

# Communication Networks

Spring 2017



Laurent Vanbever

[www.vanbever.eu](http://www.vanbever.eu)

ETH Zürich (D-ITET)

May, 8 2017

Material inspired from Scott Shenker & Jennifer Rexford

Two weeks ago on  
**Communication Networks**

# Routing Security

[image source]

# Routing security

attacks & mitigation

intra-domain  
routing

insider

inter-domain  
routing

in/outsider

# Routing security

attacks & mitigation

intra-domain  
routing

insider

inter-domain  
routing

in/outsider

Most of the attacks on intra-domain routing aim at performing Denial-of-Service (DoS) or intercept traffic

Interception	eavesdrop on/drop/modify/inject/delay traffic steer traffic along paths controlled by the attacker
--------------	---

DoS	induce churn to overload the routers announce/withdraw at fast pace  floods the routers link-state database inject thousands of prefixes  induce congestion/higher delay steer traffic along fewer/low-throughput paths  prevent reachability steer traffic along blackholes or loops
-----	---

The solution is quite simple:

**Rely on cryptography!**

Problem

Bogus advertisements can be injected

Legitimate advertisements can be tampered with

Solution 1  
(light)

Use Cryptographic Authentication (header)  
integrity and authentication

Solution 2  
(heavy)

Encrypt the entire advertisement (header/payload)  
integrity, authentication *and* confidentiality

# Routing security

attacks & mitigation

intra-domain  
routing

insider

inter-domain  
routing

in/outsider

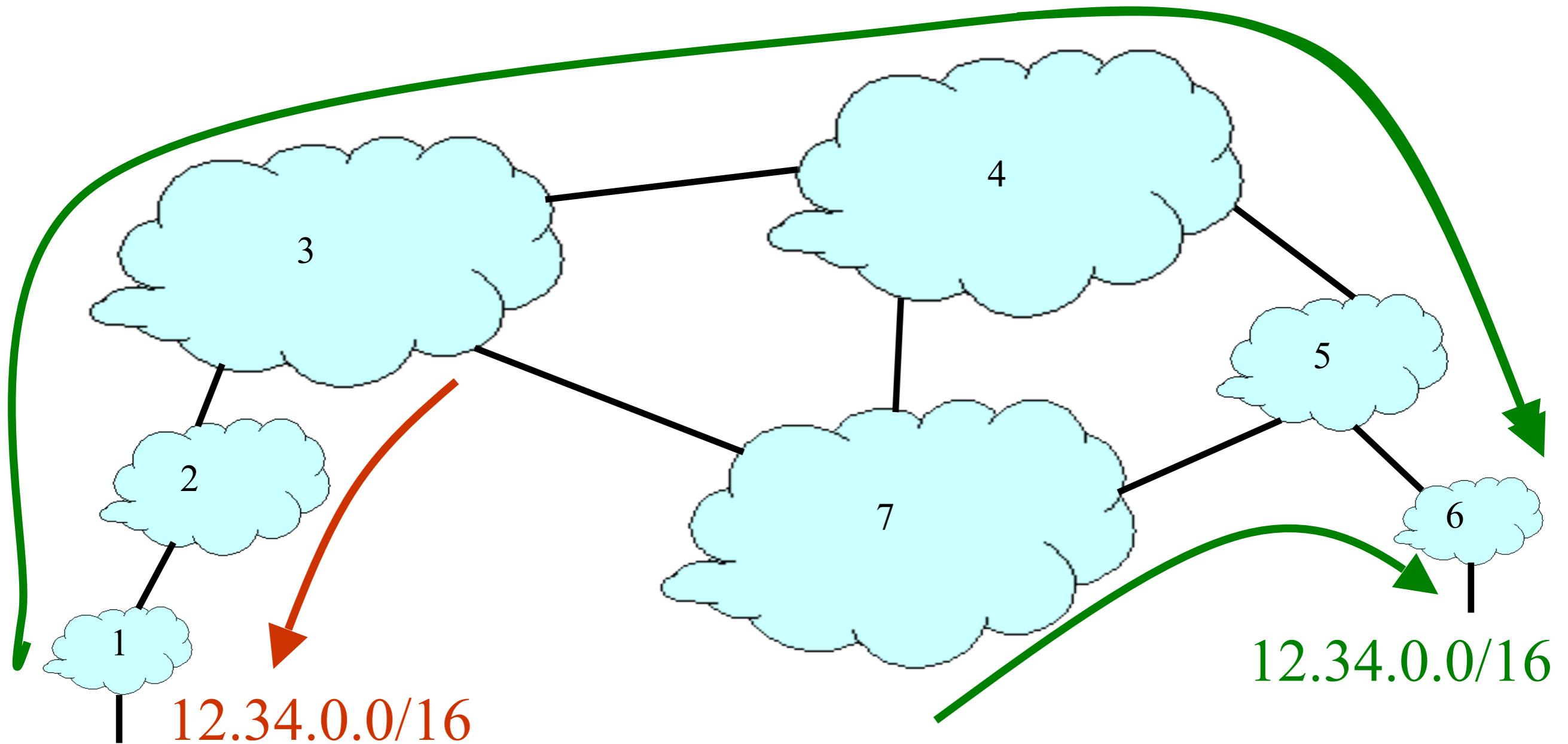
# BGP (lack of) security: problems & solutions

- #1 BGP does not validate the origin of advertisements
- #2 BGP does not validate the content of advertisements
- #3 Proposed Enhancements
- #4 What about the data plane?
- #5 What's the Internet to do anyway?

# BGP (lack of) security: problems & solutions

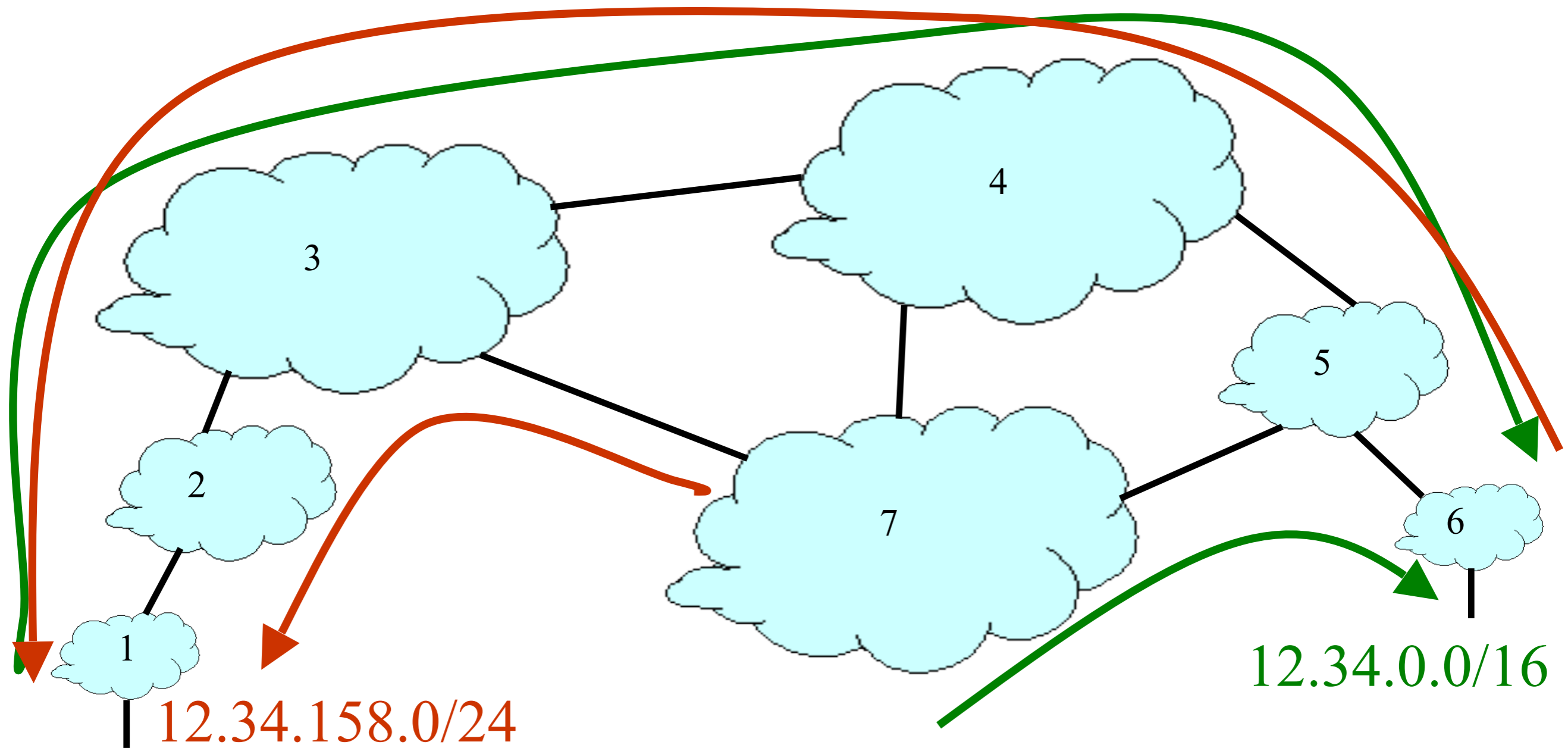
- #1 BGP does not validate the origin of advertisements
- #2 BGP does not validate the content of advertisements
- #3 Proposed Enhancements
- #4 What about the data plane?
- #5 What's the Internet to do anyway?

# Prefix Hijacking



- **Blackhole:** data traffic is discarded
- **Snooping:** data traffic is inspected, then redirected
- **Impersonation:** traffic sent to bogus destinations

# Sub-Prefix Hijacking



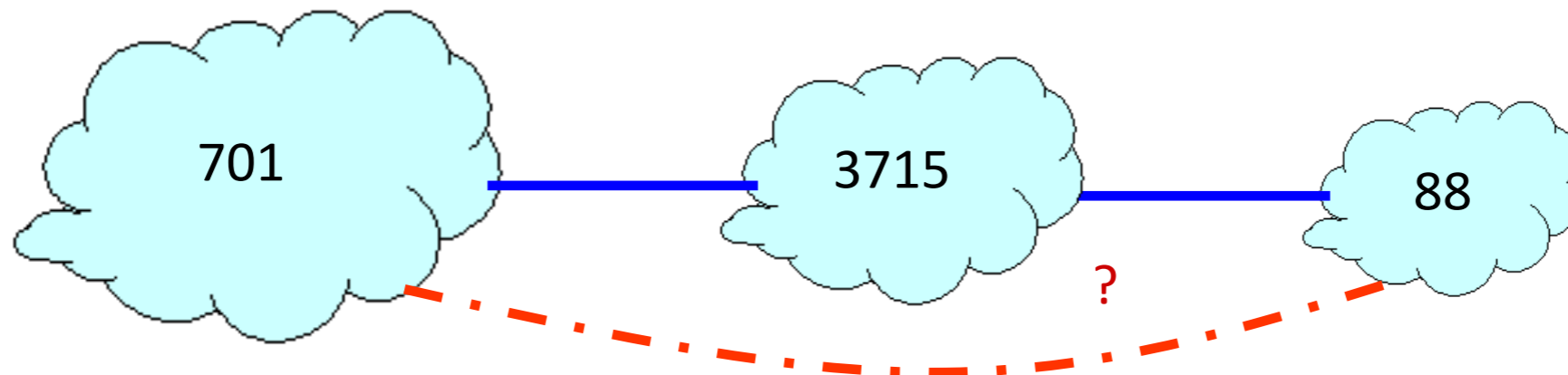
- **Originating a more-specific prefix**
  - Every AS picks the bogus route for that prefix
  - Traffic follows the longest matching prefix

# BGP (lack of) security: problems & solutions

- #1 BGP does not validate the origin of advertisements
- #2 BGP does not validate the content of advertisements
- #3 Proposed Enhancements
- #4 What about the data plane?
- #5 What's the Internet to do anyway?

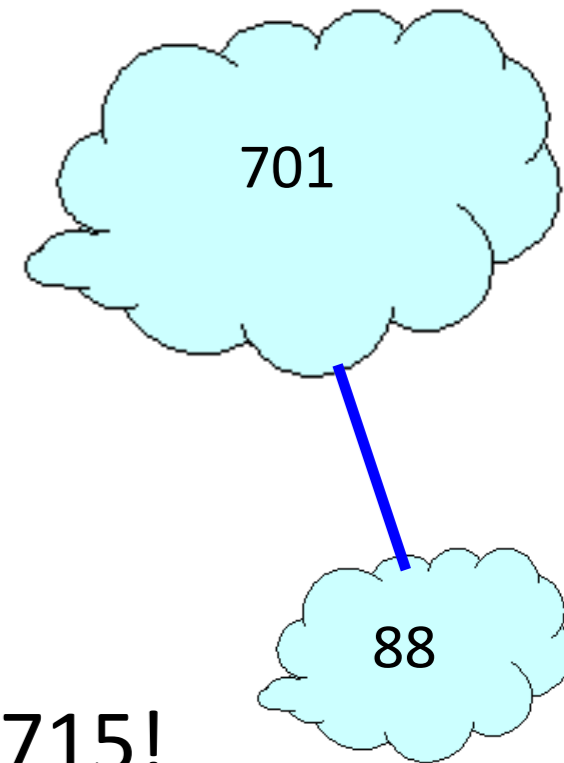
# Bogus AS Paths

- Remove ASes from the AS path
  - E.g., turn “701 3715 88” into “701 88”
- Motivations
  - Attract sources that normally try to avoid AS 3715
  - Help AS 88 look like it is closer to the Internet’s core
- Who can tell that this AS path is a lie?
  - Maybe AS 88 *does* connect to AS 701 directly



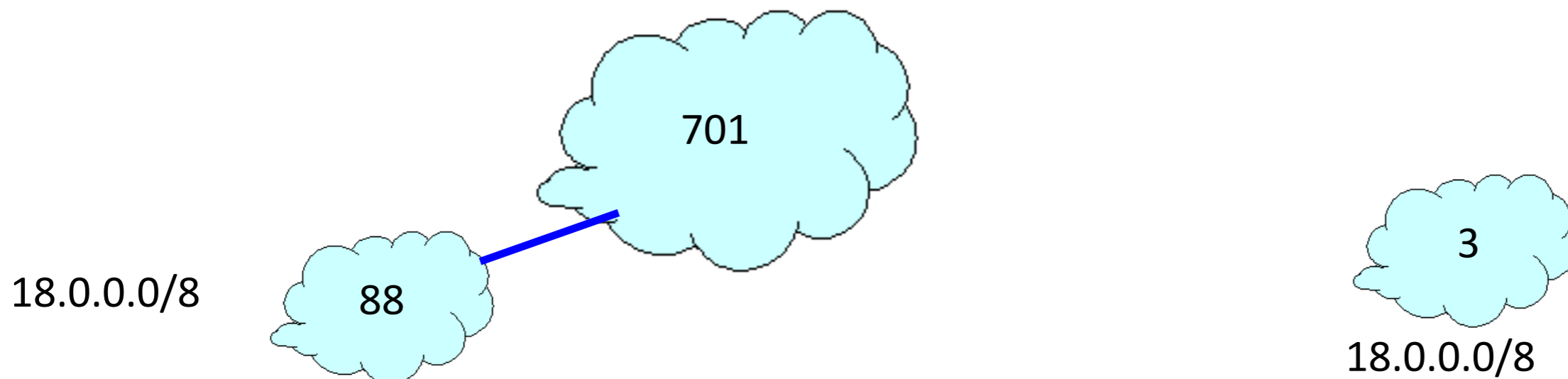
# Bogus AS Paths

- **Add ASes to the path**
  - E.g., turn “701 88” into “701 3715 88”
- **Motivations**
  - Trigger loop detection in AS 3715
    - Denial-of-service attack on AS 3715
    - Or, blocking unwanted traffic coming from AS 3715!
  - Make your AS look like it has richer connectivity
- **Who can tell the AS path is a lie?**
  - AS 3715 could, if it could see the route
  - AS 88 could, but would it really care?



# Bogus AS Paths

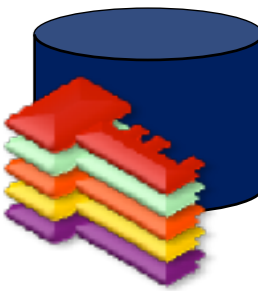
- Adds AS hop(s) at the end of the path
  - E.g., turns “701 88” into “701 88 3”
- Motivations
  - Evade detection for a bogus route
  - E.g., by adding the legitimate AS to the end
- Hard to tell that the AS path is bogus...
  - Even if other ASes filter based on prefix ownership



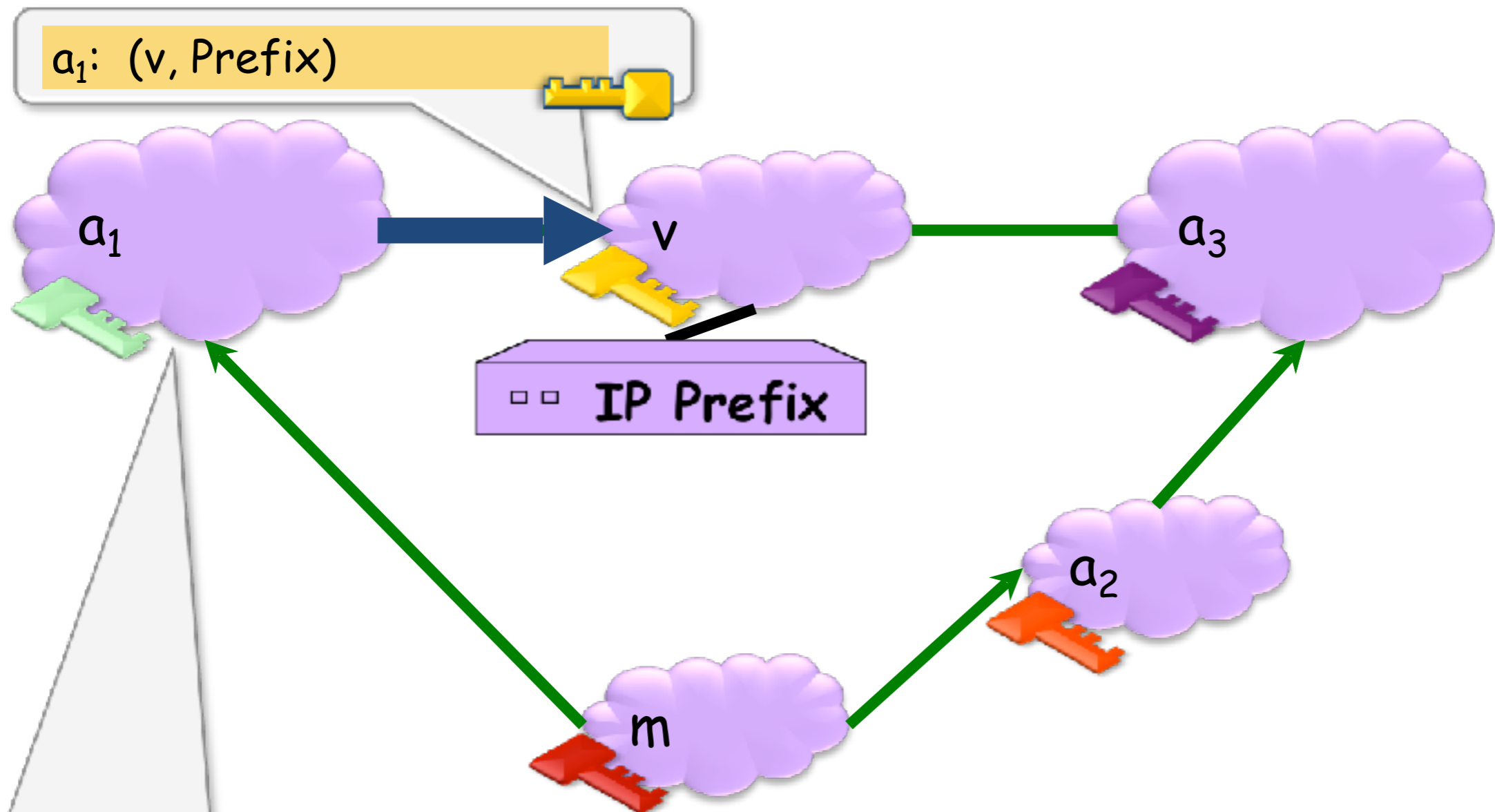
# BGP (lack of) security: problems & solutions

- #1 BGP does not validate the origin of advertisements
- #2 BGP does not validate the content of advertisements
- #3 **Proposed Enhancements**
- #4 What about the data plane?
- #5 What's the Internet to do anyway?

# Secure BGP



Origin Authentication + cryptographic signatures



$a_1: (v, \text{Prefix})$

$m: (a_1, v, \text{Prefix})$

Who knows  $v$ 's public key can  
verify the signature sent by  $v$ .

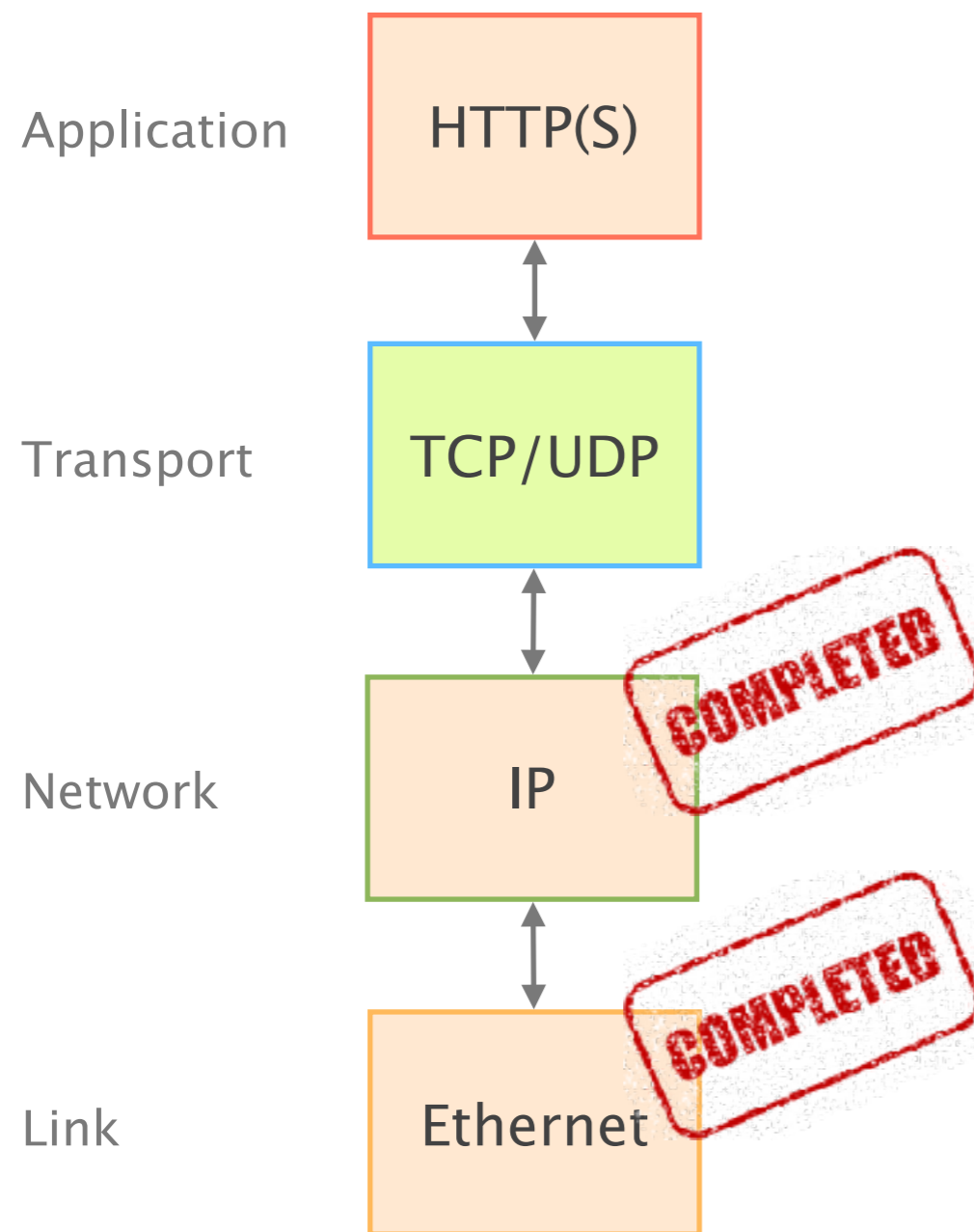


# S-BGP Secure Version of BGP

- **Address attestations**
  - Claim the right to originate a prefix
  - Signed and distributed out-of-band
  - Checked through delegation chain from ICANN
- **Route attestations**
  - Distributed as an attribute in BGP update message
  - Signed by each AS as route traverses the network
- **S-BGP can validate**
  - AS path indicates the order ASes were traversed
  - No intermediate ASes were added or removed

**This week on**  
**Communication Networks**

We're continuing our journey up the layers,  
now looking at **the transport layer**



# What do we need in the Transport layer?

Functionality implemented in **network**

- Keep minimal (easy to build, broadly applicable)

Functionality implemented in the **application**

- Keep minimal (easy to write)
- Restricted to application-specific functionality

Functionality implemented in the “**network stack**”

- The shared networking code on the host
- This relieves burden from both app and network
- The transport layer is a key component here

# What do we need in the Transport layer?

## Application layer

- Communication for specific applications
- *e.g.*, HyperText Transfer Protocol (HTTP),  
File Transfer Protocol (FTP)

## Network layer

- Global communication between hosts
- Hides details of the link technology
- *e.g.*, Internet Protocol (IP)

# What Problems Should Be Solved Here?

Data delivering, to the *correct* application

- IP just points towards next protocol
- *Transport needs to demultiplex incoming data (ports)*

Files or bytestreams abstractions for the applications

- Network deals with packets
- *Transport layer needs to translate between them*

Reliable transfer (if needed)

Not overloading the receiver

Not overloading the network

# What Is Needed to Address These?

*Demultiplexing:* identifier for application process

- Going from host-to-host (IP) to process-to-process

*Translating between bytestreams and packets:*

- Do segmentation and reassembly

*Reliability:* ACKs and all that stuff

*Corruption:* Checksum

*Not overloading receiver:* “Flow Control”

- Limit data in receiver’s buffer

*Not overloading network:* “Congestion Control”

# UDP: Datagram messaging service

UDP provides a **connectionless, unreliable** transport service

- No-frills extension of “best-effort” IP
- UDP provides **only two services** to the App layer
  - Multiplexing/Demultiplexing among processes
  - Discarding corrupted packets (optional)

# TCP: Reliable, in-order delivery

TCP provides a **connection-oriented, reliable, bytestream** transport service

## ***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!

# What transport protocols do **not** provide

Delay and/or bandwidth guarantees

- This cannot be offered by transport
- Requires support at IP level (*and let's not go there*)

Sessions that survive change-of-IP-address

- This is an artifact of current implementations
- As we shall see....

# Important Context: Sockets and Ports

**Sockets:** an operating system abstraction

**Ports:** a networking abstraction

- This is not a port on a switch (which is an interface)
- Think of it as a *logical interface* on a host

# Sockets

A socket is a software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system

- `socketID = socket(..., socket.TYPE)`
- `socketID.sendto(message, ...)`
- `socketID.recvfrom(...)`

Two important types of sockets

- UDP socket: TYPE is `SOCK_DGRAM`
- TCP socket: TYPE is `SOCK_STREAM`

# Ports

**Problem:** which app (socket) gets which packets

**Solution:** port as transport layer identifier (16 bits)

- Packet carries source/destination port numbers in transport header

OS stores mapping between sockets and ports

- Port: in packets
- Socket: in OS

# More on Ports

Separate 16-bit port address space for UDP, TCP

“Well known” ports (0-1023)

- Agreement on which services run on these ports
- *e.g.*, ssh:22, http:80
- Client (app) knows appropriate port on server
- Services can listen on well-known port

Ephemeral ports (most 1024-65535):

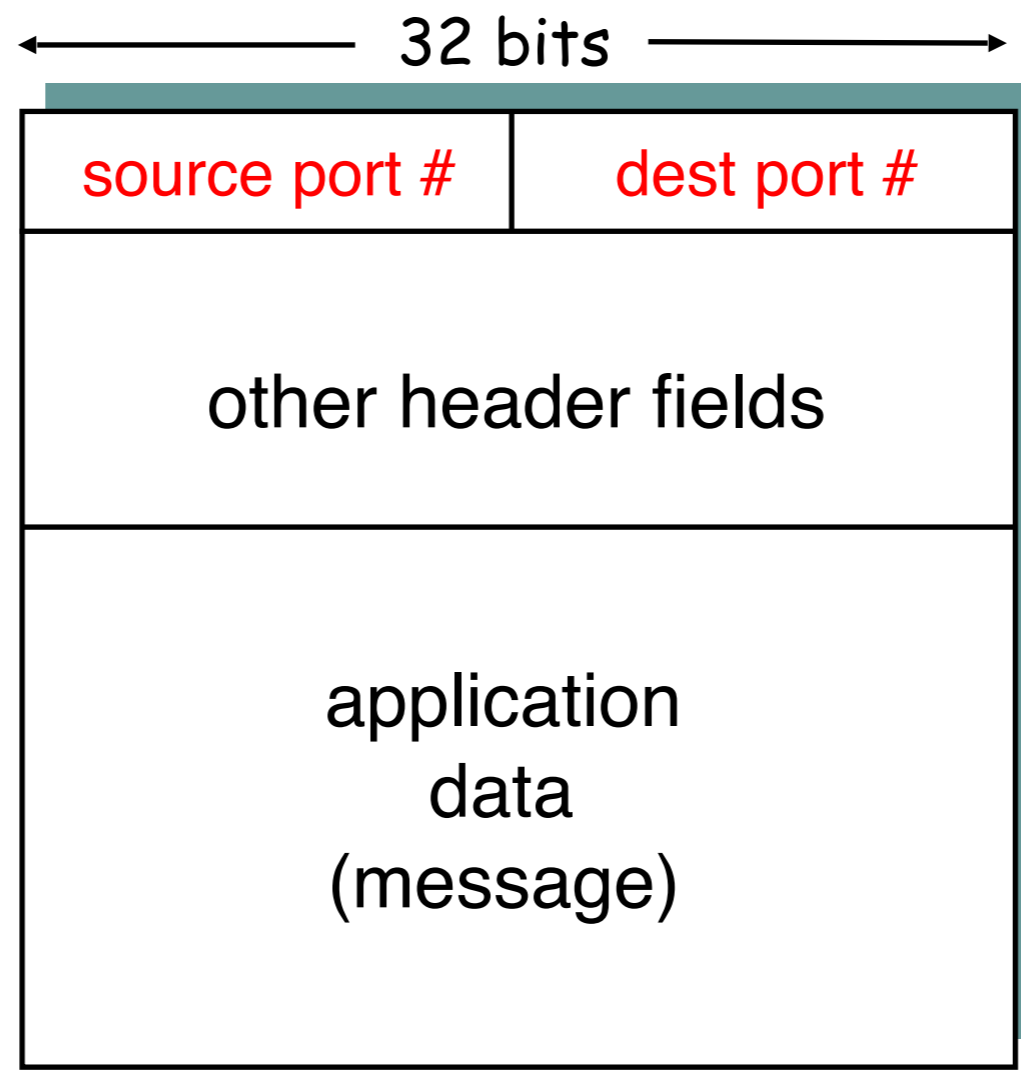
- Given to clients (at random)

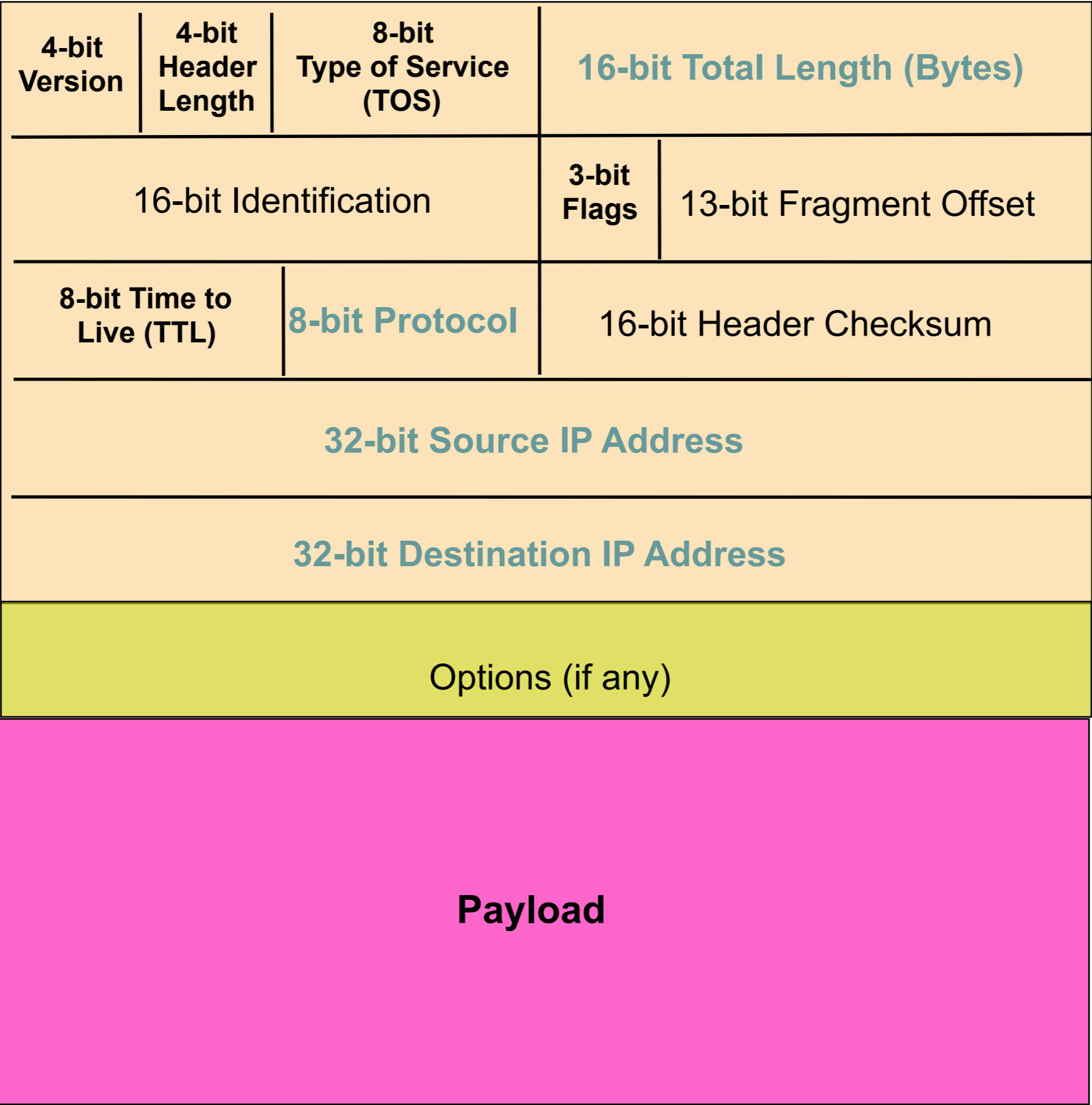
# Multiplexing and Demultiplexing

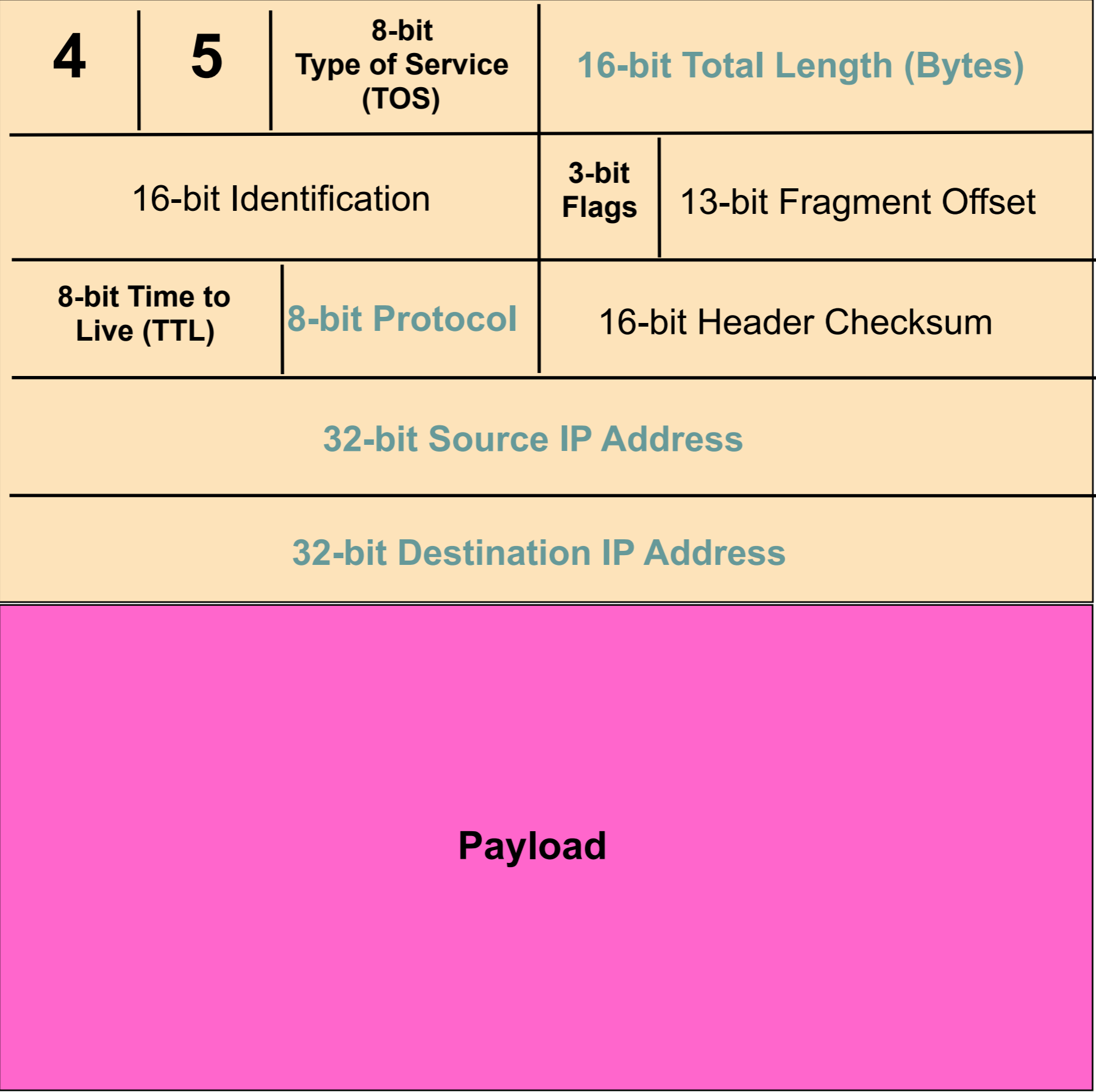
Host receives IP datagrams

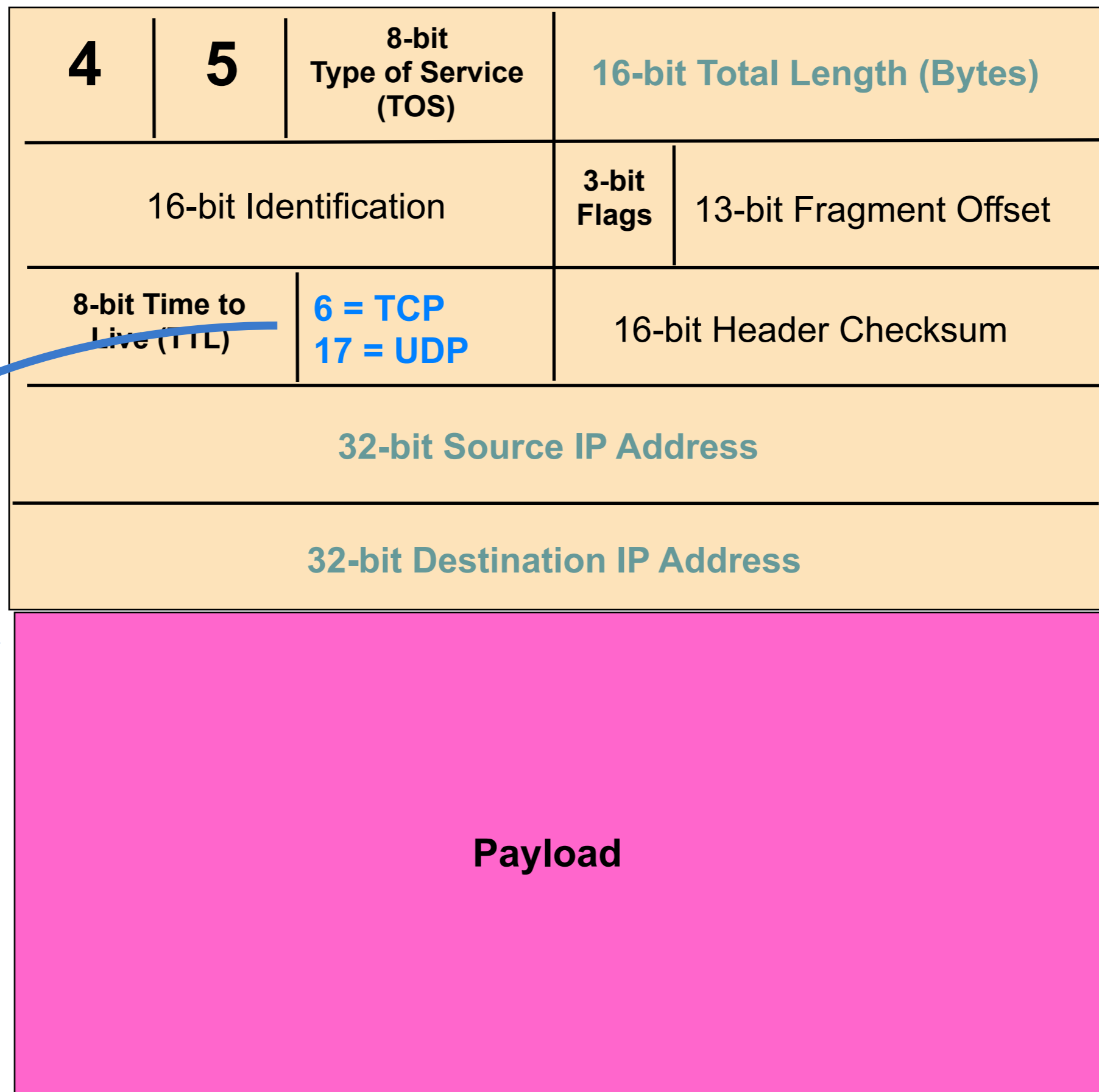
- Each datagram has source and destination IP **address**,
- Each segment has source and destination **port** number

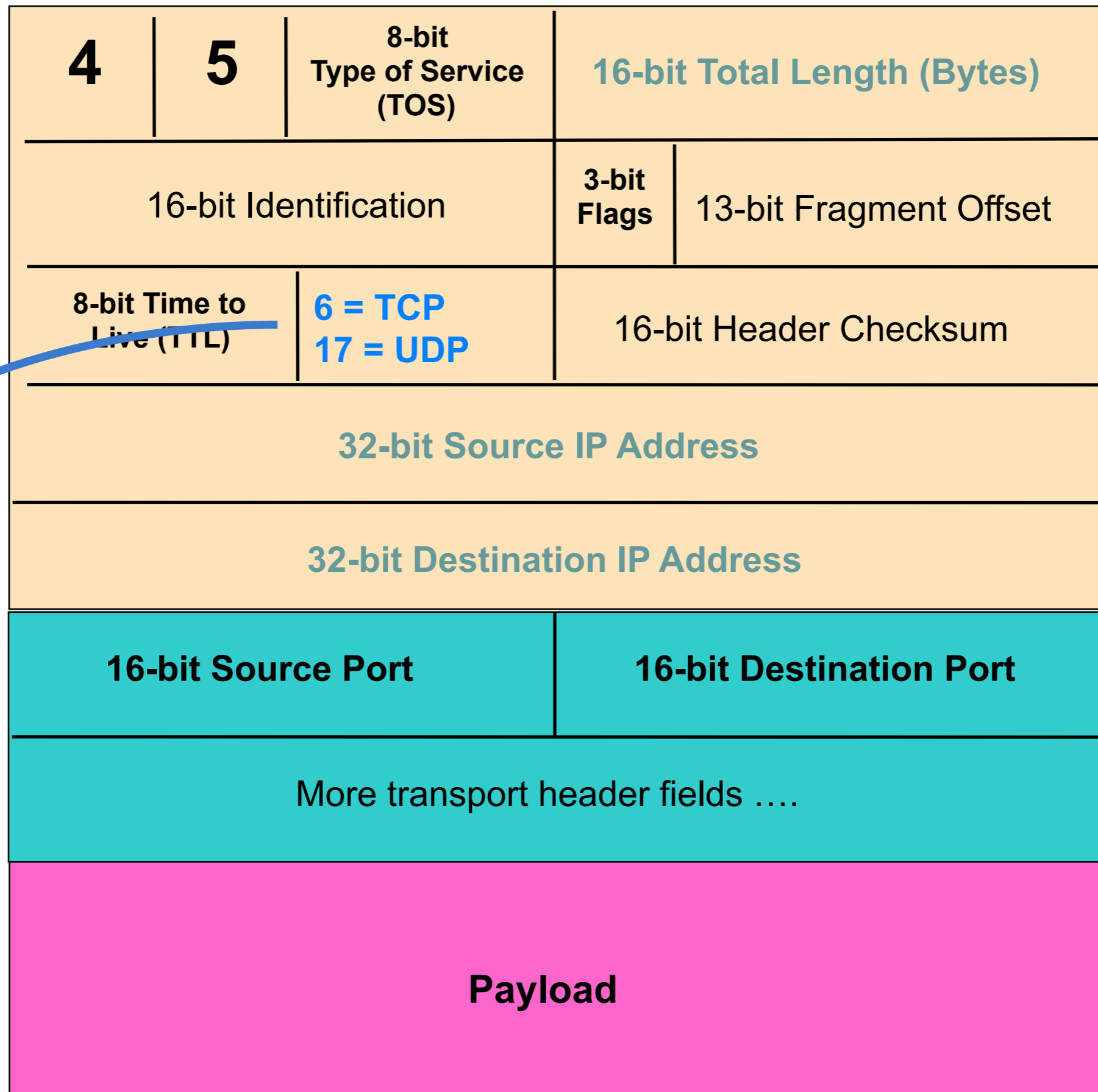
Host uses IP addresses and port numbers to direct the segment to appropriate **socket**











# Connection Mappings

For UDP ports (SOCK\_DGRAM)

- OS stores (local port, local IP address)  $\leftrightarrow$  socket

For TCP ports (SOCK\_STREAM)

- OS stores (local port, local IP, remote port, remote IP)  $\leftrightarrow$  socket

***Why the difference?***

***Implications for mobility***

***Why do you need to include local IP?***

**UDP**

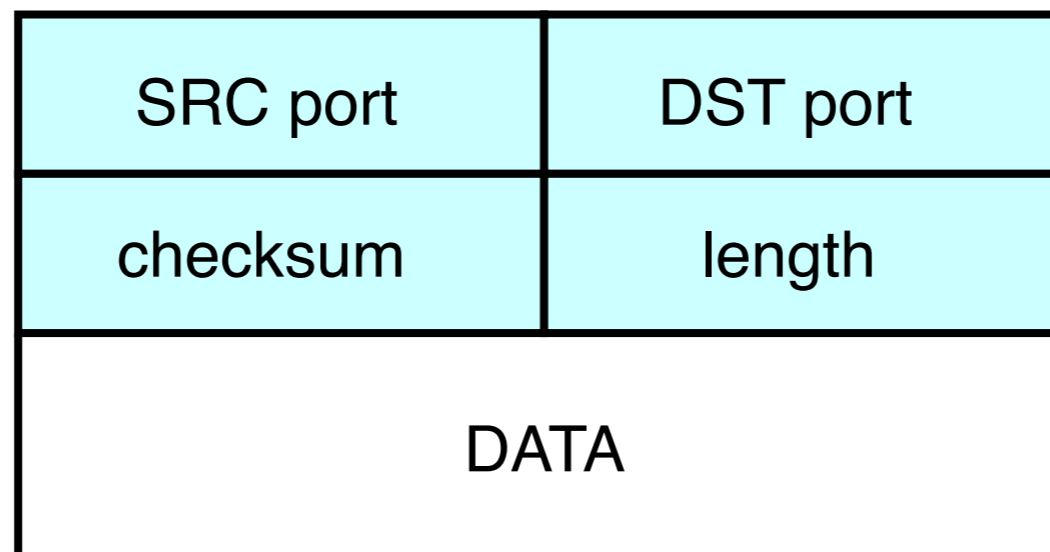
# UDP: User Datagram Protocol

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

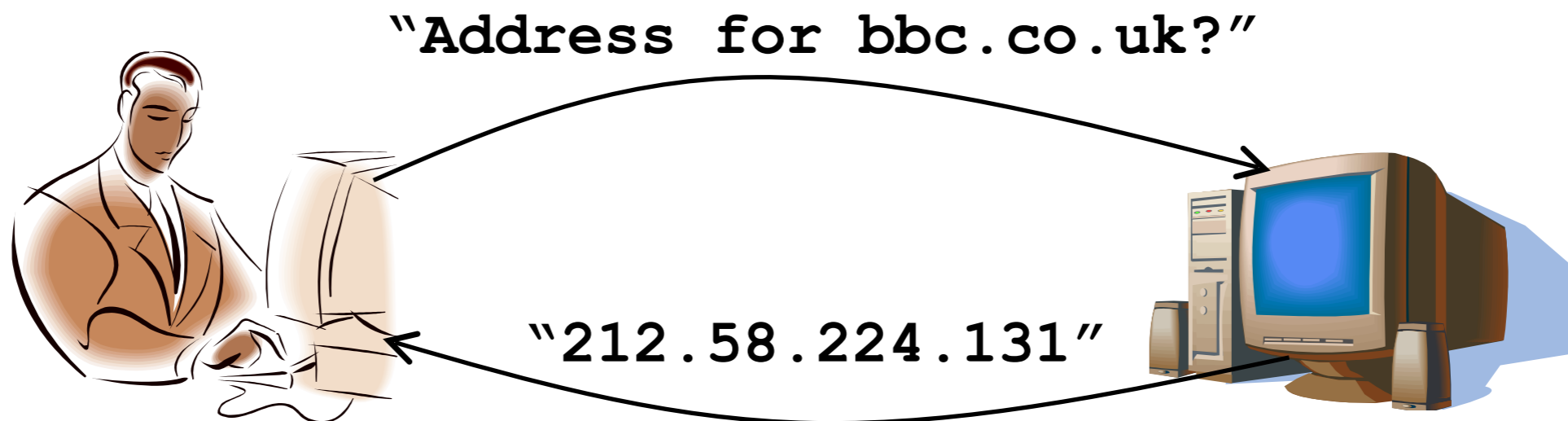
# Popular Applications That Use UDP

Some **interactive streaming** apps

- Retransmitting lost/corrupted packets often pointless: by the time the packet is retransmitted, it's too late
- telephone calls, video conferencing, gaming...
- ***Modern streaming protocols using TCP (and HTTP)***

Simple query protocols like Domain Name System (DNS)

- Connection establishment overhead would double cost
- Easier to have **application** retransmit if needed



**TCP**

# Transmission Control Protocol (TCP)

Reliable, in-order delivery *(previously, but quick review)*

- Ensures byte stream (eventually) arrives intact
  - In the presence of **corruption** and **loss**

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

Flow control *(previously, but quick review)*

- Ensures that sender doesn't overwhelm receiver

Congestion control *(next week)*

- Dynamic adaptation to network path's capacity

# Basic Components of Reliability

## ACKs

- Can't be reliable without knowing whether data has arrived
- ***TCP uses byte sequence numbers to identify payloads***

## Checksums

- Can't be reliable without knowing whether data is corrupted
- ***TCP does checksum over TCP and pseudoheader***

## Timeouts and retransmissions

- Can't be reliable without retransmitting lost/corrupted data
- ***TCP retransmits based on timeouts and duplicate ACKs***
- *Timeout based on estimate of RTT*

# Other TCP Design Decisions

Sliding window flow control

- Allow  $W$  contiguous bytes to be in flight

Cumulative acknowledgements

- Selective ACKs (full information) also supported (ignore)

Single timer set after each payload is ACKed

- Timer is effectively for the “next expected payload”
- When timer goes off, resend that payload and wait
  - And double timeout period

Various tricks related to “fast retransmit”

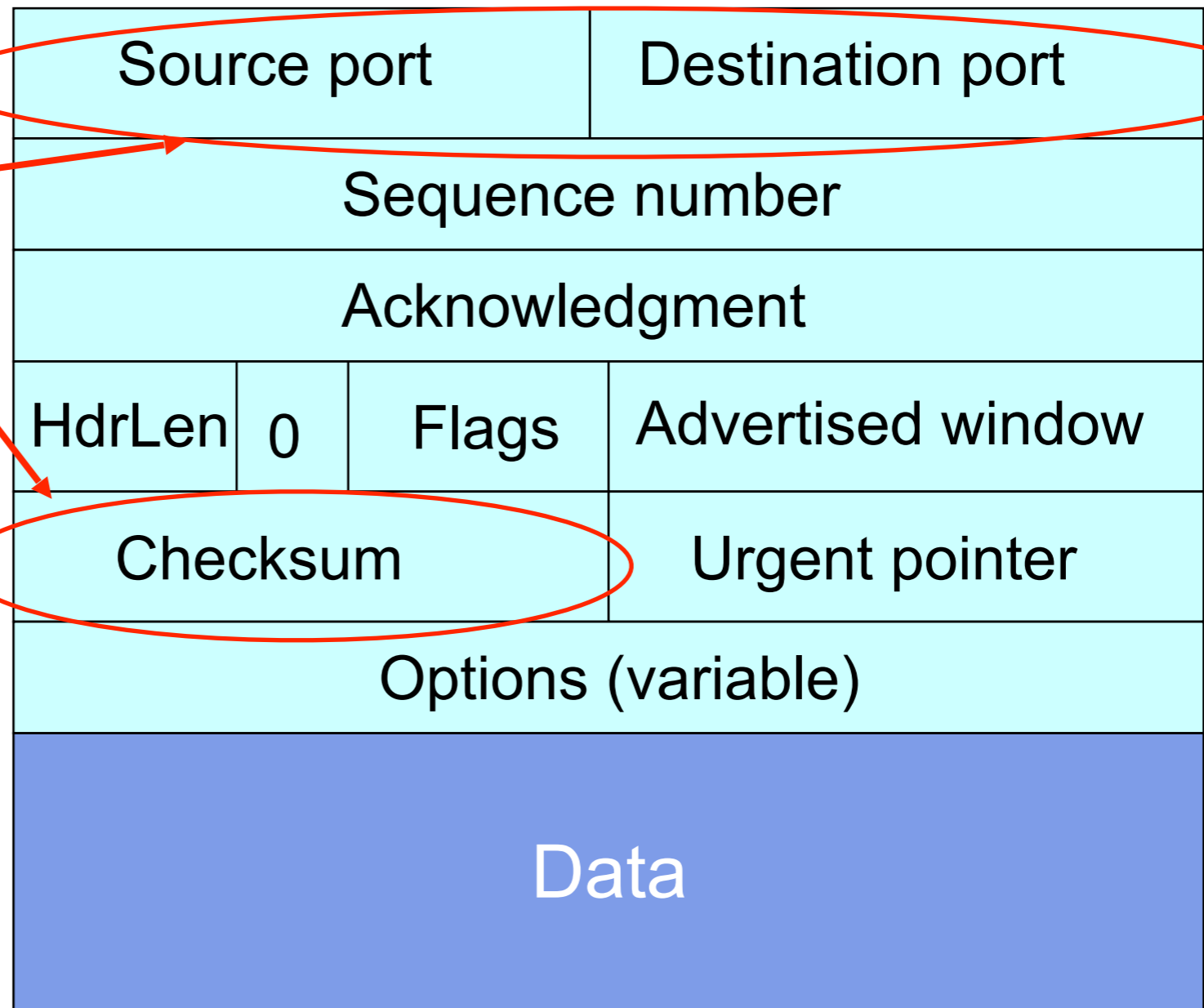
- Using duplicate ACKs to trigger retransmission

# TCP Header

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

# TCP Header

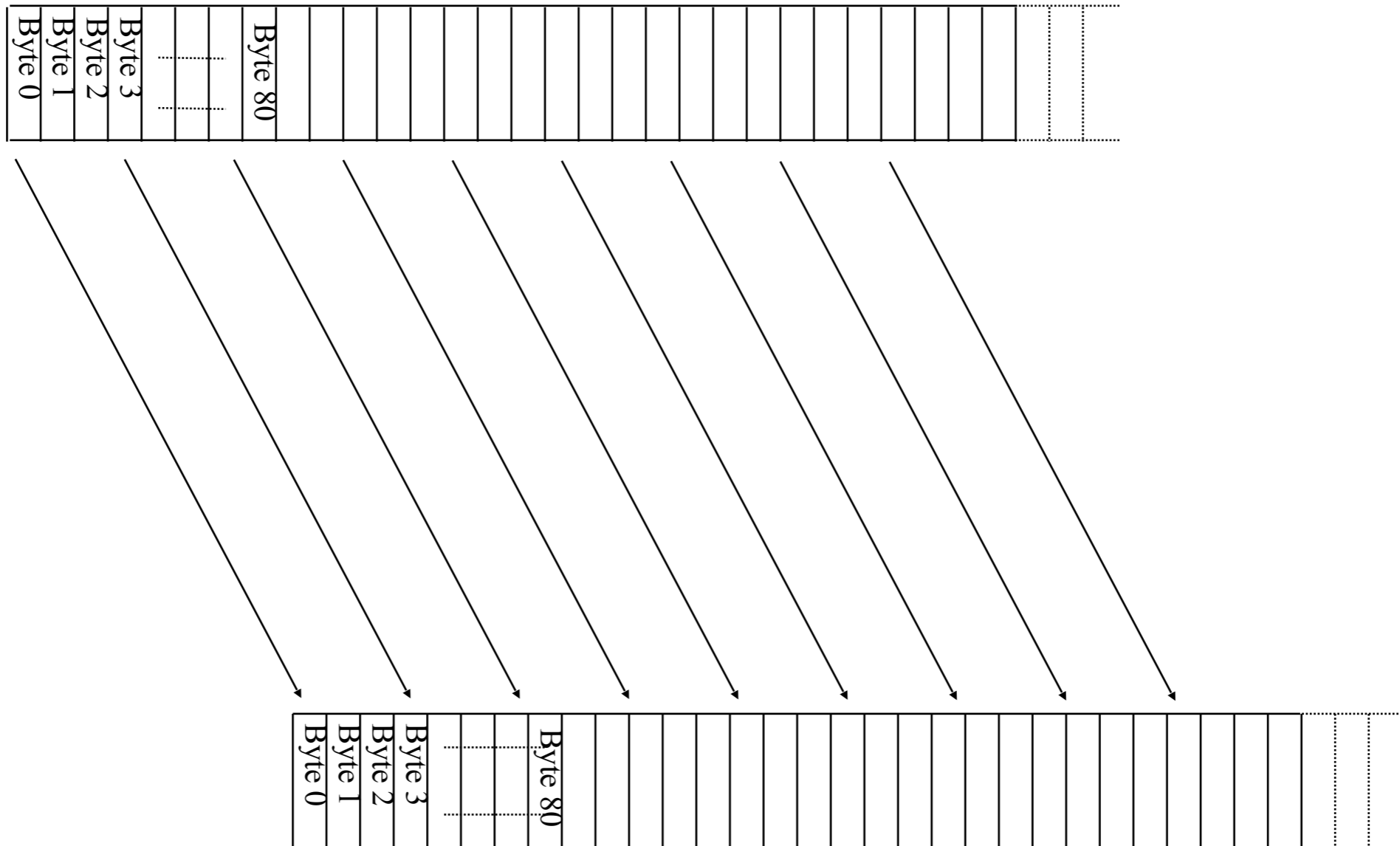
These should  
be familiar



# Segments and Sequence Numbers

# TCP “Stream of Bytes” Service...

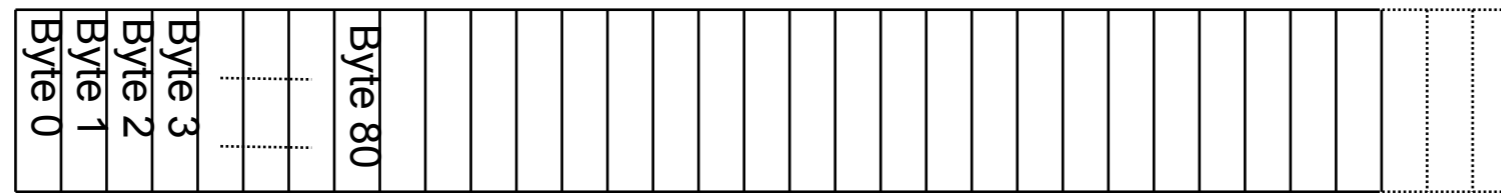
Application @ Host A



Application @ Host B

# ... Provided Using TCP “Segments”

Host A



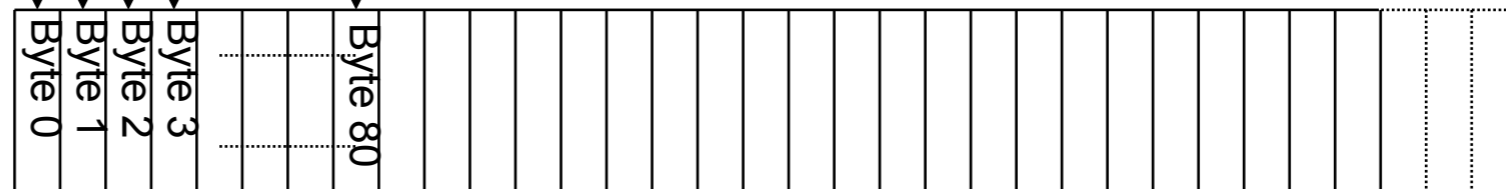
TCP Data

*Segment* sent when:

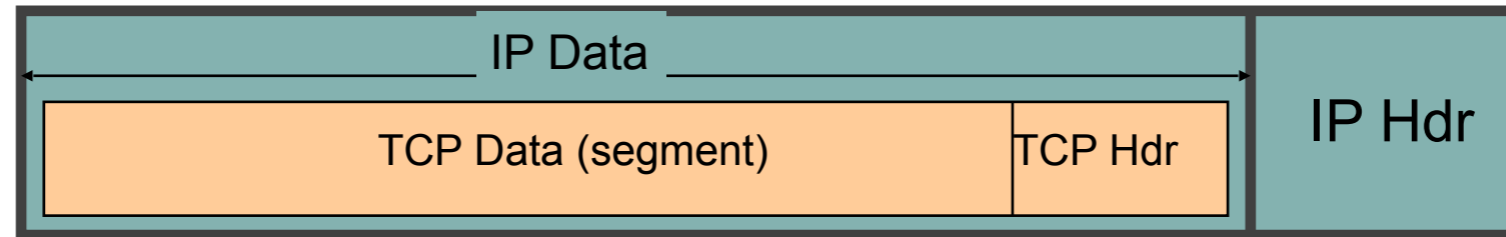
1. Segment full (Max Segment Size),
2. Not full, but times out

TCP Data

Host B



# TCP Segment



## IP packet

- No bigger than Maximum Transmission Unit (MTU)
- E.g., up to 1500 bytes with Ethernet

## TCP packet

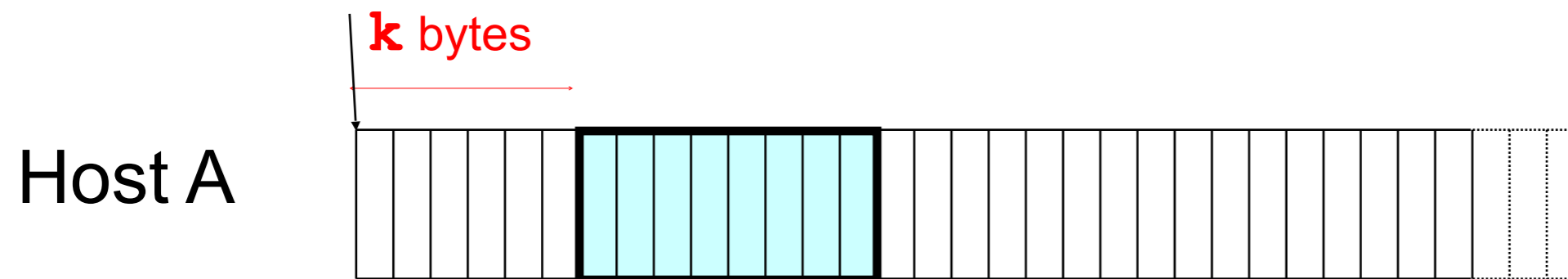
- IP packet with a TCP header and data inside
- TCP header  $\geq$  20 bytes long

## TCP segment

- No more than Maximum Segment Size (MSS) bytes
- E.g., up to 1460 consecutive bytes from the stream
- $MSS = MTU - (IP \text{ header}) - (TCP \text{ header})$

# Sequence Numbers

ISN (initial sequence number)



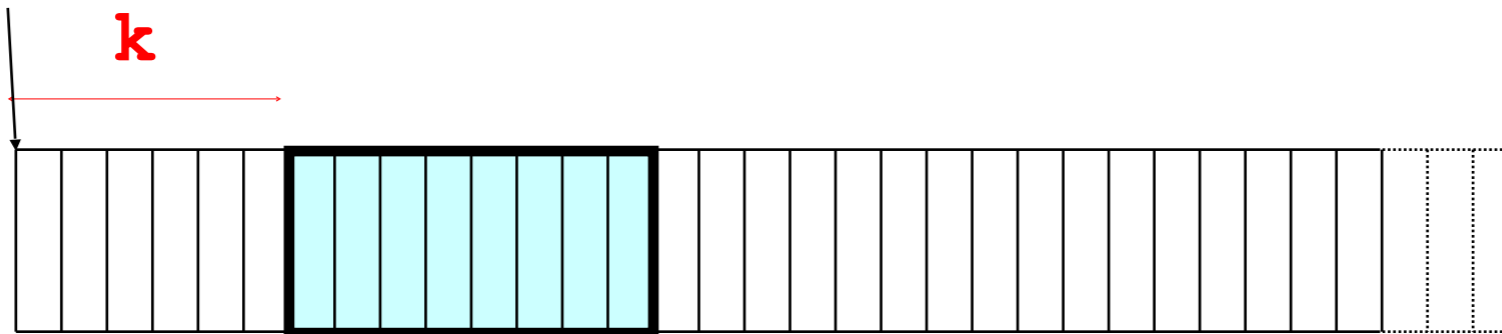
Sequence number  
= 1<sup>st</sup> byte in segment =  
 $\text{ISN} + k$

# Sequence Numbers

ISN (initial sequence number)

$k$

Host A



Sequence number  
= 1<sup>st</sup> byte in segment =  
 $\text{ISN} + k$

TCP Data TCP HDR

TCP Data TCP HDR

ACK sequence number  
= next expected byte  
=  $\text{seqno} + \text{length}(\text{data})$

Host B



# ACKing and Sequence Numbers

Sender sends packet

- Data starts with sequence number  $X$
- Packet contains  $B$  bytes
  - $X, X+1, X+2, \dots, X+B-1$

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 contiguous byte received is smaller value  $Y$ 
  - ACK acknowledges  $Y+1$
  - Even if this has been ACKed before

# Normal Pattern

Sender: seqno= $X$ , length= $B$

Receiver: ACK= $X+B$

Sender: seqno= $X+B$ , length= $B$

Receiver: ACK= $X+2B$

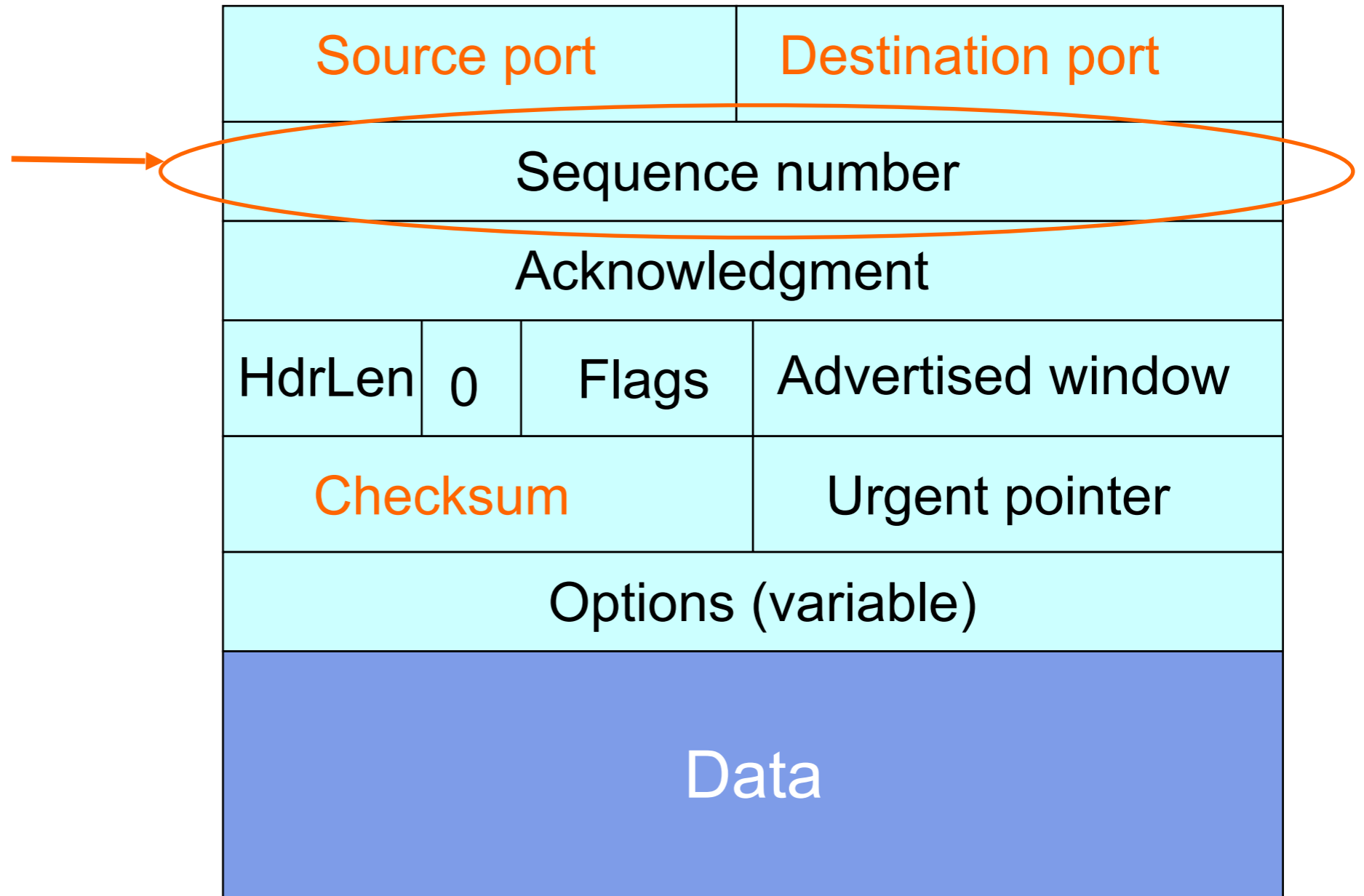
Sender: seqno= $X+2B$ , length= $B$

...

Seqno of next packet is same as last ACK field

# TCP Header

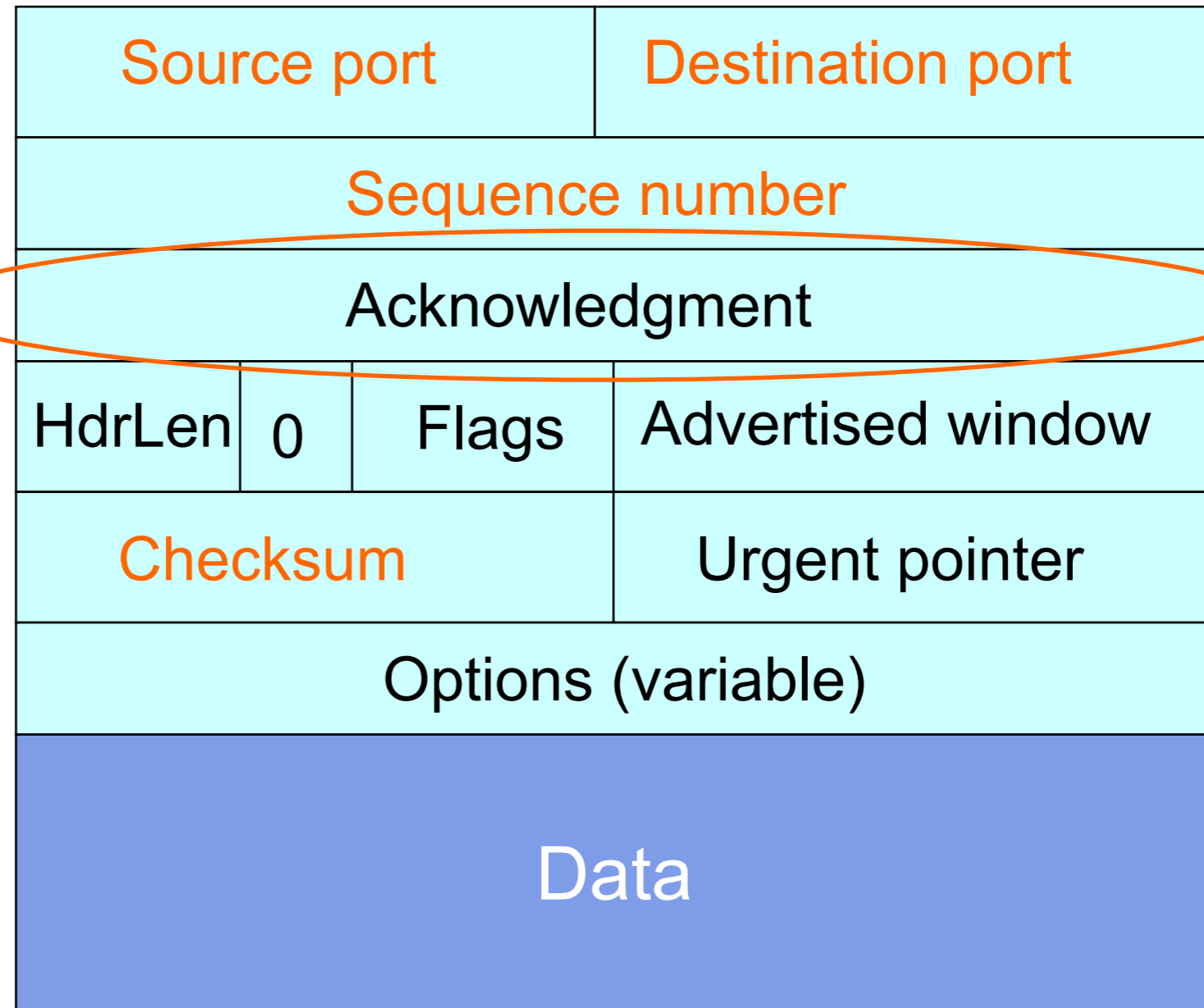
Starting **byte offset of data** carried in this segment



# TCP Header

Acknowledgment  
gives seqno just  
beyond highest  
seqno received  
**in order**

*“What Byte is Next”*



# TCP Header

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

# 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  $B$**

# 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

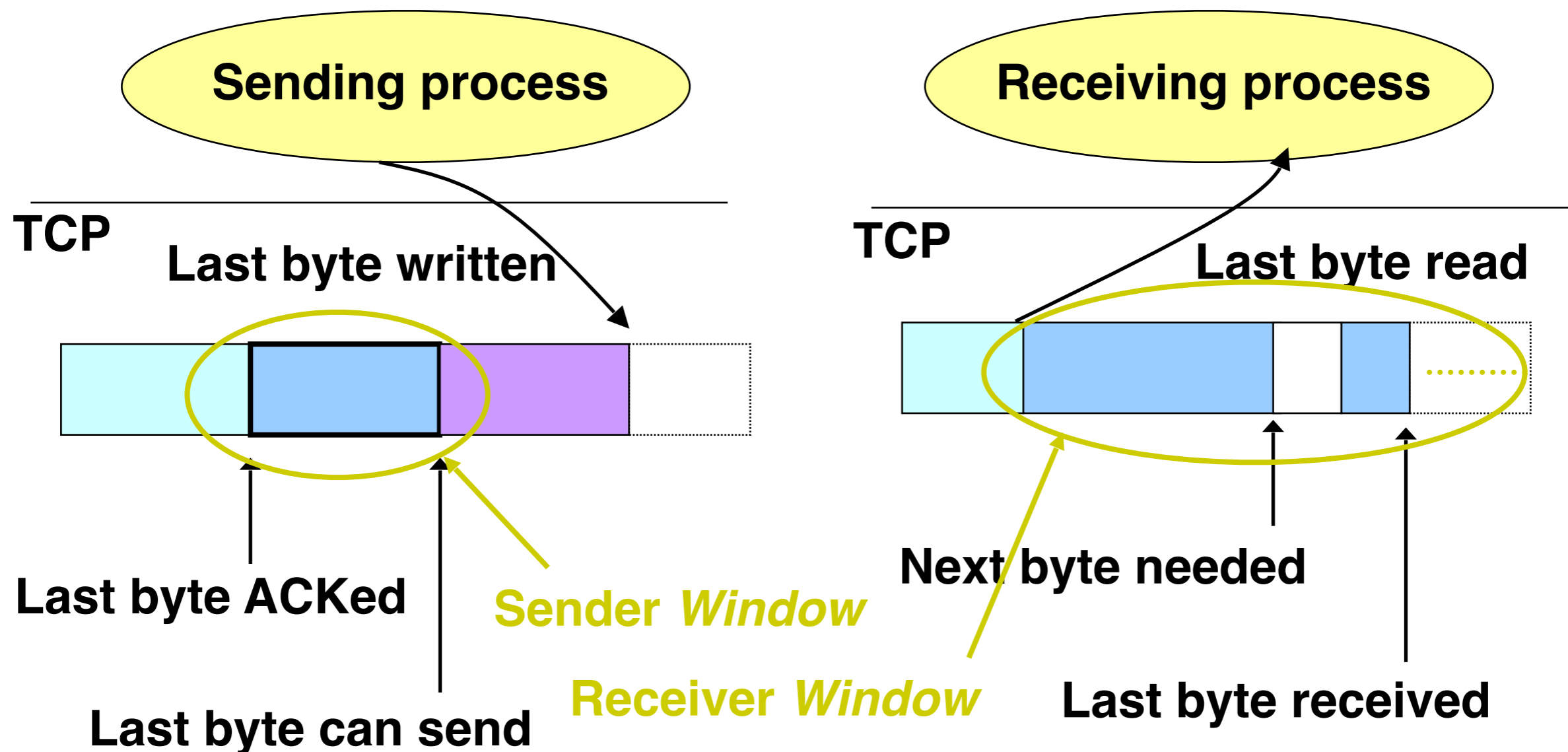
For the receiver:

- Window size = maximum amount of undelivered data

# Sliding Window

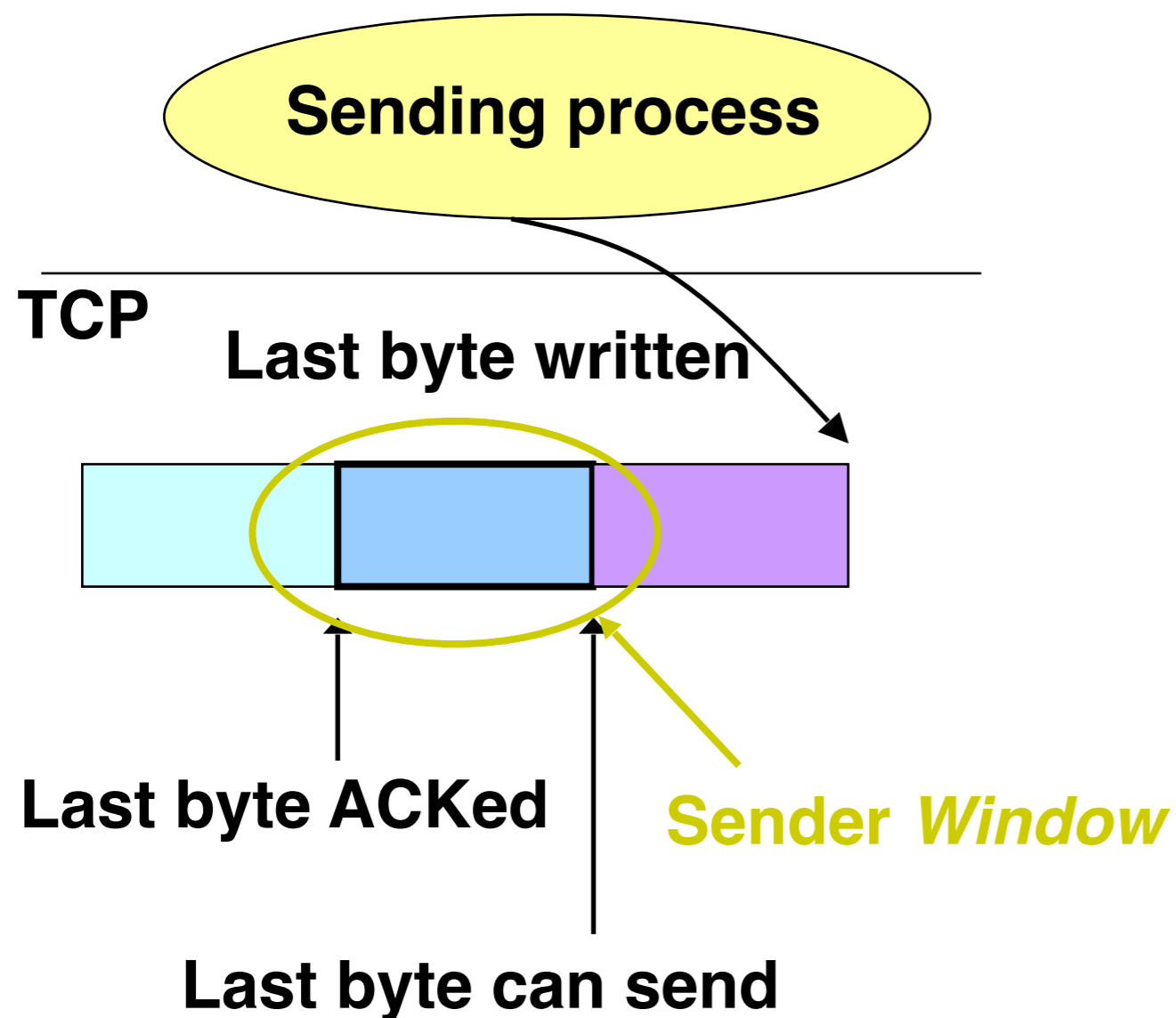
Allow a larger amount of data “in flight”

- Allow sender to **get ahead** of the receiver
- ... though not **too far** ahead



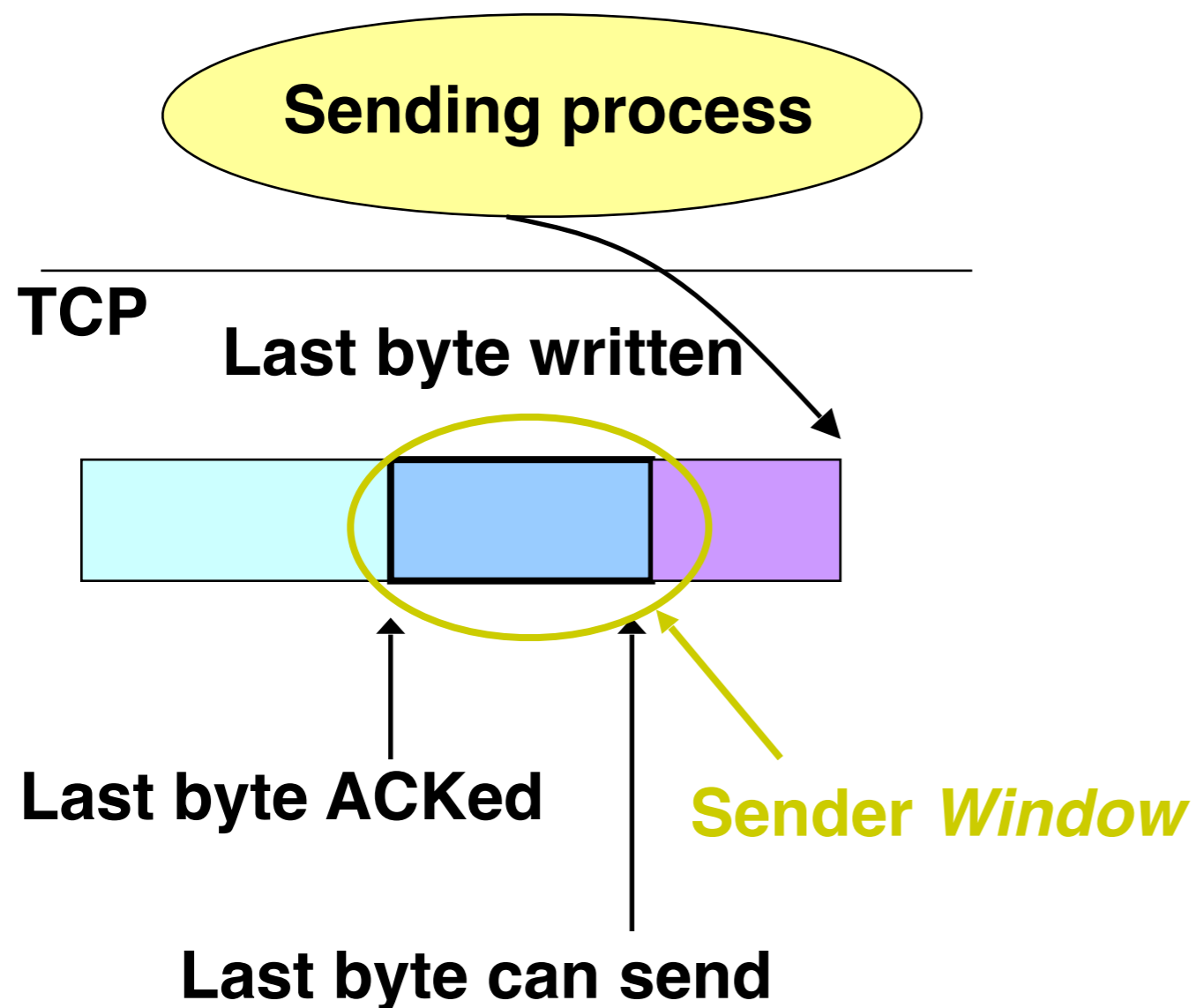
# ***Sliding Window***

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



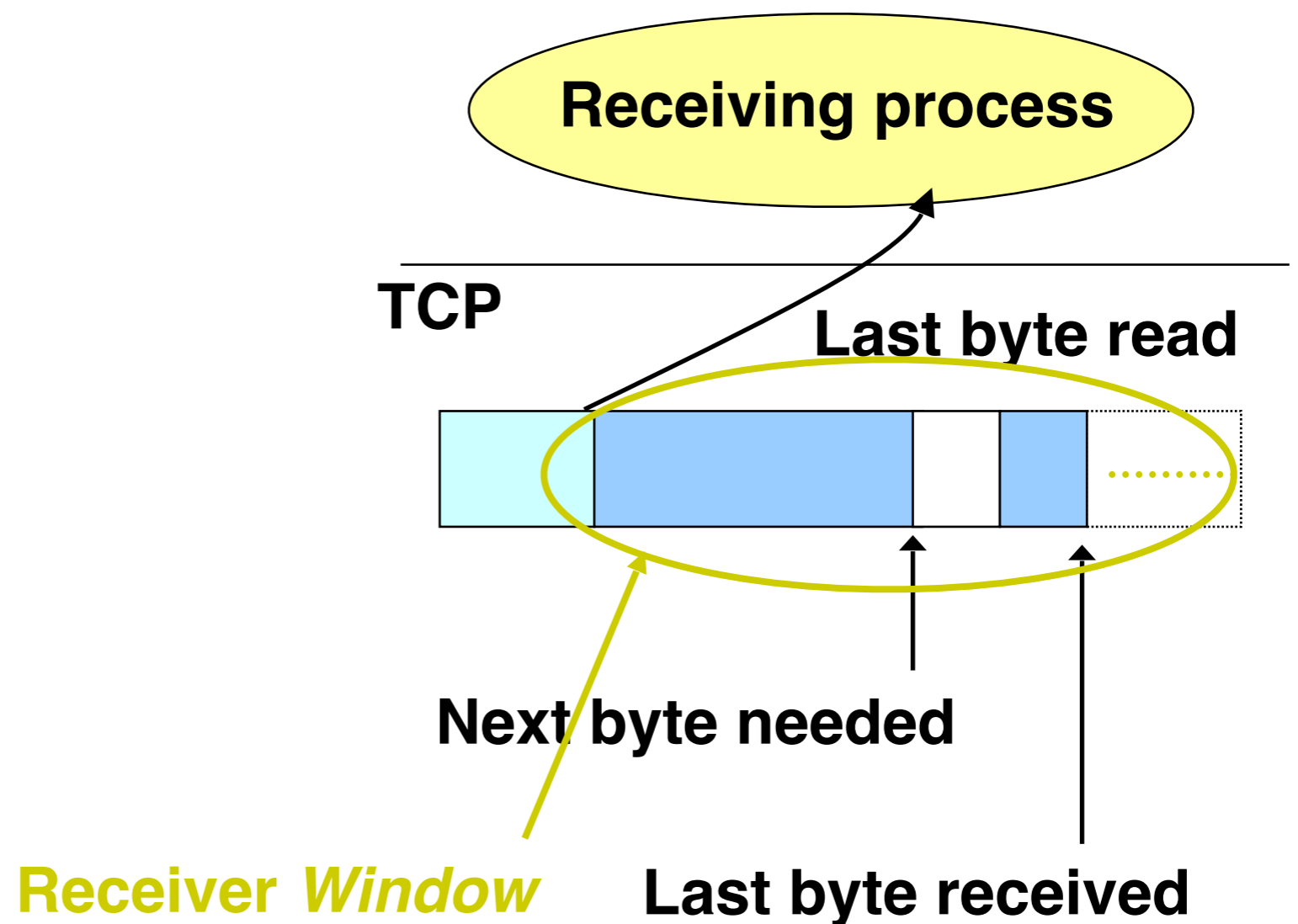
# ***Sliding Window***

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



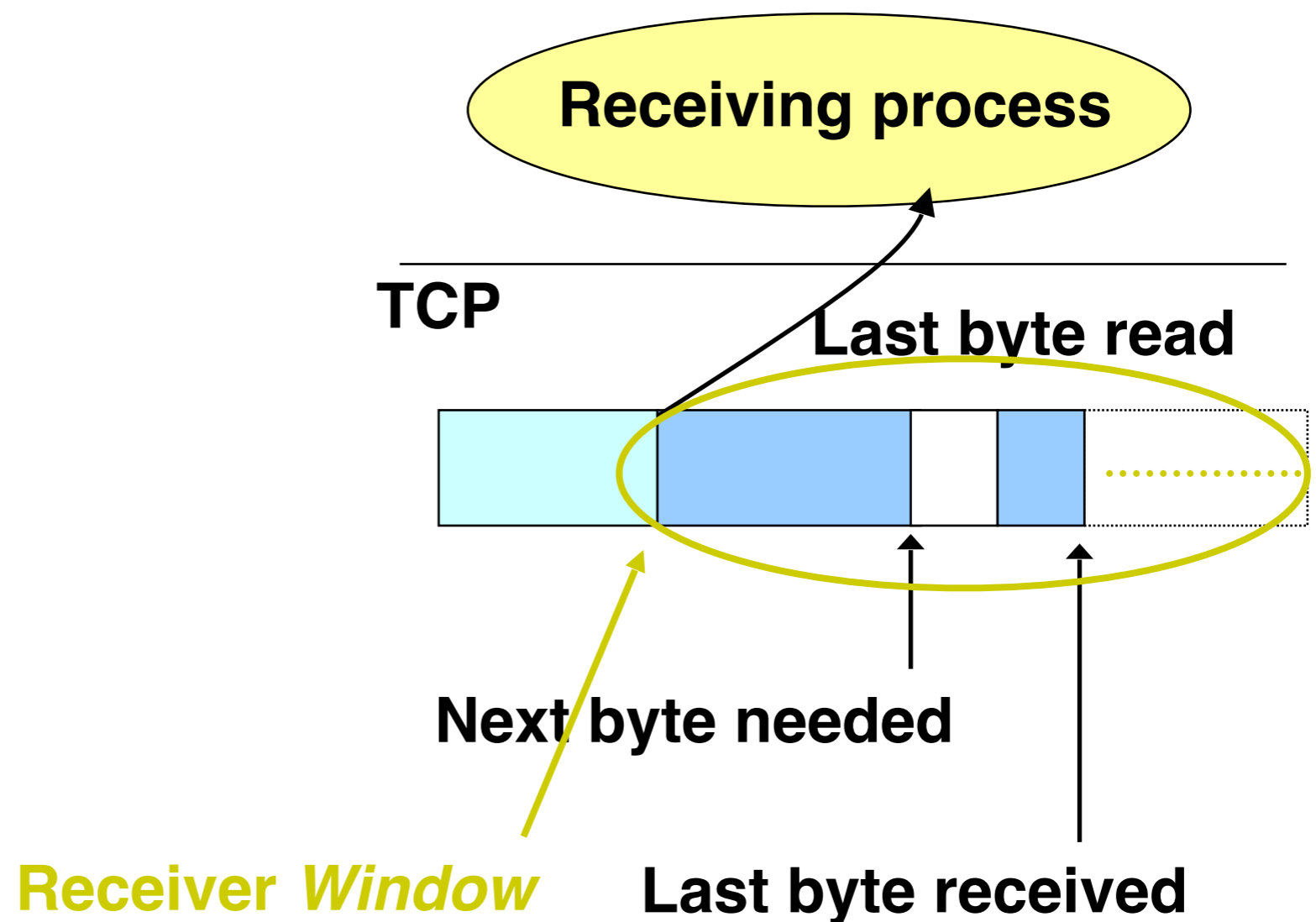
# ***Sliding Window***

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 Summary*

Sender: window **advances** when new data ack'd

Receiver: window advances as receiving process **consumes** data

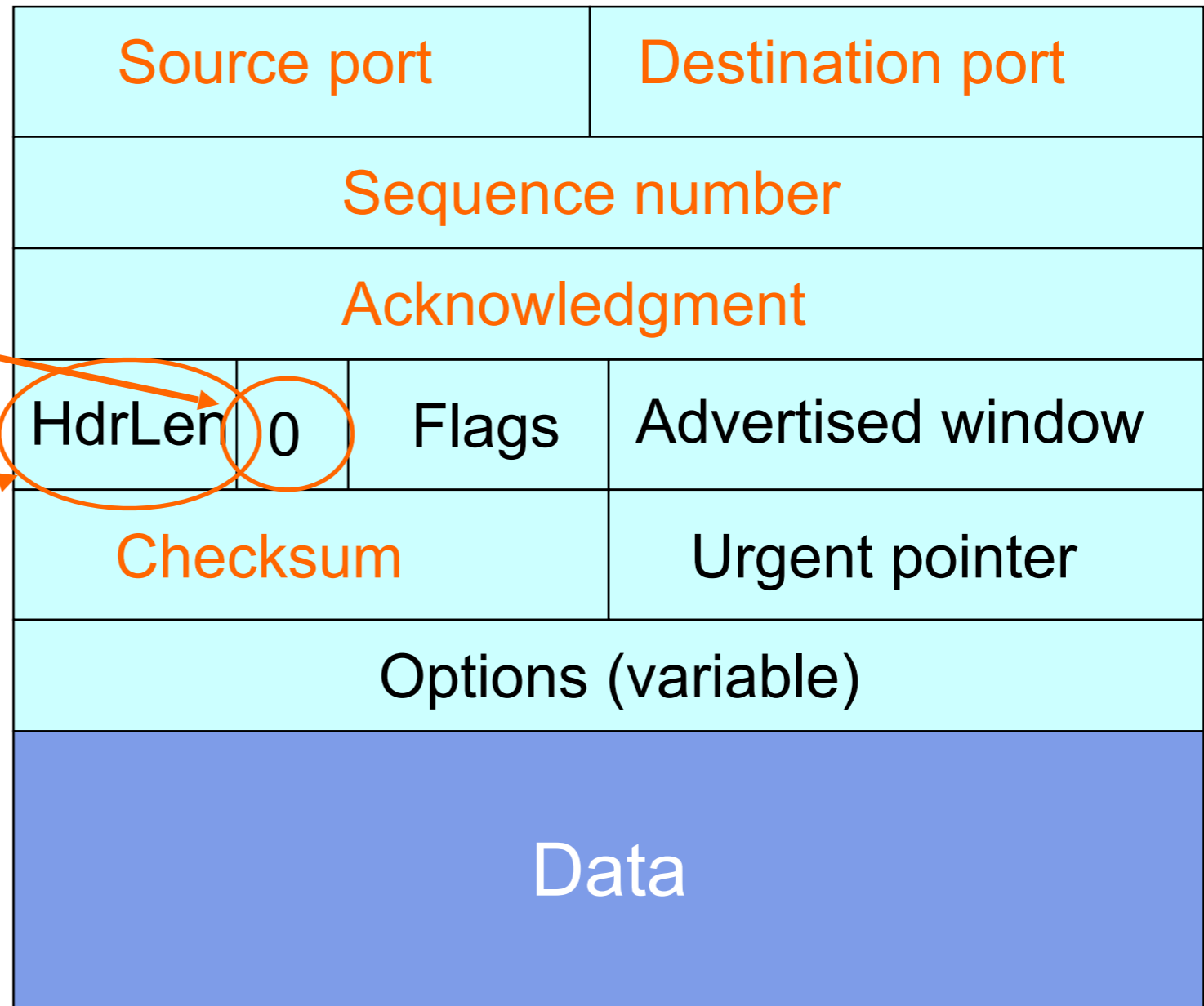
Receiver **advertises** to the sender where the receiver window currently ends (“righthand edge”)

- 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: What's left?

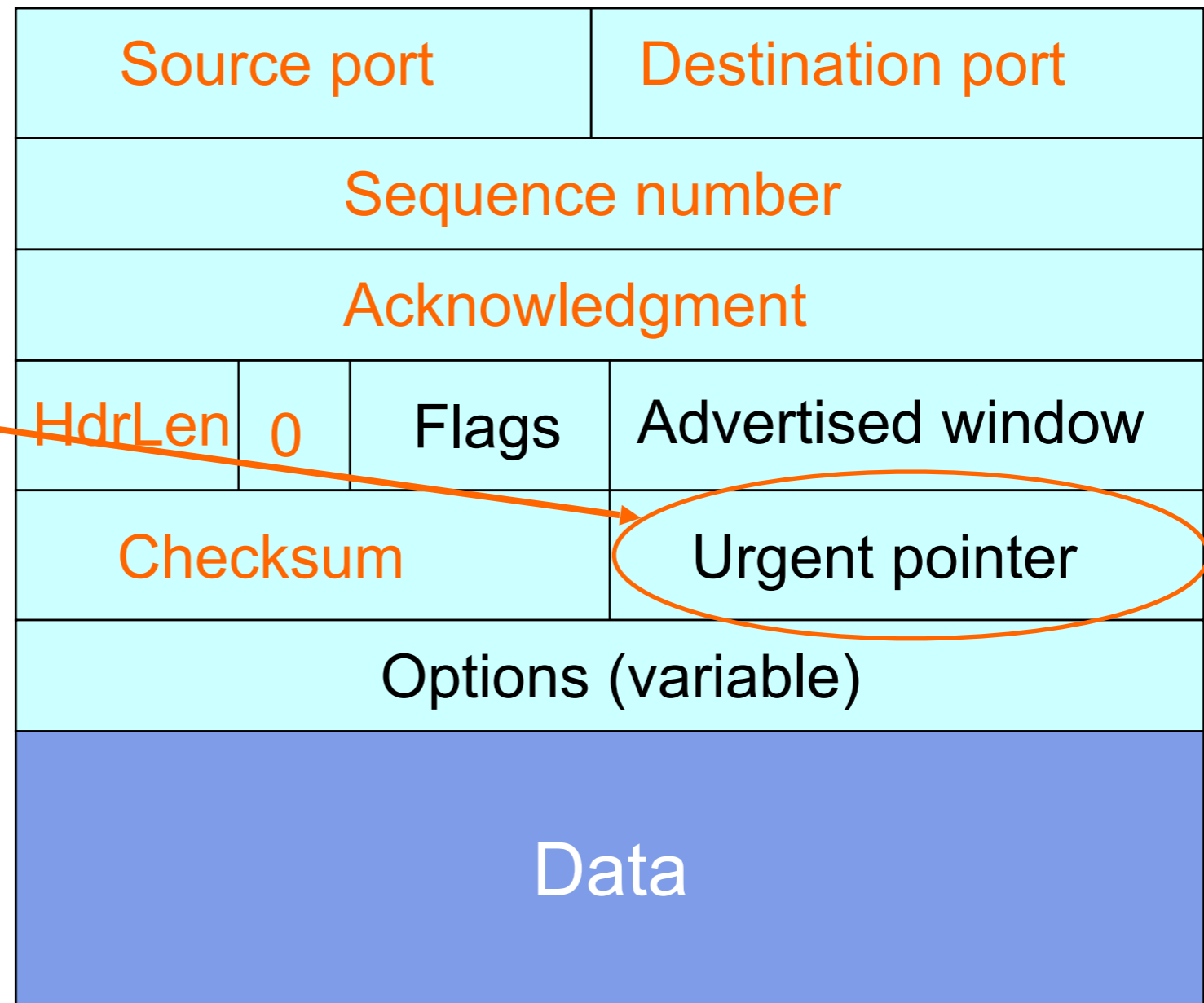
“Must Be Zero”  
6 bits reserved

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



# TCP Header: What's left?

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



# TCP Header: What's left?

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

# **TCP Connection Establishment and Initial 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**

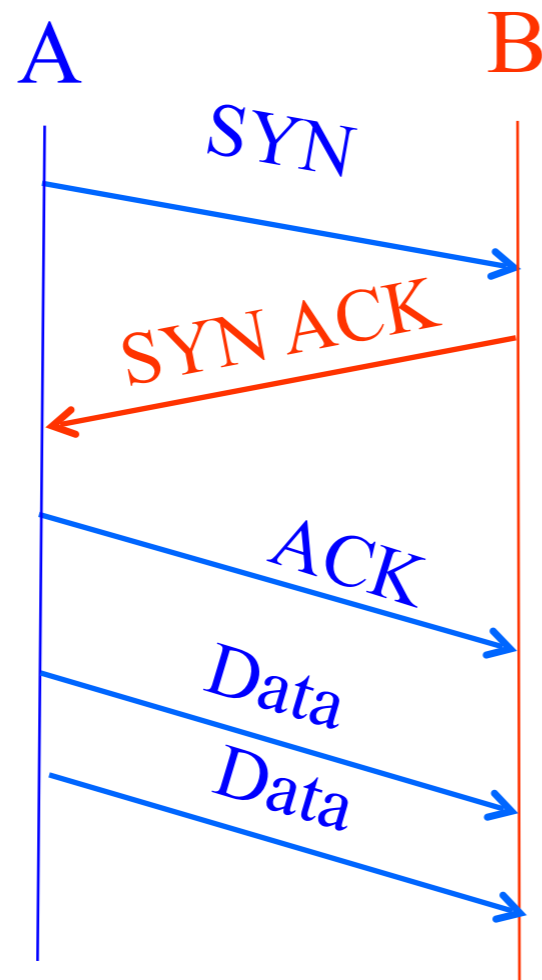
TCP therefore **requires** changing ISN

- initially set from 32-bit clock that ticks every 4 microseconds
- now drawn from a pseudo random number generator (security)

To establish a connection, hosts exchange ISNs

- **How does this help?**

# Establishing a TCP Connection



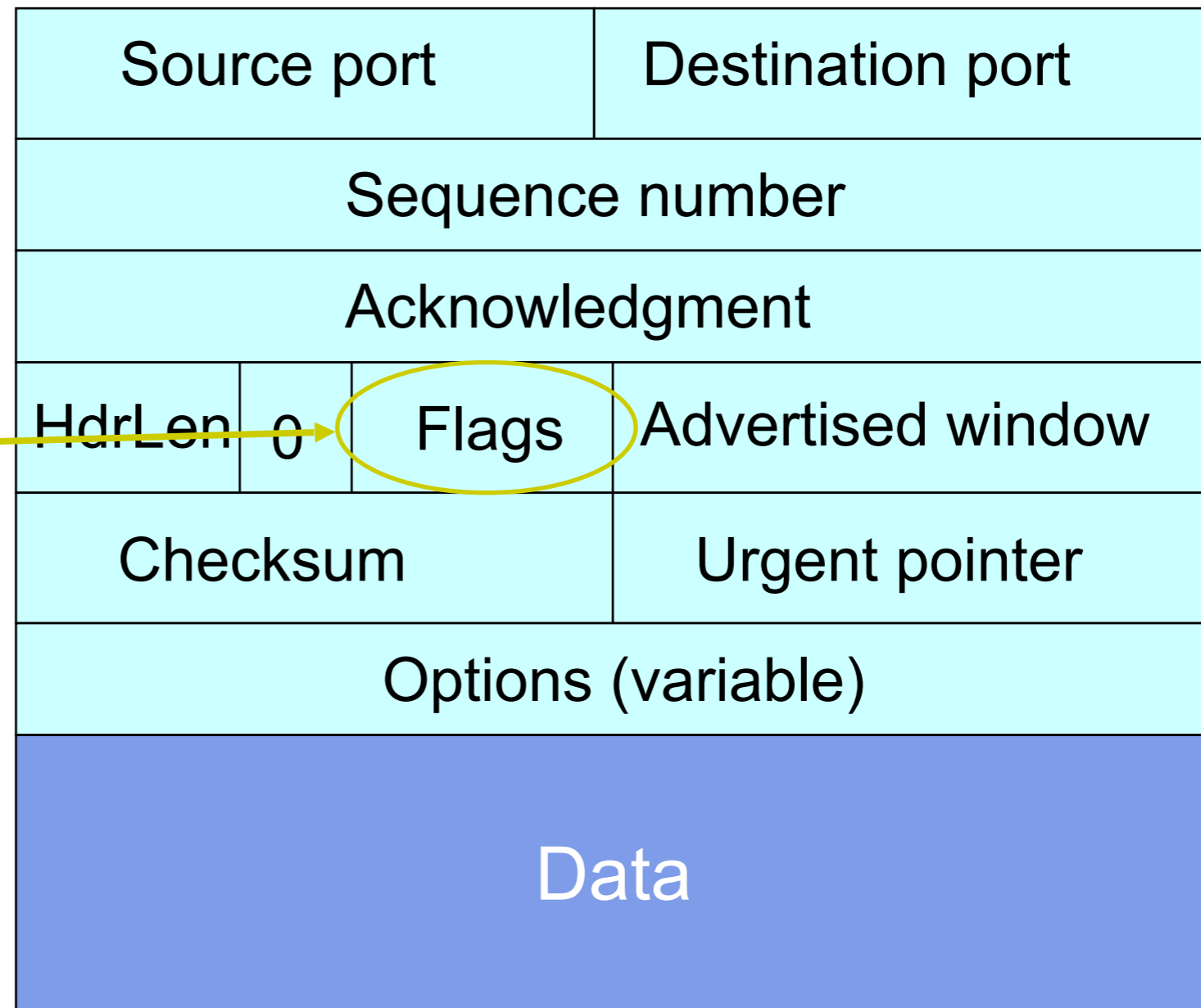
**Each host tells its ISN to the other host.**

Three-way handshake to establish connection

- Host A sends a **SYN** (open; “synchronize sequence numbers”)
- Host B returns a SYN acknowledgment (**SYN ACK**)
- Host A sends an **ACK** to acknowledge the SYN ACK

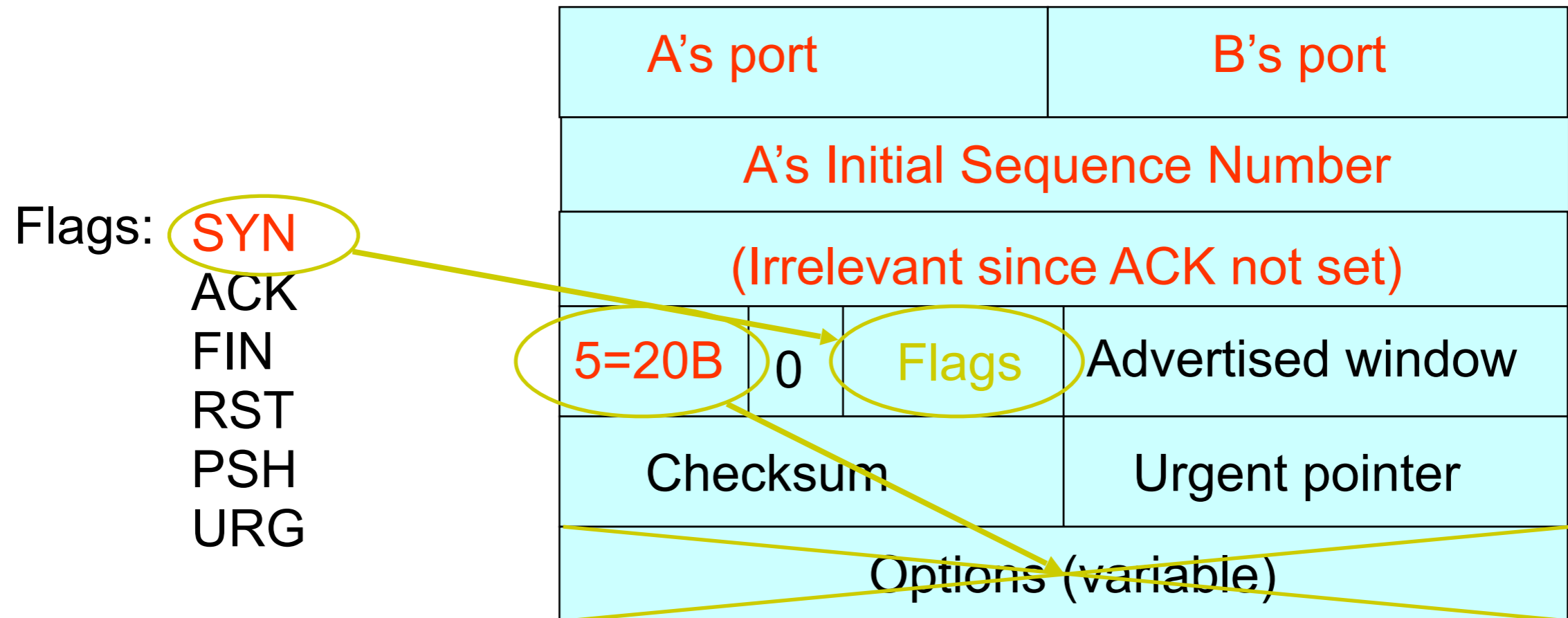
# TCP Header

Flags: SYN  
ACK  
FIN  
RST  
PSH  
URG



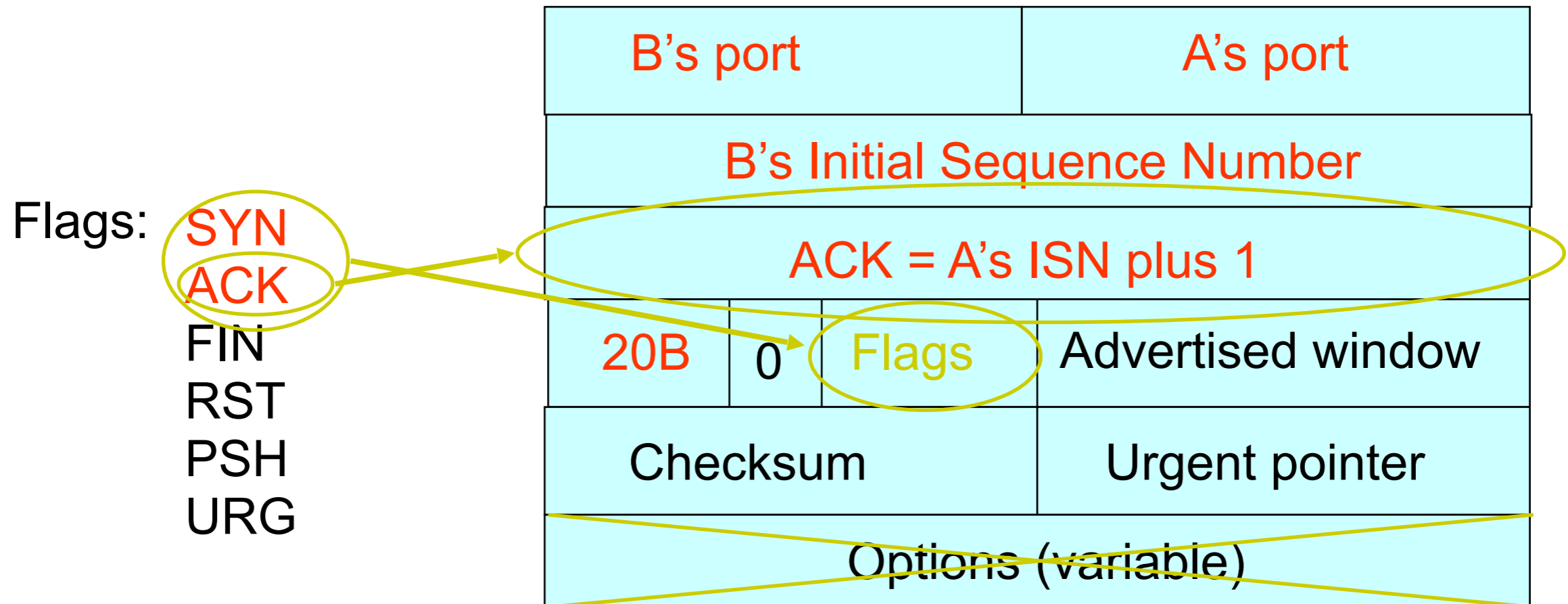
See `/usr/include/netinet/tcp.h` on Unix Systems

# Step 1: A's Initial SYN Packet



**A tells B it wants to open a connection...**

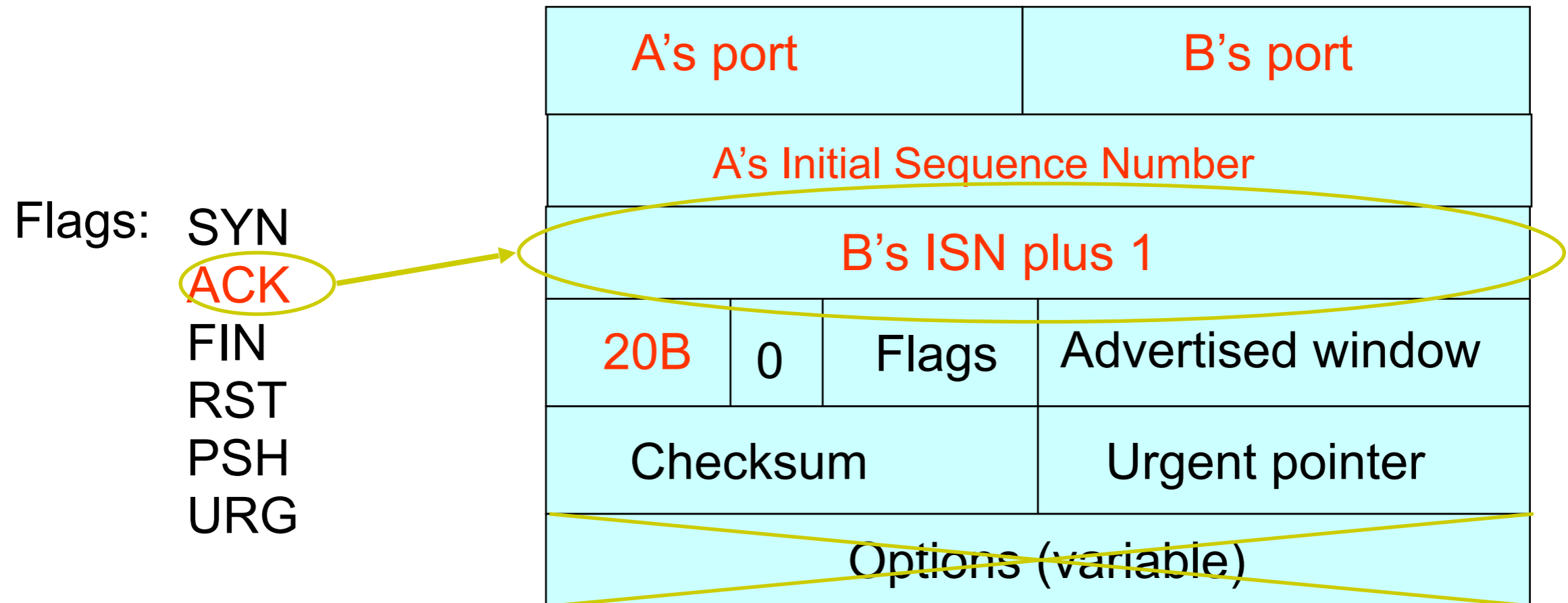
## Step 2: B's SYN-ACK Packet



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

**... upon receiving this packet, A can start sending data**

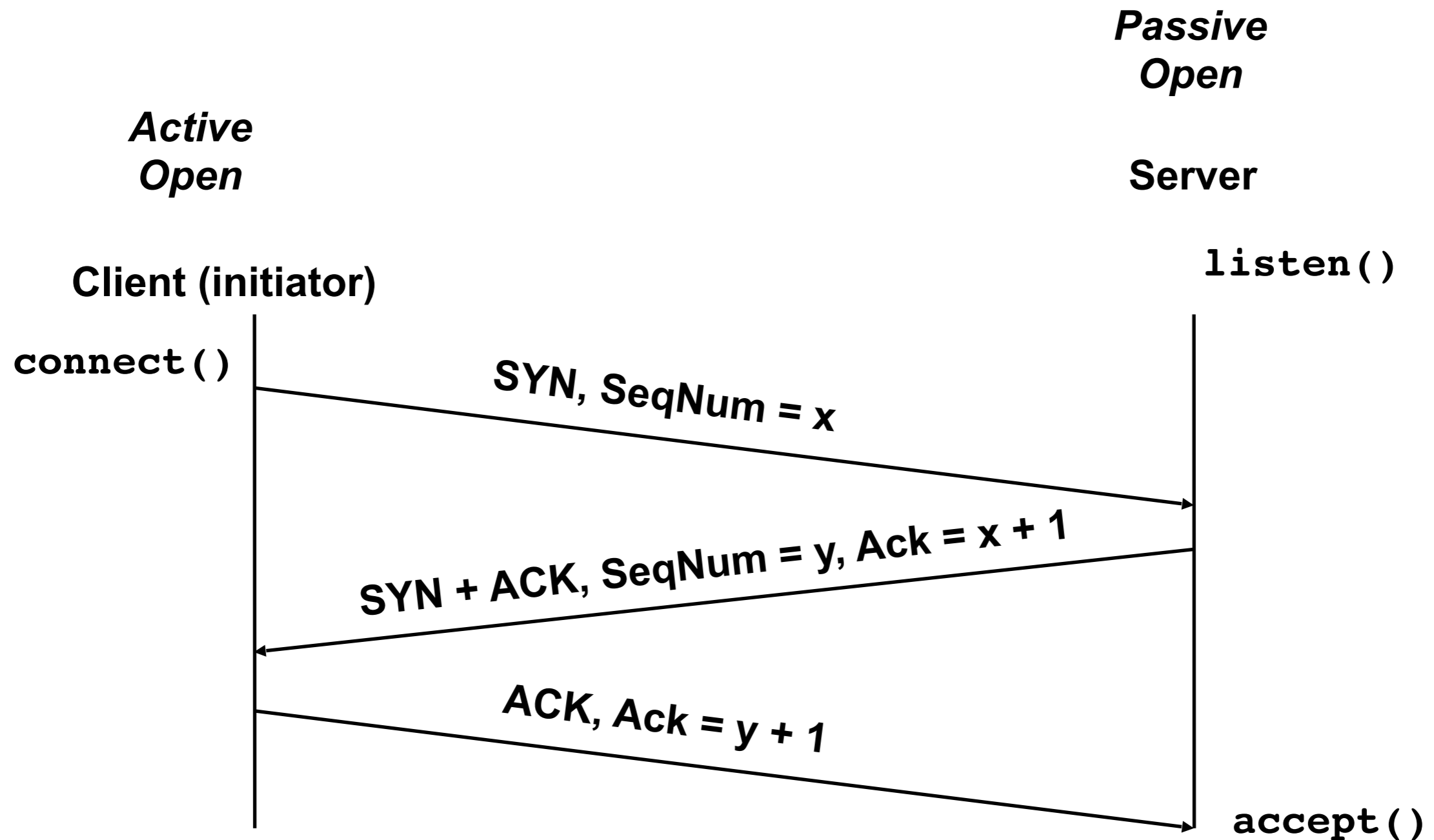
# Step 3: A's ACK of the SYN-ACK



**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...

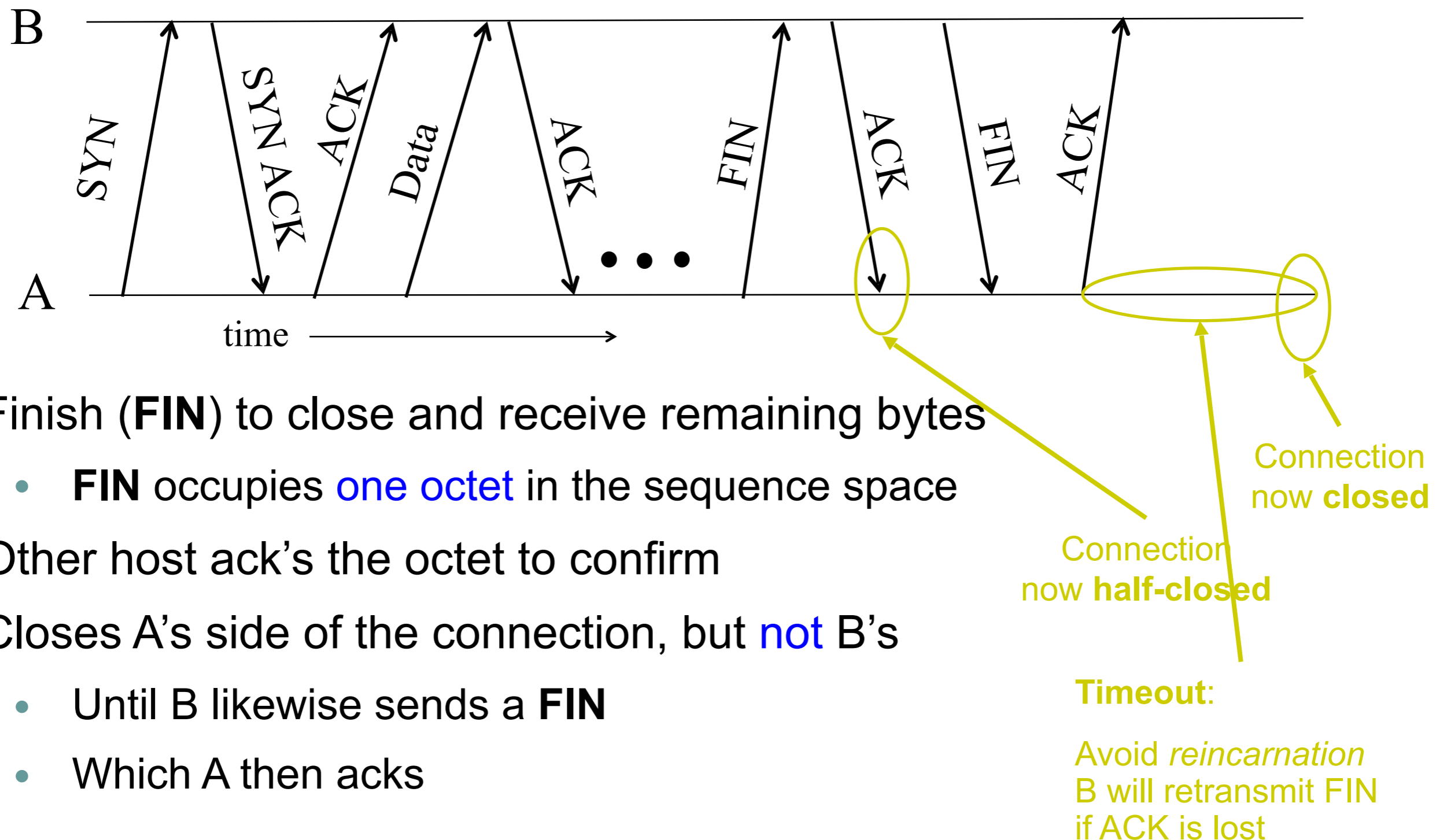
- 3-6 seconds of delay: can be **very long**
- User may become impatient
- ... and click the hyperlink again, or click “reload”

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

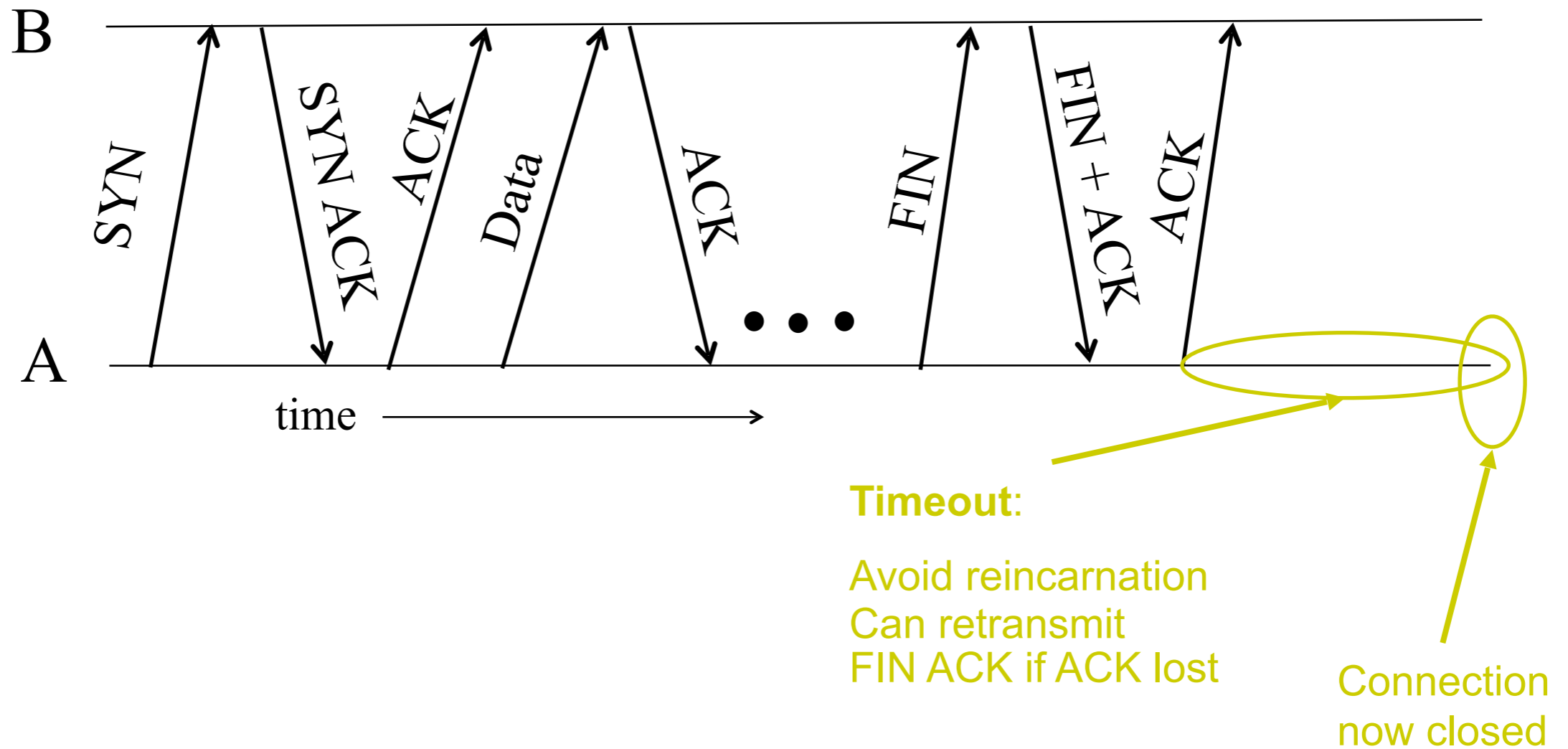
# Tearing Down the Connection

# Normal Termination, One Side At A Time

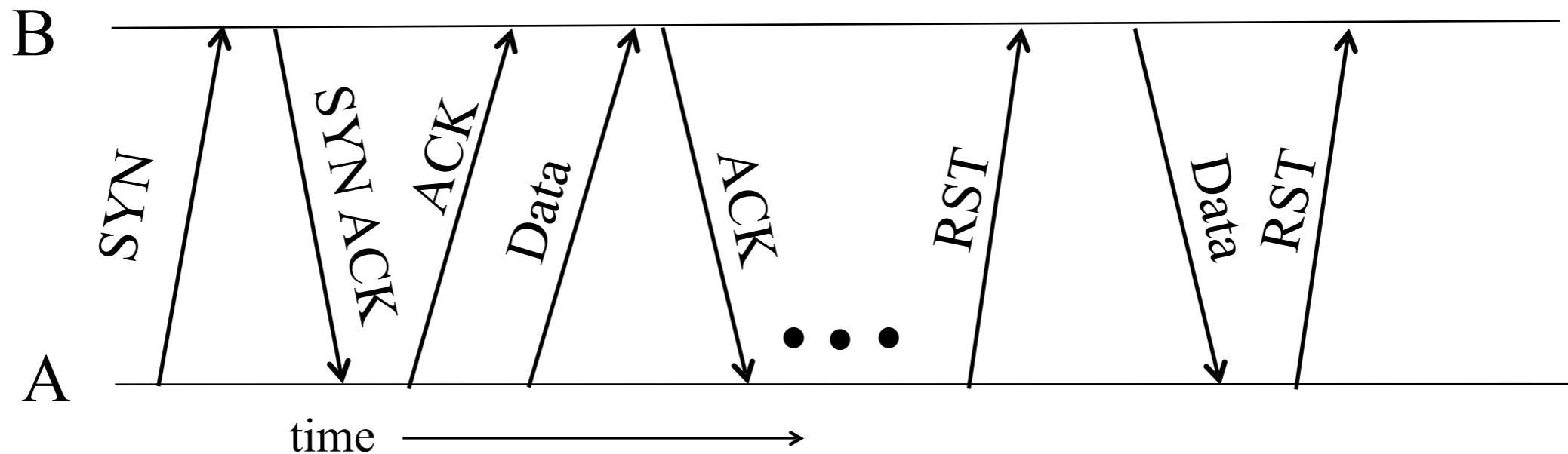


# Normal Termination, Both Together

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



# Abrupt Termination



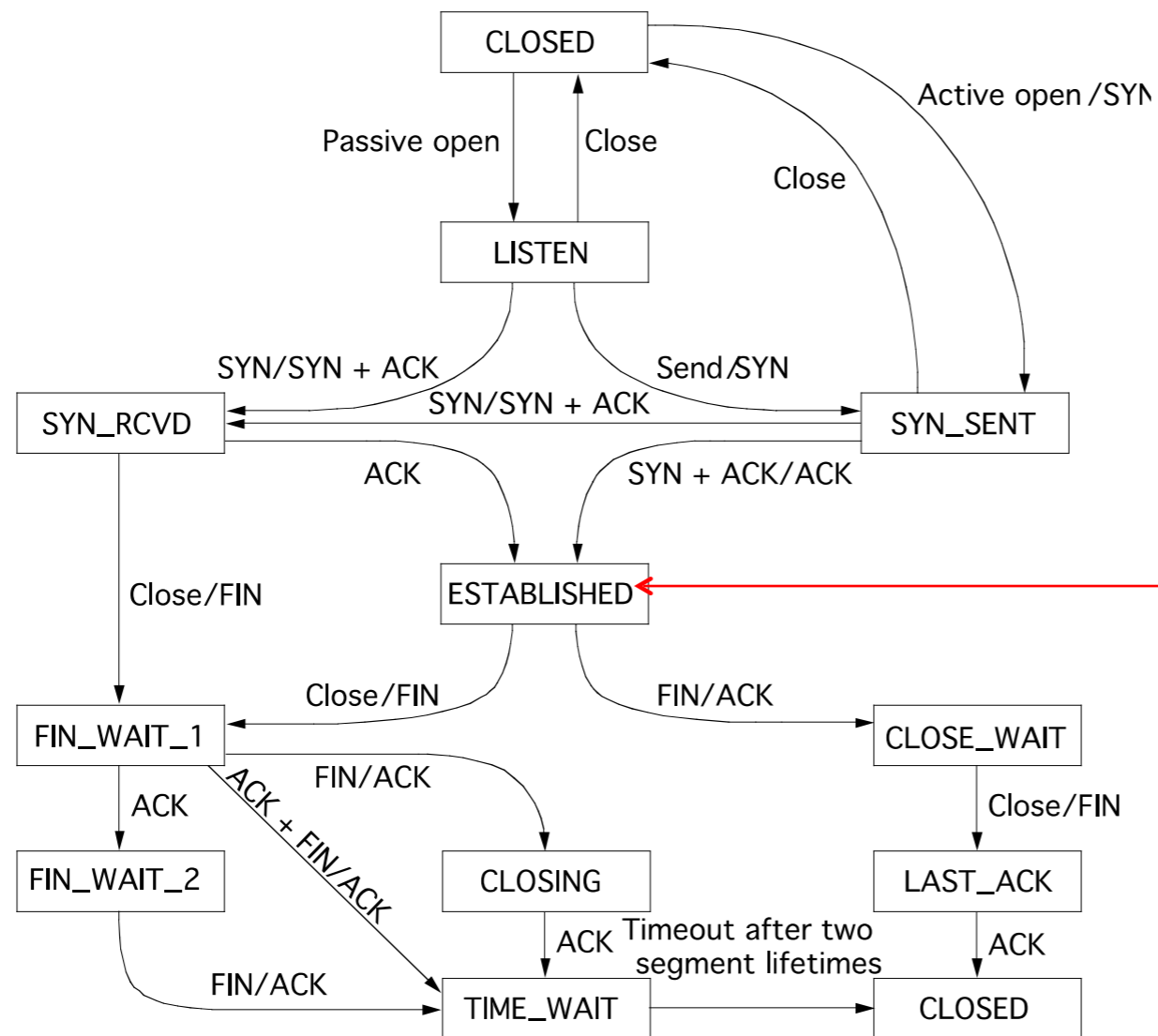
A sends a RESET (**RST**) to B

- 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**

# TCP State Transitions



Data, ACK exchanges are in here

# Reliability: TCP Retransmission

# Timeouts and Retransmissions

Reliability requires retransmitting lost data

Involves setting timer and retransmitting on timeout

TCP resets timer whenever new data is ACKed

- Retx of packet containing “next byte” when timer goes off

# Example

Arriving ACK expects 100

Sender sends packets 100, 200, 300, 400, 500

- Timer set for 100

Arriving ACK expects 300

- Timer set for 300

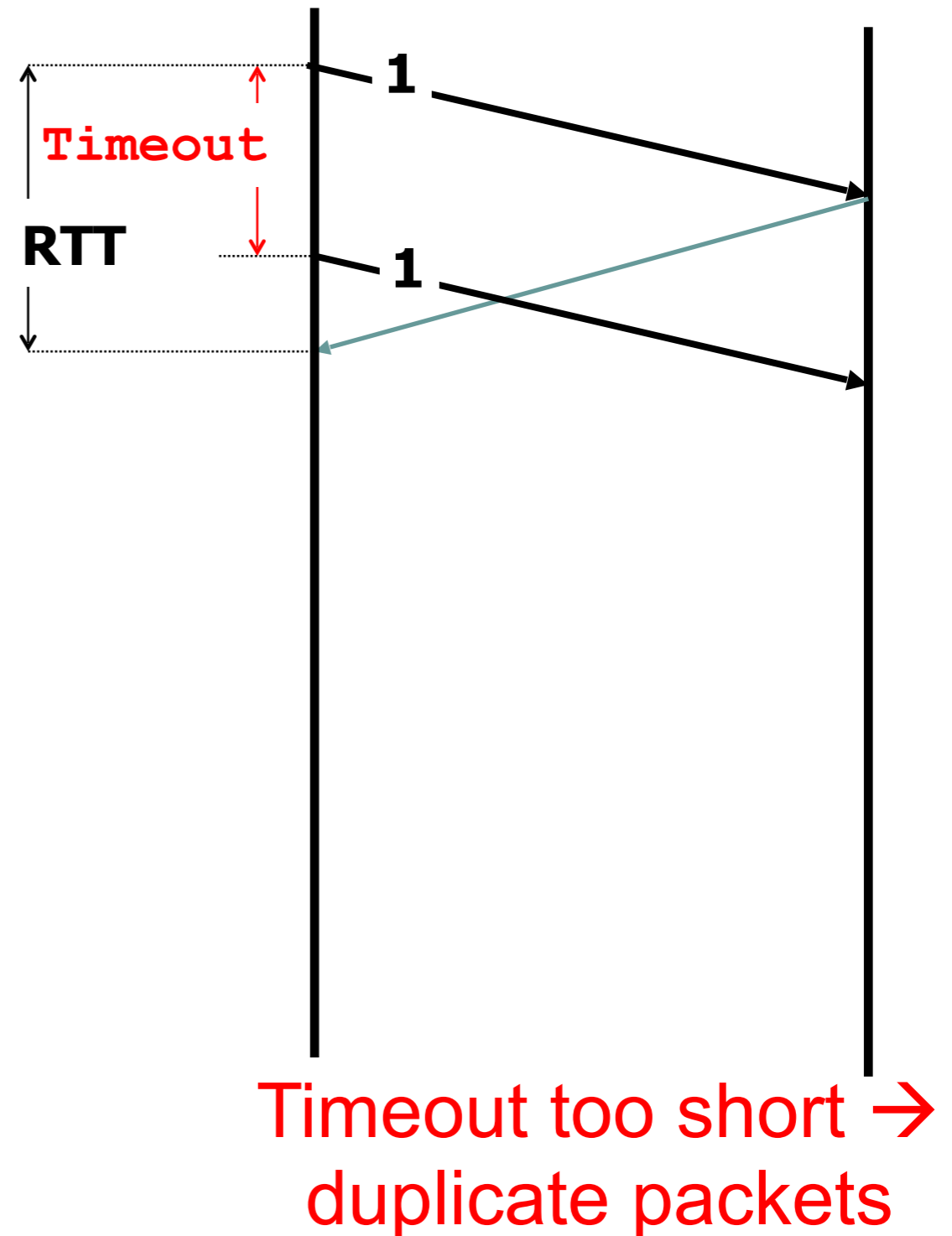
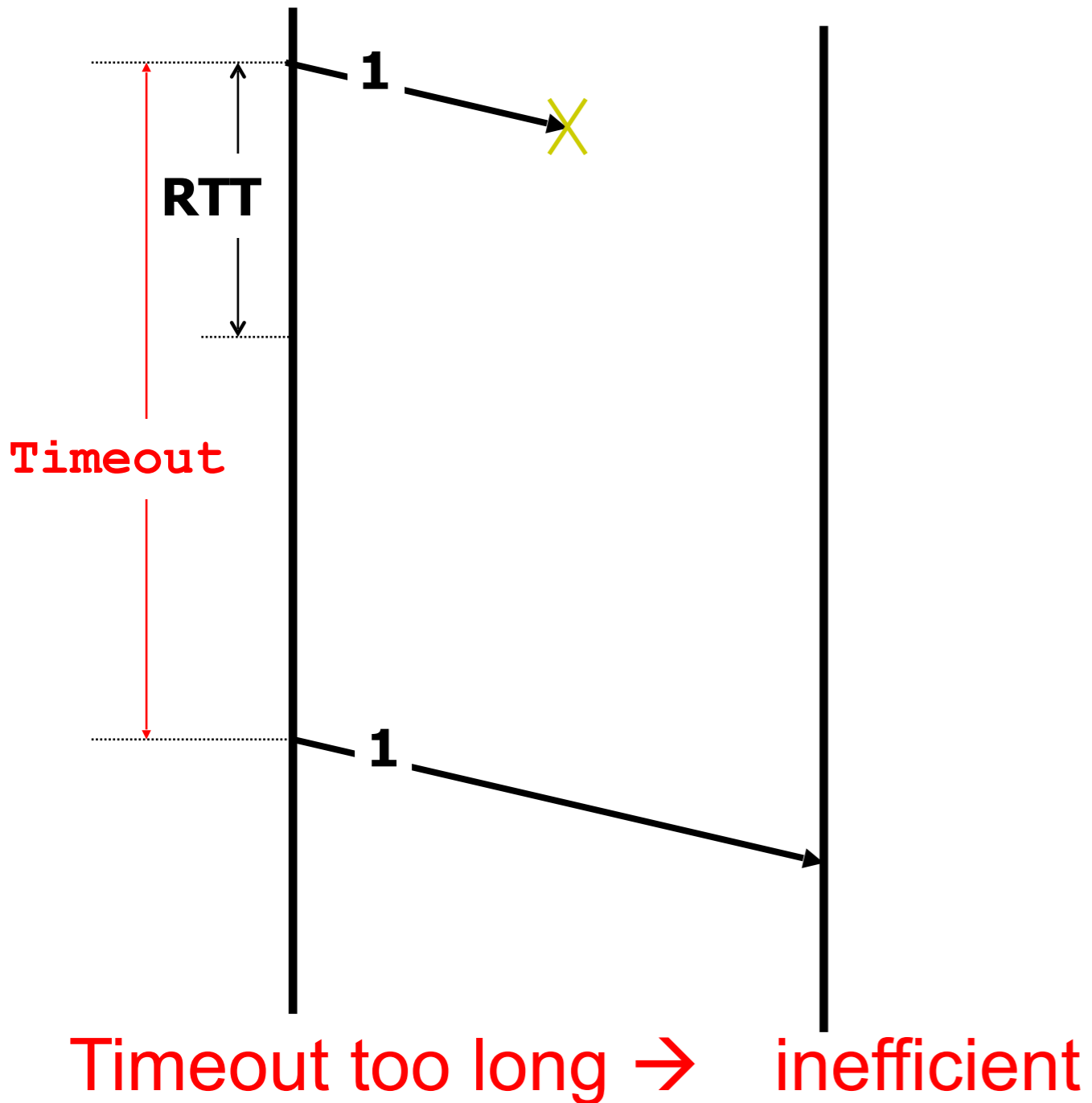
Timer goes off

- Packet 300 is resent

Arriving ACK expects 600

- Packet 600 sent
- Timer set for 600

# Setting the Timeout Value



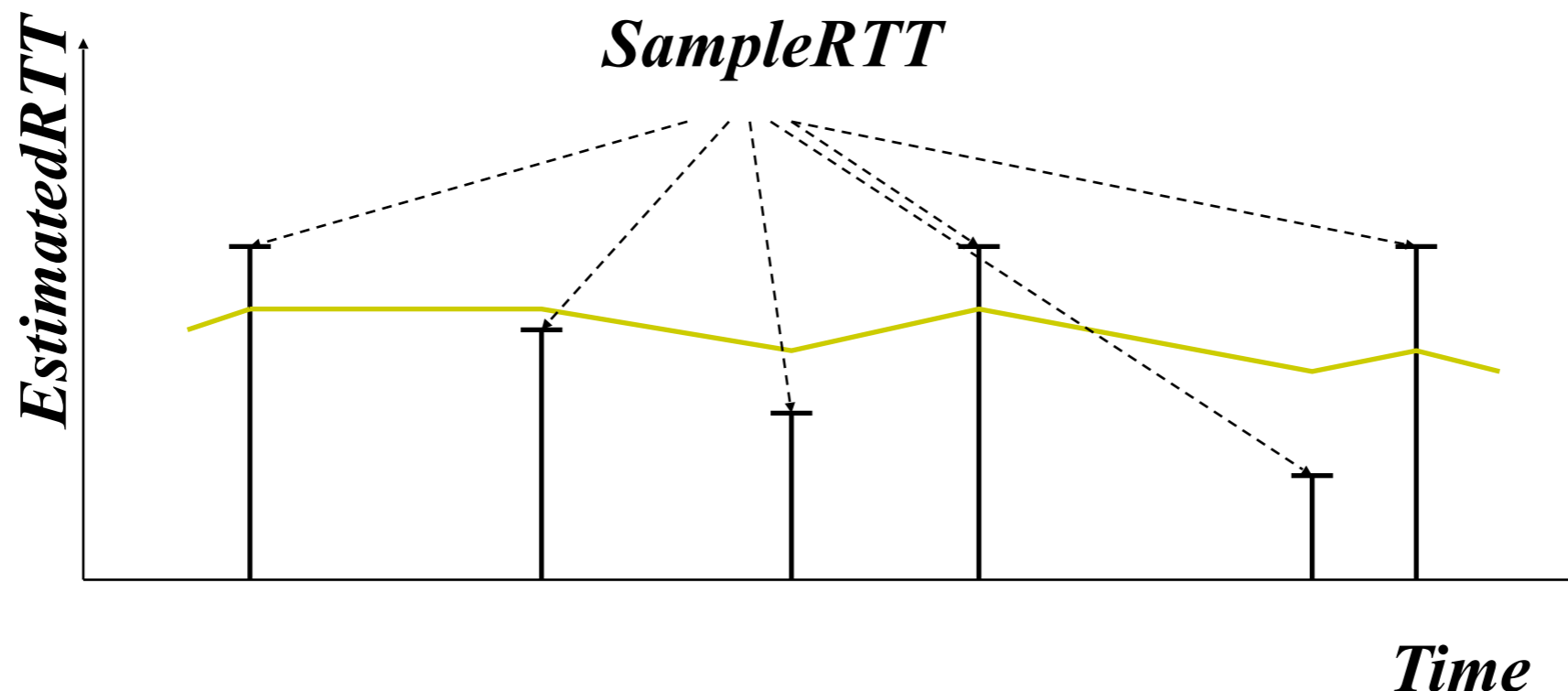
# RTT Estimation

Use exponential averaging of RTT samples

$$\text{SampleRTT} = \text{AckRcvdTime} - \text{SendPacketTime}$$

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

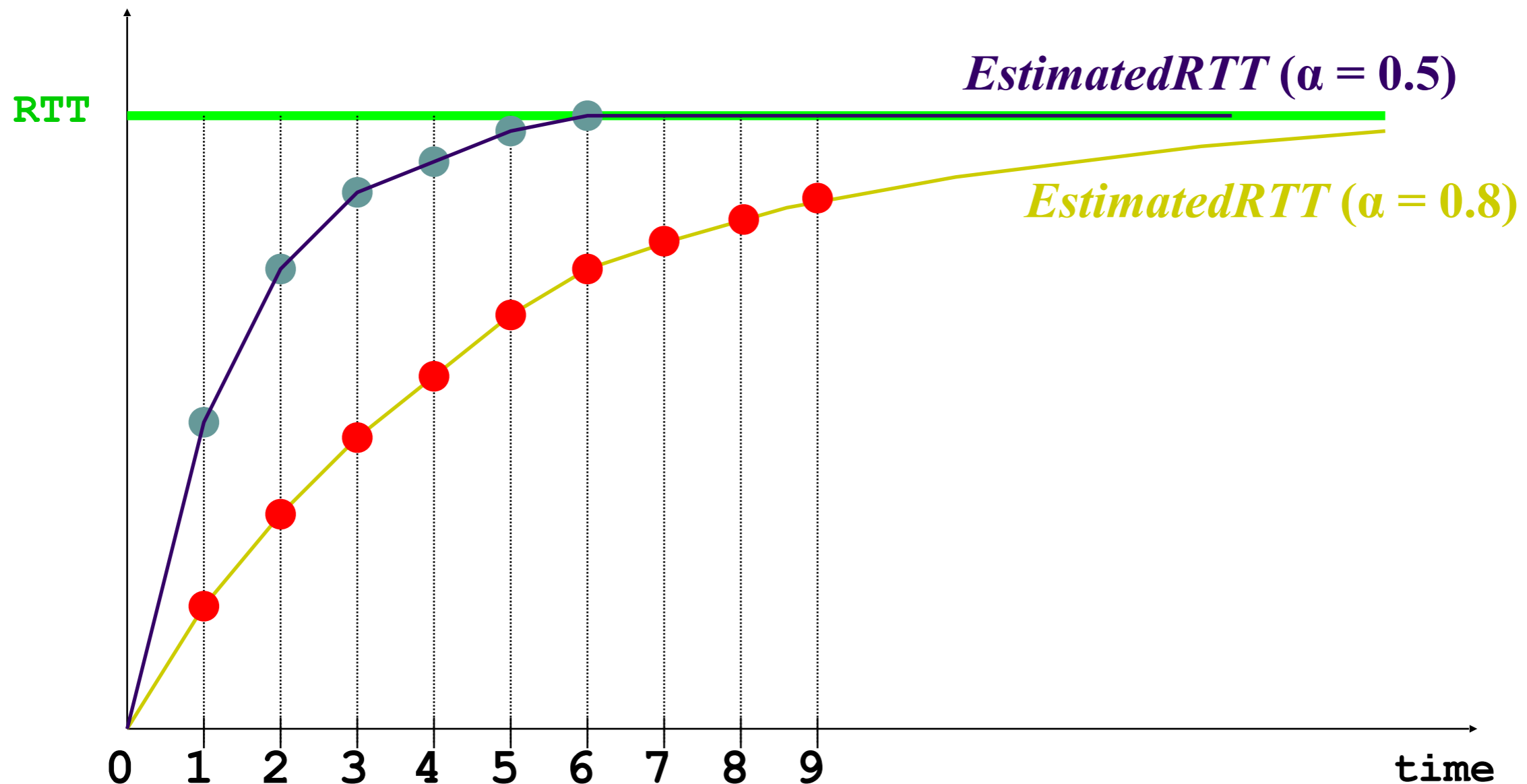
$$0 < \alpha \leq 1$$



# Exponential Averaging Example

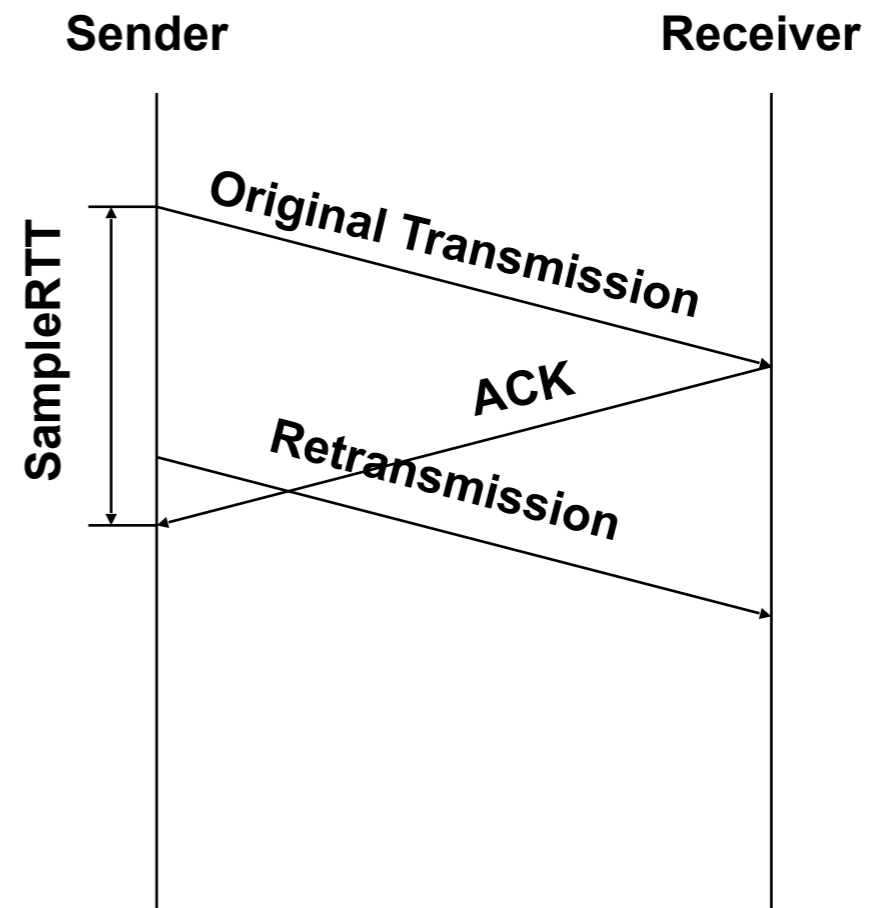
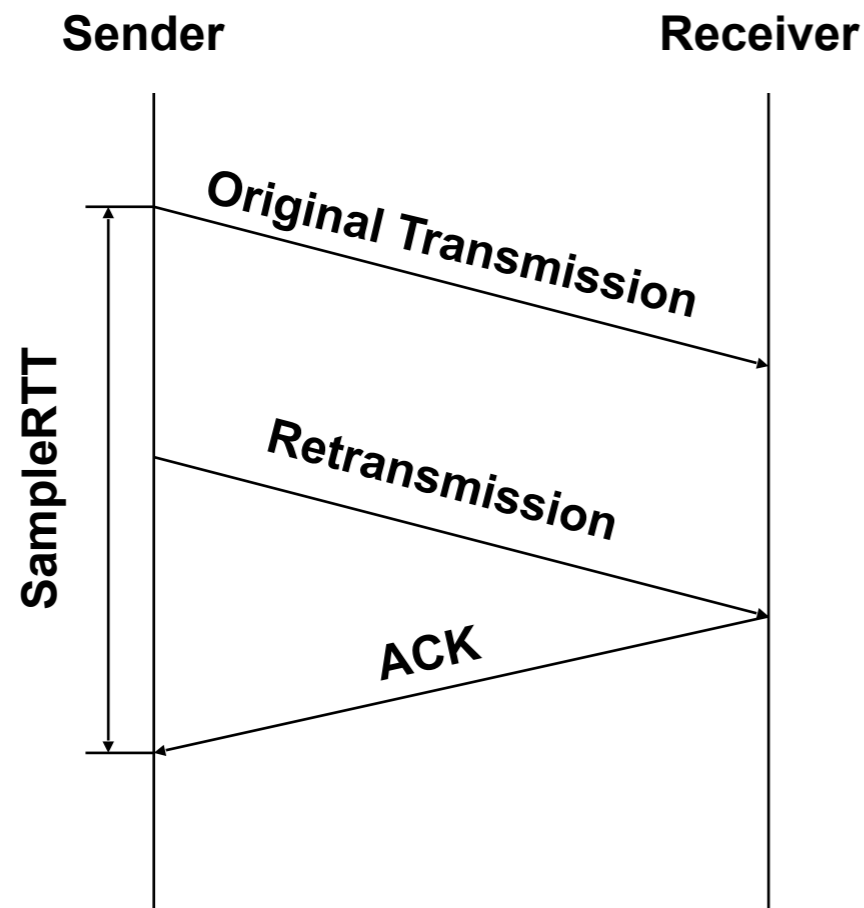
$$\text{EstimatedRTT} = \alpha * \text{EstimatedRTT} + (1 - \alpha) * \text{SampleRTT}$$

Assume RTT is constant  $\rightarrow$   $\text{SampleRTT} = \text{RTT}$



# Problem: Ambiguous Measurements

How do we 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
- Computes *EstimatedRTT* using  $\alpha = 0.875$

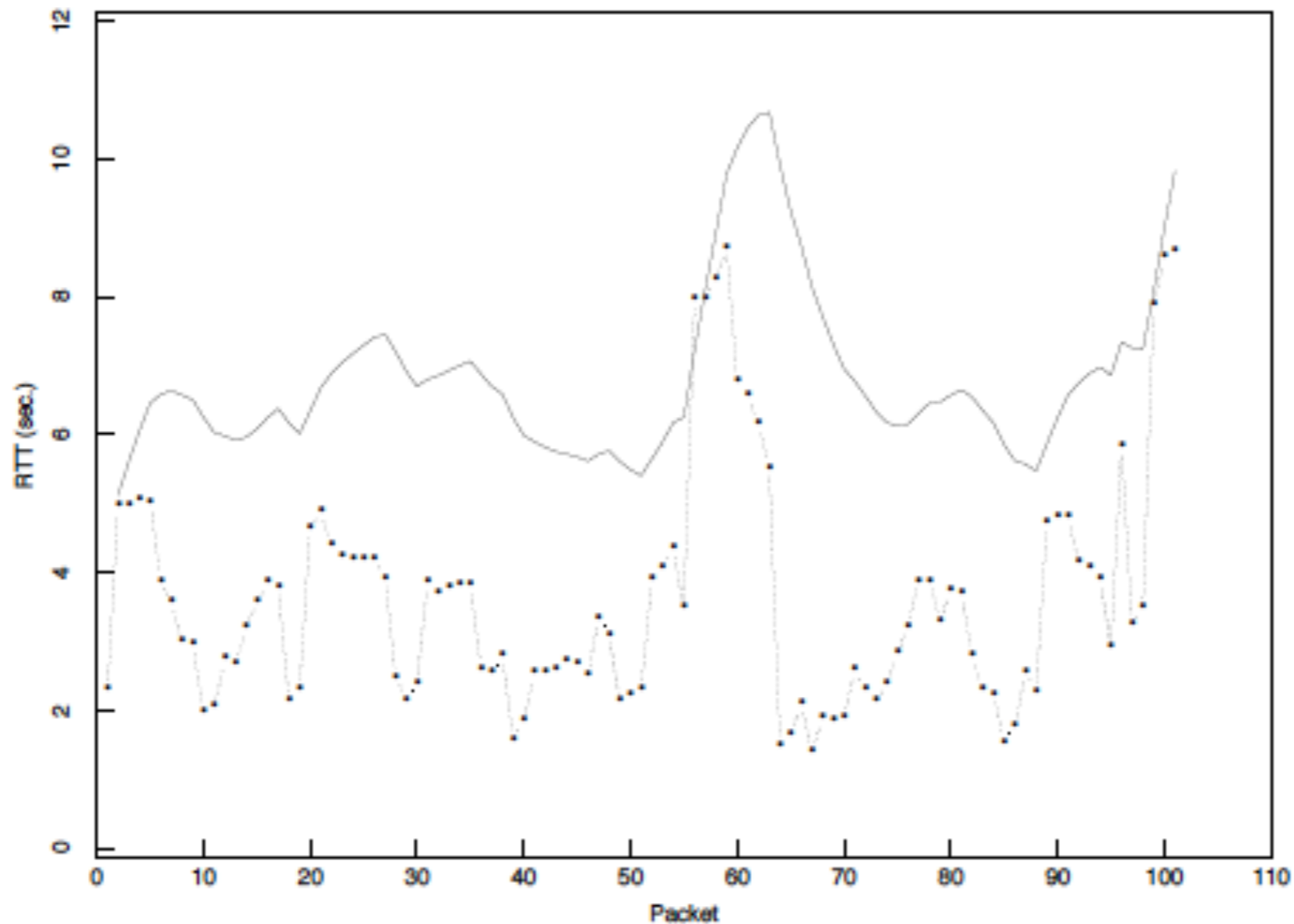
Timeout value (RTO) =  $2 \times \text{EstimatedRTT}$

Use exponential backoff for repeated retransmissions

- Every time RTO timer expires, set  $\text{RTO} \leftarrow 2 \cdot \text{RTO}$ 
  - (Up to maximum  $\geq 60$  sec)
- Every time new measurement comes in (= successful original transmission), collapse RTO back to  $2 \times \text{EstimatedRTT}$

# Karn/Partridge in action

Figure 5: Performance of an RFC793 retransmit timer



from Jacobson and Karels, SIGCOMM 1988

# 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

# Loss with cumulative ACKs

Sender sends packets with 100B and seqnos.:

- 100, 200, 300, 400, 500, 600, 700, 800, 900, ...

Assume the fifth packet (seqno 500) is lost, but no others

Stream of ACKs will be:

- 200, 300, 400, 500, 500, 500, 500, ...

# Loss with cumulative ACKs

“Duplicate ACKs” are a sign of an *isolated* loss

- The lack of ACK progress means 500 hasn't been delivered
- Stream of ACKs means some packets are being delivered

Therefore, could trigger resend upon receiving  $k$  duplicate ACKs

- TCP uses  $k=3$

We will revisit this in congestion control

# Communication Networks

Spring 2017



Laurent Vanbever

[www.vanbever.eu](http://www.vanbever.eu)

ETH Zürich (D-ITET)

May, 8 2017