CS4450

Computer Networks:
Architecture and Protocols

Lecture 23
Reliable Transport and TCP

Rachit Agarwal

Goal of Today's Lecture

- Continue our understanding of reliable transport **conceptually**
- Understanding TCP will become infinitely easier
    - TCP involves lots of detailed mechanisms
    - **Knowing WHY TCP uses these mechanisms is most important**

Lets start with recapping last lecture

Recap: Best Effort Service (L3)

- Packets can be **lost**
- Packets can be **corrupted**
- Packets can be **reordered**
- Packets can be **delayed**
- Packets can be **duplicated**
- …

Transport layer:
Enabling reliability over such a best-effort service model

Recap: Complete Correctness Condition for reliability

A transport mechanism is "reliable" if and only if

(a) It resends all dropped or corrupted packets

(b) It attempts to make progress

Recap: Four Goals for Reliable Transfer

- **Correctness**
  - As defined in the last slide
- **“Fairness”**
  - Every flow must get a fair share of network resources
- **Flow Performance (Latency-related)**
  - Latency, jitter, etc.
- **Utilization (Throughput-related)**
  - Would like to maximize bandwidth utilization
  - If network has bandwidth available, flows should be able to use it!

Recap: Solution v1

- **Send every packet as often and fast as possible…**

- Not correct
  - **if** condition **not** satisfied: Transport must **attempt to make progress**
  - No way to check whether the packet was dropped or corrupted
    - So, must continue sending the same packet

- **What did we learn from this incorrect solution?**
  - why we need receiver feedback

Recap: Solution v2

- **Resend packet until you get an ACK**
  - **And receiver sends per-packet ACKs until data finally stops**

- Correct, fair, good (but suboptimal) latency, suboptimal utilization
  - A specific kind of under-utilization:
    - The source is unnecessarily sending the same packet

- **What did we learn from this solution?**
  - why we must wait for an ACK after sending a packet
    - But how long shall we wait for an ACK?
    - Indeed, the ACK may be lost as well

Recap: Solution v3

- Send packet
  - But now, set a timer
- receiver sends per-packet ACKs
- If sender receives ACK, done
- If no ACK when timer expires, resend

- Correct, fair, good (but suboptimal latency and utilization)
  - A different kind of under-utilization
    - source is not “work conserving”: could send, but is not

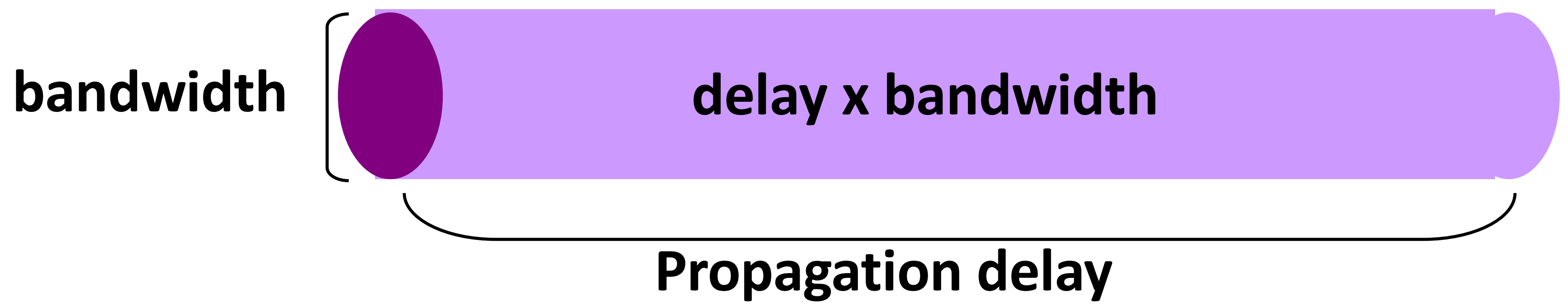
- **What did we learn from this solution?**
  - We should not be just waiting; sender-side bandwidth wasted
    - Keep more than one packet “in flight”
    - How many?

Recap: Window-based Algorithms

- Very simple concept
  - Send W packets
  - When one gets ACK'ed send the next packet in line
- **We want to set W such that:**
  - if I am sending at rate = link bandwidth, then
  - the ACK of the first packet arrives
  - exactly when I just finish sending the last of my W packets
  - **(assuming same transmission time for data and ACK packets)**
- **Lets me send as fast as the path can deliver…**

RTT x B ~ W x Packet Size

- Recall that **Bandwidth Delay Product**
  - BDP = bandwidth x propagation delay



- **B x RTT is merely 2x BDP**

- Window sizing rule:
  - Total bits in flight is roughly the amount of data that fits into forward and reverse "pipes"
    - Here pipe is complete path, not single link…
    - **This is not "detail", this is a fundamental concept…**

Where Are We?

- **Figured out correctness condition:**
  - Always resend lost/corrupted packets
  - Always try to make progress (but can give up entirely)

- **Figured out single packet case:**
  - Send packet, set timer, resend if no ACK when timer expires

- **Some progress towards multiple packet case:**
  - Allow many packets (W) in flight at once
  - And know what the ideal window size is
    - RTT x B / Packet size

- What's left to design?

Questions?

Three Design Considerations

- Nature of feedback
    - What should ACKs tell us when we have many packets in flight
- Detection of loss
- Response to loss

ACK Individual Packets

The receiver sends ACK for each individual packet that it receives

Example:

- Assume that packet 5 is lost, but no others
- Stream of ACKs will be
    - 1
    - 2
    - 3
    - 4
    - **6**
    - **7**
    - **8**
    - …

ACK Individual Packets

- **Nature of feedback**: simple - the receiver ACKs each packet
- **Loss detection:** simple - ACKs tell the fate of each packet to the source
- **Response to loss: moderate**:
  - + Retransmit the packet for which ACK not received
  - + Reordering not a problem
  - + Simple window algorithm
    - W independent single packet algorithms
    - When one finishes grab next packet
  - - **Loss of ACK packet requires a retransmission**

Full Information Feedback

- **List all packets that have been received**
  - Give highest cumulative ACK plus any additional packets

Same Example (suppose packet 5 gets lost):

- Same story, except that the “hole” is explicit in each ACK
- Stream of ACKs will be
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4, plus 6
  - Up to 4, plus 6,7
  - Up to 4, plus 6,7,8
  - …

Full Information Feedback

- **Nature of feedback**: **complex** - feedback may have high overheads
    - If packets 1, 5, 6, …., 100 received: ACK(1, 5, 6, …,100)
- **Loss detection:** simple - the source still knows fate of each packet
- **Response to loss:** simple:
    - + Retransmit the packet for which ACK not received
    - + Reordering not a problem
    - + Simple window algorithm
    - + **Loss of ACK does not necessarily requires a retransmission**
        - **The next ACK will tell that the packet was indeed received**
        - Resilient form of individual ACKs

Cumulative ACK

- **Individual ACKs** can get lost, and require **unnecessary retransmission**
- **Full information feedback** can handle lost ACKs but has **high overheads**
- **Cumulative ACKs: a sweet spot between the two**
- Just the first part of full information feedback
- ACK the highest sequence number for all previously received packets

Cumulative ACKs (same example; say packet 5 lost)

Full information feedback:

- Stream of ACKs will be
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4, plus 6
  - Up to 4, plus 6,7
  - Up to 4, plus 6,7,8
  - …

Tells **“which” packet arrived**, and **which packet did not**

Cumulative ACKs:

- Stream of ACKs will be
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4
  - Up to 4
  - Up to 4
  - …

Tells **“some” packet arrived,** and **which packet did not**

Loss With Cumulative ACKs (cont'd)

- Duplicate ACKs are a sign of loss
  - The lack of ACK progress means 5 hasn't been delivered
  - Stream of duplicate ACKs means some packets are being delivered (one for each subsequent packet)
- Response to loss is trickier… When shall the source retransmit packet 5?
  - Packet may be delayed (so, source should wait)
  - Packet may be reordered (so, source should wait)
  - Or, packet may be dropped (source should immediately retransmit)
  - Impossible to know which one is the case
    - Life lesson: **be optimistic**!
    - Until optimism starts hurting
  - **Solution: retransmit after k duplicate ACKs**
    - **for some value of k, depending on how optimistic you feel!**

Cumulative ACKs (how is reordering handled; large k)

Receiver events:

- Packet 1 received
- Packet 2 received
- Packet 3 received
- Packet 4 received
- **Packet 6 received**
- Packet 7 received
- **Packet 5 received**
- **Packet 8 received**
- …

Cumulative ACKs:

- Up to 1
- Up to 2
- Up to 3
- Up to 4
- **Up to 4**
- Up to 4
- **Up to 7**
- **Up to 8**
- …

Cumulative ACKs **naturally handle packet reordering**

(Packet delays are similar to reordering)

Cumulative ACKs (confusion with duplication)

- Produce duplicate ACKs
  - Could be confused for loss with cumulative ACKs
  - But duplication is rare…

Source events:	**Receiver events:**	**Cumulative ACKs:**
Packet 1 sent	Packet 1 received	Up to 1
Packet 2 sent	Packet 2 received	Up to 2
Packet 3 sent	**Packet 4 received**	**Up to 2**
Packet 4 sent	Packet 5 received	**Up to 2**
Packet 5 sent	Packet 6 received	**Up to 2**
Packet 6 sent	**Packet 3 received**	Up to 6
Packet 3 resent	**Packet 3 received**	**Up to 6**
Packet 7 sent	Packet 7 received	**Up to 7**
…	…	…

Possible Design For Reliable Transport

- Cumulative ACKs
- Window based, with retransmissions after
  - Timeout
  - k subsequent ACKs
- This is correct, high-performant and high-utilization
  - At least as much as we can efficiently
- How about fairness?

Fairness? (Come back to later)

- The question of fairness comes up when:
  - Senders want to send data at rate higher than bandwidth
  - There will be packet loss!
- Adjust W based on losses…
- In a way that flows receive same shares
- Short version:
  - Loss: cut W by 2
  - Successful receipt of window: W increased by 1

Overview of Reliable Transport

- Window based self control separate concerns
  - Size of W
  - Nature of feedback
  - Response to loss
- Can design each aspect relatively independently
- Can be correct, fair, high-performant and high-utilization
- All of these are important concerns
  - **But correctness is most fundamental**
- Design **must** start with correctness
  - Can then “engineer” its performance with various hacks
  - These hacks can be “fun”, but don’t let them distract you

What Have We Done so far?

- Started from first principles
    - Correctness condition for reliable transport
- … to understanding **why feedback from receiver is necessary** (sol-v1)
- … to understanding **why timers may be needed** (sol-v2)
- … to understanding **why window-based design may be needed** (sol-v3)
- … to understanding **why cumulative ACKs may be a good idea**
    - Very close to modern TCP
- **You are now ready to learn TCP**

Lets learn TCP

Transport layer

- Transport layer offer a “pipe” abstraction to applications
- Data goes in one end of the pipe and emerges from other
- **Pipes are between processes, not hosts**
- There are two basic pipe abstractions

Two Pipe Abstractions

- **Unreliable packet** delivery (UDP)
    - Unreliable (application responsible for resending)
    - Messages limited to single packet
- **Reliable byte stream** delivery
    - Bytes inserted into pipe by sender
    - They emerge, in order at receiver (to the app)
- What features must transport protocol implement to support these abstractions?

UDP (Datagram Messaging Service)

- Sources send packets
- **Destinations do nothing**, but receive packets
- If packets delayed/reordered/lost:
    - Meh!
    - Let application handle packet loss (or be oblivious to drops)
    - If application needs reliable delivery, it must use reliable transport
- Discarding corrupted packets (optional)
- Nothing else!
- A minimal extension of IP

Transmission Control Protocol (TCP)

- Full duplex stream of **byte service**
  - **Sends and receives stream of bytes (segments), not messages**
- **Reliable, in-order delivery**
  - Ensures byte stream (eventually) arrives intact
  - In the presence of corruption, delays, reordering, loss

From design to implementation: major notation change

- Previously we focused on packets
    - Packets had numbers
    - ACKs referred to those numbers
    - Window sizes expressed in terms of # of packets
- TCP focuses on bytes, thus
    - Packets identified by the bytes they carry
    - ACKs refer to the bytes received
    - Window size expressed in terms of # of bytes

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

Connection/Session

Connections (Or Sessions)

- Reliability requires keeping state
  - Sender: packets sent but not yet ACKed, and related timers
  - Receiver: packets that arrived out-of-order

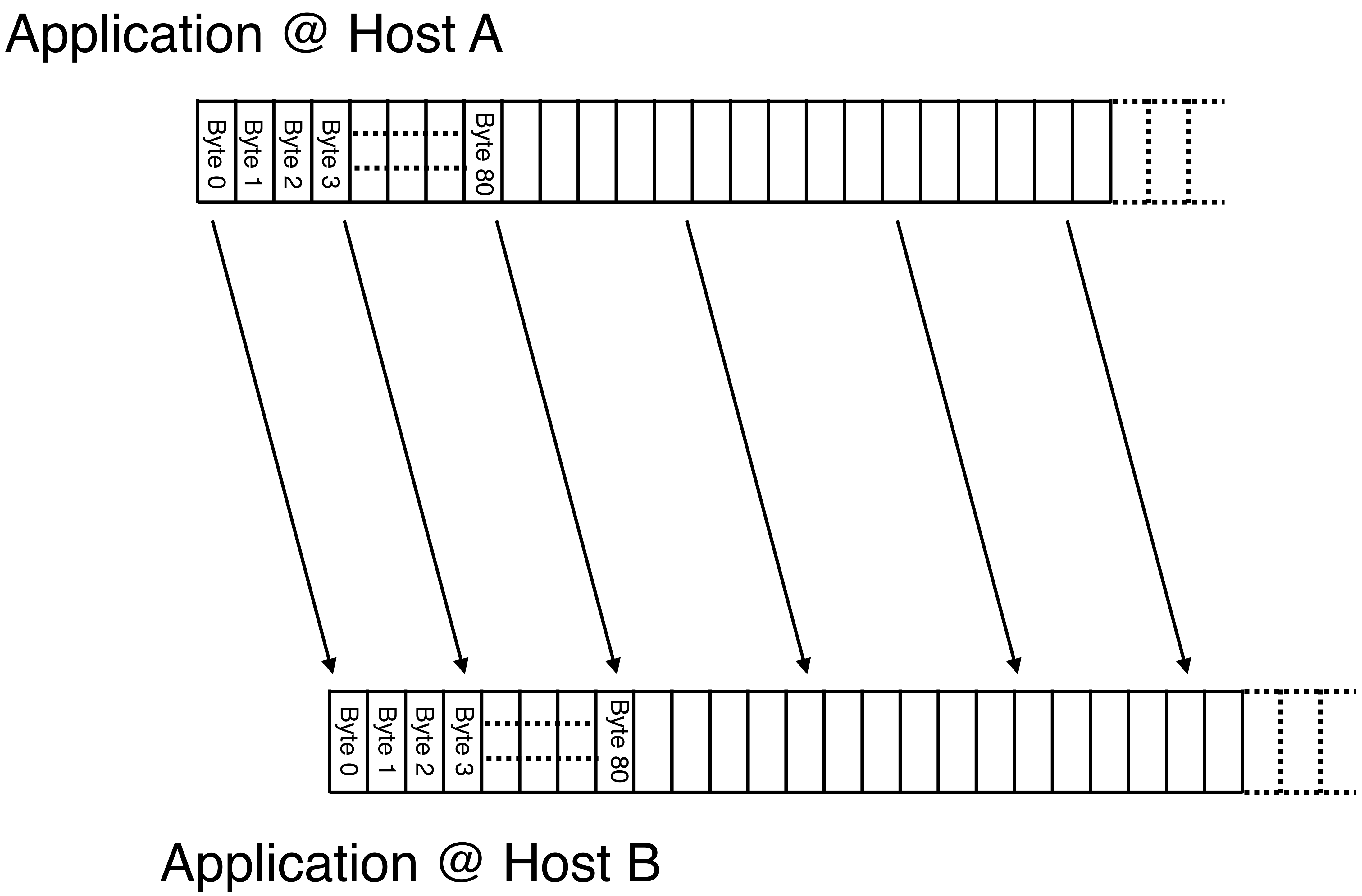
- Each byte stream is called a **connection** or **session**
  - Each with their own connection state
  - State is in hosts, not network

Basic Components of TCP

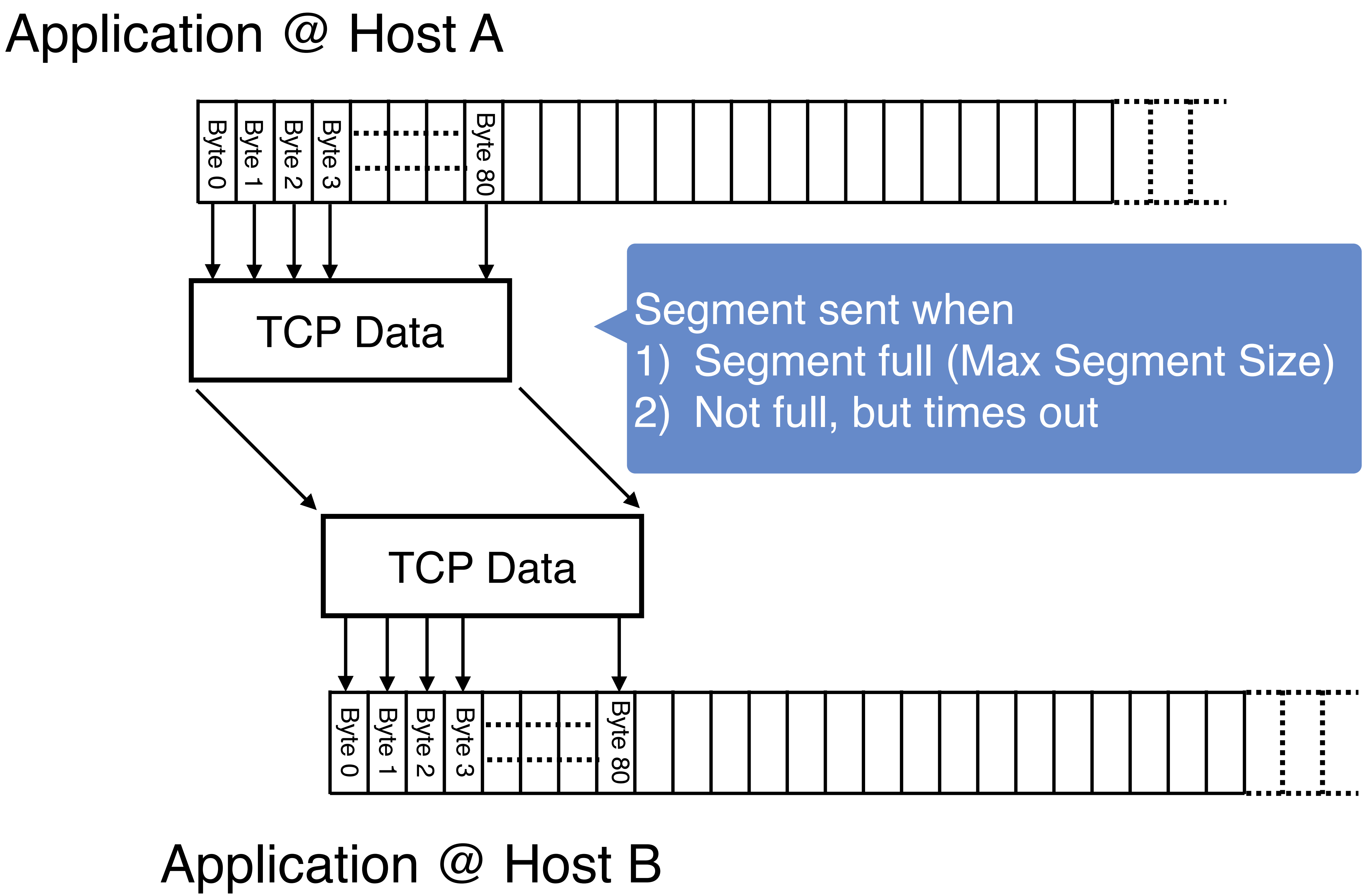
- **Connections:** Explicit set-up and tear-down of TCP sessions/connections
- **Segments, Sequence numbers, ACKs**
  - TCP uses byte sequence numbers to identify payloads
  - ACKs referred to sequence numbers
  - Window sizes expressed in terms of # of bytes
- **Retransmissions**
  - Can’t be correct 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

Segments and Sequence Numbers

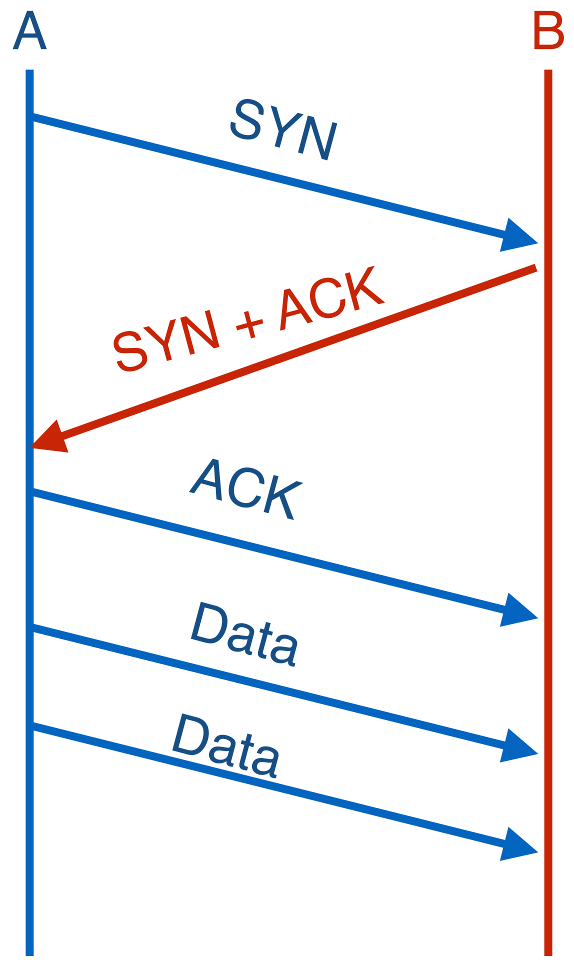
TCP “Stream of Bytes” Service



TCP "Stream of Bytes" Service



Establishing a TCP Connection



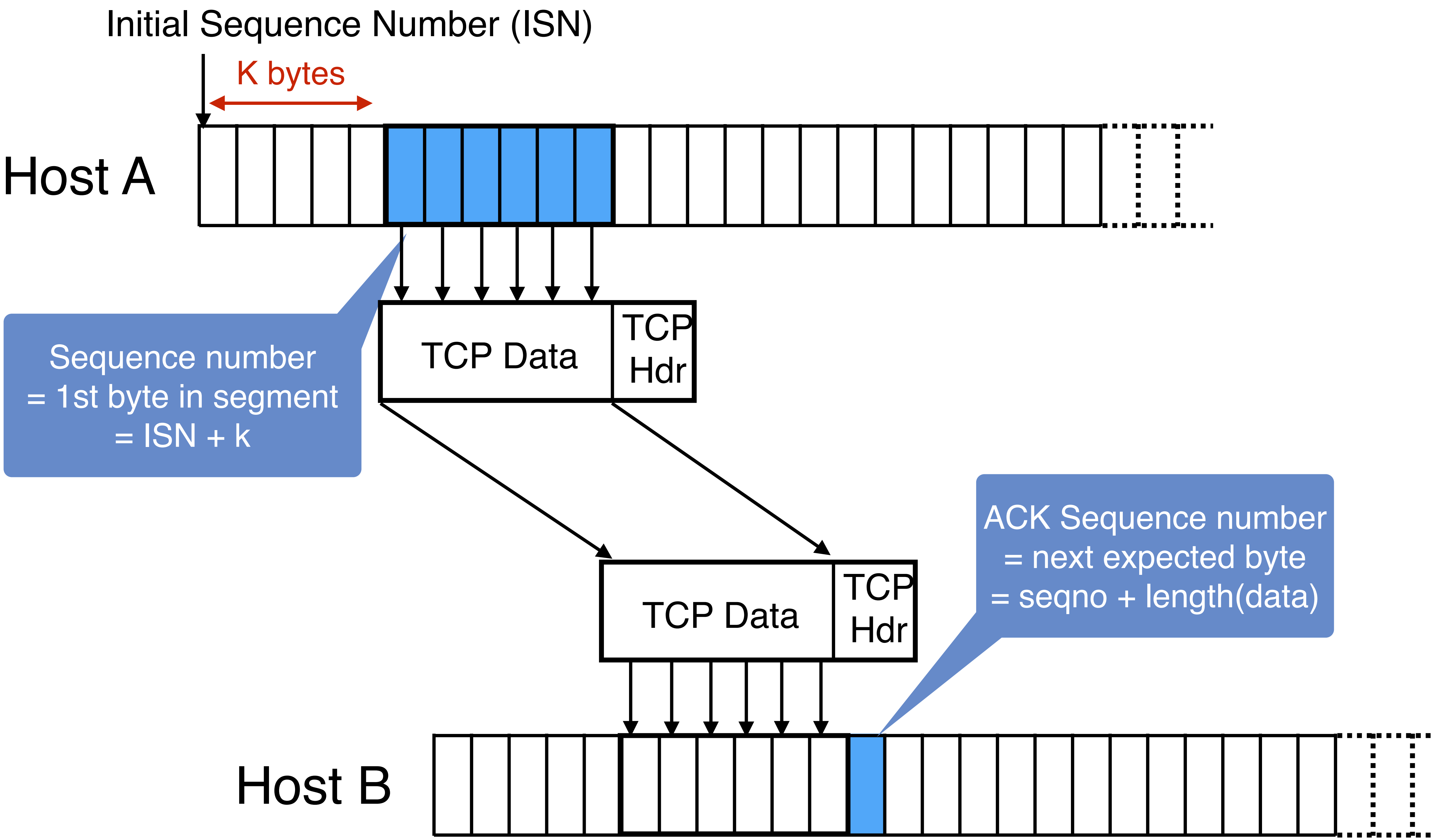
Each host tells its ISN to the other host.

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

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?

Sequence Numbers



ACKing and Sequence Numbers

- Sender sends segments (byte stream)
  - Data starts with sequence number X
  - Packet contains B bytes
    - X, X+1, X+2, …, X+B-1
- Upon receipt of a segment, 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

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

TCP Retransmission

Two Mechanisms for Retransmissions

- Duplicate ACKs
- Timeouts

Loss with Cumulative ACKs

- Sender sends packets with 100B and seqnos
  - 100, 200, 300, 400, 500, 600, 700, 800, 900
- Assume 5th packet (seqno 500) is lost, but no others
- Stream of ACKs will be
  - 200, 300, 400, 500, 500, 500, 500, 500

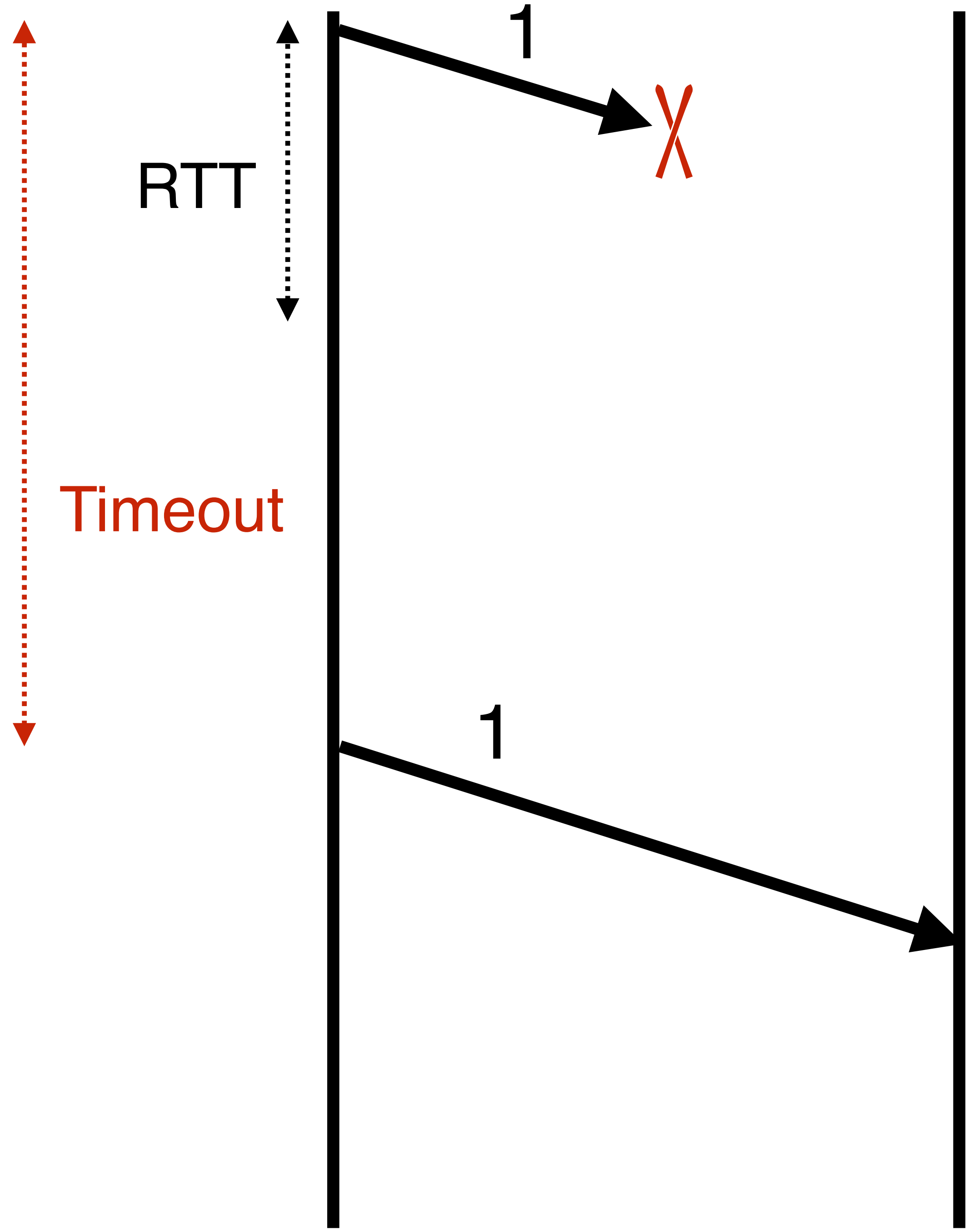
Loss with Cumulative ACKs

- 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

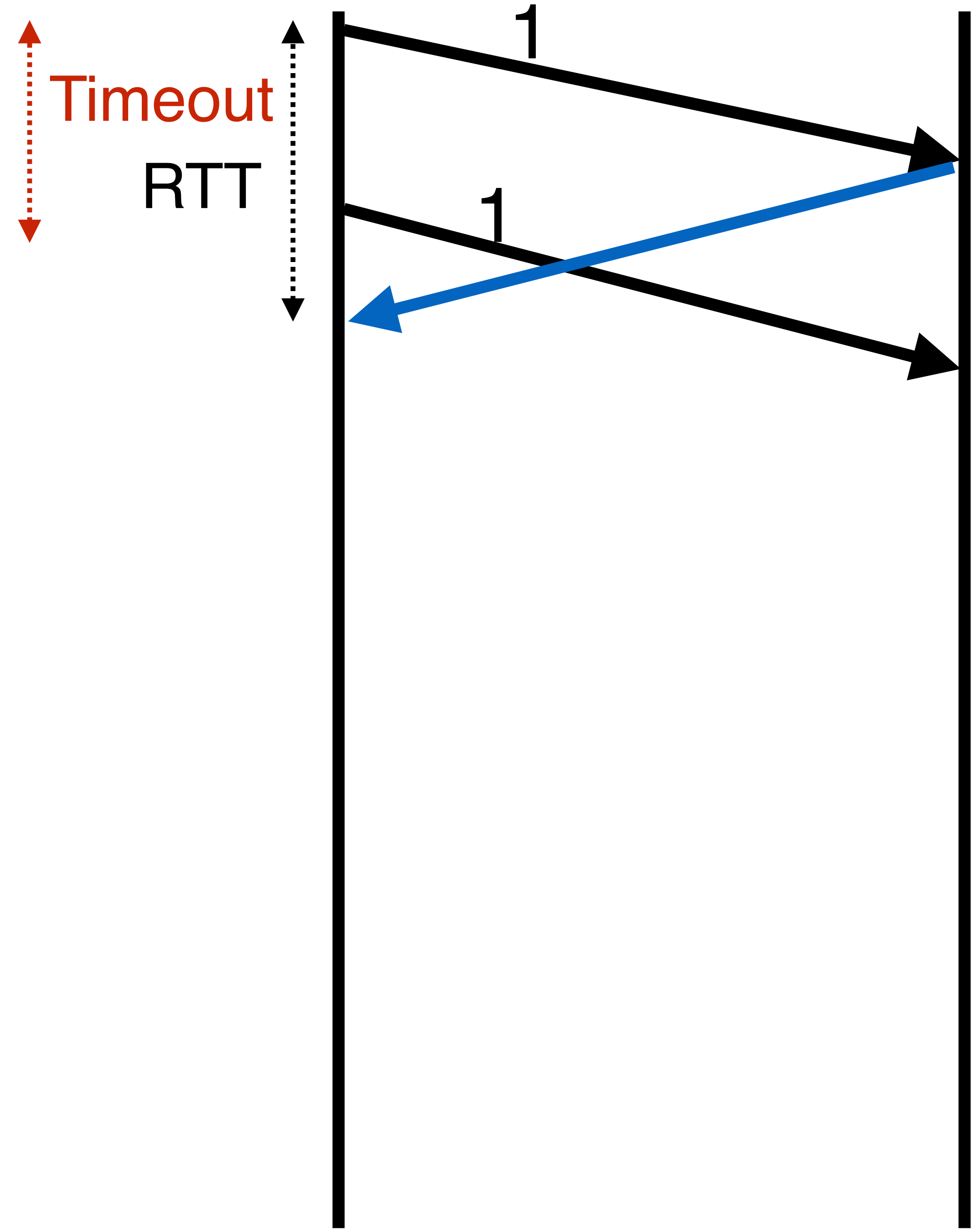
Timeouts and Retransmissions

- Reliability requires retransmitting lost data
- Involves setting timers and retransmitting on timeouts
- TCP only has a single timer
- TCP resets timer whenever new data is ACKed
- Retransmit packet containing “next byte” when timer expires
- RTO (Retransmit Time Out) is the basic timeout value

Setting the Timeout Value (RTO)



Timeout too long -> inefficient



Timeout too short -> duplicate packets

Setting RTO value

- Many ideas
  - See backup slides for some examples (not needed for exams)
- Implementations often use a coarser-grained timer
  - 500 msec is typical
- **Incurring a timeout is expensive**
- So we rely on duplicate ACKs

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

TCP Flow Control

Flow Control (Sliding Window)

- 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
- In original TCP, that was the sole protocol mechanism controlling sender's rate
- What's missing?
- **Congestion control** about how to adjust W to avoid network congestion

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

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- In original TCP, that was the sole protocol mechanism controlling sender's rate
- What's missing?
- **Congestion control** about how to adjust W to avoid network congestion

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

TCP Congestion Control

TCP congestion control: high-level idea

- End hosts adjust sending rate
- Based on implicit feedback from the network
  - Implicit: router drops packets because its buffer overflows, not because it's trying to send message
- Hosts probe network to test level of congestion
  - Speed up when no congestion (i.e., no packet drops)
  - Slow down when when congestion (i.e., packet drops)
- How to do this efficiently?
  - Extend TCP's existing window-based protocol…
  - Adapt the window size based in response to congestion

All These Windows…

- **Flow control window**: Advertised Window (RWND)
    - How many bytes can be sent without overflowing receivers buffers
    - Determined by the receiver and reported to the sender
- **Congestion Window** (CWND)
    - How many bytes can be sent without overflowing routers
    - Computed by the sender using congestion control algorithm
- **Sender-side window** = minimum{CWND,RWND}
    - Assume for this lecture that RWND >> CWND

Note

- This lecture will talk about CWND in units of MSS
  - Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet
  - This is only for pedagogical purposes
- Keep in mind that real implementations maintain CWND in bytes

Basics of TCP Congestion

- Congestion Window (CWND)
  - Maximum # of unacknowledged bytes to have in flight
  - Rate ~CWND/RTT
- Adapting the congestion window
  - Increase upon lack of congestion: optimistic exploration
  - Decrease upon detecting congestion
- But how do you detect congestion?

Not All Losses the Same

- **Duplicate ACKs: isolated loss**
  - Still getting ACKs
- **Timeout: possible disaster**
  - Not enough duplicate ACKs
  - Must have suffered several losses

How to Adjust CWND?

- Consequences of over-sized window much worse than having an under-sized window
  - Over-sized window: packets dropped and retransmitted
  - Under-sized window: somewhat lower throughput
- Approach
  - Gentle increase when un-congested (exploration)
  - Rapid decrease when congested

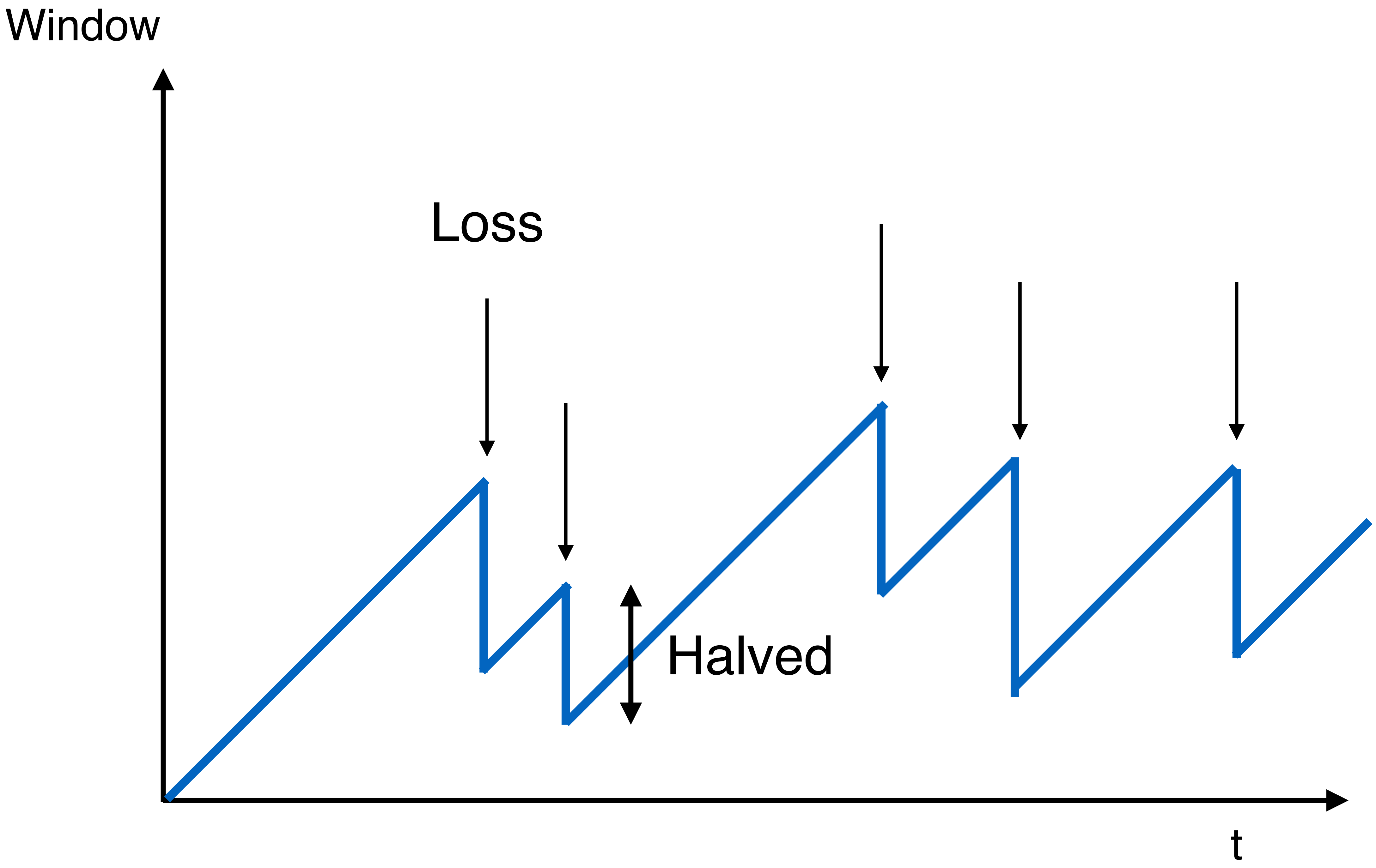
Additive Increase, Multiplicative Decrease (AIMD)

- Additive increase
  - On success of last window of data, increase by one MSS
  - If W packets in a row have been ACKed, increase W by one
  - i.e., +1/W per ACK
- Multiplicative decrease
  - On loss of packets by DupACKs, divide congestion window by half
  - Special case: when timeout, reduce congestion window to one MSS

AIMD

- ACK: CWND -> CWND + 1/CWND
    - When CWND is measured in MSS
    - Note: after a full window, CWND increase by 1 MSS
    - Thus, **CWND increases by 1 MSS per RTT**
- 3rd DupACK: CWND -> CWND/2
- Special case of timeout: CWND -> 1 MSS

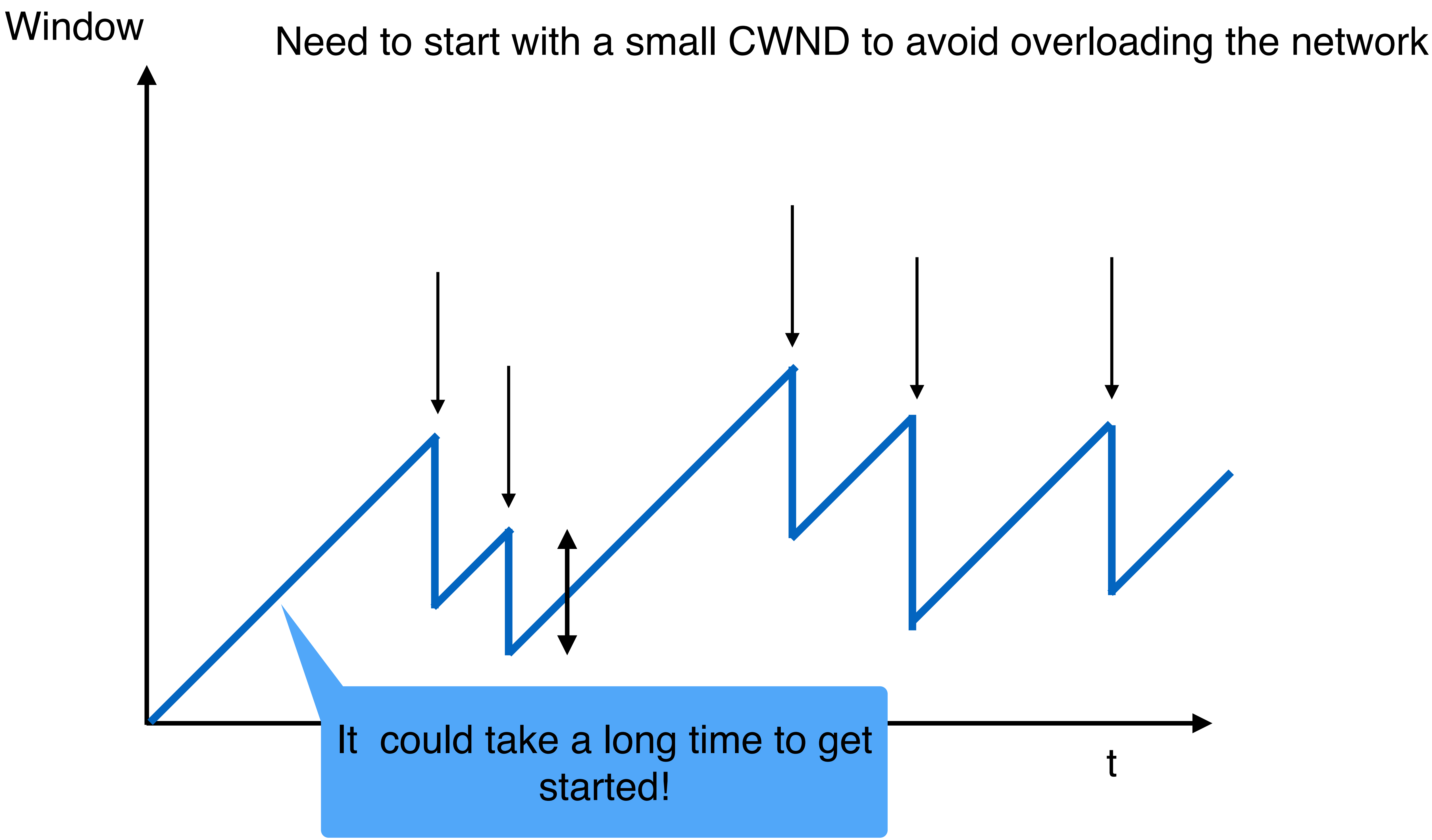
Leads to the TCP Sawtooth



Questions?

Slow Start

AIMD Starts Too Slowly



Bandwidth Discovery with Slow Start

- Goal: estimate available bandwidth
  - Start slow (for safety)
  - But ramp up quickly (for efficiency)

- Consider
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 MSS
  - Window size to fill 1 Gbps = 12,500 MSS
    - With just AIMD, it takes about 12500 RTTs to get to this window size!
    - ~21 mins

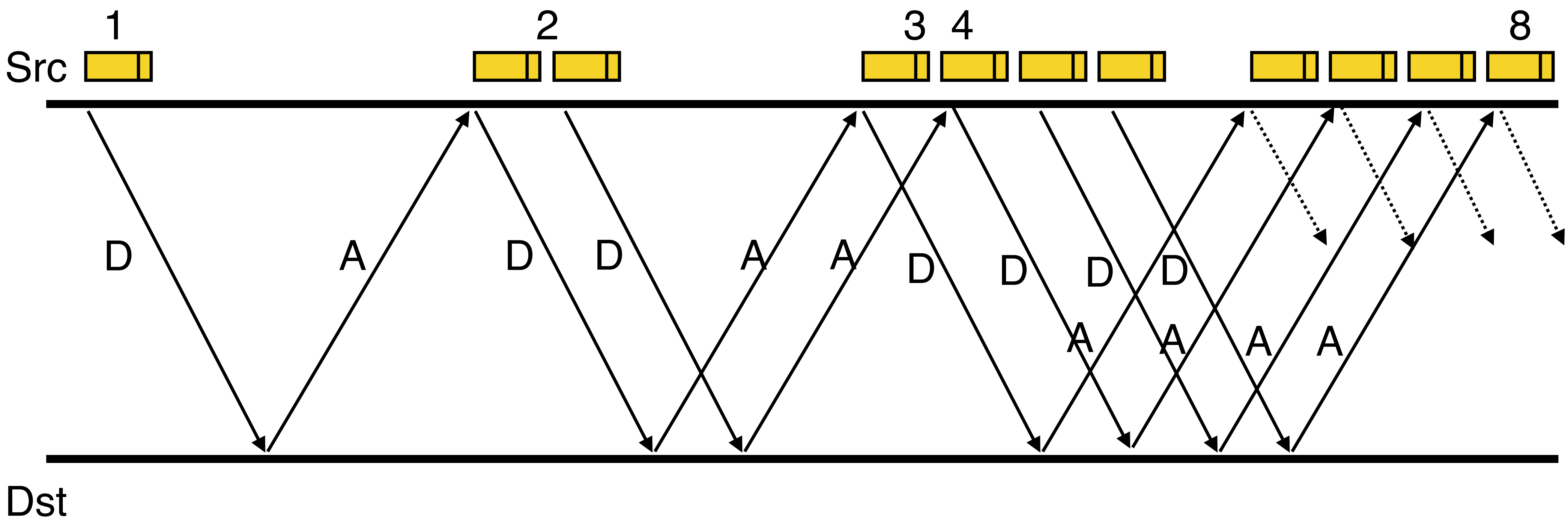
“Slow Start” Phase

- Start with a small congestion window
  - Initially, CWND is 1 MSS
  - So, initial sending rate is MSS/RTT
- That could be pretty wasteful
  - Might be much less than the actual bandwidth
  - Linear increase takes a long time to accelerate
- Slow-start phase (**actually “fast start”**)
  - Sender starts at a slow rate (hence the name)
  - … but increases exponentially until first loss

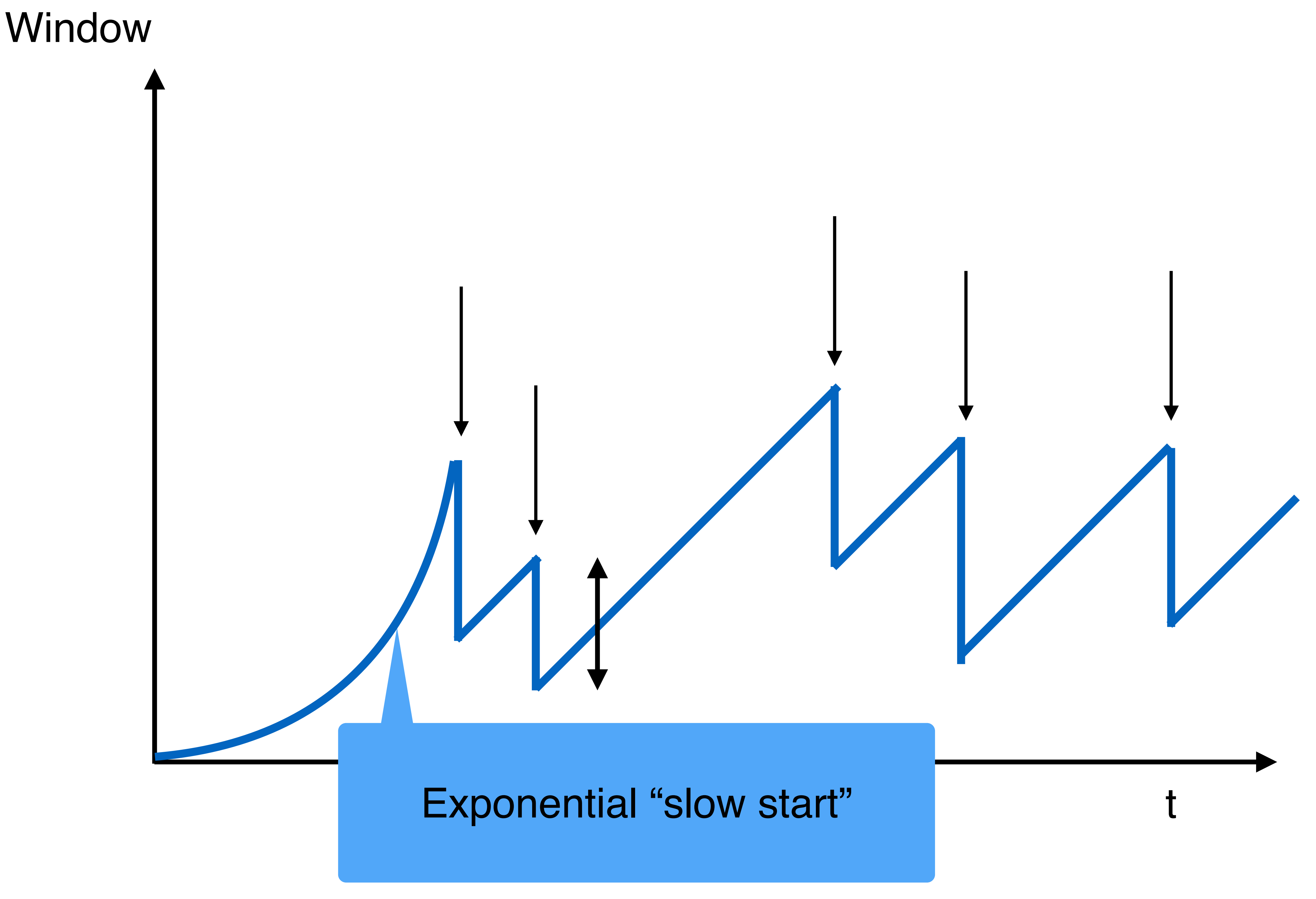
Slow Start in Action

Double CWND per round-trip time

Simple implementation: on each ACK, CWND += MSS



Slow Start and the TCP Sawtooth



Why is it called slow-start? Because TCP originally had no congestion control mechanism. The source would just start by sending a whole window’s worth of data.

Slow-Start vs AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Introduce a "slow start threshold" (ssthresh)
  - Initialized to a large value
  - On timeout, ssthresh = CWND/2
- When CWND > ssthresh, sender switches from slow-start to AIMD-style increase

Timeouts

Loss Detected by Timeout

- Sender starts a timer that runs for RTO seconds
- **Restart timer whenever ACK for new data arrives**
- If timer expires
    - Set SSHTHRESH <- CWND/2 (“Slow Start Threshold”)
    - Set CWND <- 1 (MSS)
    - Retransmit **first** lost packet
    - Execute Slow Start until CWND > SSTHRESH
    - After which switch to Additive Increase

Summary of Increase

- "Slow start": increase CWND by 1 (MSS) for each ACK
    - A factor of 2 per RTT
- Leave slow-start regime when either:
    - CWND > SSTHRESH
    - Packet drop detected by dupacks
- Enter AIMD regime
    - Increase by 1 (MSS) for each window's worth of ACKed data

Summary of Decrease

- Cut CWND half on loss detected by dupacks
  - **Fast retransmit to avoid overreacting**
- Cut CWND all the way to 1 (MSS) on **timeout**
  - Set ssthresh to CWND/2
- Never drop CWND below 1 (MSS)
  - Our correctness condition: always try to make progress

TCP Congestion Control Details

Implementation

- State at sender
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
  - dupACKcount
  - Timer, as before
- Events at sender
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout
- What about receiver? Just send ACKs upon arrival

Event: ACK (new data)

- If in slow start
  - CWND += 1

CWND packets per RTT
Hence after one RTT with
no drops:
CWND = 2 x CWND

Event: ACK (new data)

- If CWND <= ssthresh
  - CWND += 1
- Else
  - CWND = CWND + 1/CWND

Slow Start Phase

Congestion Avoidance Phase

(additive increase)

CWND packets per RTT
Hence after one RTT with no drops:
CWND = CWND + 1

Event: Timeout

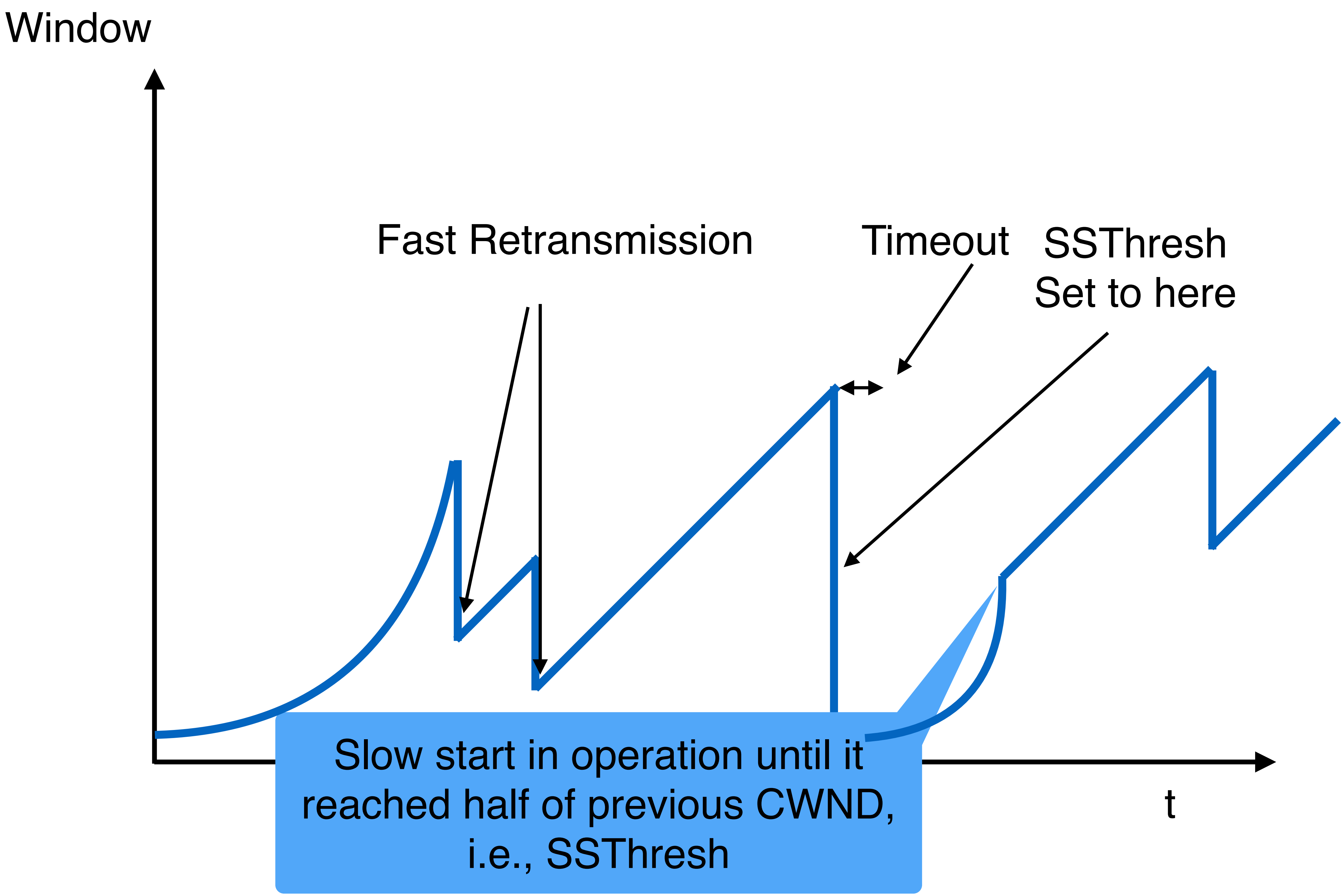
- On Timeout
    - ssthresh <- CWND/2
    - CWND <- 1

Event: dupACK

- dupACKcount++
- If dupACKcount = 3 /* fast retransmit */
    - ssthresh <- CWND/2
    - CWND <- CWND/2

Remains in congestion avoidance after fast retransmission

Time Diagram



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

TCP Flavors

- TCP Tahoe
  - CWND = 1 on triple dupACK
- TCP Reno
  - CWND = 1 on timeout
  - CWND = CWND/2 on triple dupACK
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - Incorporates selective acknowledgements

Our default assumption

TCP and fairness guarantees

Consider A Simple Model

- Flows **ask** for an amount of bandwidth r_i
  - In reality, this request is implicit (the amount they send)
- The link gives them an amount a_i
  - Again, this is implicit (by how much is forwarded)
  - $a_i <= r_i$
- There is some total capacity C
  - Sum $a_i <= C$

Fairness

- When all flows want the same rate, fair is easy
  - Fair share = C/N
  - C = capacity of link
  - N = number of flows
- Note:
  - This is fair share per link. This is not a global fair share
- When not all flows have the same demand?
  - What happens here?

Example 1

- Requests: r_i Allocations: a_i
- $C = 20$
  - Requests: $r_1 = 6$, $r_2 = 5$, $r_3 = 4$
- Solution
  - $a_1 = 6$, $a_2 = 5$, $a_3 = 4$
- When bandwidth is plentiful, everyone gets their request
- This is the easy case

Example 2

- Requests: r_i Allocations: a_i
- $C = 12$
  - Requests: $r_1 = 6$, $r_2 = 5$, $r_3 = 4$
- One solution
  - $a_1 = 4$, $a_2 = 4$, $a_3 = 4$
  - Everyone gets the same
- Why not proportional to their demands?
  - $a_i = (12/15)\ r_i$
- Asking for more gets you more!
  - Not incentive compatible (i.e., cheating works!)
  - You can't have that and invite innovation!

Example 3

- Requests: r_i Allocations: a_i
- $C = 14$
    - Requests: $r_1 = 6$, $r_2 = 5$, $r_3 = 4$
- $a_3 = 4$ (can't give more than a flow wants)
- Remaining bandwidth is 10, with demands 6 and 5
    - From previous example, if both want more than their share, they both get half
    - $a_1 = a_2 = 5$

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 $\mathrm{Sum}(a_i) = C$ 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

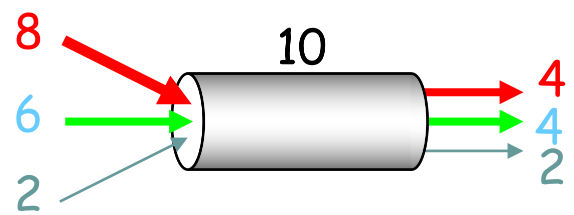
- Assume demands are in increasing order…
- If $C/N <= r_1$, then $a_i = C/N$ for all i
- Else, $a_1 = r1$, set $C = C - a_1$ and $N = N-1$
- Repeat
- Intuition: all flows requesting less than fair share get their request. Remaining flows divide equally

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: $f = 4$

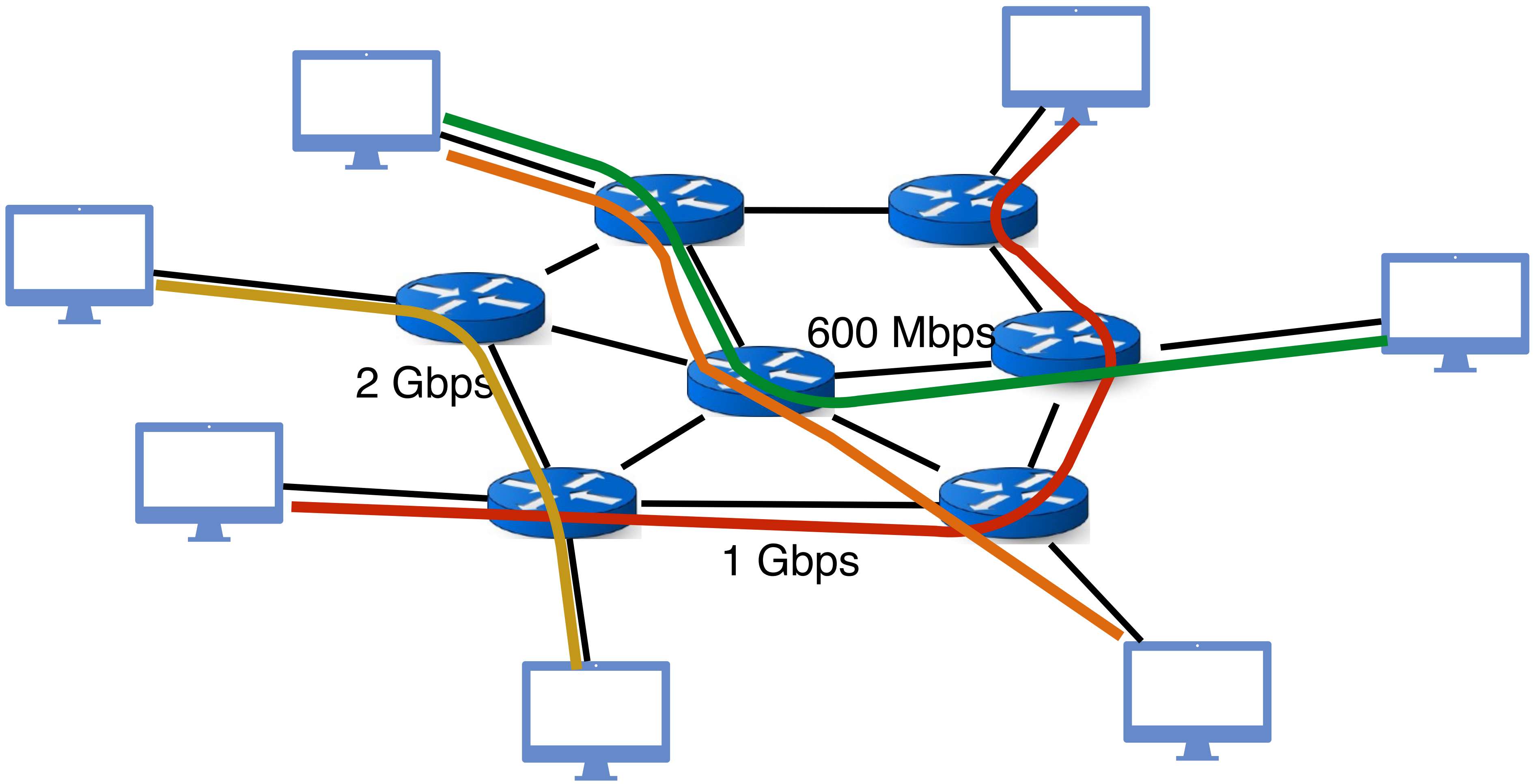


$f = 4$:
min(8, 4) = 4
min(6, 4) = 4
min(2, 4) = 2

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

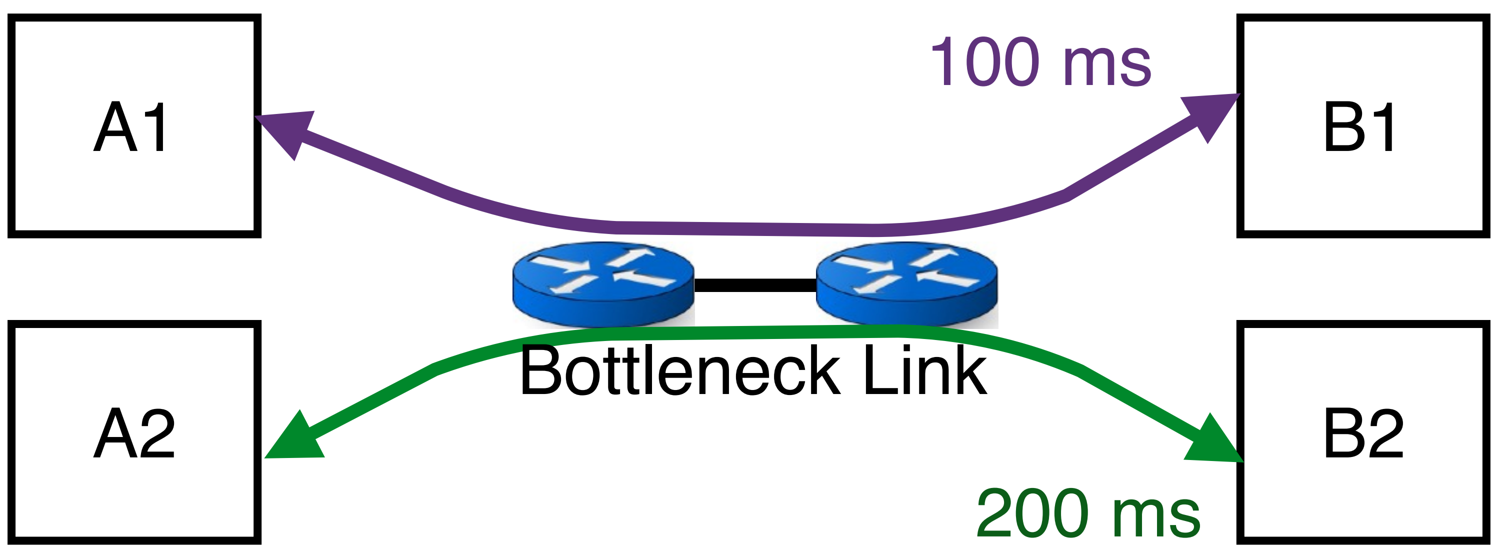
(3) Sawtooth Behavior Uneven

- 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 ~=~ 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?

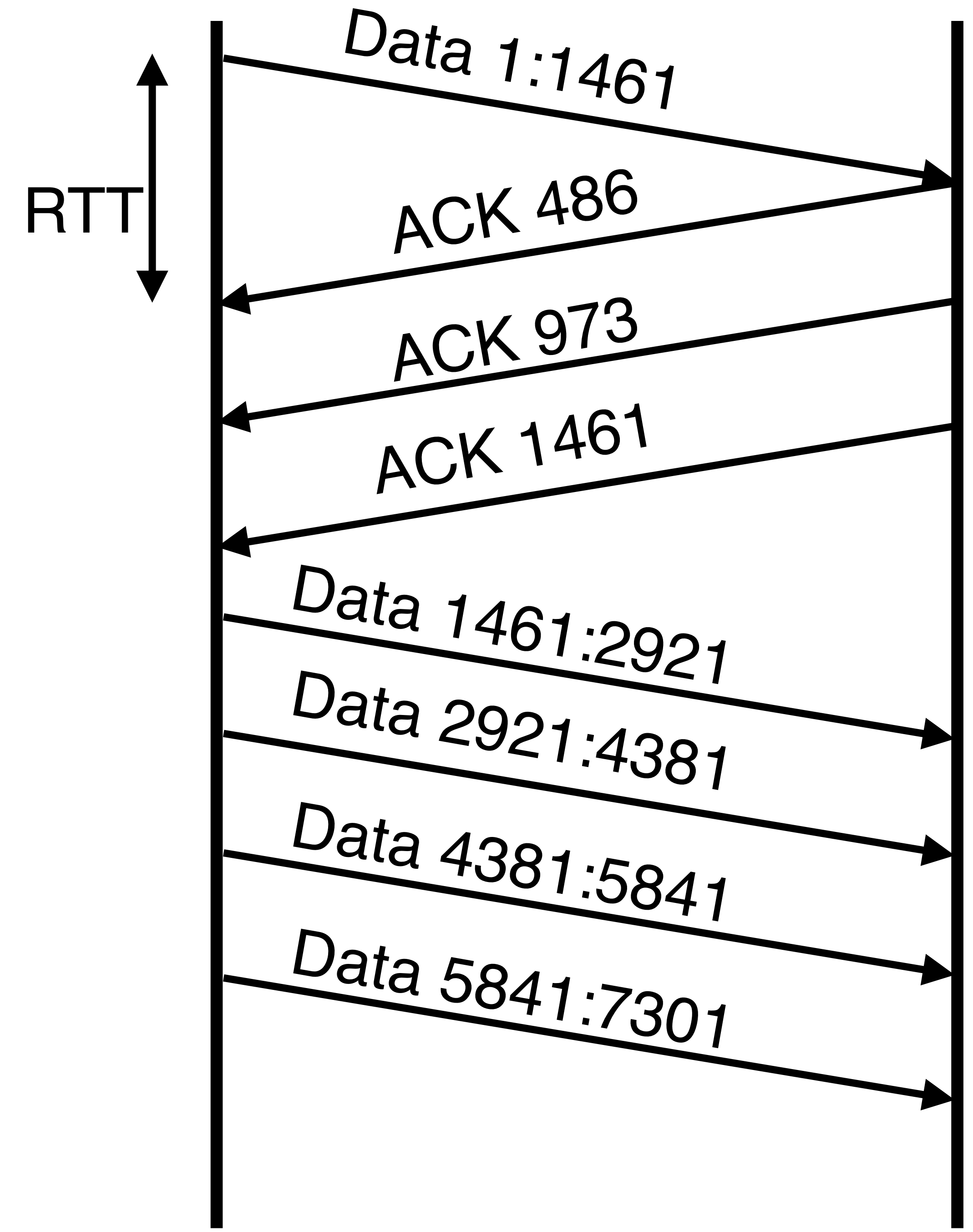
- Larger initial window?
  - Google proposed moving from ~4KB to ~15KB
  - Covers ~90% of HTTP Web
  - Decreases delay by 5%
- Many recent research papers on the timeout problem
  - Require network support

(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)

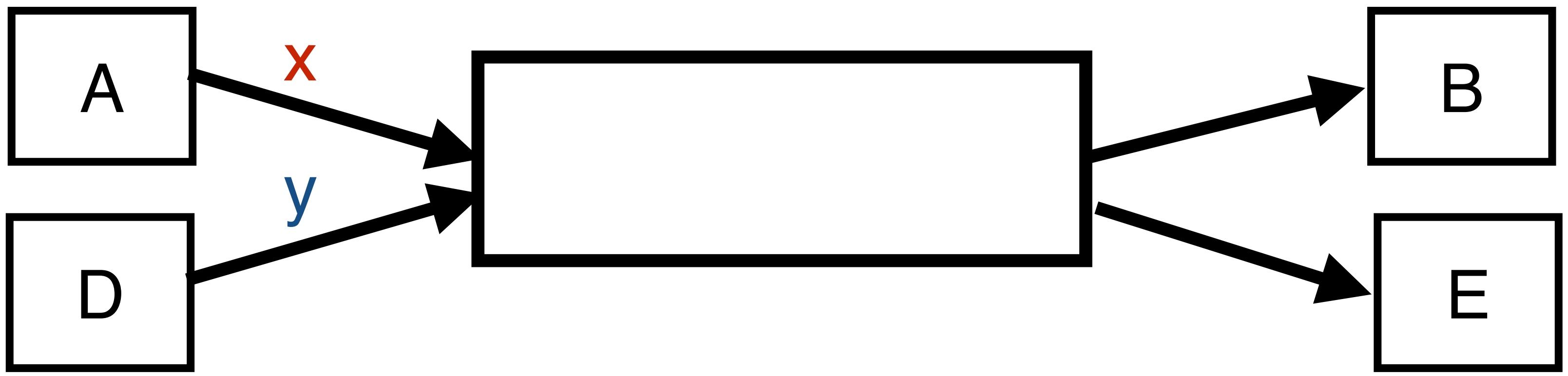
- TCP Rule: grow window by one MSS for each valid ACK received
- Send **M** (distinct) ACKs for one MSS
- Growth factor proportional to **M**



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?

How Do You Solve These Problems?

- Bias against long RTTs
- Slow to ramp up (for short-flows)
- Cheating
- Need for uniformity

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) <—> socket

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

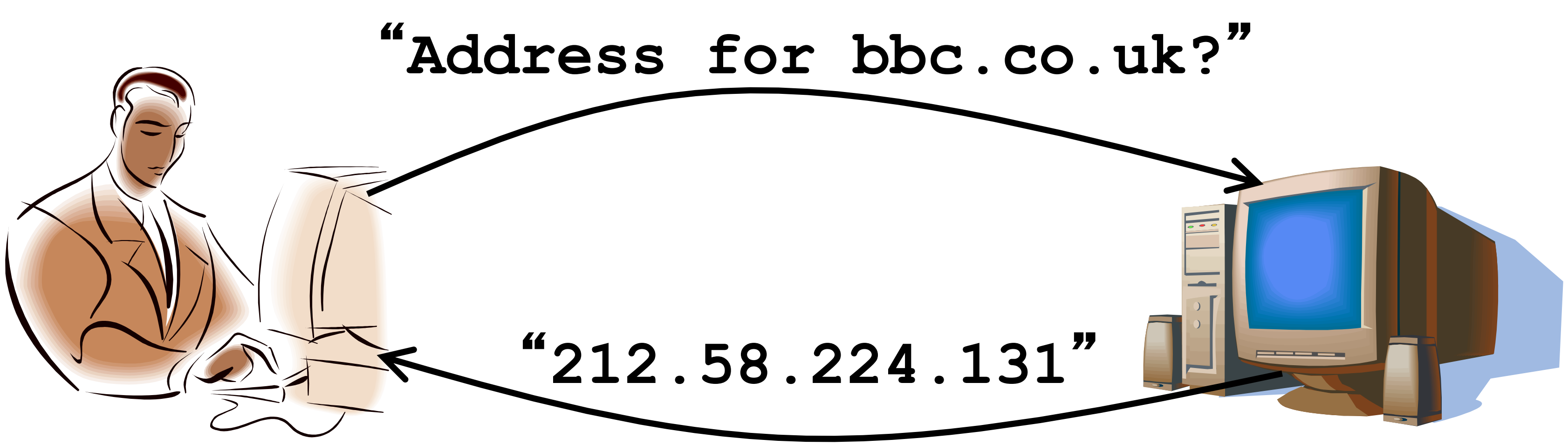
Question

- 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

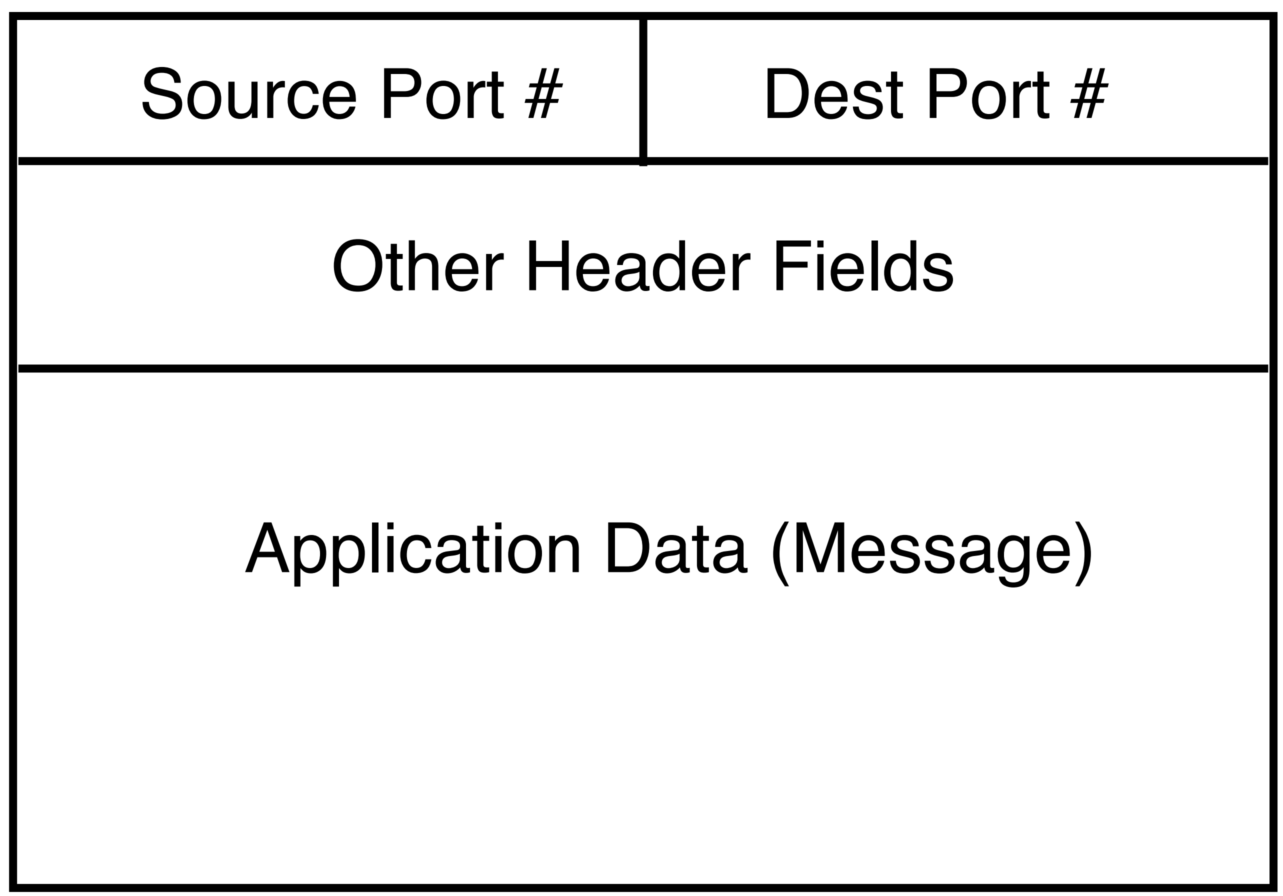
(not needed for exams)

Ports

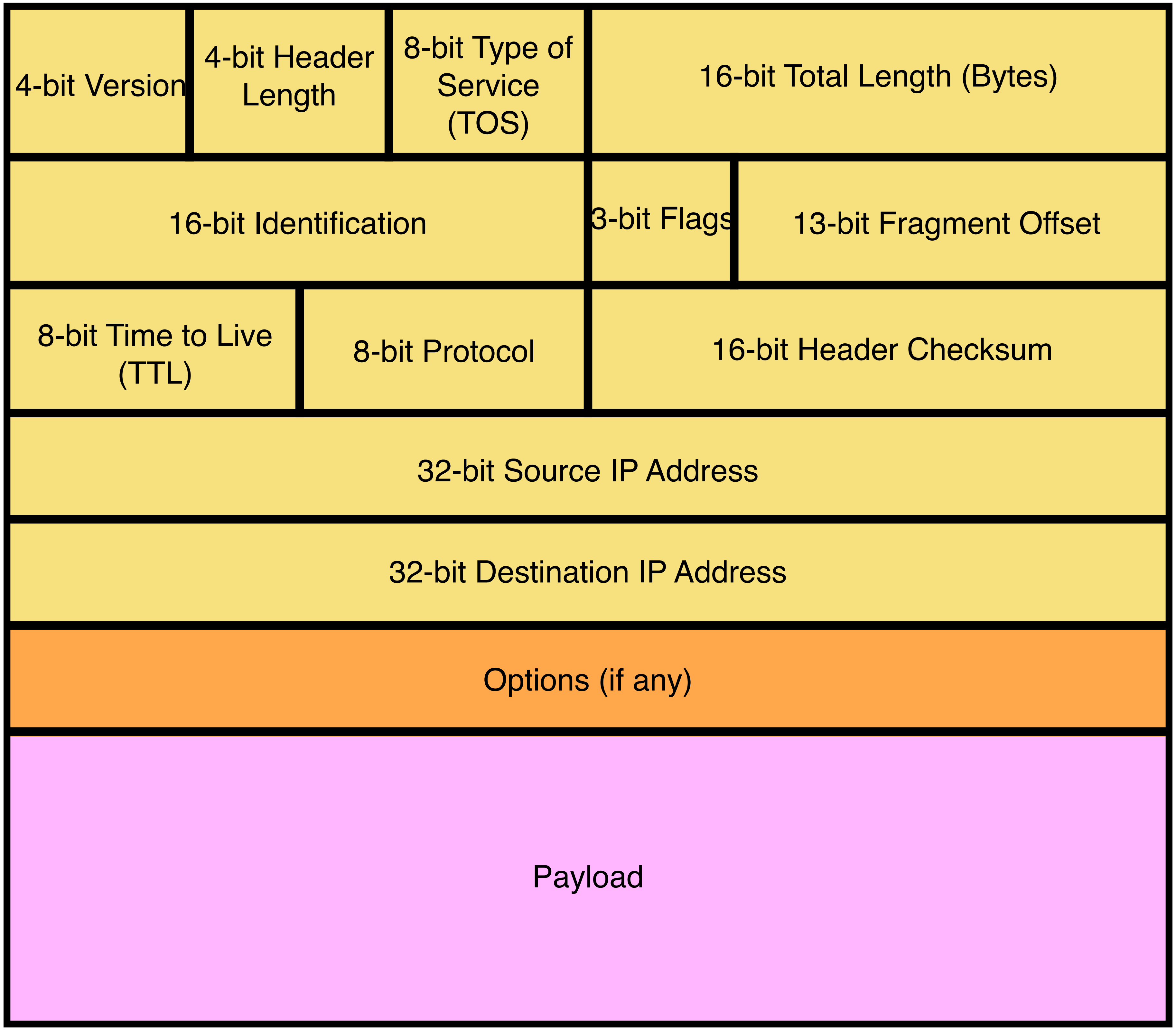
- Separate 16-bit port address space for UDP, TCP
- “Well known” ports (0-1023)
    - Agreement on which services run on these ports
    - e.g., ssh:22, http:80
    - Client (app) knows appropriate port on sender
    - Services can listen on well-known ports

Multiplexing and Demultiplexing

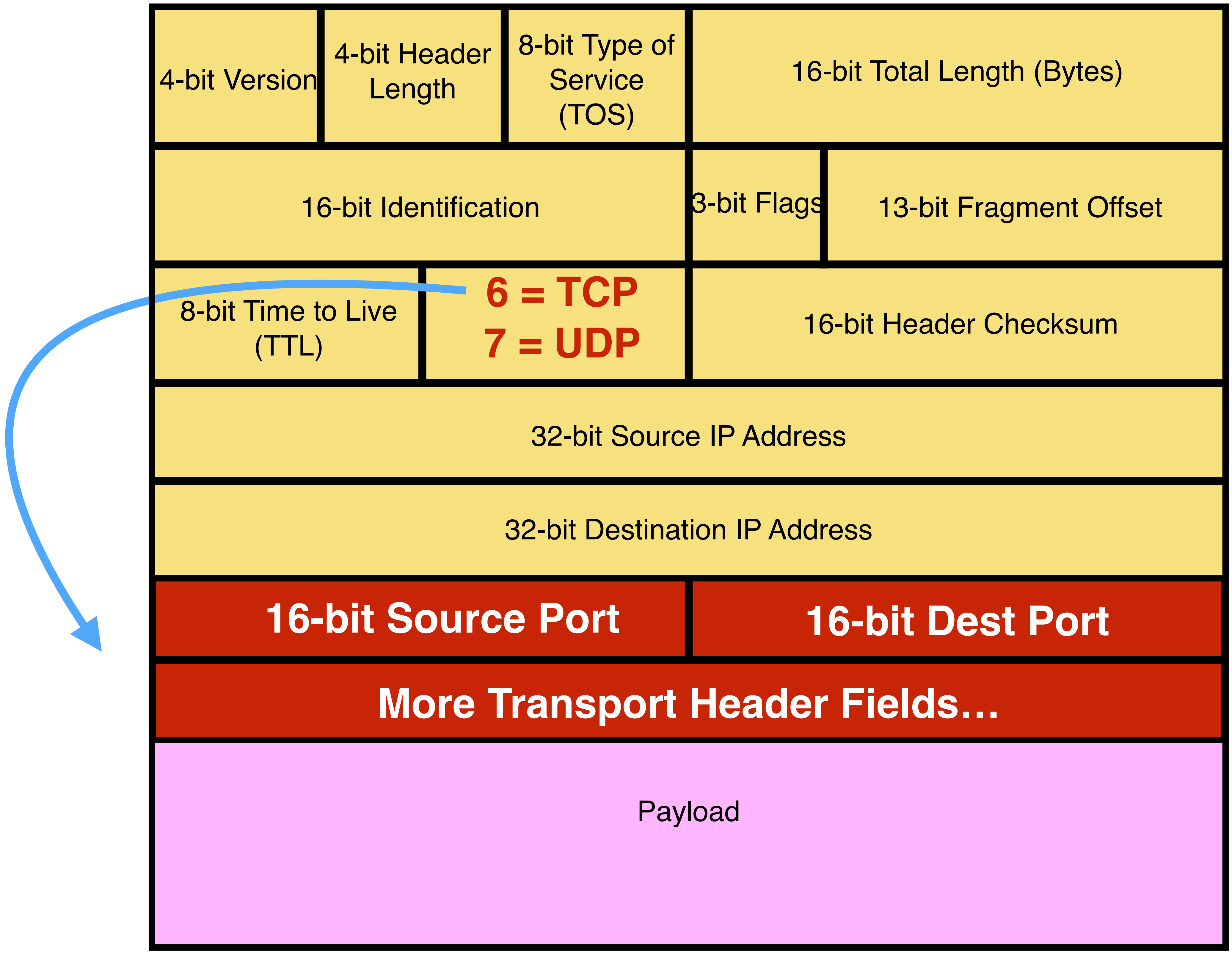
- Host receives IP datagrams
  - Each datagram has source and destination IP address
  - Each segment has source and destination port number
- Host uses IP address and port numbers to direct the segment to appropriate socket



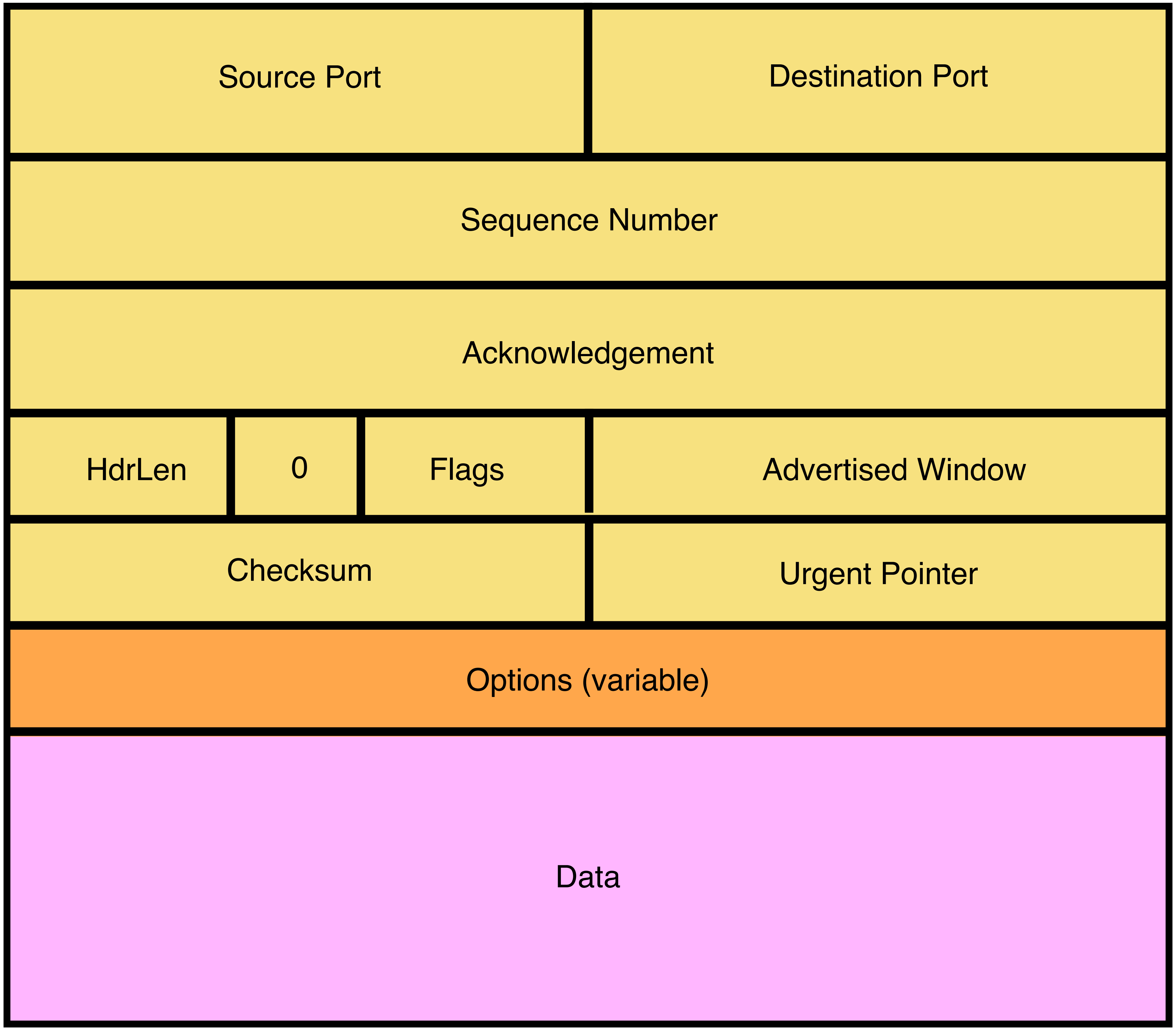
IP Packet Structure



IP Packet Structure

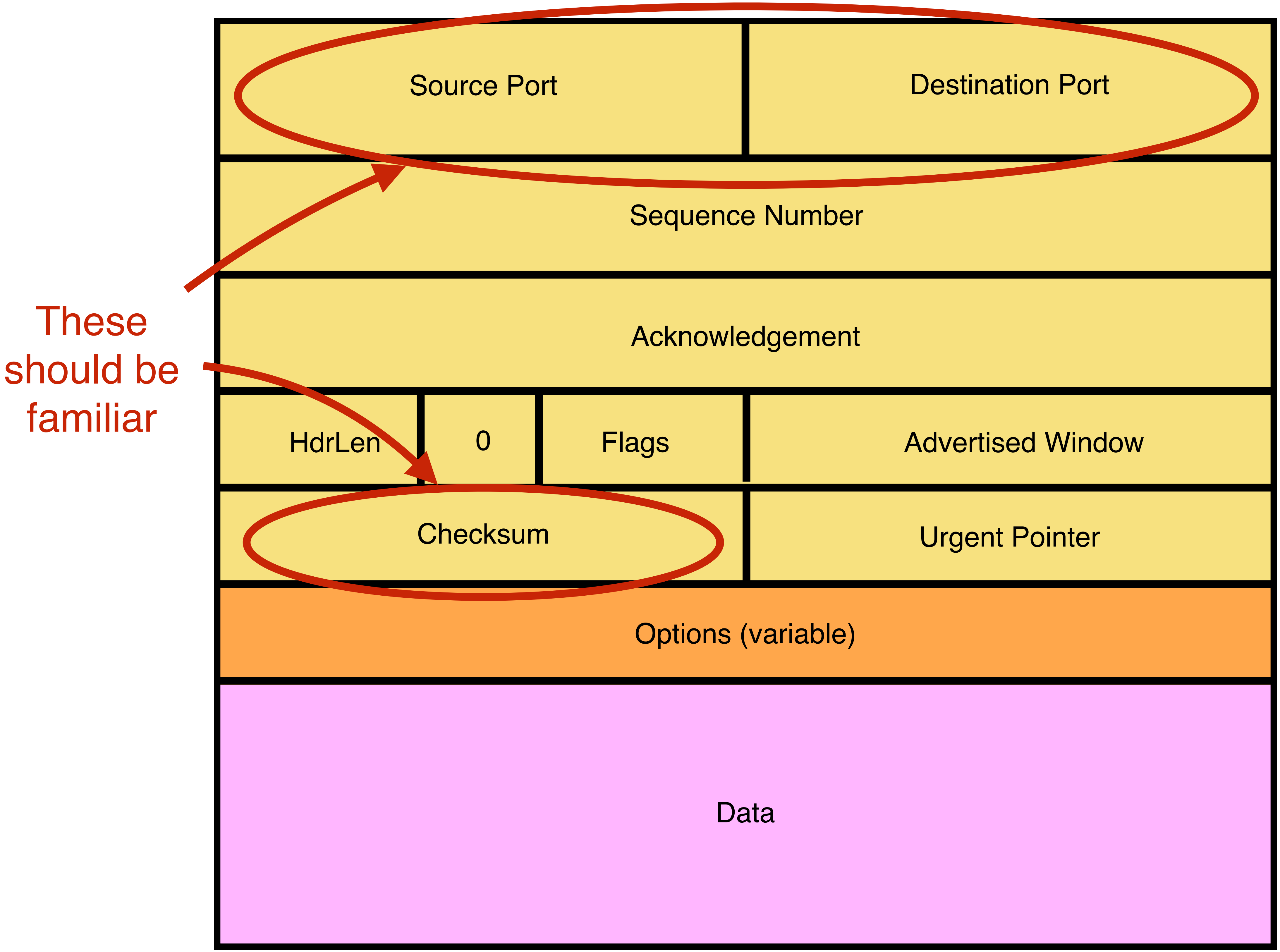


TCP Header



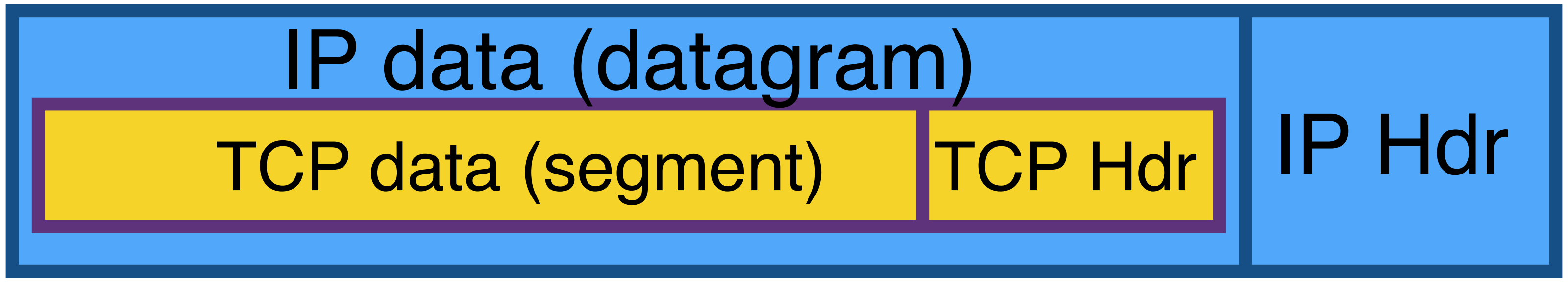
Source Port
Destination Port
Sequence Number
Acknowledgement
HdrLen
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Data

TCP Header



Source Port
Destination Port
Sequence Number
Acknowledgement
HdrLen
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Data
These should be familiar

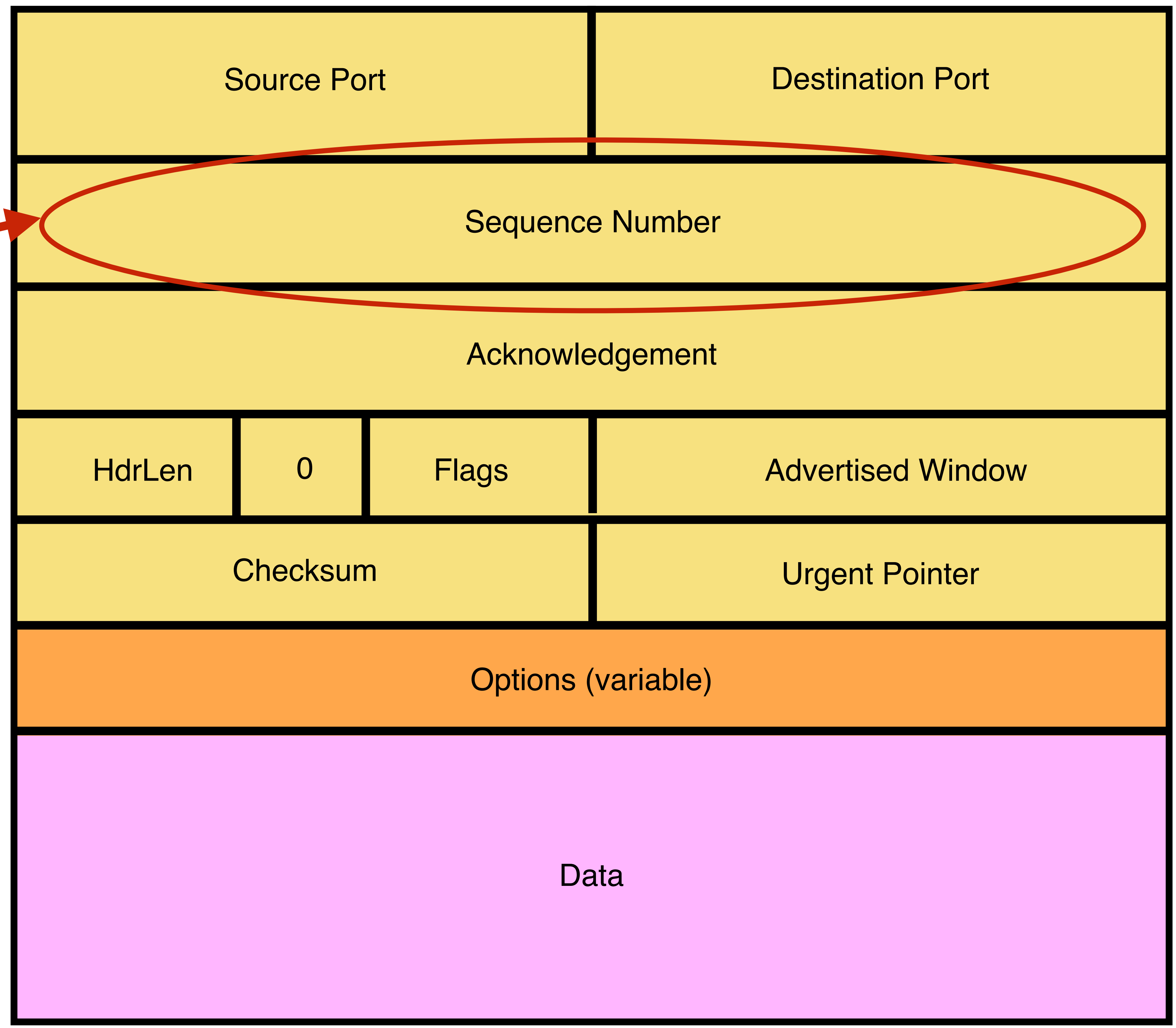
TCP Segment



- 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 >= 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

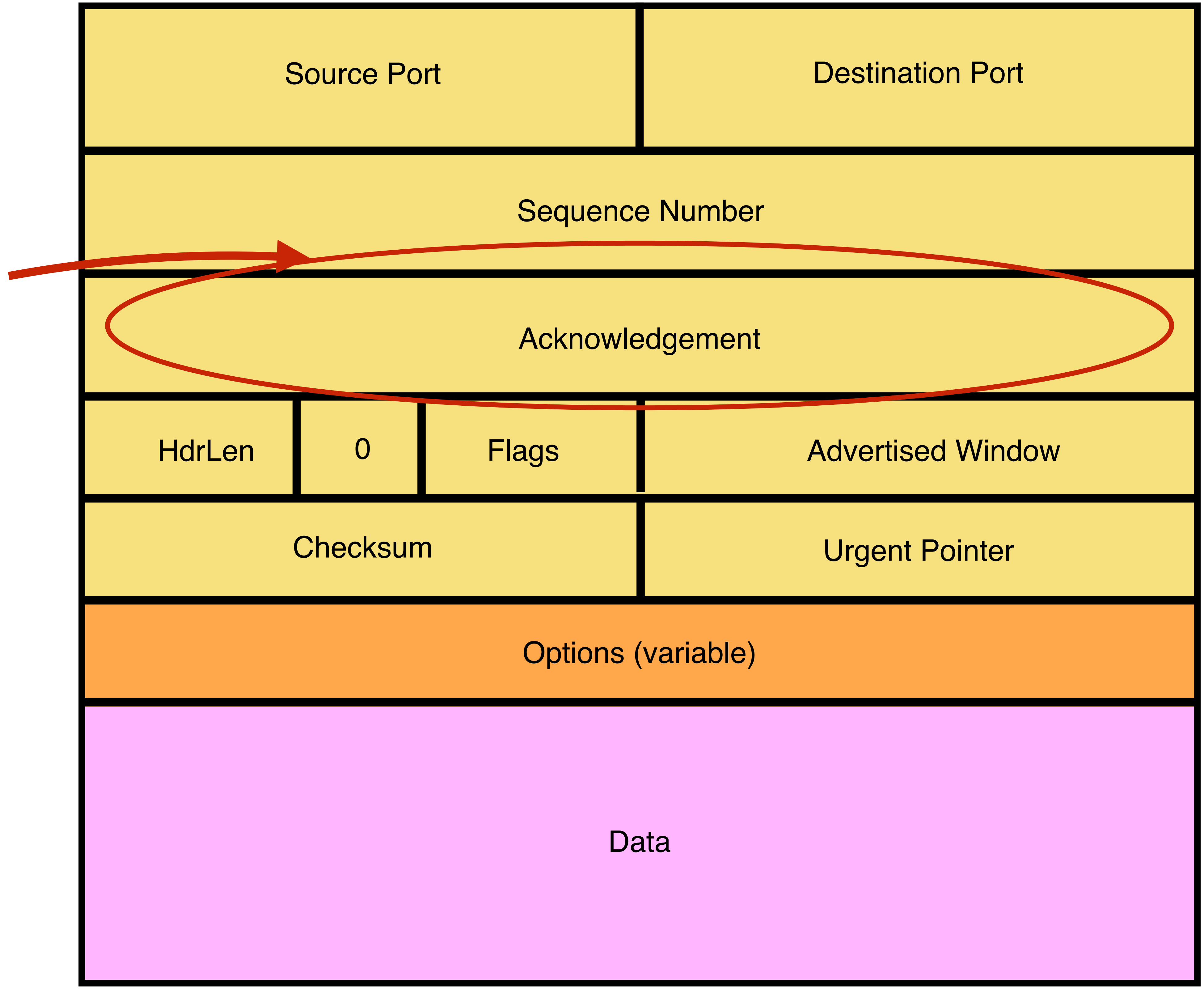
TCP Header

Starting byte offset of data carried in this segment

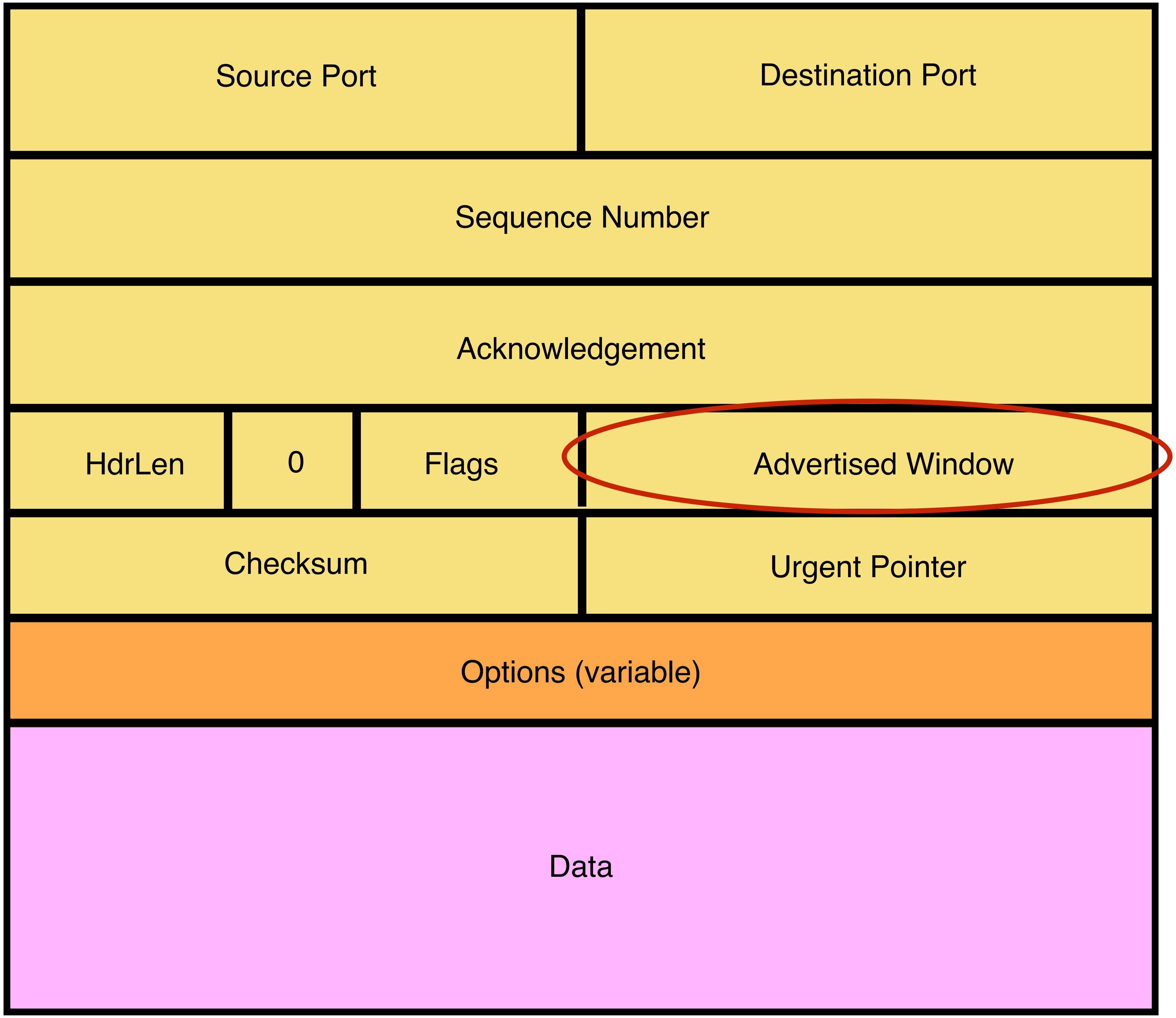


TCP Header

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

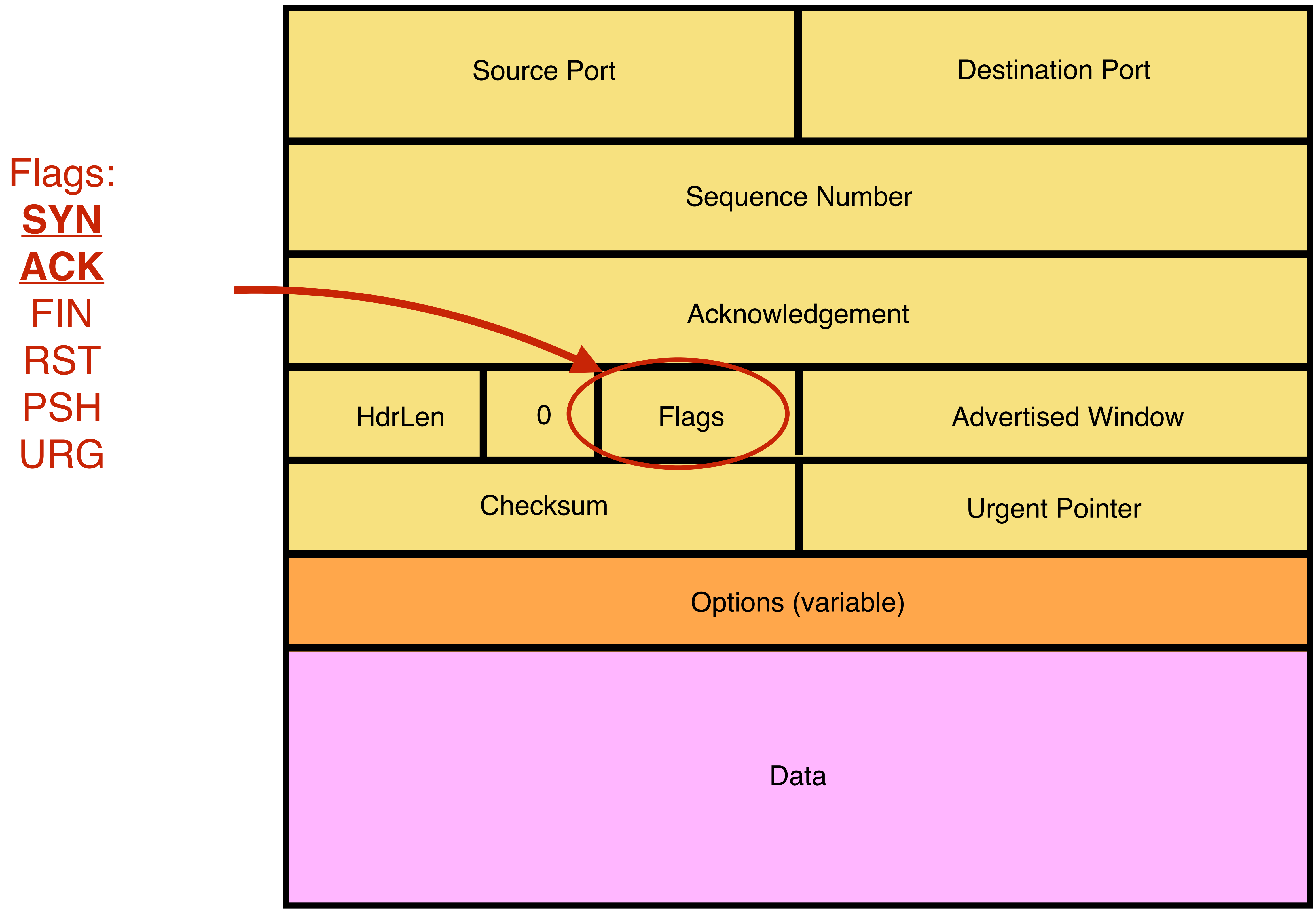


TCP Header



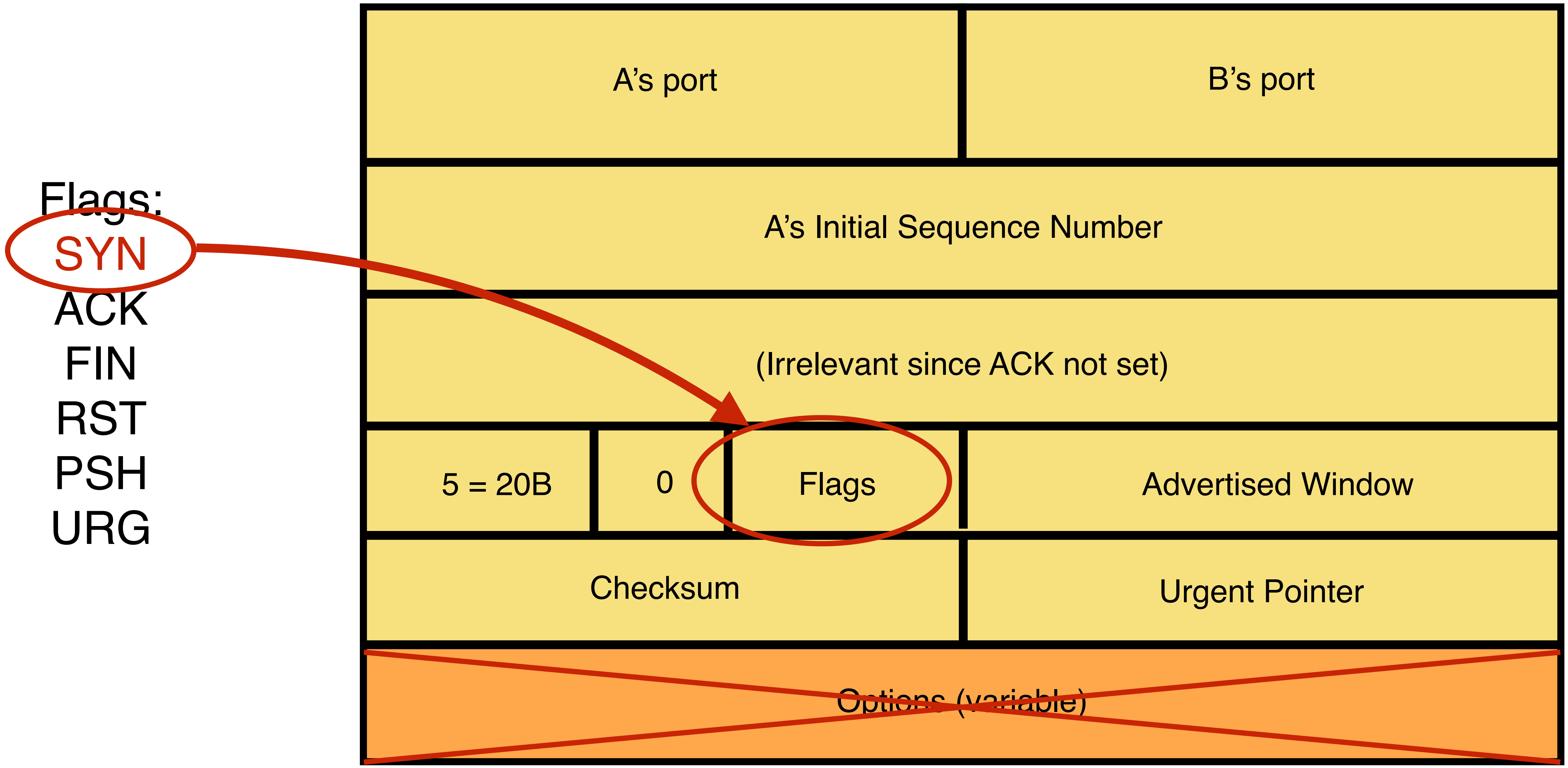
Source Port
Destination Port
Sequence Number
Acknowledgement
HdrLen
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Data

TCP Header



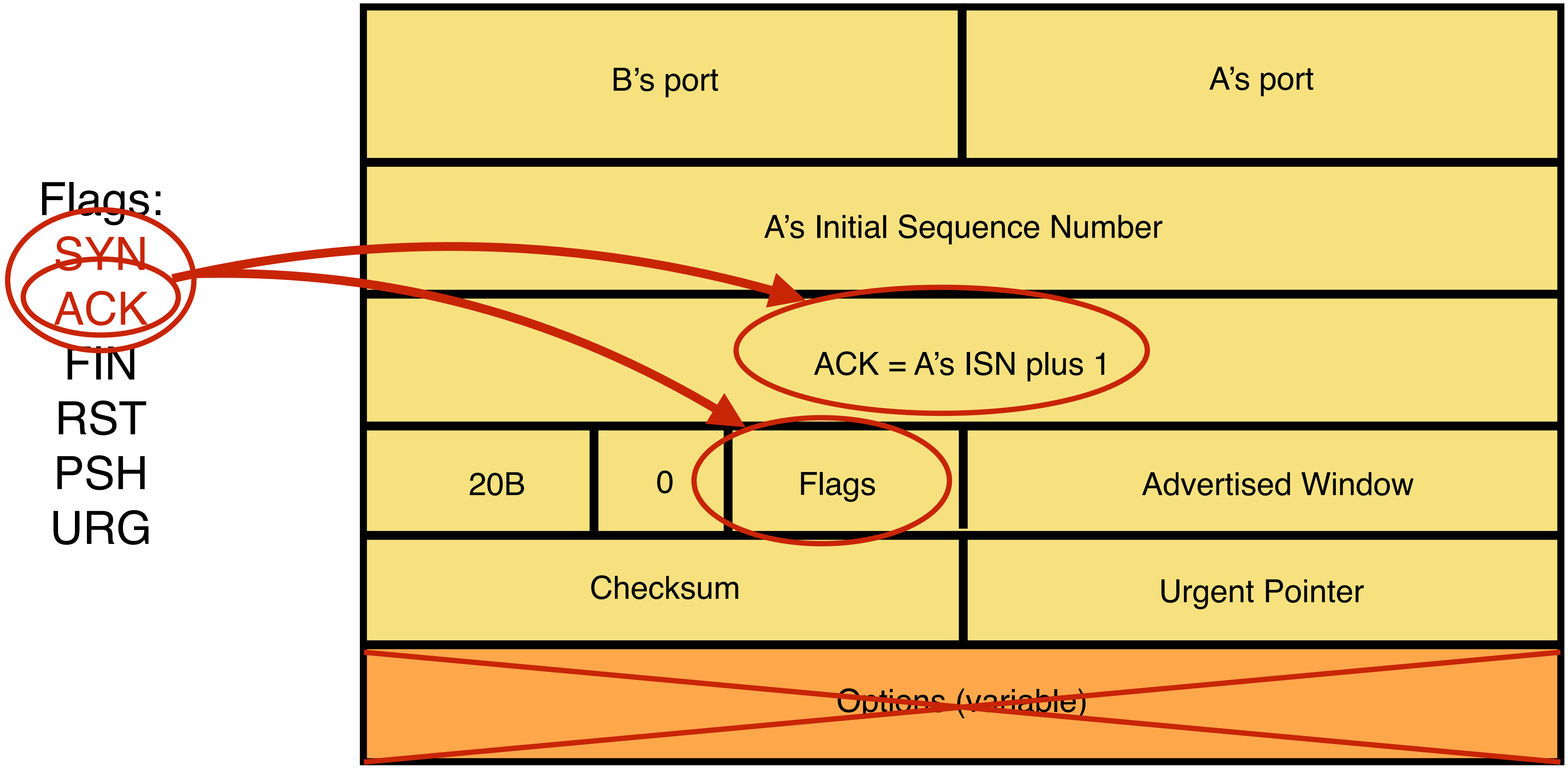
See /usr/include/netinet/tcp.h on Unix Systems

Step 1: A's Initial SYN Packet



A tells B it wants to open a connection…

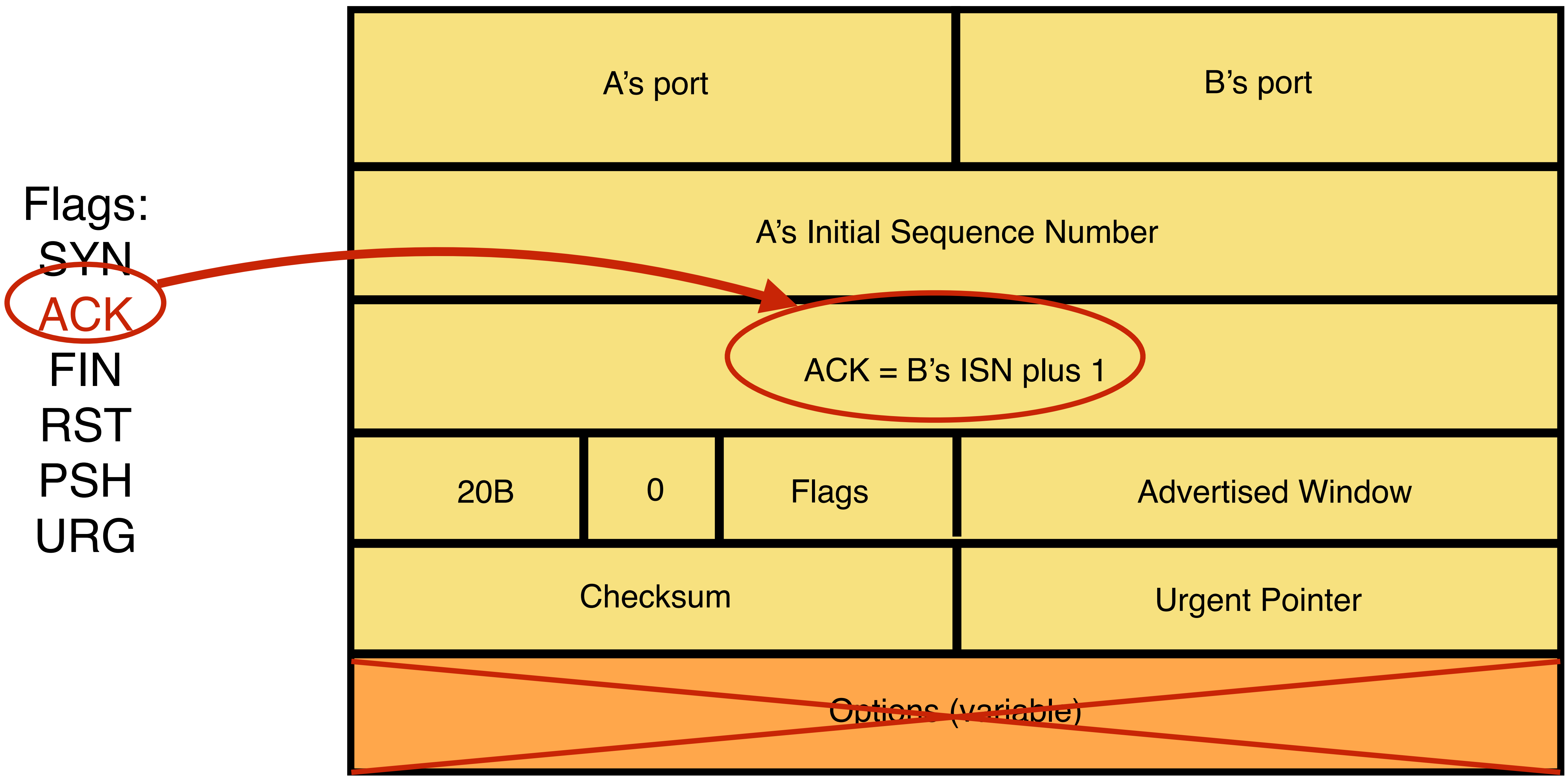
Step 2: B's SYN-ACK Packet



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

… upon receiving this packet, A can start sending data

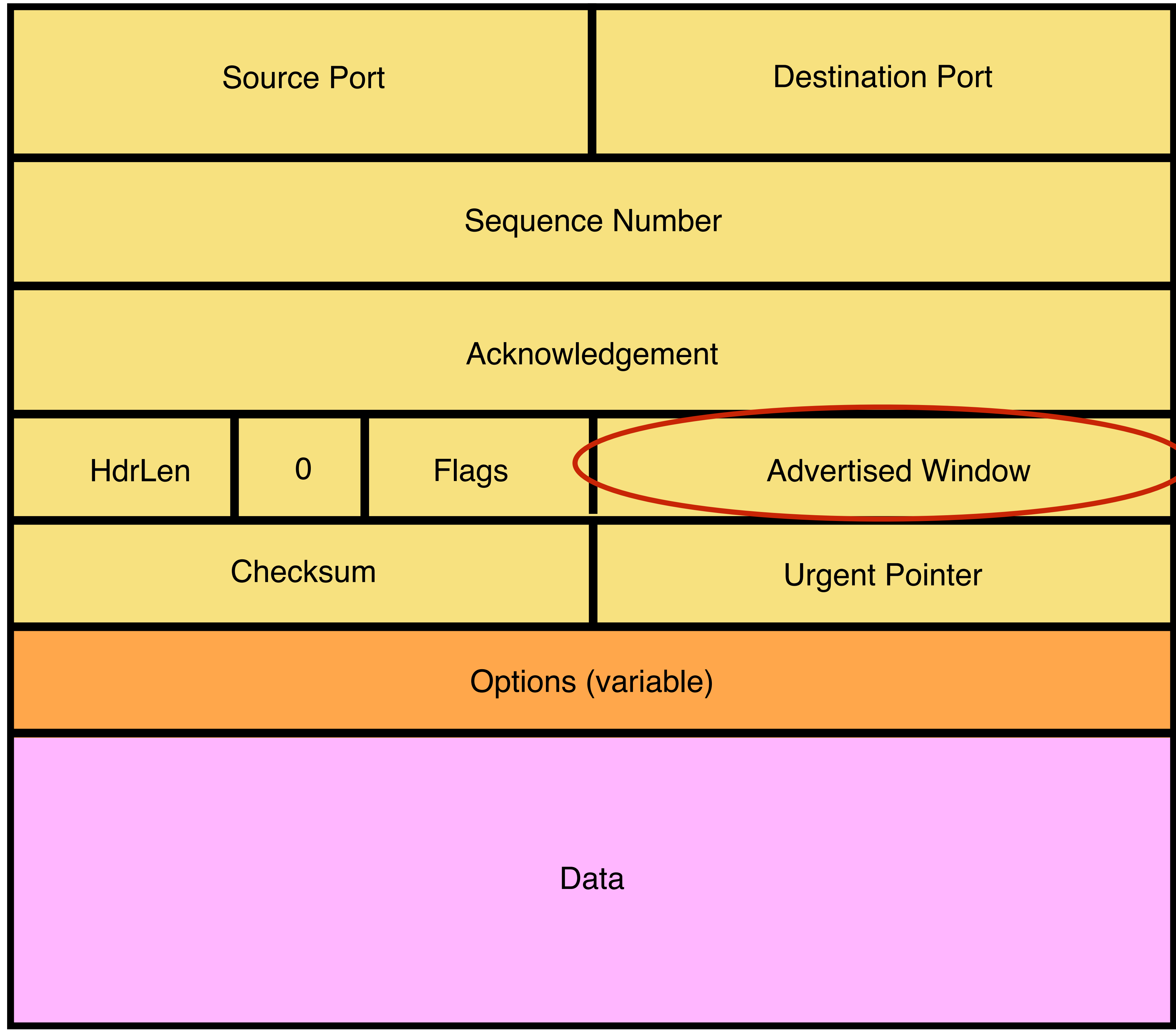
Step 3: A's ACK of the SYN-ACK



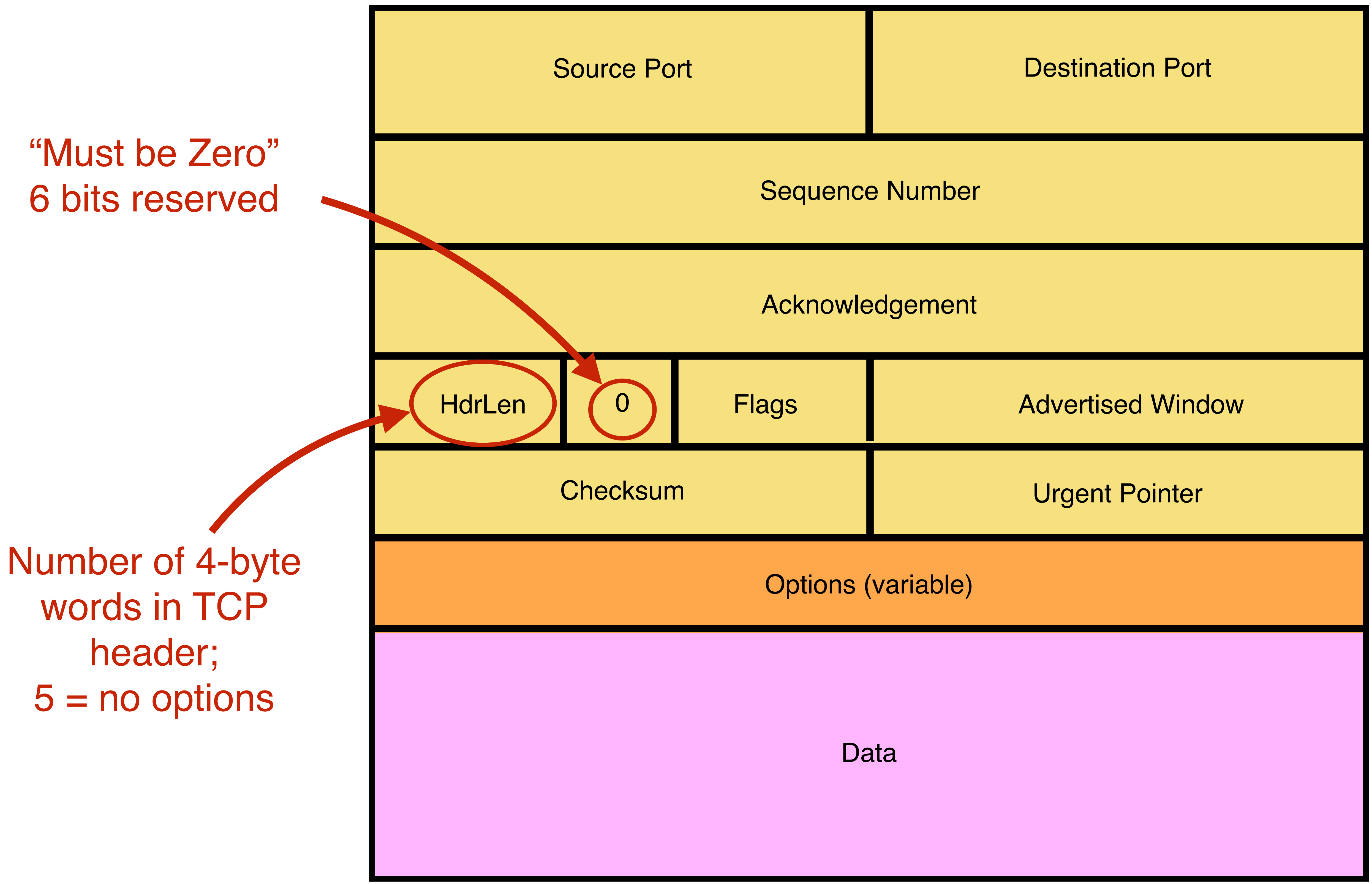
A tells B it's likewise okay to start sending

… upon receiving this packet, B can start sending data

TCP Header

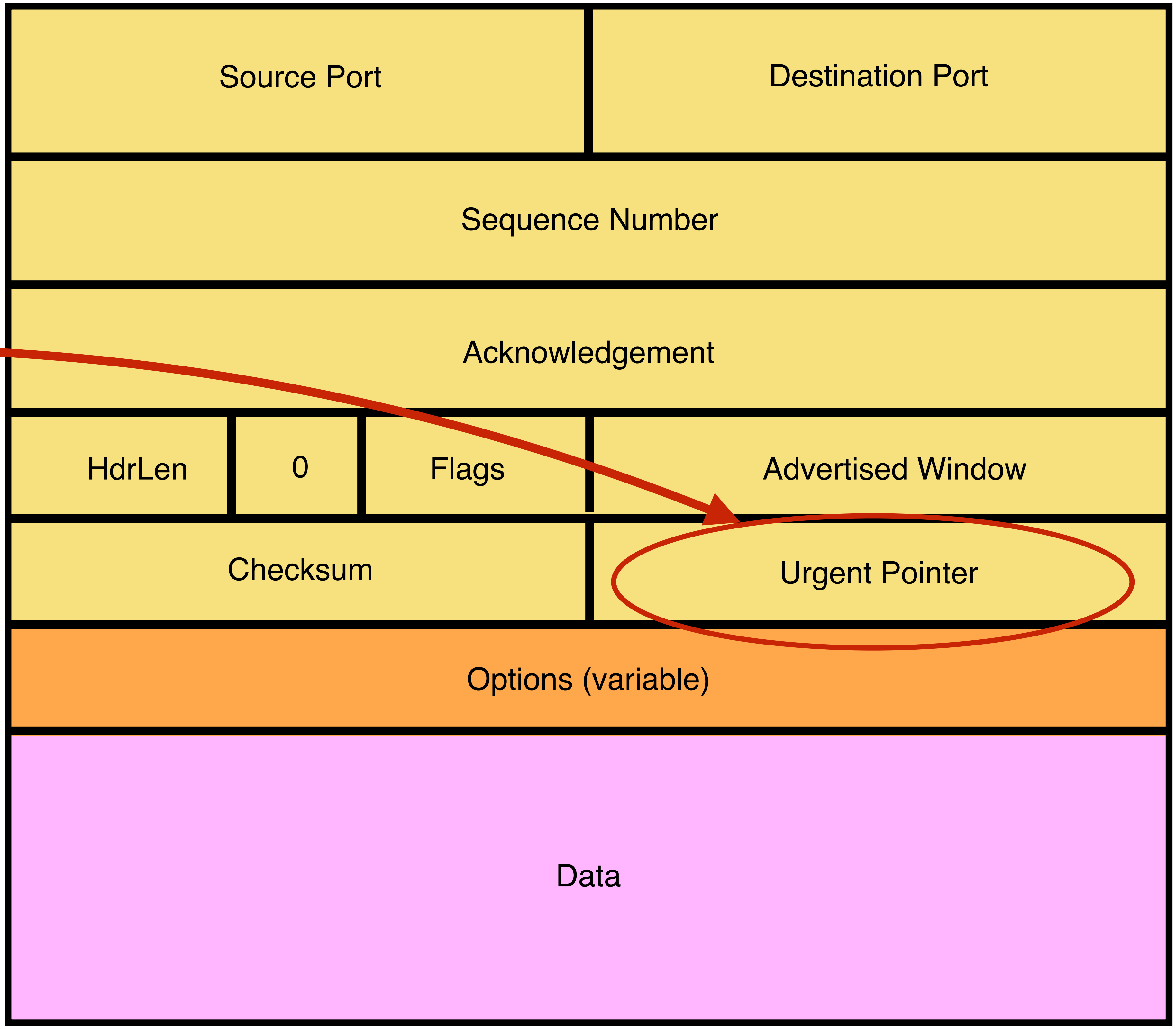

Source Port
Destination Port
Sequence Number
Acknowledgement
HdrLen
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Data

TCP Header: What's left?



TCP Header: What's left?

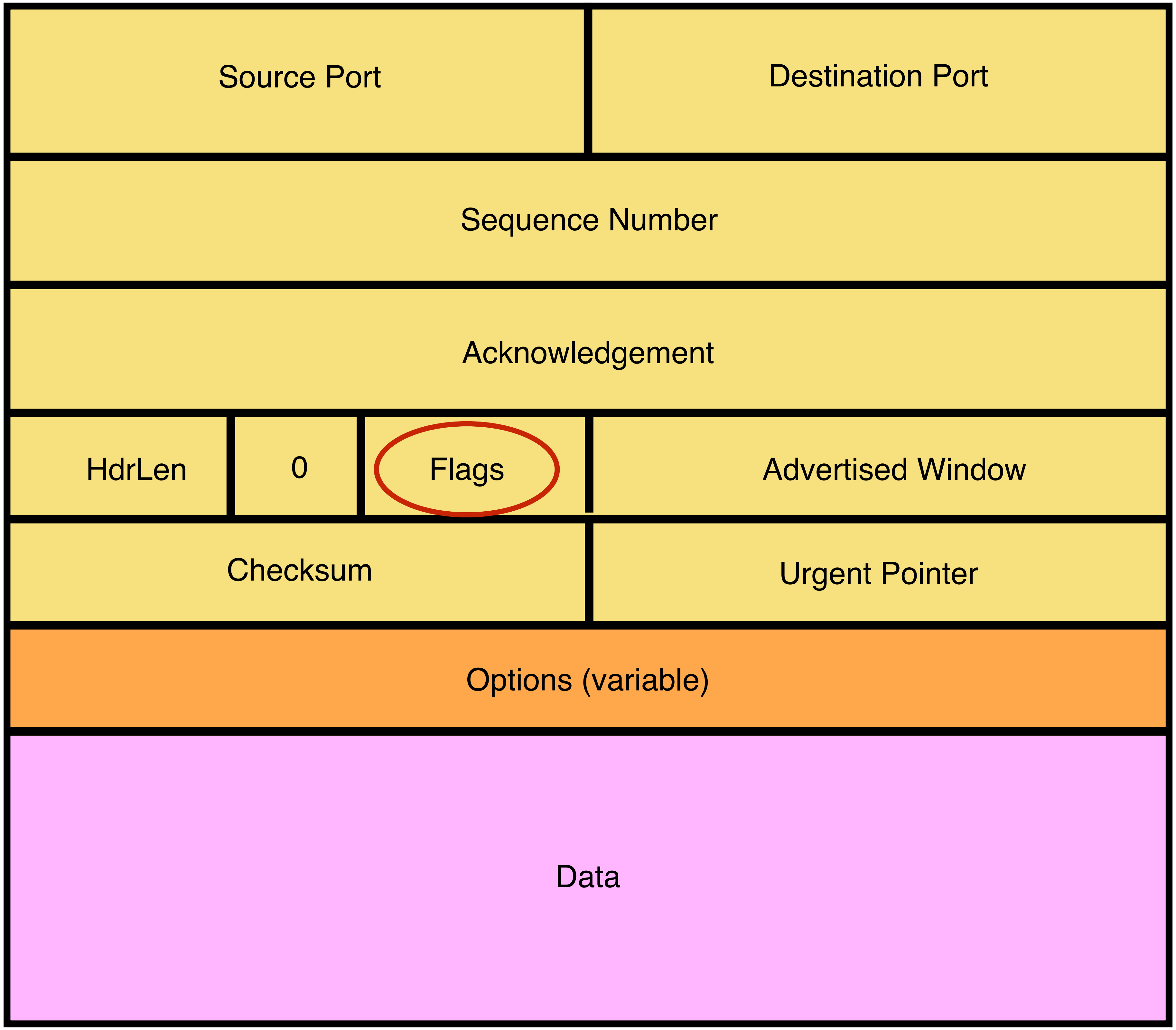
Used with URG flag to indicate urgent data (not discussed further)



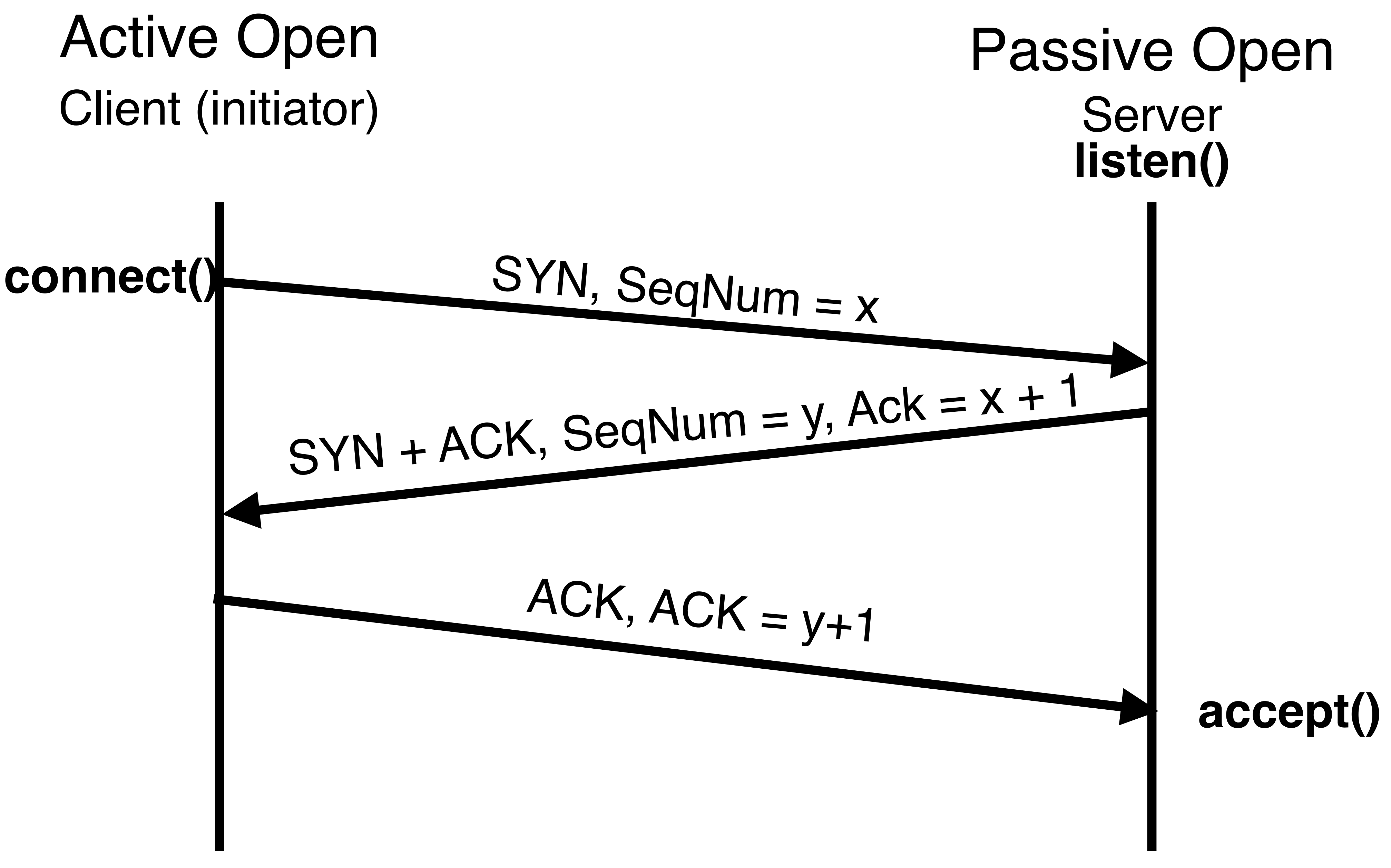
Implementing Sliding Window

- 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?

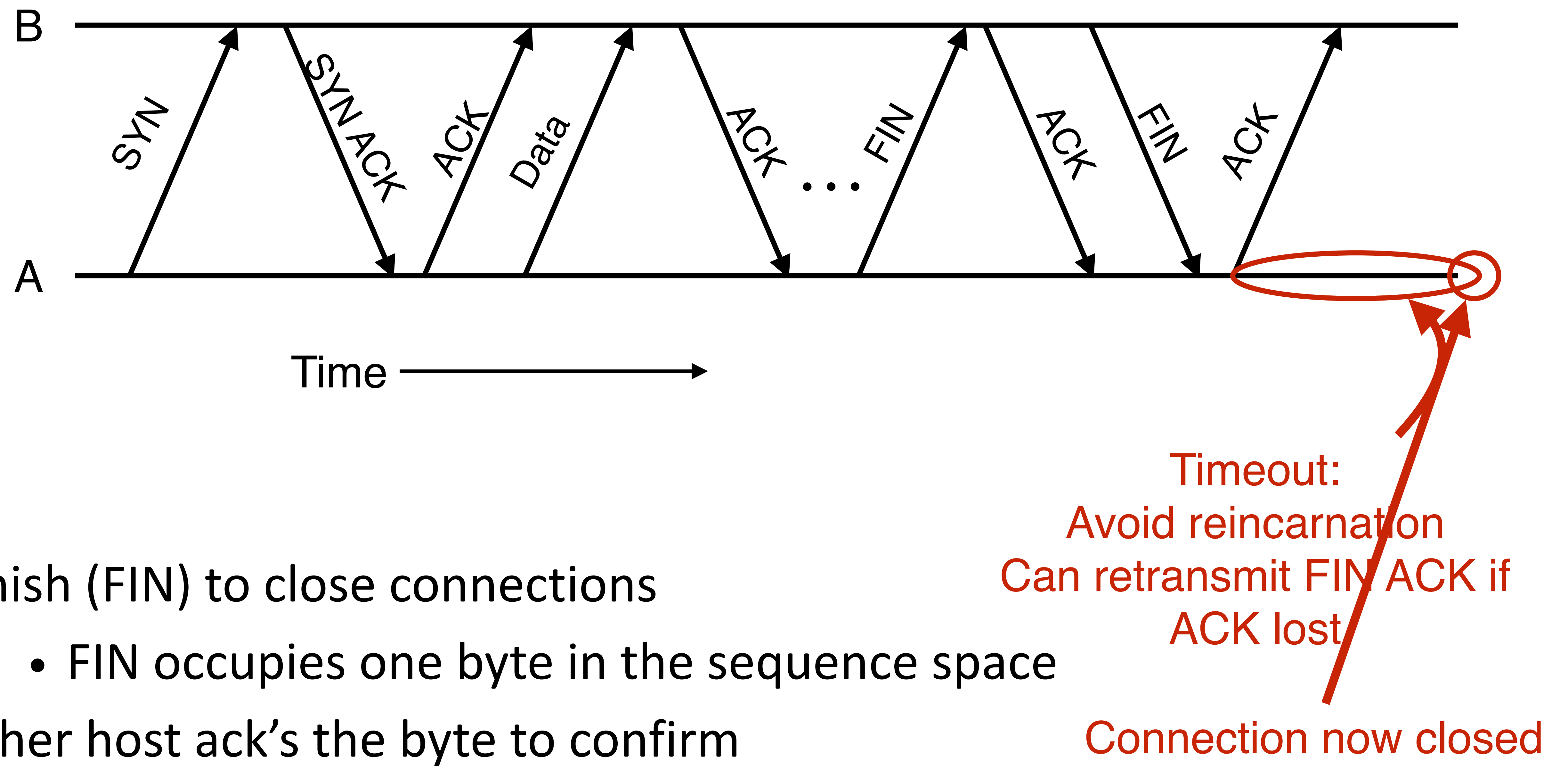
- Suppose the SYN packet gets lost
    - Packet is lost inside the network, or
    - Server **discards** the packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
    - Sender sets a **timer** and **waits** for the SYN-ACK
    - … and retransmits the SYN if needed
- How should the TCP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

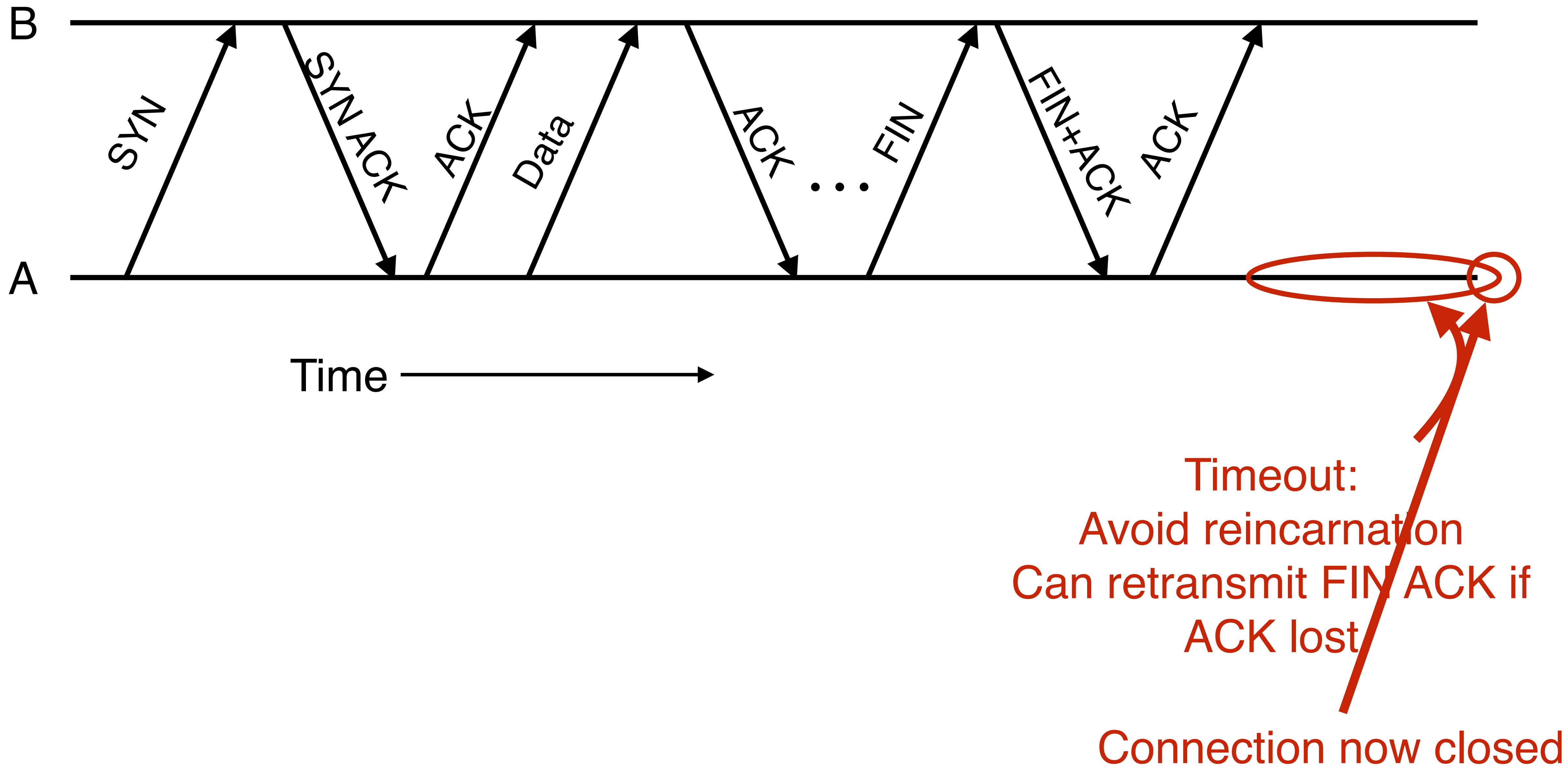
Normal Termination



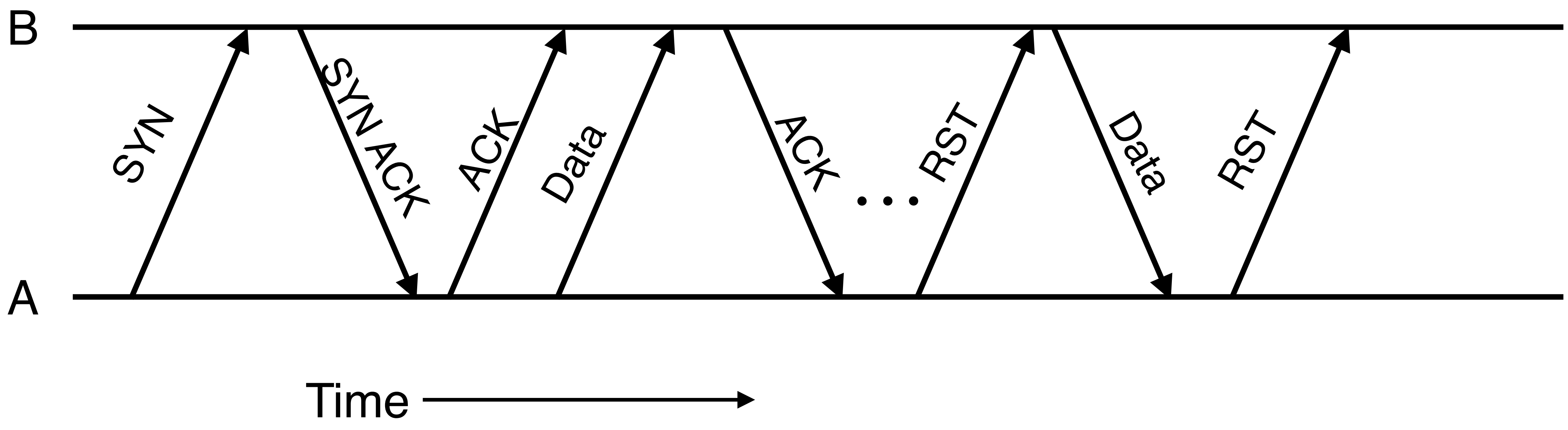
- Finish (FIN) to close connections
    - FIN occupies one byte in the sequence space
- Other host ack's the byte to confirm
- Closes A's side of connection, but not B's
    - Until B likewise sends a FIN
    - Which A then acks

Normal Termination, Both Together

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

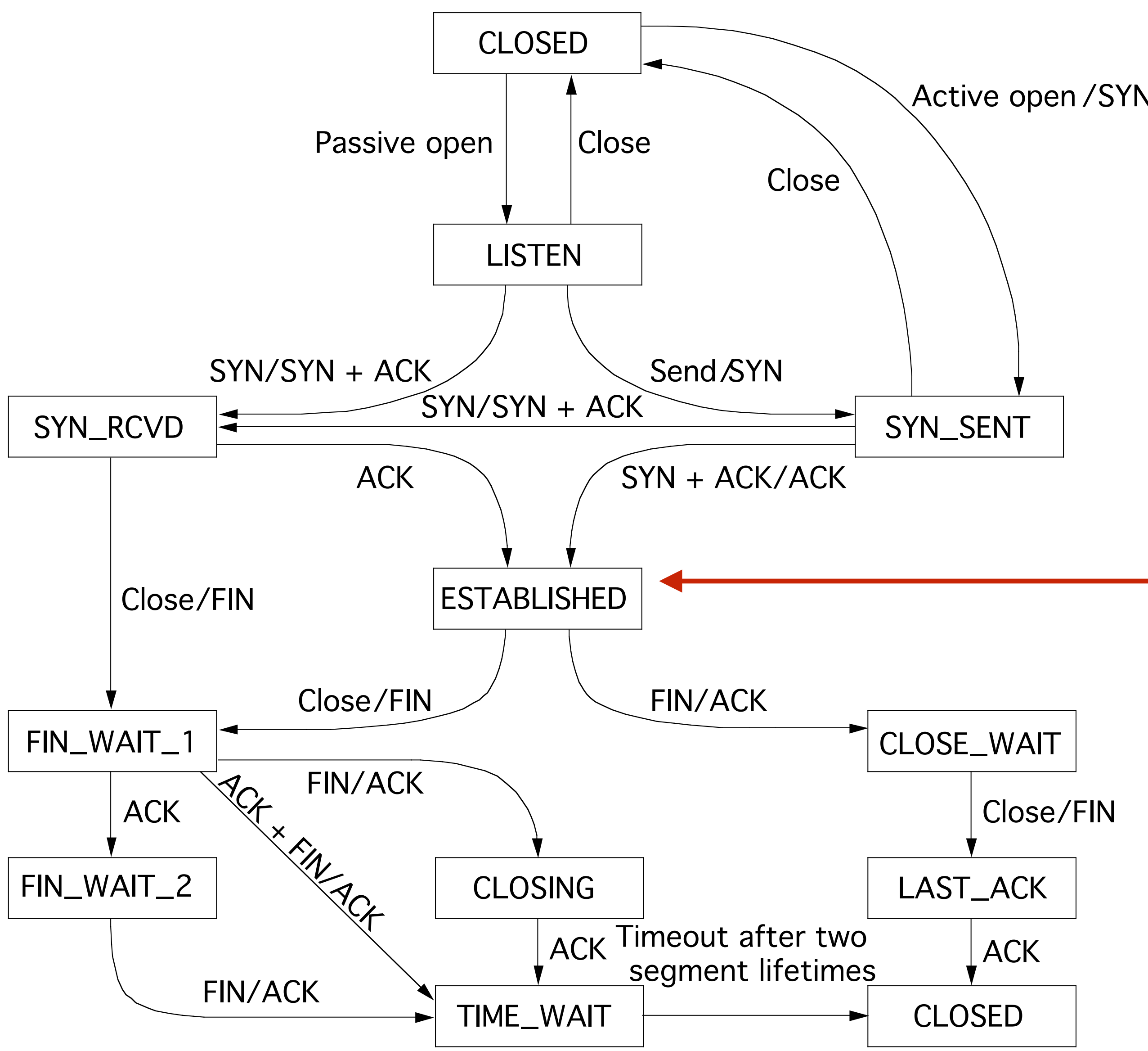


Abrupt Termination



- 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