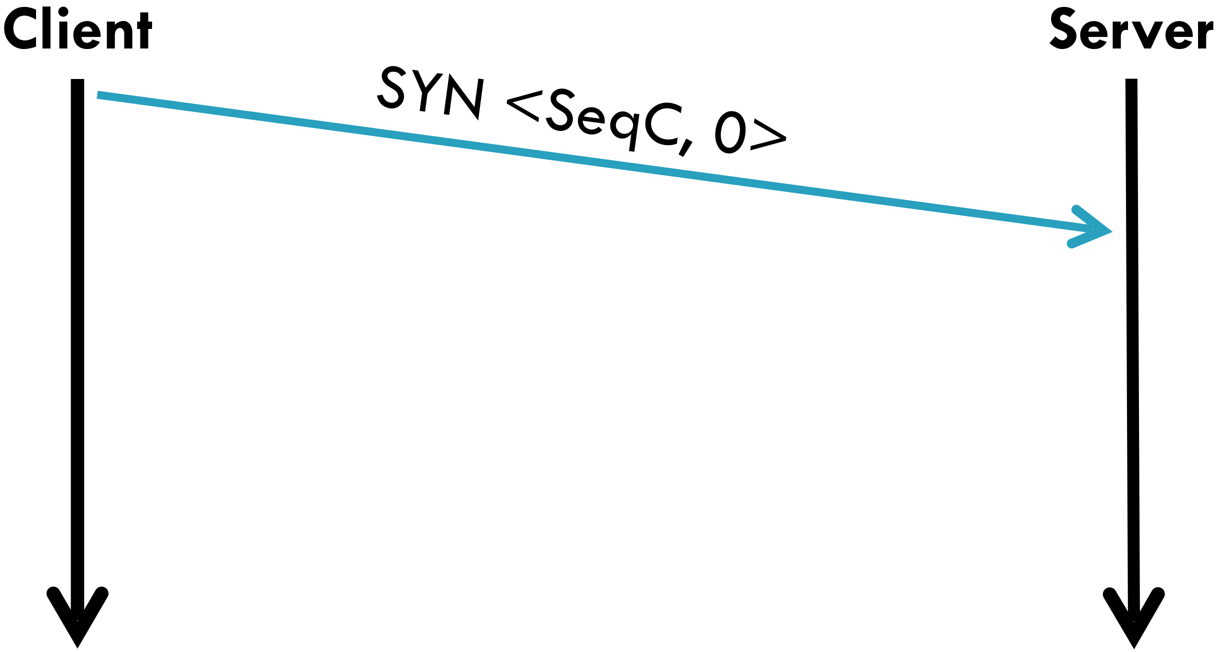
Three Way Handshake

Client

Server

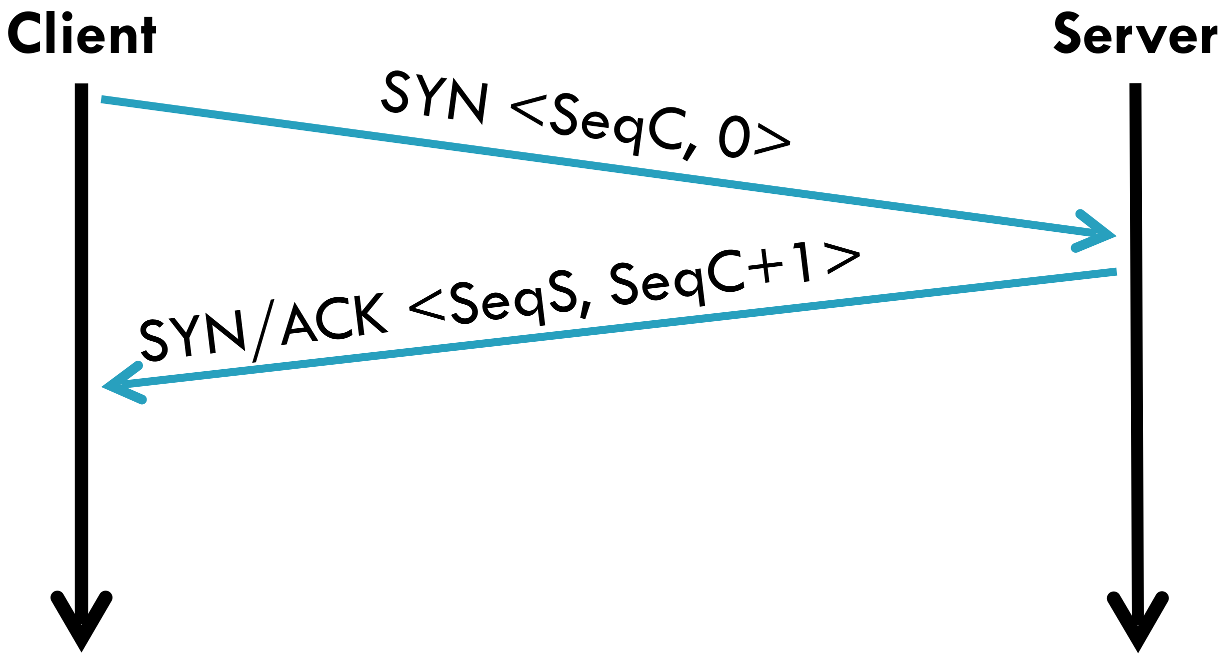
- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side’s starting sequence number

Three Way Handshake



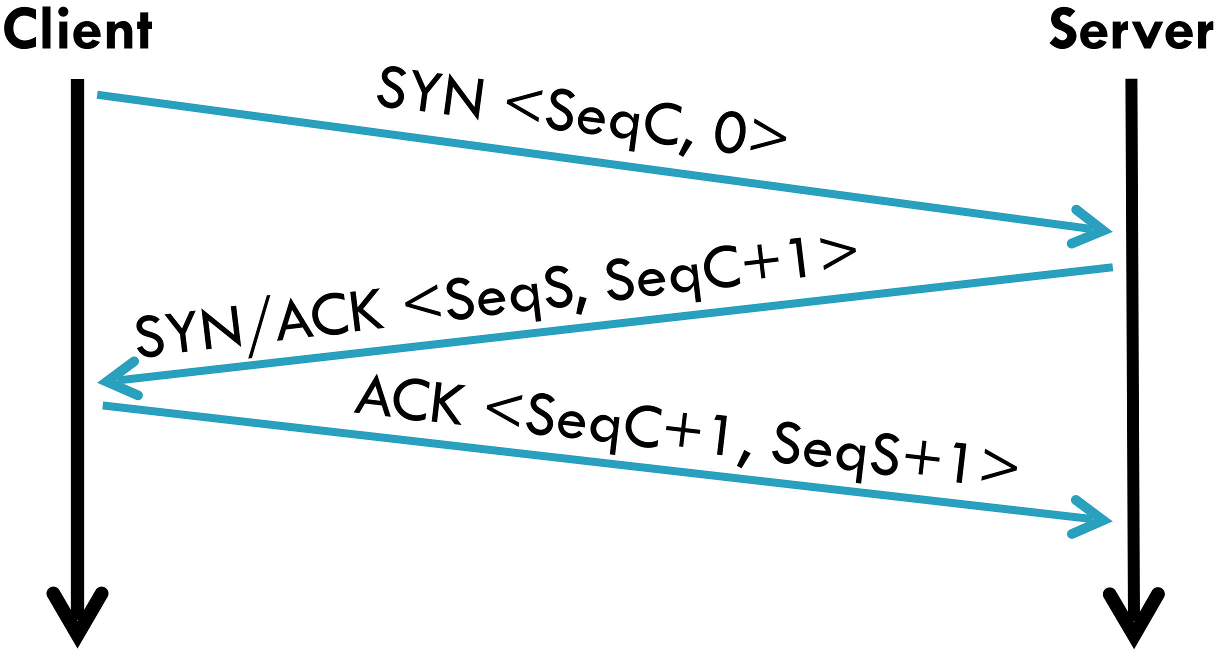
- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side's starting sequence number

Three Way Handshake



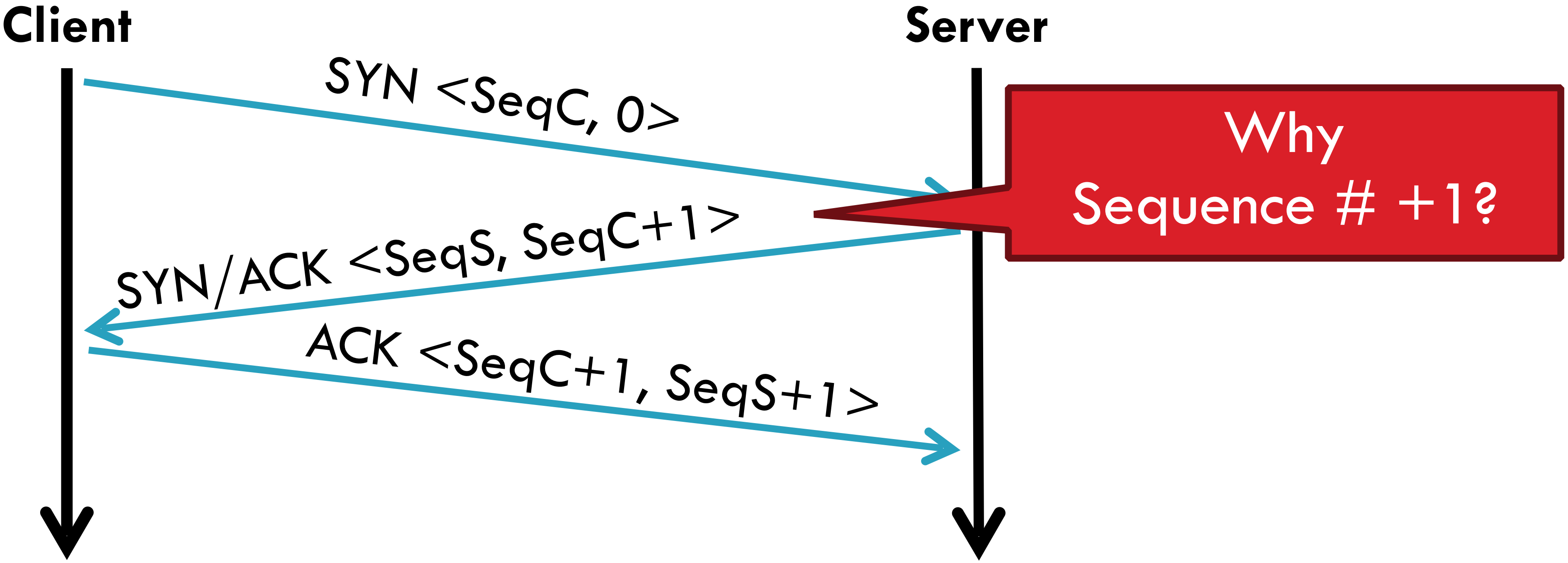
- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side’s starting sequence number

Three Way Handshake



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

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

Connection Setup Issues

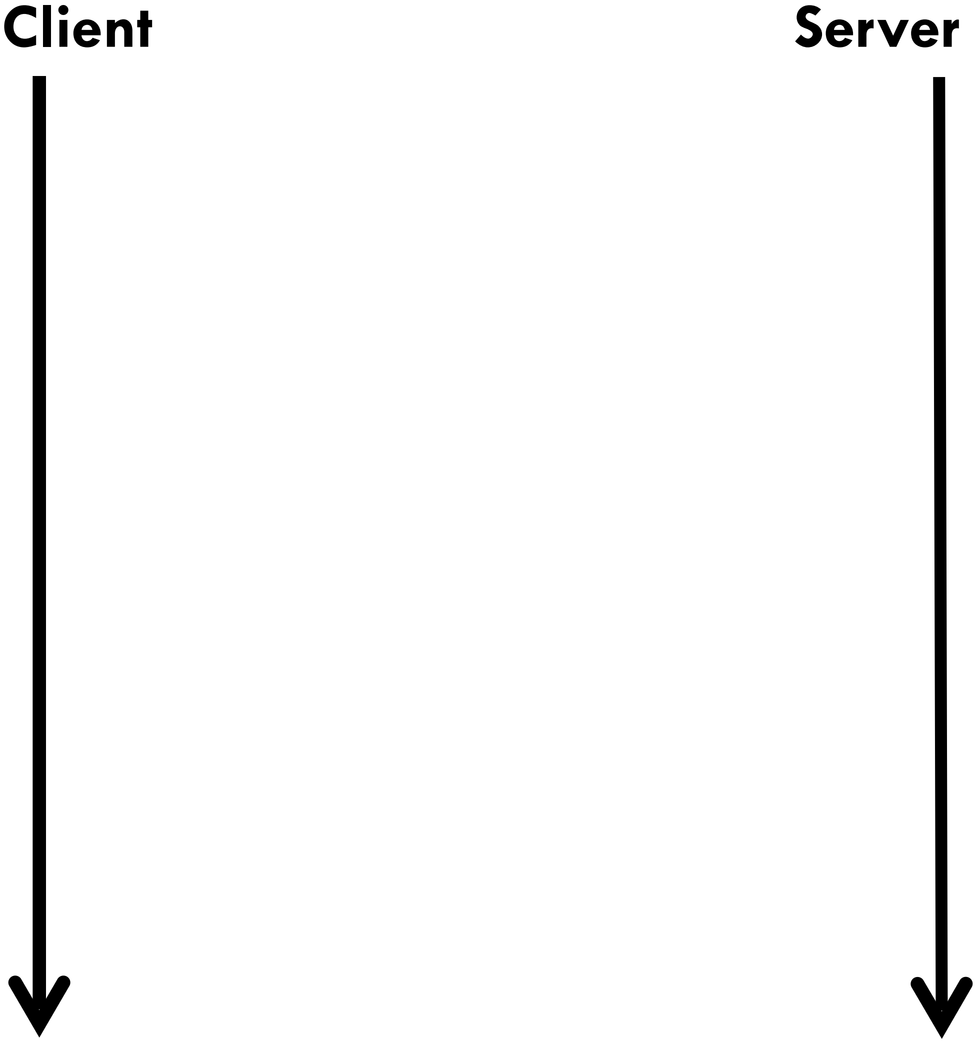
- Connection confusion
  - How to disambiguate connections from the same host?
  - Random sequence numbers
- Source spoofing
  - Kevin Mitnick
  - Need good random number generators!

Connection Setup Issues

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

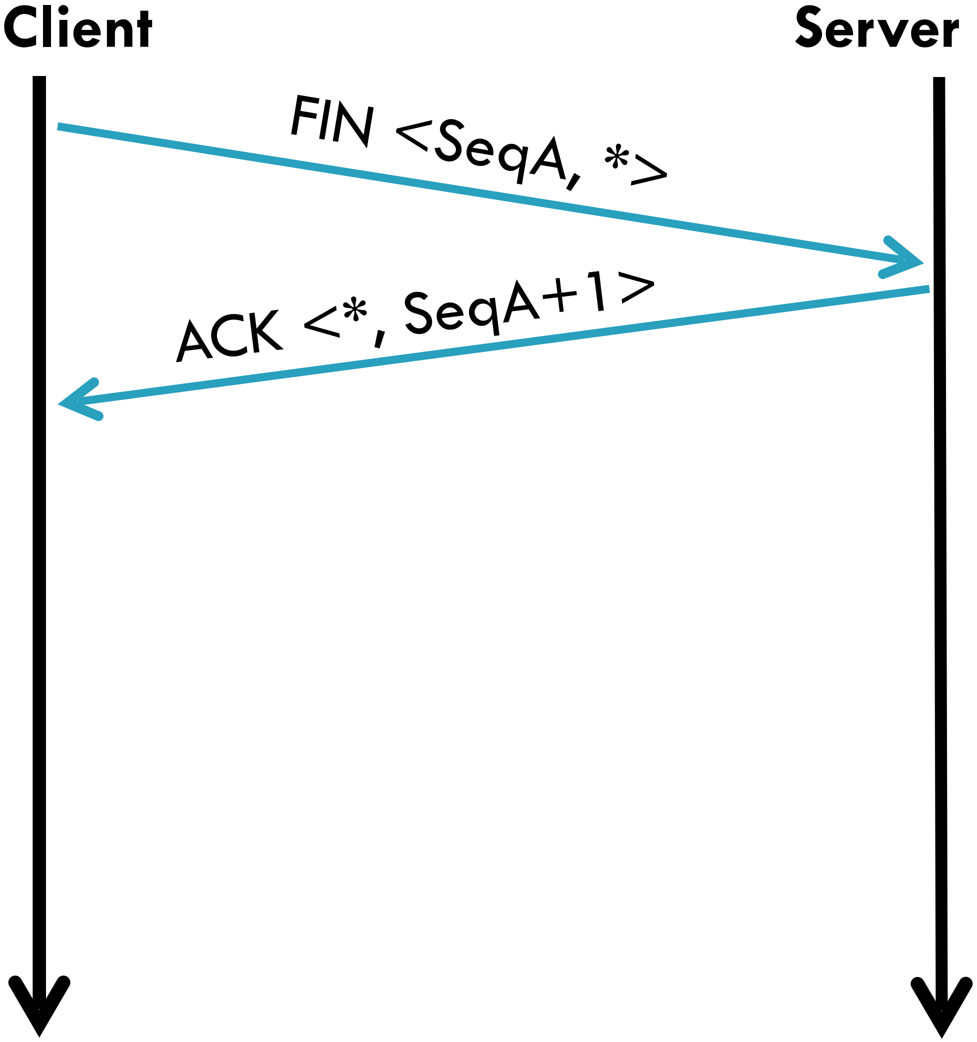
Connection Tear Down

- Either side can initiate tear down



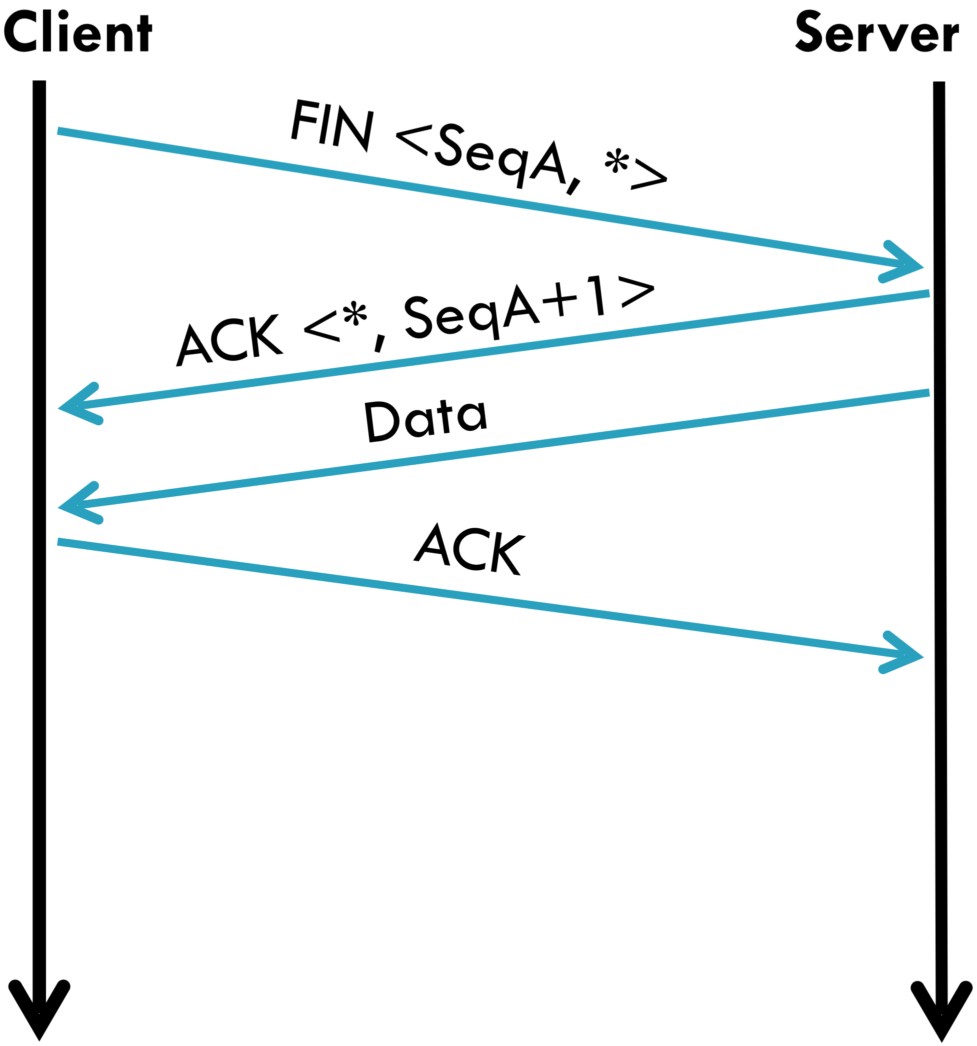
Connection Tear Down

- Either side can initiate tear down



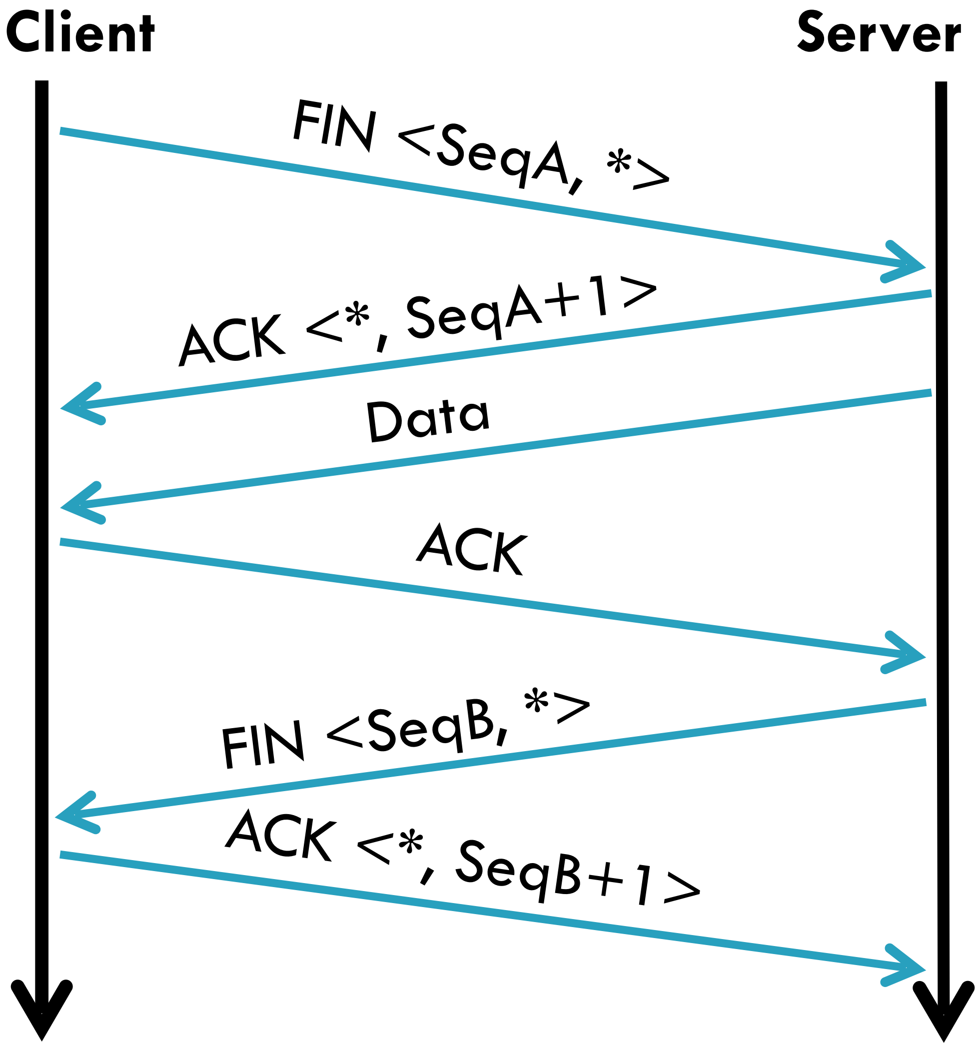
Connection Tear Down

- Either side can initiate tear down
- Other side may continue sending data
  - Half open connection
  - *shutdown()*



Connection Tear Down

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



Sequence Number Space

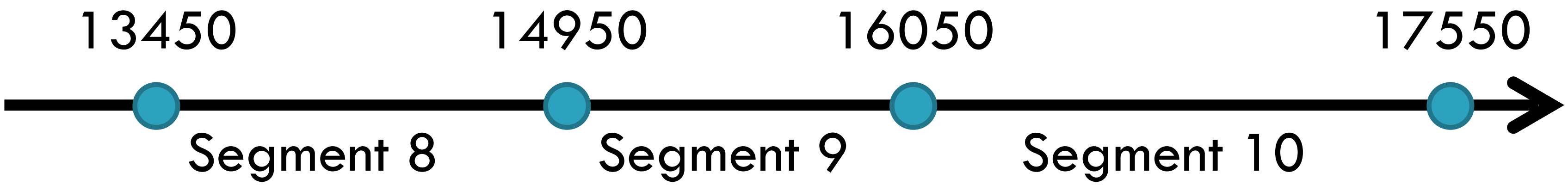
- TCP uses a byte stream abstraction
  - Each byte in each stream is numbered
  - 32-bit value, wraps around
  - Initial, random values selected during setup

Sequence Number Space

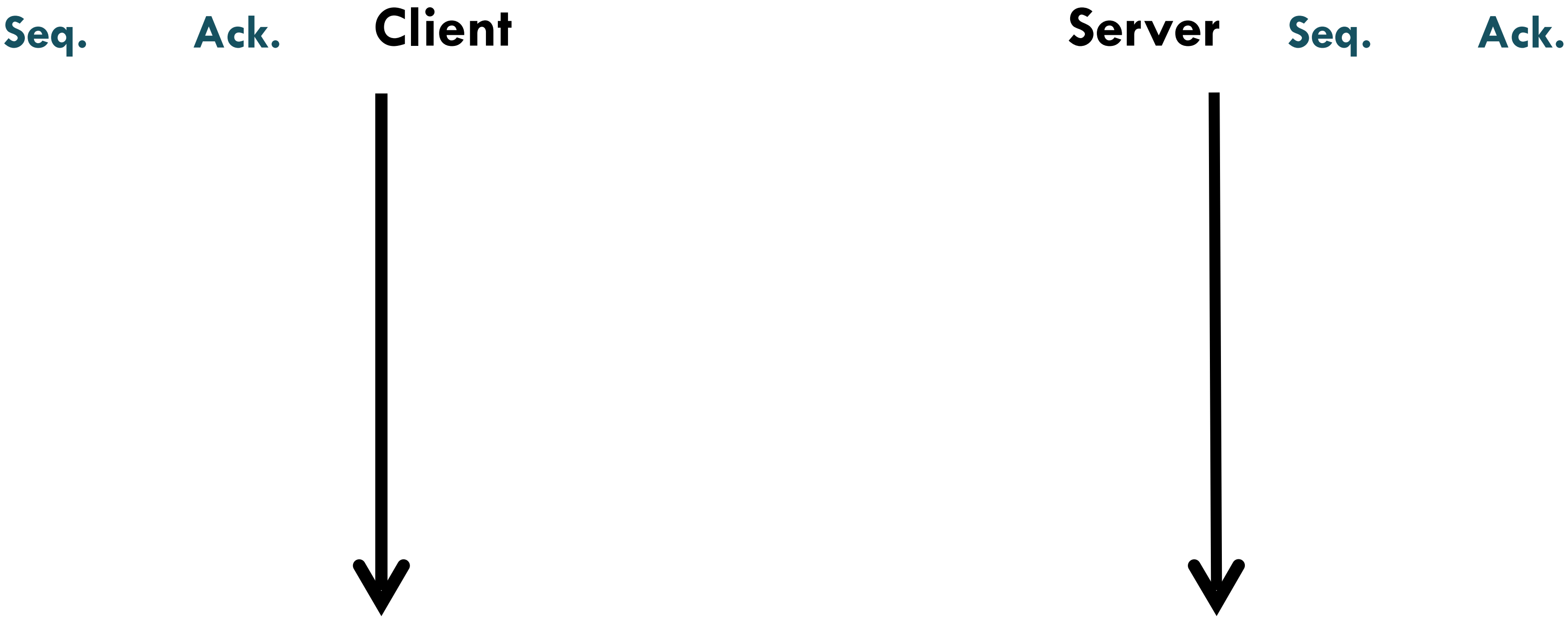
- TCP uses a byte stream abstraction
  - Each byte in each stream is numbered
  - 32-bit value, wraps around
  - Initial, random values selected during setup
- Byte stream broken down into segments (packets)
  - Size limited by the Maximum Segment Size (MSS)
  - Set to limit fragmentation

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

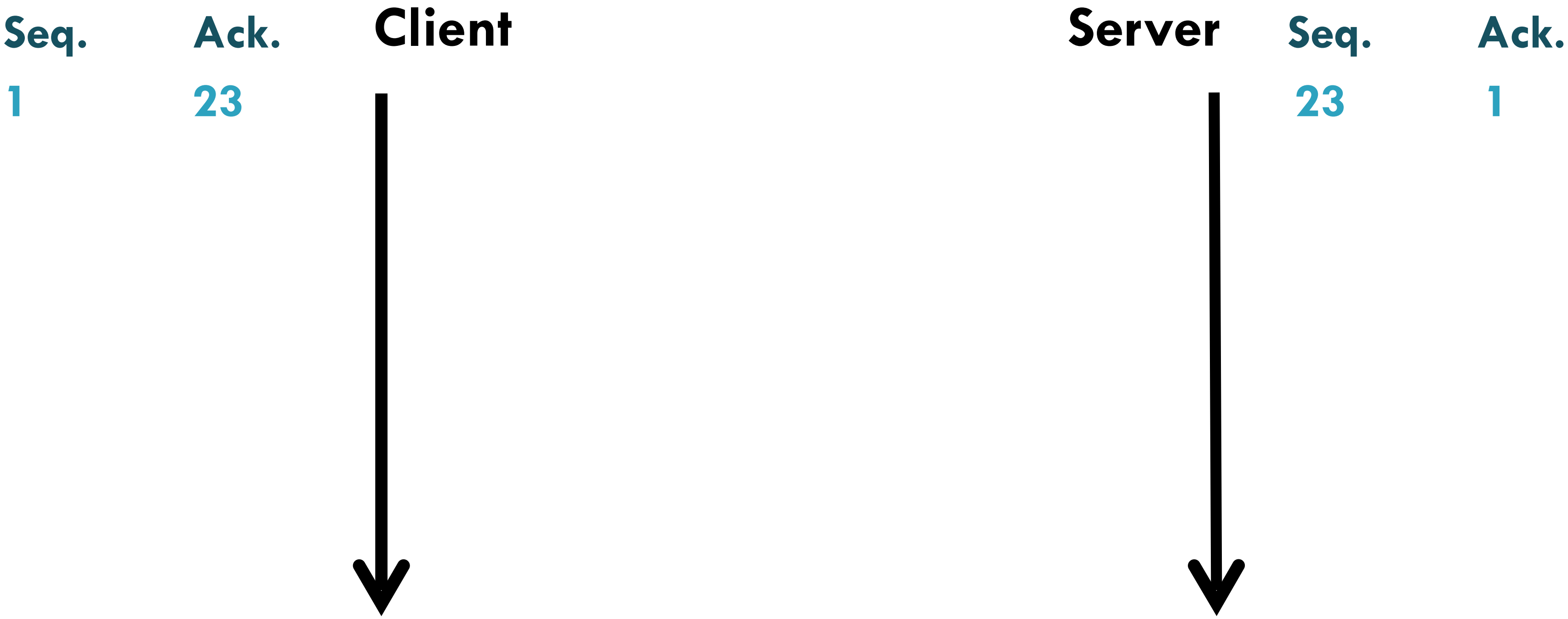


Bidirectional Communication



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

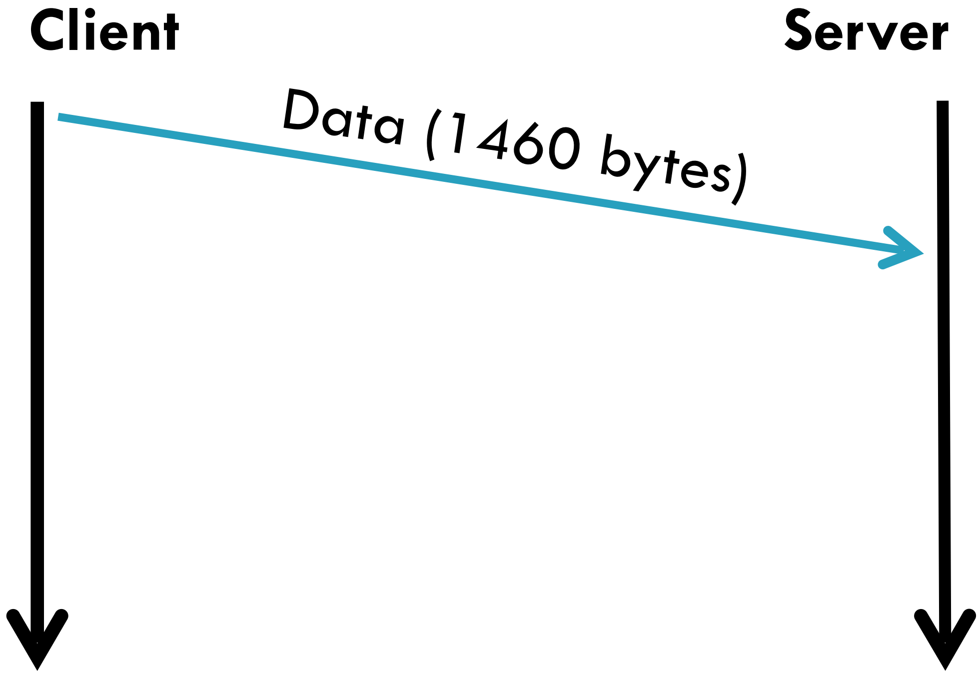
Bidirectional Communication



Seq.	Ack.	Client	Server	Seq.	Ack.
1	23			23	1

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

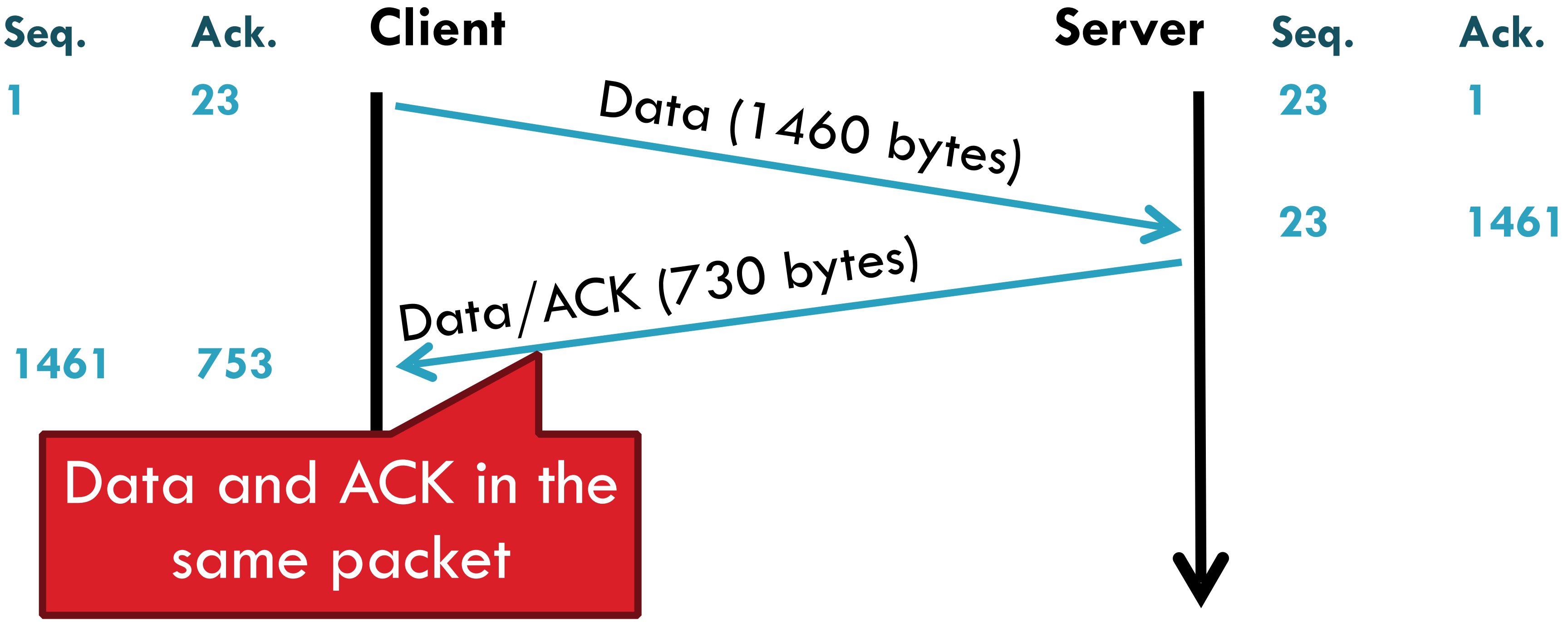
Bidirectional Communication



Seq.	Ack.	Client	Server	Seq.	Ack.
1	23			23	1
				23	1461

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

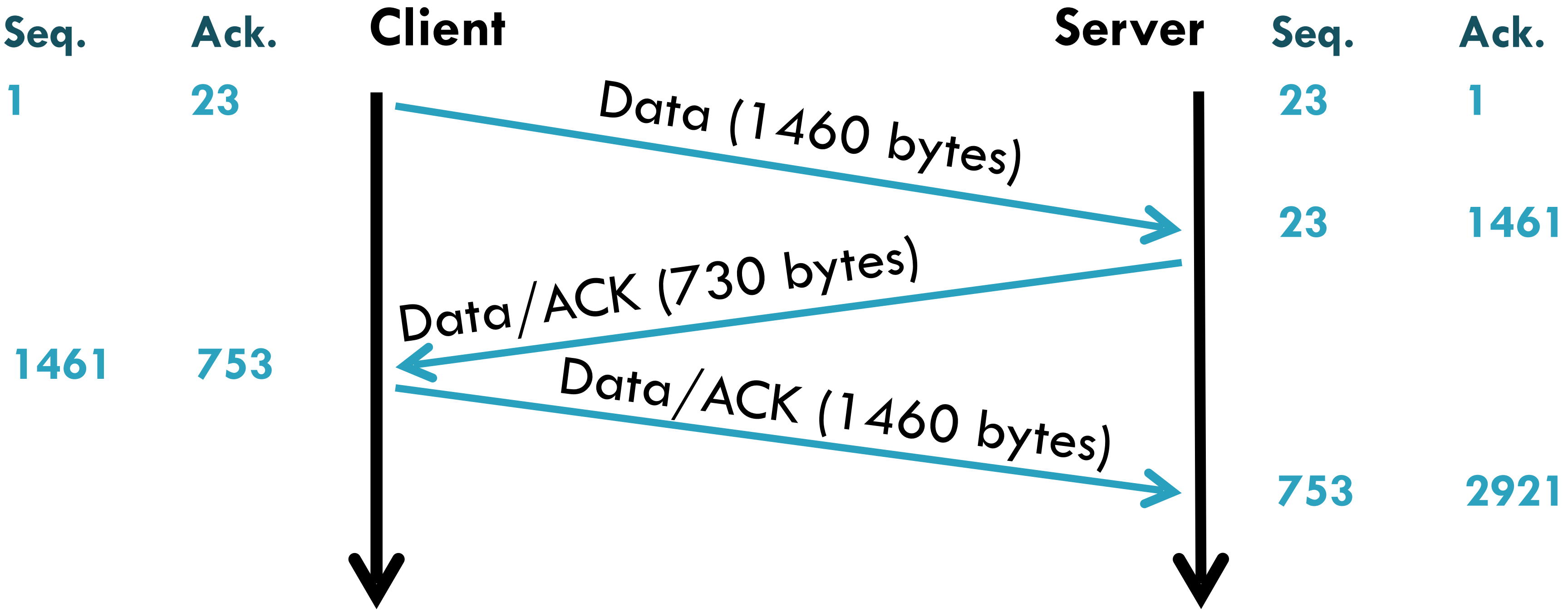
Bidirectional Communication



Data and ACK in the same packet

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

Bidirectional Communication



Seq.	Ack.	Client		Server	Seq.	Ack.
1	23		Data (1460 bytes) →		23	1
					23	1461
			← Data/ACK (730 bytes)			
1461	753		Data/ACK (1460 bytes) →			
					753	2921

- 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

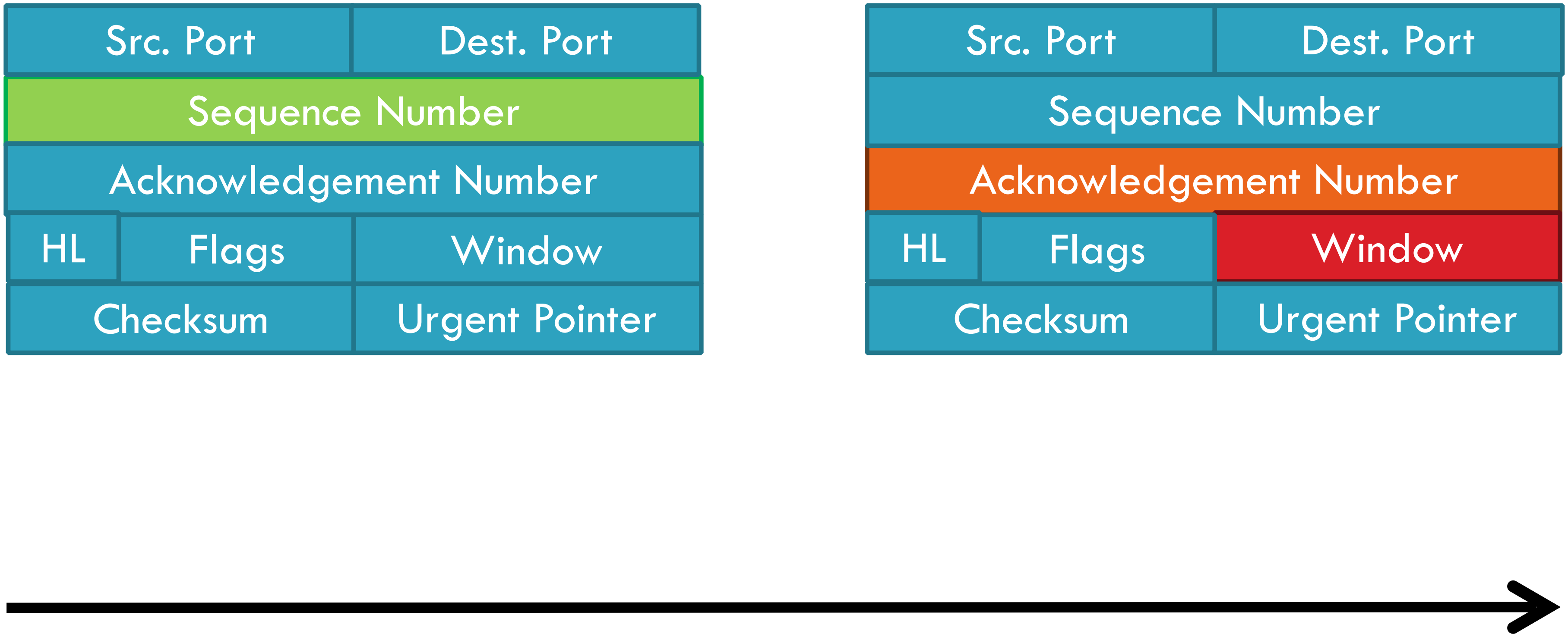
Flow Control

- 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

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



Packet Sent

Src. Port | Dest. Port

Sequence Number

Acknowledgement Number

HL | Flags | Window

Checksum | Urgent Pointer

Packet Received

Src. Port | Dest. Port

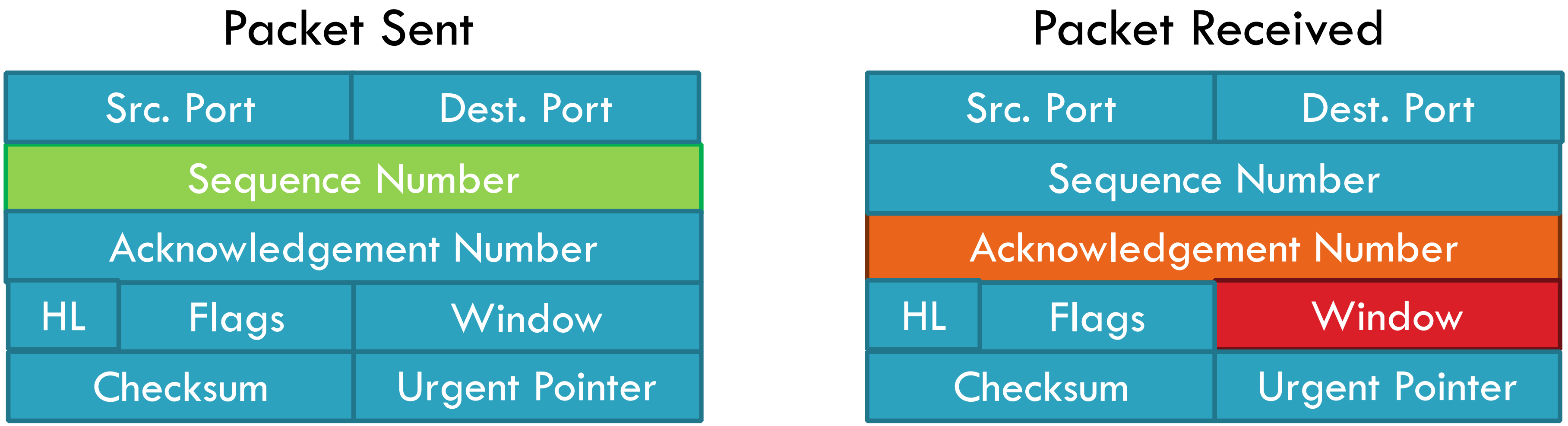
Sequence Number

Acknowledgement Number

HL | Flags | Window

Checksum | Urgent Pointer

Flow Control: Sender Side



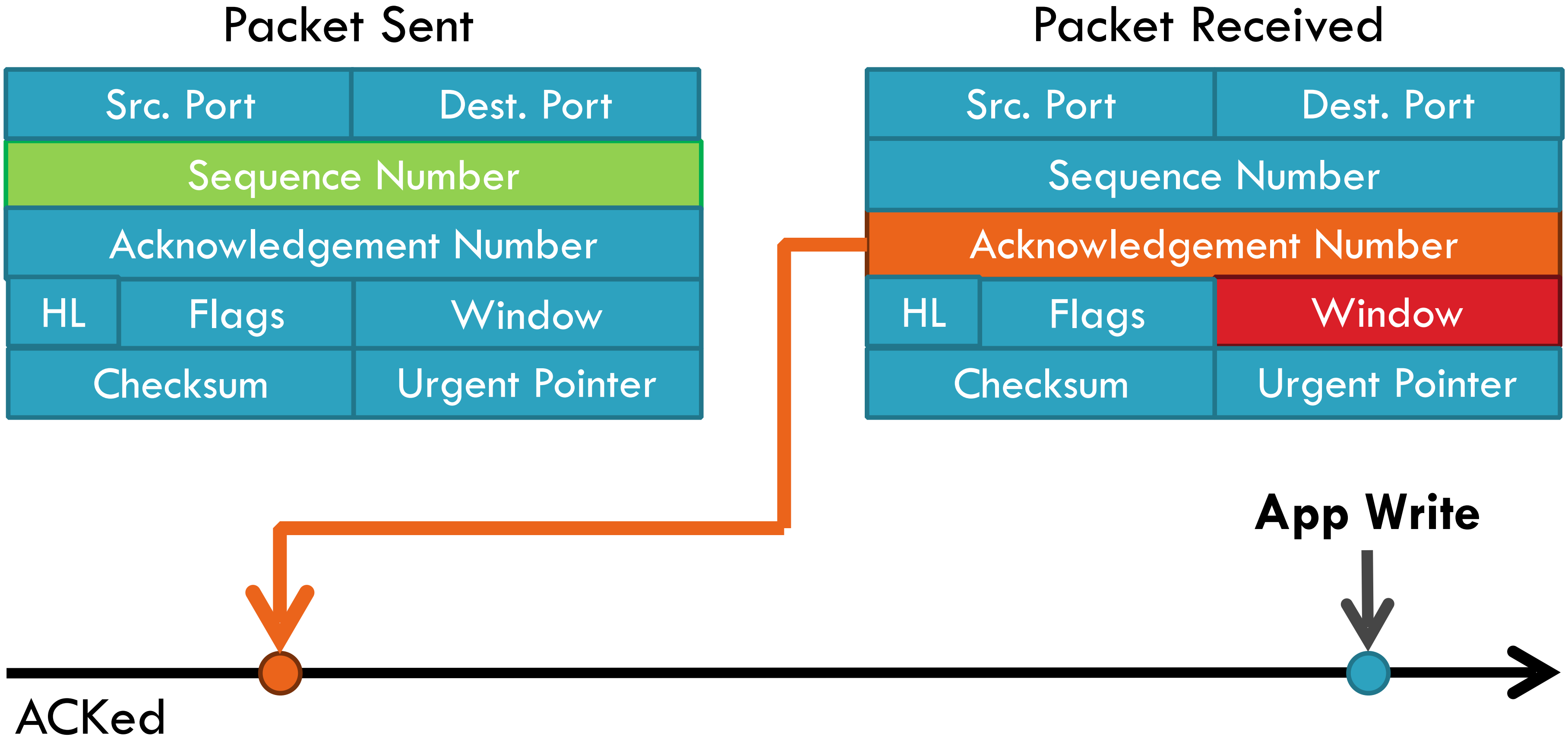
Packet Sent

Src. Port | Dest. Port
Sequence Number
Acknowledgement Number
HL | Flags | Window
Checksum | Urgent Pointer

Packet Received

Src. Port | Dest. Port
Sequence Number
Acknowledgement Number
HL | Flags | Window
Checksum | Urgent Pointer

Flow Control: Sender Side



Packet Sent

Src. Port | Dest. Port

Sequence Number

Acknowledgement Number

HL | Flags | Window

Checksum | Urgent Pointer

Packet Received

Src. Port | Dest. Port

Sequence Number

Acknowledgement Number

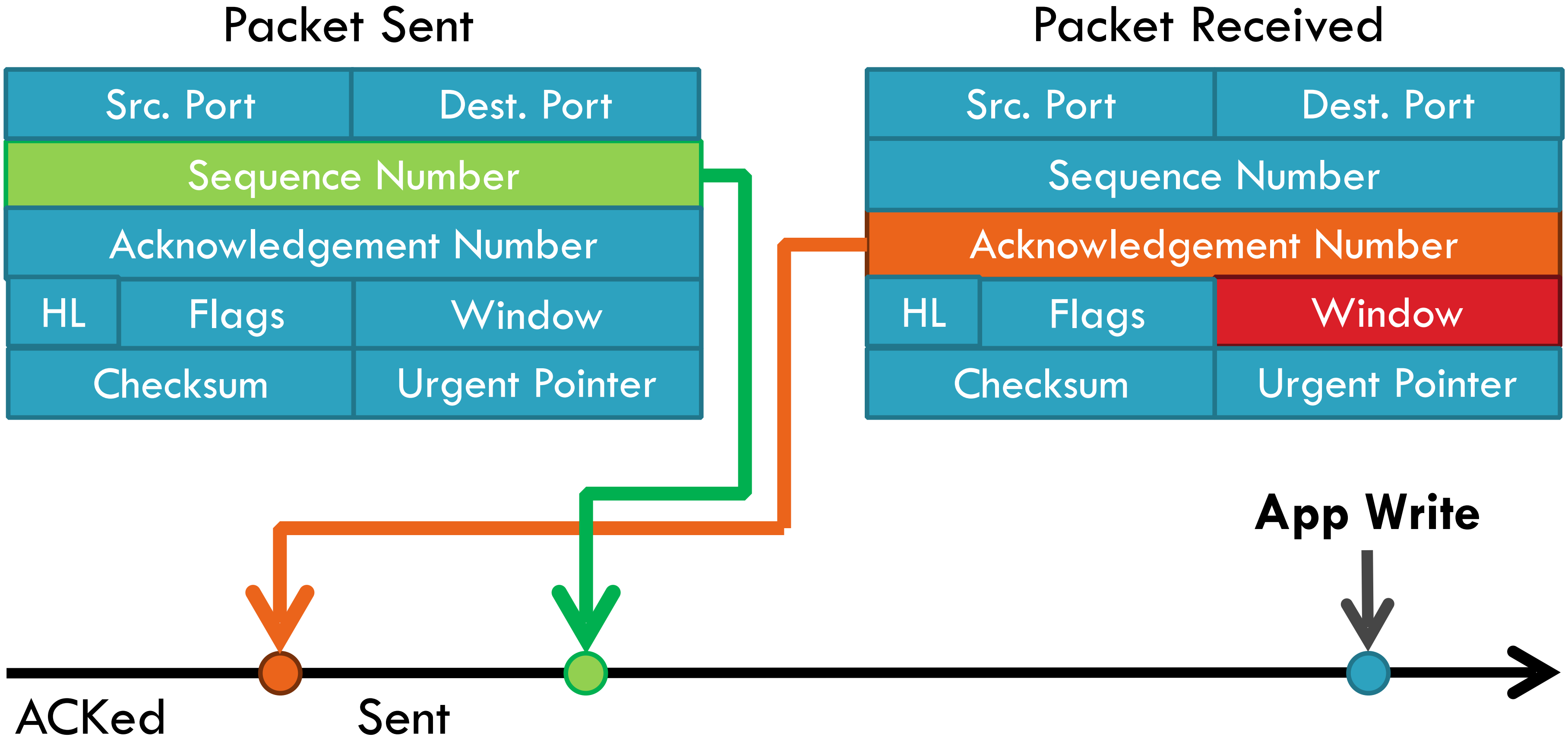
HL | Flags | Window

Checksum | Urgent Pointer

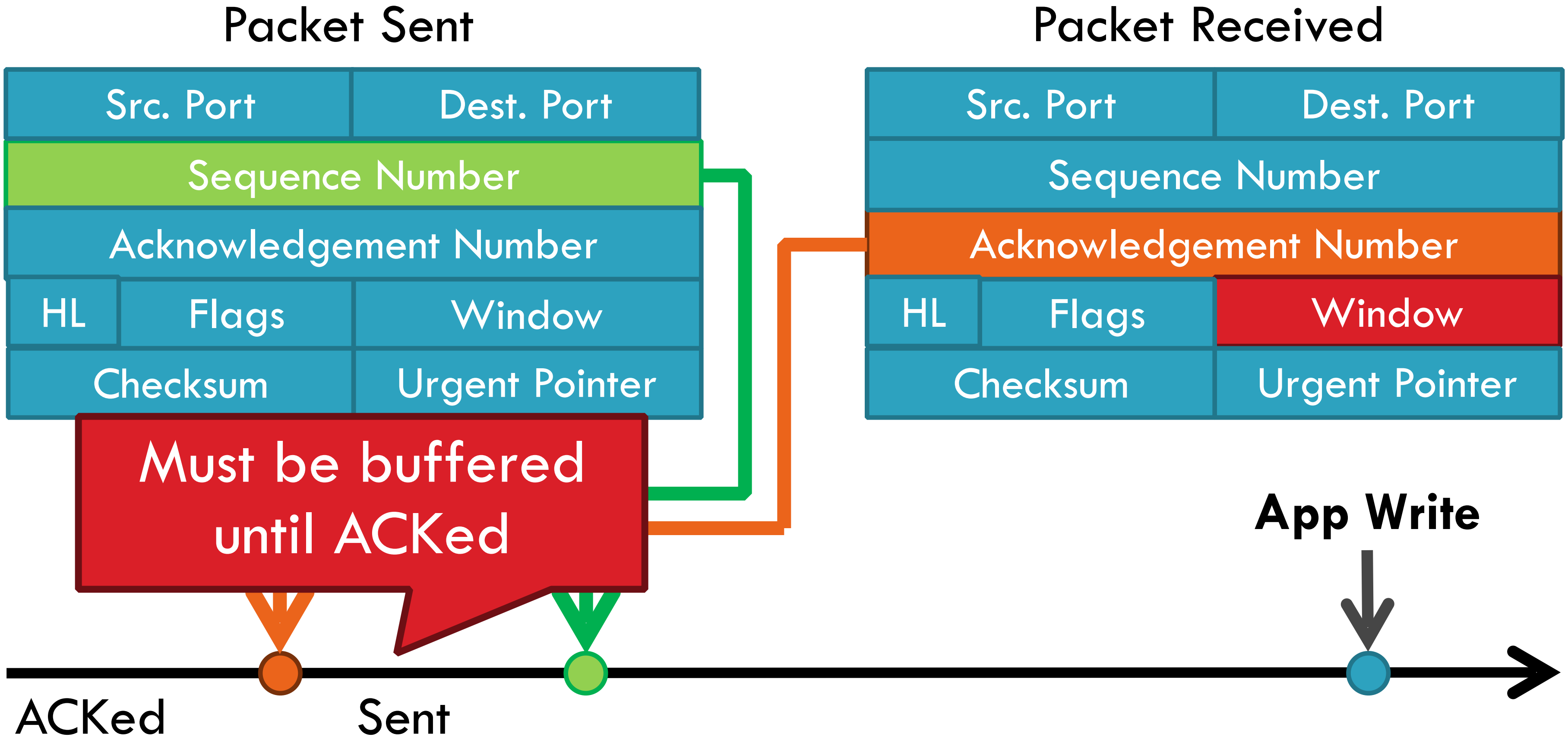
App Write

ACKed

Flow Control: Sender Side



Flow Control: Sender Side



Packet Sent

Src. Port | Dest. Port

Sequence Number

Acknowledgement Number

HL | Flags | Window

Checksum | Urgent Pointer

Packet Received

Src. Port | Dest. Port

Sequence Number

Acknowledgement Number

HL | Flags | Window

Checksum | Urgent Pointer

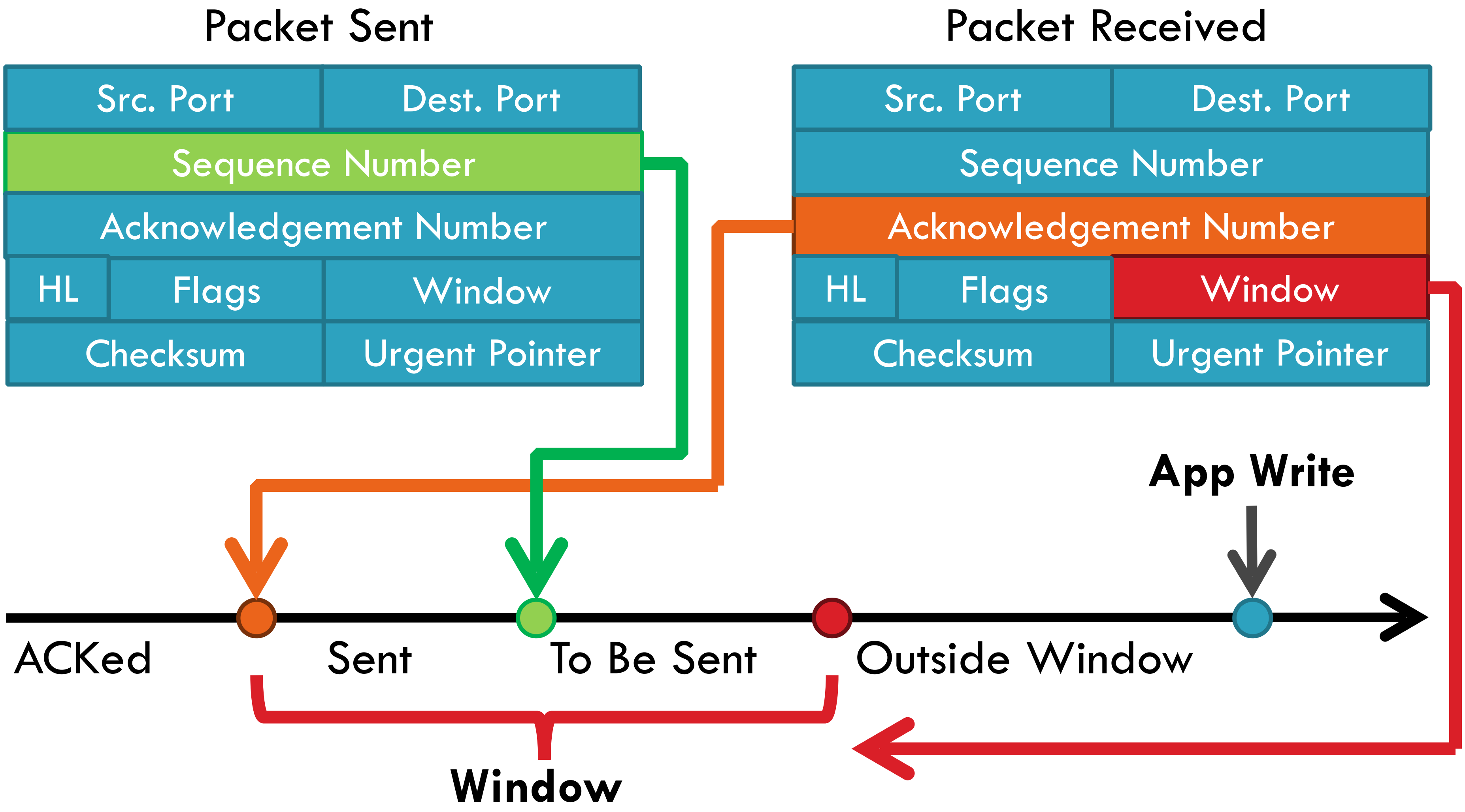
Must be buffered until ACKed

App Write

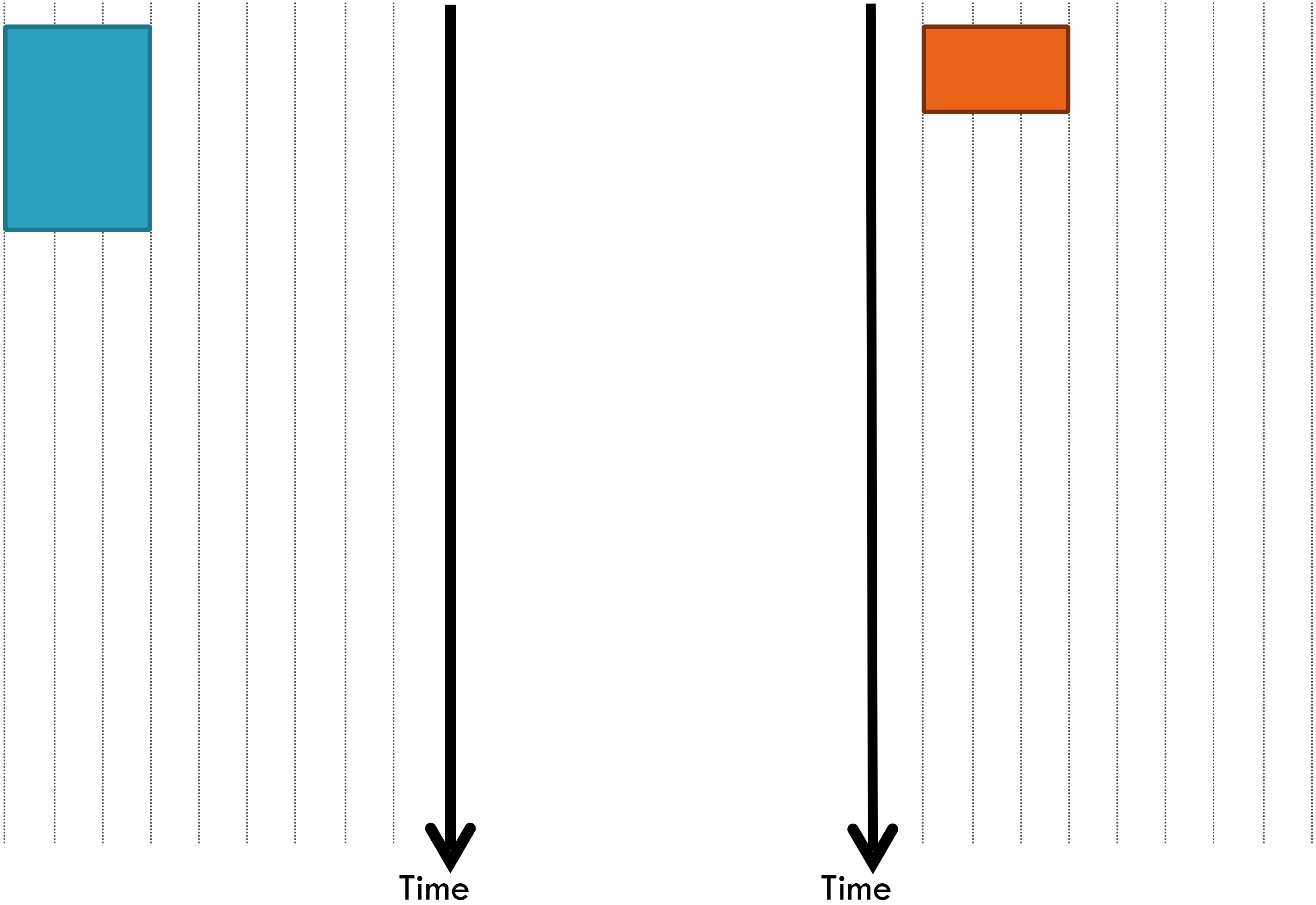
ACKed

Sent

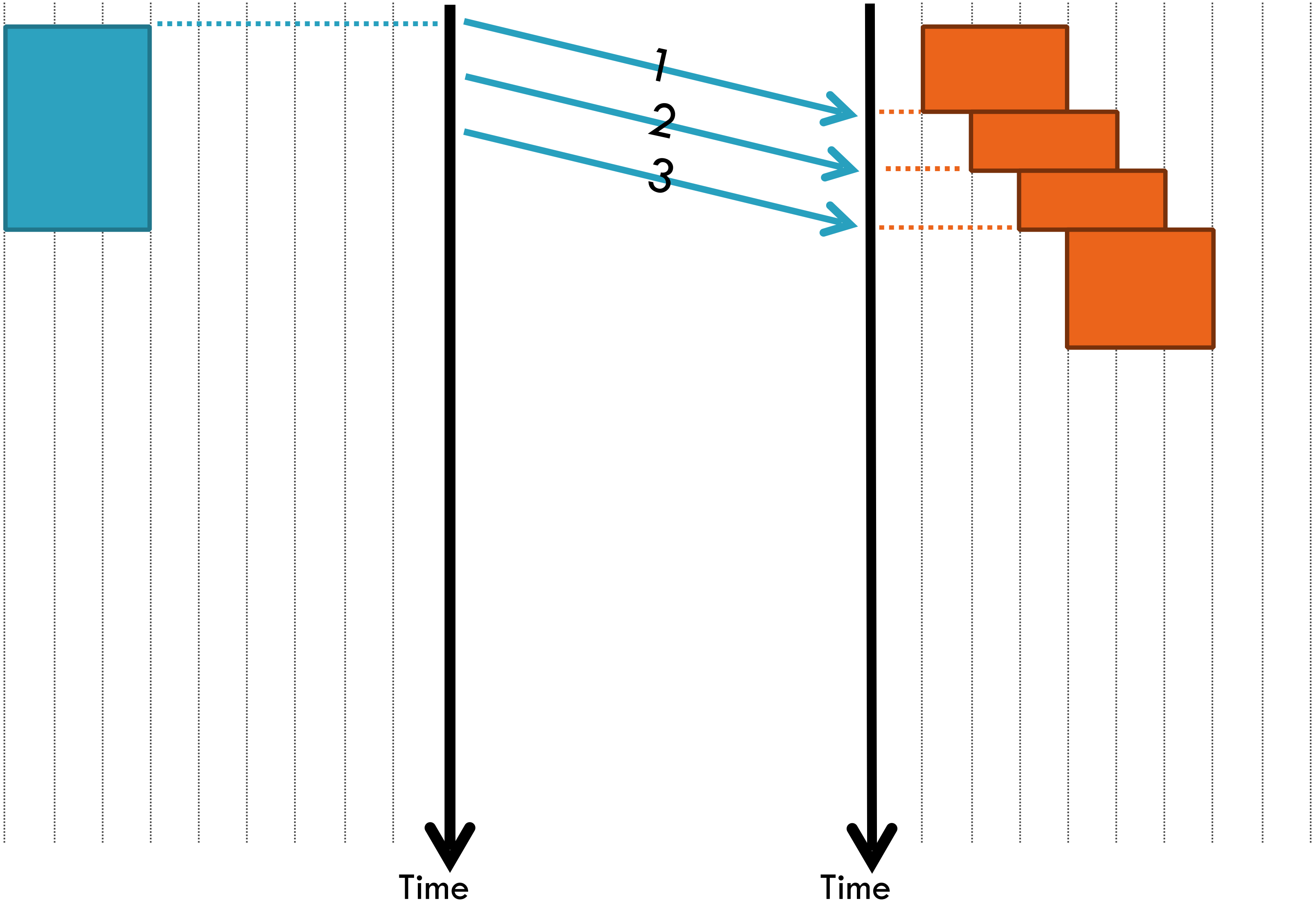
Flow Control: Sender Side



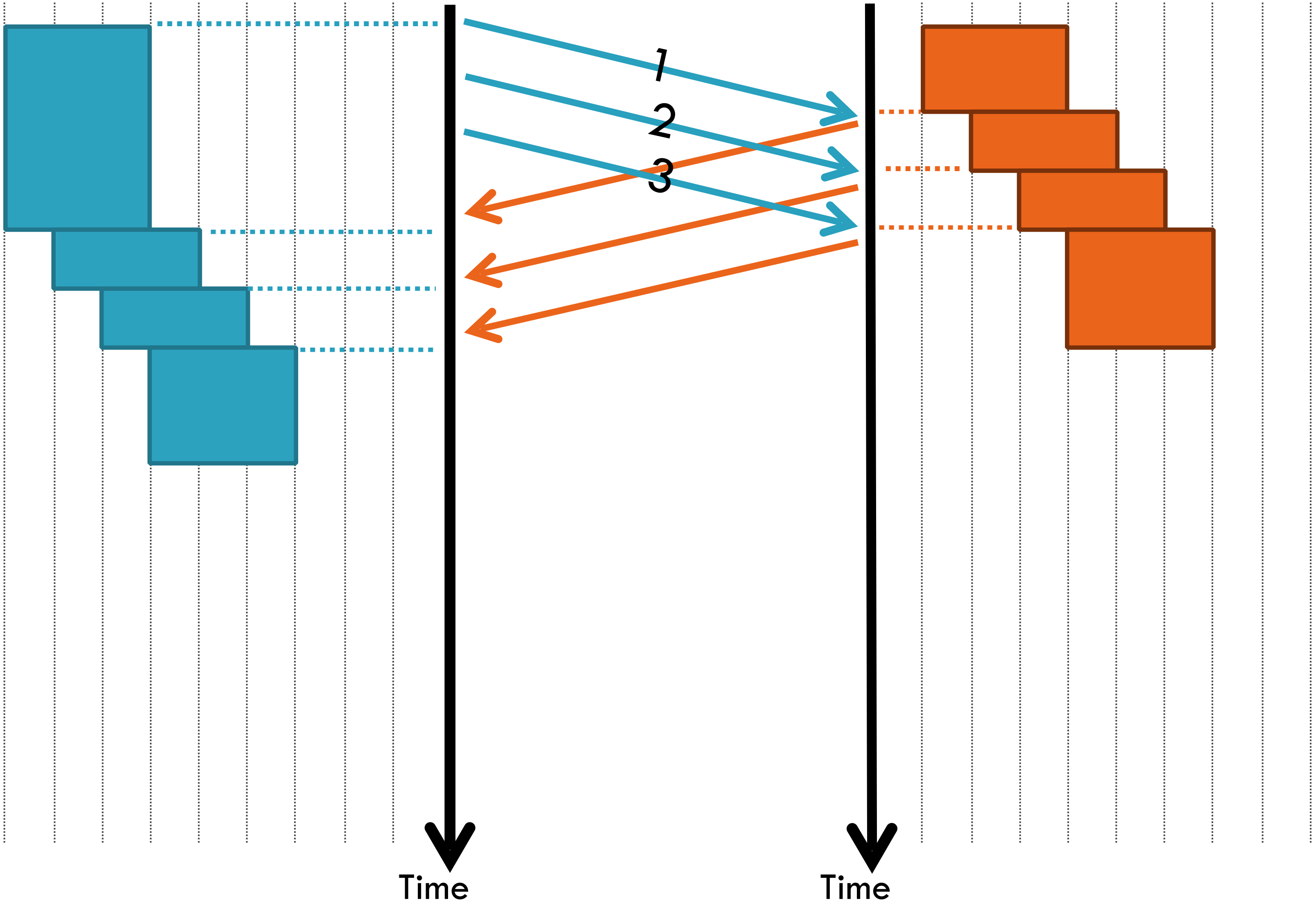
Sliding Window Example



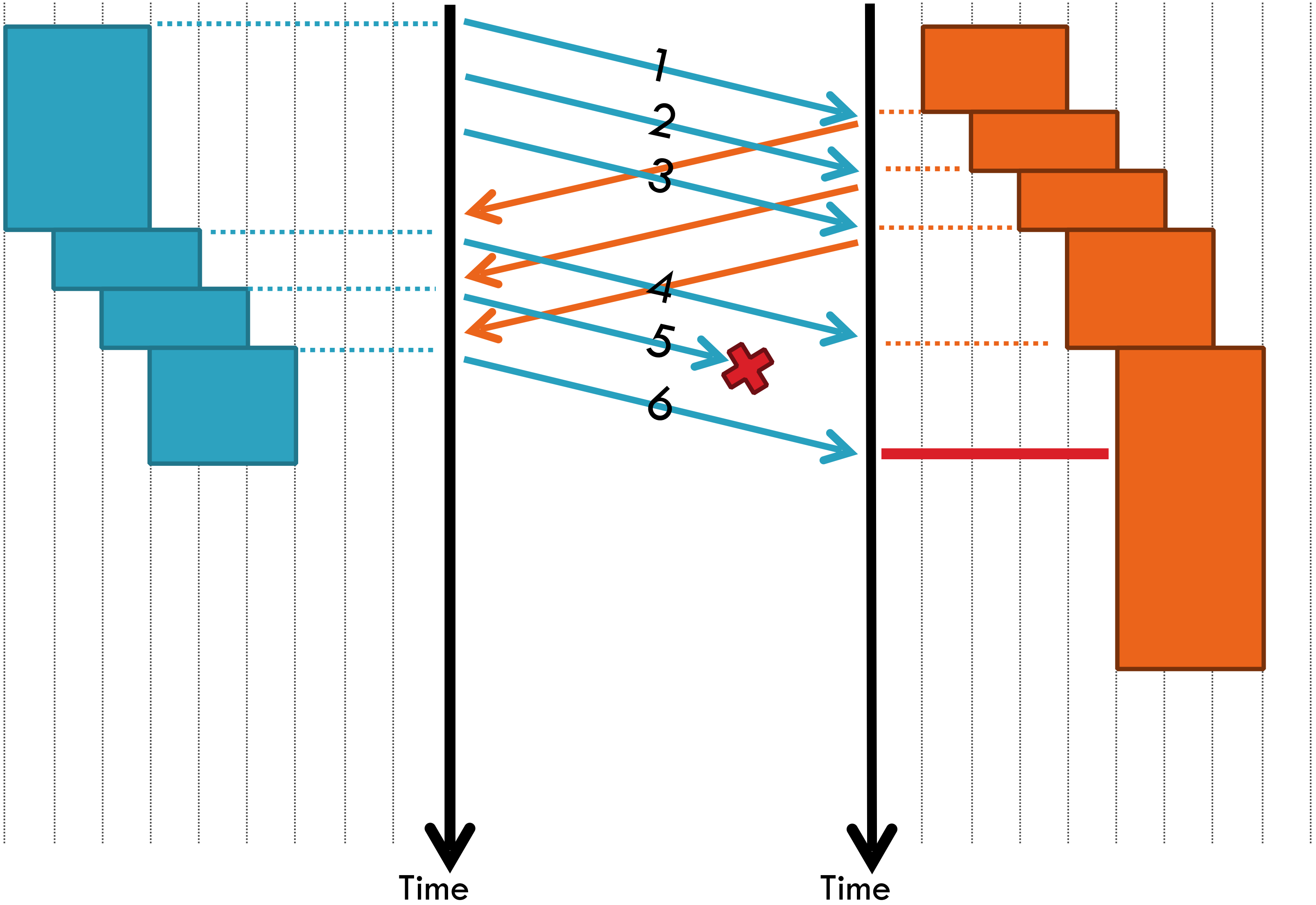
Sliding Window Example



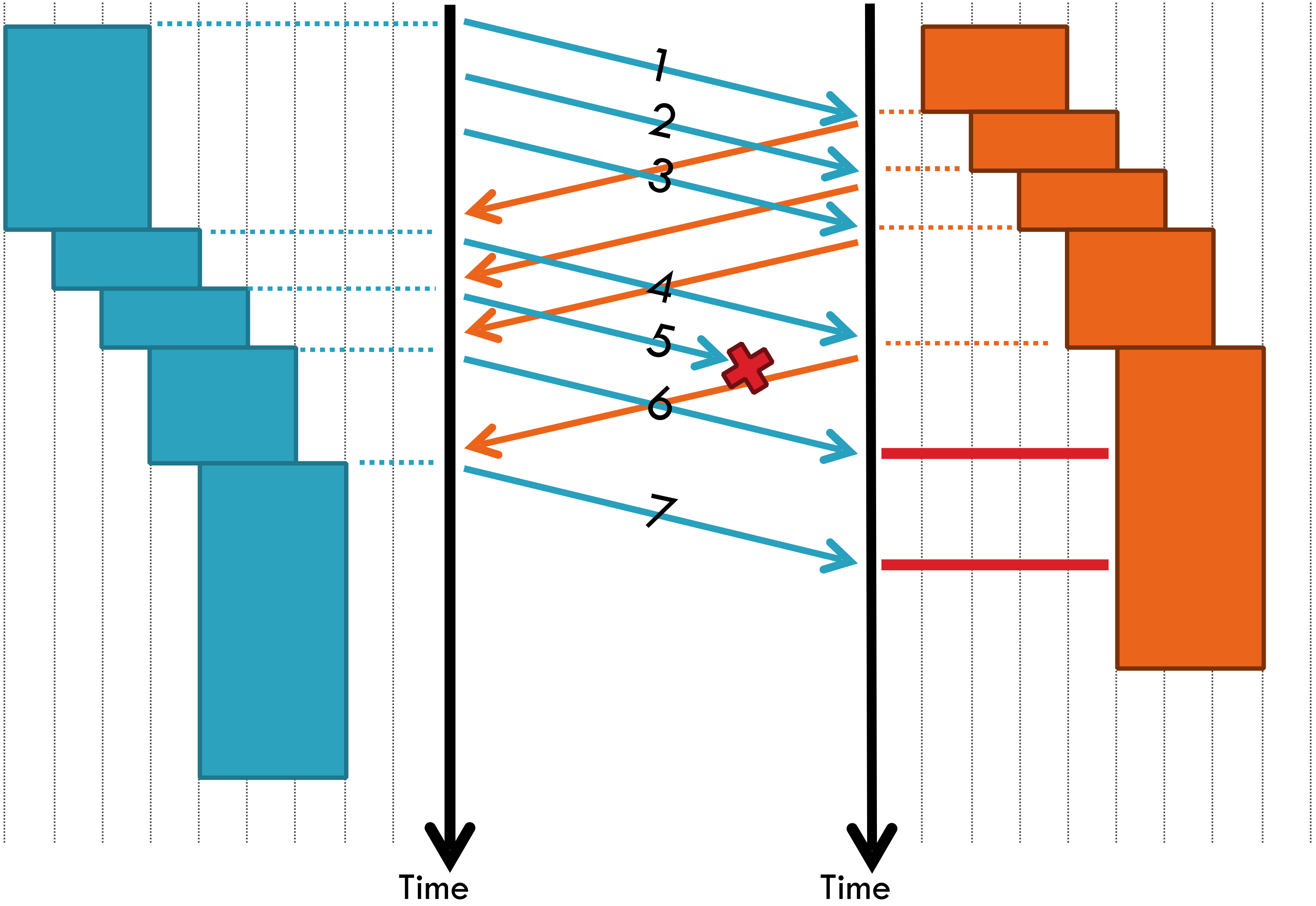
Sliding Window Example



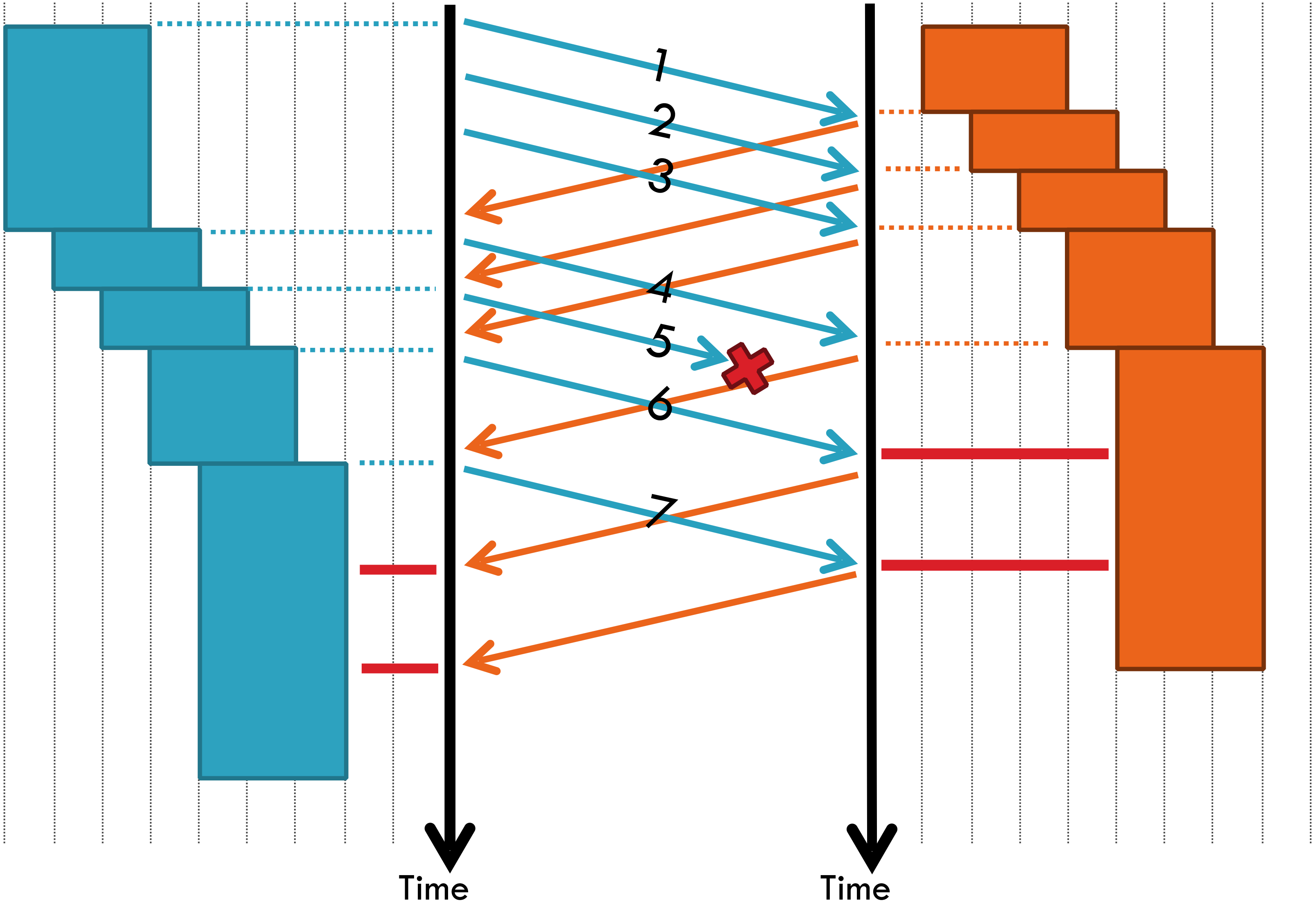
Sliding Window Example



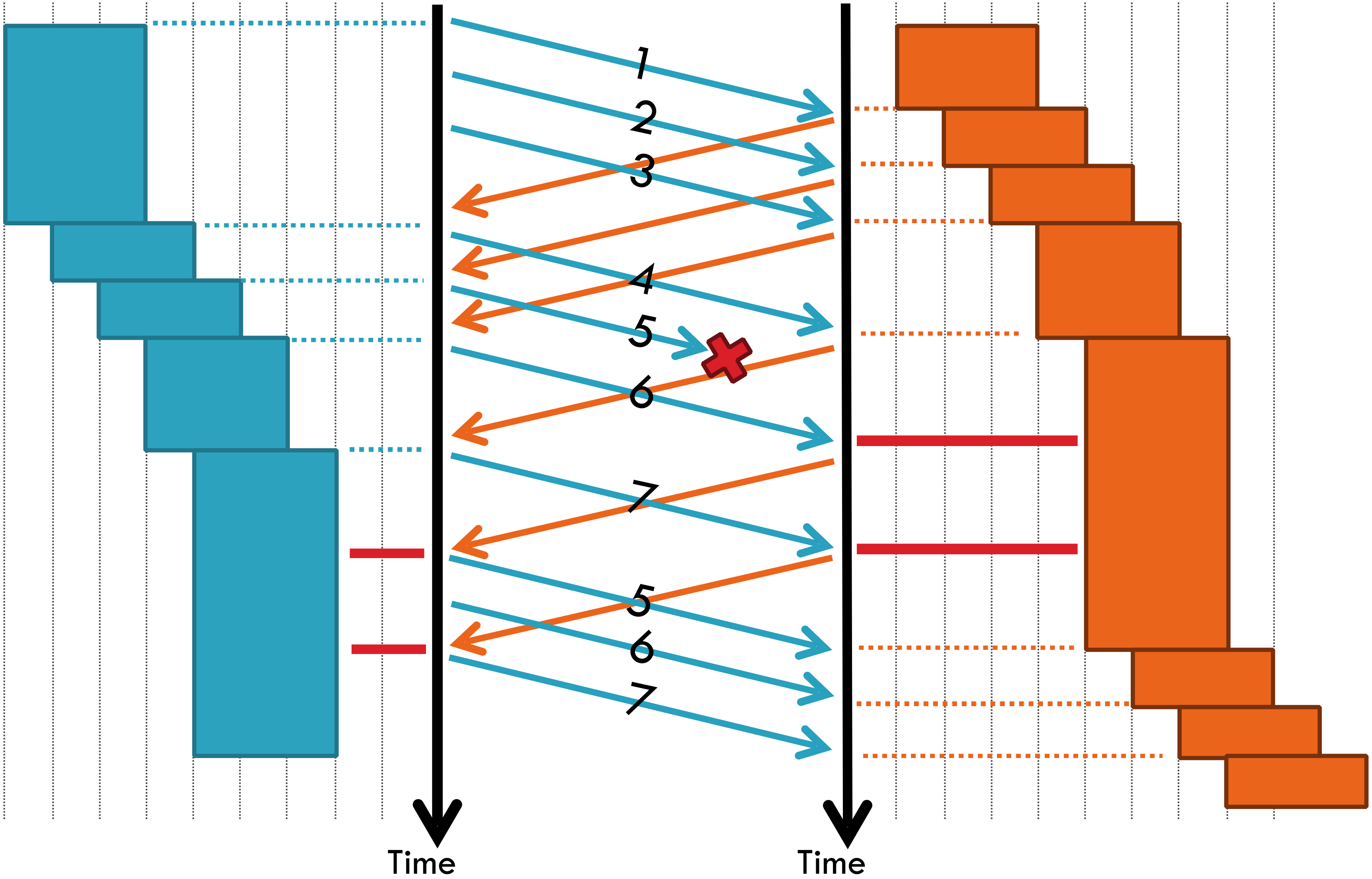
Sliding Window Example



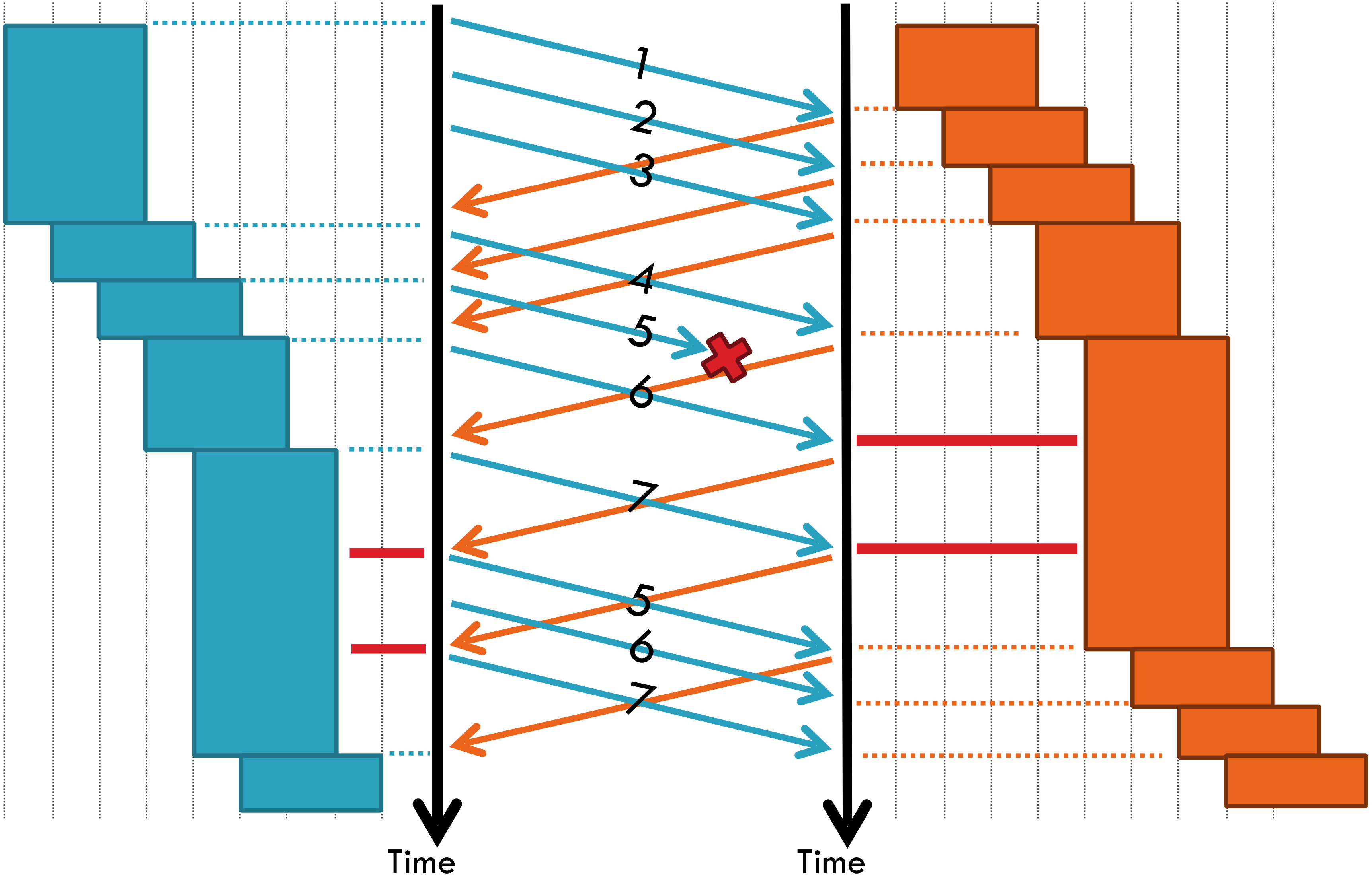
Sliding Window Example



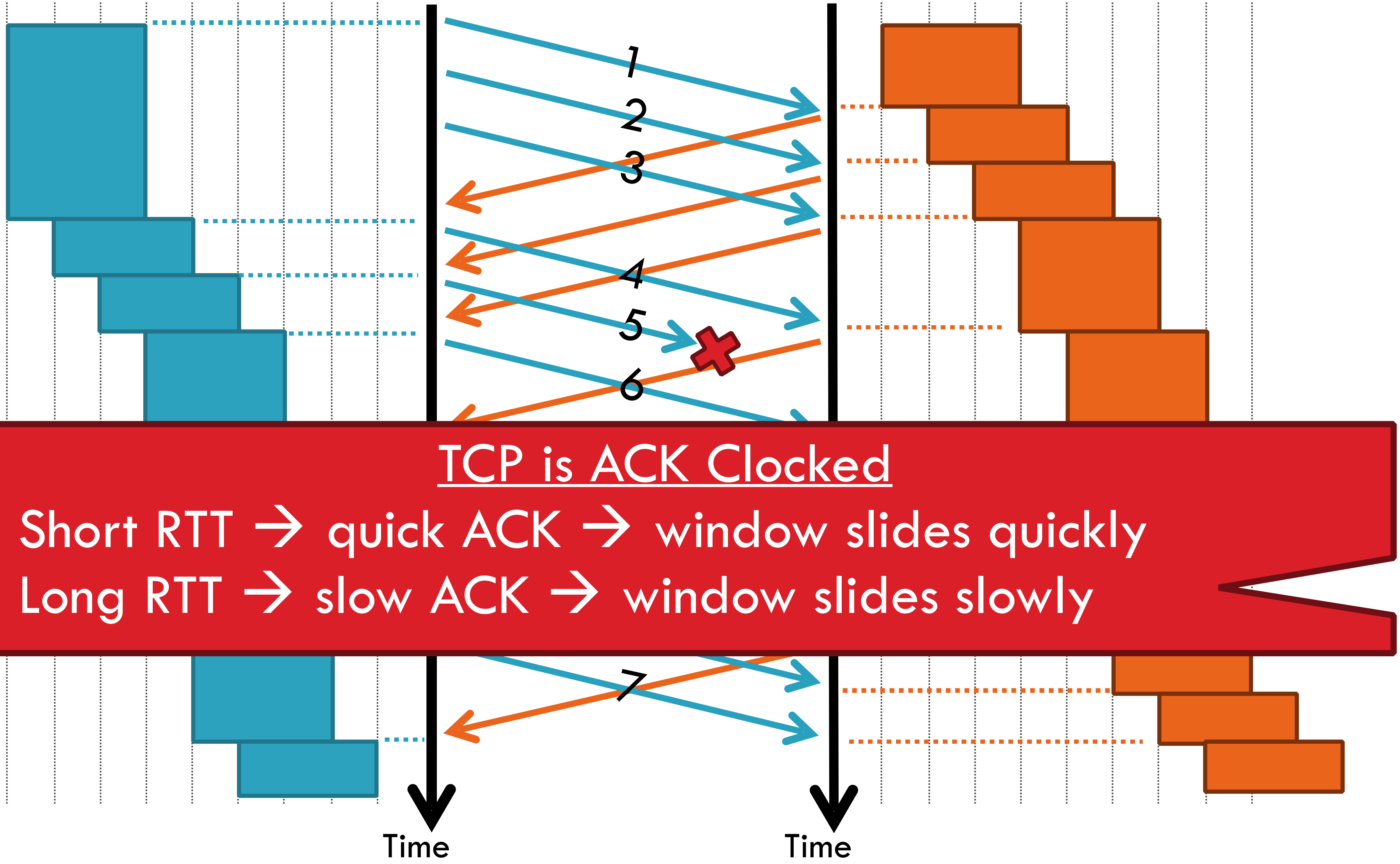
Sliding Window Example



Sliding Window Example



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

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

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

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

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?

Sequence Numbers, Revisited

- 32 bits, unsigned
  - Why so big?
- For the sliding window you need…
  - |Sequence # Space| > 2 * |Sending Window Size|
  - $2^{32} > 2 * 2^{16}$

Sequence Numbers, Revisited

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

Sequence Numbers, Revisited

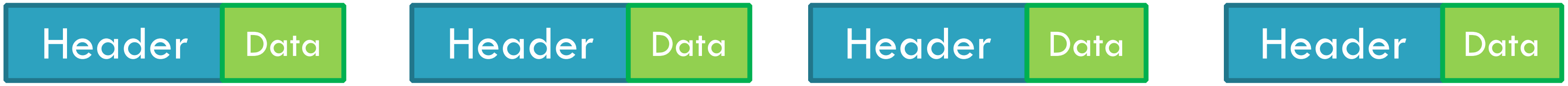
- 32 bits, unsigned
  - Why so big?
- For the sliding window you need…
  - |Sequence # Space| > 2 * |Sending Window Size|
  - $2^{32} > 2 * 2^{16}$
- Guard against stray packets
  - IP packets have a maximum segment lifetime (MSL) of 120 seconds
    - i.e. a packet can linger in the network for 3 minutes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

Silly Window Syndrome

- Problem: what if the window size is very small?

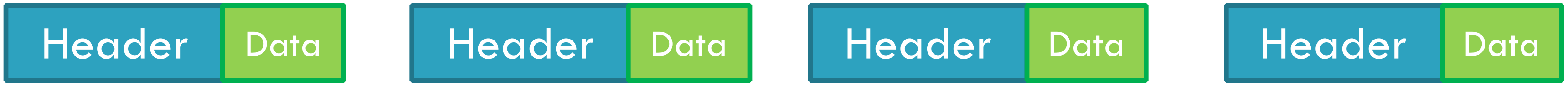
Silly Window Syndrome

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



Silly Window Syndrome

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



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

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

Nagle's Algorithm

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

Nagle's Algorithm

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

Send a non-full packet if nothing else is happening

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

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

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

Error Detection

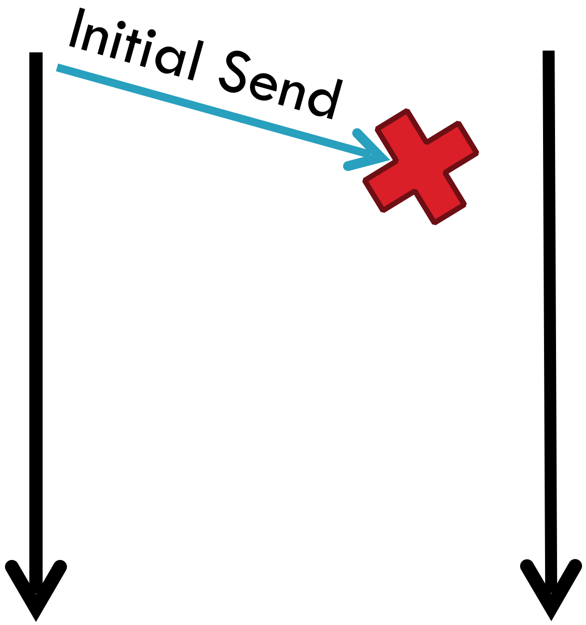
- Checksum detects (some) packet corruption
  - Computed over IP header, TCP header, and data
- Sequence numbers catch sequence problems
  - Duplicates are ignored
  - Out-of-order packets are reordered or dropped
  - Missing sequence numbers indicate lost packets
- Lost segments detected by sender
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

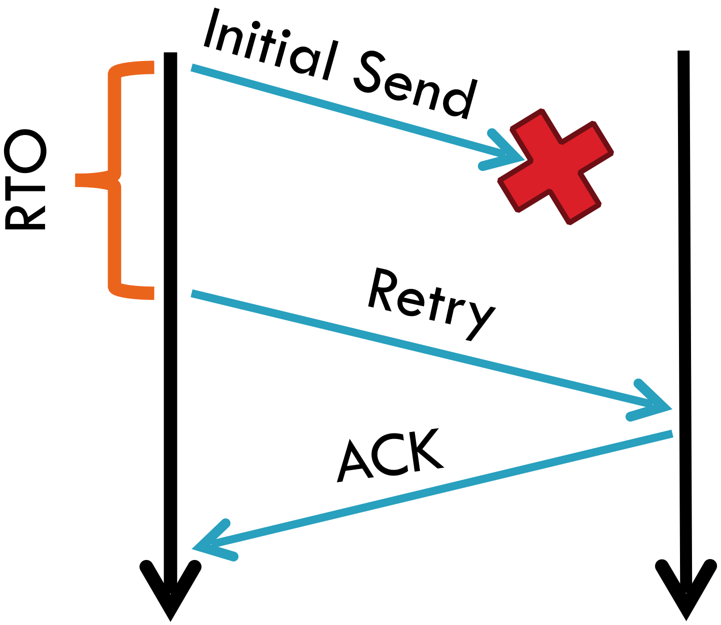
Retransmission Time Outs (RTO)

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



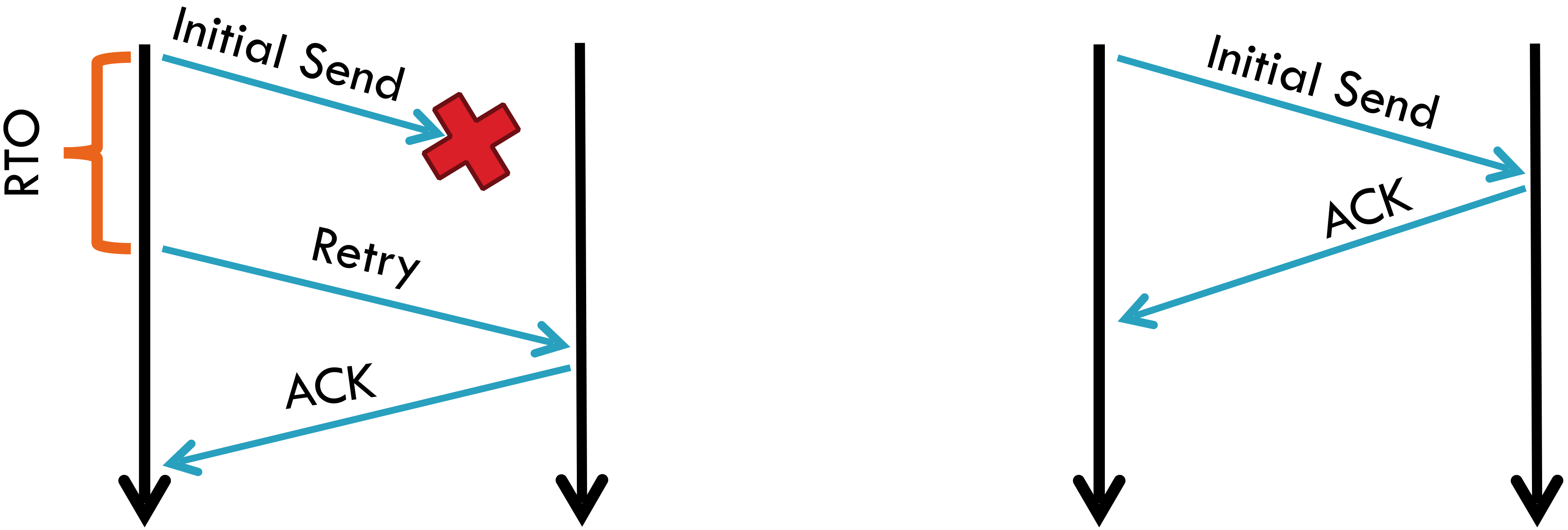
Retransmission Time Outs (RTO)

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



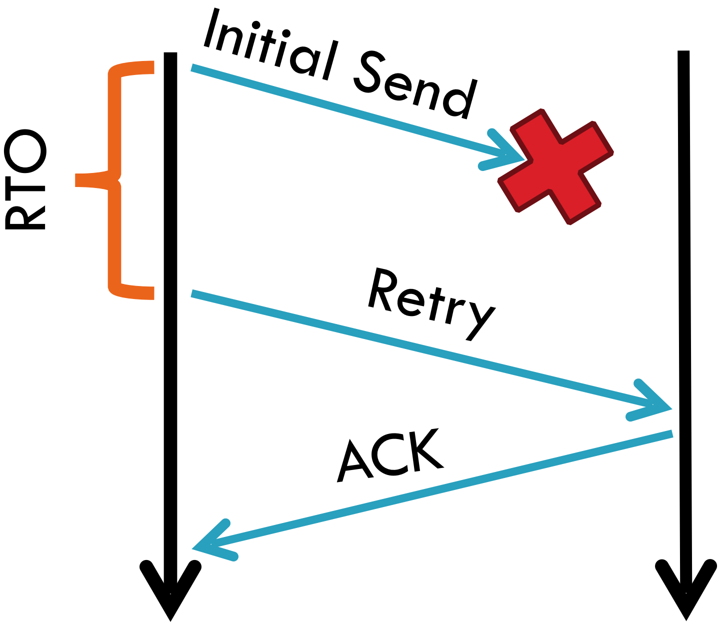
Retransmission Time Outs (RTO)

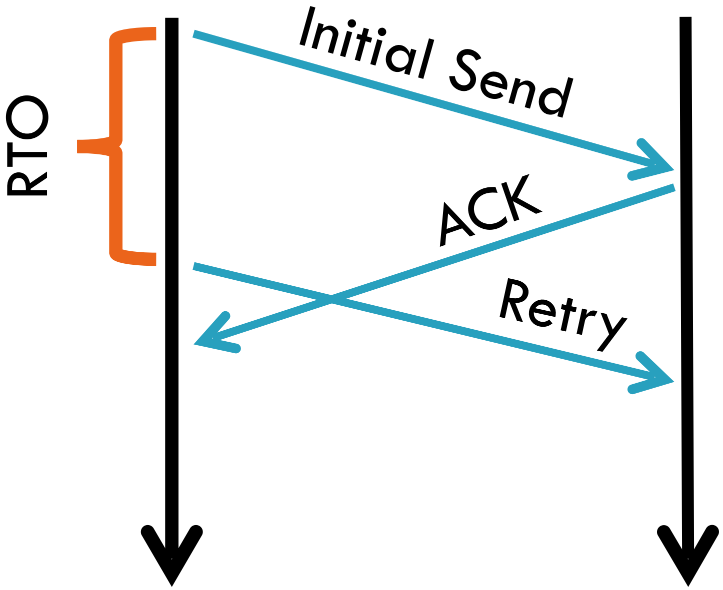
- Problem: time-out is linked to round trip time



Retransmission Time Outs (RTO)

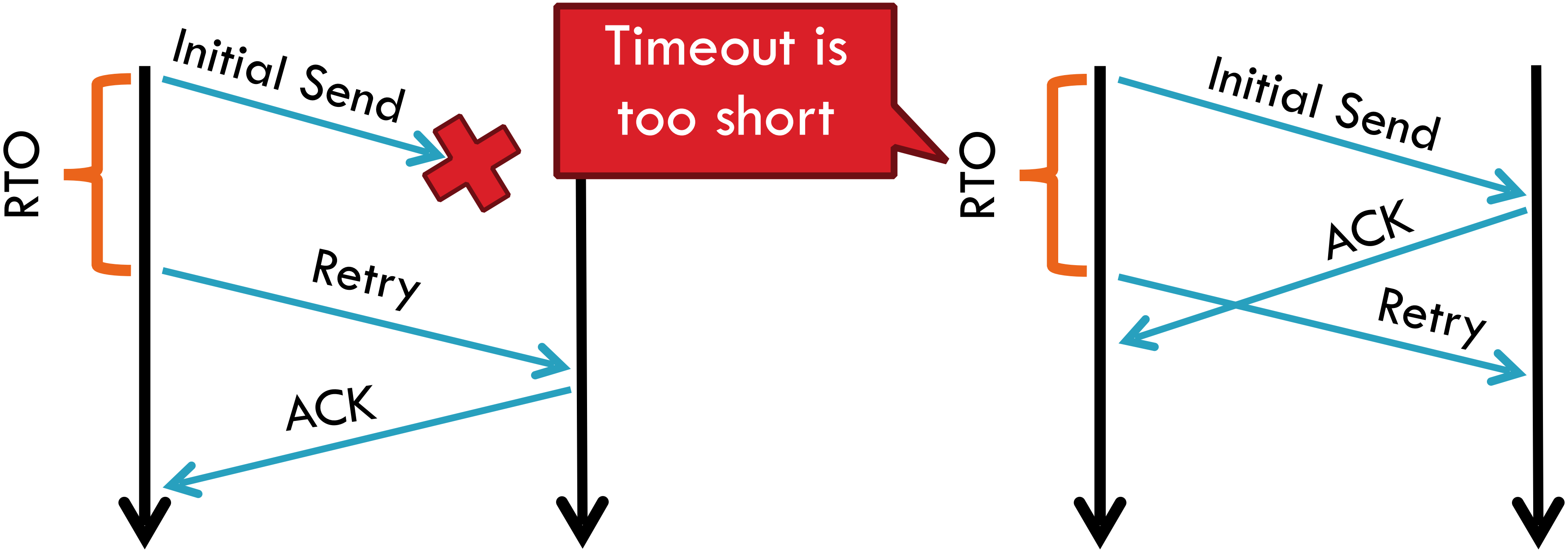
- Problem: time-out is linked to round trip time





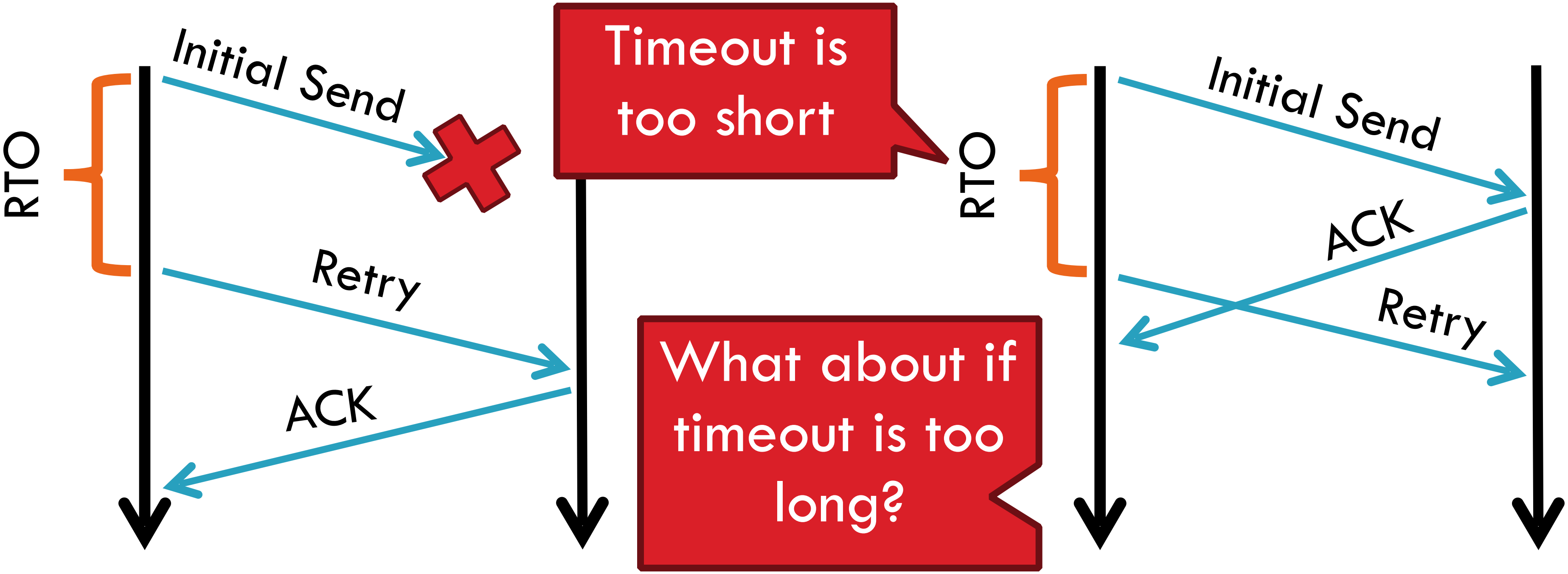
Retransmission Time Outs (RTO)

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



Retransmission Time Outs (RTO)

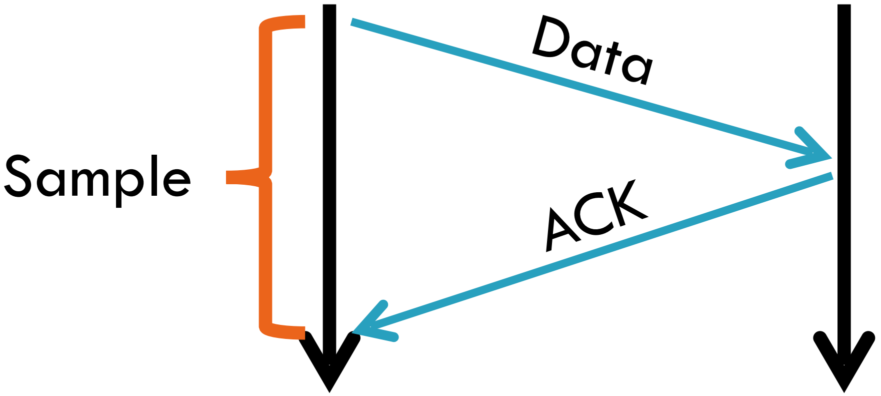
- Problem: time-out is linked to round trip time



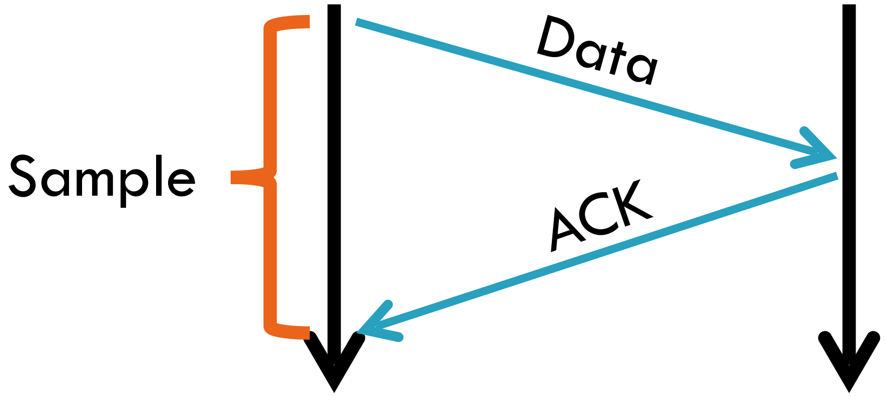
Timeout is too short

What about if timeout is too long?

Round Trip Time Estimation

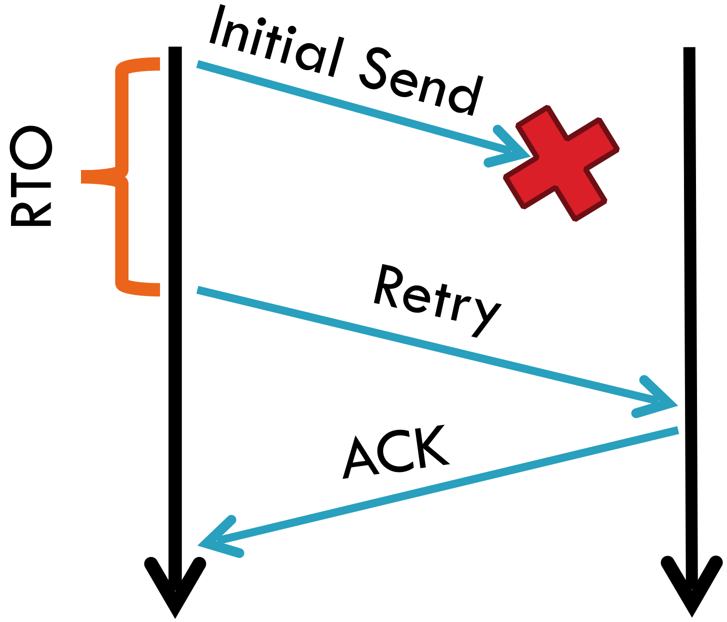


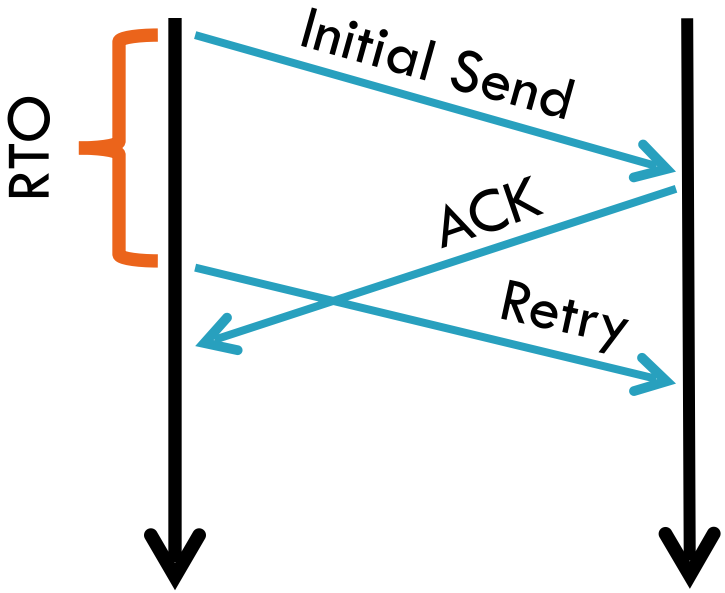
Round Trip Time Estimation



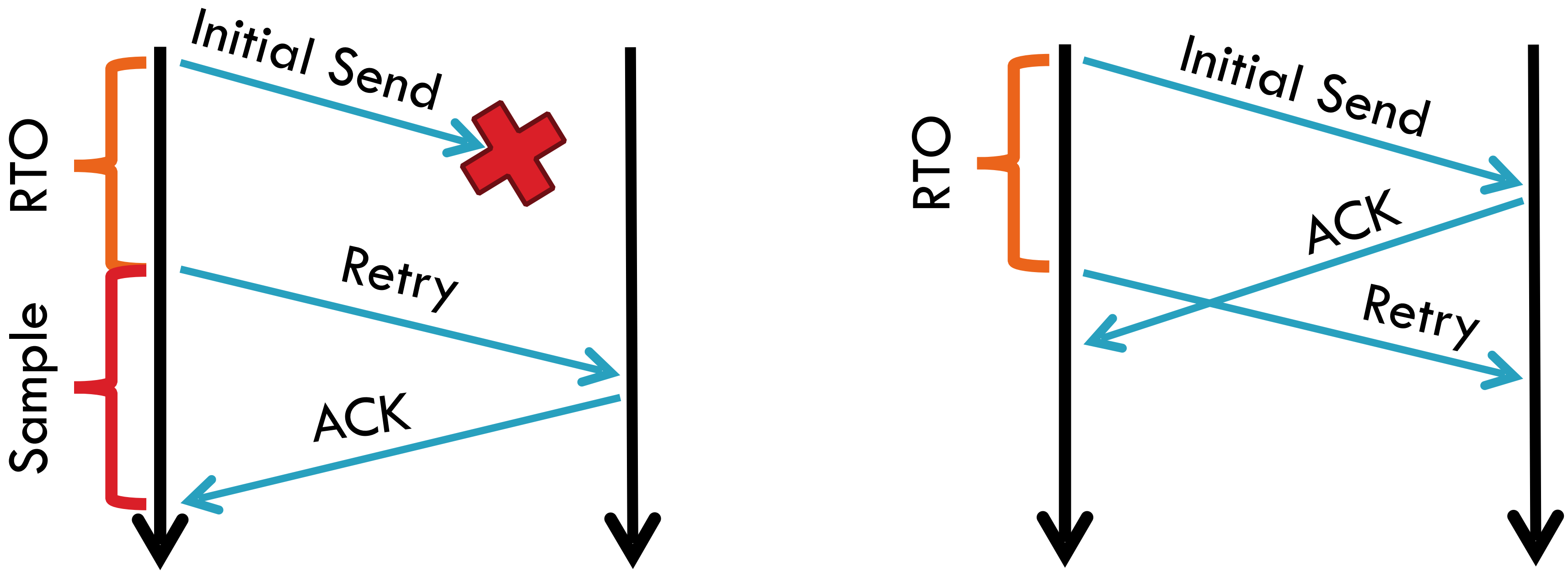
- Original TCP round-trip estimator
  - RTT estimated as a moving average
  - new_rtt = α (old_rtt) + (1 – α)(new_sample)
  - Recommended α: 0.8-0.9 (0.875 for most TCPs)
- RTO = 2 * new_rtt (i.e. TCP is conservative)

RTT Sample Ambiguity

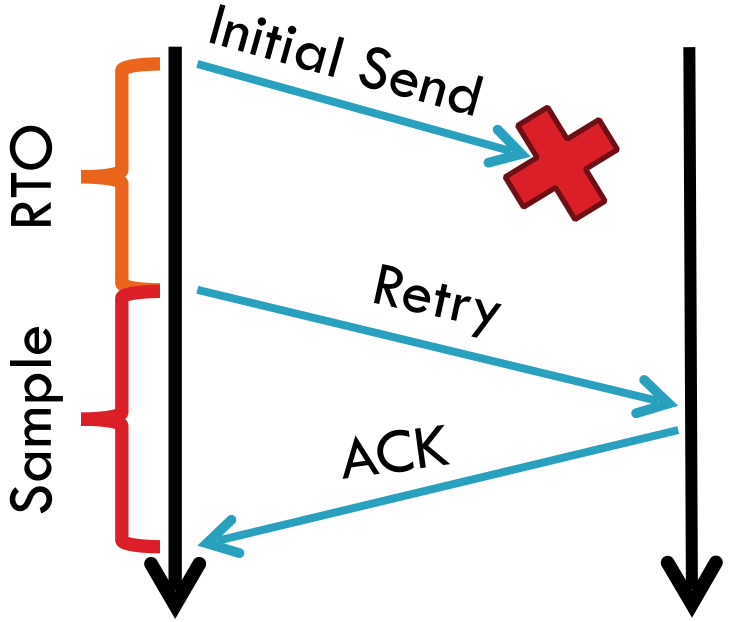


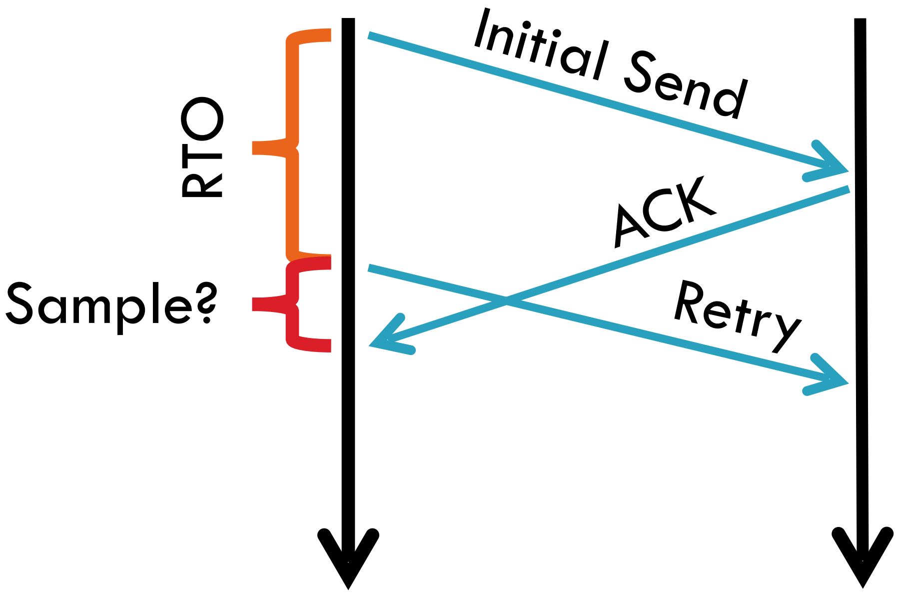


RTT Sample Ambiguity

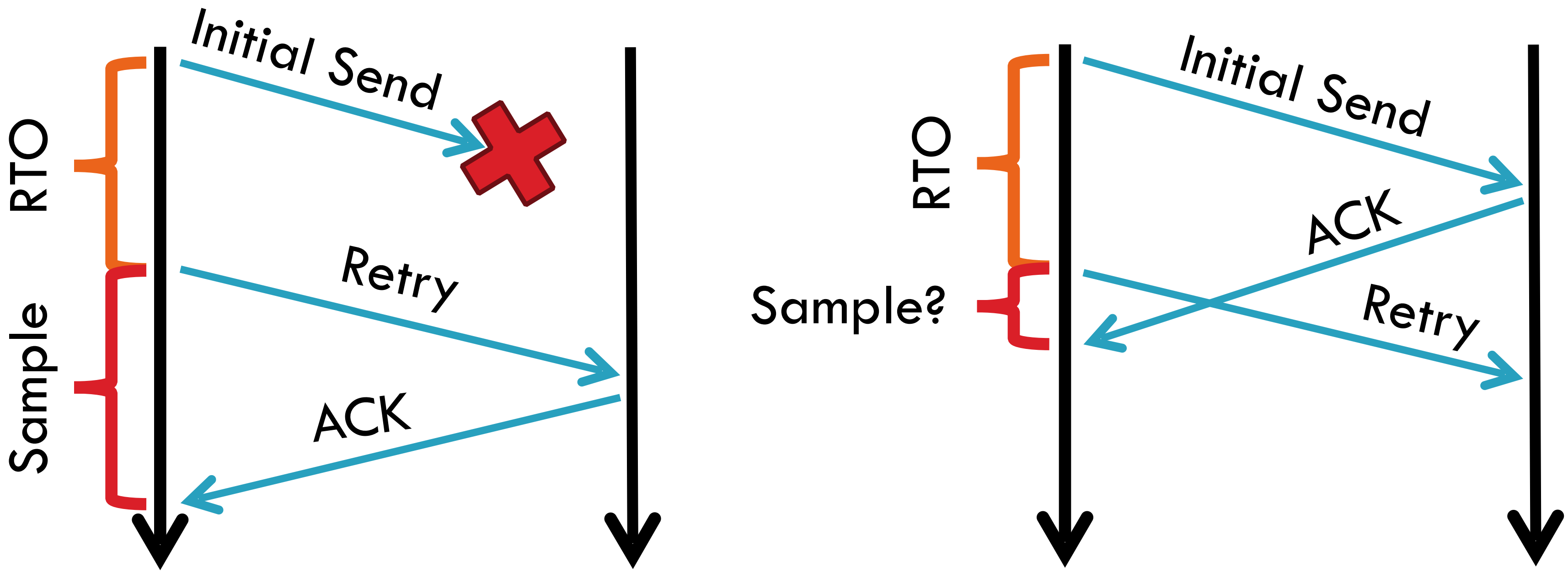


RTT Sample Ambiguity





RTT Sample Ambiguity



- Karn’s algorithm: ignore samples for retransmitted segments