ECE 46300 - Fall 2024

Transport Layer IV

Vishal Shrivastav

Transmission Control Protocol (TCP)

- **Reliable, In-order Delivery to Applications**
    - Application Multiplexing
    - ACKs and Retransmissions
    - Flow Control
    - Congestion Control
- **Byte Stream Abstraction**
    - Communicates with the applications using a byte stream and *not* packets
- **Connection-oriented**
    - Pairwise <sender, receiver> connection is established before sending data
    - Used to maintain connection-specific state, e.g., initial seq num, window scaling factor, window size, etc.

Transmission Control Protocol (TCP)

- **Reliable, In-order Delivery to Applications**
  - Application Multiplexing
  - ACKs and Retransmissions
  - Flow Control
  - Congestion Control
- **Byte Stream Abstraction**
  - Communicates with the applications using a byte stream and *not* packets
- **Connection-oriented**
  - Pairwise <sender, receiver> connection is established before sending data
  - Used to maintain connection-specific state, e.g., initial seq num, window scaling factor, window size, etc.

TCP Connection Set up

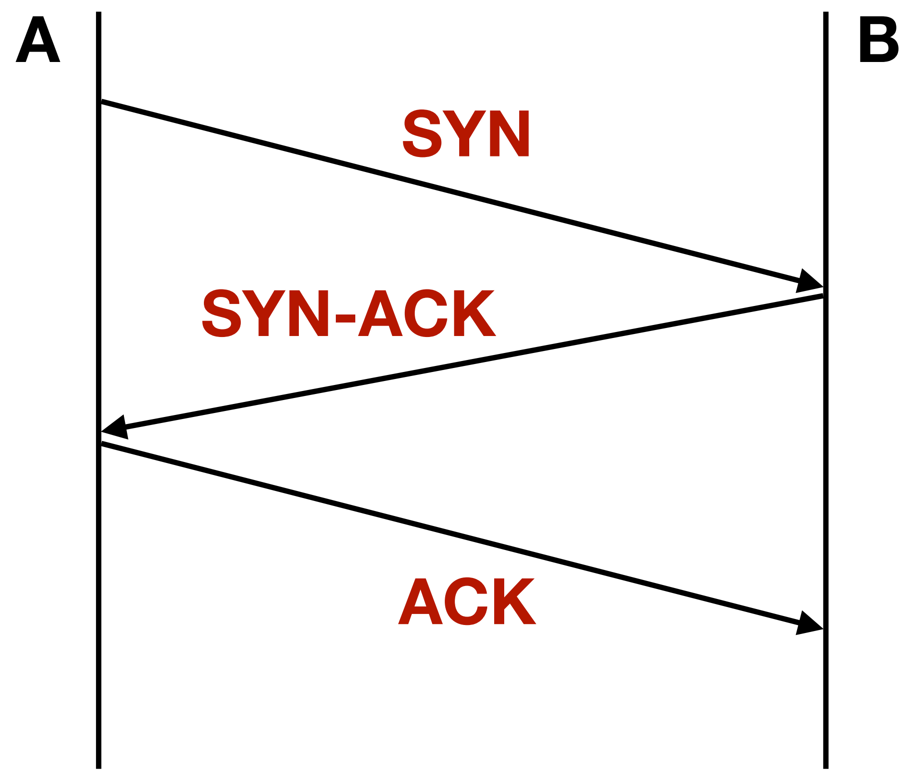
- **A TCP connection is identified using 5-tuple**
    - <source IP, destination IP, source Port, destination Port, Protocol=TCP>
- **Connection set up used to exchange <u>initial sequence number (ISN)</u>**
    - Also used to exchange **window scaling factor** (used in flow control)
- A receiver needs to know the ISN of the sender to figure which sequence number marks the start of the byte stream being received
- Each endpoint choses its ISN **randomly**
    - **Why endpoints not just choose ISN as 0 (or some fixed number)?**

Why random ISN?

- **Practical Issue 1: Port numbers are re-used**
  - Suppose a TCP connection between A and B is closed
  - … and a new TCP connection is opened between A and B with the same source and destination port numbers as the previous connection
  - But suppose a packet with sequence num 100 from previous TCP connection was still in-flight
  - B will think packet belongs to new TCP connection if ISN of new connection was 0 **… less chance of this happening if ISN is chosen randomly**
  - e.g., if random ISN for new TCP connection was >100, then not an issue in the above example

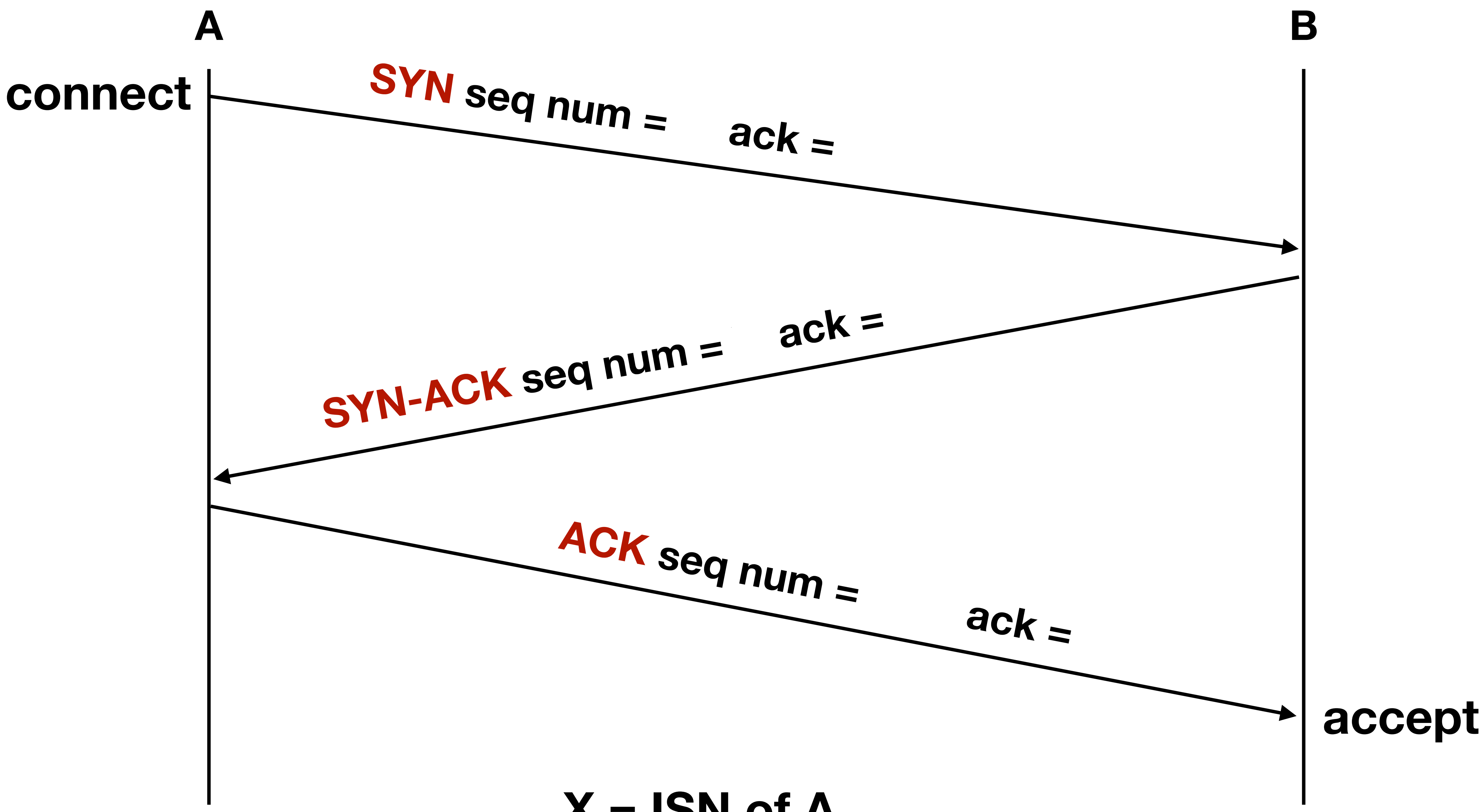
- **Practical Issue 2: TCP sequence prediction attack**
  - Easy for attacker to guess a valid sequence number if ISN is always 0
  - Can send counterfeit packets to receiver

TCP Connection Set up



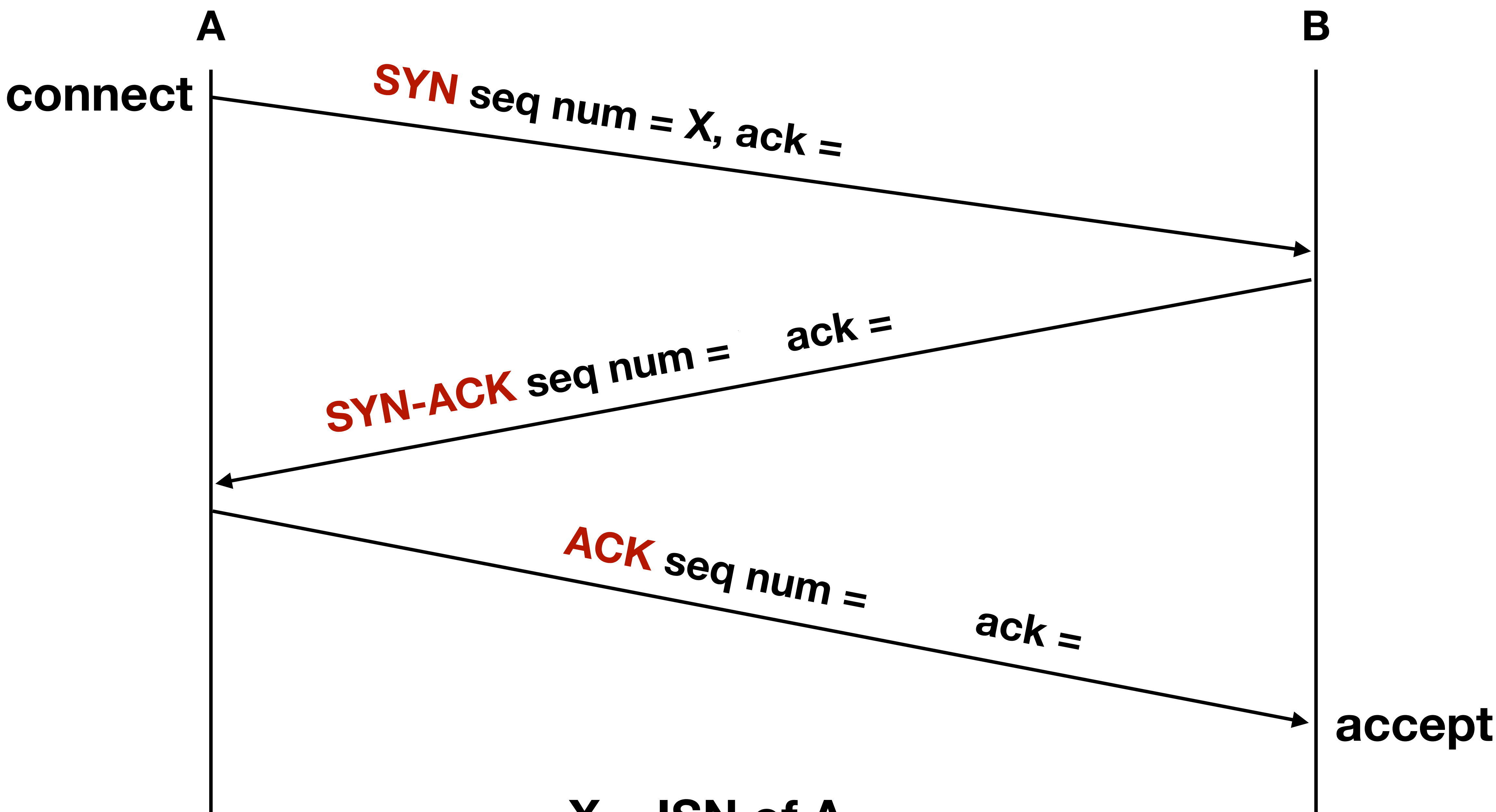
- **3-way Handshake**
  - Host A sends a **SYN** ("Synchronize") packet to B
  - Host B returns a SYN acknowledgement packet **(SYN-ACK)**
  - Host A sends an **ACK** packet to acknowledge the SYN-ACK
  - **What should be the seq and ack numbers for these packets?**

3-Way Handshake



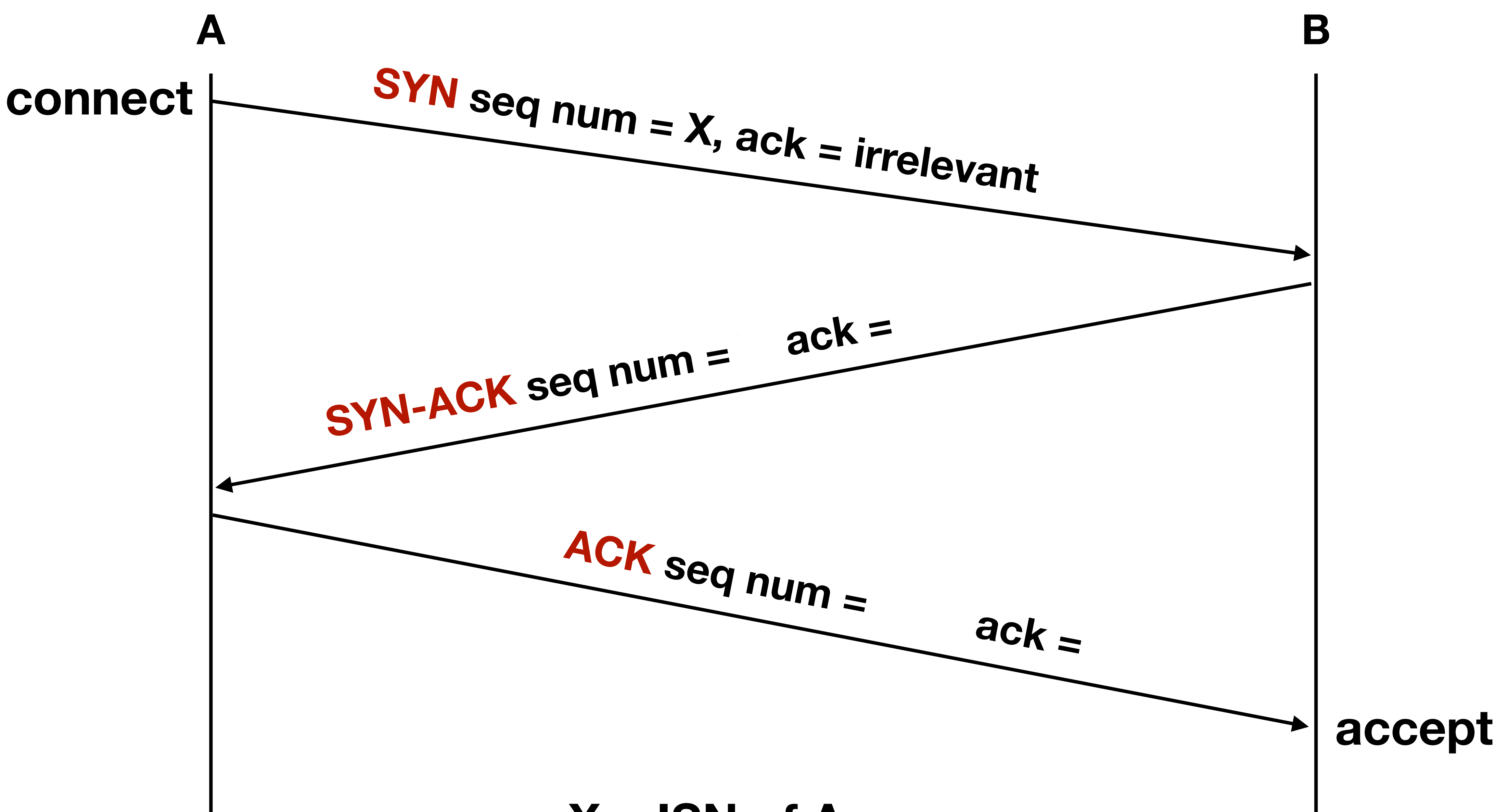
X = ISN of A
Y = ISN of B

3-Way Handshake



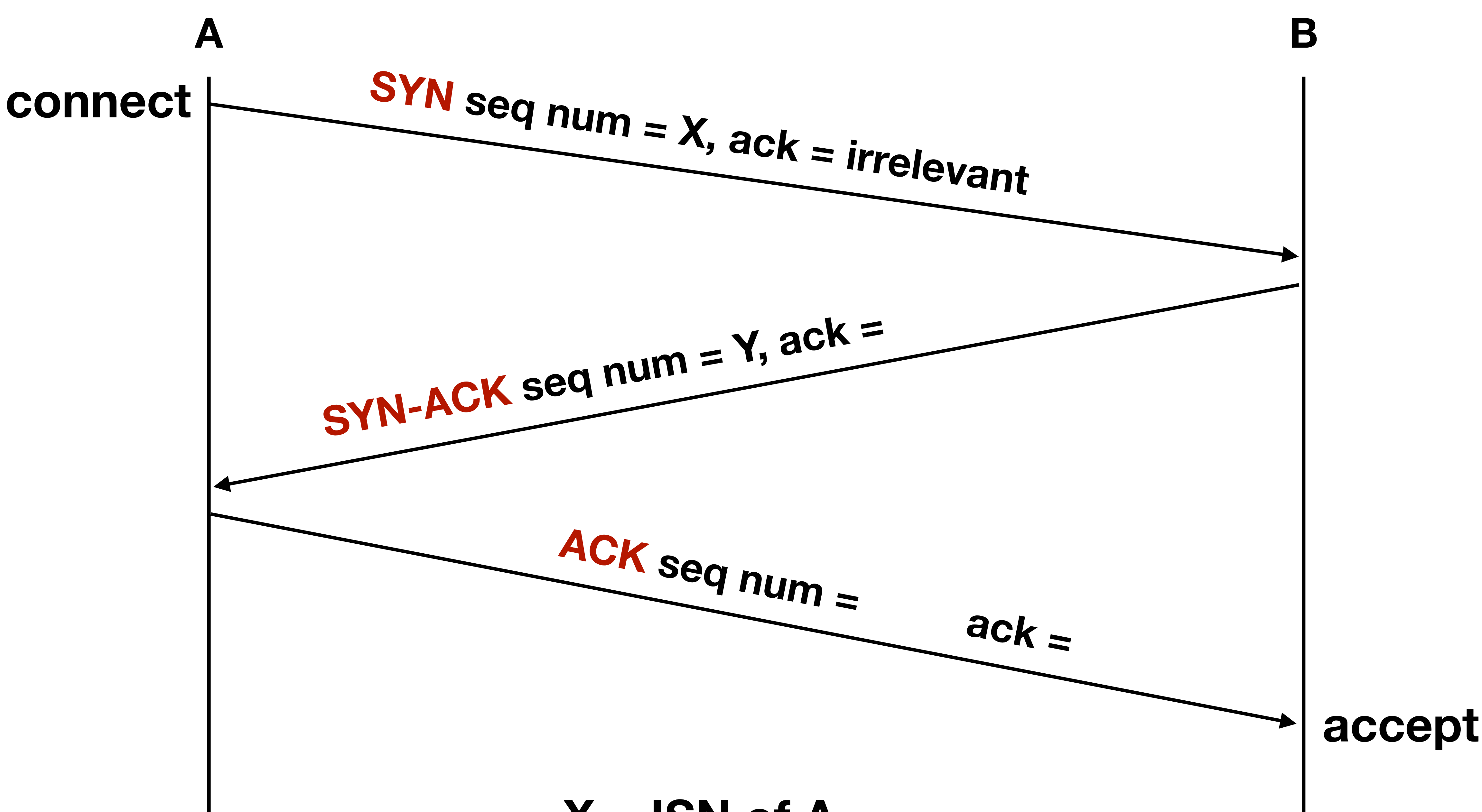
X = ISN of A
Y = ISN of B

3-Way Handshake



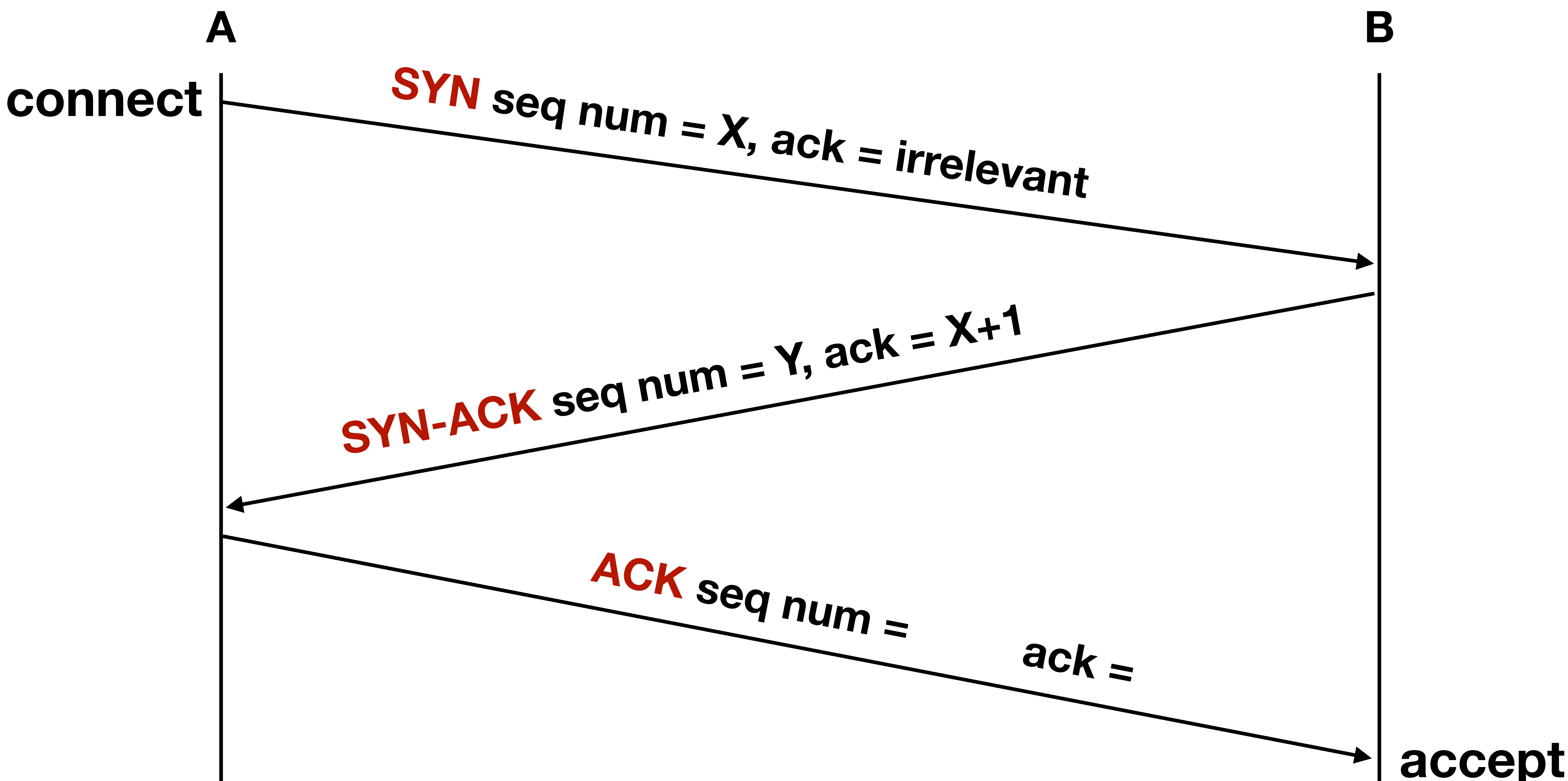
X = ISN of A
Y = ISN of B

3-Way Handshake



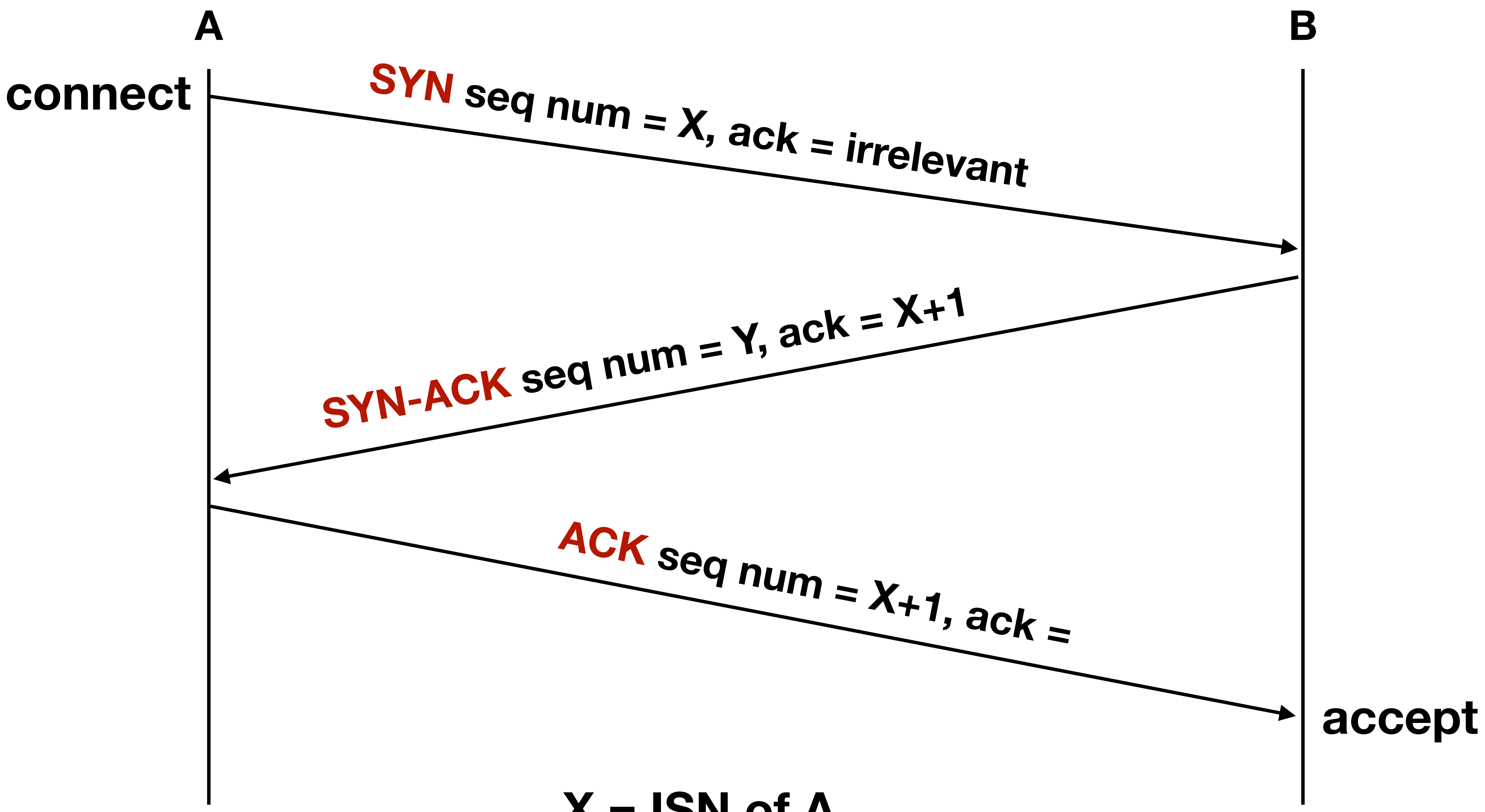
X = ISN of A
Y = ISN of B

3-Way Handshake



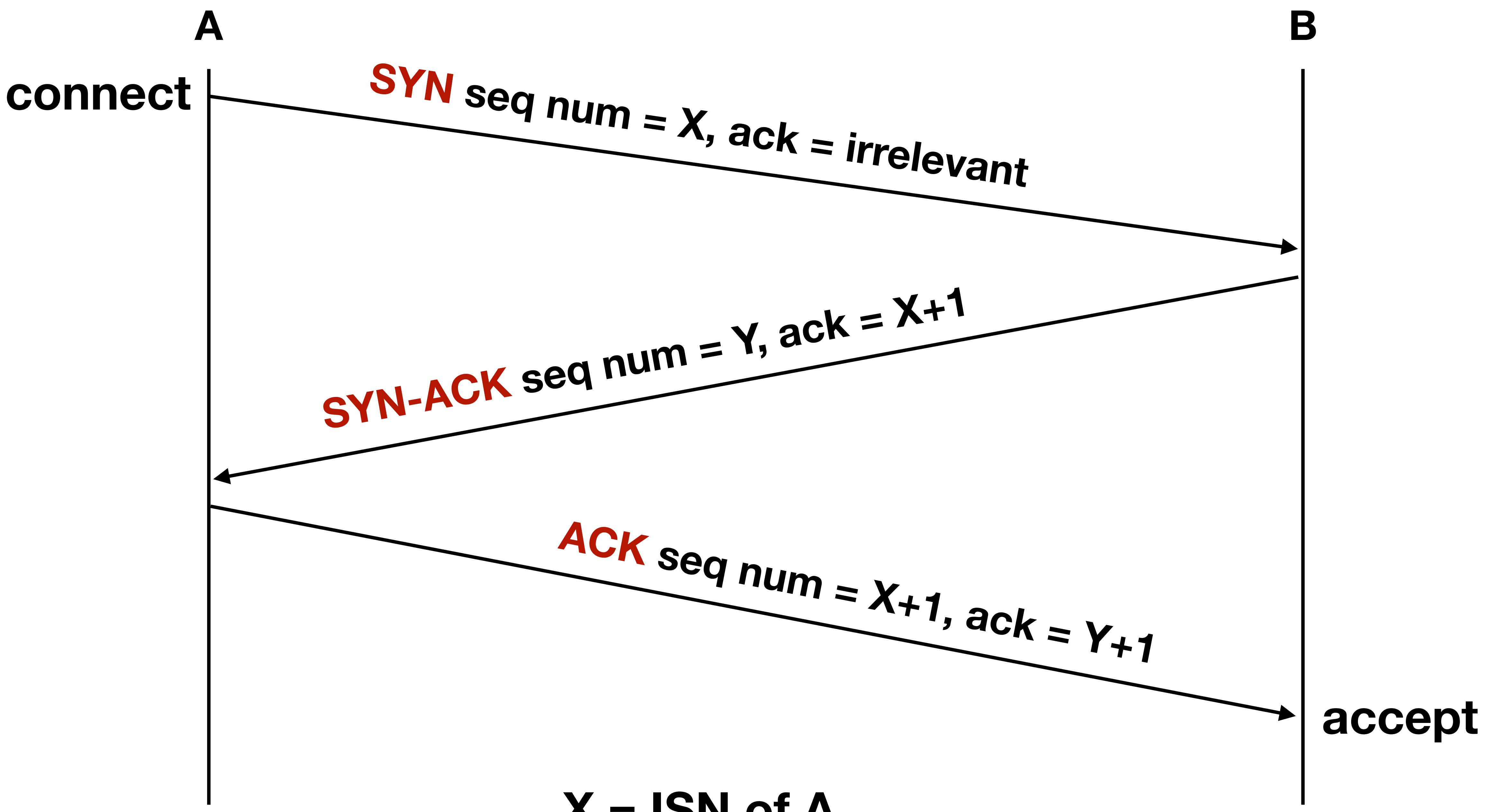
X = ISN of A
Y = ISN of B

3-Way Handshake



X = ISN of A
Y = ISN of B

3-Way Handshake



X = ISN of A
Y = ISN of B

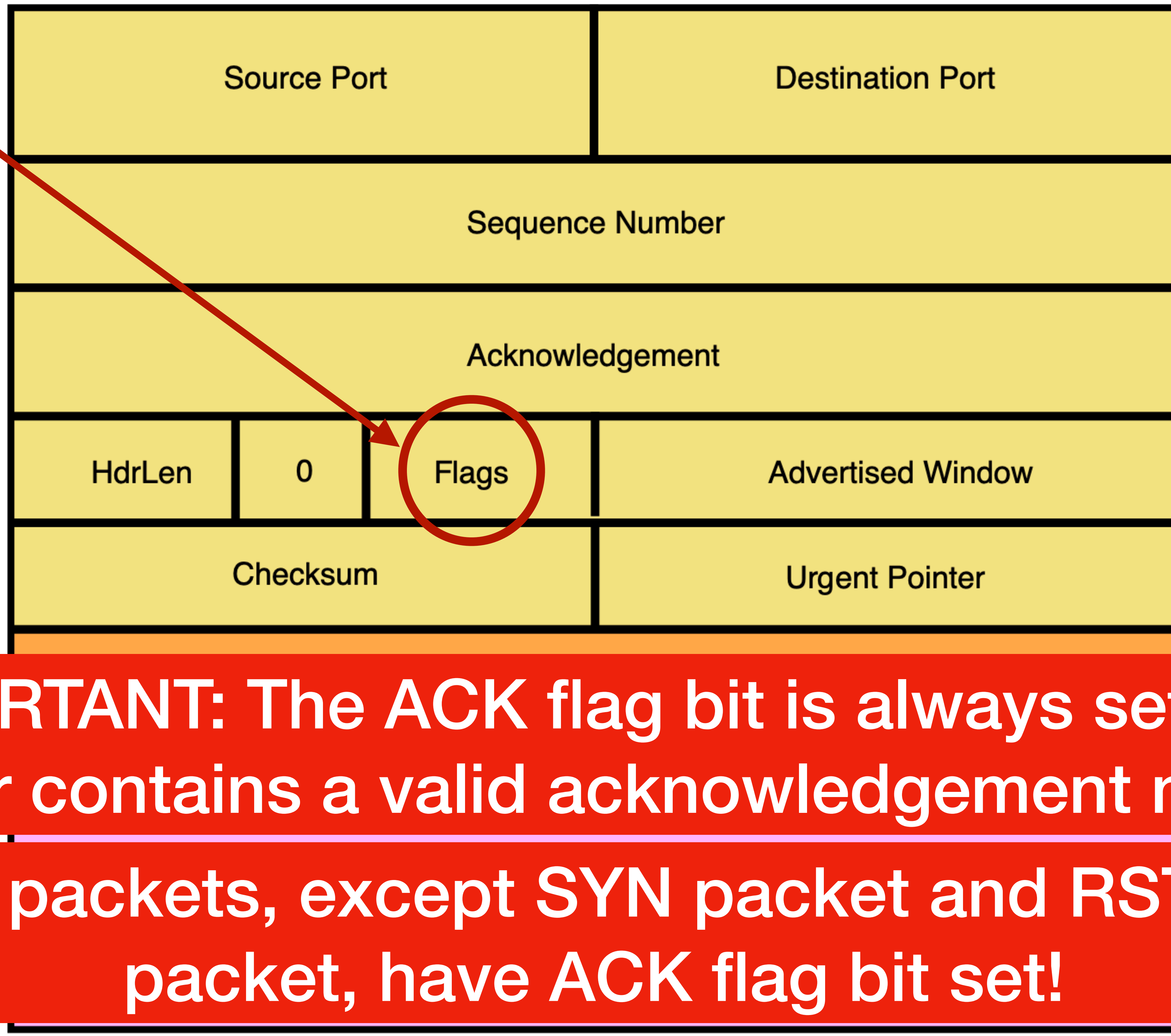
How to recognize Special Packets?

- How does TCP distinguish special packets (e.g., SYN, SYN-ACK) from normal data packets?
  - **Answer:** Using **TCP Flags**

TCP Flags

9-bits

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS

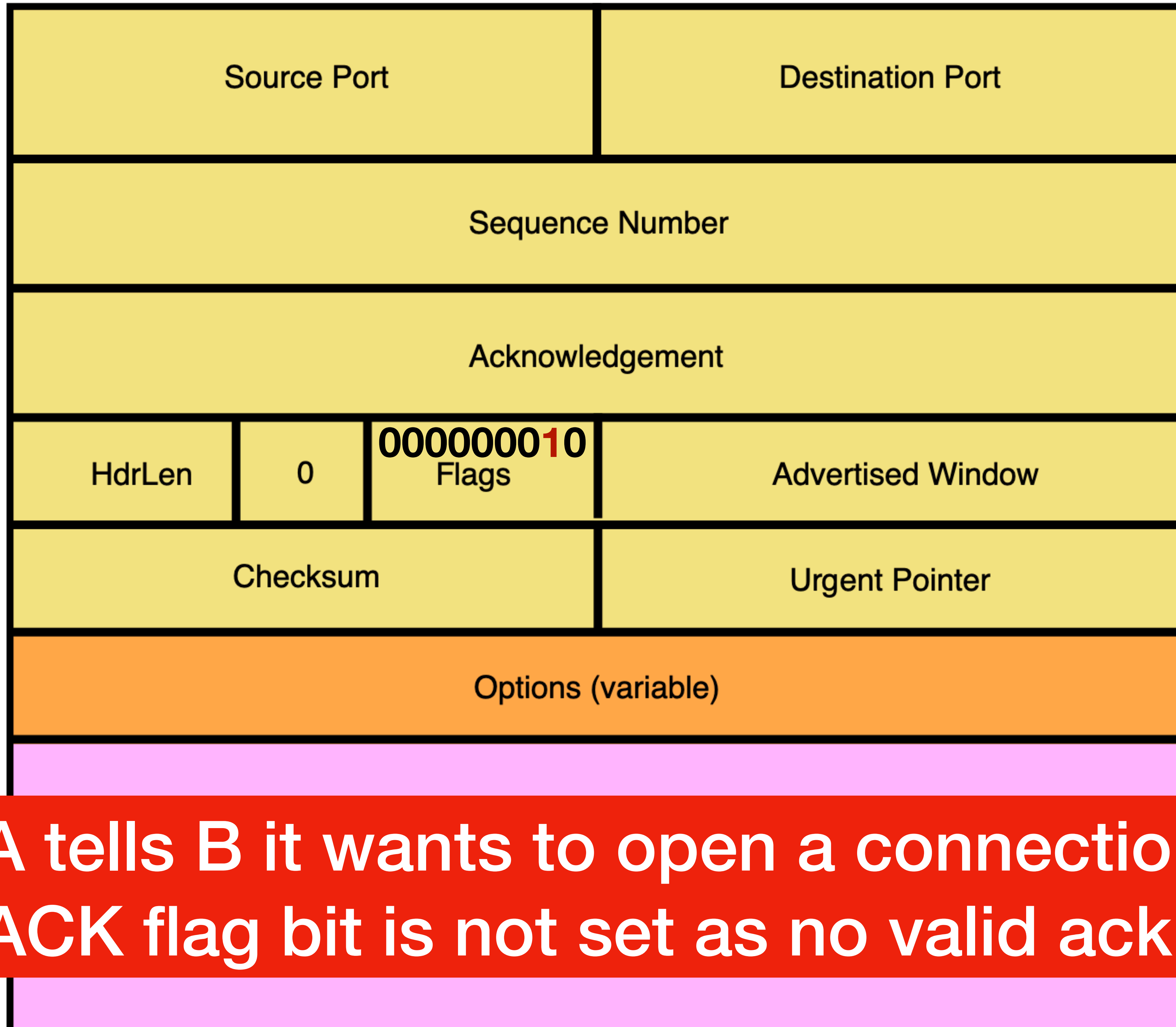


IMPORTANT: The ACK flag bit is always set if the header contains a valid acknowledgement number

All TCP packets, except SYN packet and RST-REPLY packet, have ACK flag bit set!

Step 1: SYN Packet from A

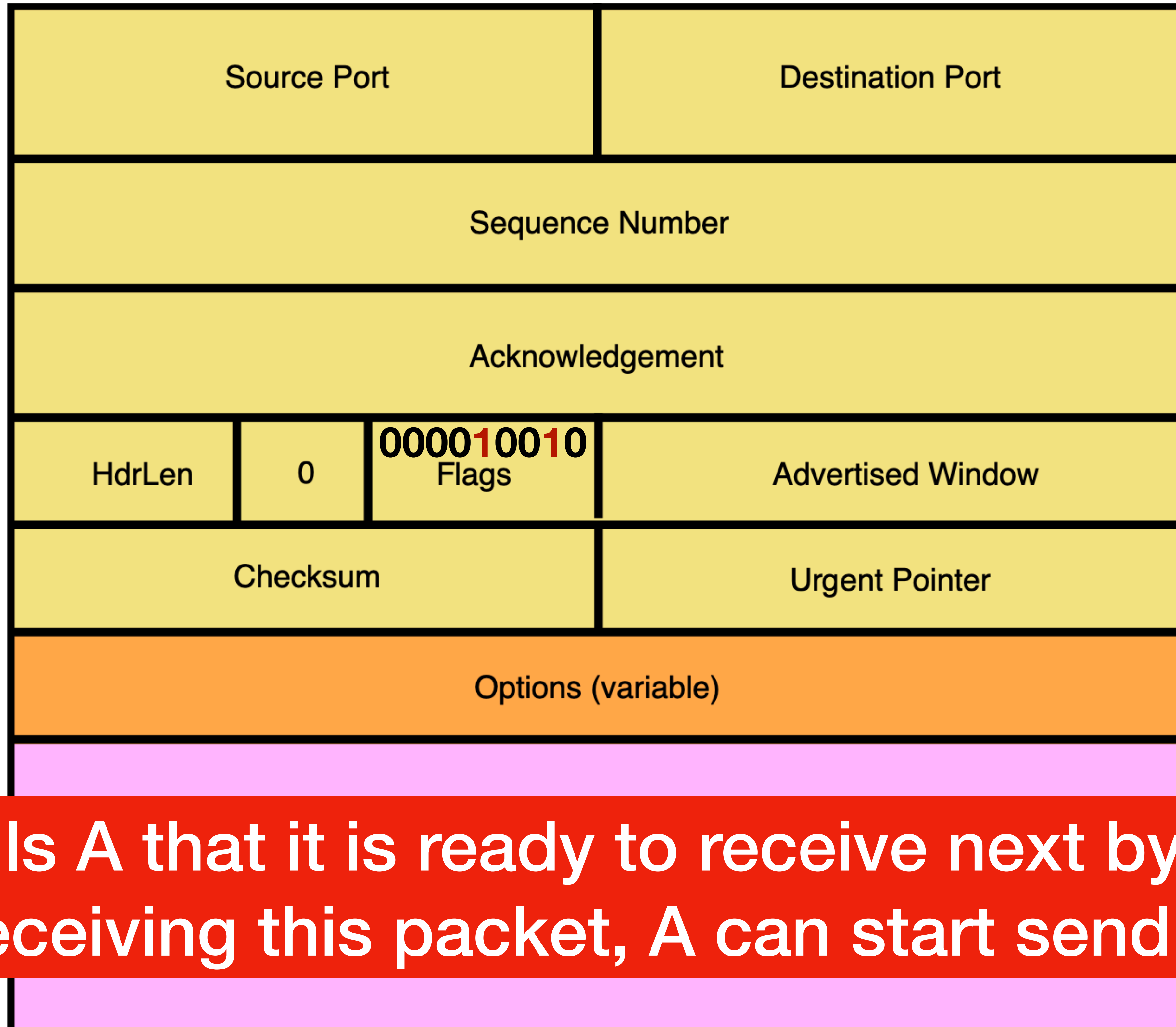
bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS



A tells B it wants to open a connection
NOTE: ACK flag bit is not set as no valid ack number

Step 2: SYN-ACK Packet from B

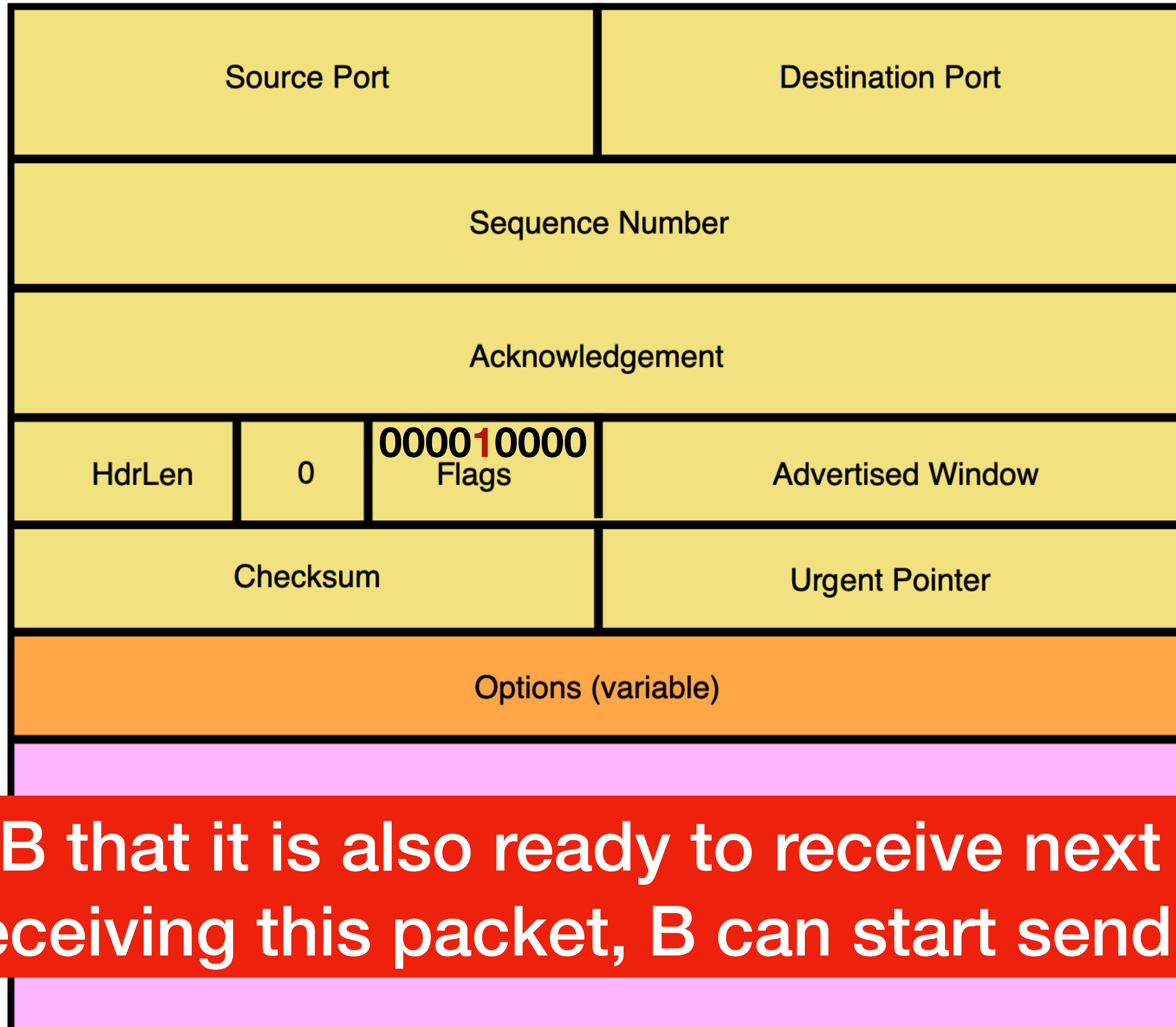
bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS



B tells A that it is ready to receive next byte …
… on receiving this packet, A can start sending data

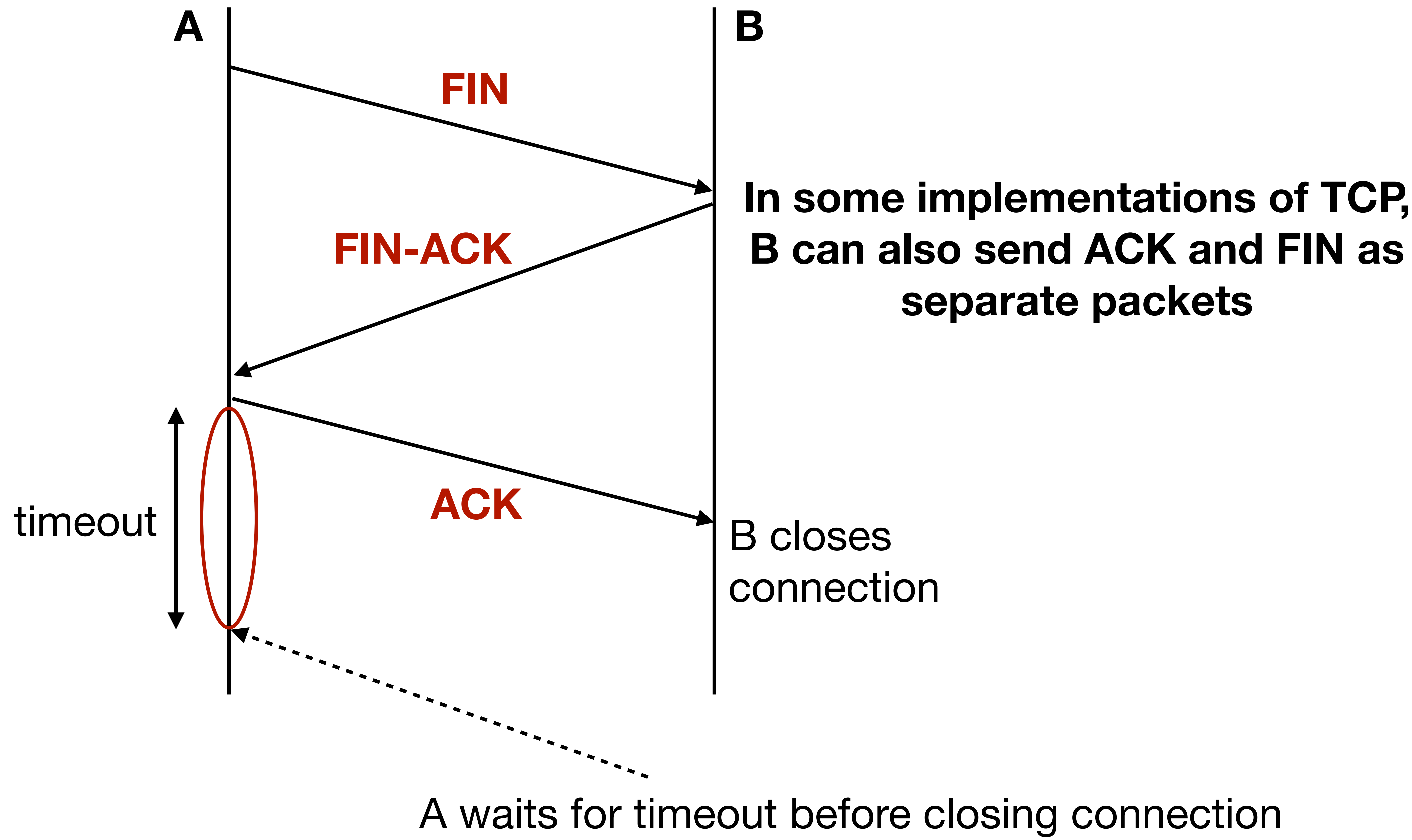
Step 3: ACK Packet from A

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS

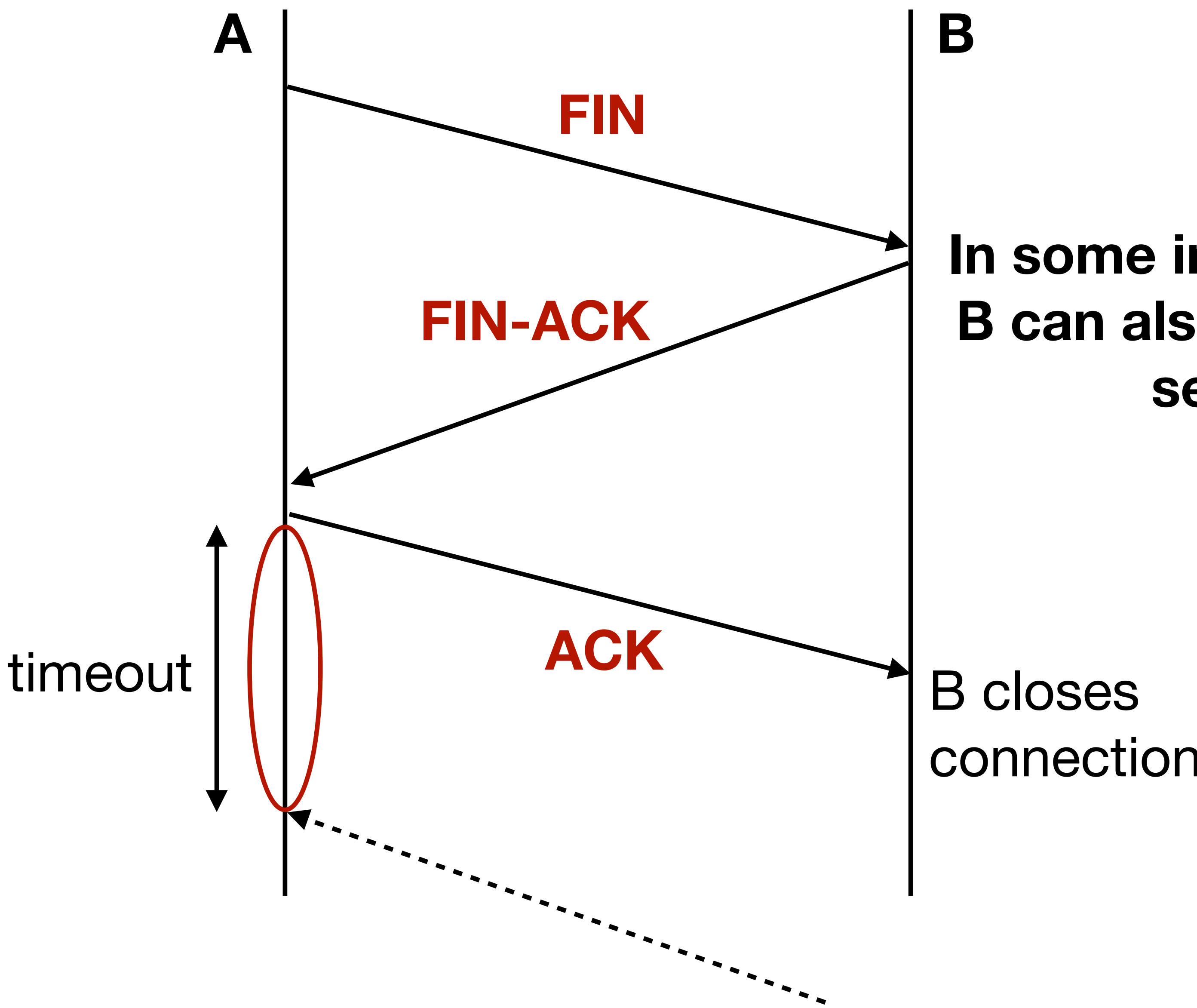


A tells B that it is also ready to receive next byte …
… on receiving this packet, B can start sending data

TCP (Normal) Connection Termination



TCP (Normal) Connection Termination

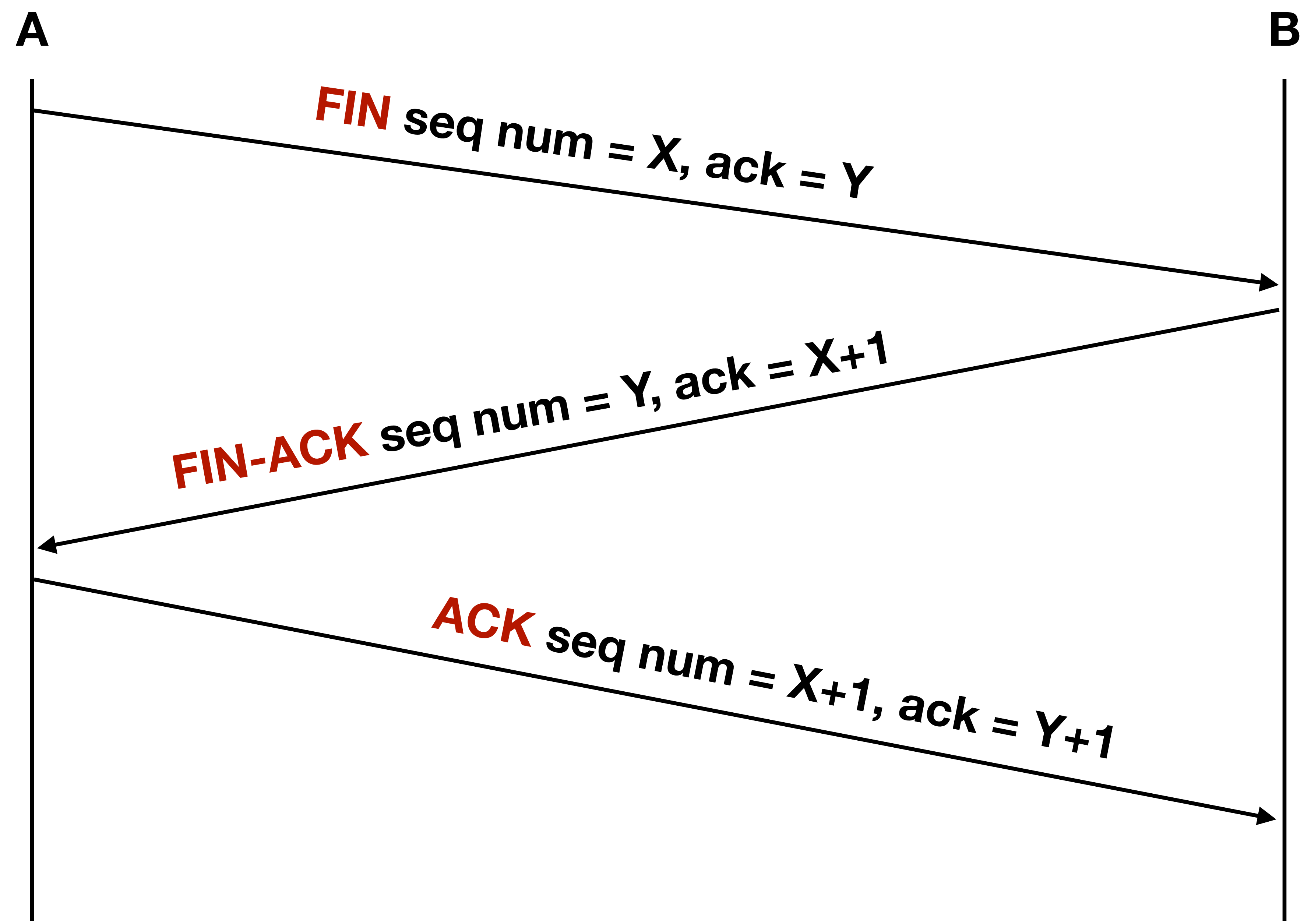


In some implementations of TCP, B can also send ACK and FIN as separate packets

A waits for timeout before closing connection

To make sure ACK is received by B, else A will receive a retransmitted FIN-ACK from B before timeout

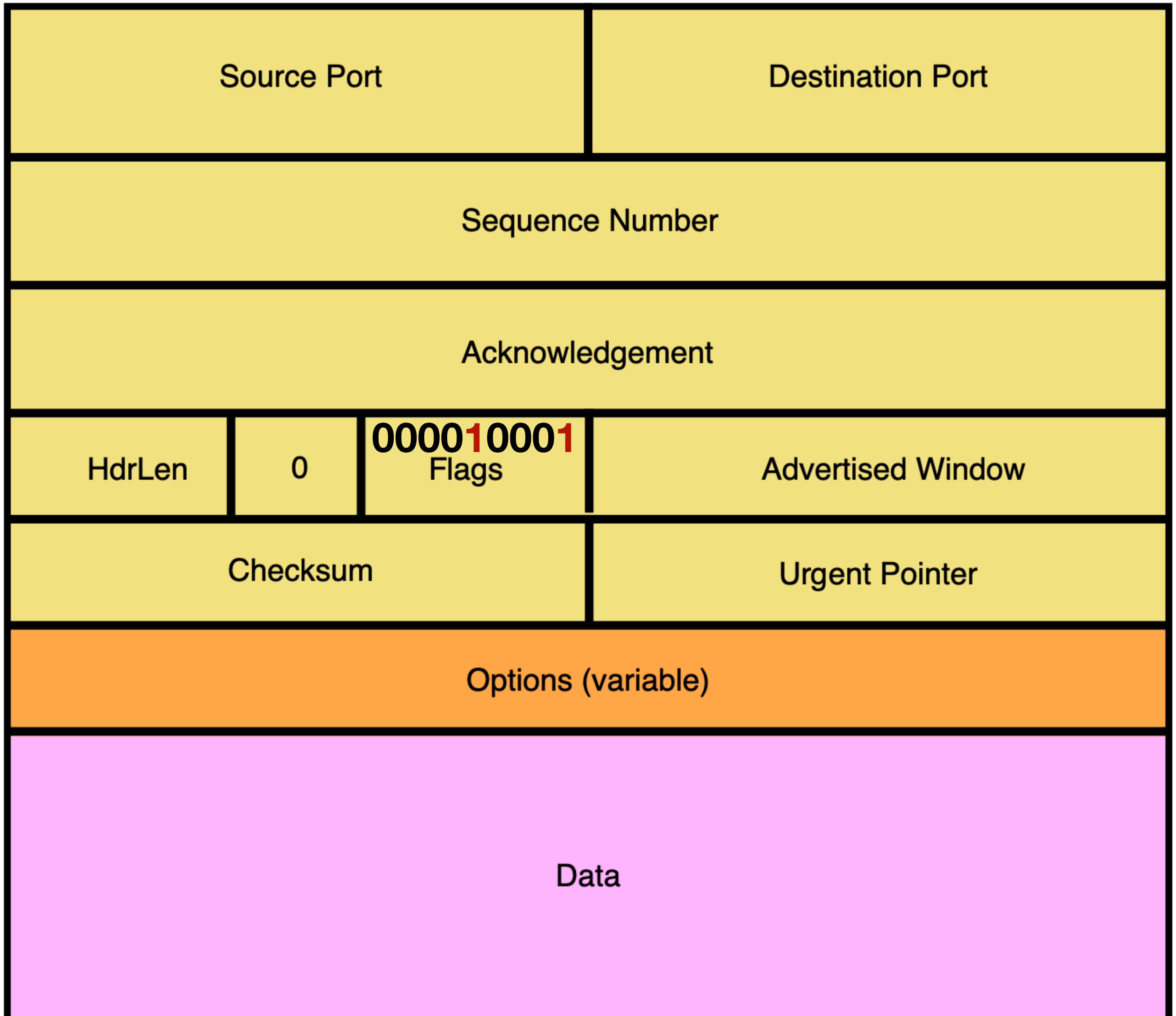
3-Way Termination



X = A's current sequence number
Y = sequence number of the next byte expected from B

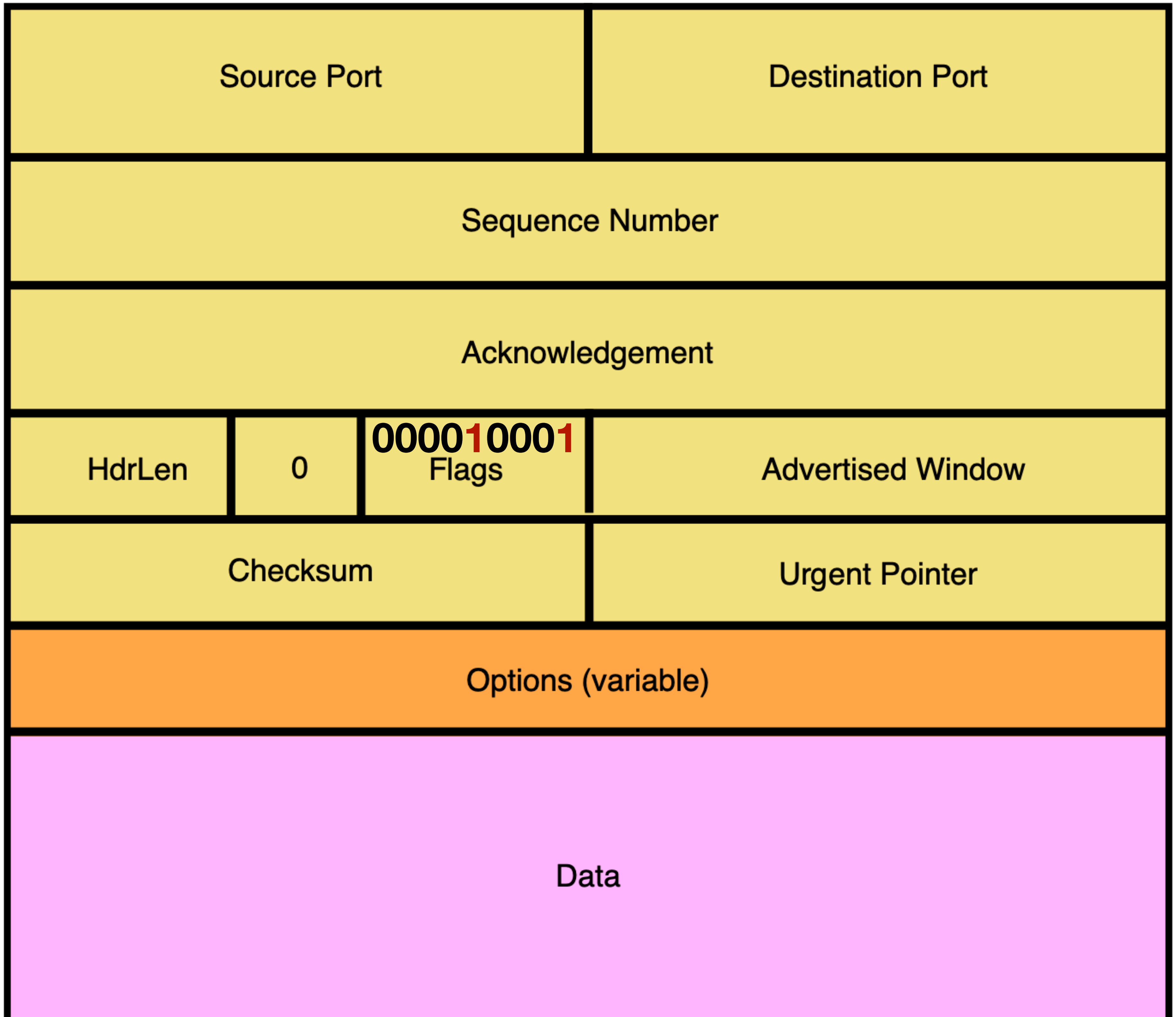
Step 1: FIN Packet from A

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS



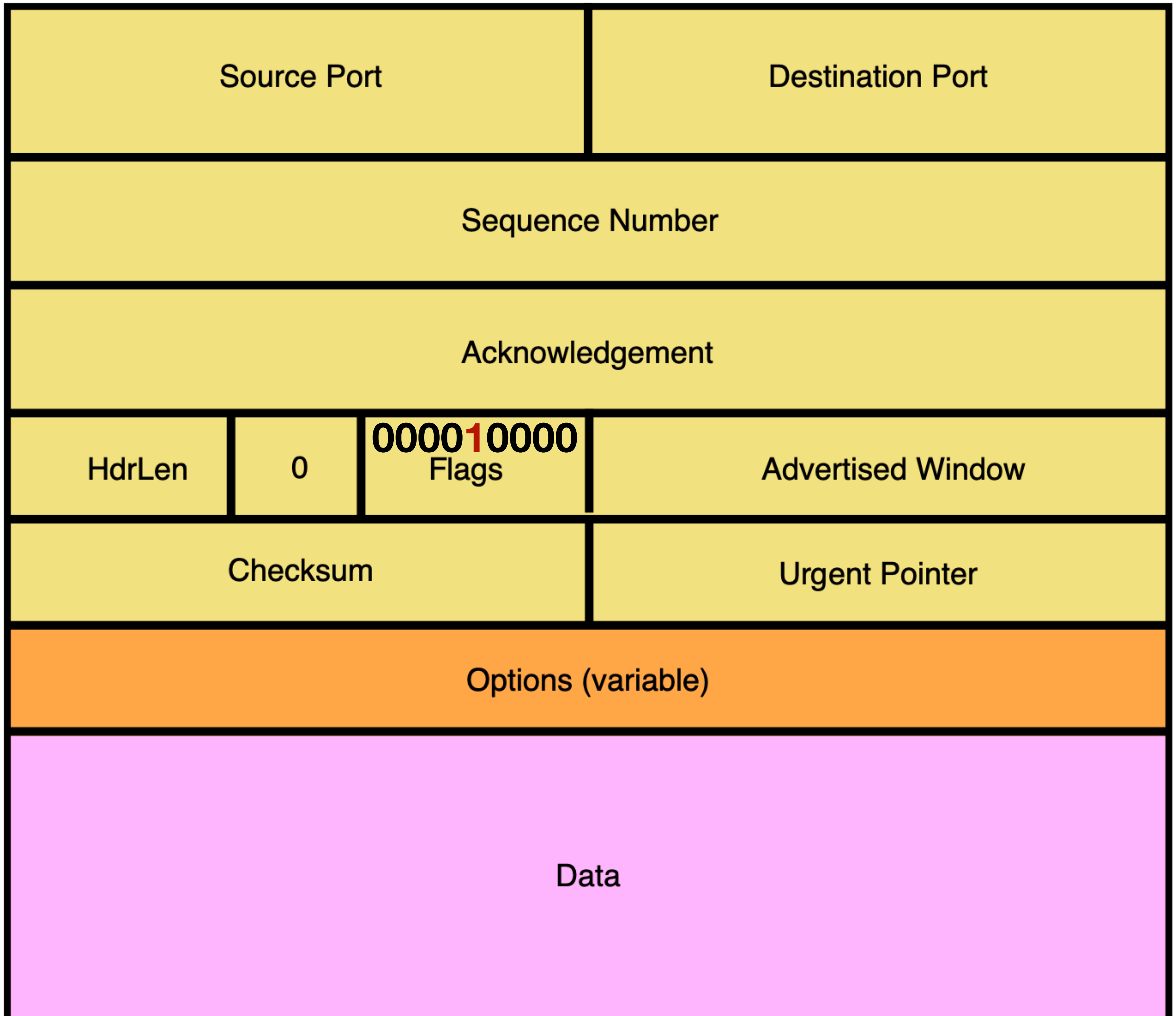
Step 2: FIN-ACK Packet from B

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS



Step 3: ACK Packet from A

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS



TCP (Abrupt) Connection Termination

- A TCP endpoint can close a connection any time by sending a **RESET (RST)** pkt

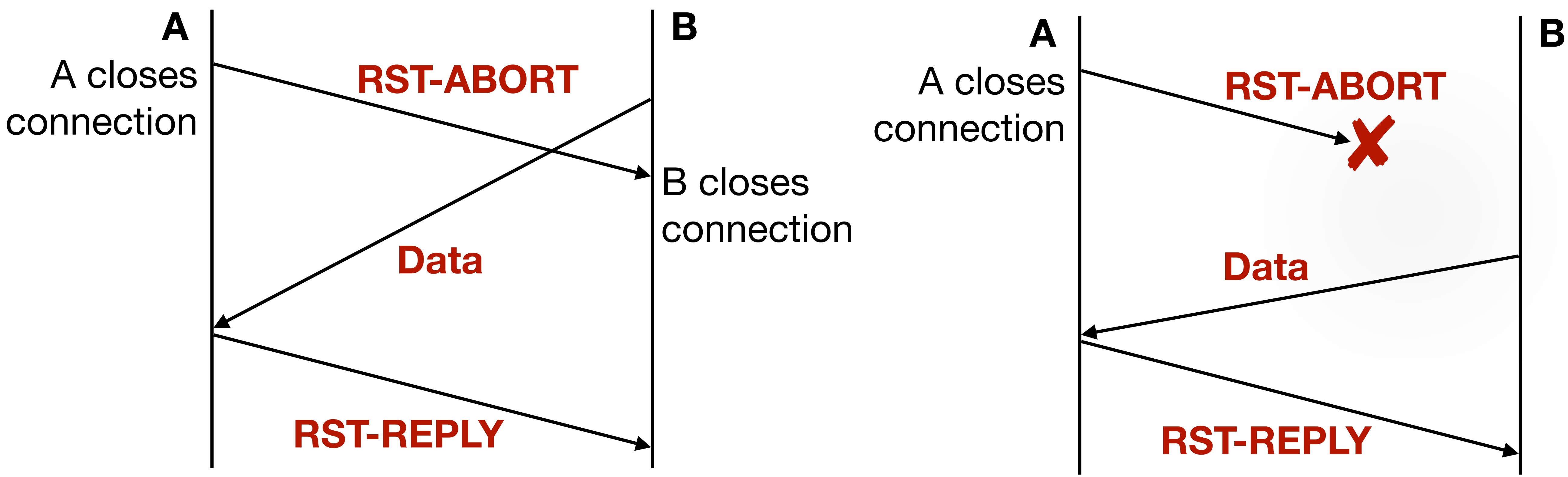
```
$ error: connection reset by peer
```

- **Two kinds of RST packets:**

  1. **RST-ABORT:** Generated when a connection is explicitly aborted by an endpoint (e.g., the socket at the endpoint is being killed)

  2. **RST-REPLY:** Generated on receipt of certain kinds of invalid packets (e.g., data packet for a connection that does not exist anymore)

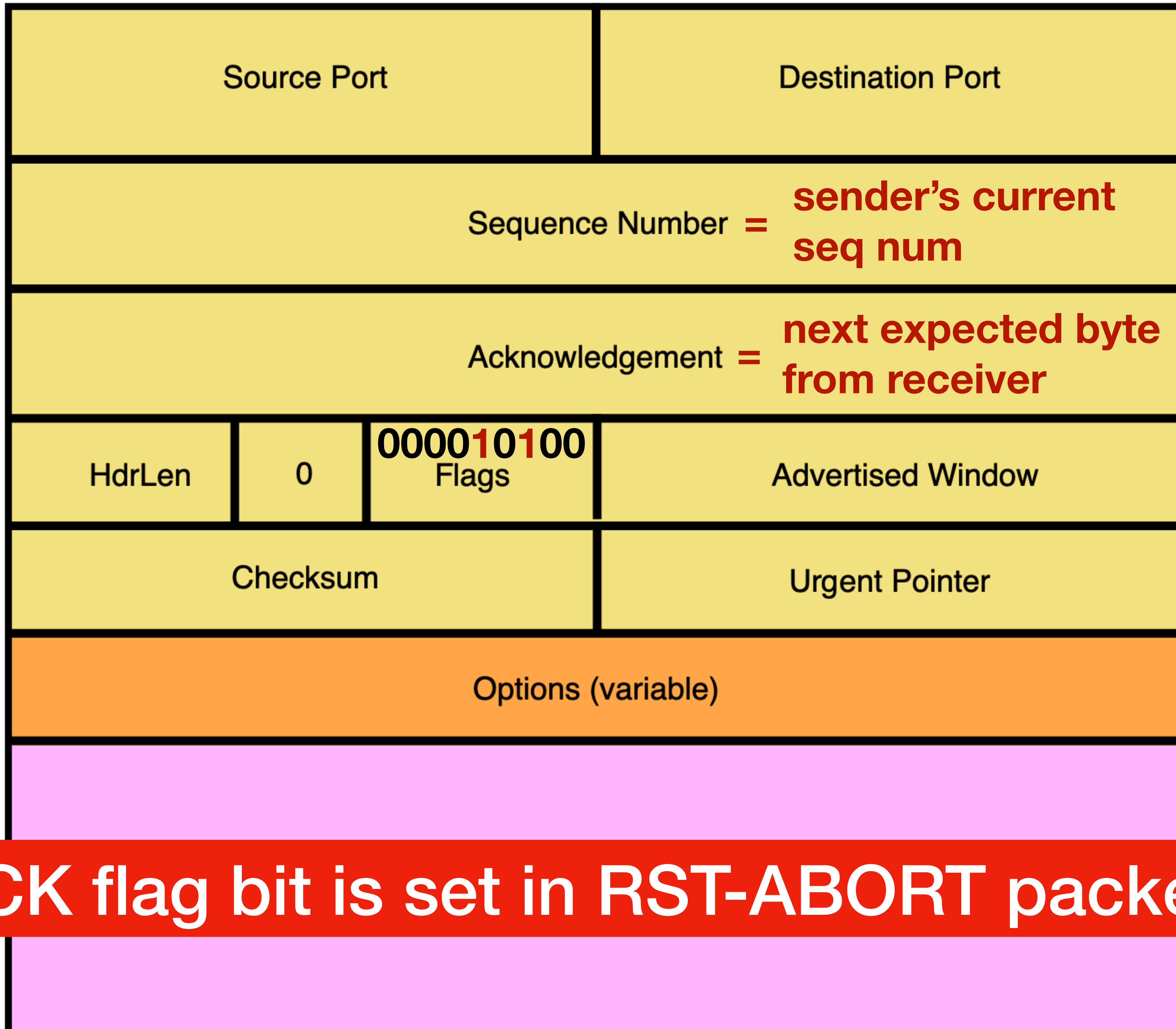
TCP (Abrupt) Connection Termination



- **A sends a RESET (RST-ABORT) packet to B to abruptly close connection**
  - That's it… B closes its connection on receipt and does *not* ACK the RST packet
    - Thus, RST packet is not delivered reliably!
- If an endpoint receives data for a closed conn, it will trigger **RST-REPLY** from A

RST-ABORT Packet

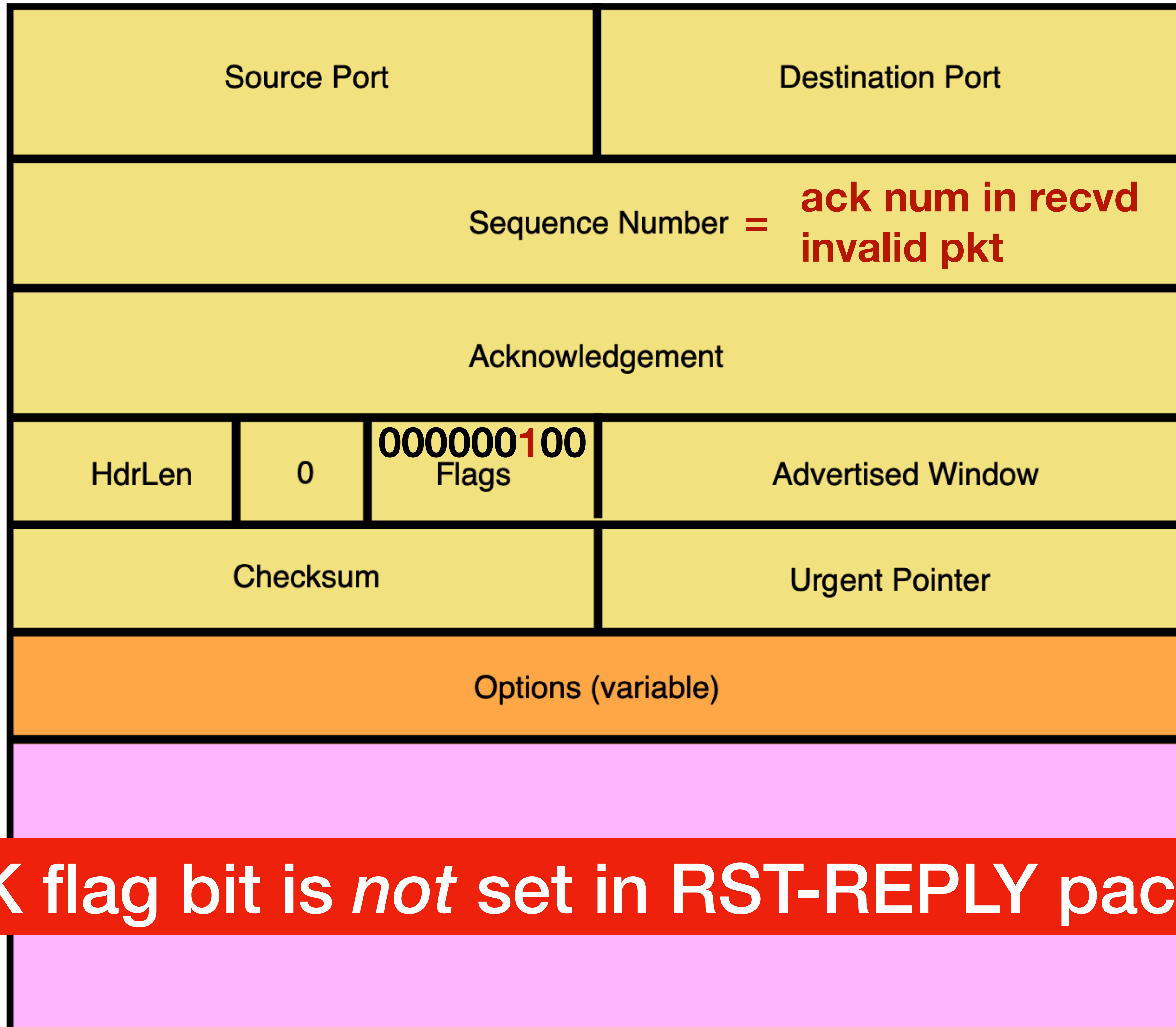
bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS



ACK flag bit is set in RST-ABORT packets

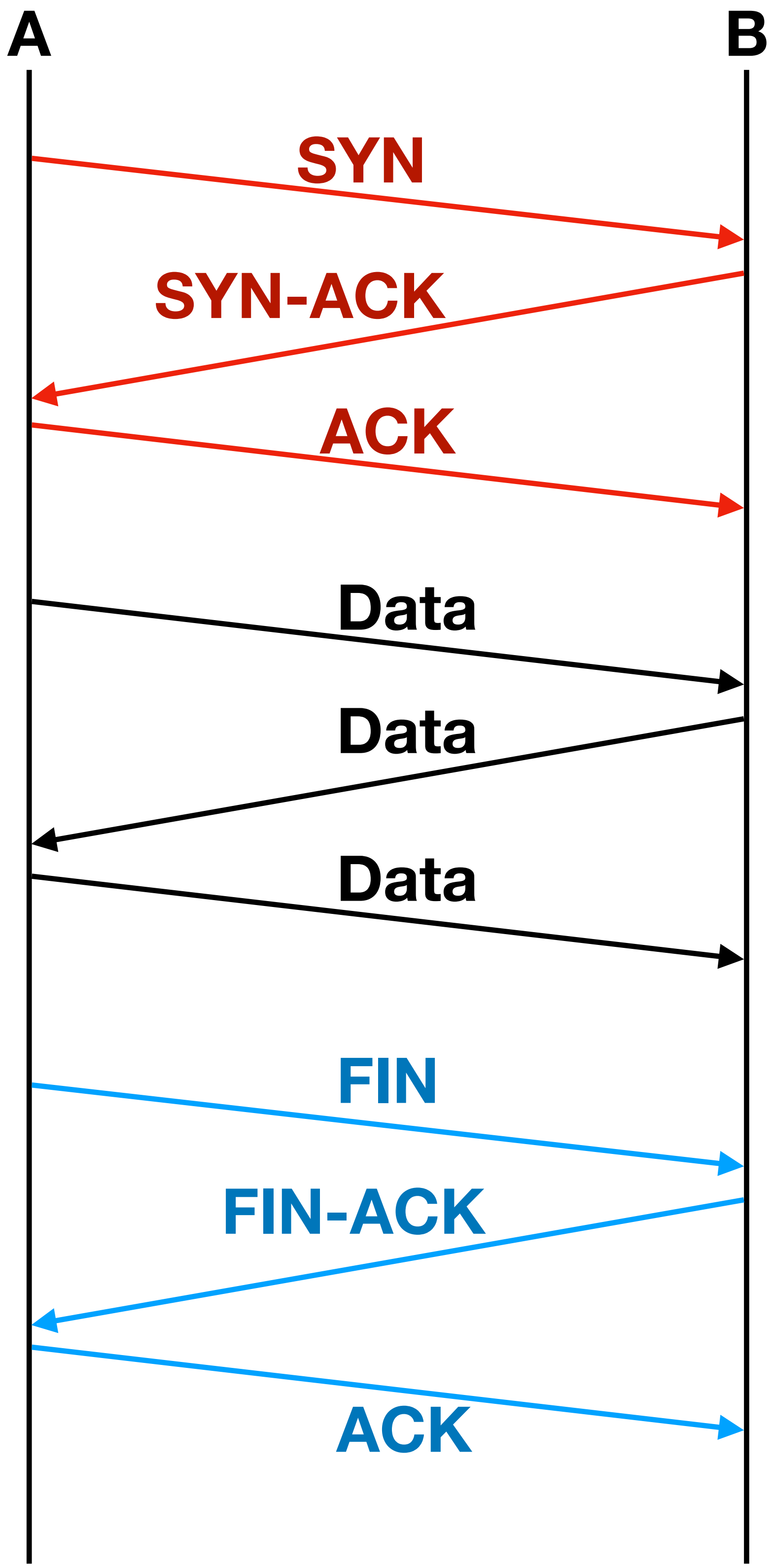
RST-REPLY Packet

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS



ACK flag bit is *not* set in RST-REPLY packets

TCP Full Timing Diagram



Connection set up

TCP is a "duplex" connection

— Data can be sent in both directions at the same time

— Data packets carry both **data** and **ACK**

— Carry own data, ACK other endpoint's data

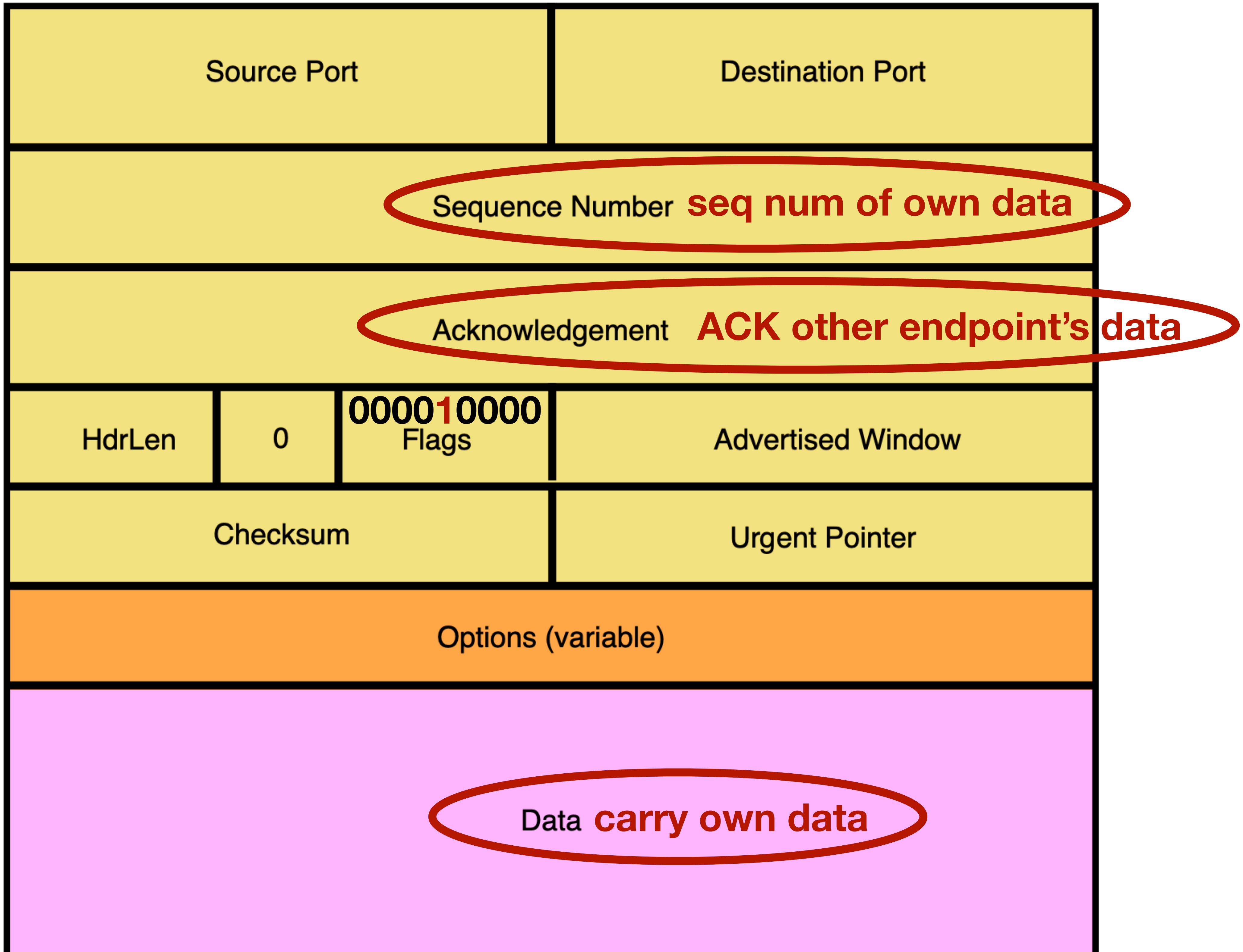
— If ACK flag bit is set, ACK number in header is valid

— All TCP data packets have ACK flag bit set

Connection termination

Data Packet

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS

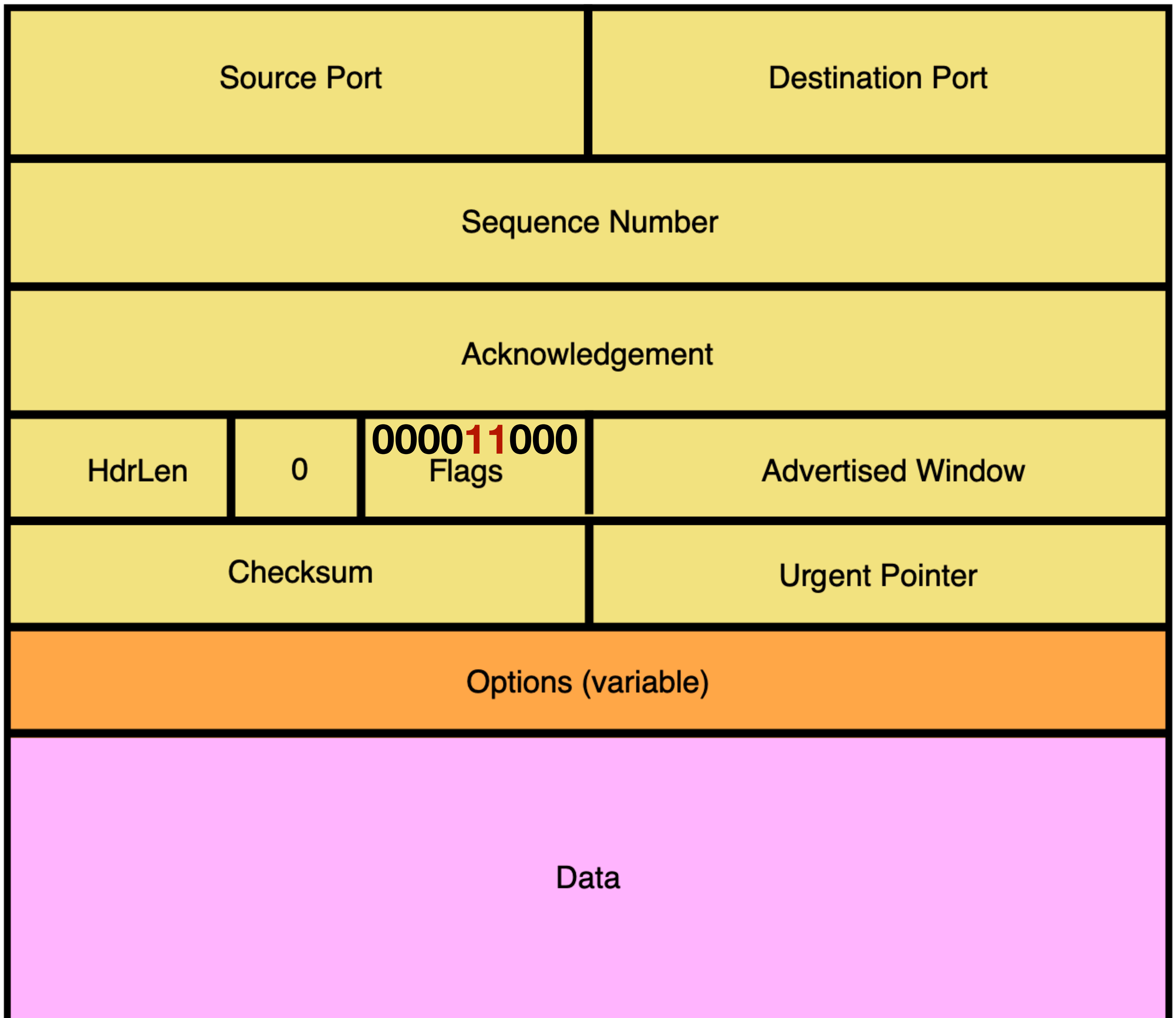


Remaining TCP Flags

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS

PSH Flag

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS

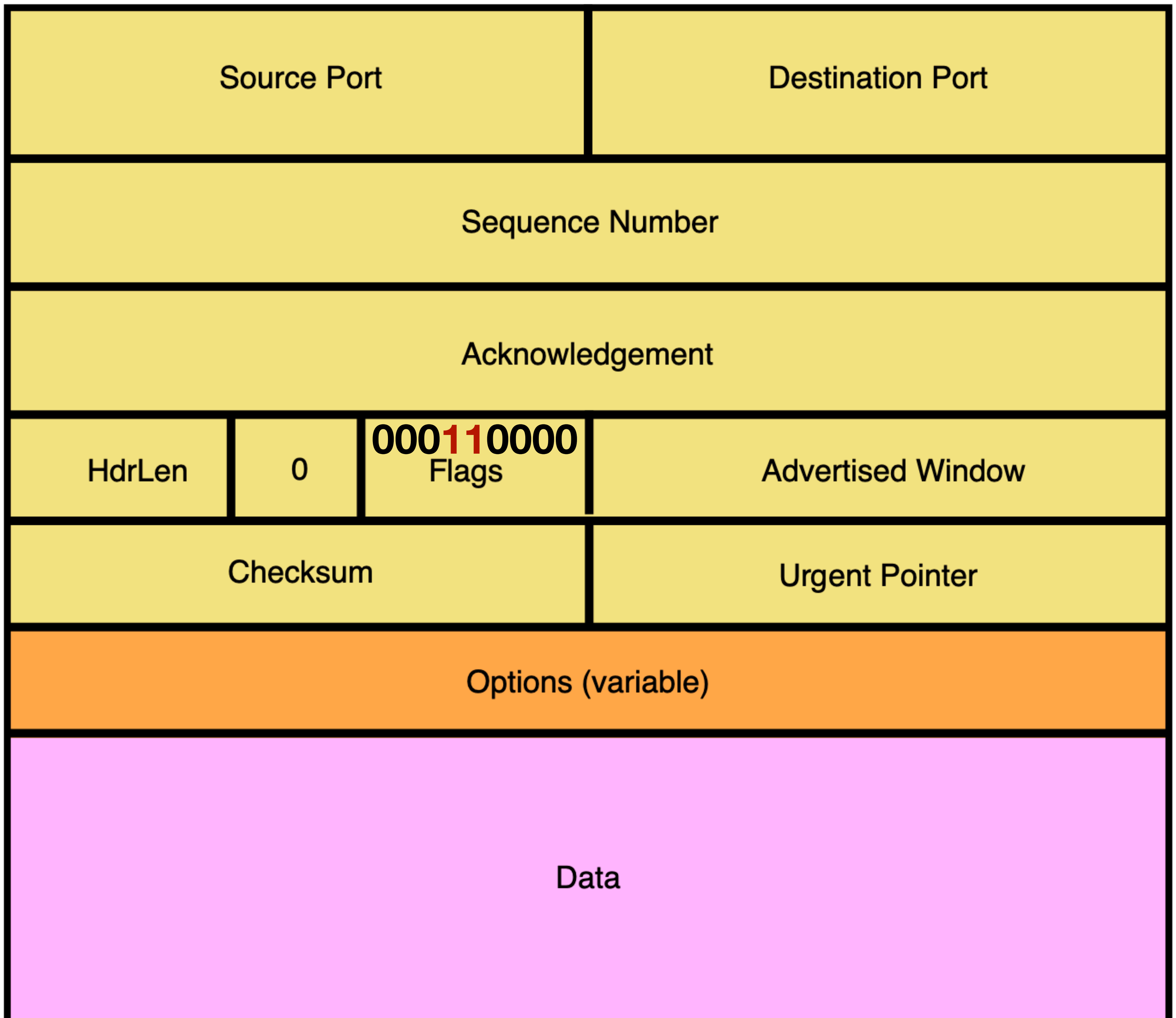


PSH Flag

- TCP by default can do data **“batching”** on both send and recv side
  - **Send: TCP can batch data in send buffer into a single packet**
    - Reduces header overhead and # of TCP packets sent to the network
  - **Recv: TCP can wait to batch data in recv buffer before sending to app**
    - Reduces number of receive calls from the application
- **But, batching also adds delay … PSH flag bit is used to disable batching**
- **Sender application can set PSH option in its socket interface**
  - TCP sends out whatever data is present in the send buffer immediately
  - Also sets the PSH flag bit in the outgoing packet
- **On the recv side, if TCP receives a packet with PSH flag bit set …**
  - … does not wait to batch that data before sending to the application
- **Desirable for interactive applications (e.g., chat applications)**

URG Flag

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS

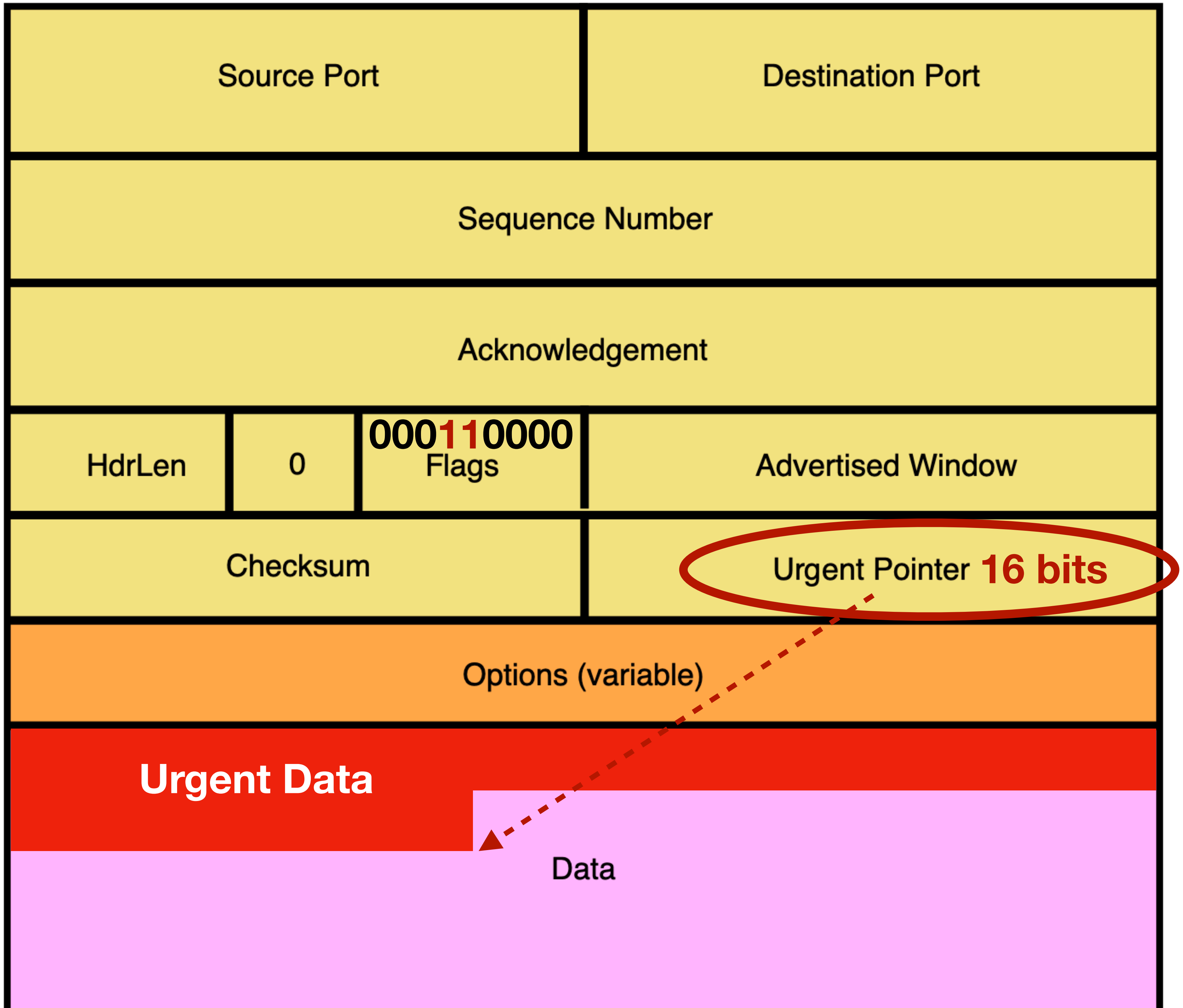


URG Flag

- **TCP by default delivers data to application layer “in-order”**
  - i.e., received data from network layer is added to the recv buffer …
  - … and dequeued and sent to application in a First-In-First-Out (FIFO) manner
- **URGENT (URG) flag bit is used to deliver data “out-of-order”**
  - If URG flag bit is set in a TCP packet,
    - Urgent data in TCP segment is delivered to application immediately
    - … bypassing the recv buffer
  - **Urgent Pointer** in TCP header stores offset to the **last** urgent byte in TCP segment, starting from the first byte in the segment

Urgent Pointer

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE
bit 7 - CWR
bit 8 - NS



ECE, CWR, and NS Flags

bit 0 - FIN
bit 1 - SYN
bit 2 - RST
bit 3 - PSH
bit 4 - ACK
bit 5 - URG
bit 6 - ECE (ECN-Echo)
bit 7 - CWR (Congestion Window Reduced)
bit 8 - NS (Nonce Sum) Experimental

Used by congestion control protocols that use Explicit Congestion Notification (ECN)

Transmission Control Protocol (TCP)

- **Reliable, In-order Delivery to Applications**
  - Application Multiplexing
  - ACKs and Retransmissions
  - Flow Control
  - Congestion Control
- **Byte Stream Abstraction**
  - Communicates with the applications using a byte stream and *not* packets
- **Connection-oriented**
  - Pairwise <sender, receiver> connection is established before sending data
  - Used to maintain connection-specific state, e.g., initial seq num, window scaling factor, window size, etc.

Questions?