Chapter 1

TCP
Congestion Control

Outline

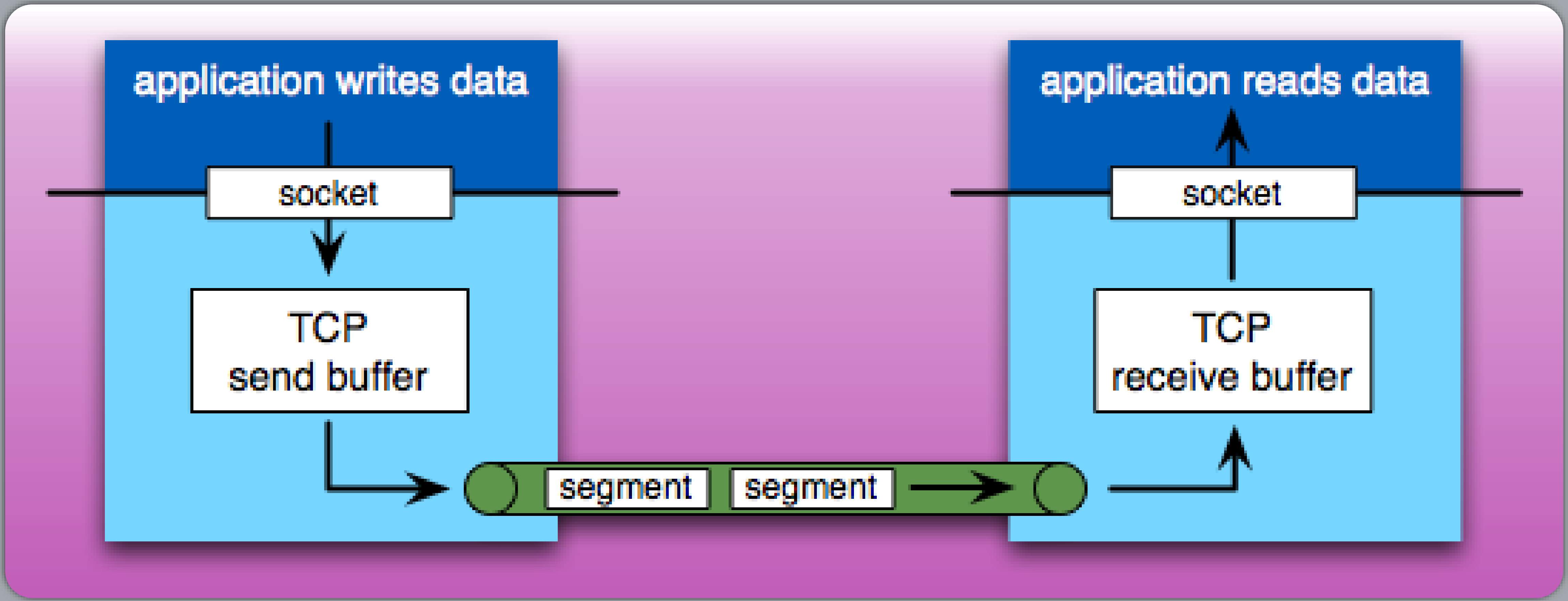
- Connection-oriented Transport: TCP
- Segment Structure
- Connection Management
- Reliable Data Transfer

TCP Overview

- Point-to-point
  - One sender, one receiver
- Reliable
  - Segments delivered in-order without loss
- In-order byte stream
  - No message / record boundaries
- Pipelined
  - Sliding-window type control algorithm

TCP Overview

- Full duplex data
  - bi-directional data flow in the same connection
- Connection-oriented
  - handshaking (exchange of control messages) initializes sender & receiver state before data exchange



- Both sides have buffers
  - Lots of “Producer-Consumer” coordination problems to overcome

Transmission Control

- Flow Control
  - Sender will not overwhelm receiver
- Congestion Control
  - Various algorithms employed to limit sending of segments
  - Don’t want to overwhelm the network

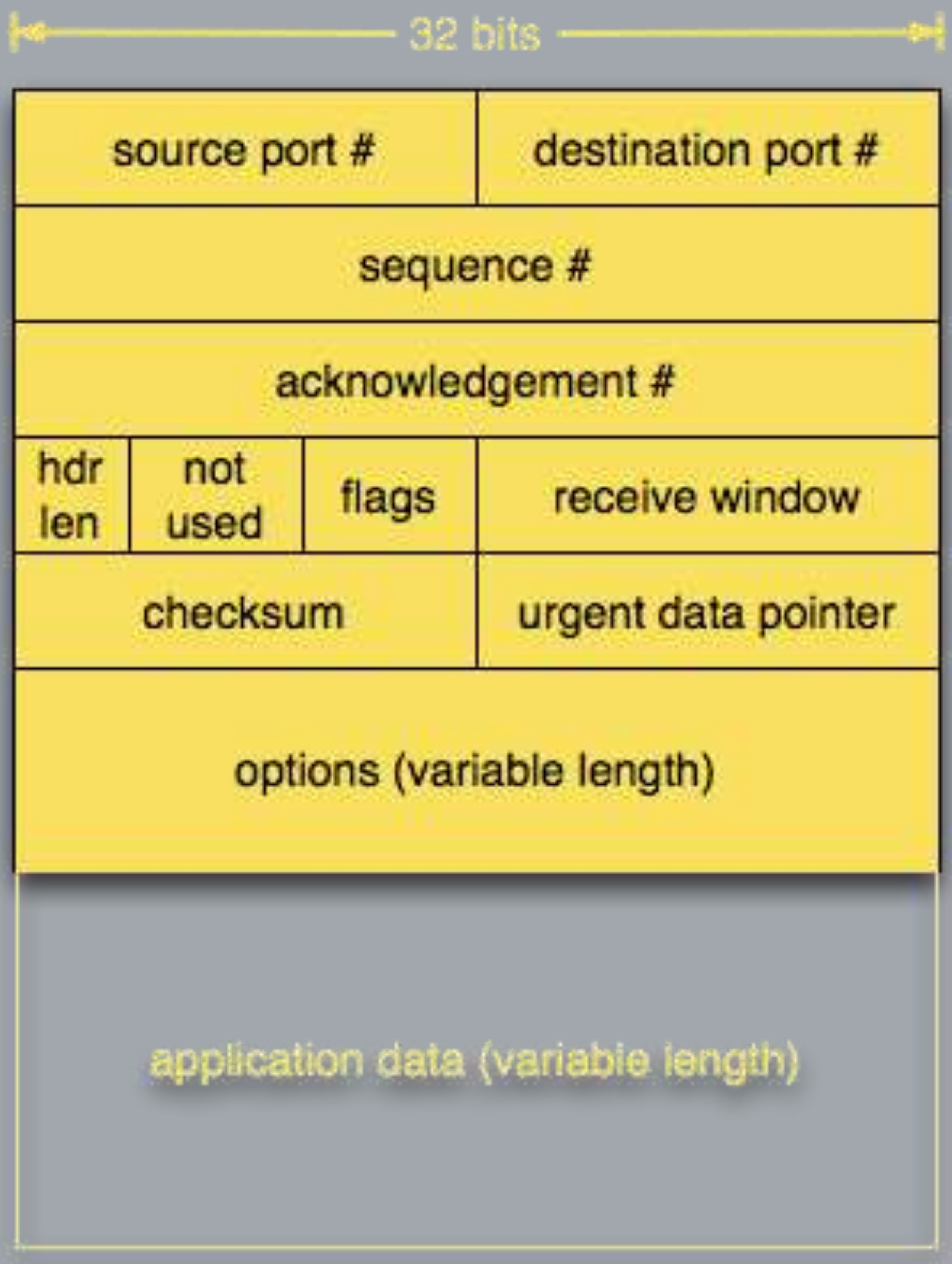
Outline

- Connection-oriented Transport: TCP
- Segment Structure
- Connection Management
- Reliable Data Transfer

Format

- Source port #: Identifies the **sending application** (process) on the source host.
- Example: HTTP client might use port 49152.

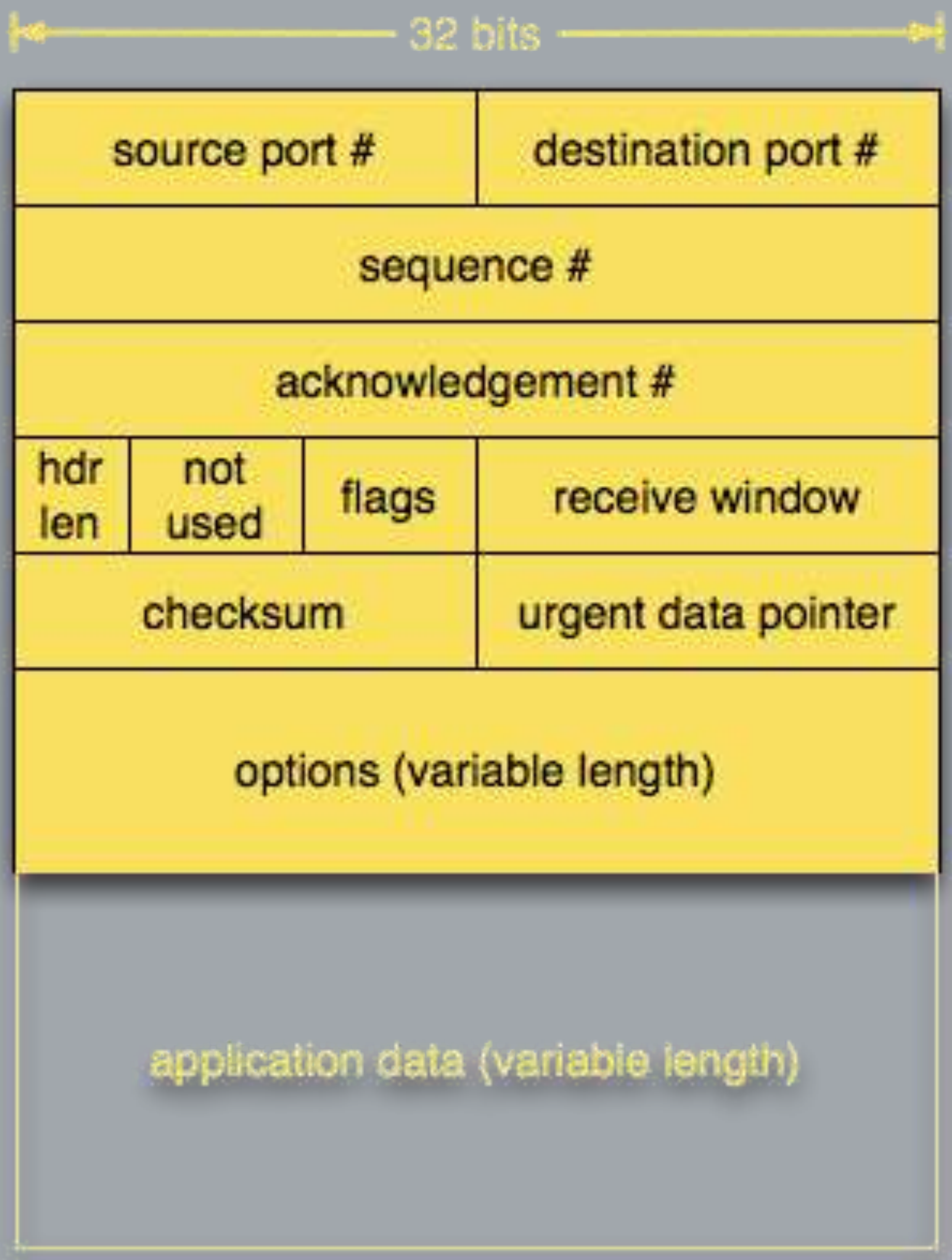
- Destination port: Identifies the **receiving application** on the destination host.
- Example: HTTP server listens on port 80 (or 443 for HTTPS).



Format

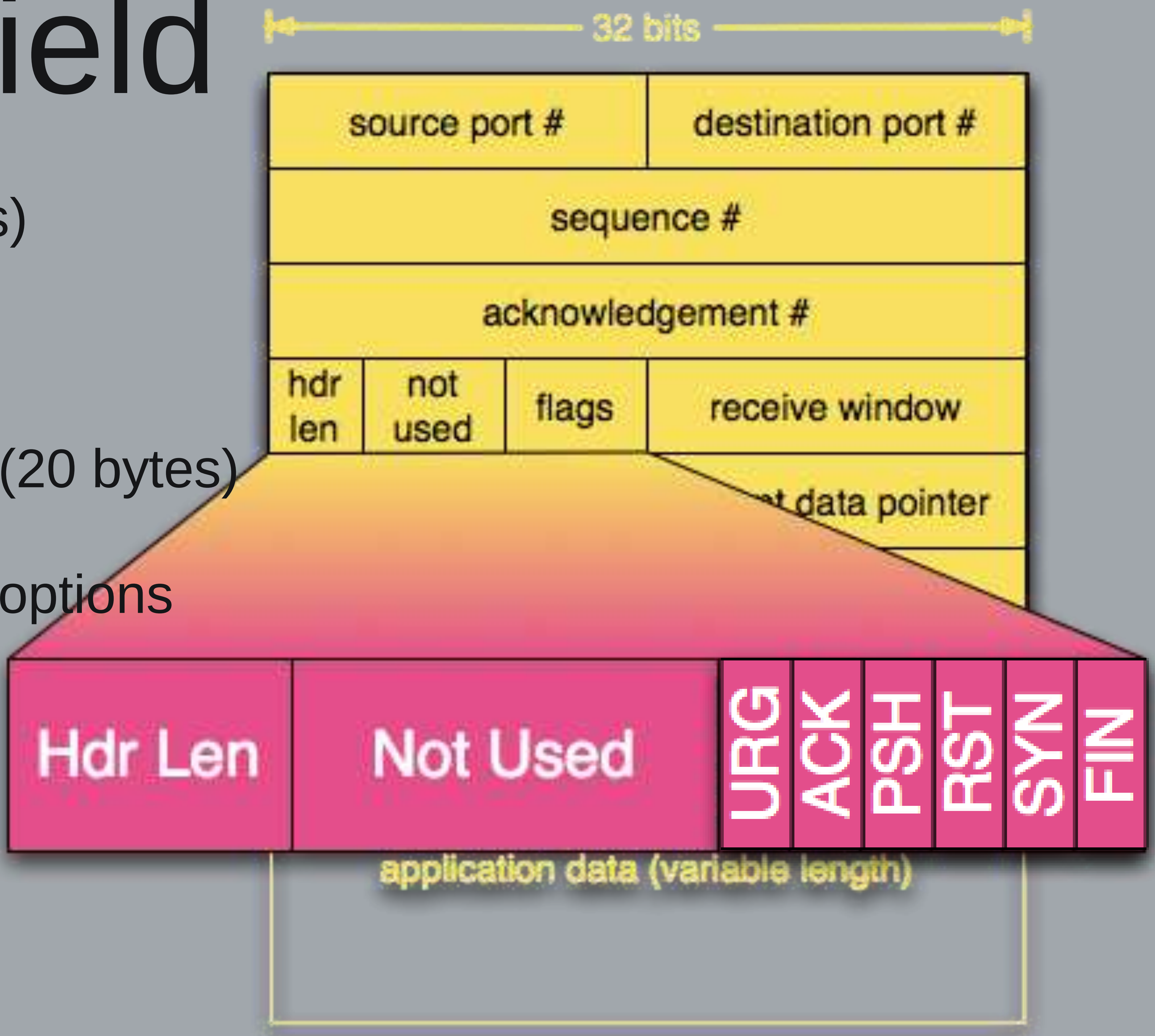
- Seq# and Ack#
  - counted by bytes of data, not segments
- Seq: number of the first byte in this segment

- Ack: number of the NEXT expected byte
  - Acks are cumulative
  - Example: ACK = 101 means bytes up to 100 have been received correctly.



Flag Field

- Header Length (4 bits)
  - # words (32-bit)
  - Minimum values: 5 (20 bytes)
  - Can be longer with options

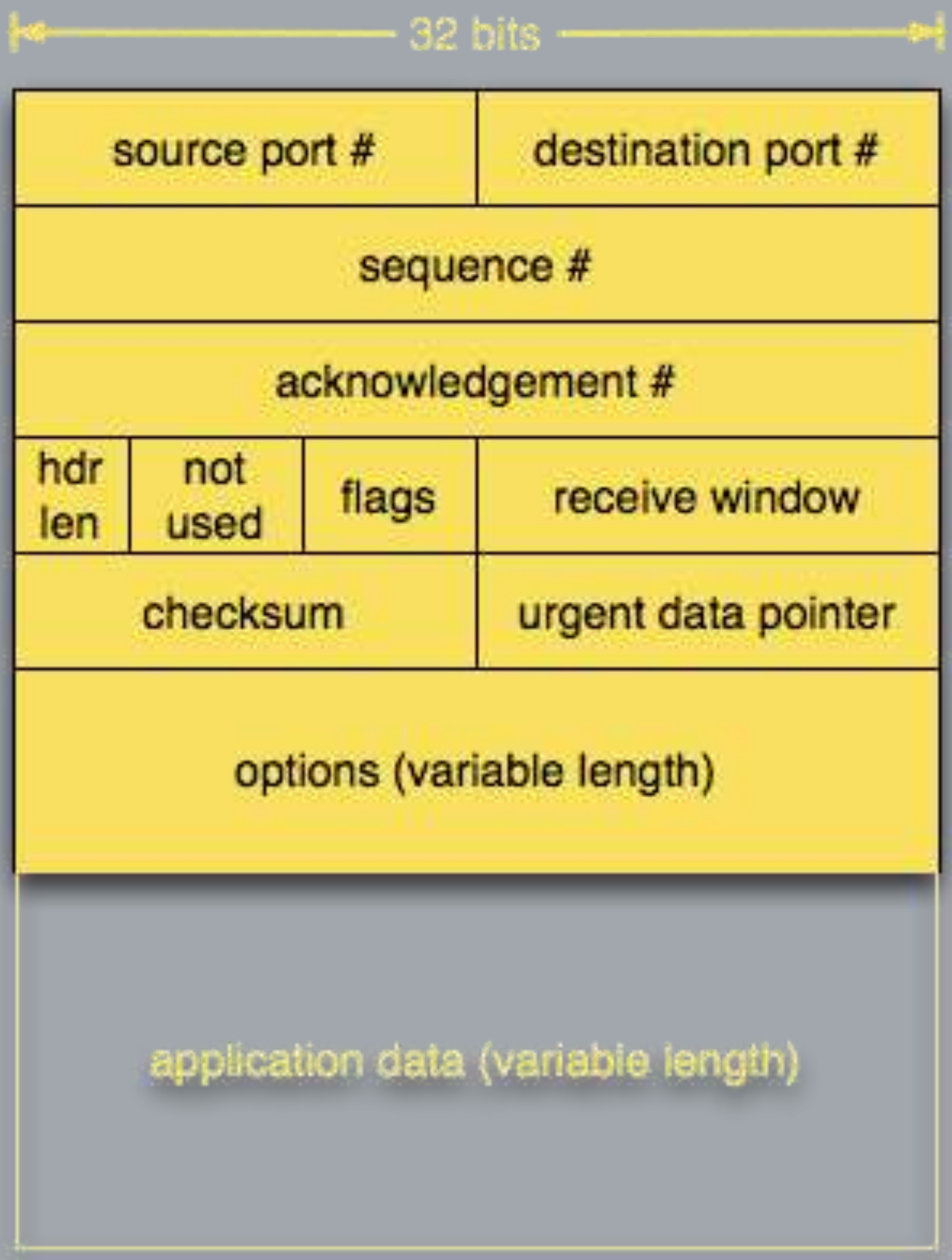


Flag Field

- ACK: Segment acknowledges receipt of another segment (i.e. ack seq# is good)
- Setup and teardown signals
  - SYN: Synchronize seq#
  - FIN: No more data from sender
  - RST: Reset connection
- Rarely used
  - PSH: Push the data to app-layer immediately
  - URG: Urgent data, indicated by Urgent Data Pointer

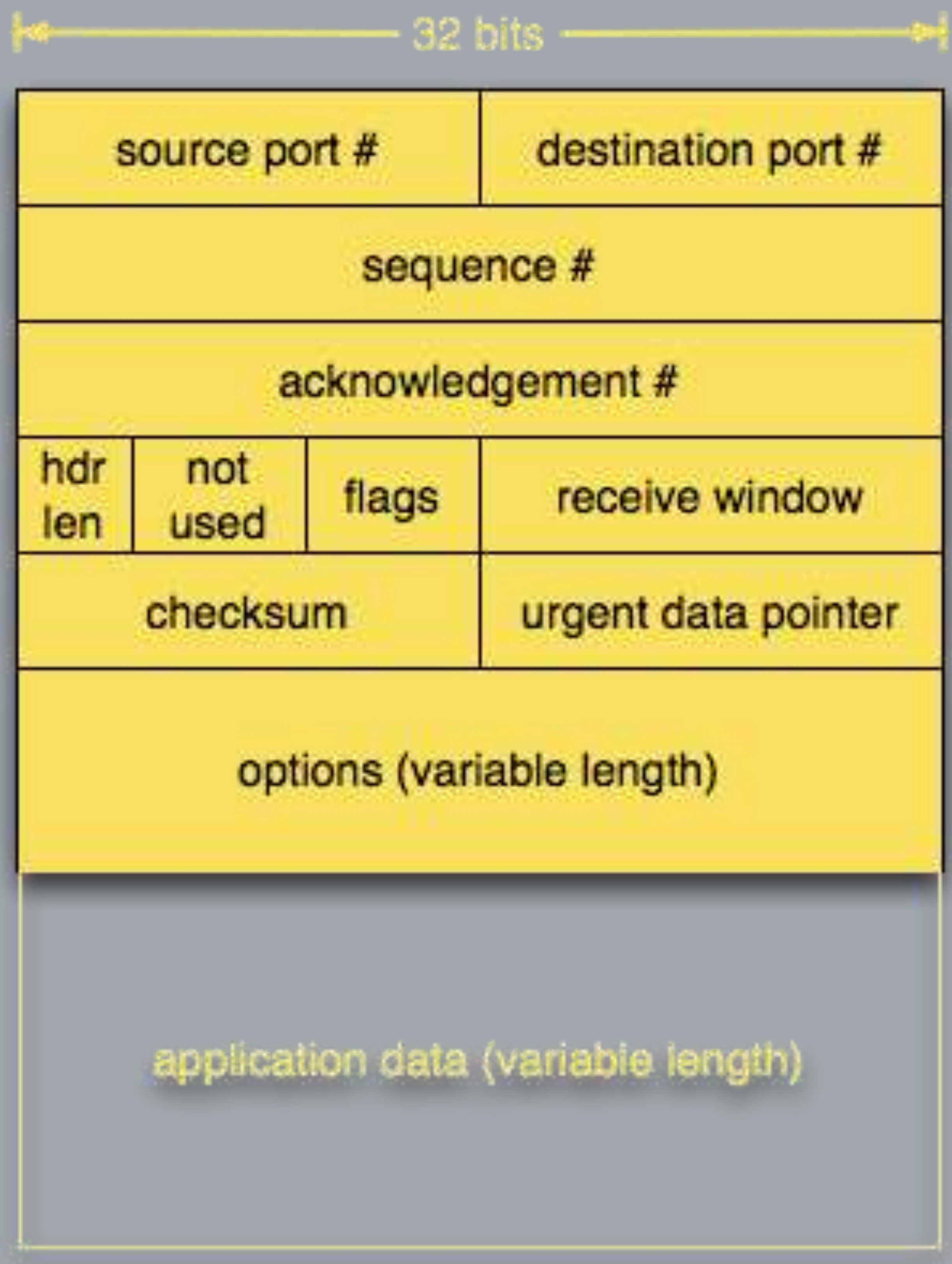
Format

- Receive window
  - #bytes receiver is willing to accept
  - For flow-control



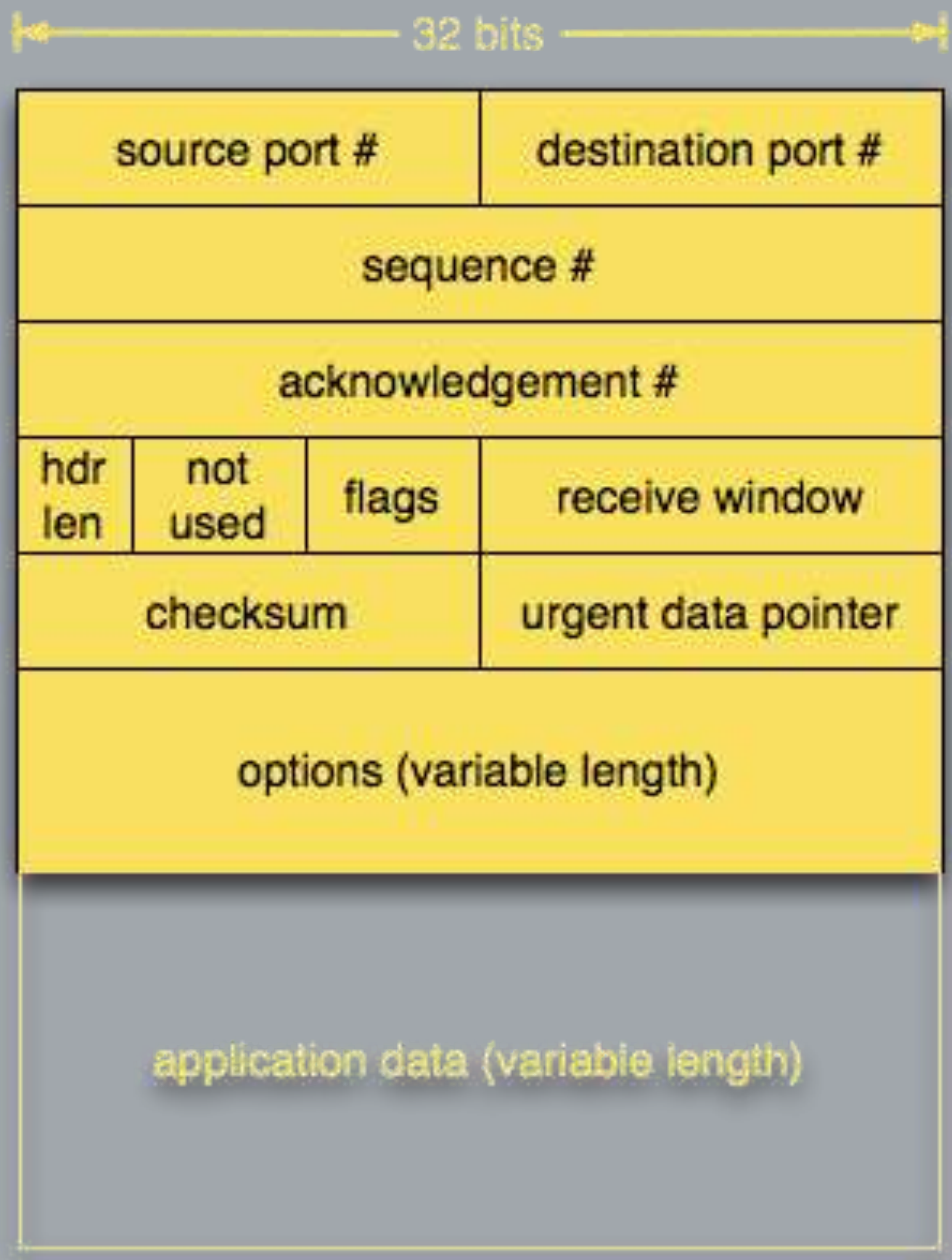
Format

- Checksum
  - Error-checking for header + data.
  - Ensures integrity.
- Urgent data pointer
  - Offset in data field
- Options
  - Time stamps
  - Window scaling factors
  - Negotiating MSS



Format

- Application Data
  - Size limited by MSS
- MSS=Maximum Segment Size
  - Despite name, MSS is most app data that can be carried
- MTU (Max Trans Unit) of lower level generally drives MSS
  - App data + TCP header + IP overhead must fit in MTU
  - MSS is often 1460, 536 or 512 bytes



Outline

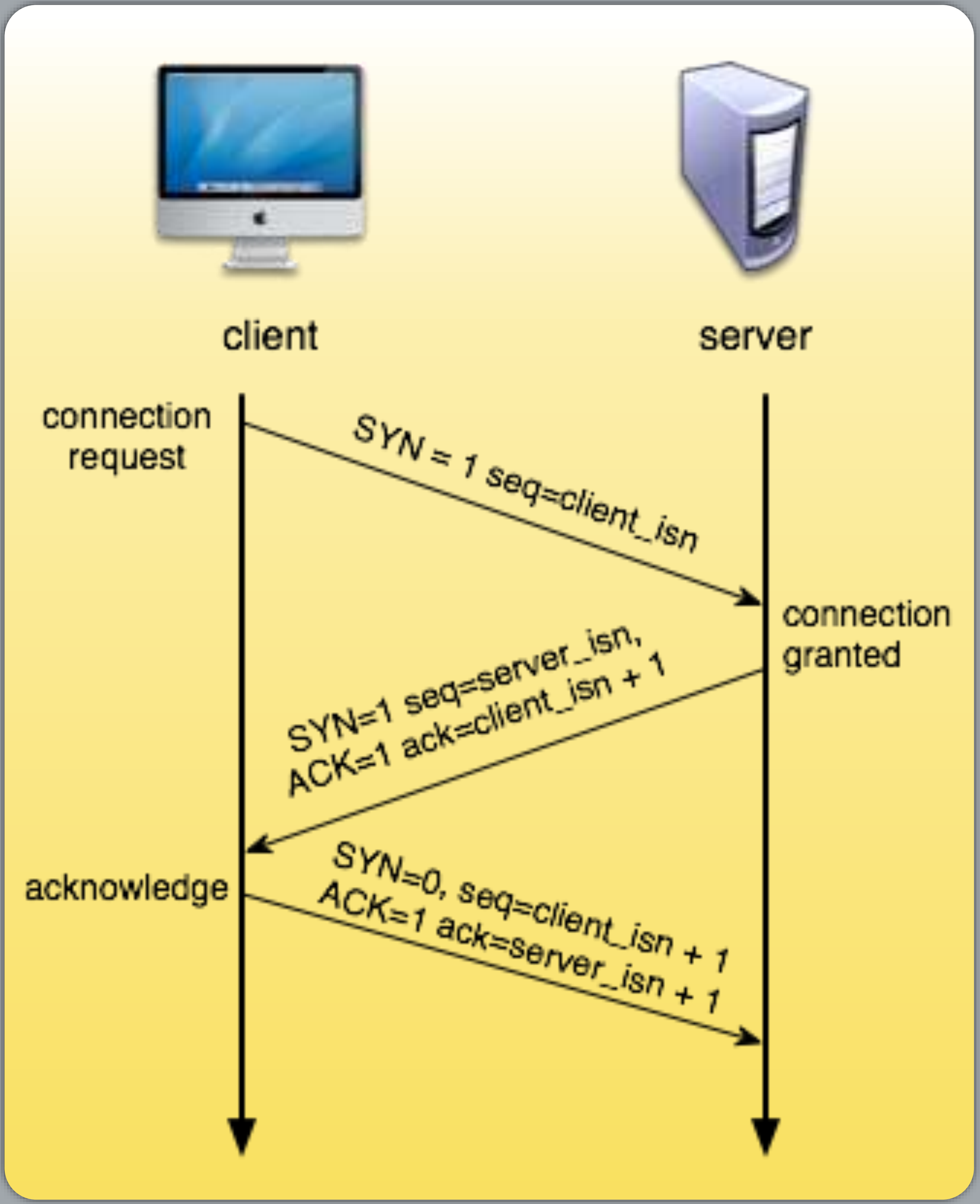
- Connection-oriented Transport: TCP
- Segment Structure
- Connection Management
- Reliable Data Transfer

TCP Connection Management

- Why connection establishment?
  - TCP sender, receiver setup state before exchanging data segments
  - Initialize TCP variables:
    - seq#s
    - buffers, flow control info (RcvWindow)
- Why requiring connection teardown?
  - **Free up state** – remove connection info from memory (control blocks, buffers, sequence tracking).
  - **Release resources** – so ports, memory, and buffers can be reused for new connections.
  - **Ensure reliability** – confirm both sides agree that the connection is finished (no data left in transit).

Three-way Handshake

- Step 1: client sends TCP SYN segment to server
  - specifies initial seq#
  - holds no data
- Step 2: server responds with SYNACK segment
  - server allocates buffers
  - specifies initial seq#
- Step 3: Client replies with ACK
  - May contain data



Question

- TCP specification requires each side of a connection to select an initial starting sequence number at random[1]. Why?
    - Avoid confusion with old connections
    - Security benefit
        - ❖ Predictable ISNs (like always starting at 0 or incrementing by 1) can be exploited for TCP session hijacking.
        - ❖ Random ISNs make it harder for attackers to inject fake packets into a connection.

Question

- Let's say TCP does not exchange initial sequence numbers, and just use 0 as the starting point. What can happen?
  - A client connects to a server → sends data (seq numbers 0,1,2,...).
  - Connection closes, but some stray packets are still in the network (delayed).
  - Client reconnects immediately to the same server (same IP/port tuple).
  - New connection also starts at 0 → delayed old packets fall into the valid sequence number range and get accepted as new data.

Result:

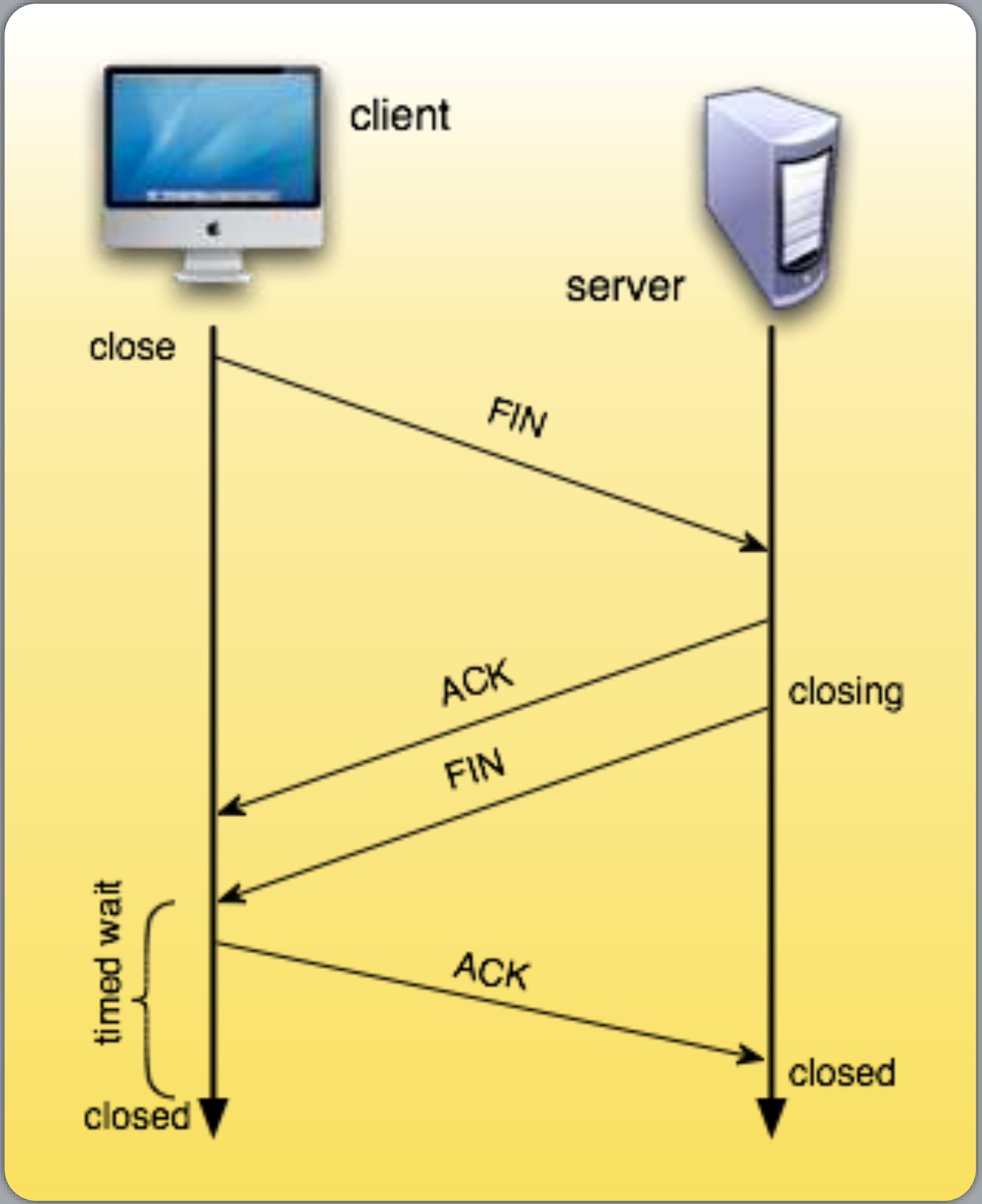
- Data corruption or wrong application behavior.

Example

- If a host receives a TCP SYN segment to a closed port (no process is listening)
  - The host replies with a TCP RST (Reset) segment.
  - This tells the sender:
“That port is closed — no connection will be established.”
  - RST is TCP’s way of signaling an error condition.

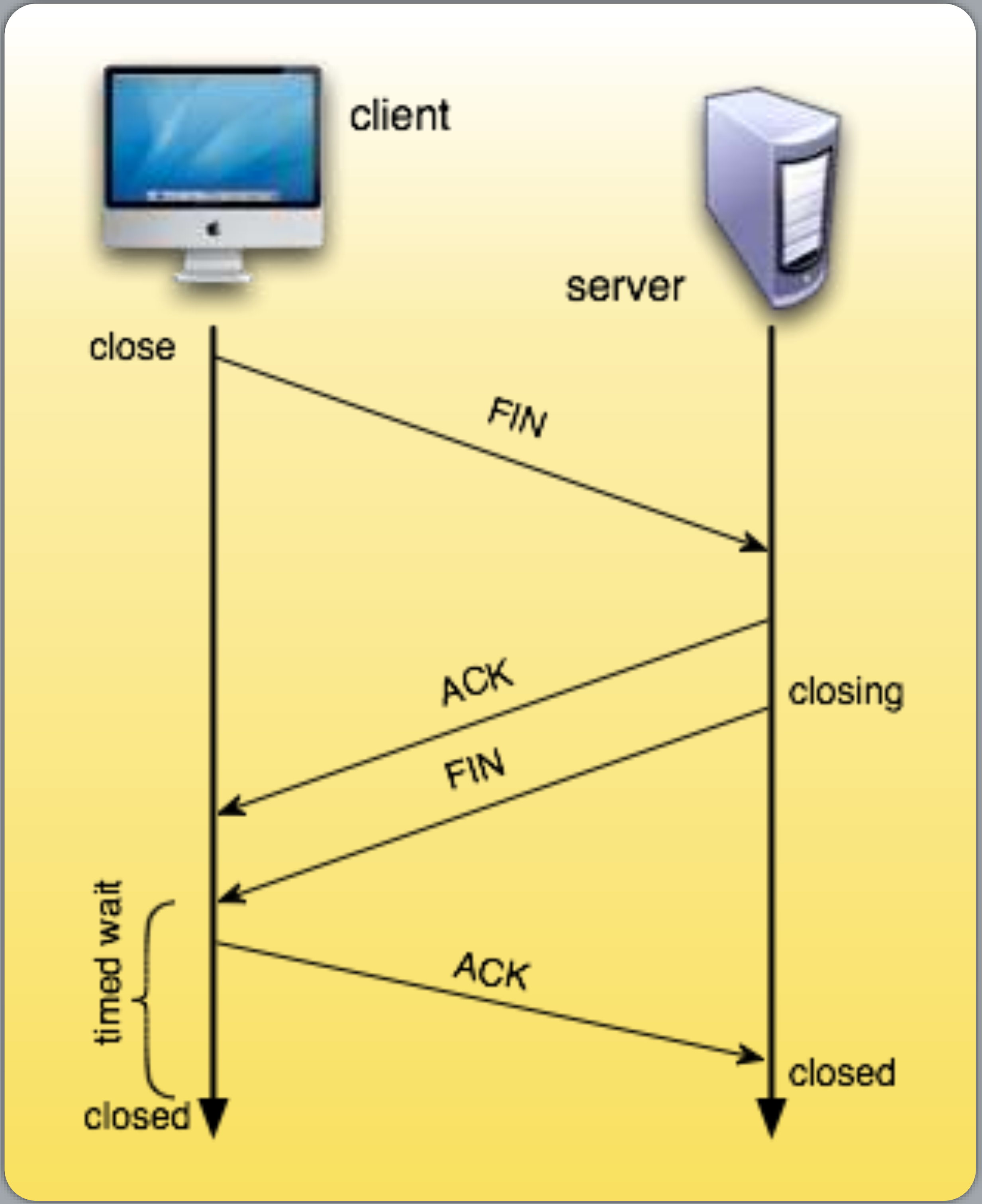
Closing a Connection

- Step 1: Client sends TCP FIN segment to server
- Step 2: Server receives FIN, responds with ACK
  - Closes connection
  - Sends FIN
- Recall: Connection is bi-directional, needs to be shut down from each side



Closing a Connection

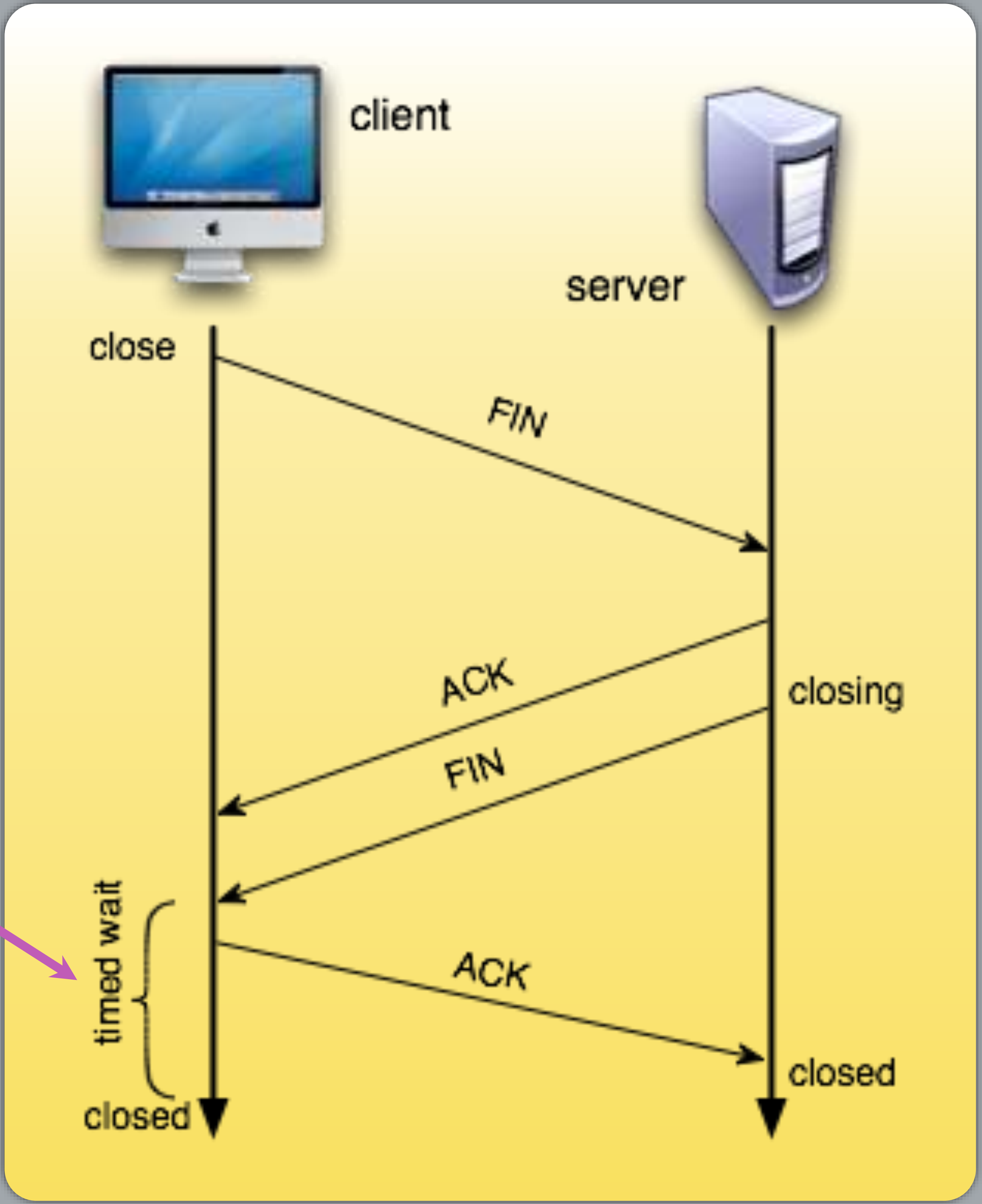
- Step 3: client receives FIN, replies with ACK
  - Client enters “timed wait”
  - Will respond to FIN with ACK
  - 240 seconds (2 * max segment lifetime – MSL[1])
- Step 4: server receives ACK, closes connection



- [1]MSL = longest time a TCP segment is allowed to exist in the network before being discarded.
- Typical MSL = 120 seconds

Question

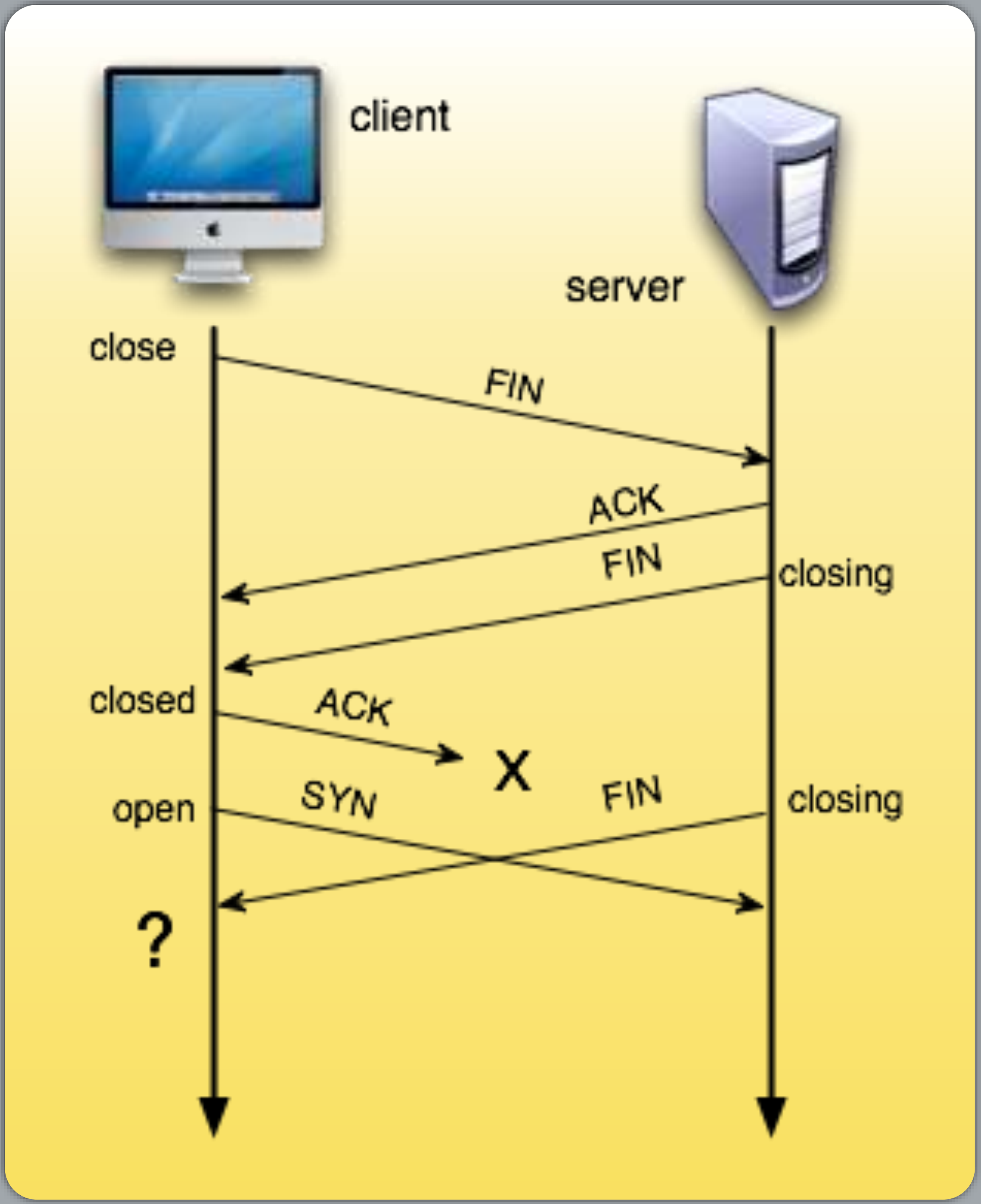
- Why does client enter “timed wait” state before “closed,” even after receiving FIN from server?



Answer

Result:

- If ACK from the client is lost, and the client does not wait?
  - Client may open the same connection again (same pair of port#s)
  - Receives FIN from earlier incarnation of connection
  - Immediately initiate closing of the later incarnation
- Any stray packets from the old connection have expired (1 × MSL).



Outline

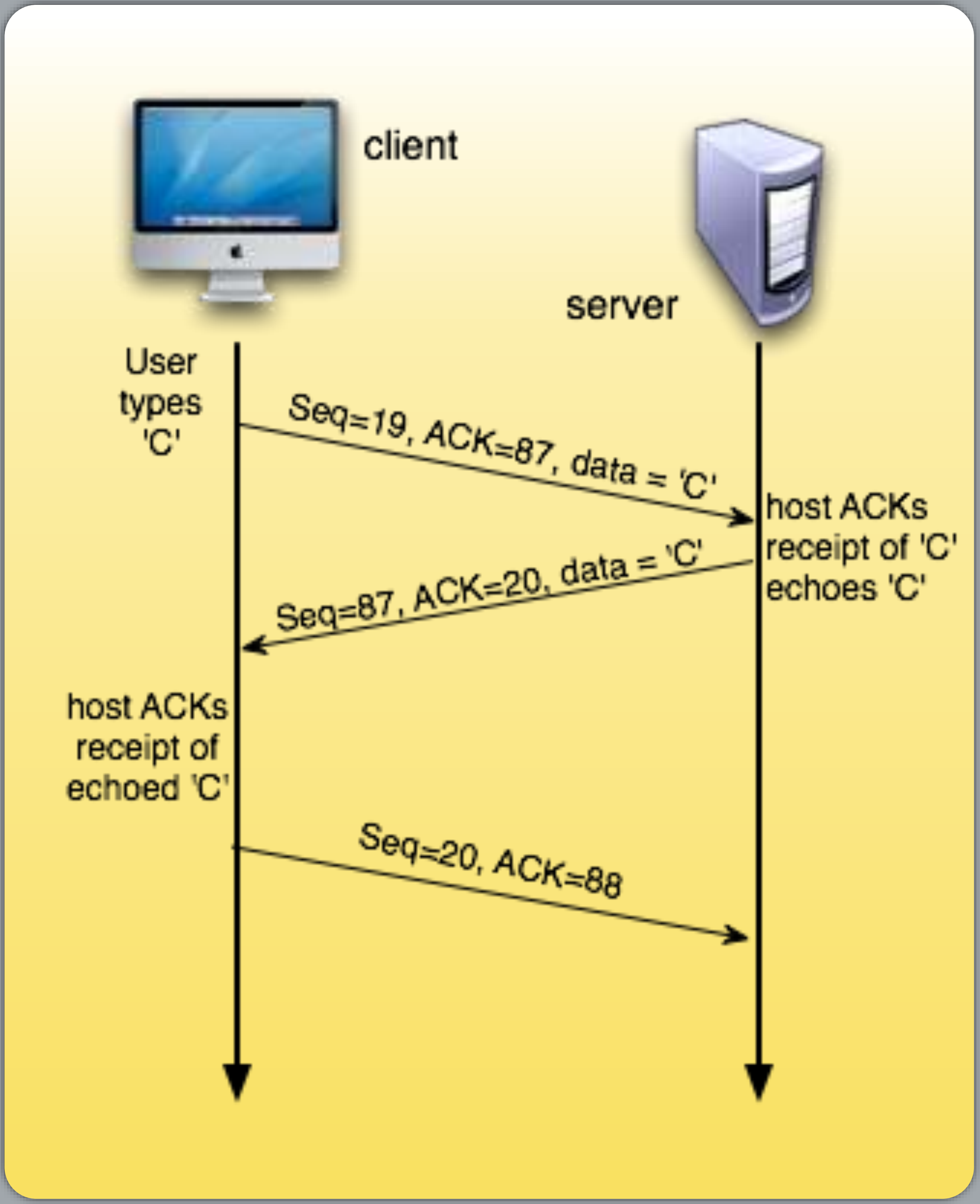
- Connection-oriented Transport: TCP
- Segment Structure
- Connection Management
- Reliable Data Transfer (RDT)

TCP RDT

- TCP creates RDT service on top of IP's unreliable service
  - Pipelined segments
  - Cumulative acks
  - Retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Duplicate acks
- Initially, we consider simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control
  - assume RTT is estimated somehow

TCP Seq# and ACKs

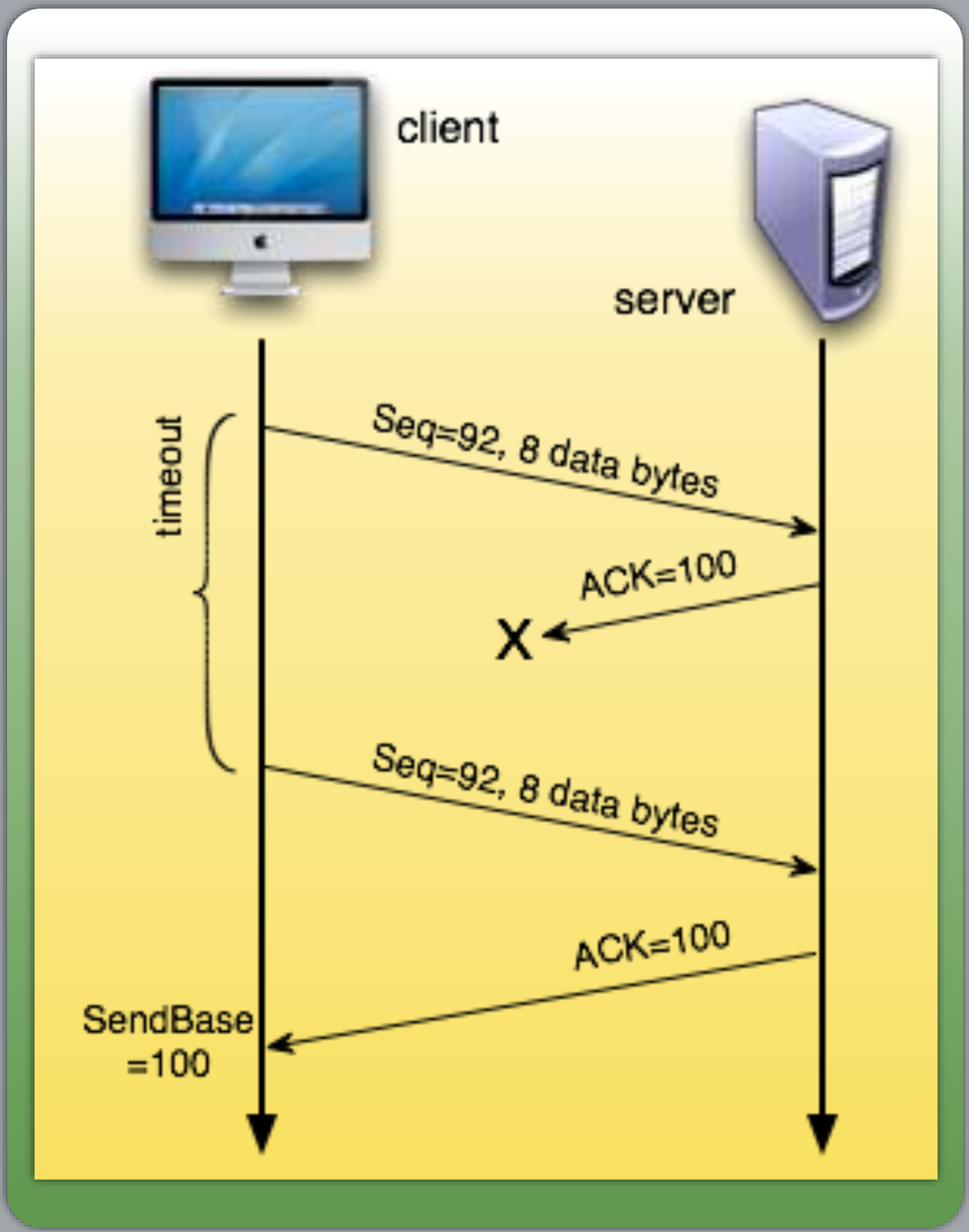
- Sequence numbers:
  - byte stream "number" of first byte in segment's data
- ACKs:
  - seq # of next byte expected from other side
  - cumulative ACK
    - acknowledges bytes up to the first missing byte in the stream
  - piggybacked
    - with data, if possible



TCP Sender Events

- Data received from app:
  - Create segment
  - seq# is byte-stream number of first data byte in segment
  - Send, if allowed by congestion & flow-control
  - start timer if not already running (think of timer as for oldest unacked segment)
  - expiration interval: TimeOutInterval
- Timeout:
  - retransmit segment that caused timeout
  - restart timer
- ACK received:
  - If acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are outstanding segments

Retransmission Scenarios

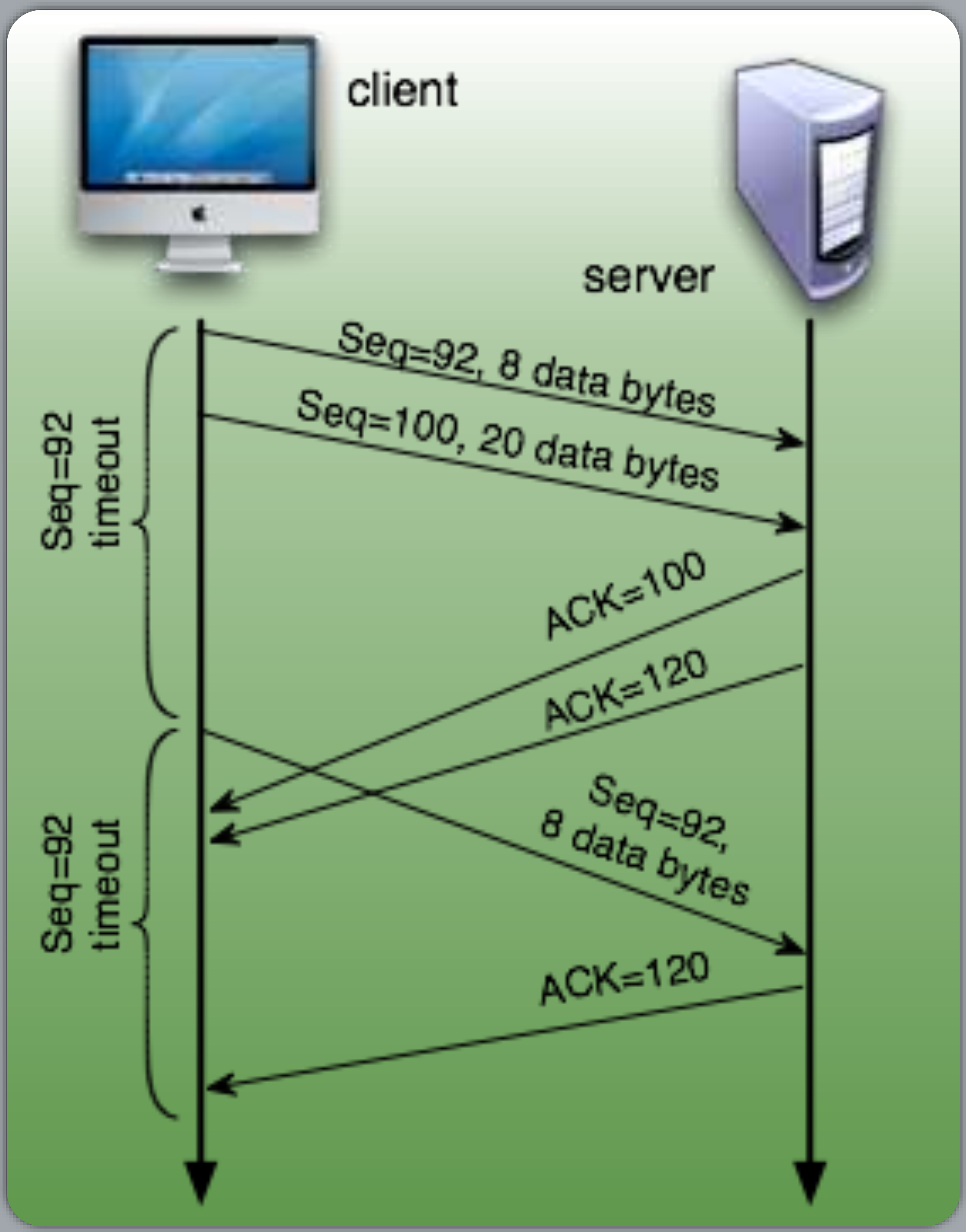


Lost ACK

- What does server do when it gets the retransmitted segment?

- Discard it! Expected seq# is 100. Since 92 < 100 it knows this is duplicate data

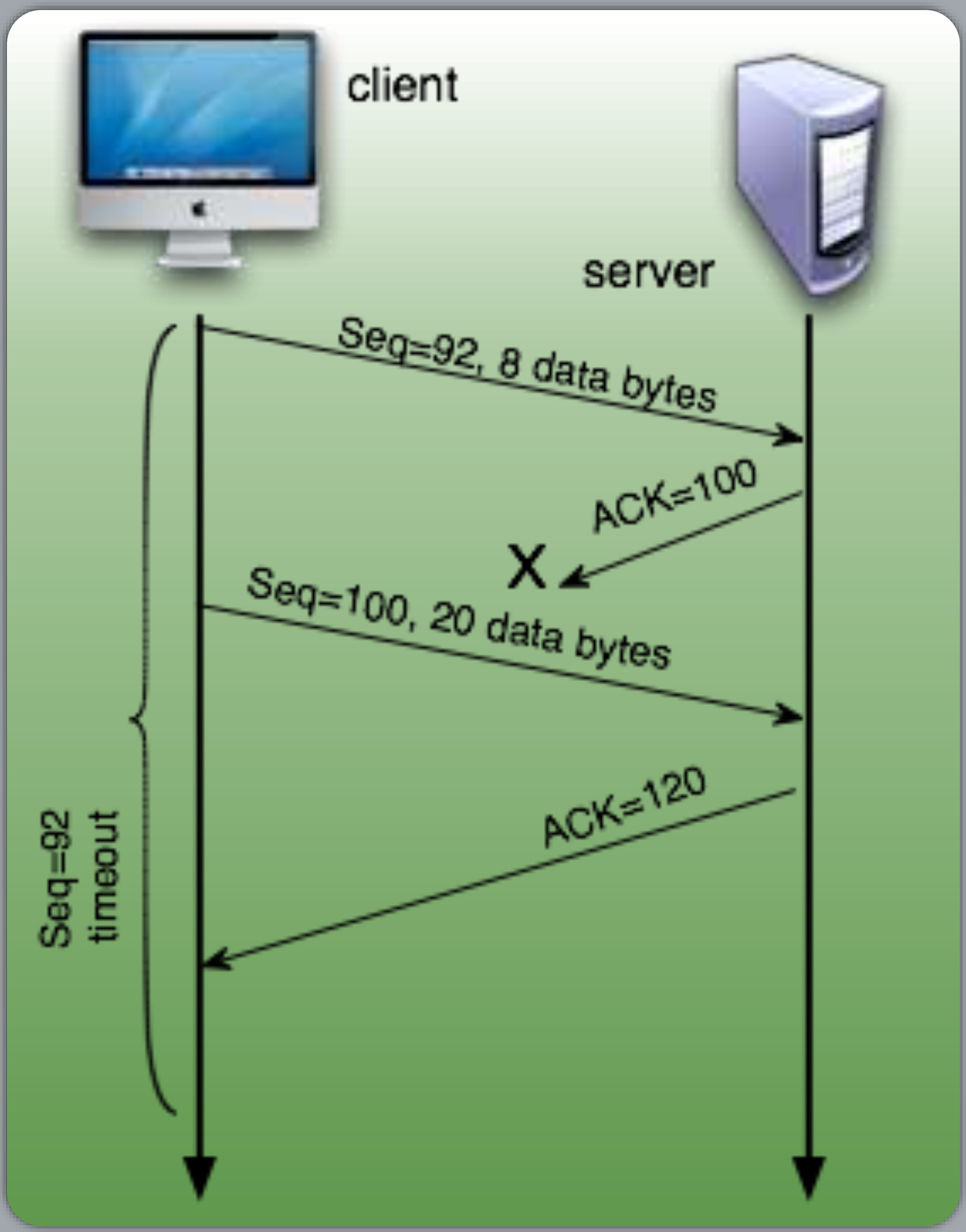
Retransmission Scenarios



Premature Timeout

- Will client retransmit 2nd segment (Seq=100)?
- No! ACK=120 arrives before the timeout

Retransmission Scenarios



Cumulative ACK

- Host hasn't received ACK for first segment, so why doesn't it retransmit?
- ACK=120 is cumulative, which means everything up to byte 120 has been received (including 92)

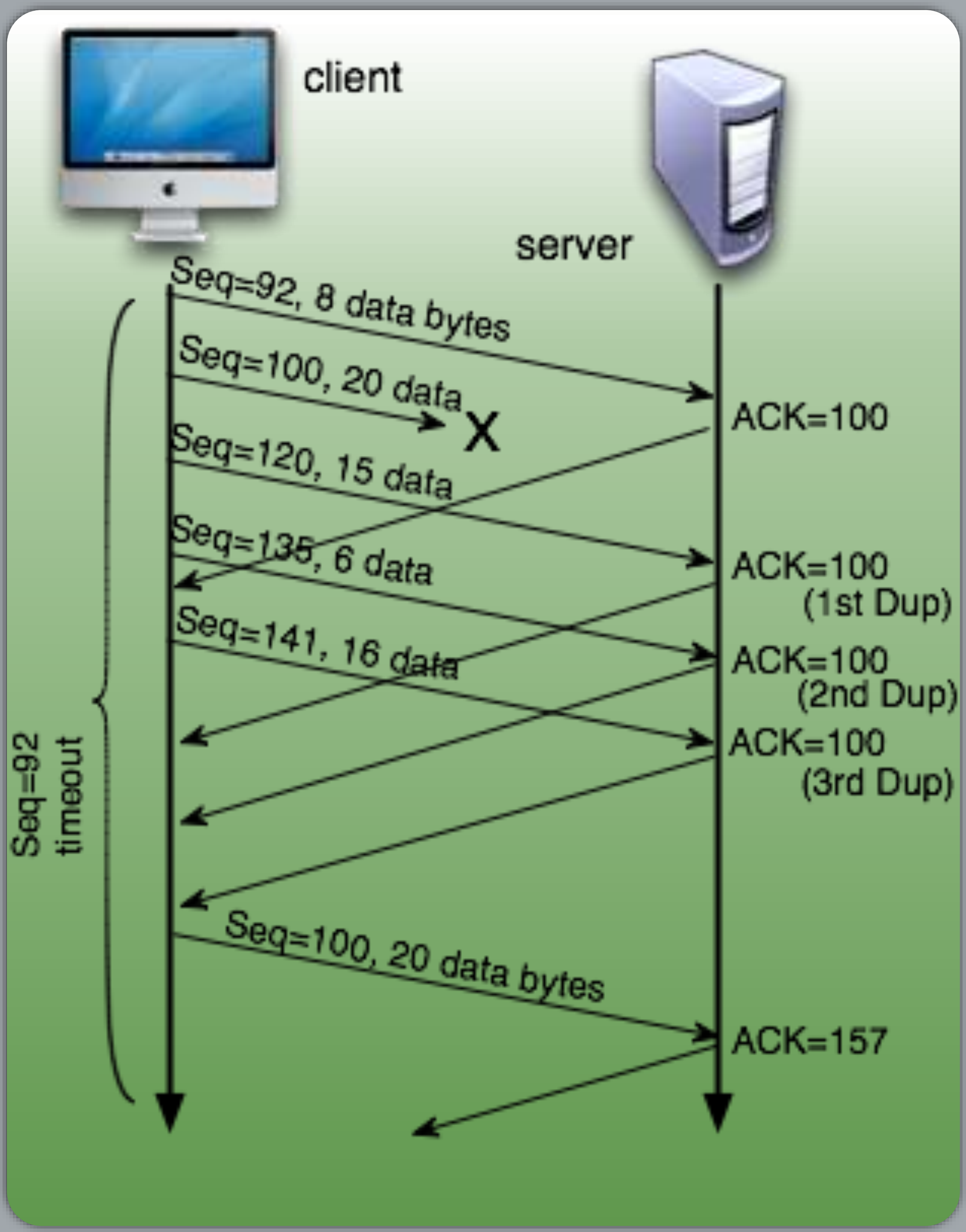
ACK Generation

Event at receiver	Receiver Action
Arrival of in-order segment with expected seq#. All data up to seq# already ACKed	*Delayed ACK.* Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq#. One other segment has ACK pending	Immediately send single cumulative ACK for both in-order segments
Arrival of out-of-order segment higher than expected seq#. Gap detected	Immediately send *duplicate ACK,* indicating seq# of next expected byte
Arrival of segment that partially or completely fills gap	Immediately send ACK, provided that segment starts at lower end of gap

Fast Retransmit

- Time-out period often relatively long:
  - Long delay before resending lost segment
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
- If sender receives 3 duplicate ACKs, it supposes that segment after ACKed data was lost:
  - Fast retransmit: resend segment before timer expires

Scenario



Fast Retransmission

- Sender doesn't have to wait for a timeout to notice probable loss of seq=100 segment
- Sort of a NACK

Why 3?

- Why 3 duplicate ACKs? Why not do fast retransmit after the first duplicate ACK for a segment is received?

- If n+1 and n+2 (or n+3) are just reordered, then waiting for 2 duplicate ACKs will not retrigger retransmission
    - Voodoo constant