

# CS/ECE 4457

## Computer Networks: Architecture and Protocols

### Lecture 24 TCP congestion control II

**Qizhe Cai**



# Recap: Four Goals for Reliable Transfer

- **Correctness**

- It resends all dropped or corrupted packets
- It attempts to make progress

- **“Fairness”**

- Every flow must get a fair share of network resources

- **Flow Performance**

- Latency, jitter, etc.

- **Utilization**

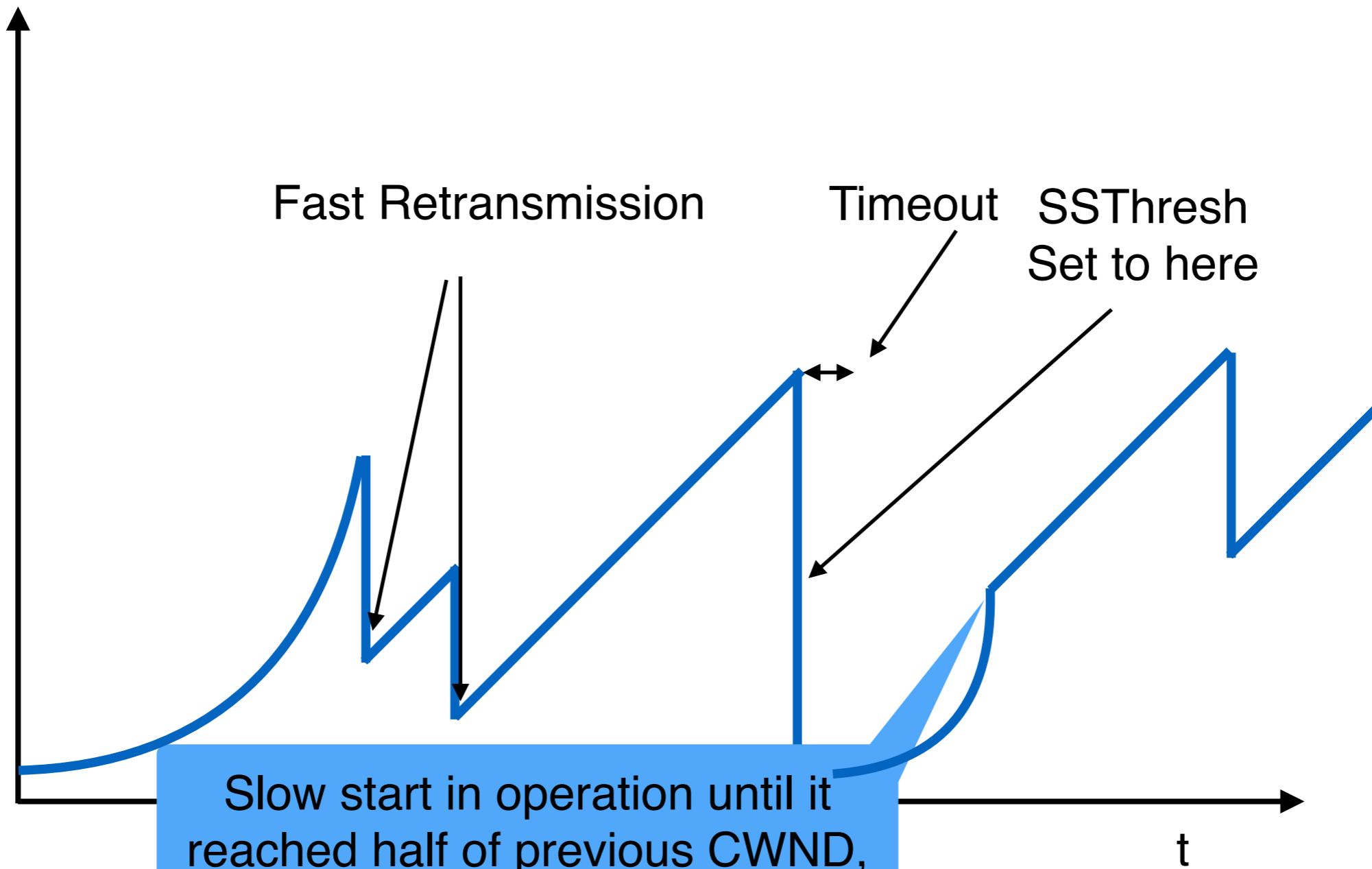
- Would like to maximize bandwidth utilization
- If network has bandwidth available, flows should be able to use it!

# Recap: Basic Components of TCP

- **Connections:** Explicit set-up and tear-down of TCP sessions/connections
- **Segments, Sequence numbers, ACKs**
  - TCP uses byte sequence numbers to identify payloads
  - ACKs referred to sequence numbers
  - Window sizes expressed in terms of # of bytes
- **Retransmissions**
  - Can't be correct without retransmitting lost/corrupted data
  - TCP retransmits based on timeouts and duplicate ACKs
    - Timeouts based on estimate of RTT
- **Flow Control:** Ensures the sender does not overwhelm the receiver
- **Congestion Control:** Dynamic adaptation to network path's capacity

# Recap: Final Time Diagram

Window



Slow-start restart: Go back to CWND of 1 MSS, but take advantage of knowing the previous value of CWND.

# Recap: Max-Min Fairness

- Given a set of bandwidth demands  $r_i$  and total bandwidth  $C$ , max-min bandwidth allocations are  $a_i = \min(f, r_i)$ 
  - Where  $f$  is the unique value such that  $\text{Sum}(a_i) = C$  or set  $f$  to be infinite if no such value exists
- **This is what round-robin service gives**
  - If all packets are MTU
- Property:
  - If you don't get full demand, no one gets more than you

# Computing Max-Min Fairness

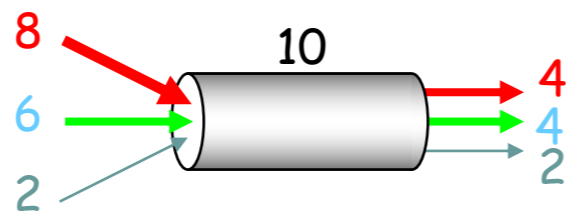
- Assume demands are in increasing order...
- If  $C/N \leq r_1$ , then  $a_i = C/N$  for all  $i$
- Else,  $a_1 = r_1$ , set  $C = C - a_1$  and  $N = N - 1$
- Repeat
- Intuition: all flows requesting less than fair share get their request.  
Remaining flows divide equally

# Example

- Assume link speed  $C$  is 10Mbps
- Have three flows:
  - Flow 1 is sending at a rate 8 Mbps
  - Flow 2 is sending at a rate 6 Mbps
  - Flow 3 is sending at a rate 2 Mbps
- How much bandwidth should each get?
  - According to max-min fairness?
- Work this out, talk to your neighbors

# Example

- Requests:  $r_i$       Allocations:  $a_i$
- Requests:  $r_1 = 8, r_2 = 6, r_3 = 2$
- $C = 10, N = 3, C/N = 3.33$ 
  - Can serve all for  $r_3$
  - Remove  $r_3$  from the accounting:  $C = C - r_3 = 8, N = 2$
- $C/2 = 4$ 
  - Can't service all for  $r_1$  or  $r_2$
  - So hold them to the remaining fair share:  $f = 4$

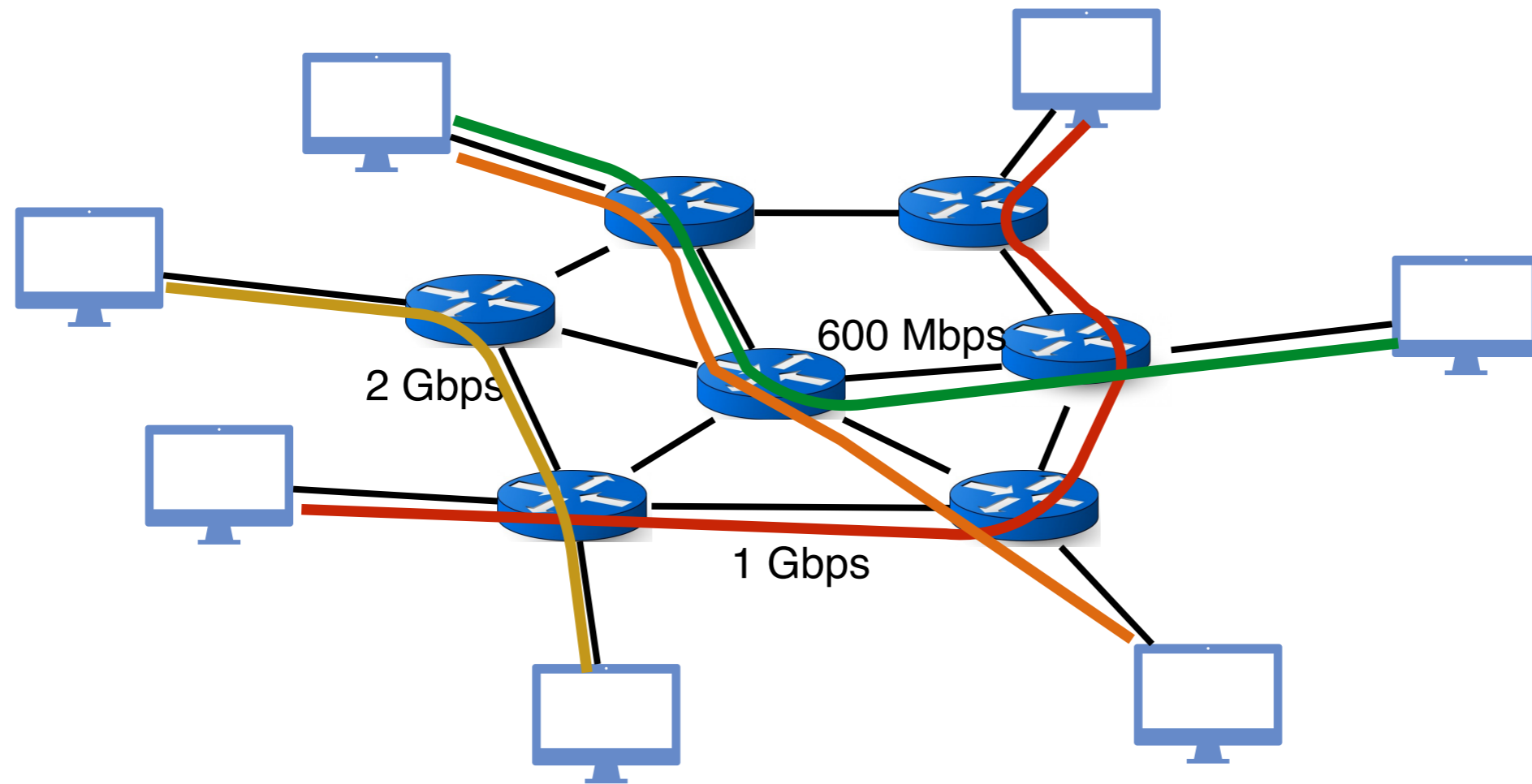


|                  |
|------------------|
| $f = 4:$         |
| $\min(8, 4) = 4$ |
| $\min(6, 4) = 4$ |
| $\min(2, 4) = 2$ |

# Max-Min Fairness

- Max-min fairness the natural per-link fairness
- Only one that is
  - Symmetric
  - Incentive compatible (asking for more doesn't help)

# Reality of Congestion Control



**Congestion control is a resource allocation problem involving many flows, many links and complicated global dynamics**

**Classical result:**

**In a stable state**

**(no dynamics; all flows are infinitely long; no failures; etc.)**

**TCP guarantees max-min fairness**

**Any Questions?**

# The Many Failings of TCP Congestion Control

1. Fills up queues (large queueing delays)
2. Every segment not ACKed is a loss (non-congestion related losses)
3. Produces irregular saw-tooth behavior
4. Biased against long RTTs (unfair)
5. Not designed for short flows
6. Easy to cheat

# (1) TCP Fills Up Queues

- TCP only slows down when queues fill up
  - High queueing delays
- Means that it is not optimized for latency
  - What is it optimized for then?
    - **Answer: Fairness (discussion in next few slides)**
- And many packets are dropped when buffer fills
- Alternative 1: Use small buffers
  - Is this a good idea?
  - Answer: No, bursty traffic will lead to reduced utilization
- Alternative: **Random Early Drop (RED)**
  - Drop packets on purpose **before** queue is full
  - A very clever idea

# Random Early Drop (or Detection)

- Measure average queue size  $A$  with exponential weighting
  - Average: Allows for short bursts of packets without over-reacting
- Drop probability is a function of  $A$ 
  - No drops if  $A$  is very small
  - Low drop rate for moderate  $A$ 's
  - Drop everything if  $A$  is too big
- Drop probability applied to incoming packets
- Intuition: link is fully utilized well before buffer is full

# Advantages of RED

- Keeps queues smaller, while allowing bursts
  - Just using small buffers in routers can't do the latter
- Reduces synchronization between flows
  - Not all flows are dropping packets at once
  - Increases/decreases are more gentle
- Problem
  - Turns out that RED does not guarantee fairness

## (2) Non-Congestion-Related Losses?

- For instance, RED drops packets intentionally
  - TCP would think the network is congested
- Can use **Explicit Congestion Notification (ECN)**
- Bit in IP packet header (actually two)
  - TCP receiver returns this bit in ACK
- When RED router would drop, it sets bit instead
  - Congestion semantics of bit exactly like that of drop
- Advantages
  - Doesn't confuse corruption with congestion

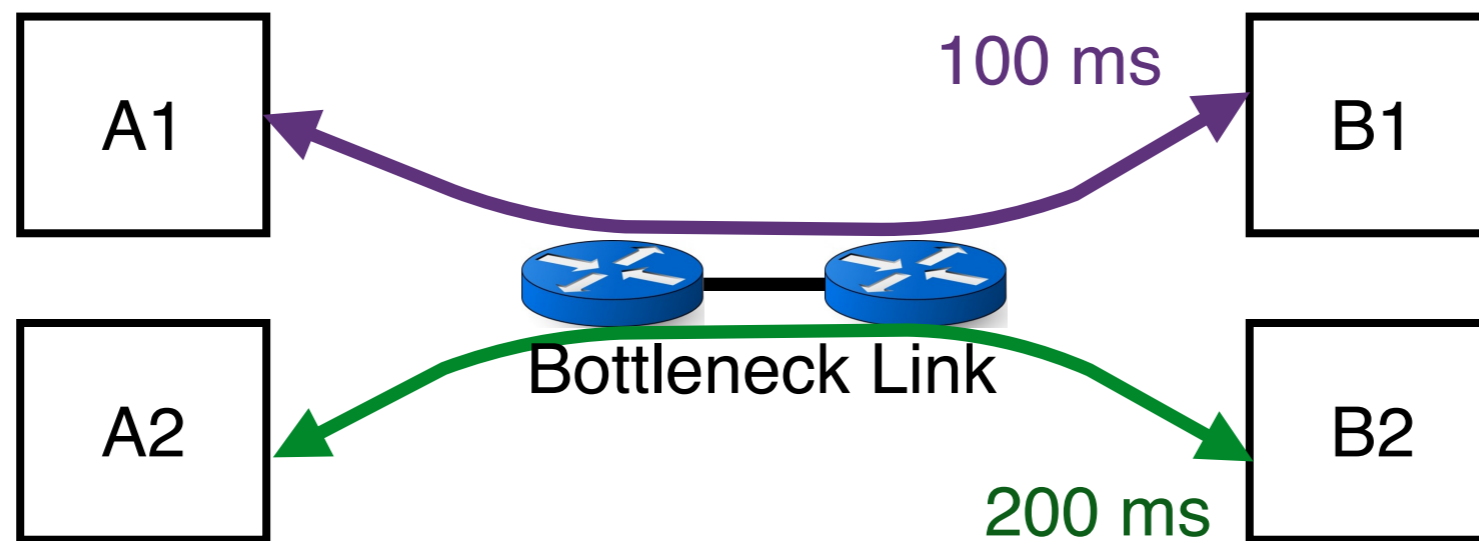
### (3) Sawtooth Behavior Uneven

- TCP throughput is “choppy”
  - Repeated swings between  $W/2$  to  $W$
- Some apps would prefer sending at a steady rate
  - E.g., streaming apps
- A solution: “Equation-based congestion control”
  - Ditch TCP’s increase/decrease rules and just follow the equation:
  - **[Matthew Mathis, 1997] TCP Throughput =  $MSS/RTT \sqrt{3/2p}$** 
    - **Where  $p$  is drop rate**
  - Measure drop percentage  $p$  and set rate accordingly
- Following the TCP equation ensures we’re TCP friendly
  - I.e., use no more than TCP does in similar setting

**Any Questions?**

## (4) Bias Against Long RTTs

- Flows get throughput inversely proportional to RTT
- **TCP unfair in the face of heterogeneous RTTs!**
- [Matthew Mathis, 1997] TCP Throughput =  $MSS/RTT \sqrt{3/2p}$ 
  - Where  $p$  is drop rate
- Flows with long RTT will achieve lower throughput



# Possible Solutions

- Make additive constant proportional to RTT
- But people don't really care about this...

## (5) How Short Flows Fare?

- Internet traffic:
  - Elephant and mice flows
  - Elephant flows carry most bytes (>95%), but are very few (<5%)
  - Mice flows carry very few bytes, but most flows are mice
    - 50% of flows have < 1500B to send (1 MTU);
    - 80% of flows have < 100KB to send
- Problem with TCP?
  - Mice flows do not have enough packets for duplicate ACKs!!
  - Drop  $\approx$  Timeout (unnecessary high latency)
  - These are precisely the flows for which latency matters!!!
- Another problem:
  - Starting with small window size leads to high latency

# Possible Solutions?

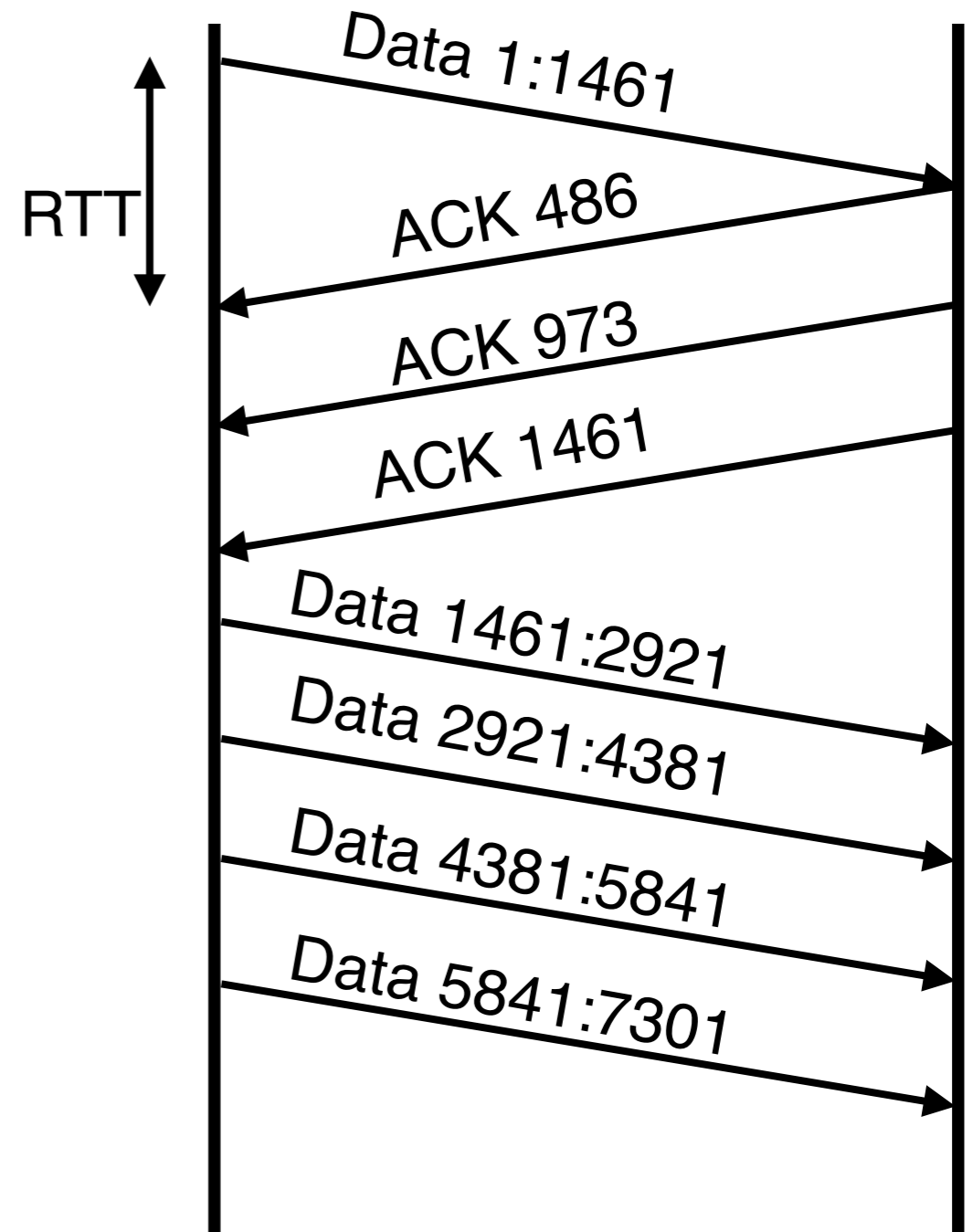
- Larger initial window?
  - Google proposed moving from ~4KB to ~15KB
  - Covers ~90% of HTTP Web
  - Decreases delay by 5%
- Many recent research papers on the timeout problem
  - Require network support

## (6) Cheating

- TCP was designed assuming a cooperative world
- No attempt was made to prevent cheating
- Many ways to cheat, will present three

# Cheating #1: ACK-splitting (receiver)

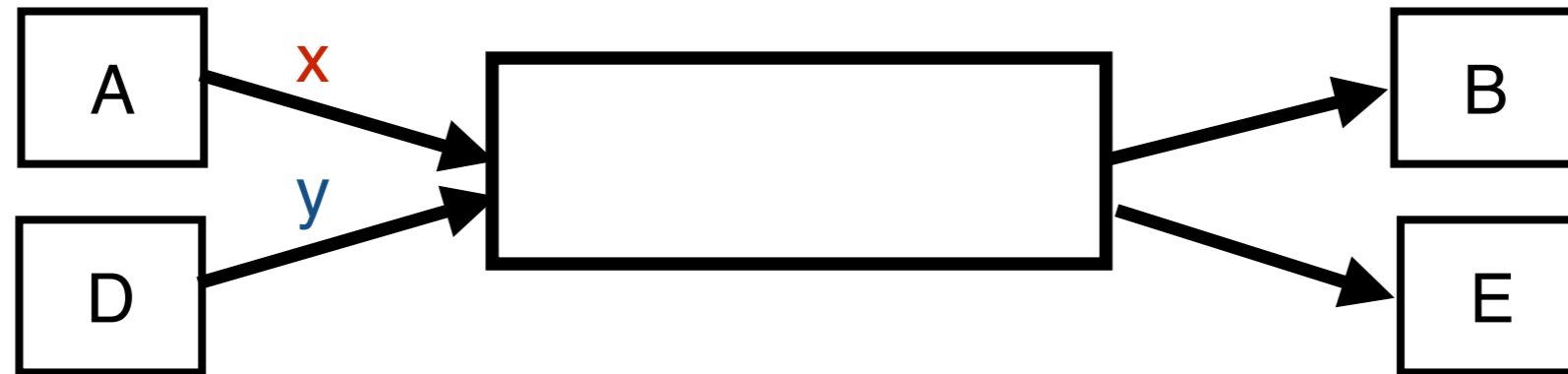
- TCP Rule: grow window by one MSS for each valid ACK received
- Send **M** (distinct) ACKs for one MSS
- Growth factor proportional to **M**



# Cheating #2: Increasing CWND Faster (source)

- TCP Rule: increase window by one MSS for each valid ACK received
- Increase window by **M** per ACK
- Growth factor proportional to **M**

# Cheating #3: Open Many Connections (source/receiver)



- Assume
  - A start 10 connections to B
  - D starts 1 connection to E
  - Each connection gets about the same throughput
- Then A gets 10 times more throughput than D

# Cheating

- Either sender or receiver can independently cheat!
- **Why hasn't Internet suffered congestion collapse yet?**
  - Individuals don't hack TCP (not worth it)
  - Companies need to avoid TCP wars
- How can we prevent cheating
  - Verify TCP implementations
  - Controlling end points is hopeless
- Nobody cares, really

**Any Questions?**

**Now you know about computer networking  
as much as I do :-)**

Back up slides on UDP  
(not needed for exams)

# UDP: User Datagram Protocol

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive 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”
  - (local port, local IP, remote port, remote IP)  $\longleftrightarrow$  socket

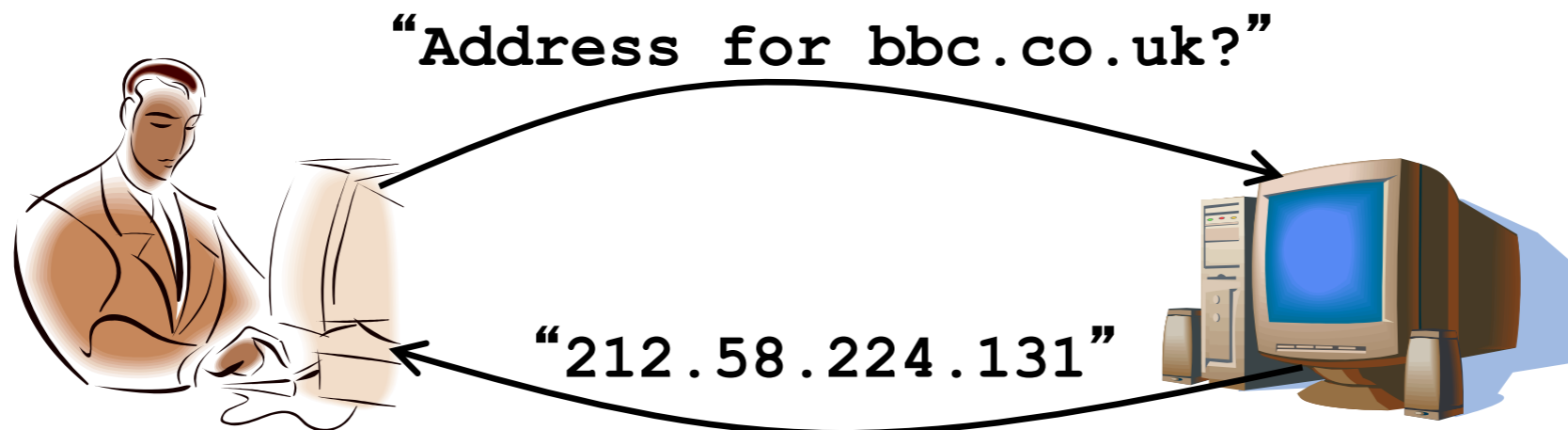
|                            |             |
|----------------------------|-------------|
| Source Port #              | Dest Port # |
| Checksum                   | Length      |
| Application Data (Message) |             |

# Question

- Why do UDP packets carry sender's port?

# Popular Applications That Use UDP

- Some interactive streaming apps
  - Retransmitting lost/corrupted packets is often pointless — by the time the packet is transmitted, 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



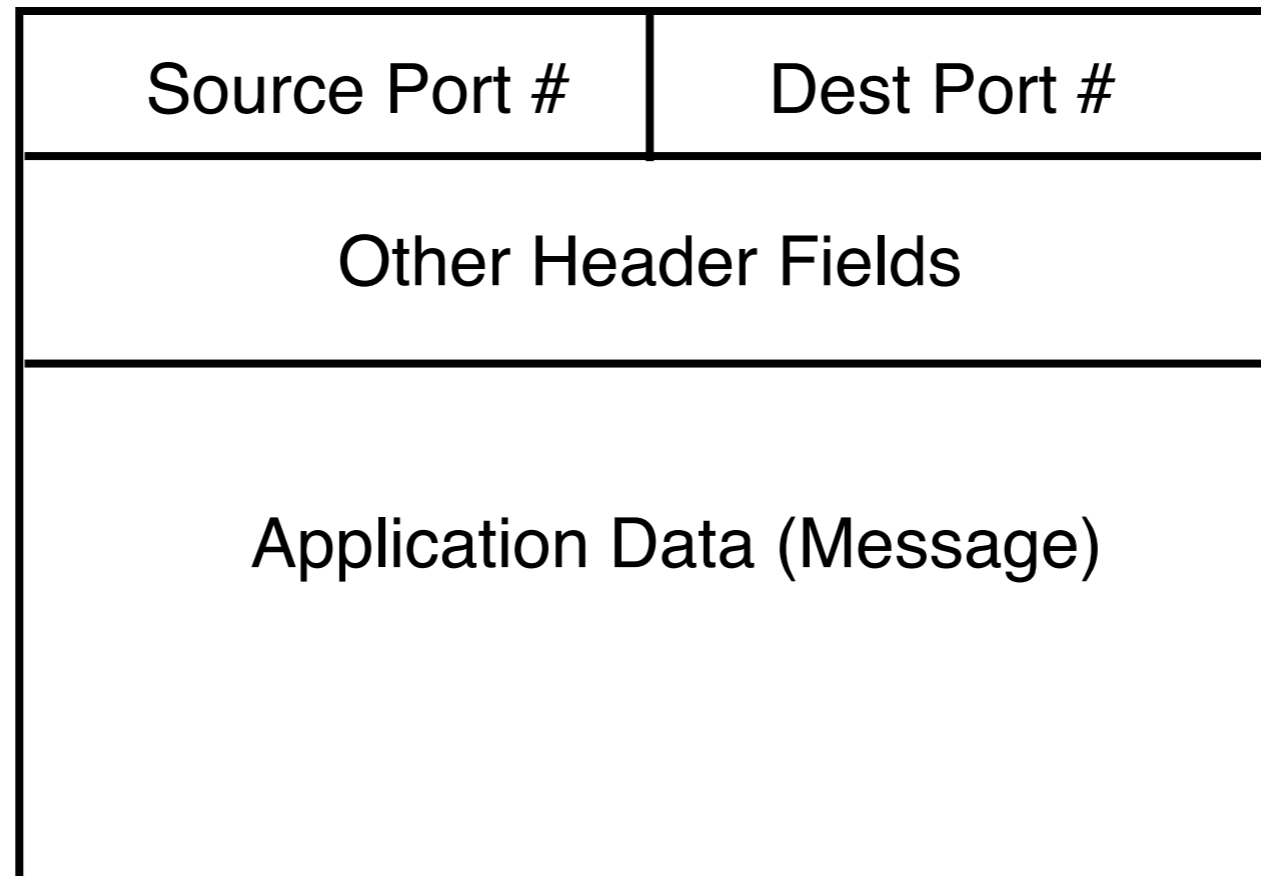
Back up slides on TCP  
(not needed for exams)

# Ports

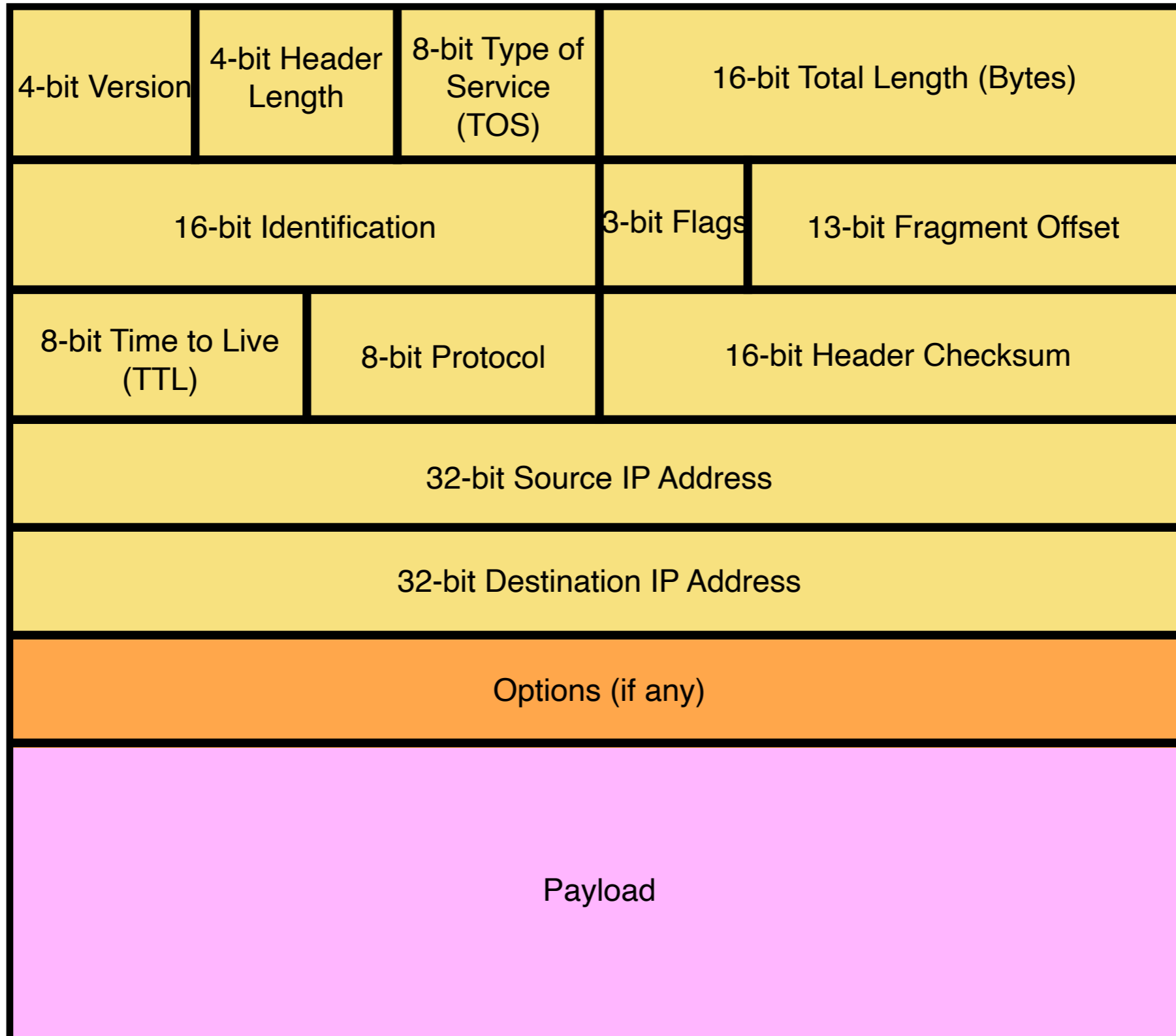
- Separate 16-bit port address space for UDP, TCP
- “Well known” ports (0-1023)
  - Agreement on which services run on these ports
  - e.g., ssh:22, http:80
  - Client (app) knows appropriate port on sender
  - Services can listen on well-known ports

# Multiplexing and Demultiplexing

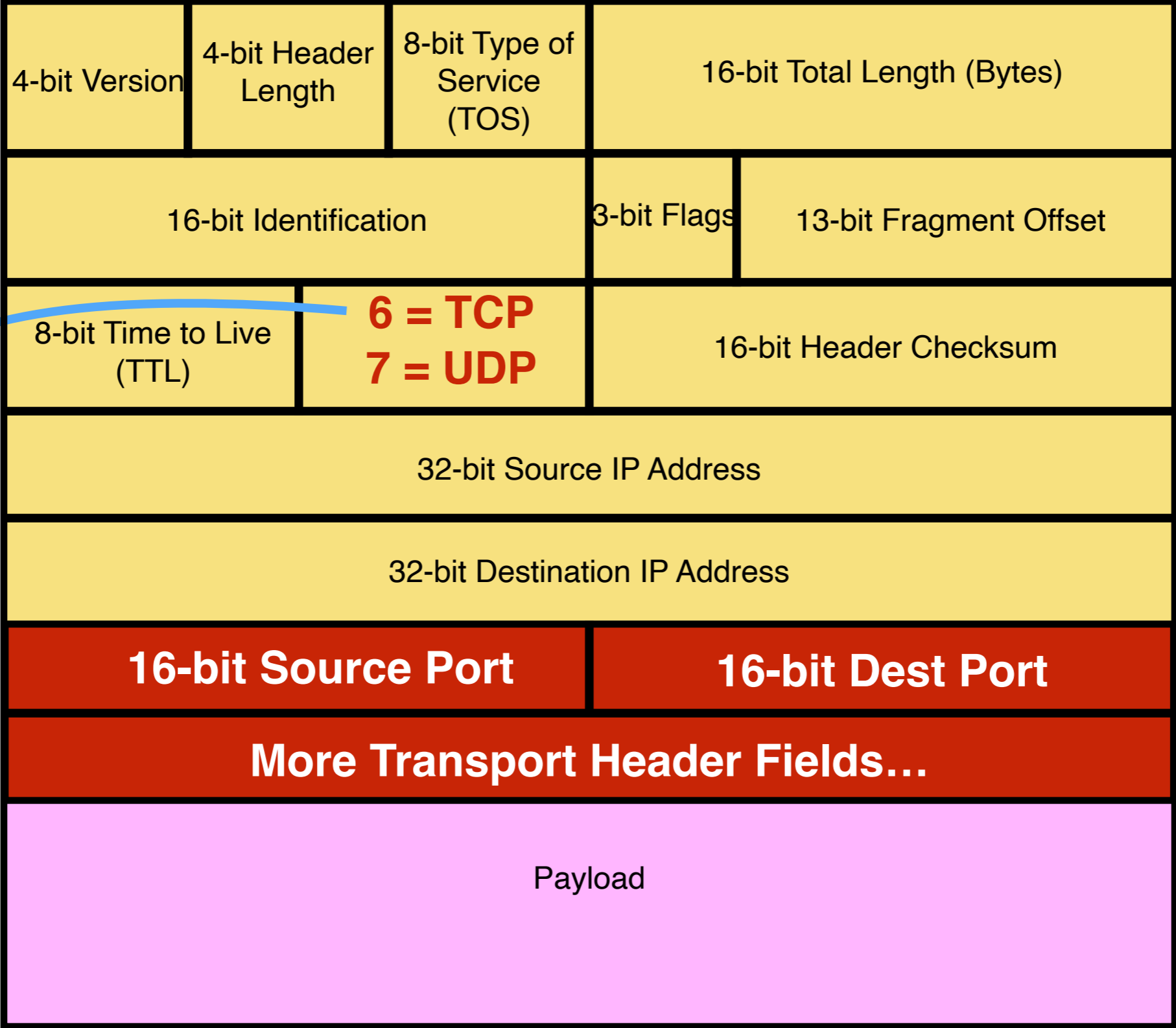
- Host receives IP datagrams
  - Each datagram has source and destination IP address
  - Each segment has source and destination port number
- Host uses IP address and port numbers to direct the segment to appropriate socket



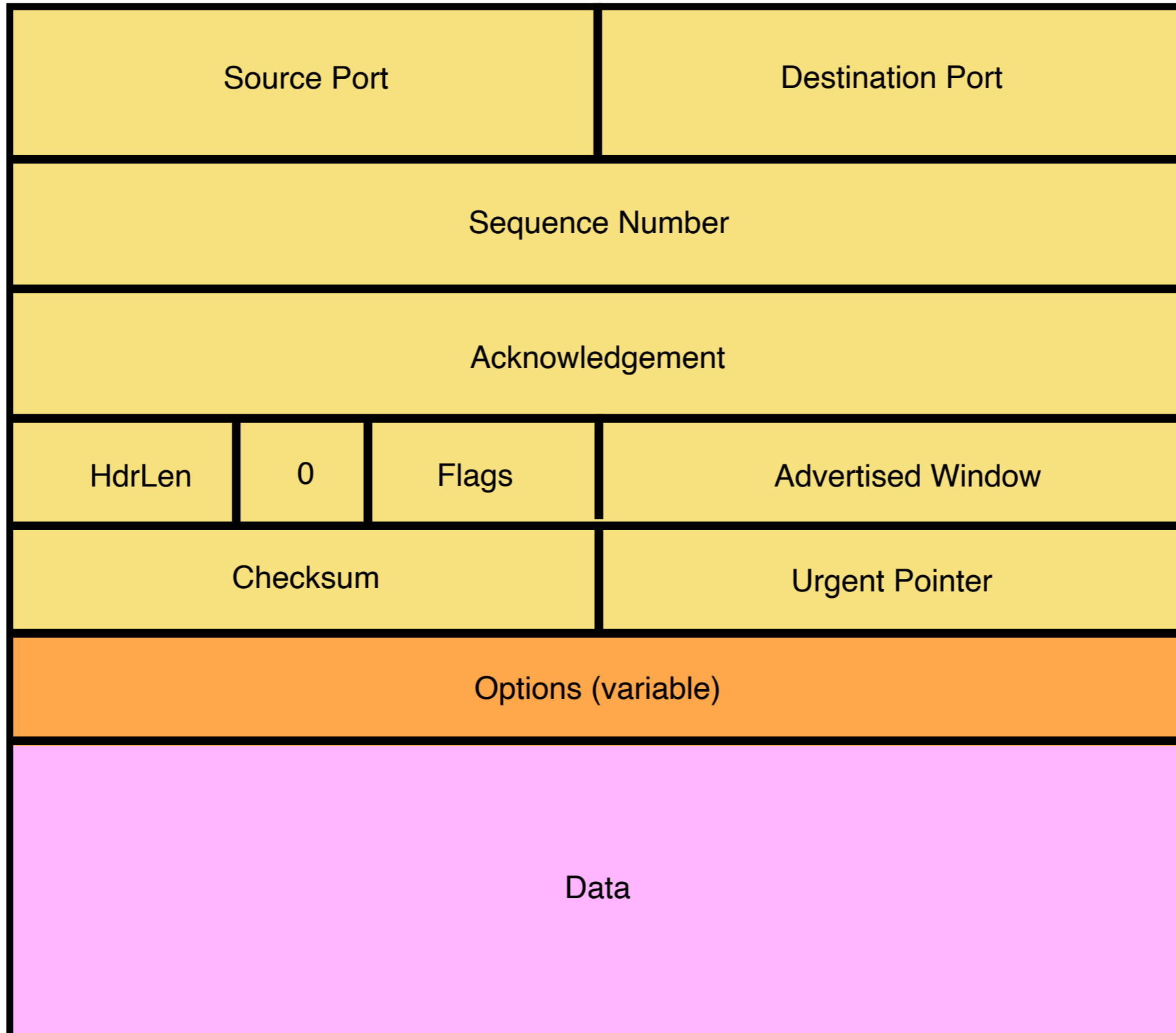
# IP Packet Structure



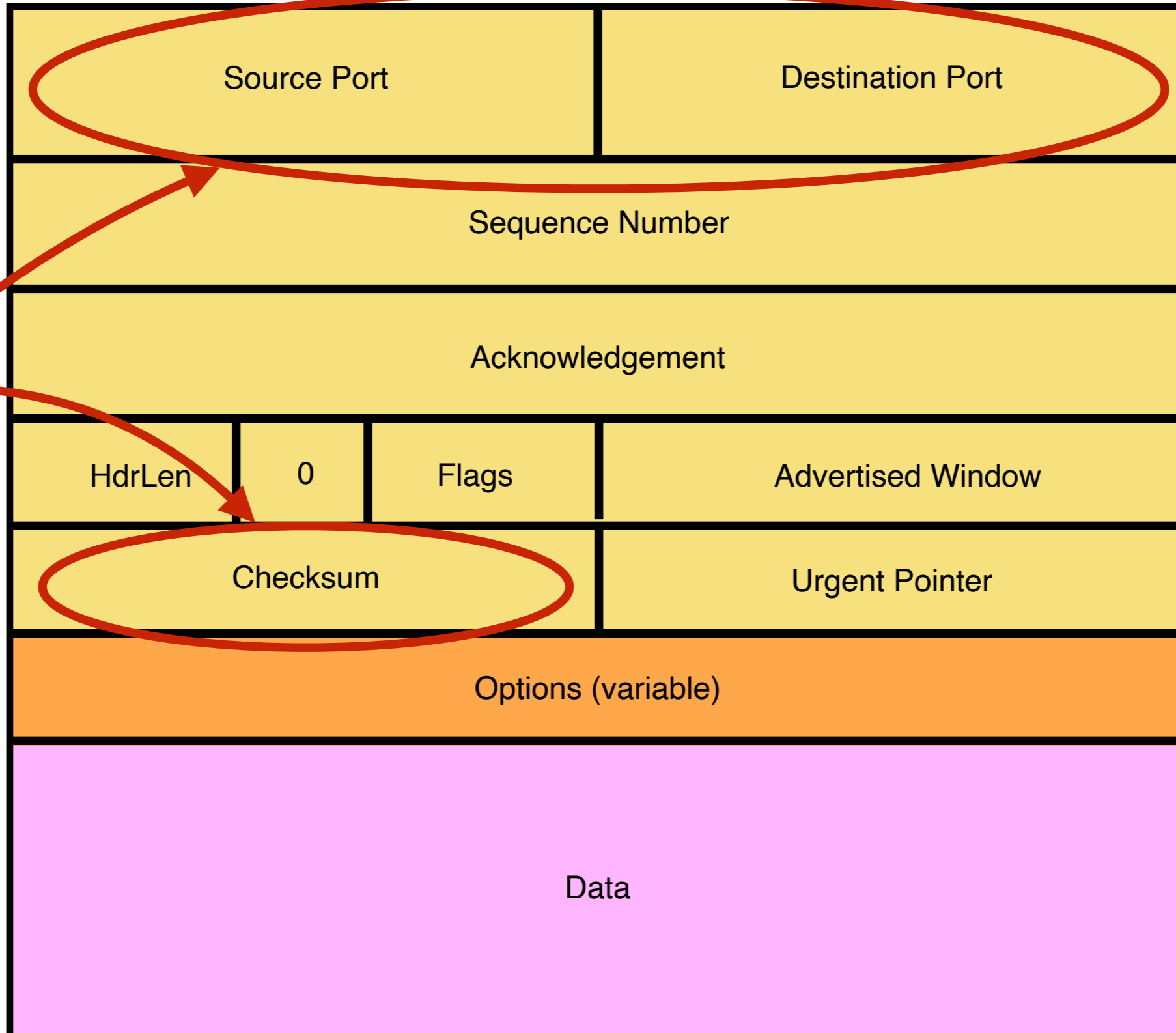
# IP Packet Structure



# TCP Header



# TCP Header



These  
should be  
familiar

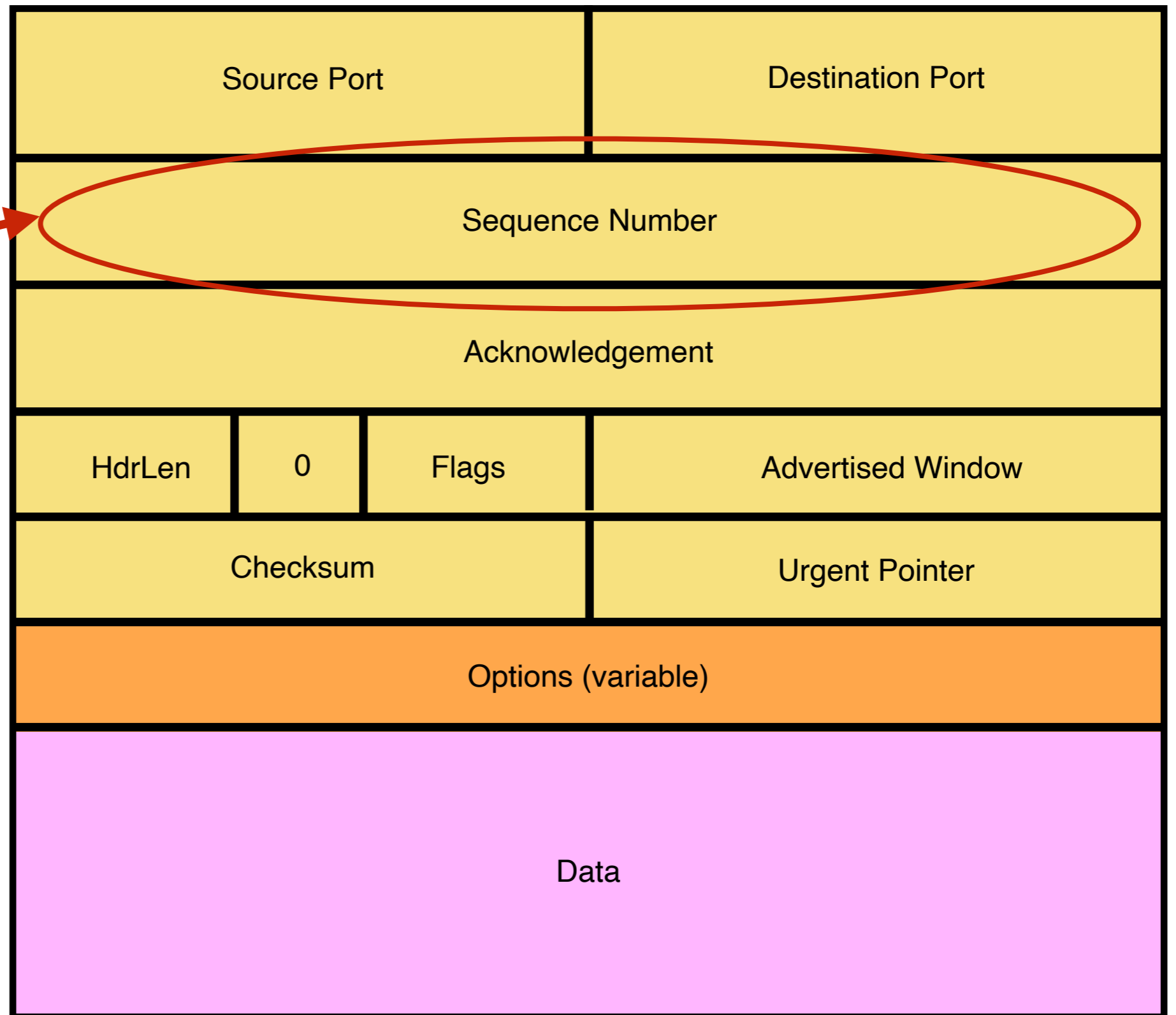
# TCP Segment



- 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 MSS (Maximum Segment Size) bytes
  - E.g., upto 1460 consecutive bytes from the stream
  - $MSS = MTU - IP\ header - TCP\ header$

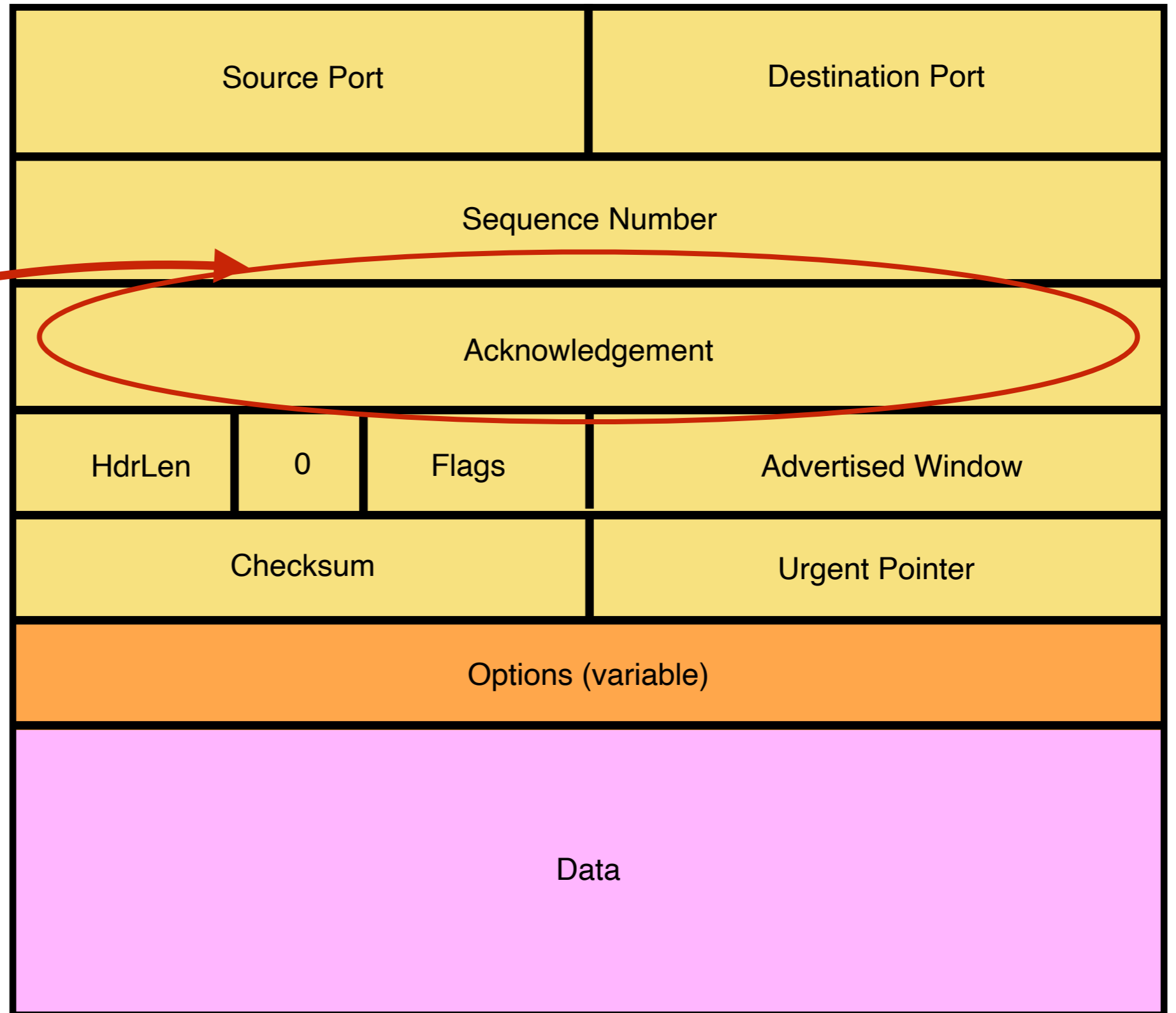
# TCP Header

Starting byte offset  
of data carried in  
this segment

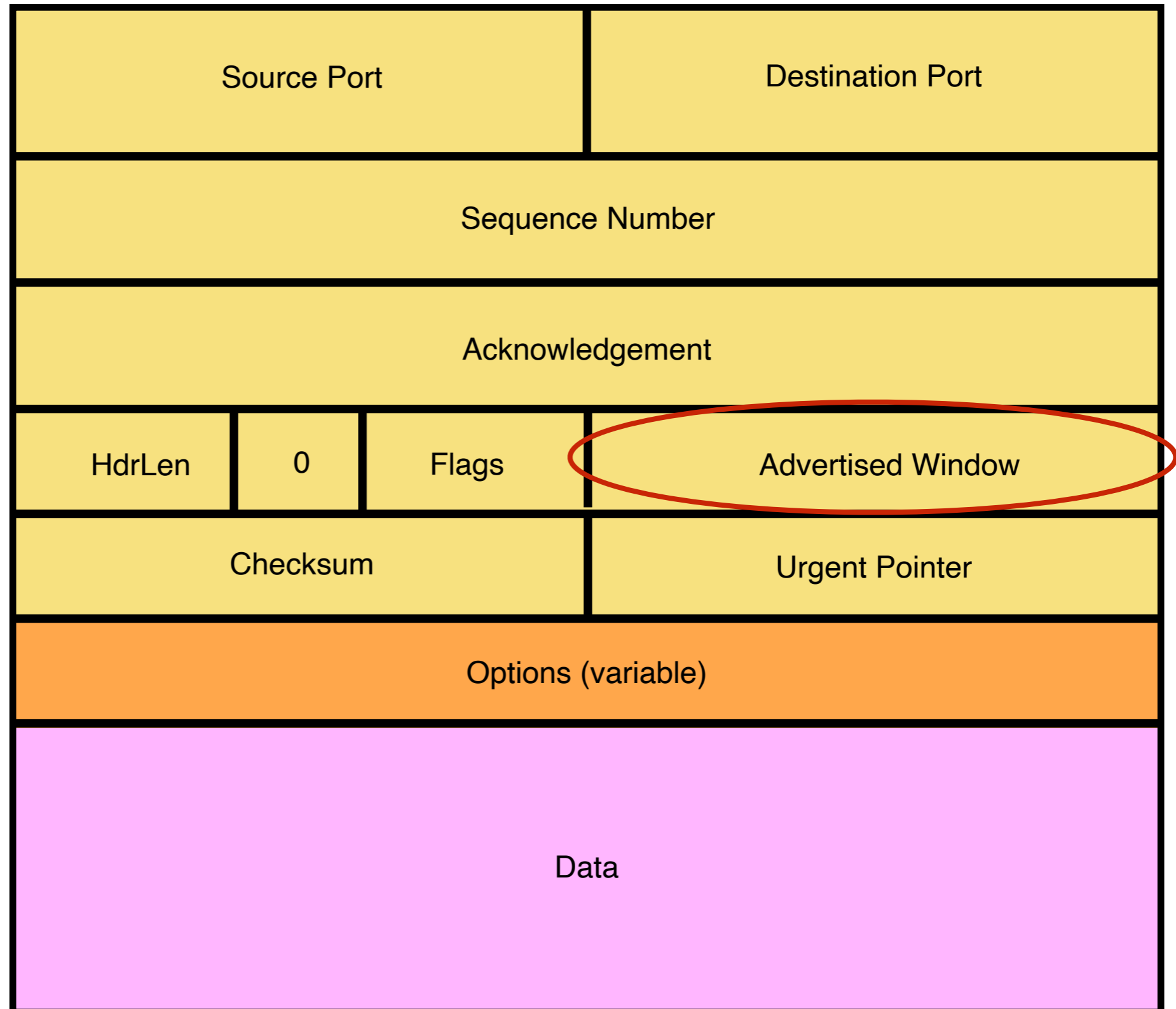


# TCP Header

Acknowledgement gives sequence number just beyond highest sequence number received in order (“What byte is next”)

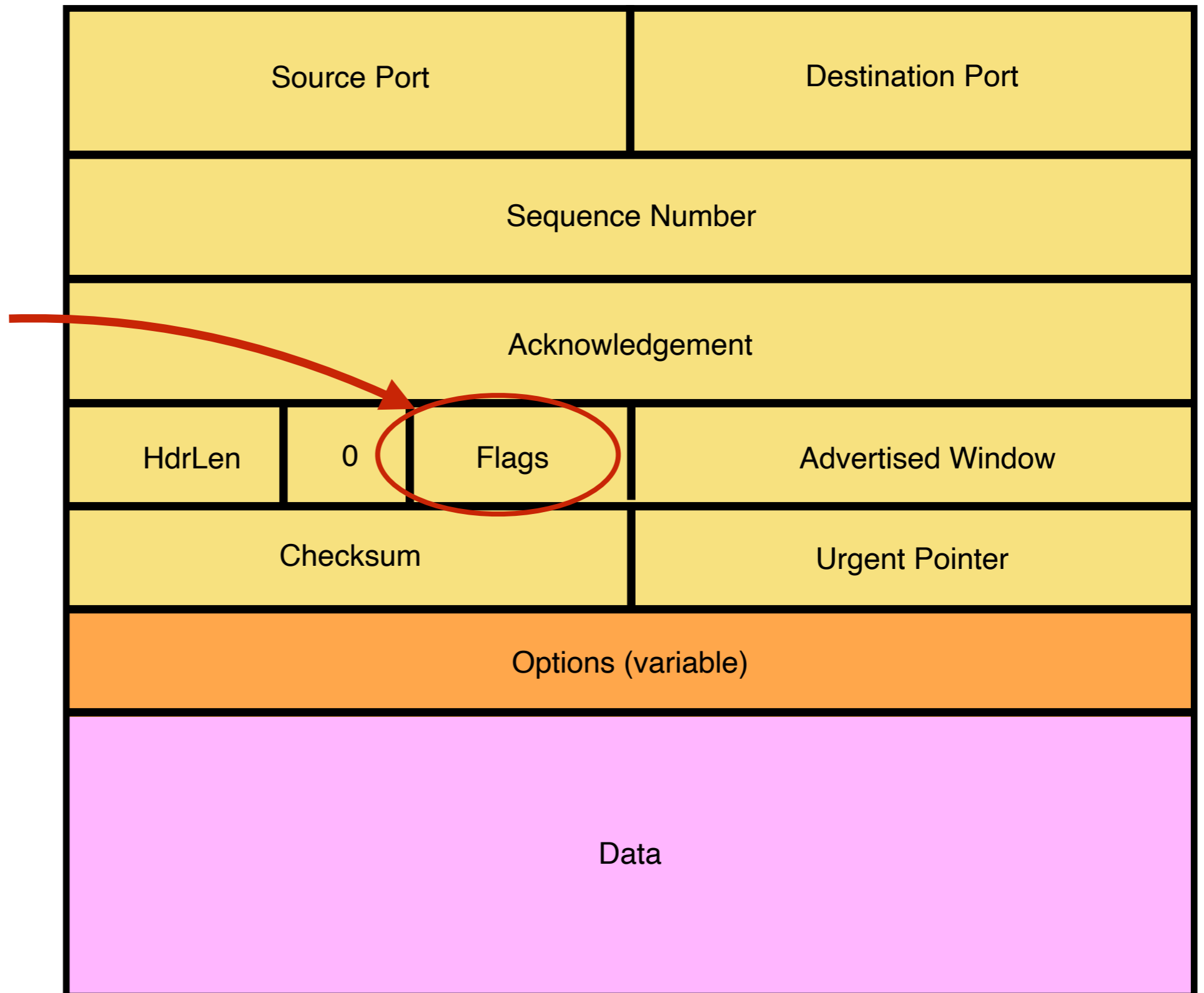


# TCP Header



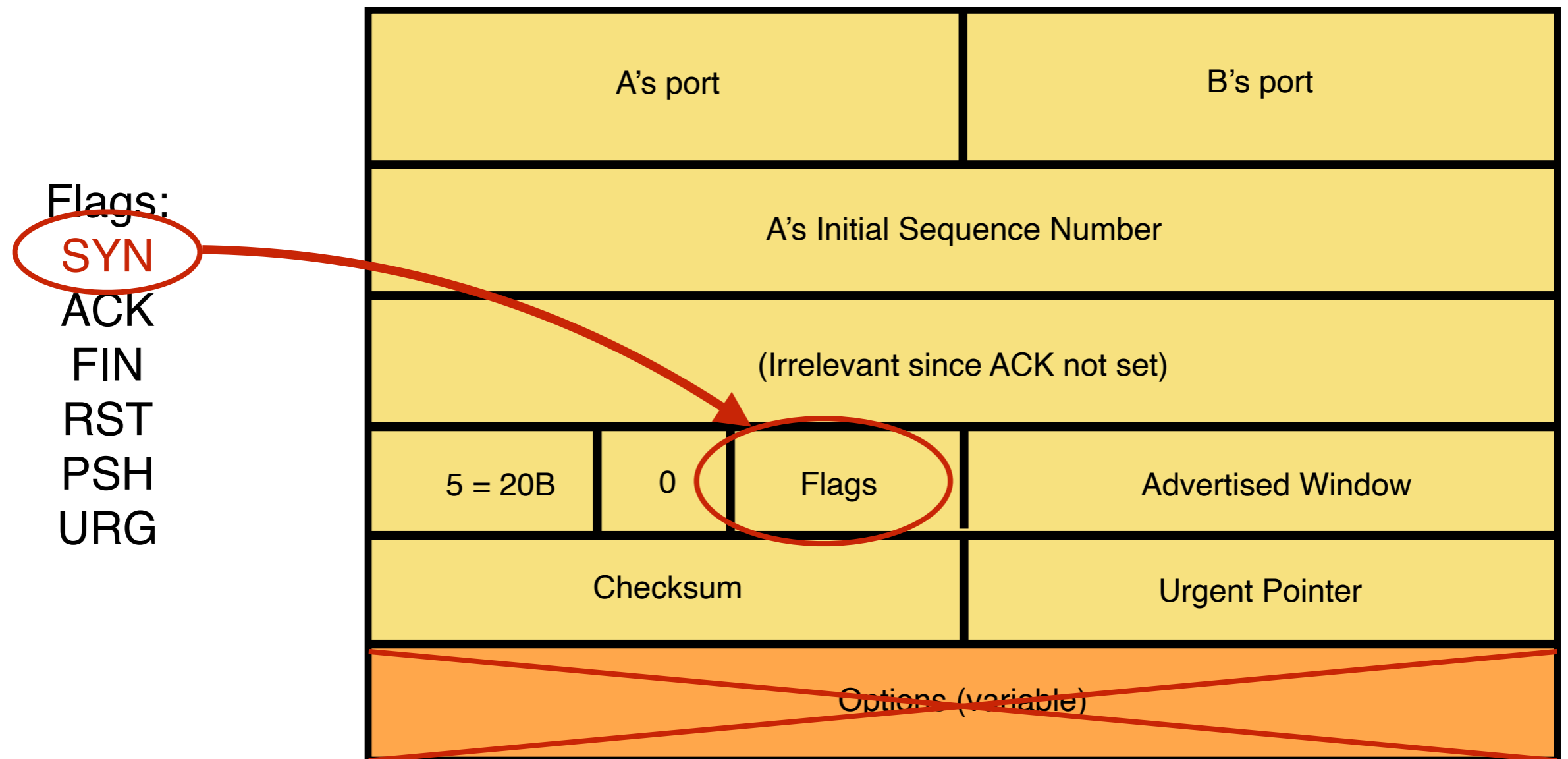
# TCP Header

Flags:  
SYN  
ACK  
FIN  
RST  
PSH  
URG



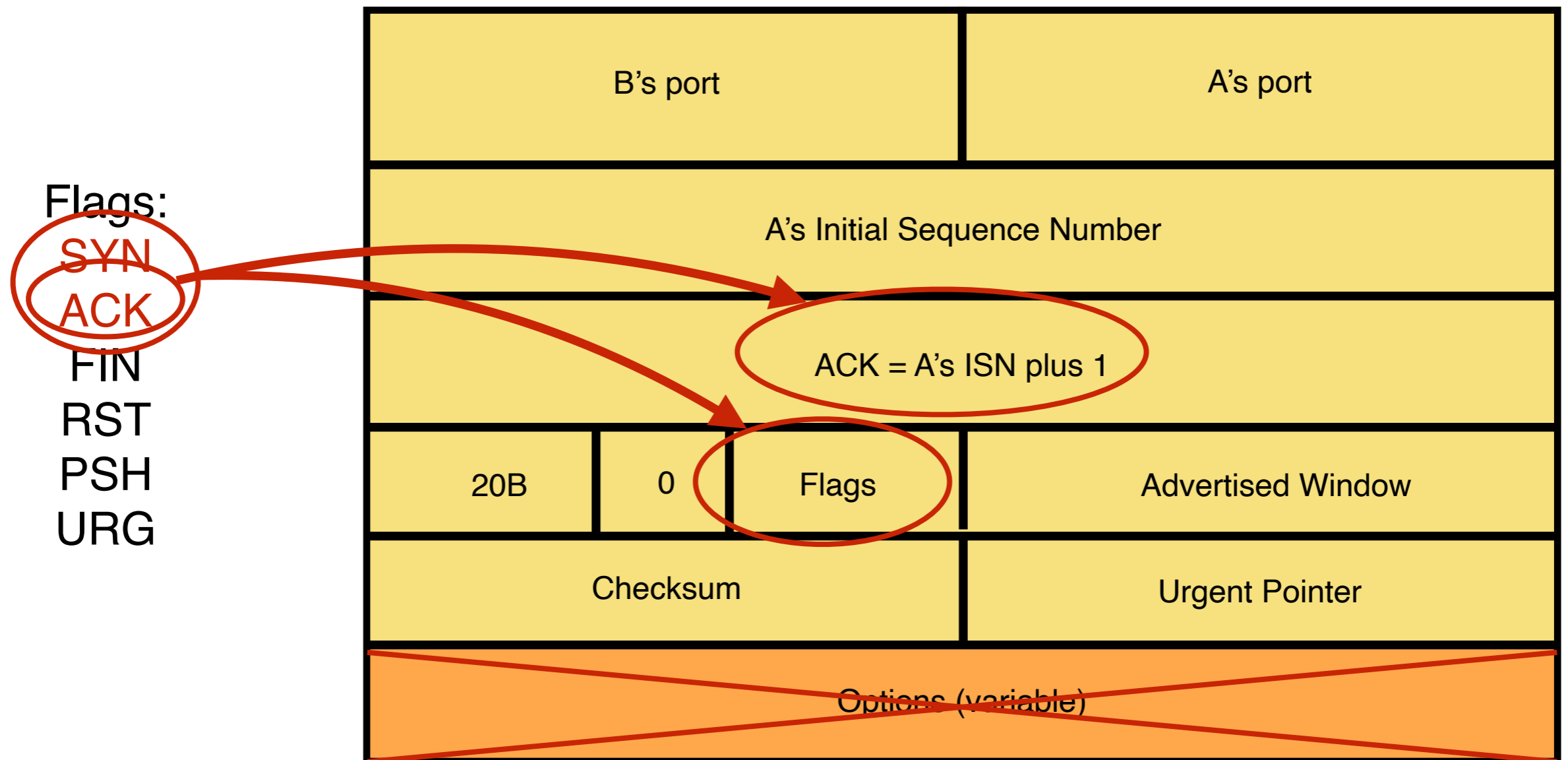
See `/usr/include/netinet/tcp.h` on Unix Systems

# Step 1: A's Initial SYN Packet



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

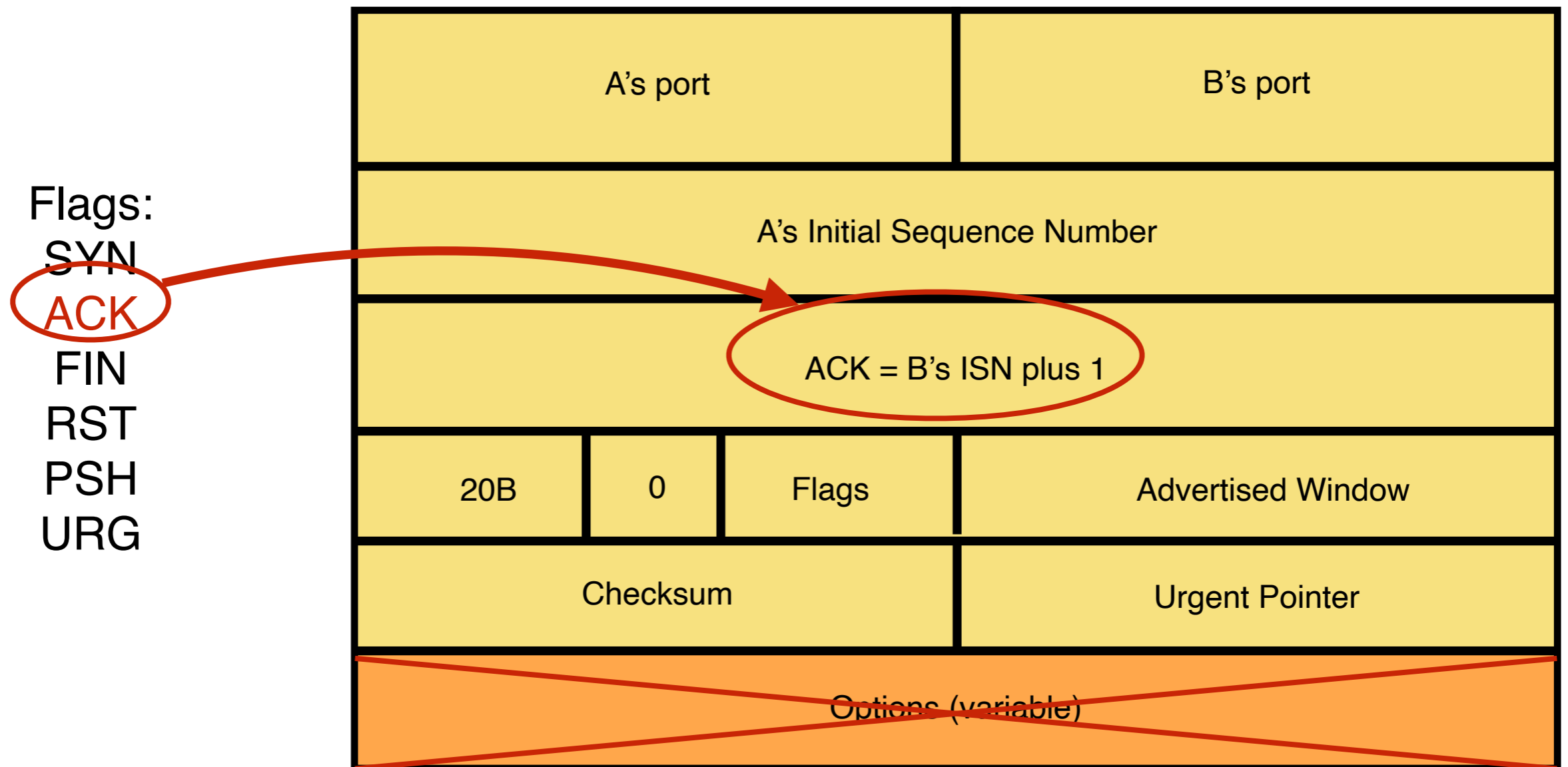
## Step 2: B's SYN-ACK Packet



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

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

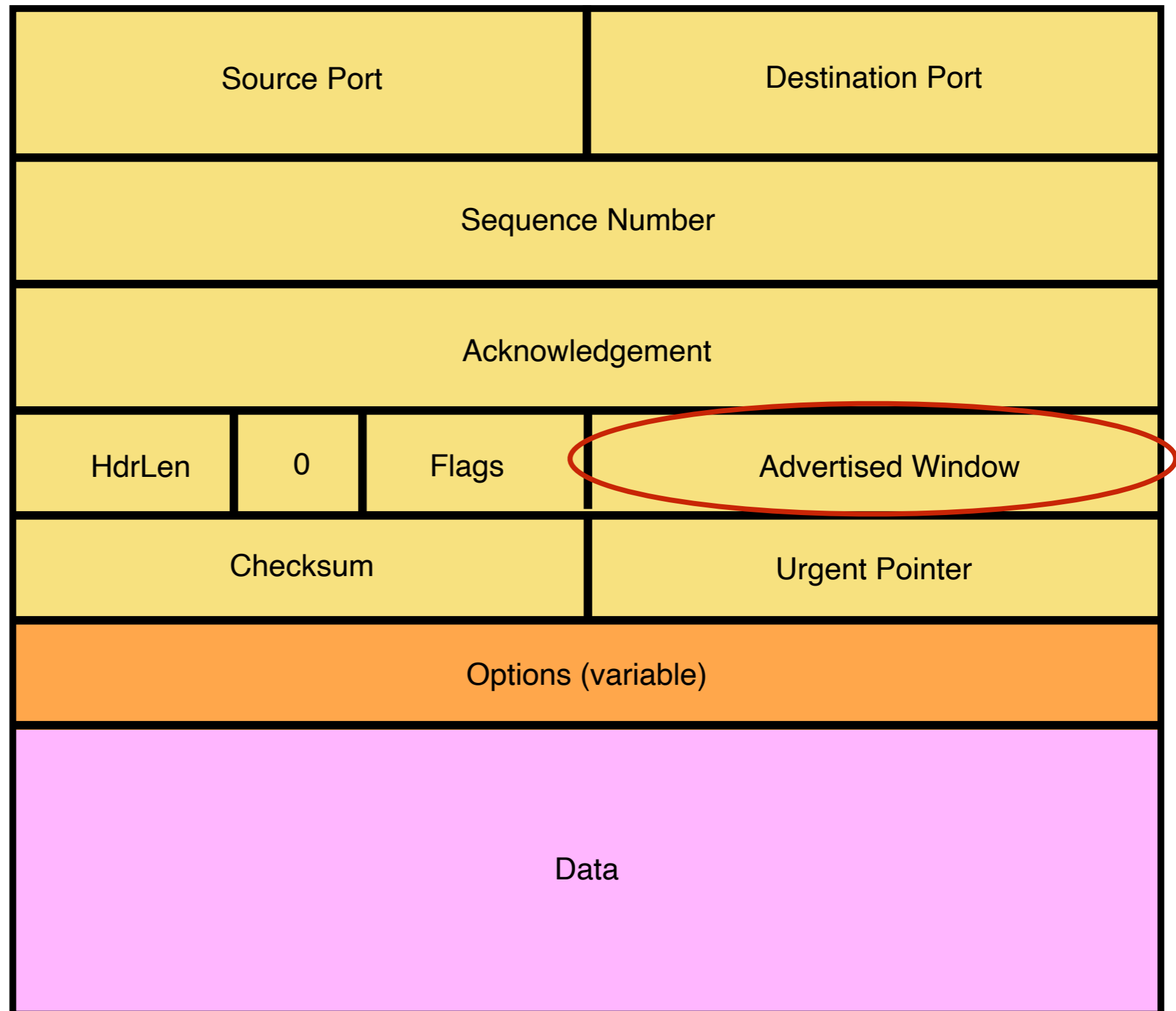
## Step 3: A's ACK of the SYN-ACK



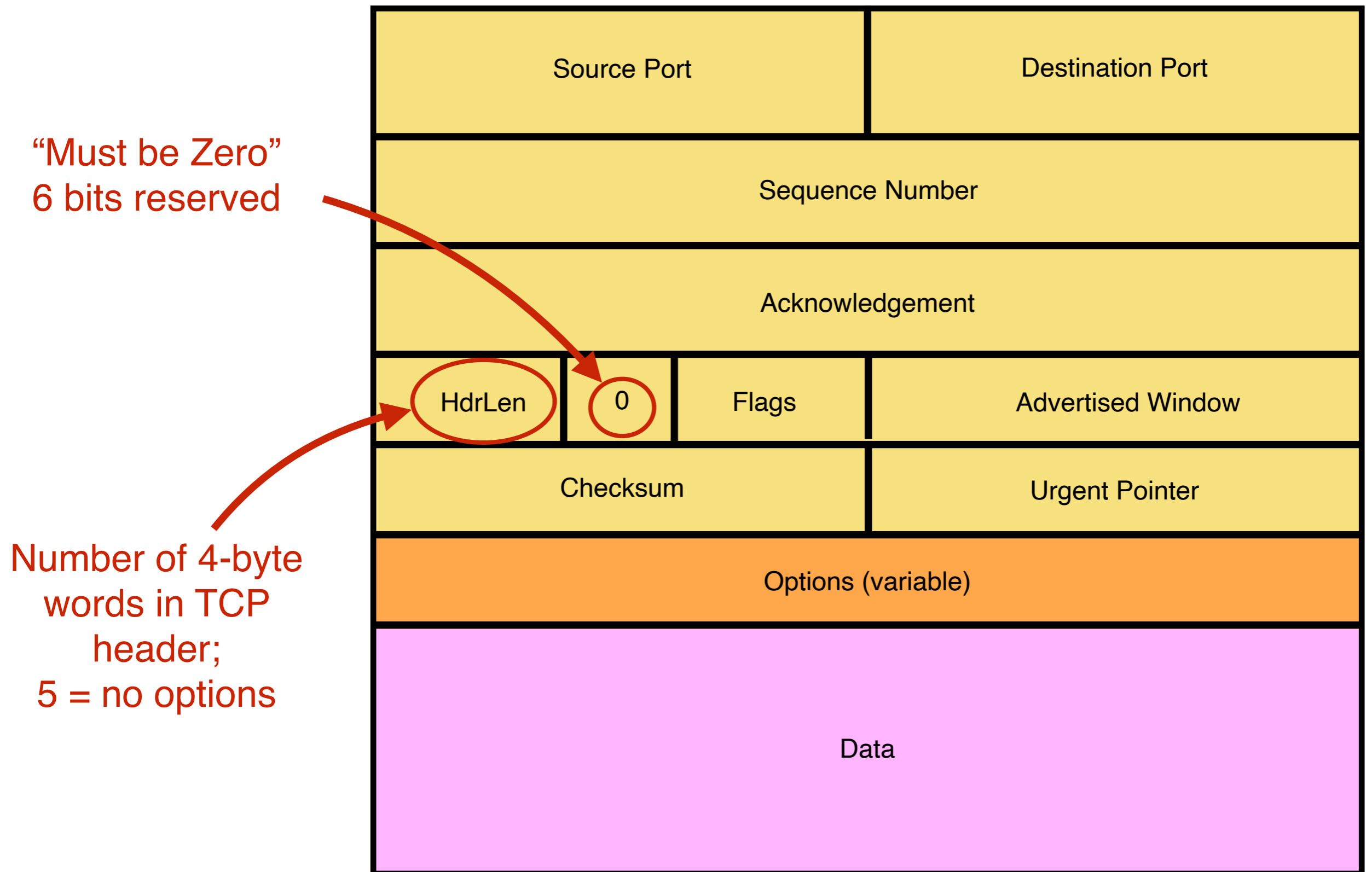
A tells B it's likewise okay to start sending

... upon receiving this packet, B can start sending data

# TCP Header

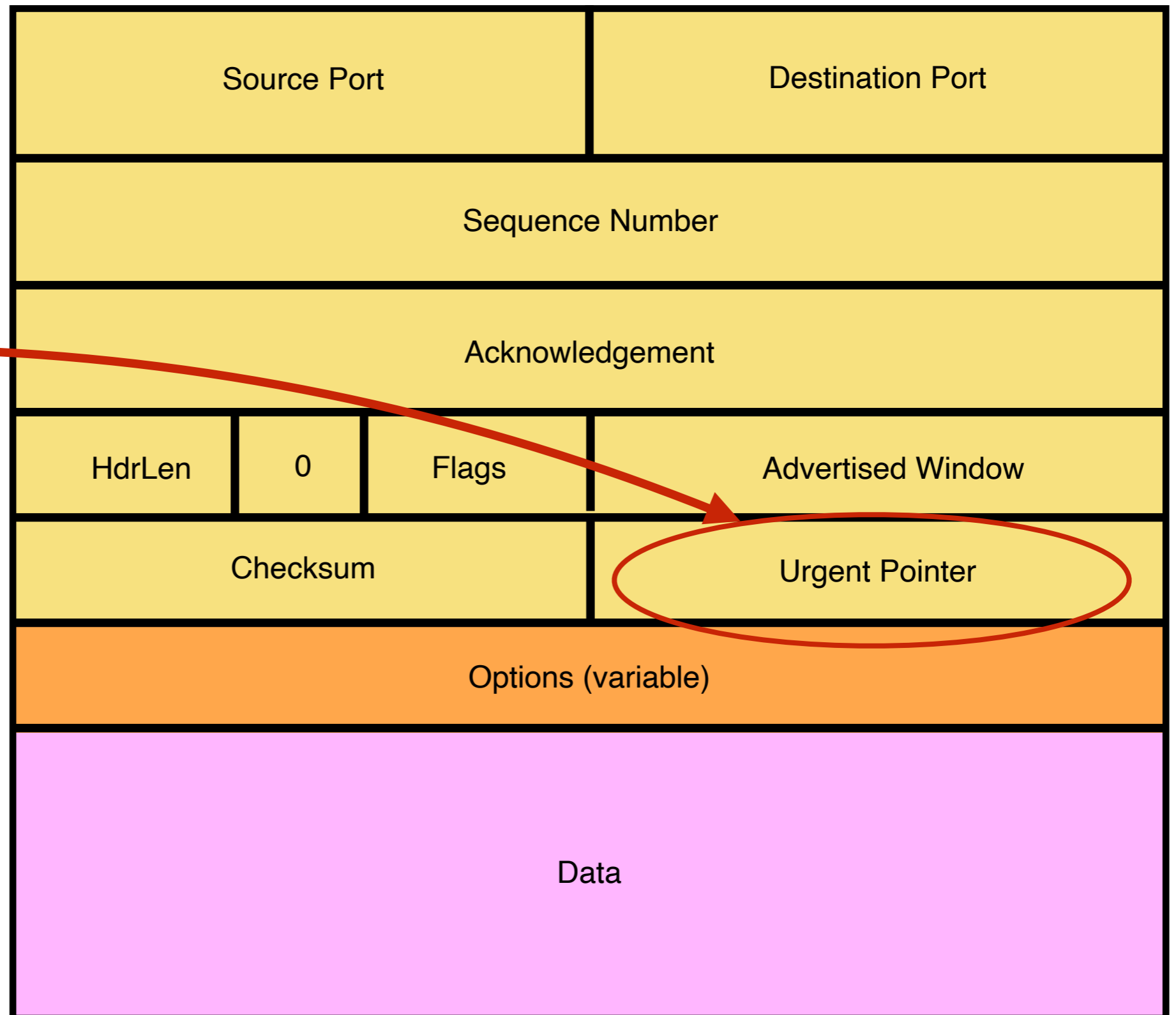


# TCP Header: What's left?



# TCP Header: What's left?

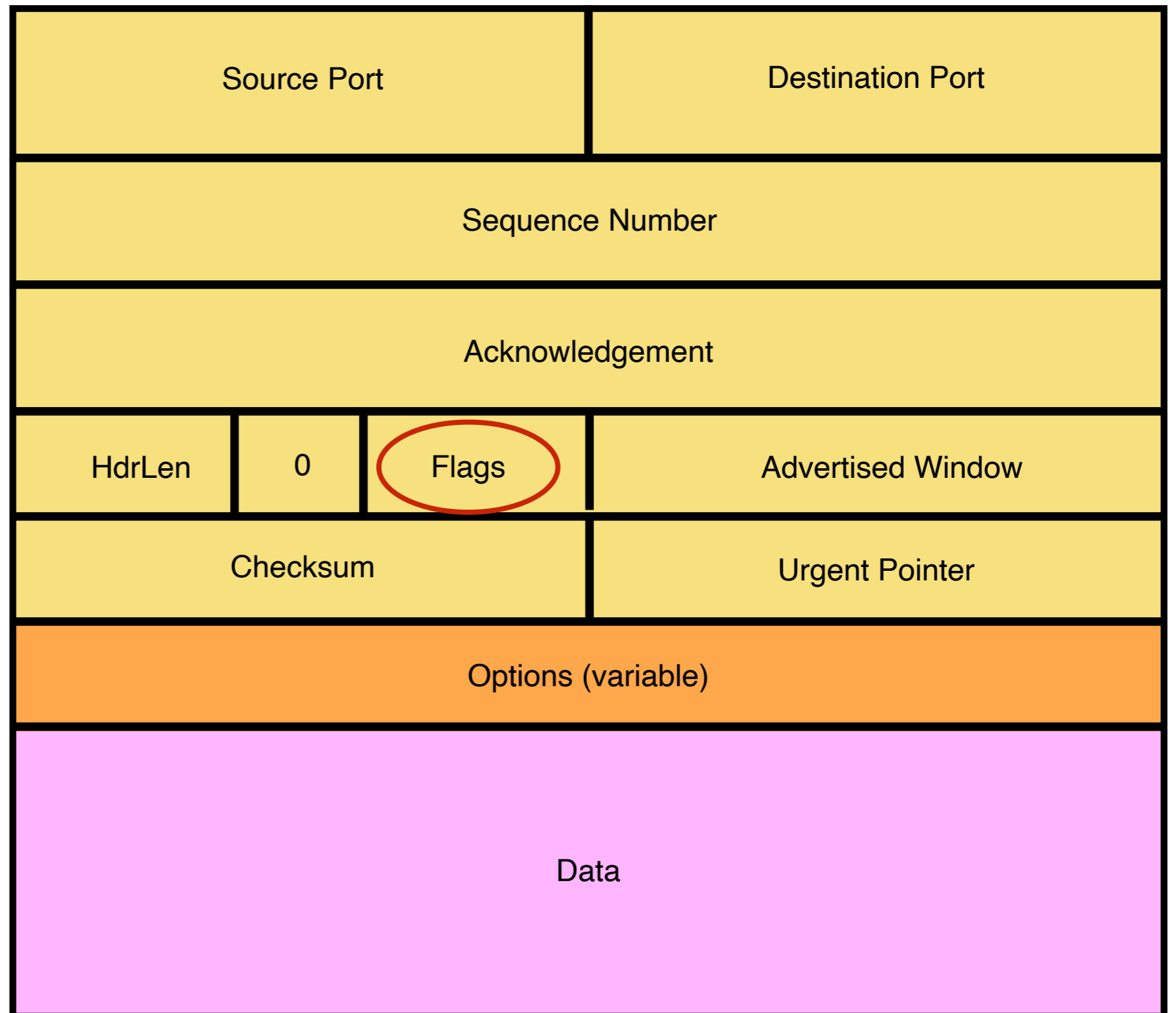
Used with URG flag to indicate urgent data (not discussed further)



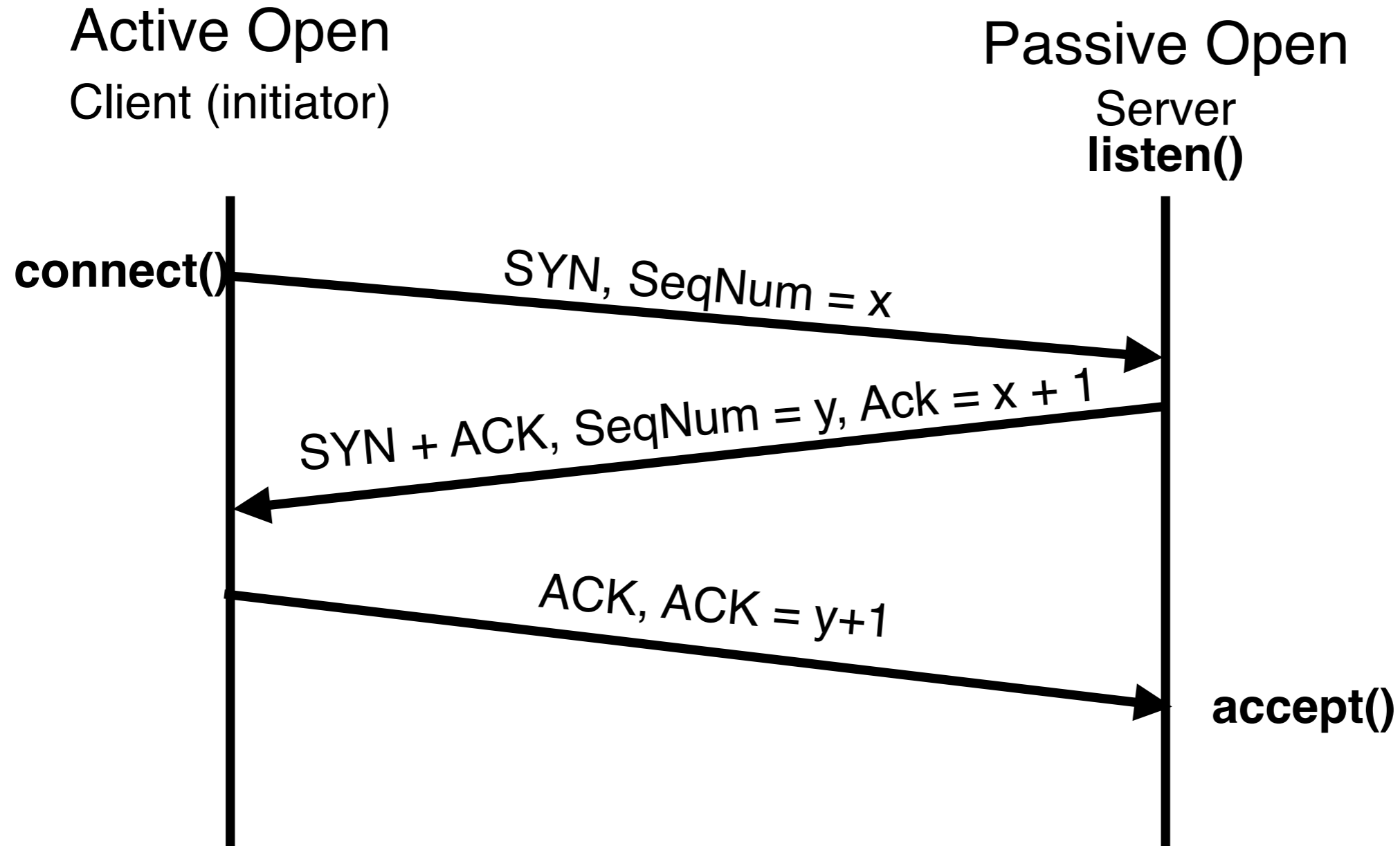
# Implementing Sliding Window

- Sender maintains a window
  - Data that has been sent out but not yet ACK'ed
- Left edge of window:
  - Beginning of unacknowledged data
  - Moves when data is ACKed
- Window size = maximum amount of data in flight
- Receiver sets this amount, based on its available buffer space
  - If it has not yet sent data up to the app, this might be small

# TCP Header: What's left?



# Timing Diagram: 3-Way Handshaking



## Note: TCP is Duplex

- A TCP connection between A and B can carry data in both directions
- Packets can both carry data and ACK data
- If the ACK flag is set, then it is ACKing data
- (details to follow ...)

# What if the SYN Packet Gets Lost?

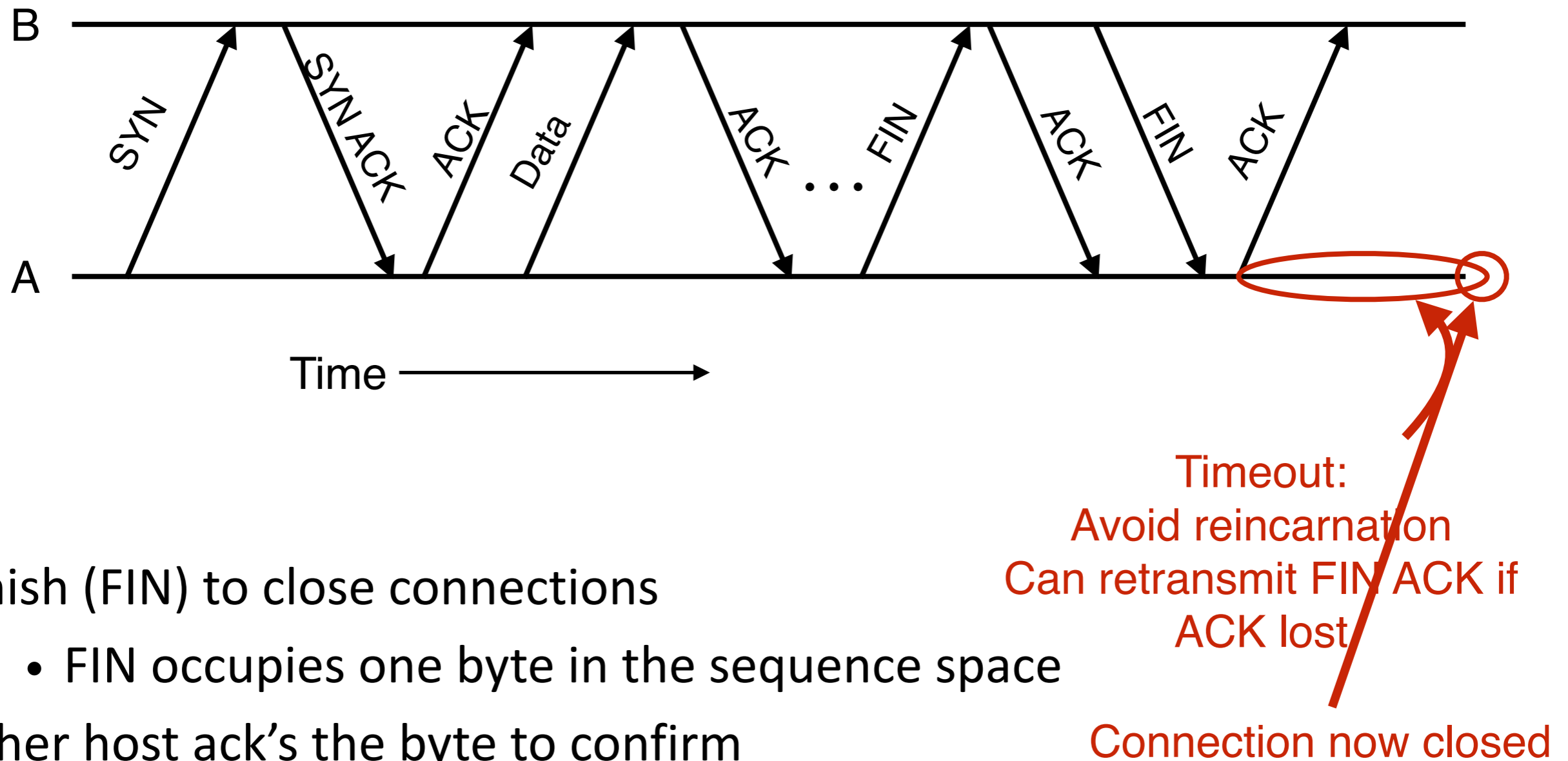
- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or
  - Server **discards** the packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
  - Sender sets a **timer** and **waits** for the SYN-ACK
  - ... and retransmits the SYN if needed
- How should the TCP sender 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 and 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...
  - 3-4 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

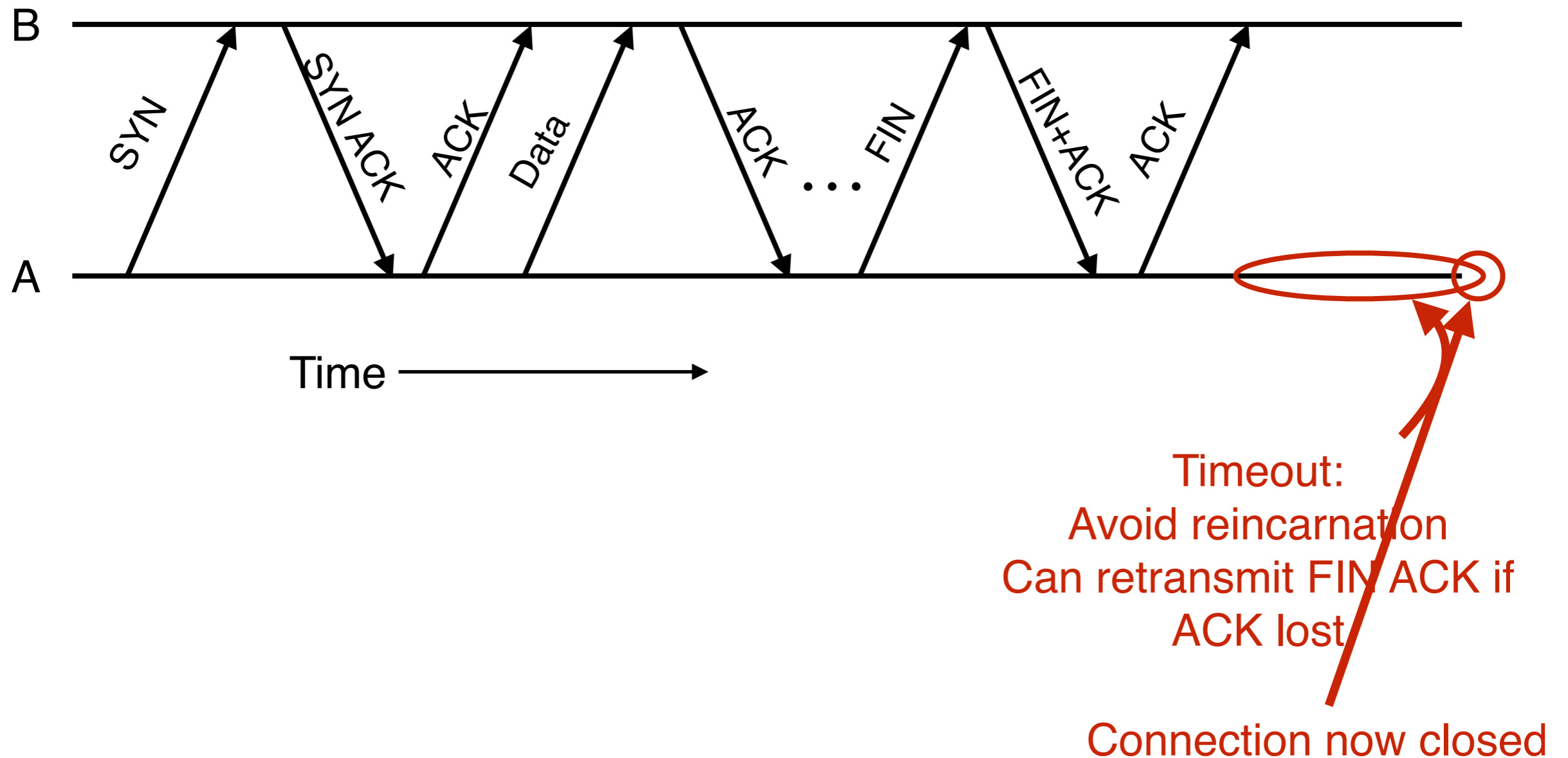
# Normal Termination



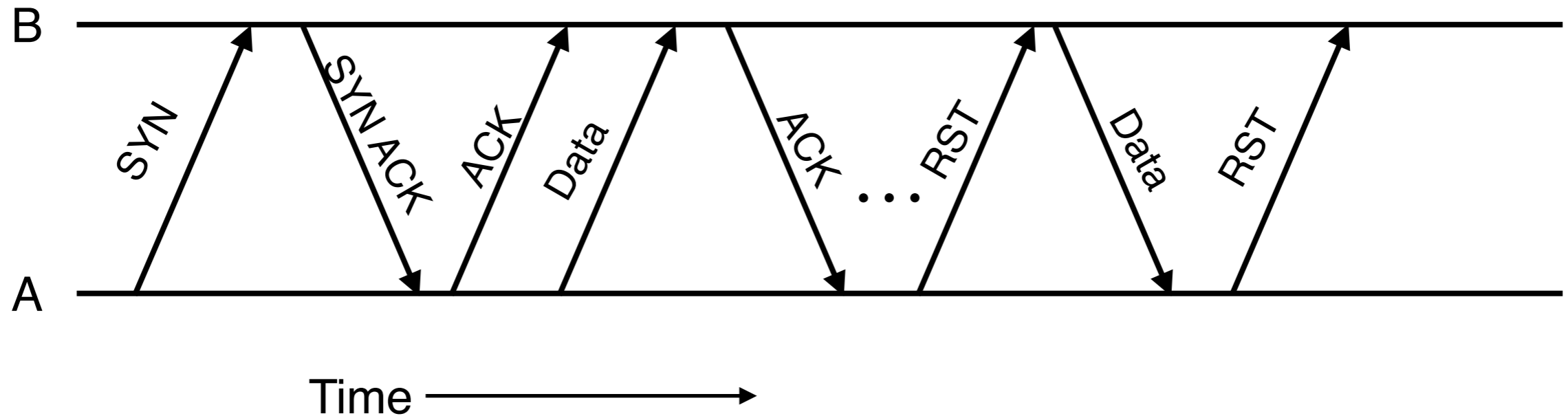
- Finish (FIN) to close connections
  - FIN occupies one byte in the sequence space
- Other host ack's the byte to confirm
- Closes A's side of connection, but not B's
  - Until B likewise sends a FIN
  - Which A then acks

# Normal Termination, Both Together

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

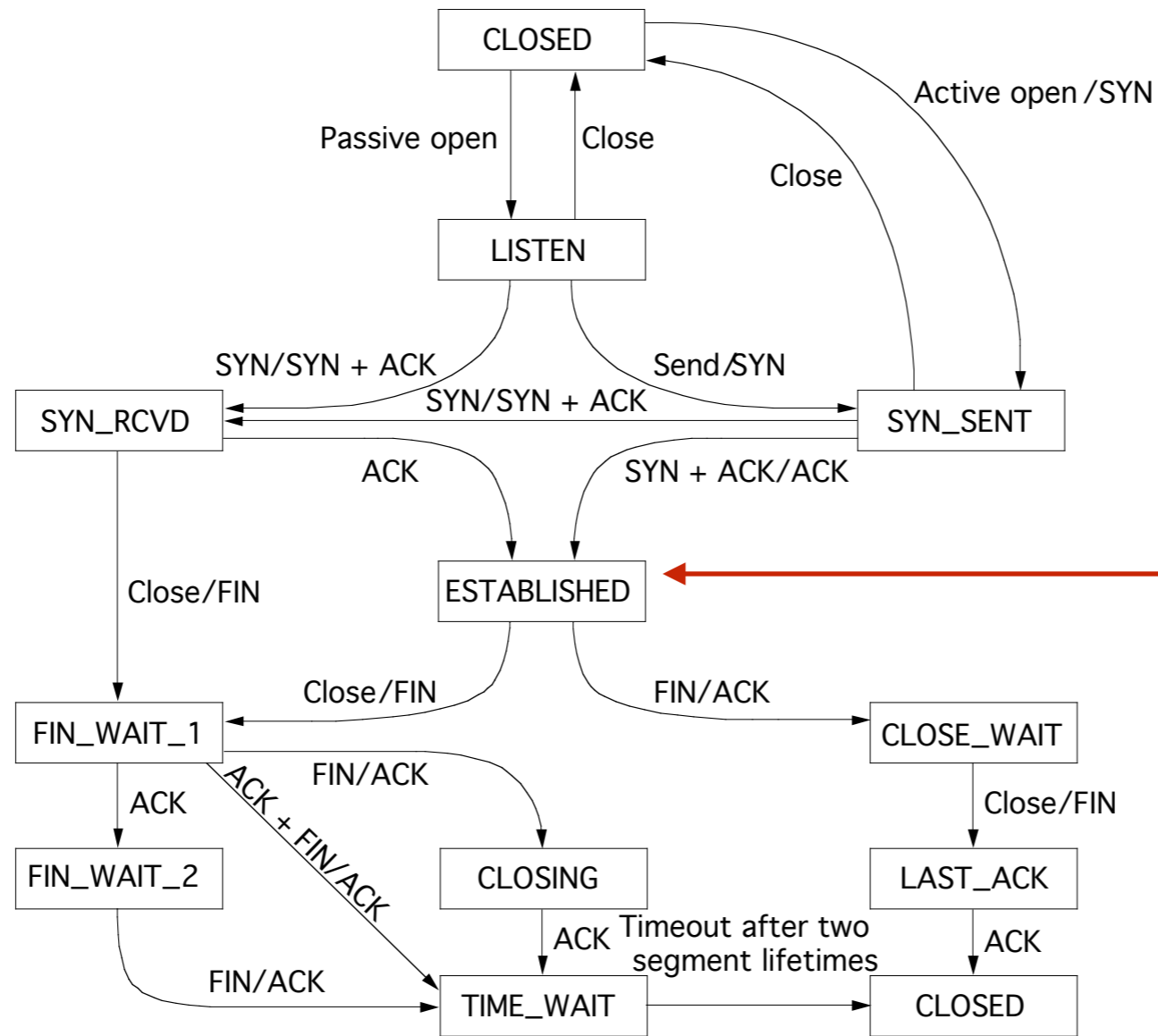


# Abrupt Termination



- A sends a RESET (RST) to B
  - E.g., because app. Process on A crashed
- That's it
  - B does not ack the RST
  - This, 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

# A Simpler View of the Client Side

