Transport and TCP

EE122 Fall 2012

Scott Shenker

http://inst.eecs.berkeley.edu/~ee122/

Materials with thanks to Jennifer Rexford, Ion Stoica, Vern Paxson
and other colleagues at Princeton and UC Berkeley

Announcing Project 2

- Gautam will explain everything…

Announcements

- HW1 grading going more slowly than anticipated
- HW2 due on Thursday
- You MUST show your work!
  - Answers without reasons get no credit
  - Can’t just say “That’s what the lecture said”
- Just do your best on questions 13-20
  - Give us something, even though it might not be complete
  - E.g., the key change in the routing table

Clarification #1

- Addresses in packets do not have masks
- Router has masks for entries, so it knows prefixes
- Longest prefix match means:
  - See which prefixes a packet fully matches
  - Pick longest prefix which is fully matched
- **What this does not mean:**
  - Check packets against all routes and see which ones they agree with on the most bits….
  - E.g., routes 101*******(/3)… and 1*******(/1)…
  - Packet 100……

Clarification #2

- **What's the difference between the physical layer and the data-link layer?**
- Blurry line as to which functionality belongs where
- But data-link delivers packets, with semantics in the packet headers about local destinations, etc.
- Physical layer just delivers bits, typically just to the logical endpoint of the connection (or broadcasted)
  - No routing as part of the definition of the layer

Clarification #3

- **When is a port not a port?**

- When one is a transport port, and the other is a switch port.
  - **The two have nothing to do with each other….**

Clarification #4

- **Please do not post your project code!**

- **We have two choices:**
  - Come up with new projects every year
    - Frequently ends in disasters, students not happy
  - Reuse projects, hone them until everything works
    - But we can't have project code being posted

- So don't post your code!

Agenda

- My proposal for addressing
- Transport Layer
- TCP
- I have 90 slides, so fasten your seat belts…

My Addressing Proposal

My proposal for addressing

- Return to original IP addressing scheme (mostly)
  - Network name followed by host name
- Domains use any host naming system they want
- Can have a hierarchy of network addresses
  - Examples: Network:Host or N1:N2:H
- All names tied to keys
  - N is hash of network's public key
  - H is hash of host's public key

Advantages

- Addresses are verifiable (challenge-response)
  - *Prove to me that this is your address!*
  - N signs something and sends it with his public key
- Multihoming natural: host is both N1:H and N2:H
- Routing is exact match (much easier)
- Scaling not a problem…
  - Not that many network addresses
  - Can add extra layers of hierarchy if needed

Back to the future

- Original Internet addressing scheme was perfect
- Except:
  - Not enough network addresses
  - Fixed format for host addresses
  - No cryptographic verification of addresses
- Solution does not address anonymity

Biggest advantage…..

- Interdomain routing done just on N addresses
  - Everyone must understand N addresses
- Intradomain routing done on H addresses
  - Only my domain needs to understand H addresses
  - Domain could unilaterally upgrade from IPv4 to IPv6
- Universal agreement only on domain addressing
  - Which is what the original network design called for…

Transport Layer

Role of Transport Layer

- **Application layer**
  - Communication for specific applications
  - E.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP)
- Transport layer
  - Communication between processes (e.g., socket)
  - Relies on network layer; serves the application layer
  - E.g., TCP and UDP
- Network layer
  - Logical communication between nodes
  - Hides details of the link technology
  - E.g., IP

Role of Transport Layer

- Application layer
  - Communication for specific applications
  - E.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP)
- Transport layer
  - Communication between processes (e.g., socket)
  - Relies on network layer; serves the application layer
  - E.g., TCP and UDP
- **Network layer**
  - Global communication between nodes
  - Hides details of the link technology
  - E.g., IP

Role of Transport Layer

- Application layer
  - Communication for specific applications
  - E.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP)
- **Transport layer**
  - Communication between **processes** (e.g., socket)
  - Relies on network layer; serves the application layer
  - E.g., TCP and UDP
- Network layer
  - Logical communication between nodes
  - Hides details of the link technology
  - E.g., IP

Role of Transport Layer

- Provide common end-to-end services for app layer
  - **Deal with network on behalf of applications**
  - **Deal with applications on behalf of networks**

- Could have been built into apps, but want common implementations to make app development easier
  - **Since TCP runs on end host, this is about software modularity, not overall network architecture**

What Problems Should Be Solved Here?

- Applications think in terms of files or bytestreams
  - Network deals with packets
  - Transport layer needs to translate between them
- Where does host put incoming data?
  - IP just points towards next protocol
  - How do you get data to the right application?
  - Transport needs to demultiplex incoming data (ports)
- Reliability (for those apps that want it)
- Corruption **(Why?)**
- Overloading the receiving host? The network?

What Is Needed to Address These?

- Translating between bytestreams and packets
  - Do segmentation and reassembly
- Demultiplexing: identifier for application process
- Reliability: ACKs and all that stuff
  - Pieces we haven’t covered: RTT estimation, formats
- Corruption: checksum
- Not overloading receiver: limit data in recvr’s buffer
- Not overloading network: later in semester

Conclusion?

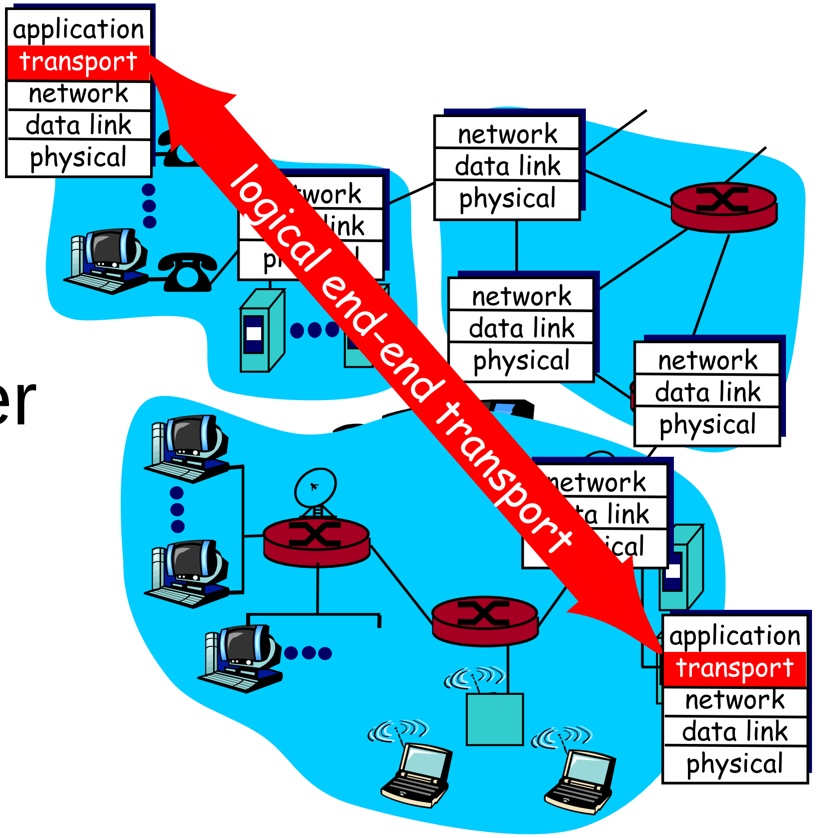
- Transport is easy!
  - except congestion control, which we cover later…
- Rest of lecture just diving into details
  - Nothing is fundamental
  - These are just current implementation choices

Logical View of Transport Protocols

- Provide *logical communication* between application processes running on different hosts

- **Sender**: breaks application messages into segments, and passes to network layer

- **Receiver**: reassembles segments into messages, passes to application layer



UDP: Datagram messaging service

- No-frills extension of “best-effort” IP
- Multiplexing/Demultiplexing among processes
- Discarding corrupted packets (optional)

TCP: Reliable, in-order delivery

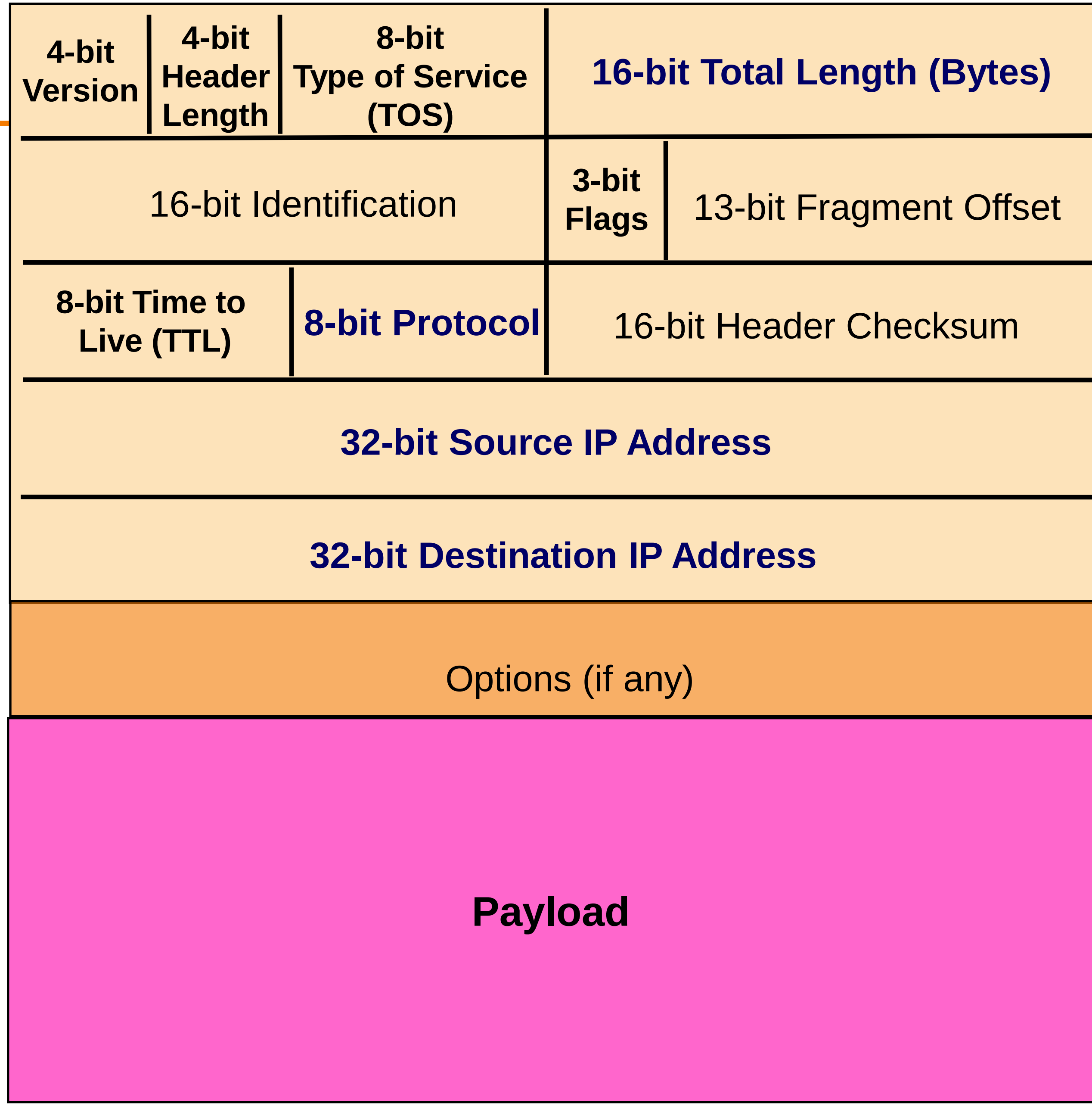
- ***What UDP provides, plus:***
- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down

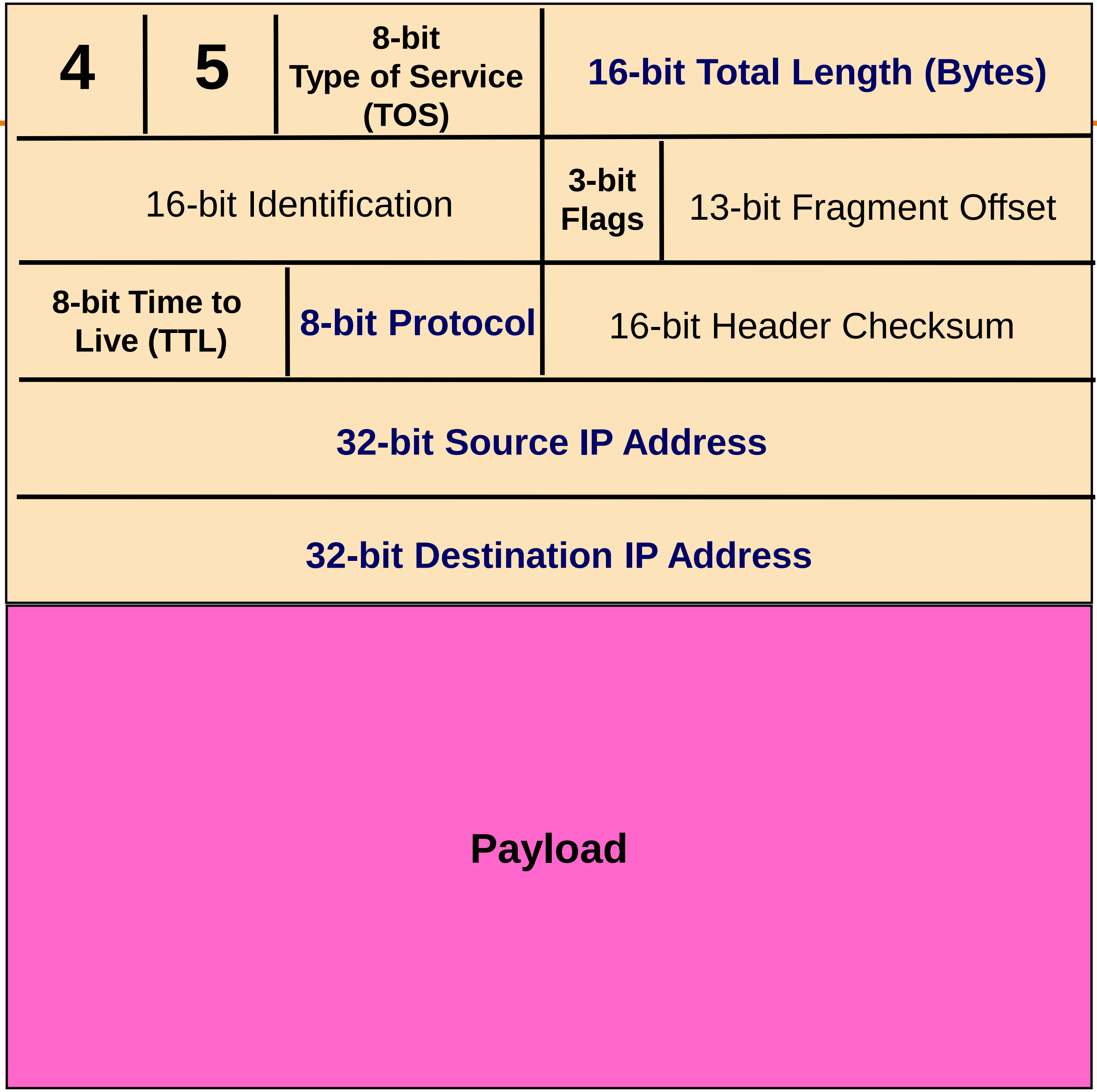
Connections (or sessions)

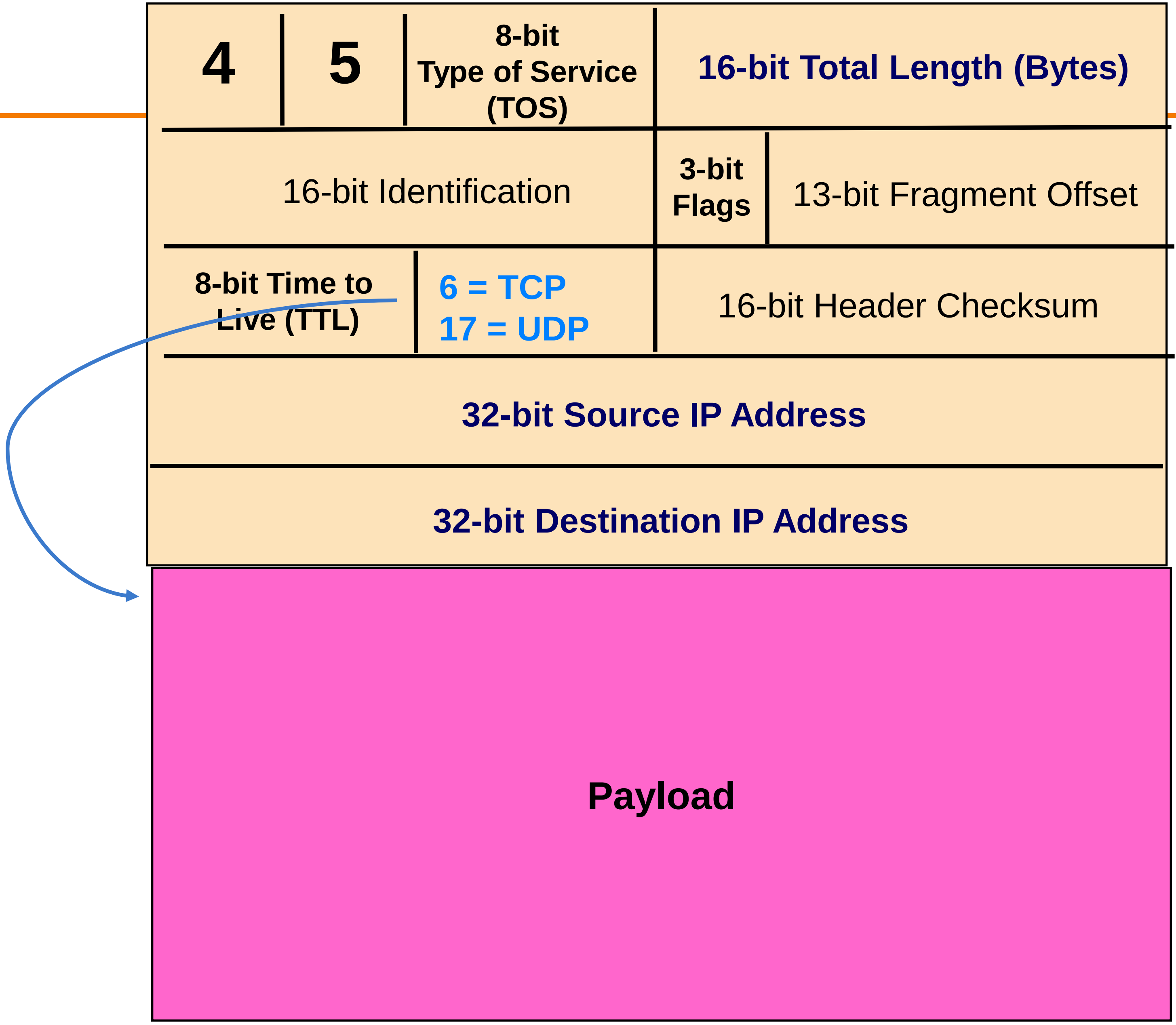
- Reliability requires keeping state
  - Sender: packets sent but not ACKed, and related timers
  - Receiver: noncontiguous packets
- Each bytestream is called a connection or session
  - Each with their own connection state
  - State is in hosts, not network!
- Example: I am using HTTP to access content on two different hosts, and I'm also ssh'ing into another host. How many sessions is this?

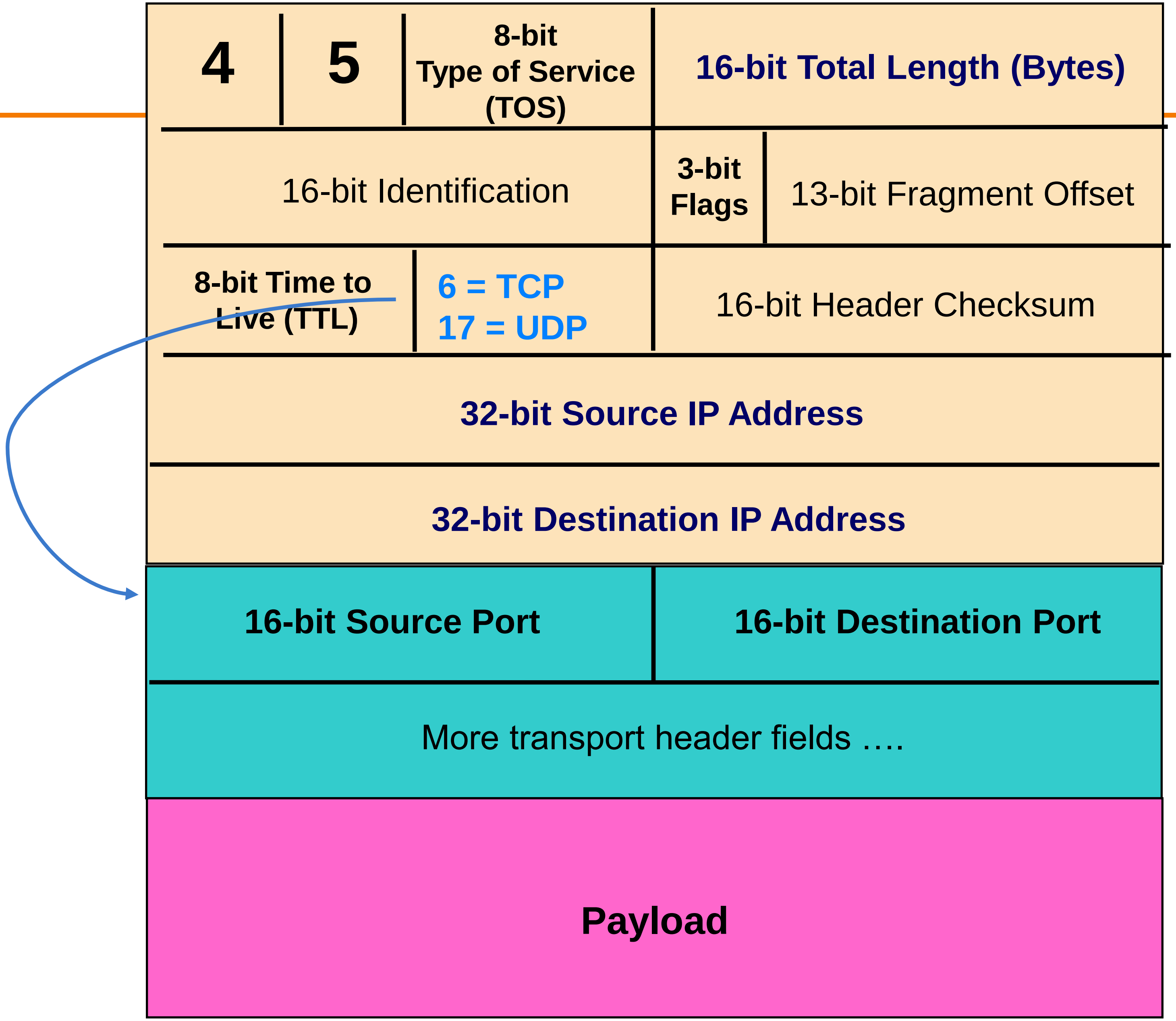
Services not available

- Delay and/or bandwidth guarantees
  - This is fundamental to the transport layer
- Sessions that survive change-of-IP-address
  - This is an artifact of current implementations


4-bit Version
4-bit Header Length
8-bit Type of Service (TOS)
16-bit Total Length (Bytes)
16-bit Identification
3-bit Flags
13-bit Fragment Offset
8-bit Time to Live (TTL)
8-bit Protocol
16-bit Header Checksum
32-bit Source IP Address
32-bit Destination IP Address
Options (if any)
Payload

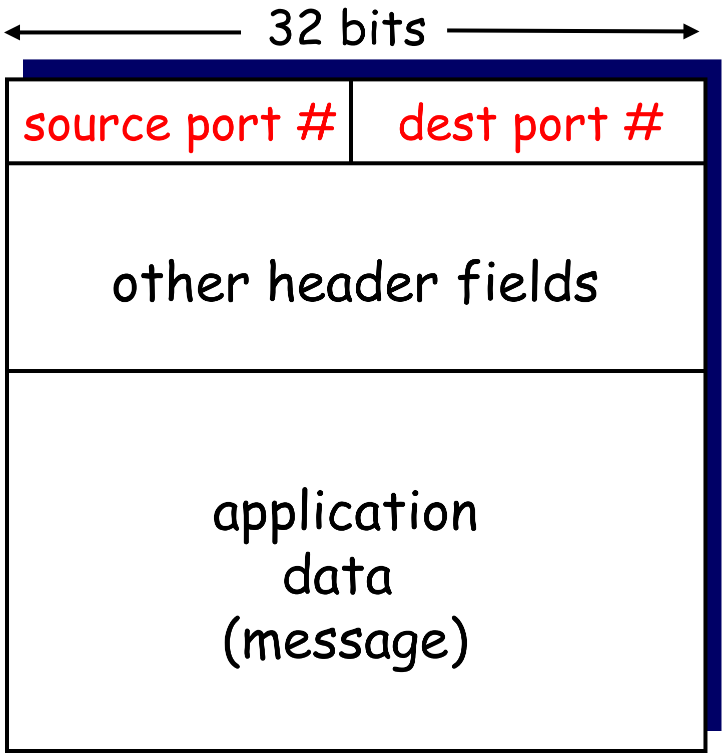

4
5
8-bit Type of Service (TOS)
16-bit Total Length (Bytes)
16-bit Identification
3-bit Flags
13-bit Fragment Offset
8-bit Time to Live (TTL)
8-bit Protocol
16-bit Header Checksum
32-bit Source IP Address
32-bit Destination IP Address
Payload


4
5
8-bit Type of Service (TOS)
16-bit Total Length (Bytes)
16-bit Identification
3-bit Flags
13-bit Fragment Offset
8-bit Time to Live (TTL)
6 = TCP
17 = UDP
16-bit Header Checksum
32-bit Source IP Address
32-bit Destination IP Address
Payload


4
5
8-bit Type of Service (TOS)
16-bit Total Length (Bytes)
16-bit Identification
3-bit Flags
13-bit Fragment Offset
8-bit Time to Live (TTL)
6 = TCP
17 = UDP
16-bit Header Checksum
32-bit Source IP Address
32-bit Destination IP Address
16-bit Source Port
16-bit Destination Port
More transport header fields ….
Payload

Multiplexing and Demultiplexing

- Host receives IP datagrams
  - Each datagram has source and destination IP address,
  - Each segment has source and destination port number
- Host uses IP addresses and port numbers to direct the segment to appropriate socket



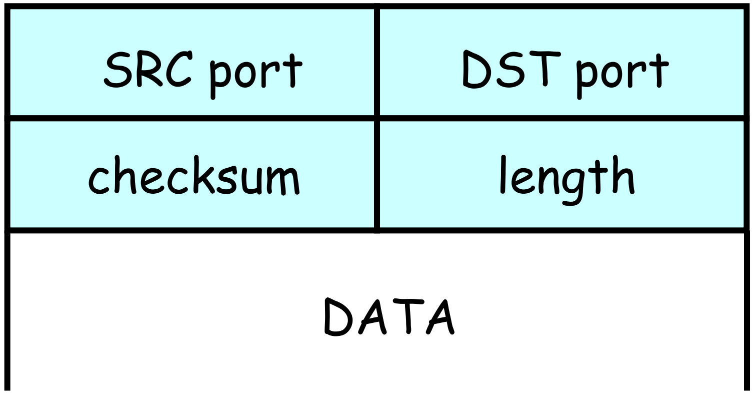
TCP/UDP segment format

Directing packets to process

- UDP: uses destination port (and address)
- TCP: uses source/destination ports and addresses
  - (src_IP, src_port, dst_IP, dst_port)
- Why the difference?
- **Implications for mobility?**

UDP: User Datagram Protocol

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them from a socket
- UDP described in RFC 768 – (1980!)
  - IP plus port numbers to support (de)multiplexing
  - Optional error checking on the packet contents
    - (checksum field = 0 means "don't verify checksum")



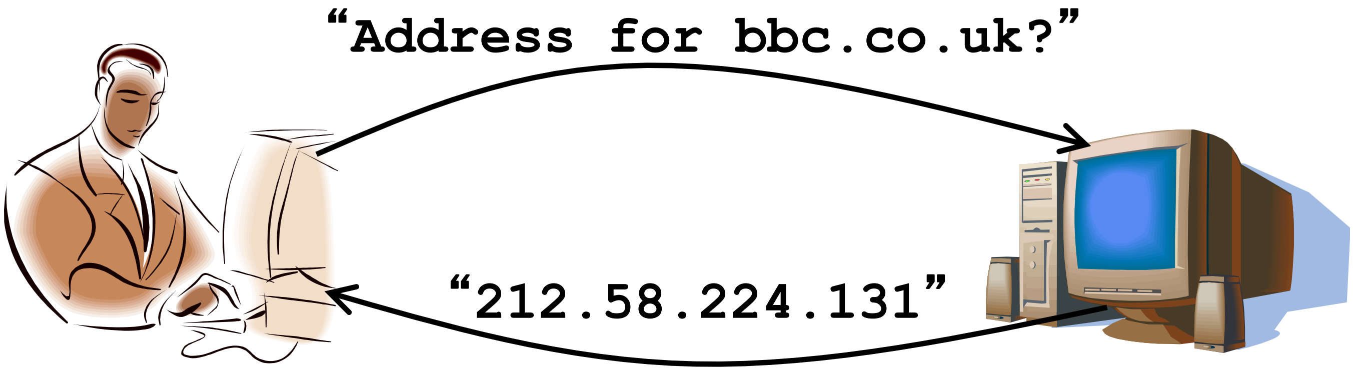
Why Would Anyone Use UDP?

- Finer control over what data is sent and when
  - As soon as an application process writes into the socket
  - … UDP will package the data and send the packet
- No delay for connection establishment
  - UDP just blasts away without any formal preliminaries
  - … which avoids introducing any unnecessary delays
- No connection state
  - No allocation of buffers, sequence #s, timers …
  - … making it easier to handle many active clients at once
- Small packet header overhead
  - UDP header is only 8 bytes

Popular Applications That Use UDP

- Some interactive streaming apps
  - Retransmitting lost/corrupted packets often pointless - by the time the packet is retransmitted, it's too late
  - E.g., telephone calls, video conferencing, gaming

  - ***Modern streaming protocols using TCP (and HTTP)***
- Simple query protocols like Domain Name System
  - Connection establishment overhead would double cost
  - Easier to have application retransmit if needed



Transmission Control Protocol (TCP)

- Connection oriented *(today)*
  - Explicit set-up and tear-down of TCP session
- Full duplex stream-of-bytes service *(today)*
  - Sends and receives a stream of bytes, not messages
- Congestion control *(later)*
  - Dynamic adaptation to network path’s capacity
- Reliable, in-order delivery *(previously, but quick review)*
  - Ensures byte stream (eventually) arrives intact
    - In the presence of corruption and loss
- Flow control *(previously, but quick review)*
  - Ensures that sender doesn’t overwhelm receiver

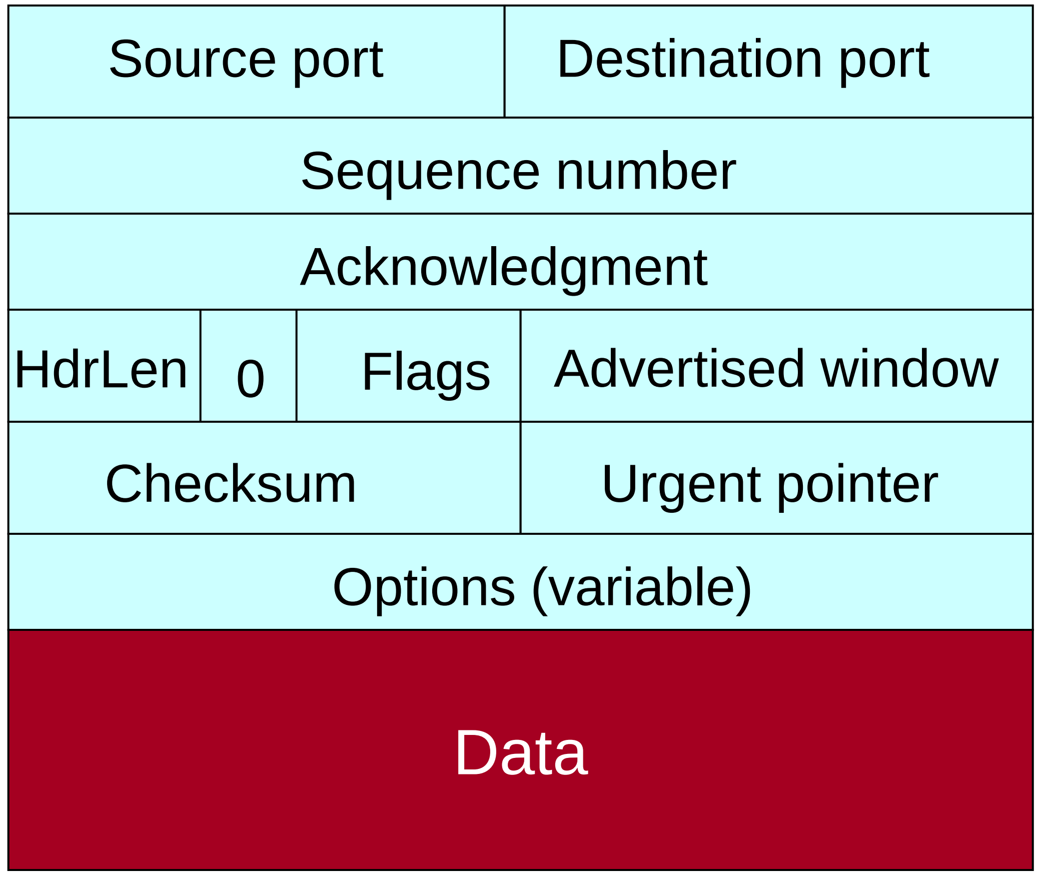
TCP

We've been studying the general properties of reliable transport. We now learn about how they are implemented today.

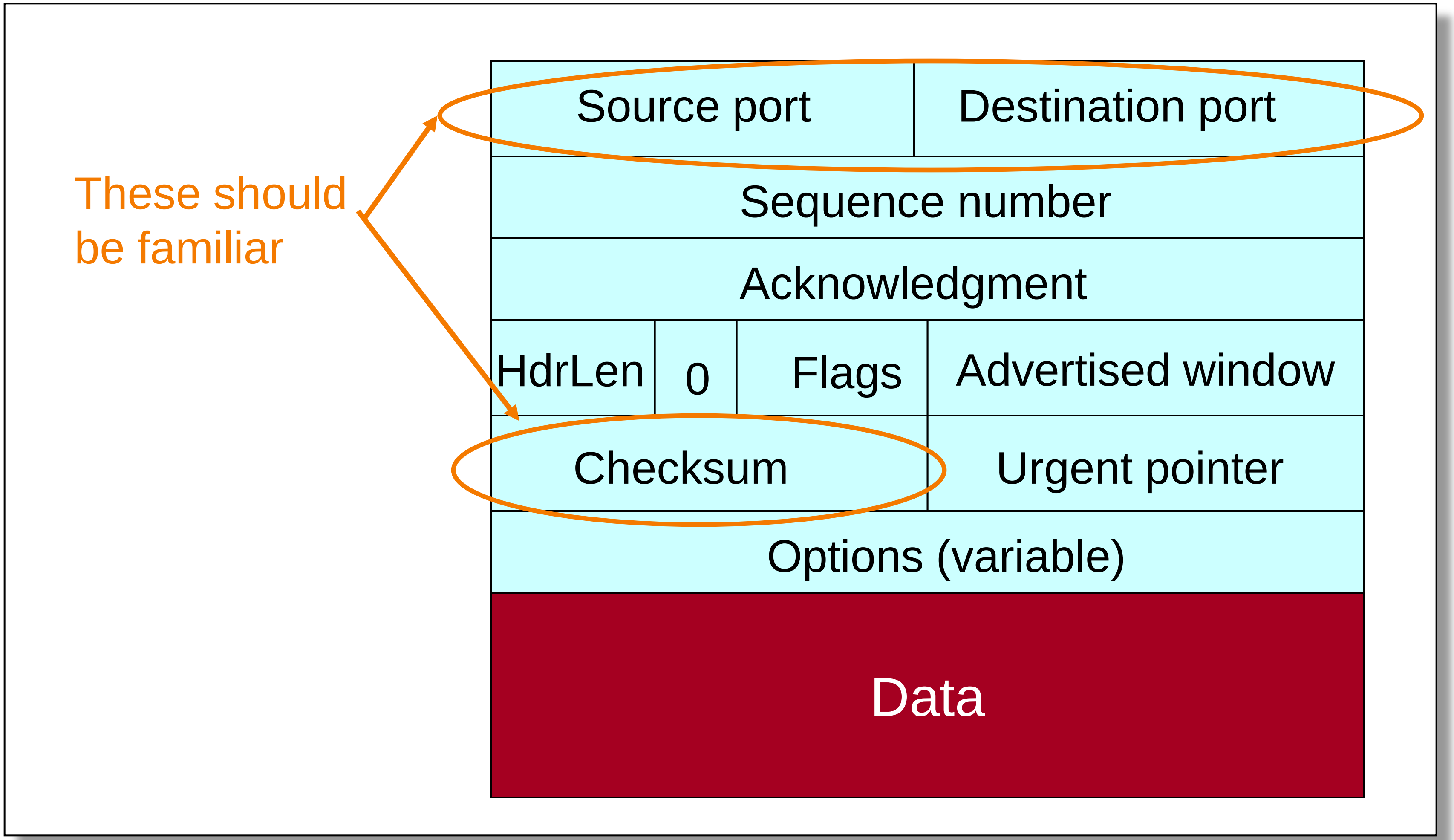
TCP Support for Reliable Delivery

- Checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- Sequence numbers
  - Used to detect missing data
  - ... and for putting the data back in order
- Retransmission
  - Sender retransmits lost or corrupted data
  - Timeout based on estimates of round-trip time
  - *Fast retransmit* algorithm for rapid retransmission

TCP Header

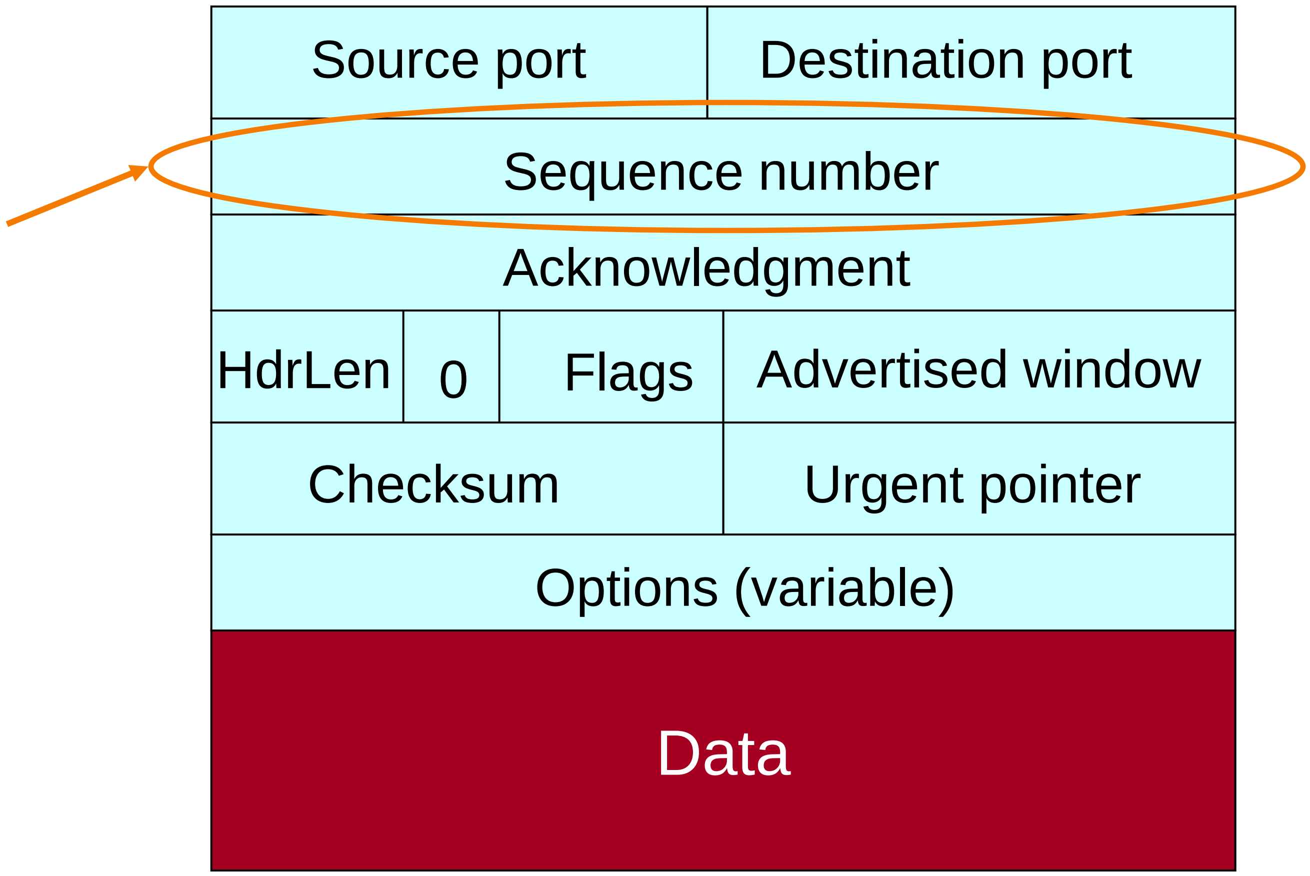

Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TCP Header


These should be familiar
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TCP Header

Starting sequence number (byte offset) of data carried in this segment

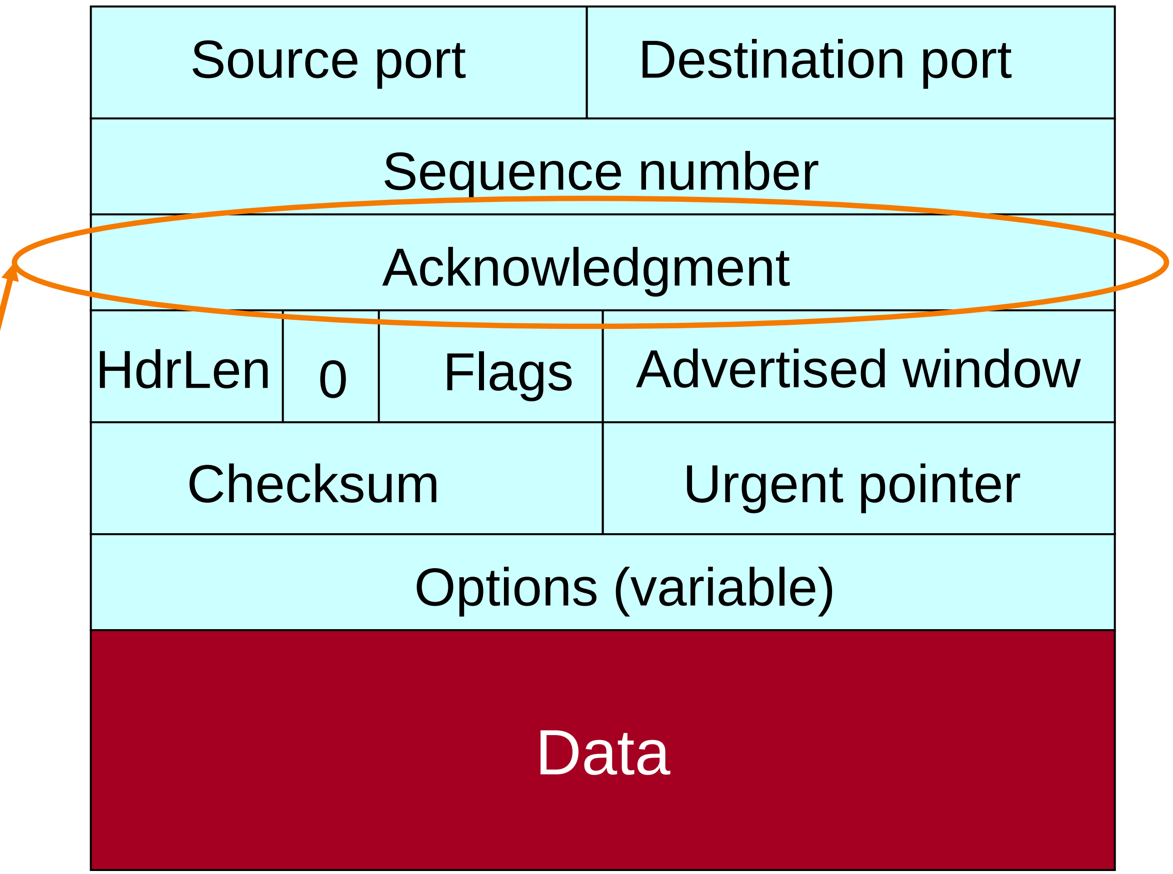


TCP Header

Acknowledgment gives seq # **just beyond** highest seq. received **in order**.

"What's Next"

If sender sends **N** in-order bytes starting at seq **S** then ack for it will be **S+N**.



ACKing and Sequence Numbers

- Sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes
    - X, X+1, X+2, ….X+B-1

- Upon receipt of packet, receiver sends an ACK
  - If all data prior to X already received:
    - ACK acknowledges X+B (because that is next expected byte)
  - If highest byte already received is some smaller value Y
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

- Sender sends(?) next packet with seqno X+B

Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

- Seqno of next packet is same as last ACK field

5 Minute Break

Anagram Contest

- What does this numerical anagram have to do with this alphabetical one?
  - Alphabetical: A Tragic Con
  - Numerical: 01235688

Anagram Contest

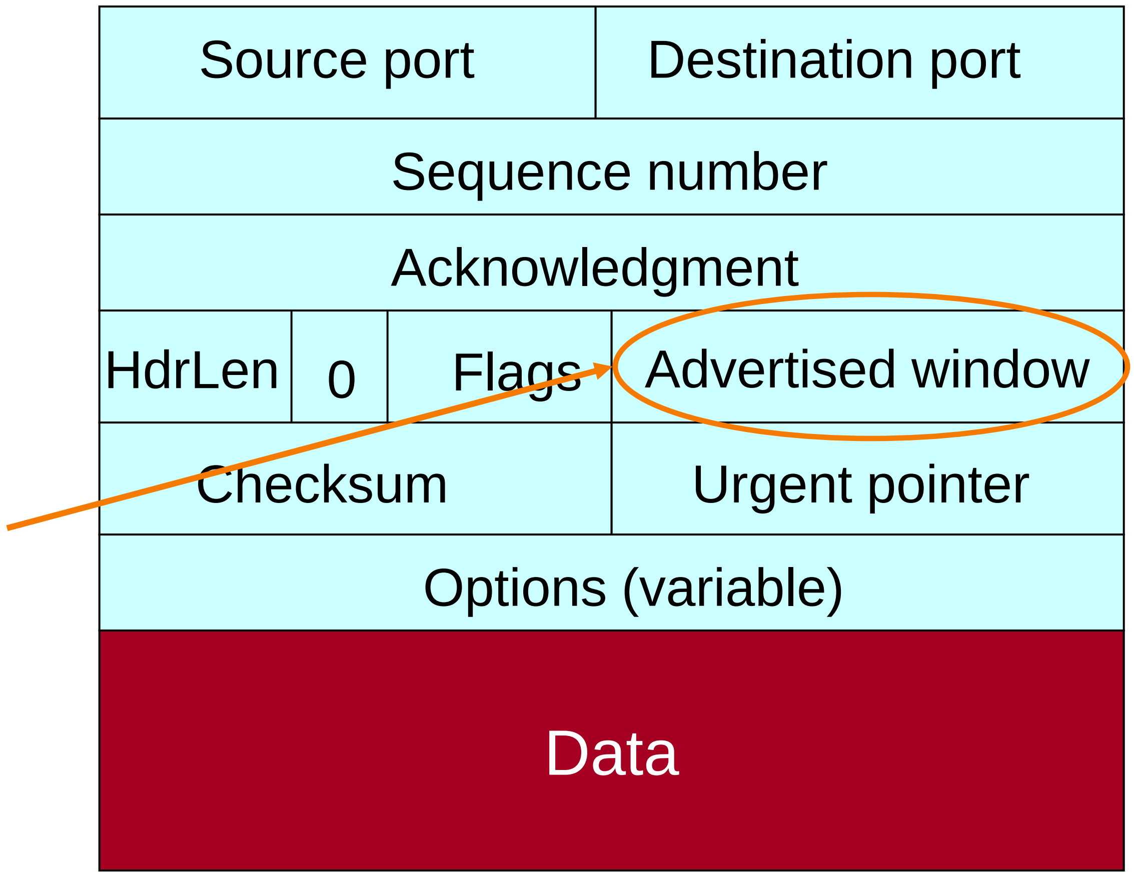
- What does this numerical anagram have to do with this alphabetical one?
  - Alphabetical: A Tragic Con
  - Numerical: 01235688

- **UC Berkeley was founded by the Organic Act which was passed on 05/23/1868**

TCP Header

Buffer space available for receiving data. Used for TCP’s sliding window.

Interpreted as offset beyond Acknowledgment field’s value.



Sliding Window Flow Control

- Advertised Window: W
  - Can send W bytes beyond the next expected byte
- Receiver uses W to prevent sender from overflowing buffer
  - Limits number of bytes sender can have in flight

Filling the Pipe

- Simple example:
  - W (in bytes), which we assume is constant
  - RTT (in sec), which we assume is constant
  - B (in **bytes**/sec)

- How fast will data be transferred?

- **If W/RTT < B, the transfer has speed W/RTT**
- If W/RTT > B, the transfer has speed B

Performance with Sliding Window

- Consider UCB□NYC 1 Mbps path (100msec RTT)
  - Q1: How fast can we transmit with W=12.5KB? (~8pkts)
  - A: 12.5KB/100msec ~ 1Mbps (we can fill the pipe)
- Q2: What if path is 1Gbps?
  - A2: Can still only send 1Mbps
- Window required to fully utilize path:
  - Bandwidth-delay product
  - 1 Gbps * 100 msec = 100 Mb = 12.5 MB
  - 12.5 MB ~ 8333 packets of 1500bytes **(lots of packets!)**

Advertised Window Limits Rate

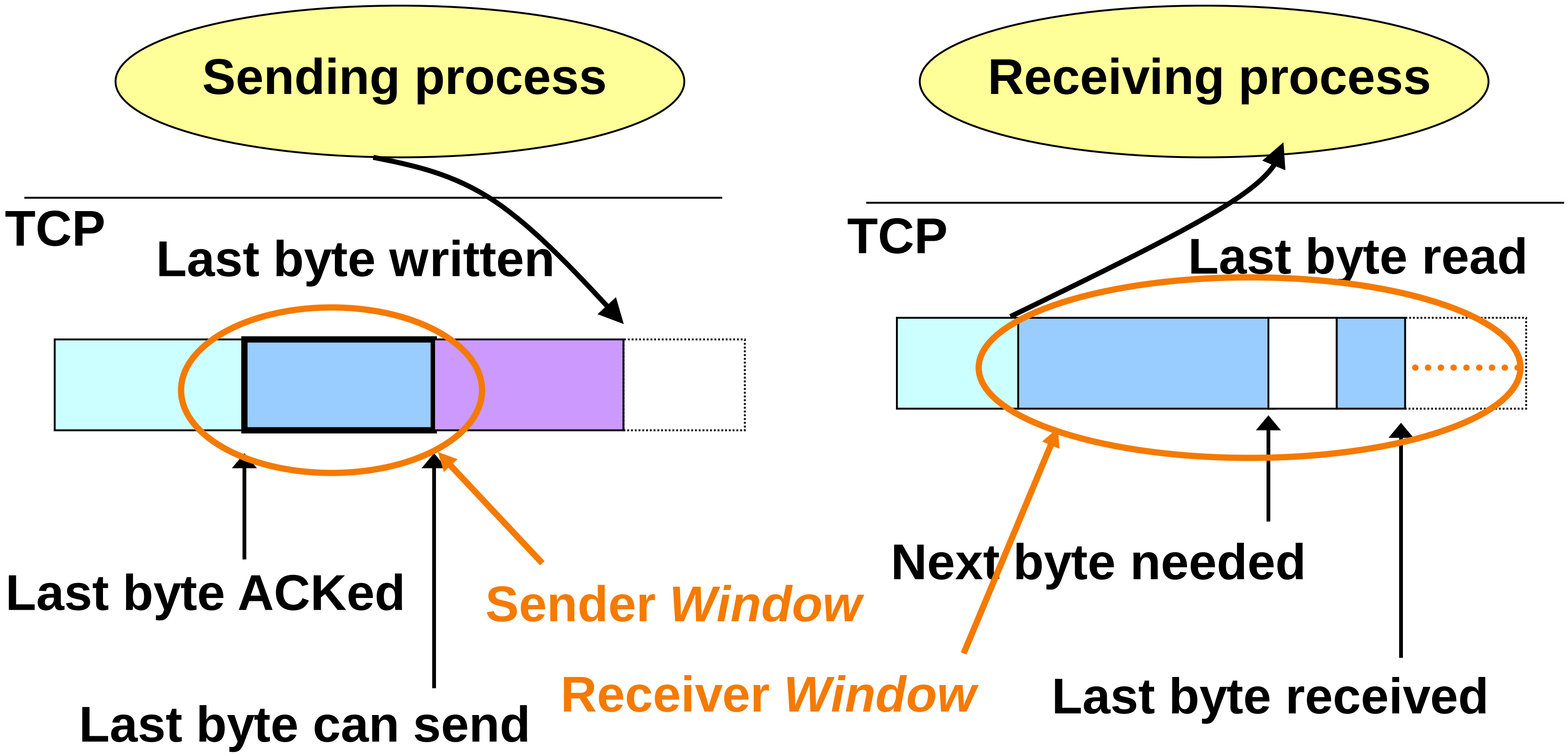
- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling sender's rate
- What's missing?

Implementing Sliding Window

- Both sender & receiver maintain a **window**
  - Sender: not yet ACK'ed
  - Receiver: not yet delivered to application
- Left edge of window:
  - Sender: beginning of unacknowledged data
  - Receiver: beginning of undelivered data
- For the sender:
  - Window size = maximum amount of data in flight
- For the receiver:
  - Window size = maximum amount of undelivered data

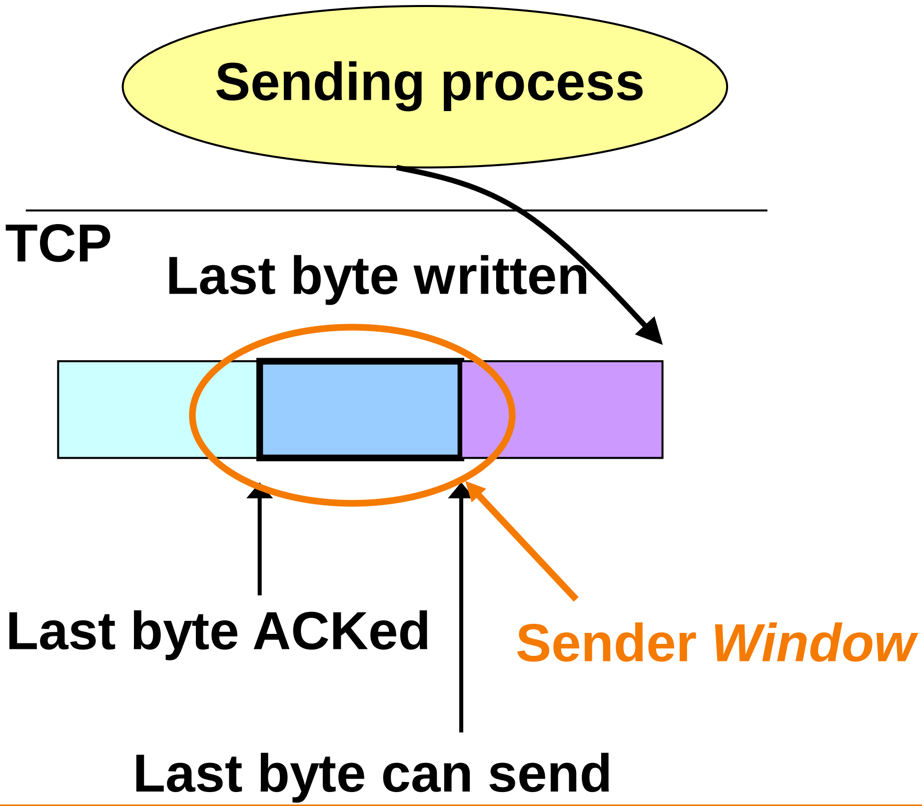
Sliding Window

- Allow a larger amount of data "in flight"
  - Allow sender to get ahead of the receiver
  - … though not *too far* ahead



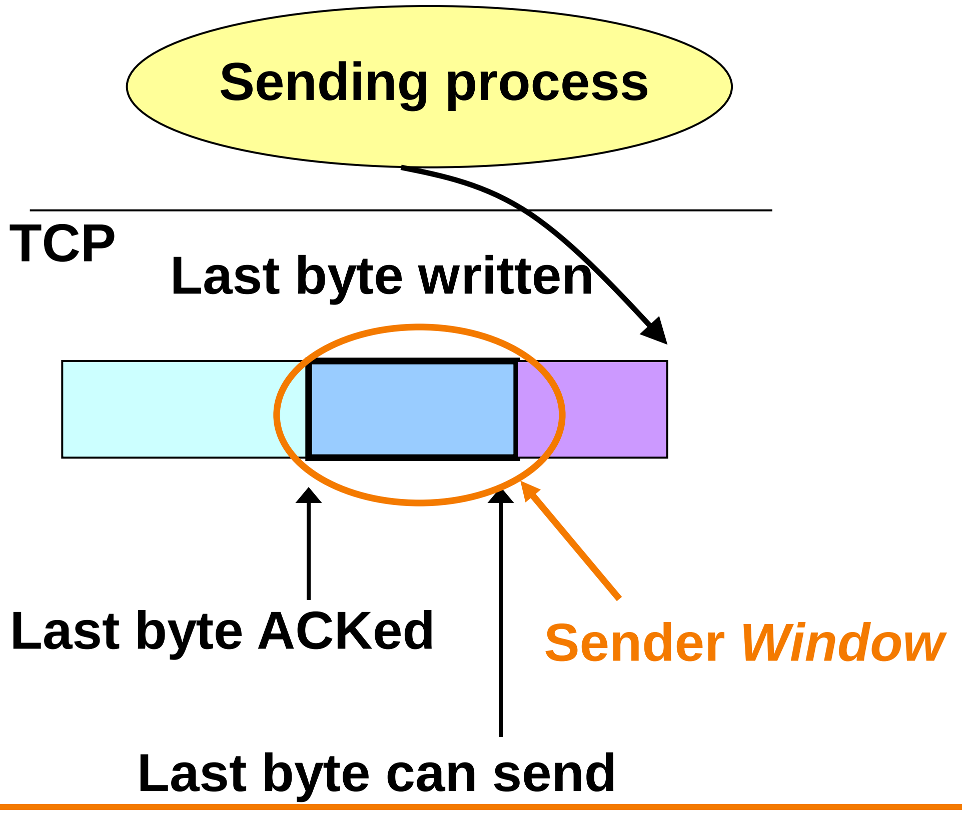
Sliding Window

- For the sender, when receives an acknowledgment for new data, window advances (*slides* forward)



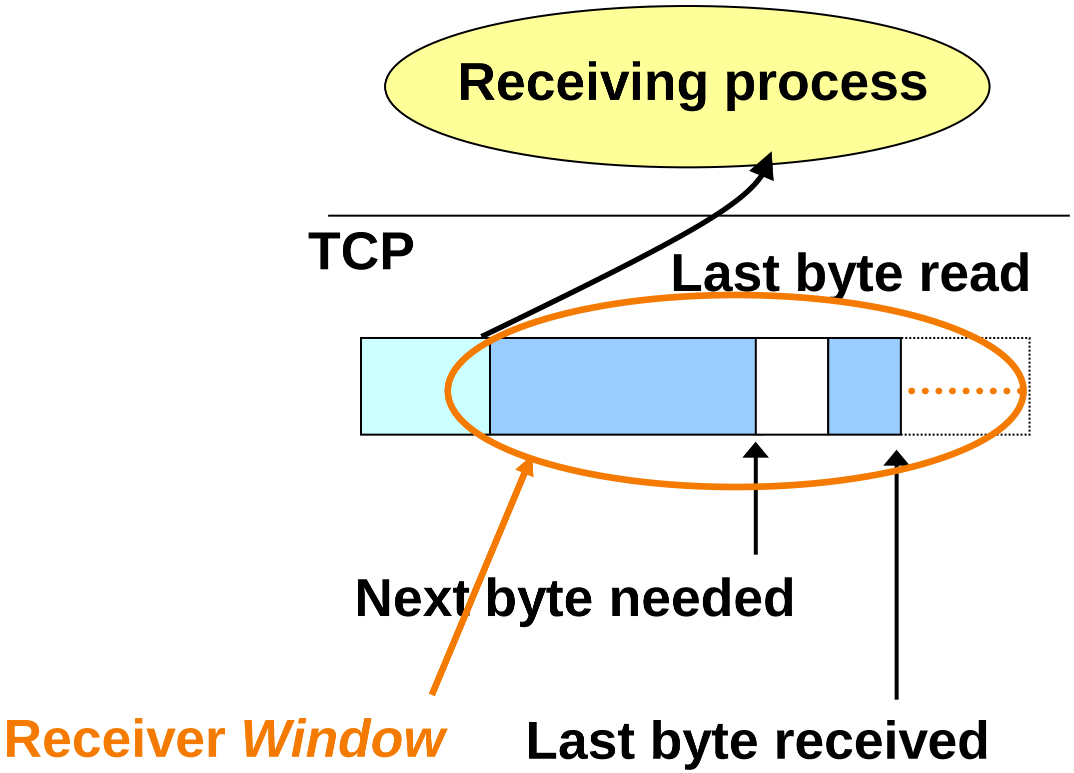
Sliding Window

- For the sender, when receives an acknowledgment for new data, window advances (*slides* forward)



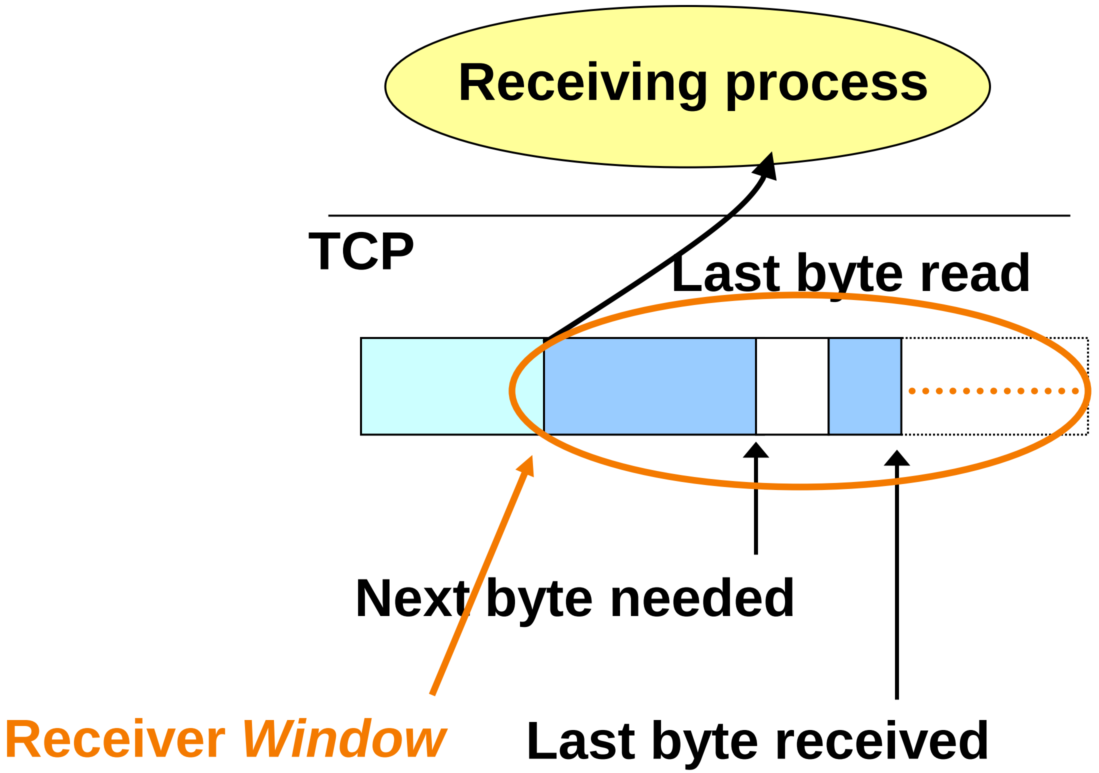
Sliding Window

- For the receiver, as the receiving process consumes data, the window slides forward



Sliding Window

- For the receiver, as the receiving process consumes data, the window slides forward

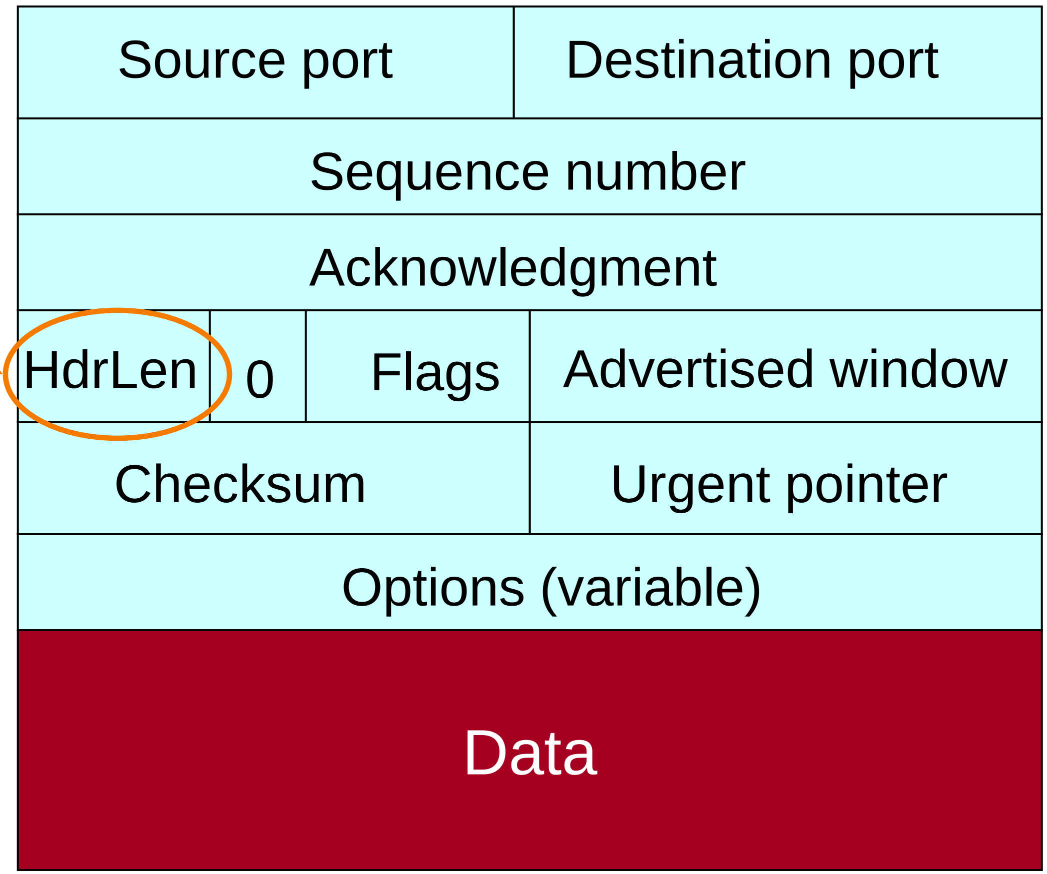


***Sliding Window*, con't**

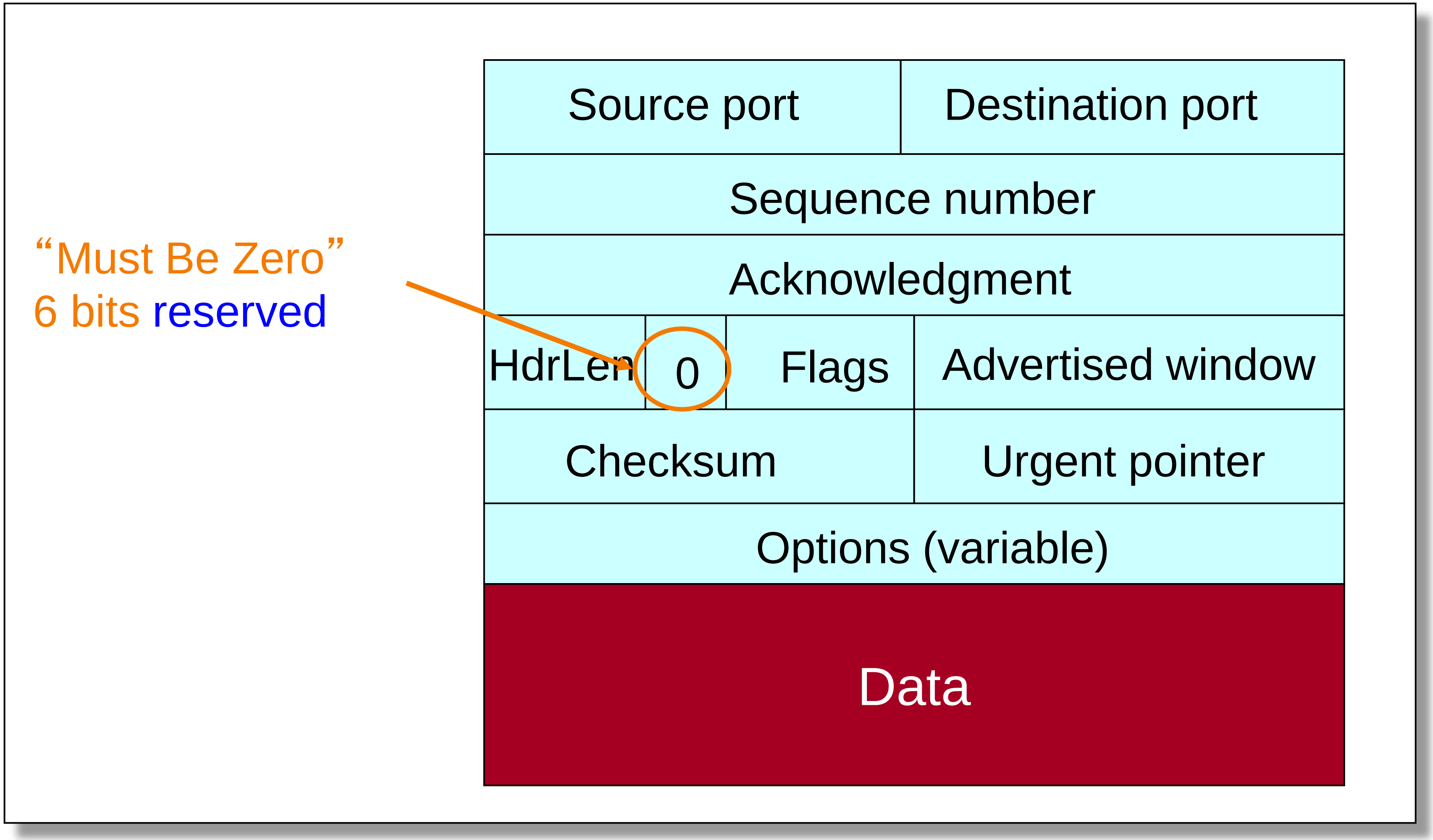
- Sender: window advances when new data ack'd
- Receiver: window advances as receiving process consumes data
- Receiver advertises to the sender where the receiver window currently ends ("righthand edge")
  - Sender agrees not to exceed this amount
  - It makes sure by setting its own window size to a value that can't send beyond the receiver's righthand edge

TCP Header

Number of 4-byte words in TCP header;
5 = no options

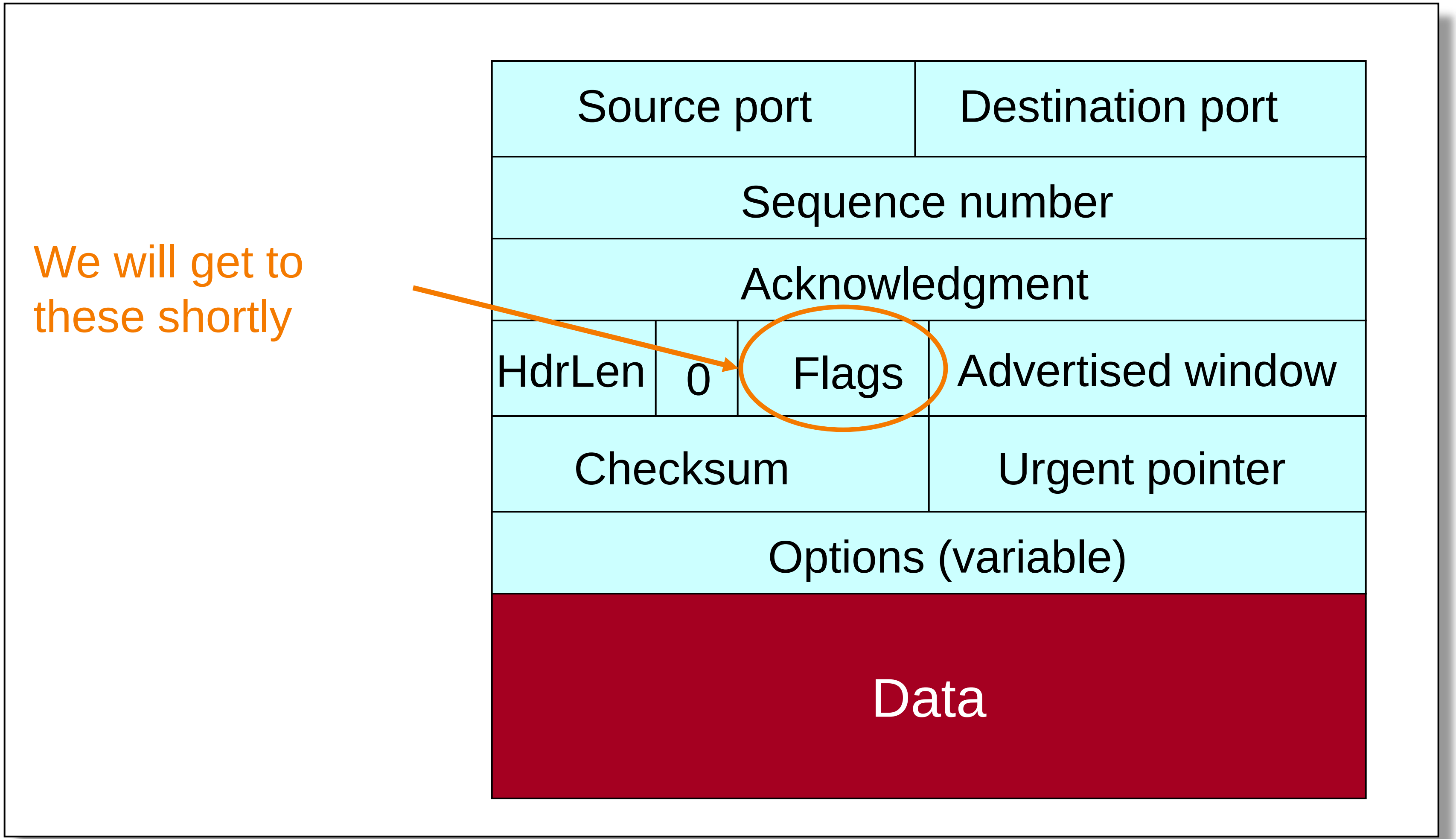


TCP Header



"Must Be Zero"
6 bits reserved
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

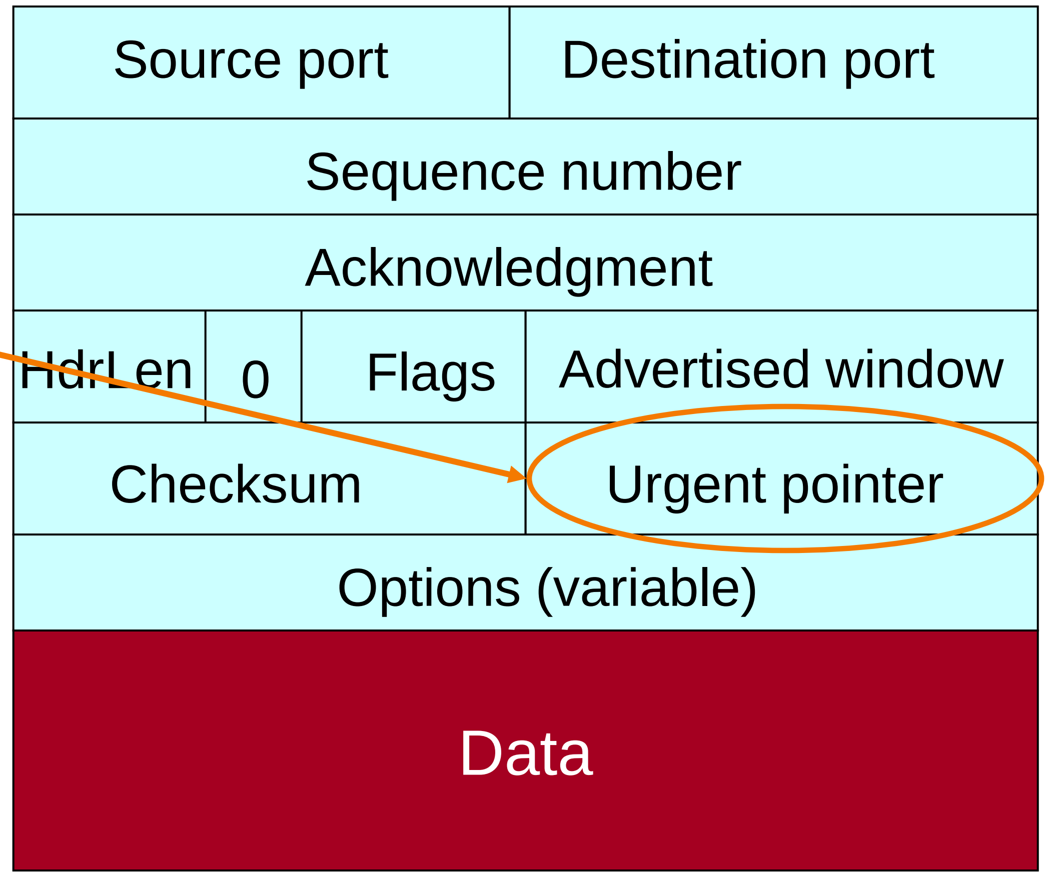
TCP Header



We will get to these shortly
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

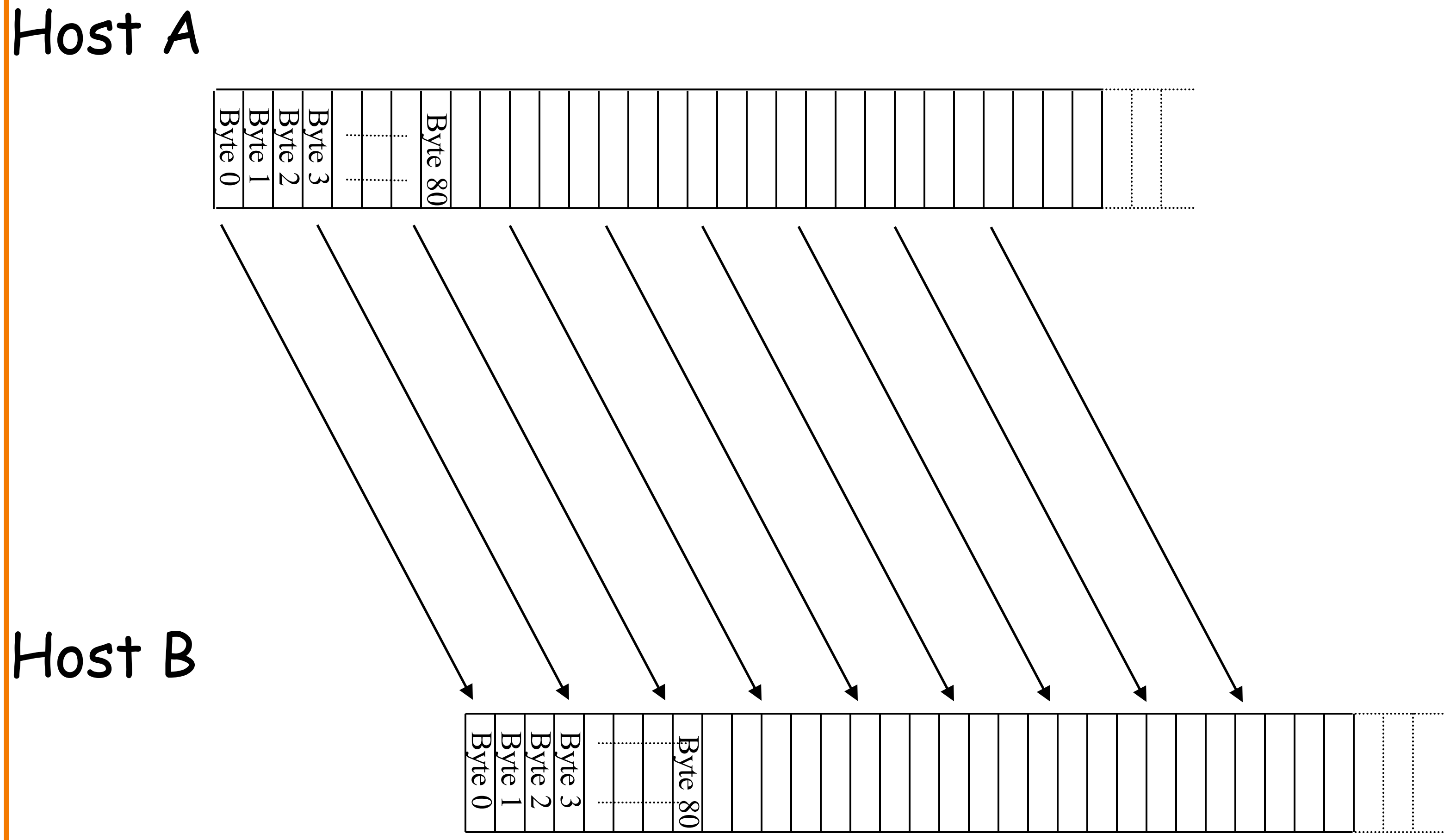
TCP Header

Used with **URG** flag to indicate urgent data (not discussed further)

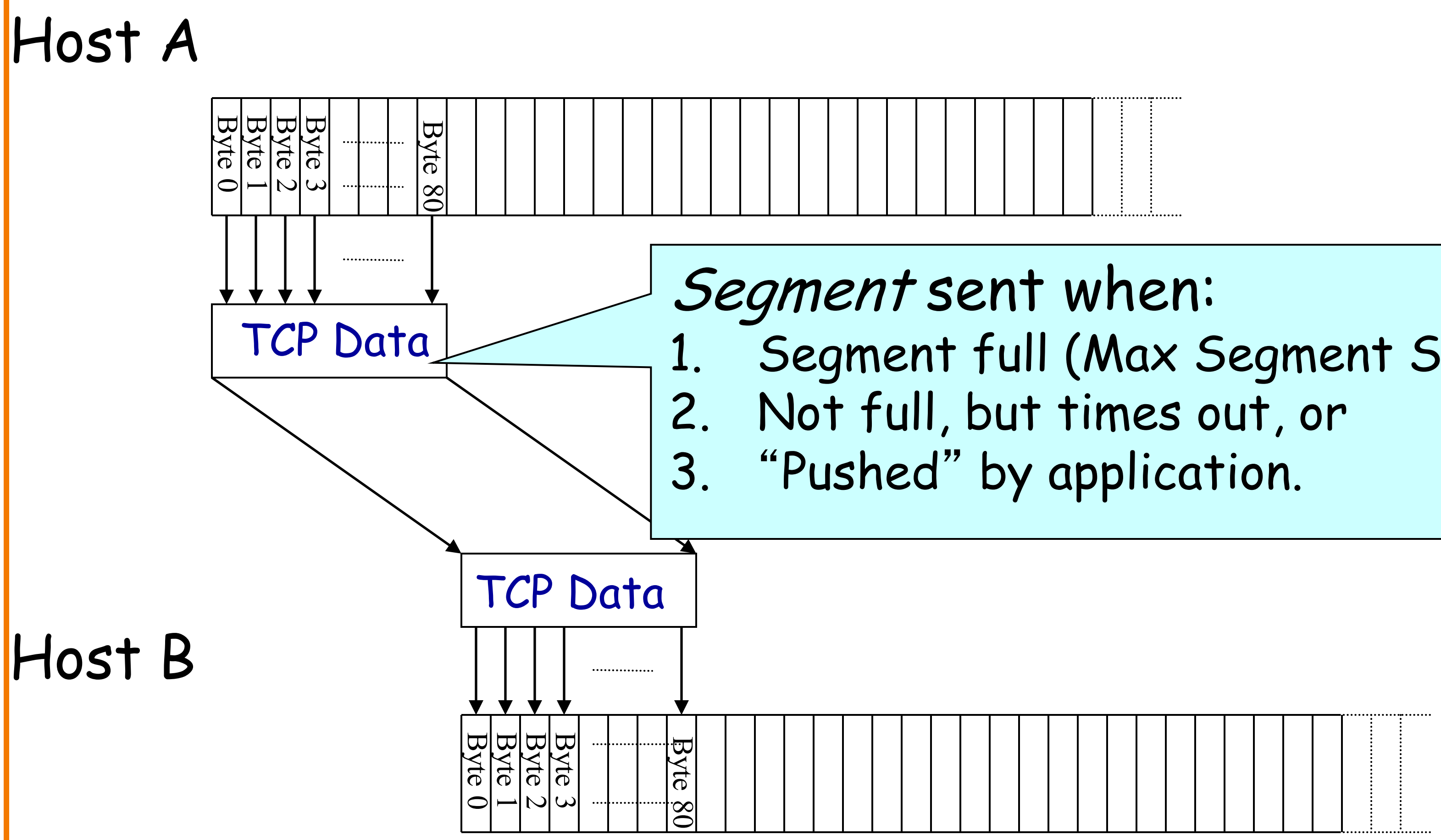


Segments and Sequence Numbers

TCP “Stream of Bytes” Service



… Provided Using TCP "Segments"



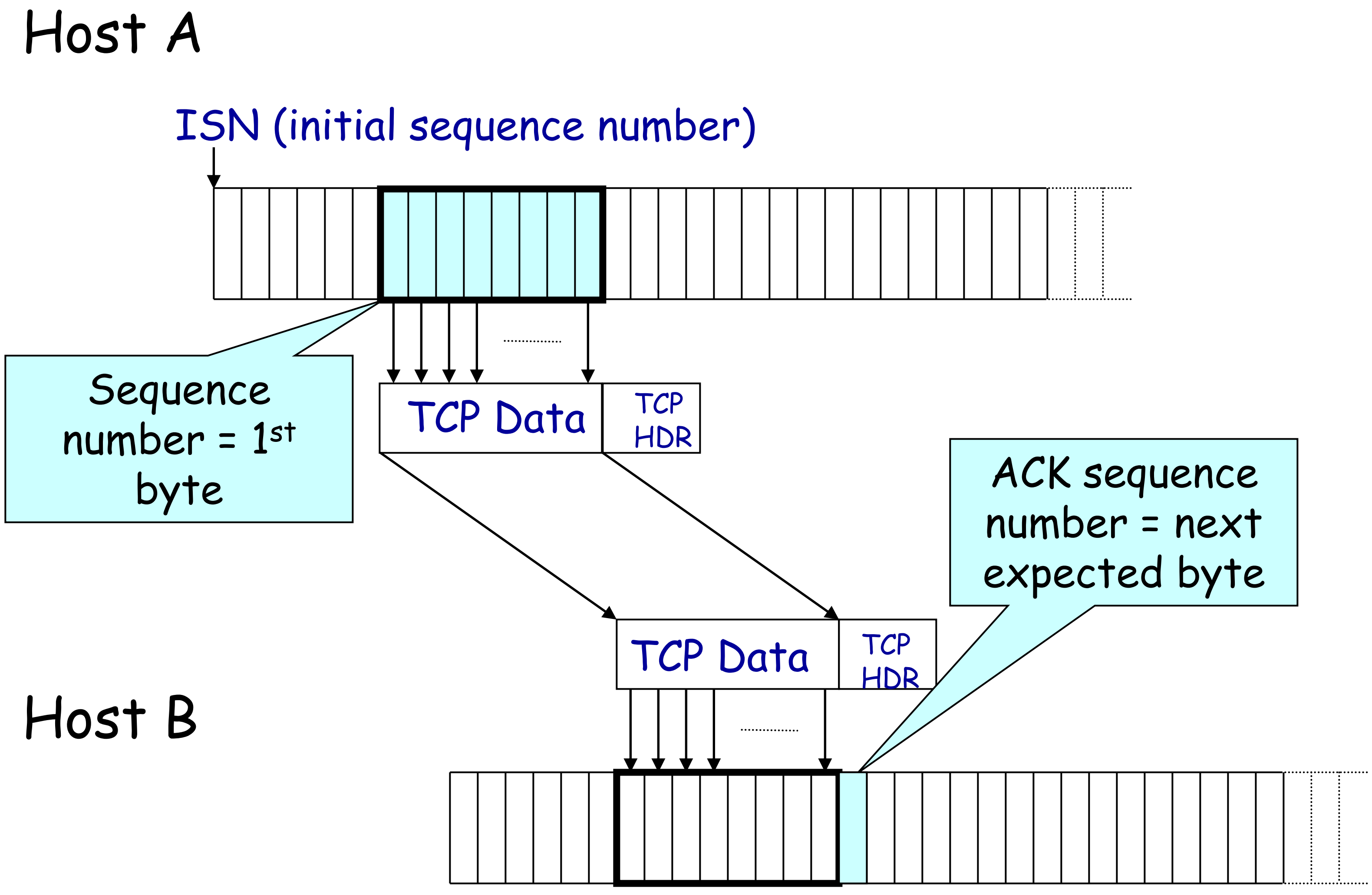
Segment sent when:

1. Segment full (Max Segment Size),
2. Not full, but times out, or
3. "Pushed" by application.

TCP Segment

- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1,500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)

Sequence Numbers

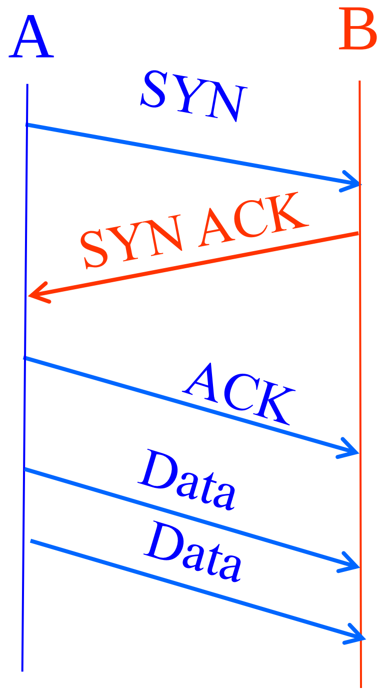


Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … small chance an old packet is still in flight
- TCP therefore requires changing ISN
  - Set from 32-bit clock that ticks every 4 microseconds
  - … only wraps around once every 4.55 hours
- To establish a connection, hosts exchange ISNs
  - **How does this help?**

Connection Establishment:
TCP's *Three-Way Handshake*

Establishing a TCP Connection

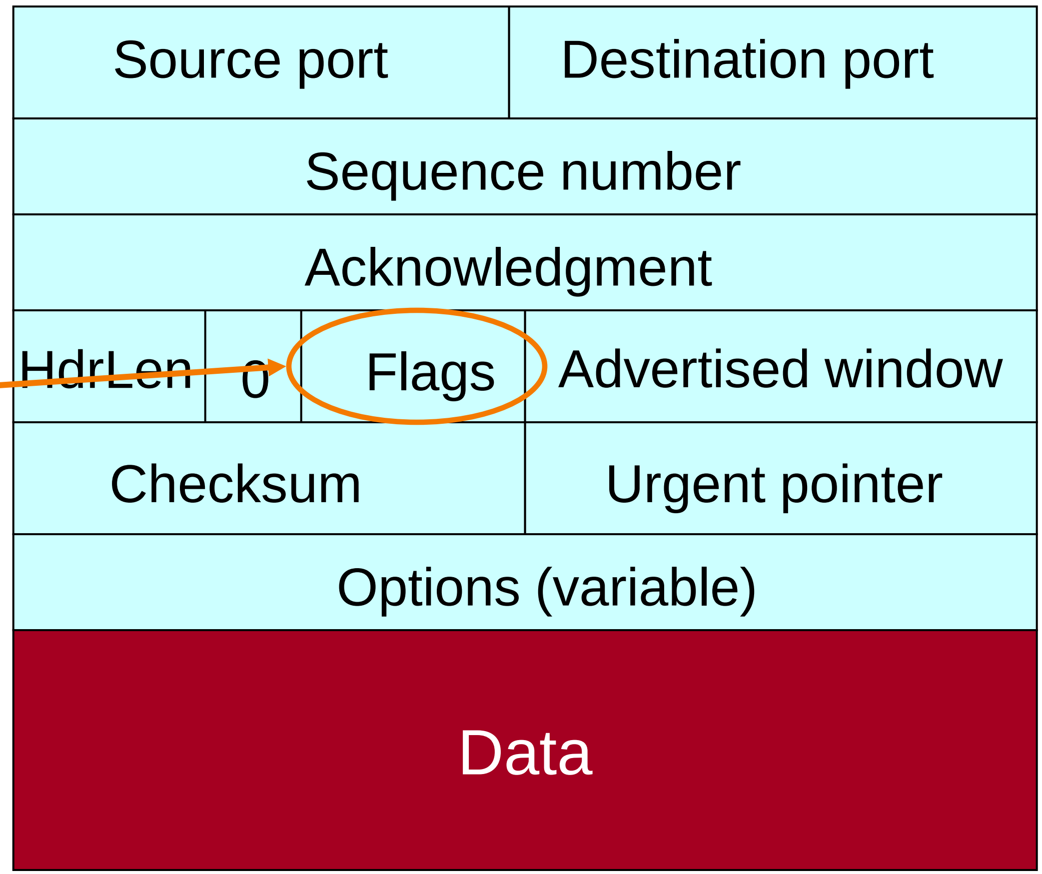


Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; "synchronize sequence numbers") to host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

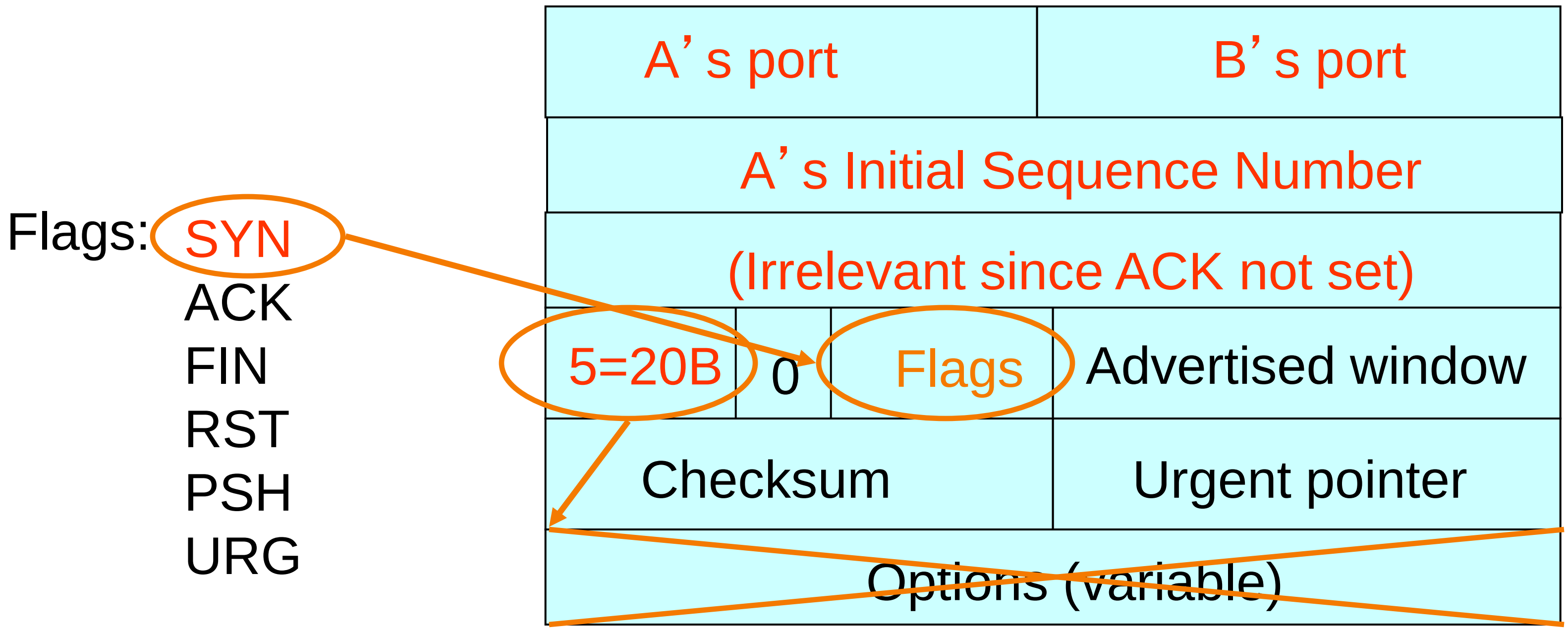
TCP Header

Flags: **SYN**
ACK
FIN
RST
PSH
URG



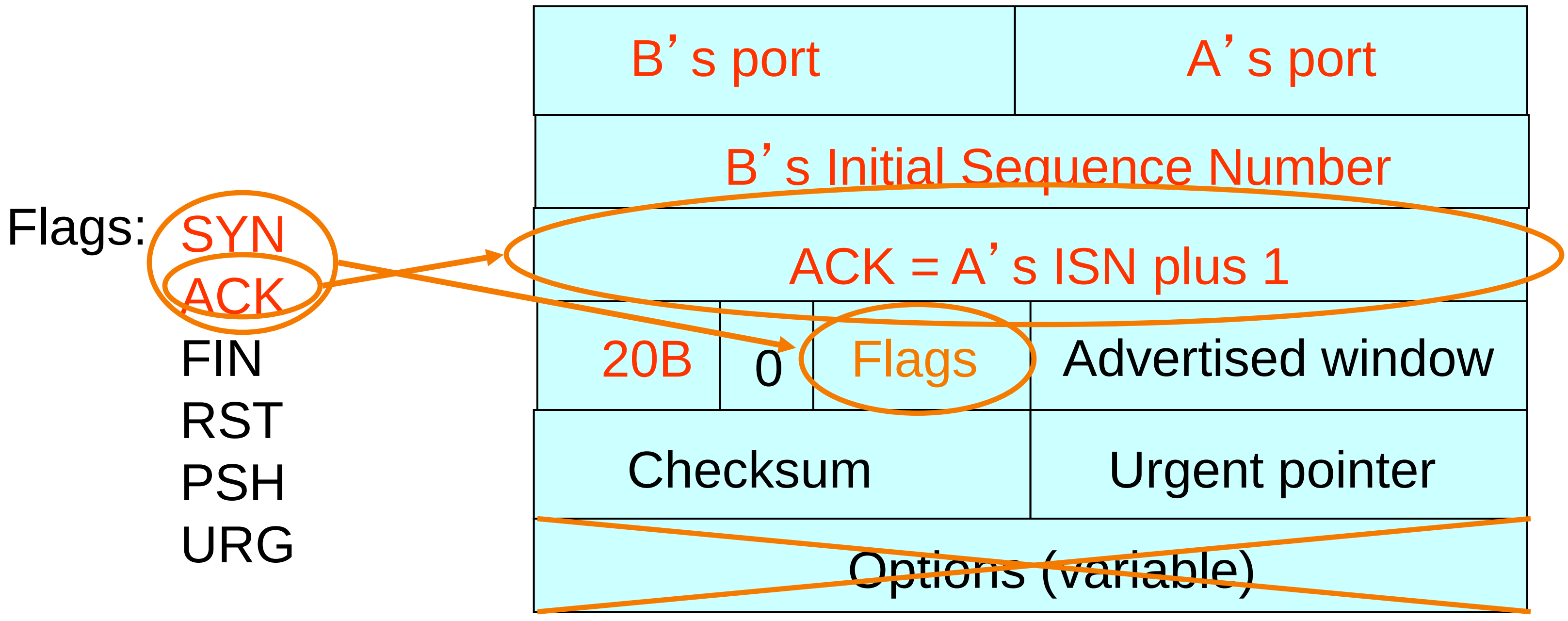
See /usr/include/netinet/tcp.h on Unix Systems

Step 1: A’s Initial SYN Packet



A tells B it wants to open a connection…

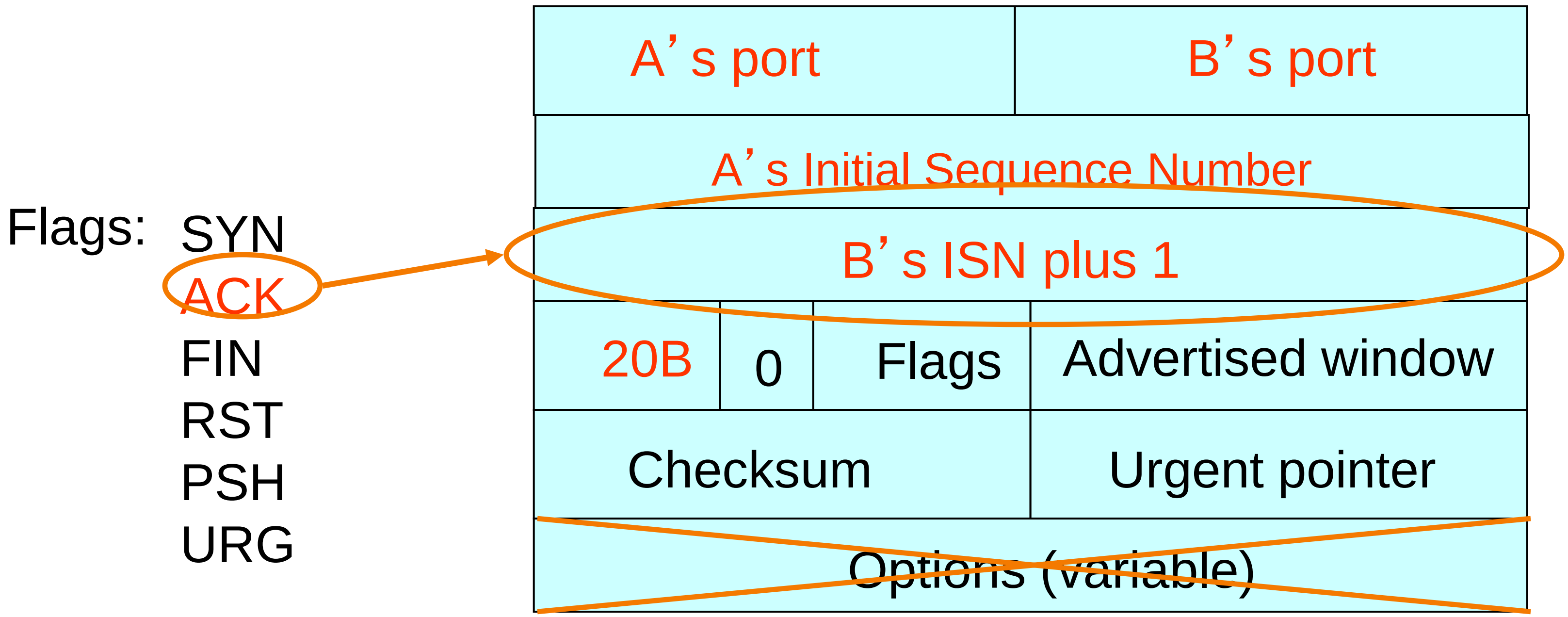
Step 2: B’s SYN-ACK Packet



B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data

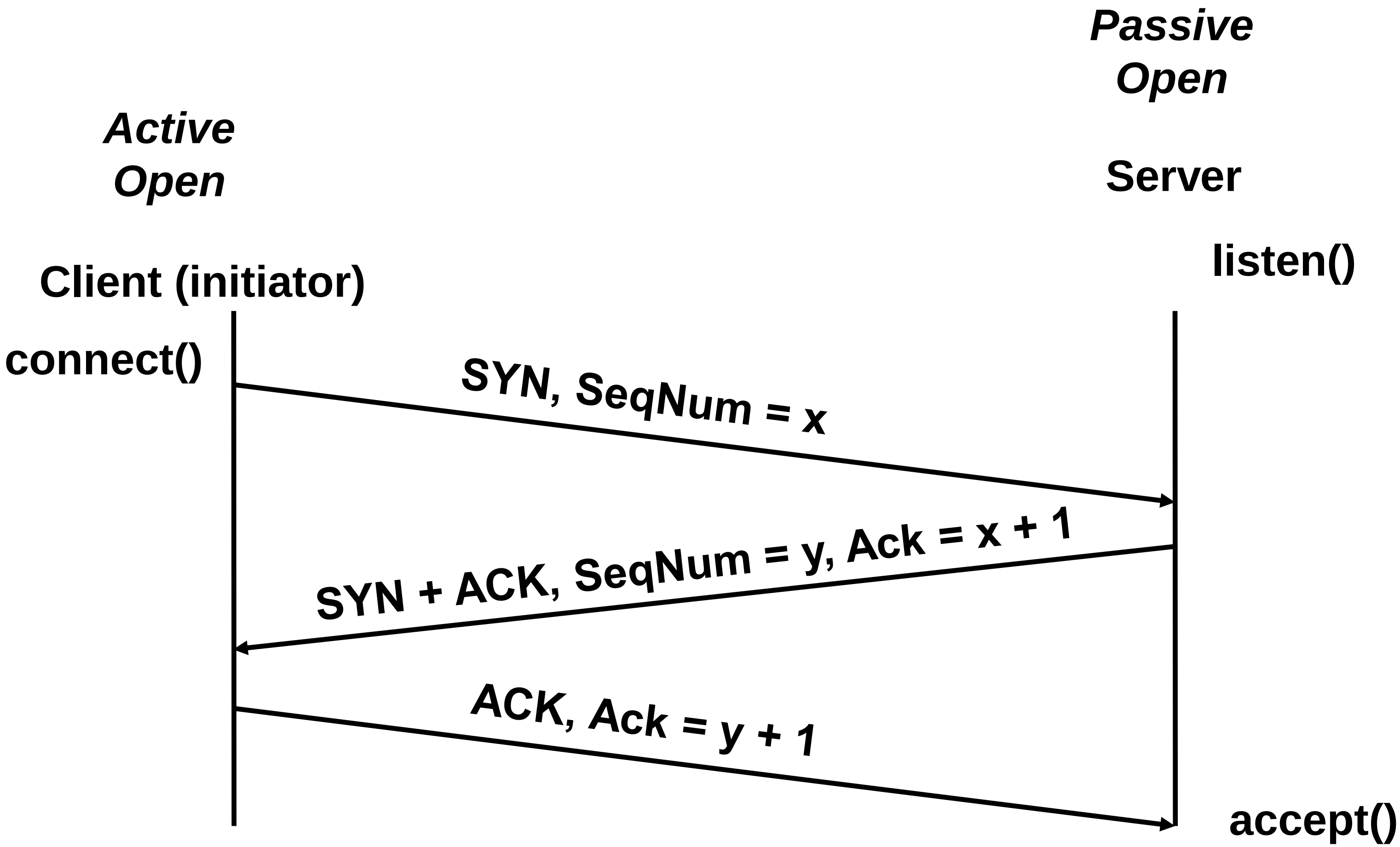
Step 3: A's ACK of the SYN-ACK



A tells B it's likewise okay to start sending

… upon receiving this packet, B can start sending data

Timing Diagram: 3-Way Handshaking



What if the SYN Packet Gets Lost?

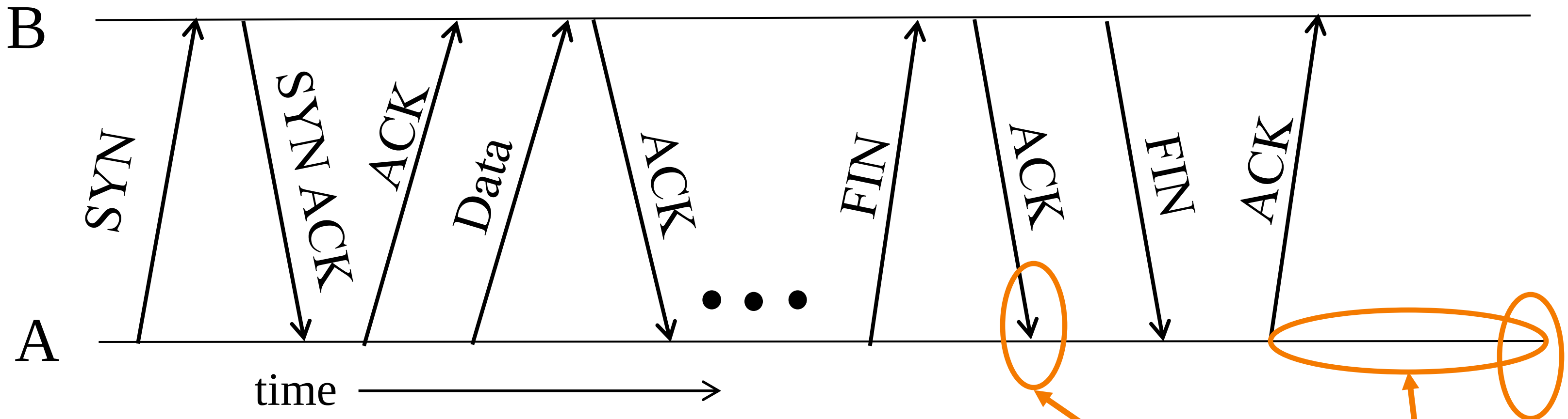
- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and waits for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - **SHOULD** (RFCs 1122 & 2988) use default of 3 seconds
    - Other implementations instead use 6 seconds

SYN Loss and Web Downloads

- User clicks on a hypertext link
  - Browser creates a socket and does a "connect"
  - The "connect" triggers the OS to transmit a SYN
- If the SYN is lost…
  - 3-6 seconds of delay: can be very long
  - User may become impatient
  - … and click the hyperlink again, or click "reload"
- User triggers an "abort" of the "connect"
  - Browser creates a new socket and another "connect"
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes quickly

Tearing Down the Connection

Normal Termination, One Side At A Time



- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one octet in the sequence space
- Other host ack's the octet to confirm
- Closes A's side of the connection, but not B's
  - Until B likewise sends a **FIN**
  - Which A then acks

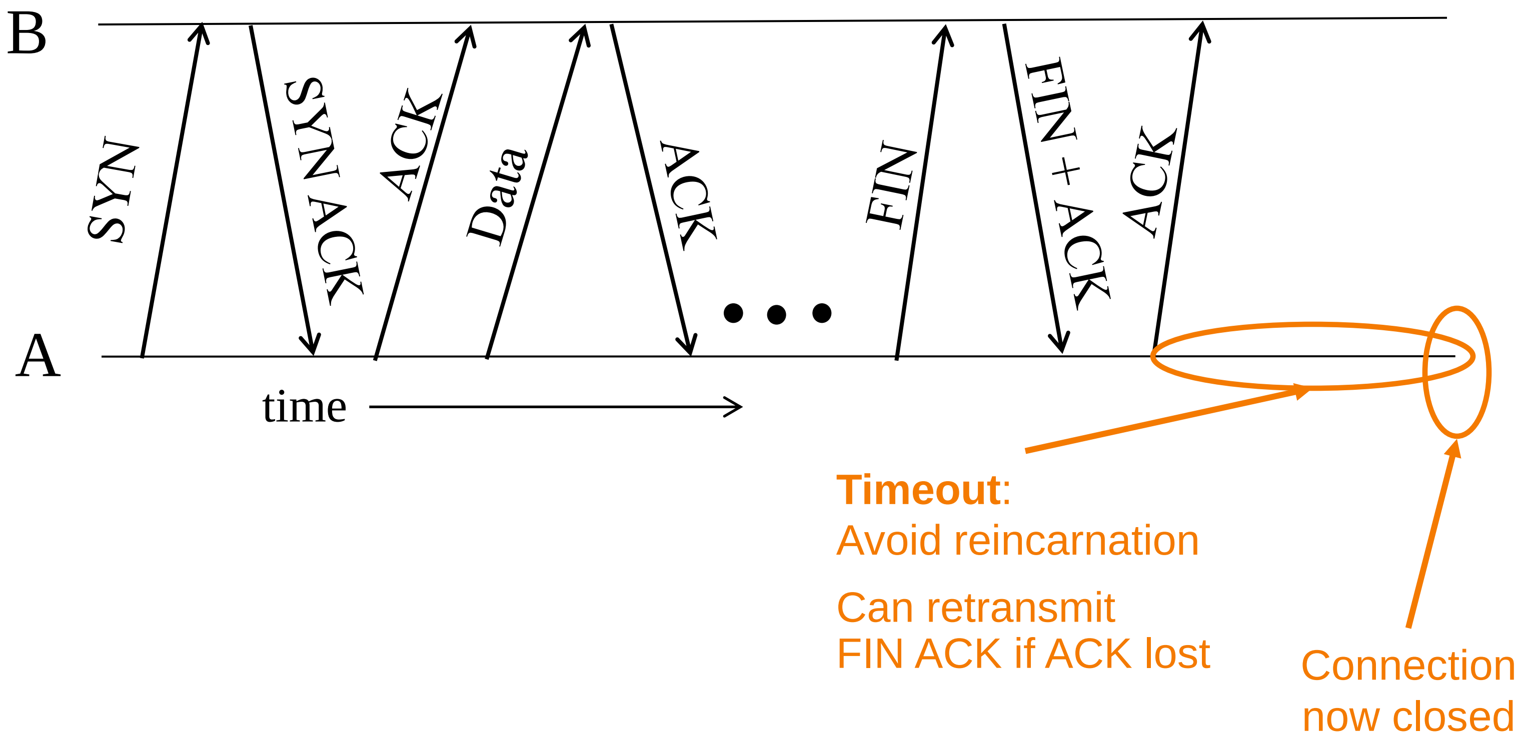
Connection now **closed**

Connection now **half-closed**

Timeout:
Avoid *reincarnation*

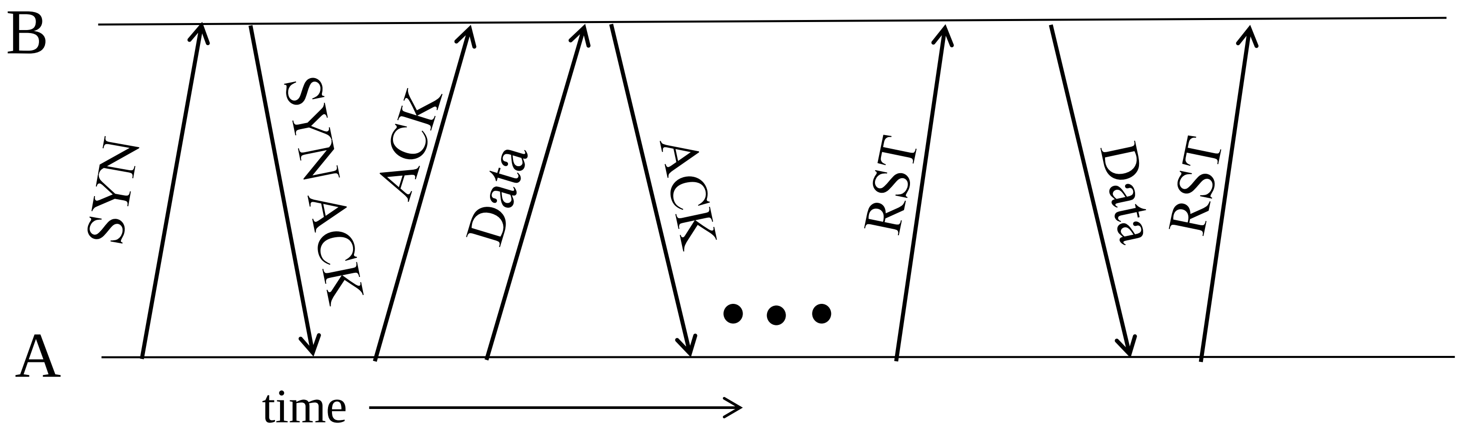
B will retransmit FIN if ACK is lost

Normal Termination, Both Together



- Same as before, but B sets **FIN** with their ack of A's **FIN**

Abrupt Termination



- A sends a RESET (**RST**) to B
  - E.g., because app. process on A crashed
- That's it
  - B does not ack the **RST**
  - Thus, **RST** is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another **RST**

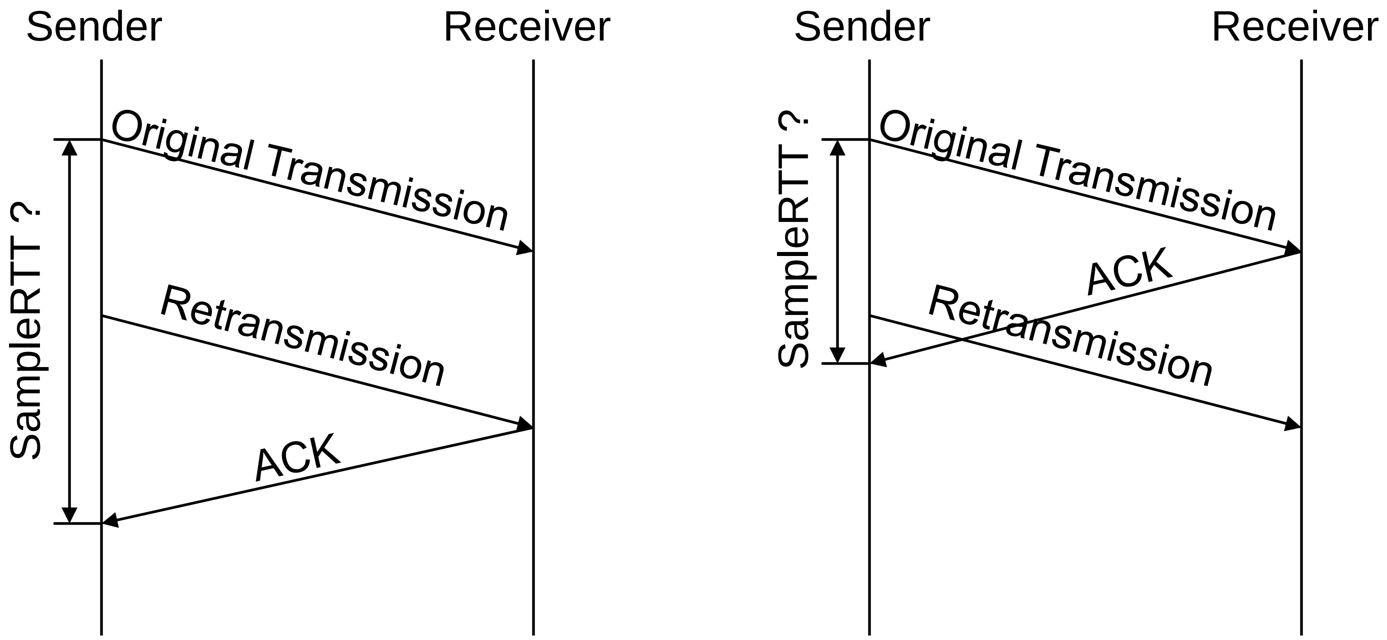
Reliability: TCP Retransmission

Setting Timeout Value

- Sender sets a timeout to wait for an ACK
  - Too short: wasted retransmissions
  - Too long: excessive delays when packet lost
- TCP sets *retransmission timeout* (RTO) as function of RTT
  - Expect ACK to arrive roughly an RTT after data sent
  - … plus slop to allow for variations (e.g., queuing, MAC)
- But: how do we measure RTT?
- And: what is a good estimate for RTT?
- And: what's a good estimate for "slop"?

Problem: Ambiguous Measurement

- How to differentiate between the real ACK, and ACK of the retransmitted packet?



Karn/Partridge Algorithm

- Measure *SampleRTT* only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements
- Also, employ exponential backoff
  - Every time RTO timer expires, set RTO $\leftarrow$ 2·RTO
  - (Up to maximum $\geq$ 60 sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to computed value

Next Step

- Turn these individual RTT measurements into an estimate of RTT that we can use to compute RTO
- Challenge:
  - Average RTT, but recent values more important

Exponential Averaging

Exponential Averaging:

- Estimate(n) = α Estimate(n-1) + (1-α) Value(n)

Expanding:

- Estimate(n) = (1-α) Sum {α^k Value(n-k)}
- Weight on historical data decreases exponentially

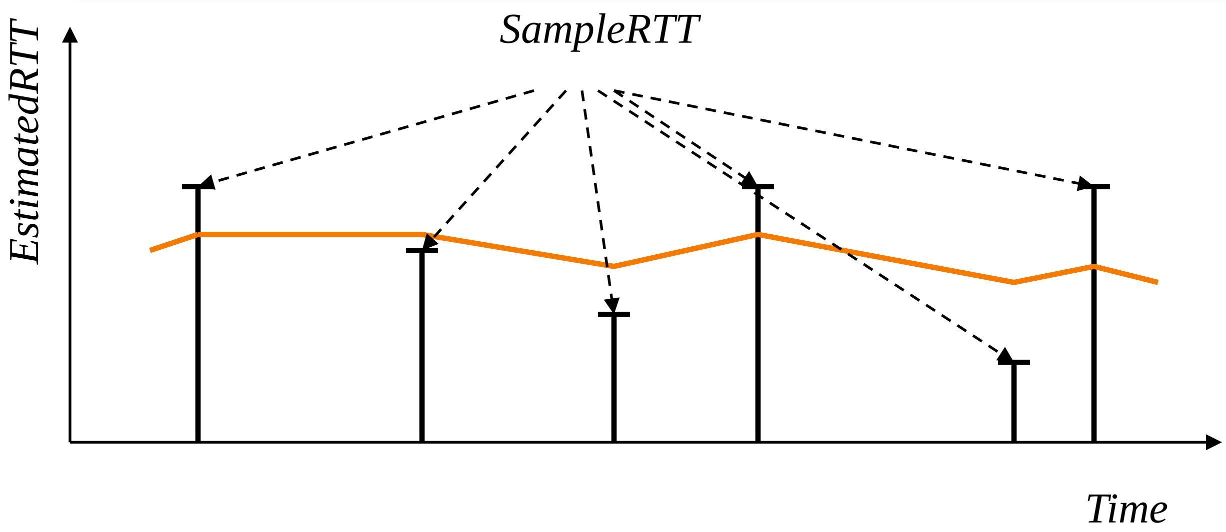
RTT Estimation

- Use exponential averaging:

$$SampleRTT = AckRcvdTime - SendPacketTime$$

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT$$

$$\alpha = 7/8 \text{ (for one measurement per flight)}$$



Jacobson/Karels Algorithm

- Compute "slop" in terms of observed variability
  - standard deviation requires expensive square root
  - Use mean deviation instead

- Deviation = | SampleRTT – EstimatedRTT |

- EstimatedDeviation: exp. average of Deviation

- RTO = EstimatedRTT + 4 x EstimatedDeviation

This is all very interesting, but…..

- Implementations often use a coarse-grained timer
  - 500 msec is typical
- So what?
  - Above algorithms are largely irrelevant
  - **Incurring a timeout is expensive**
- So we rely on duplicate ACKs