# COMP90007 Internet Technology

Week9

Yiran (Scott) Ruan

Email: yrrua@unimelb.edu.au

Web Page: https://yiranruan.github.io

# Question 1

- In determining maximum packet lifetime, we have to be careful and pick a large enough period to ensure that not only the packet but also its acknowledgements have vanished. Discuss why this is needed.

# Solution 1

- Look at the second duplicate packet in Fig. 6-11(c). When the packet arrives, it would be a disaster if acknowledgements to *y* were still floating around.
- -> have a connection set up by accident

when no one wants it

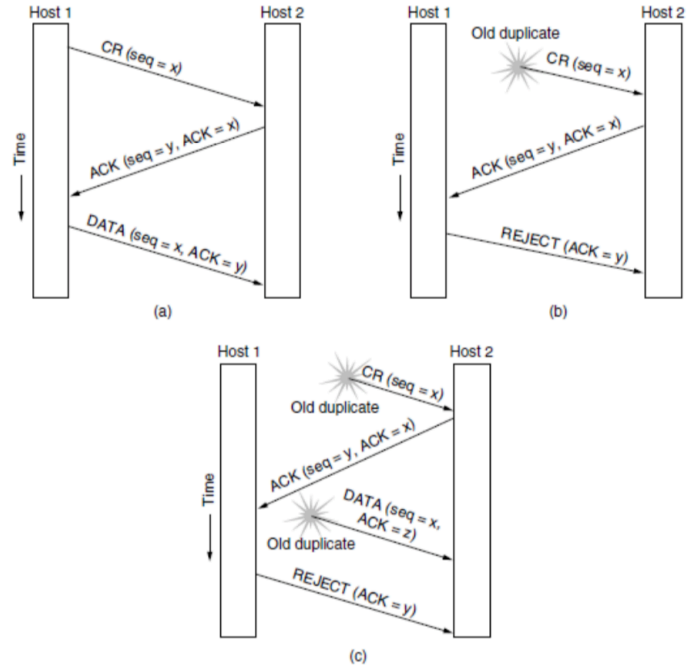



**Figure 6-11.** Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK.

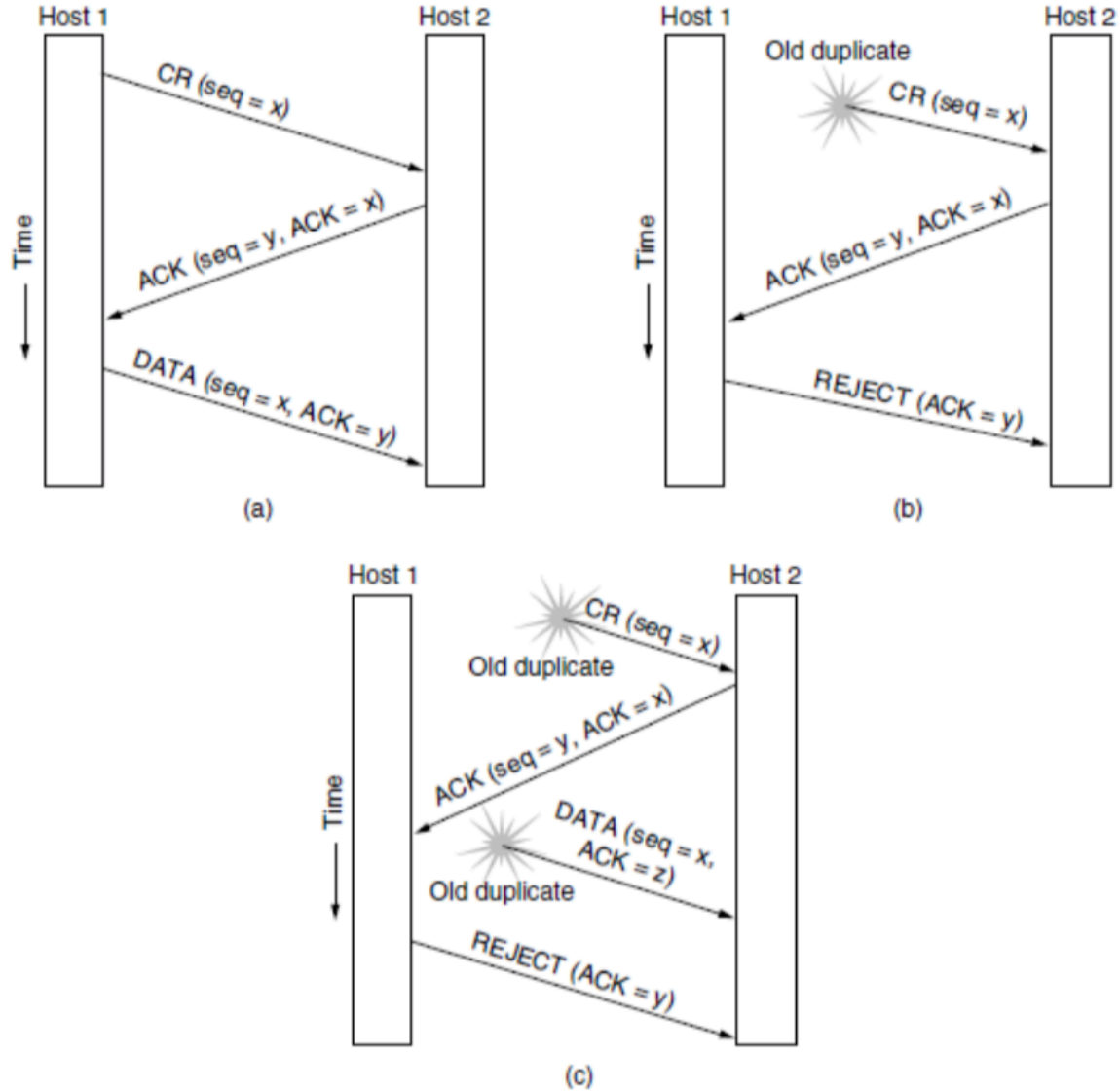



**Figure 6-11.** Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK.

# Question 2

- Imagine that a two-way handshake, rather than a three-way handshake were used to set up connections. In other words, the third message was not required. Are deadlocks now possible? Give an example or show that none exist.

# Solution 2

- Deadlocks are possible.
- For example, a packet arrives at A out of the blue, and A acknowledges it. The acknowledgement gets lost, but A is now open while B knows nothing at all about what has happened. Now the same thing happens to B, and both are open, but expecting different sequence numbers. Timeouts have to be introduced to avoid the deadlocks.

# Question 3

- Does the 3 way handshake based connection release protocol create a flawless disconnection?

# Question 4

- What is the 2 army problem? Where does it occur in networking? Provide an example.

# Two-Army Problem

- There is a famous problem that illustrates this issue. It is called the **two-army problem**. Imagine that a white army is encamped in a valley, as shown in Fig. 6-13. On both of the sur- rounding hillsides are blue armies. The white army is larger than either of the blue armies alone, but together the blue armies are larger than the white army. If either blue army attacks by itself, it will be defeated, but if the two blue armies attack simultaneously, they will be victorious.

# Two-Army Problem

**Does a protocol exist that allows the blue armies to win?**

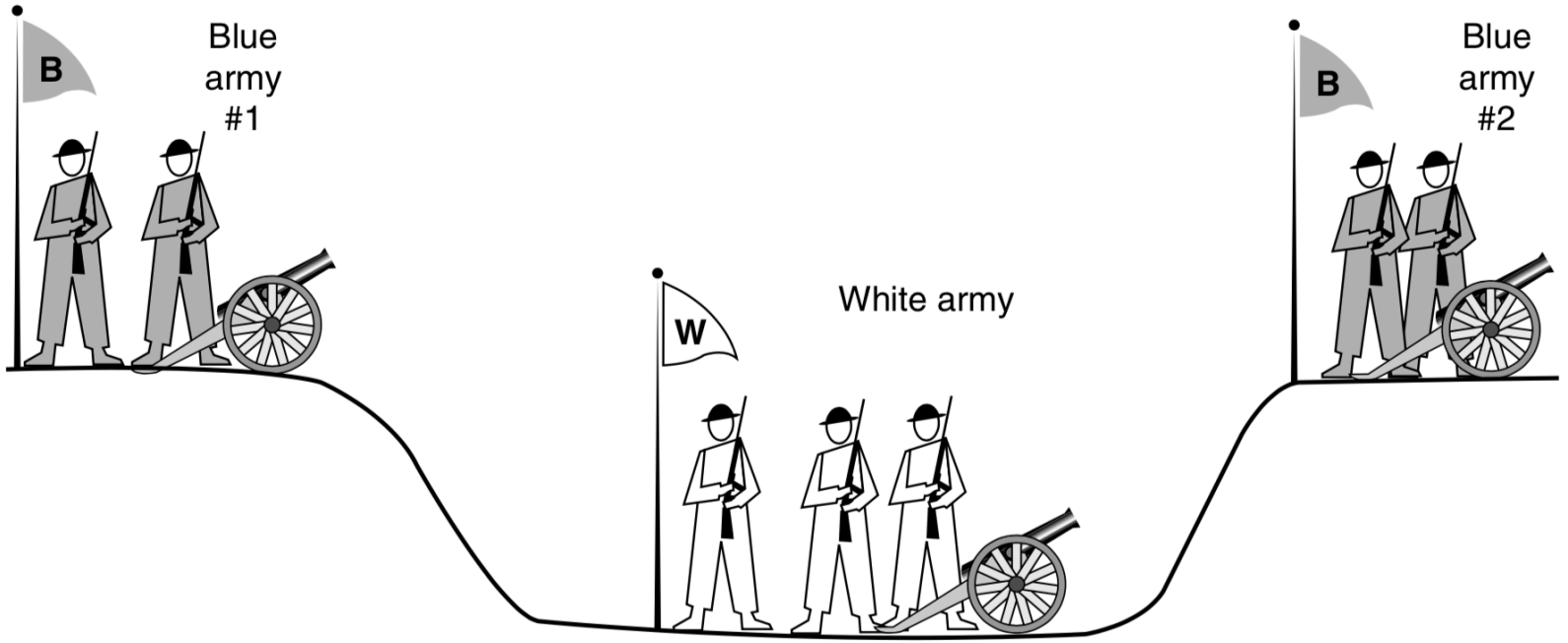



**Figure 6-13.** The two-army problem.

# Two-Army Problem

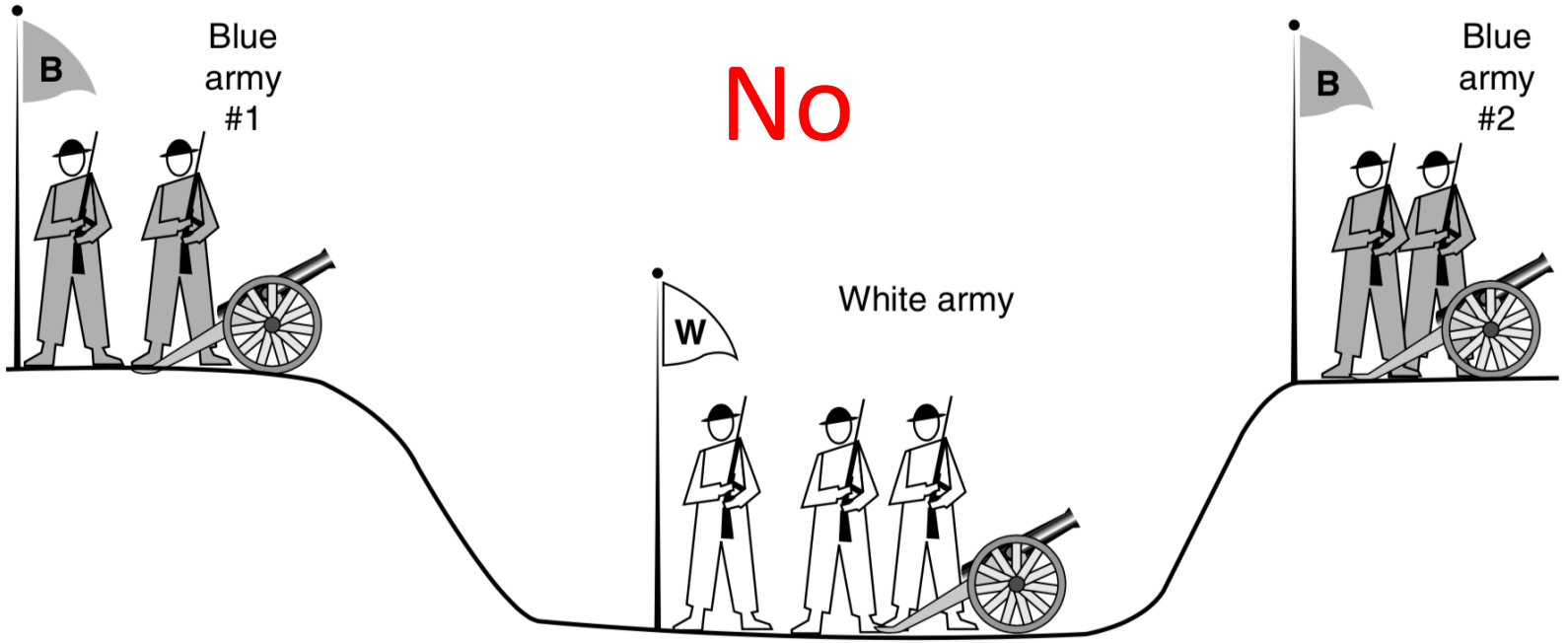



**Figure 6-13.** The two-army problem.

# Two-Army Problem

**Does a protocol exist that allows the blue armies to win?**

No

Either the last message of the protocol is essential, or it is not. If it is not, we can remove it (and any other unessential messages) until we are left with a protocol in which every message is essential. What happens if the final message does not get through? We just said that it was essential, so if it is lost, the attack does not take place. Since the sender of the final message can never be sure of its arrival, he will not risk attacking. Worse yet, the other blue army knows this, so it will **not attack** either.

**not attack -> Deadlock**

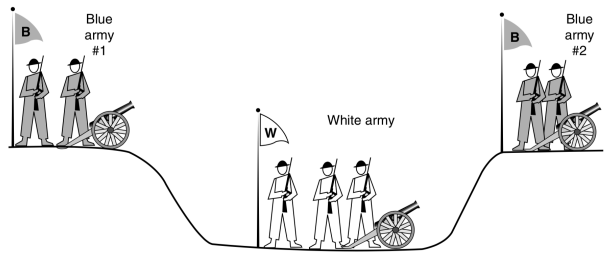



**Figure 6-13.** The two-army problem.

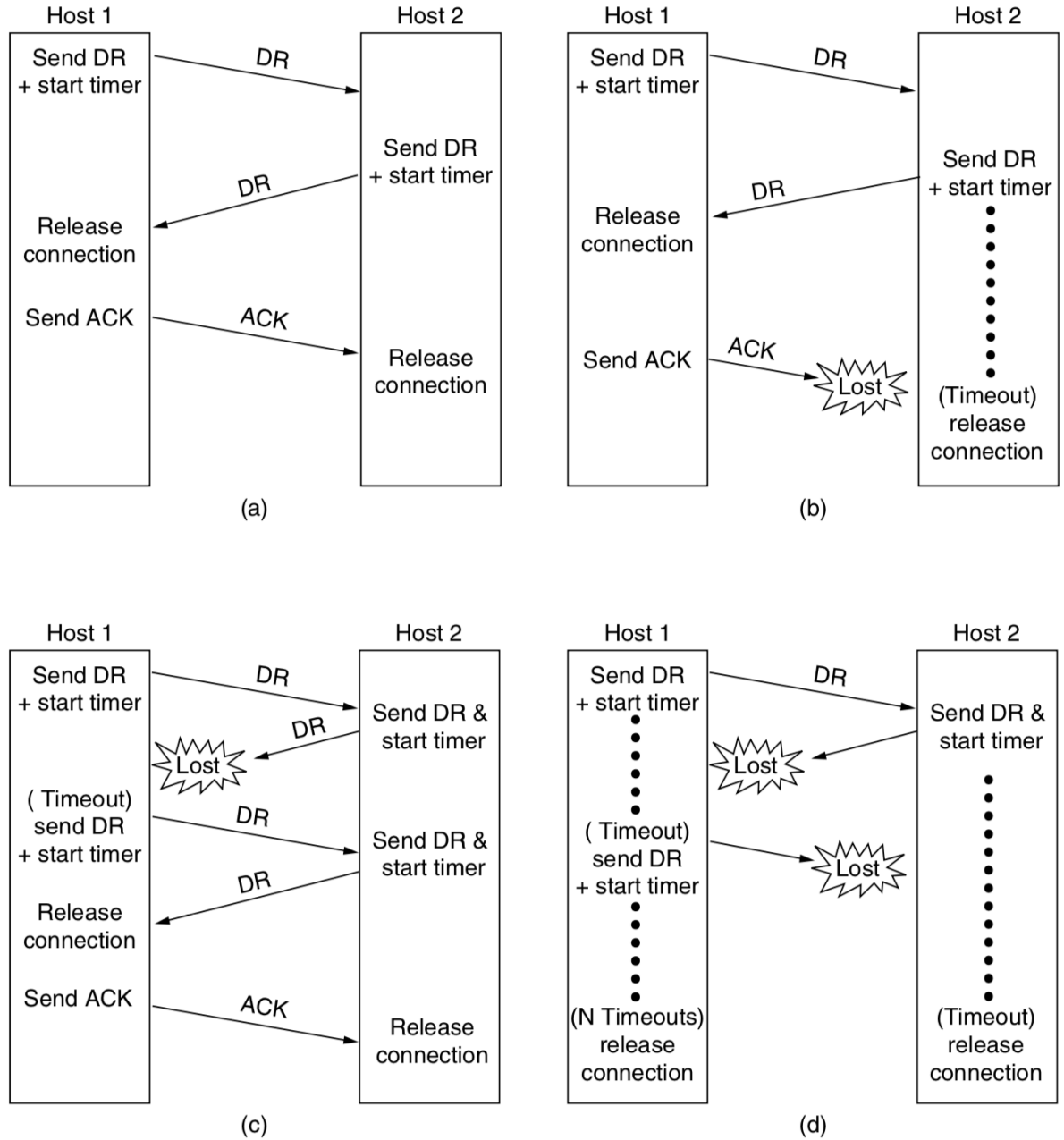



**Figure 6-14.** Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

# Example: Connection Release

- After *N* retries, the sender just gives up and releases the connection. Meanwhile, the receiver times out and also exits.
- While this protocol usually suffices, in theory it can fail if the initial DR and *N* retransmissions are all lost. The sender will give up and release the connection, while the other side knows nothing at all about the attempts to disconnect and is still fully active. This situation results in a half-open connection.
- We could have avoided this problem by not allowing the sender to give up after *N* retries and forcing it to go on forever until it gets a response. However, if the other side is allowed to time out, the sender will indeed go on forever, because no response will ever be forthcoming. If we do not allow the receiving side to time out, the protocol hangs in Fig. 6-14(d).

# Example: Connection Release

- One way to kill off half-open connections is to have a rule saying that if no segments have arrived for a certain number of seconds, the connection is automatically disconnected. That way, if one side ever disconnects, the other side will detect the lack of activity and also disconnect.

# Solution 4

- Ans. Refer to Page 519 for the explanation of two armies, one of which is split up with an enemy in the middle, and how they communicate with each other to try and coordinate for launching an attack.
- Example – Connection Release.

# Question 3

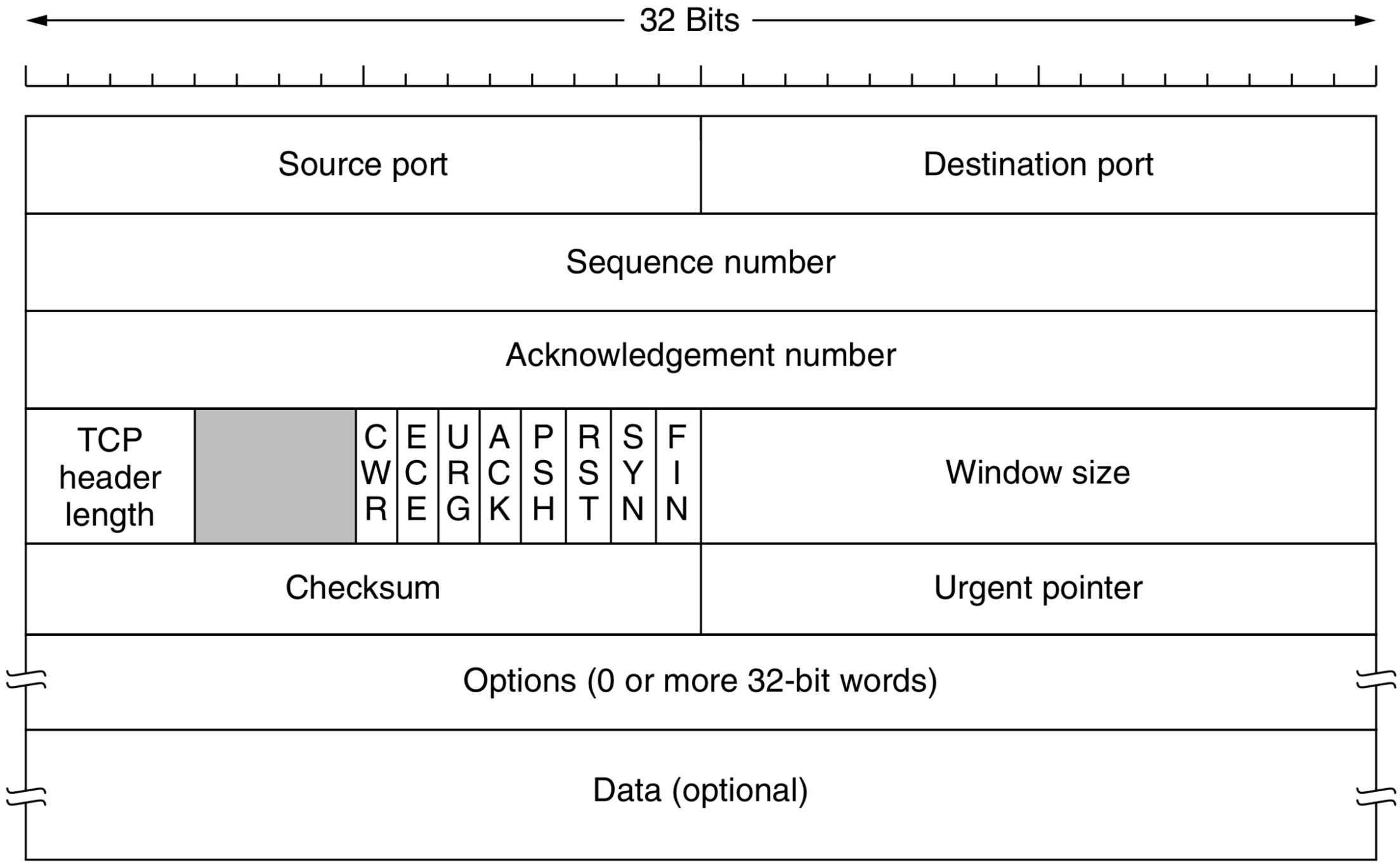
- Does the 3 way handshake based connection release protocol create a flawless disconnection?

# Solution 3

- No. If it was the case then the two-army problem would have been solved. (please refer to page 519 in textbook). The three-way handshake-based solution is an approximation.
- E.G.: Imagine the timeout for case (b) on page 521, if the timeout triggers while there is data lingering in the network then the data will be lost as connection will be terminated early.

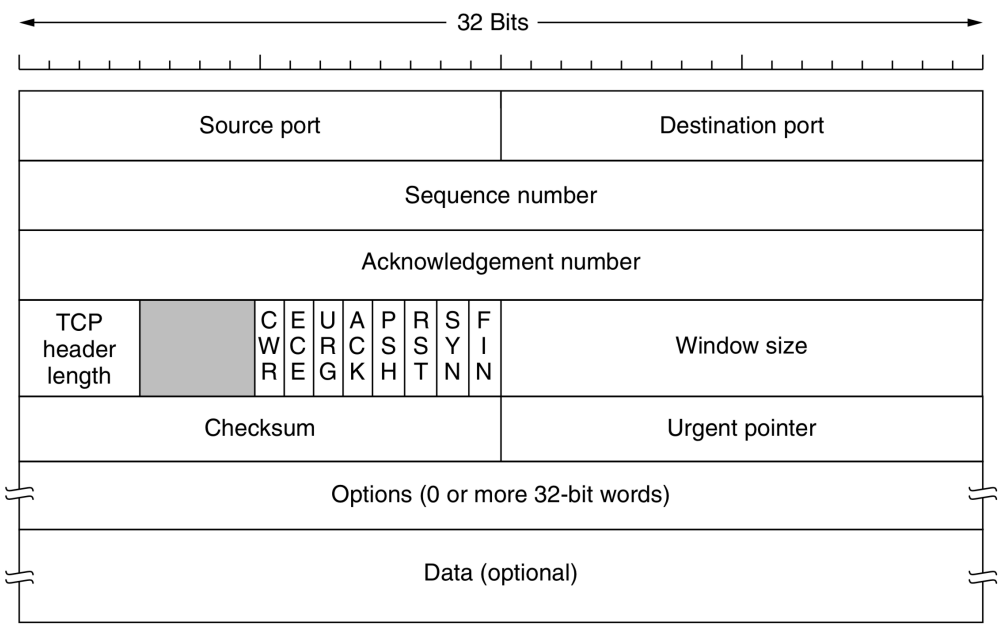
# Question 5

- What information is sent with the TCP Segment header, explain each field briefly?


32 Bits
Source port
Destination port
Sequence number
Acknowledgement number
TCP header length
CWR
ECE
URG
ACK
PSH
RST
SYN
FIN
Window size
Checksum
Urgent pointer
Options (0 or more 32-bit words)
Data (optional)

# Ports field

- Two port: Source port and Destination port
- TCP port + IP address -> unique end point (48 bits)
- The source and destination end points together identify the connection
- Connection identifier (5 tuple):
  - The protocol (TCP)
  - Source IP
  - Source port
  - Destination IP
  - Destination port

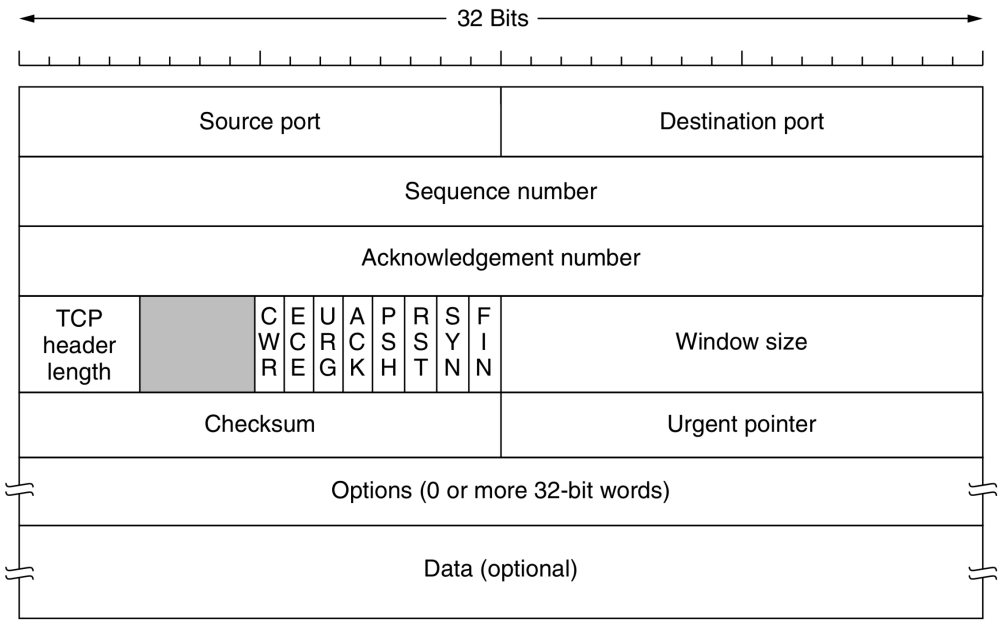


# Seq and Ack

- Both are 32 bits because every byte of data is numbered in a TCP stream.
- Note that the ACK specifies the next in-order byte expected, not the last byte correctly received.
- **cumulative acknowledgement:**

  it summarizes the received data

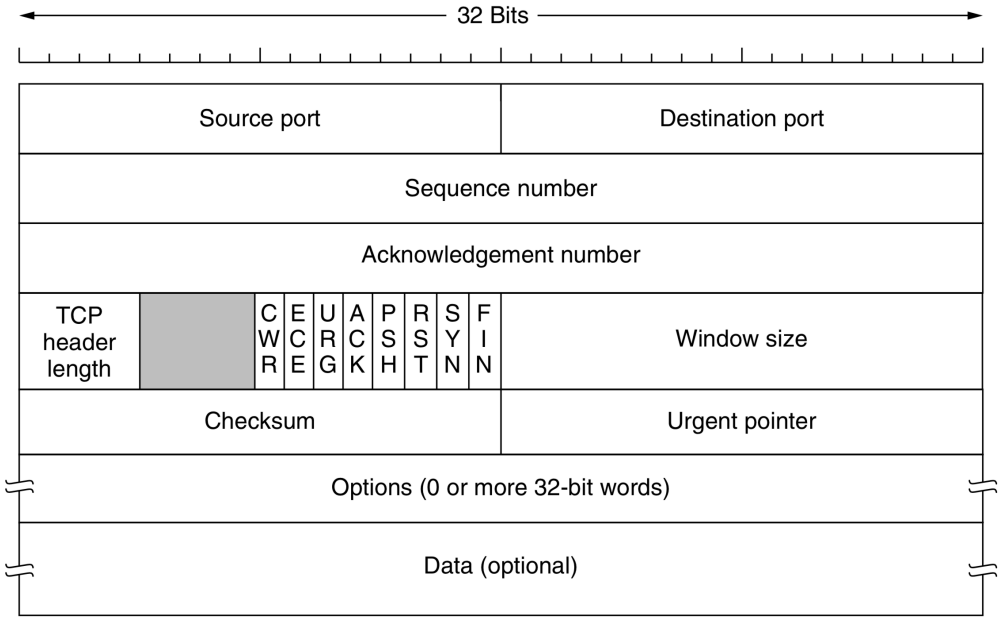
  with a single number



# TCP header length

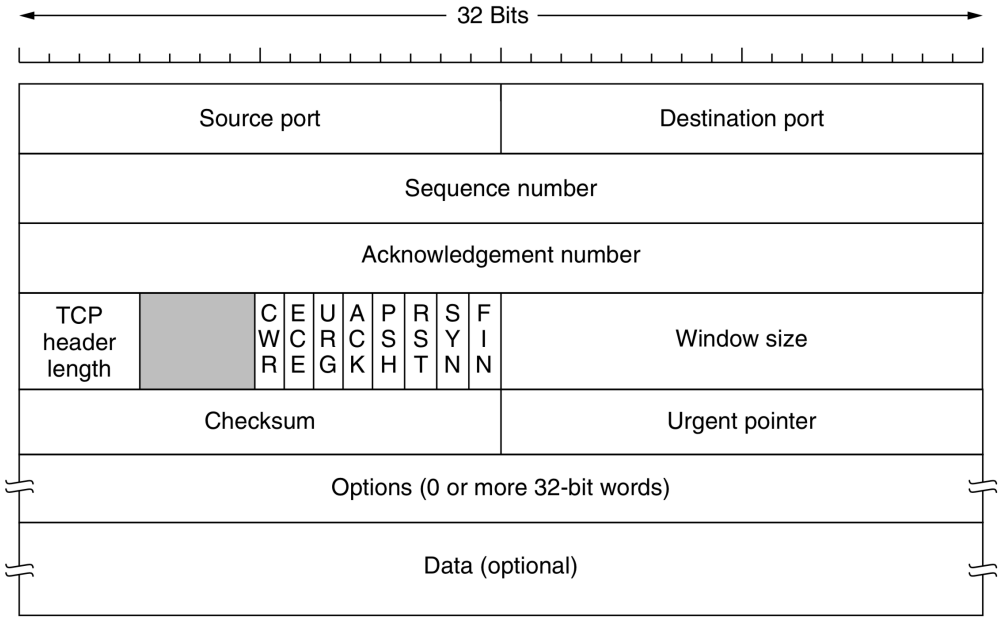
- how many 32-bit words are contained in the TCP header.
- Reason: *Options* field is of variable length

- Next comes a 4-bit field that is not used



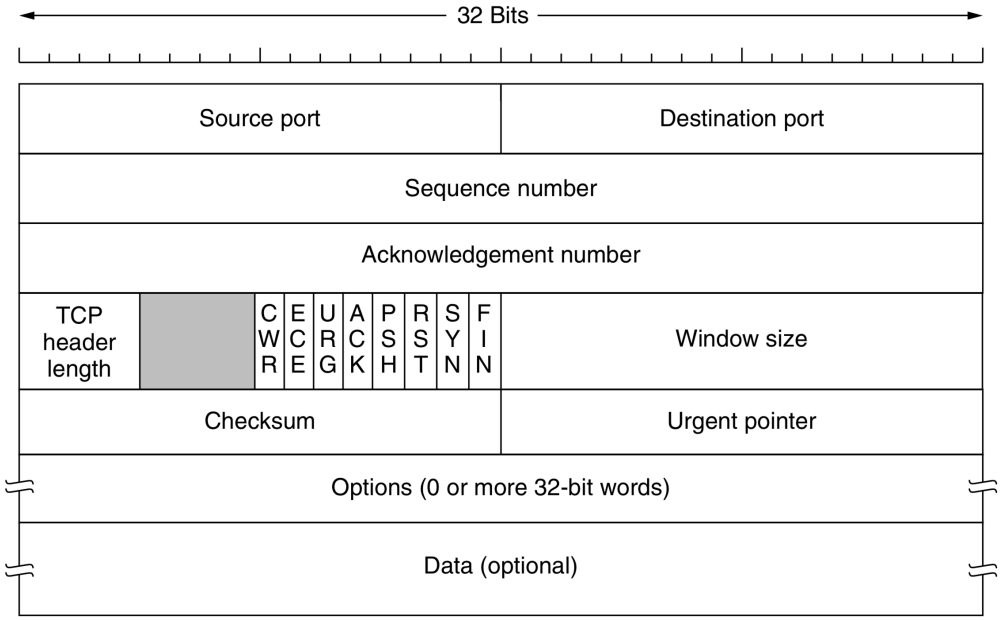
# Eight Flags

- Eight 1-bit Flags: CWR, ECE, URG, ACK, PSH, RST, SYN, FIN
- CWR, ECE:
  - used to signal congestion when ECN (Explicit Congestion Notification) is used
  - *ECE -> ECN-Echo (receiver -> sender)*
  - tell a TCP sender to slow down when the TCP receiver gets a congestion indication from the network.
  - CWR -> *Congestion Window Reduced (sender -> receiver)*
  - sender has slowed down and can stop sending the *ECN-Echo*.



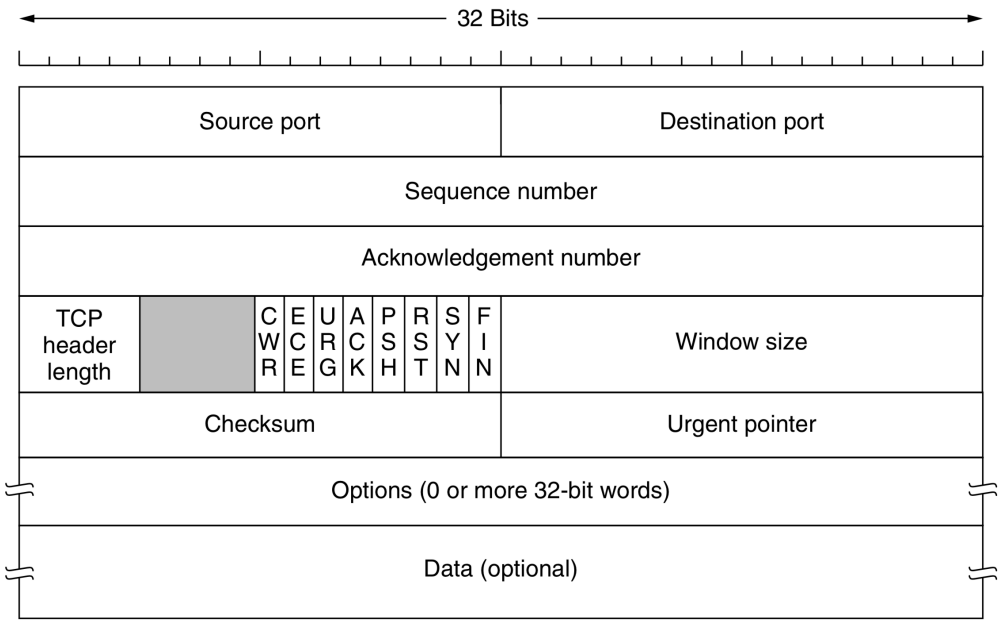
# Eight Flags

- URG
- -> 1 : Urgent pointer (interrupt messages)
- The *Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found.



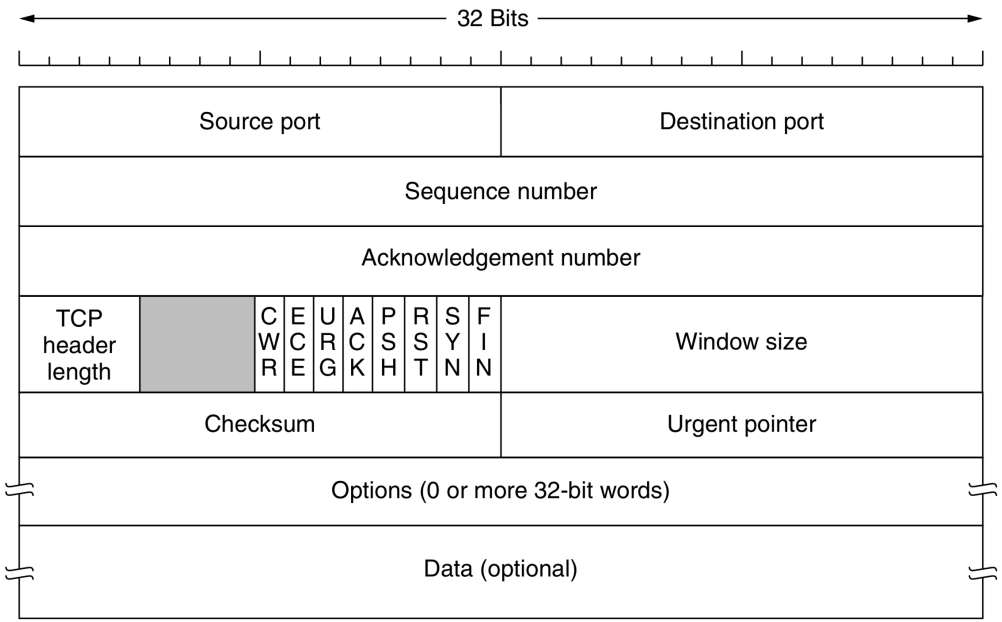
# Eight Flags

- ACK -> 1 : the *Acknowledgement number* is valid. This is the case for nearly all packets.
- *ACK* -> 0 : the segment does not contain an acknowledgement, so the *Acknowledgement number* field is ignored.



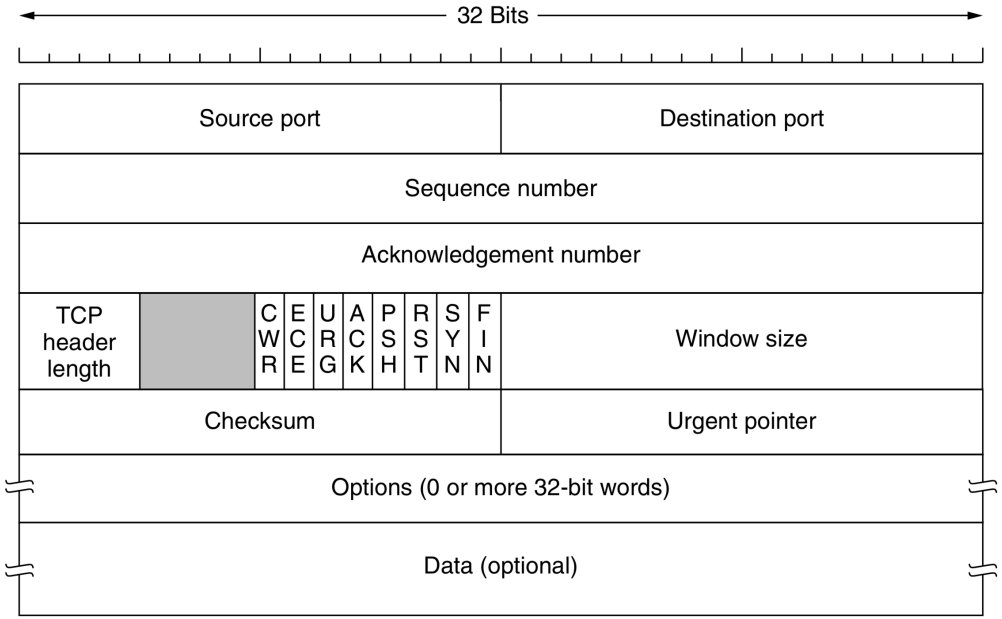
# Eight Flags

- PSH -> PUSHed data
- The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).



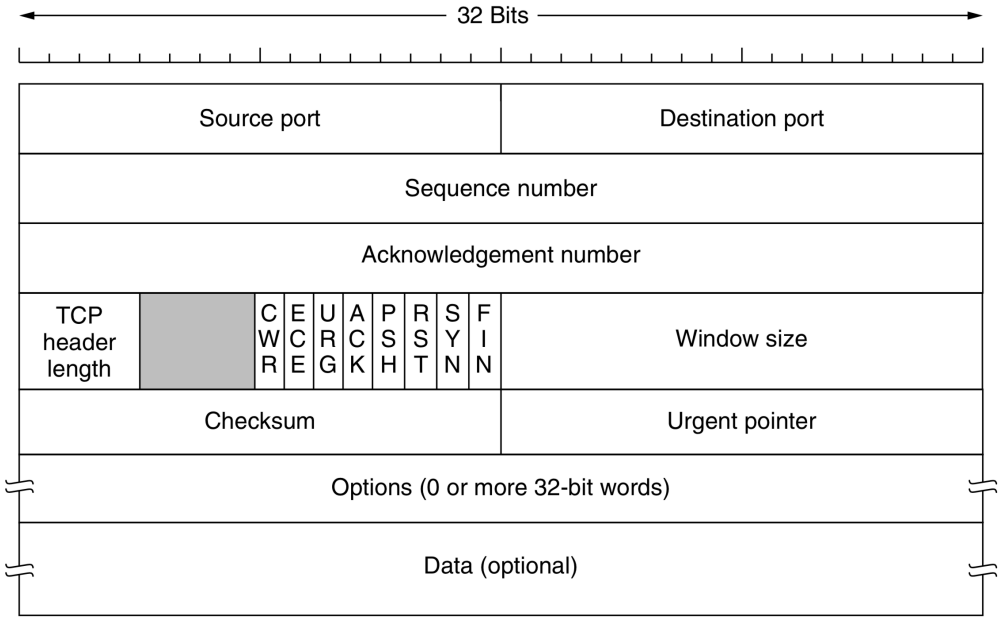
# Eight Flags

- RST
- 1. reset a connection that has become confused due to a host crash or some other reason
- 2. reject an invalid segment or refuse an attempt to open a connection.



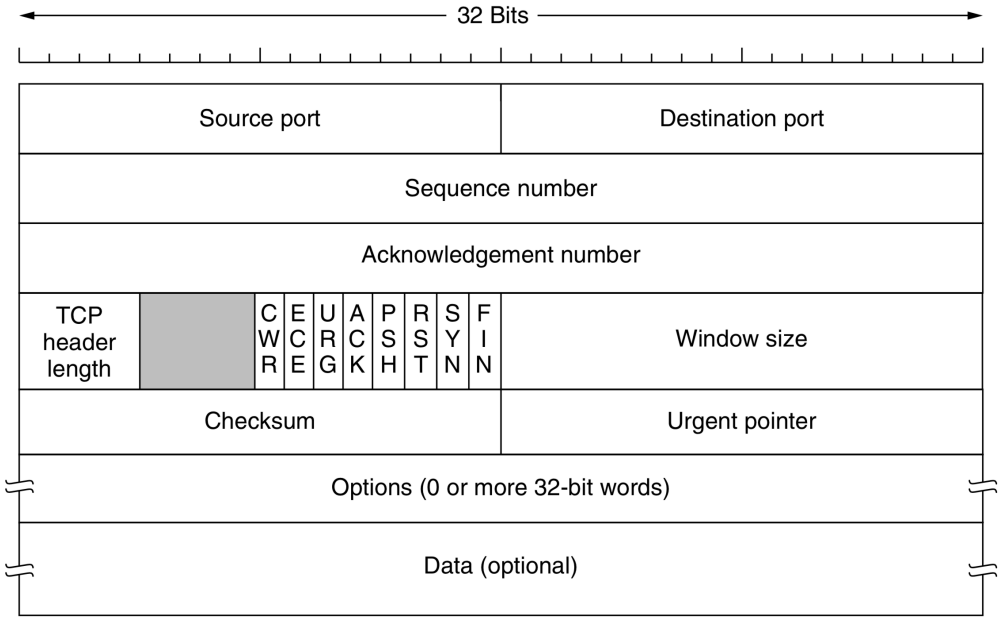
# Eight Flags

- SYN -> establish connections
- *SYN* = 1 and *ACK* = 0 -> the piggyback acknowledgement field is not in use.
- *SYN* = 1 and *ACK* = 1 -> the connection reply does bear an acknowledgement
- *SYN*
  - CONNECTION REQUEST
  - CONNECTION ACCEPTED
  - Distinguished by ACK.



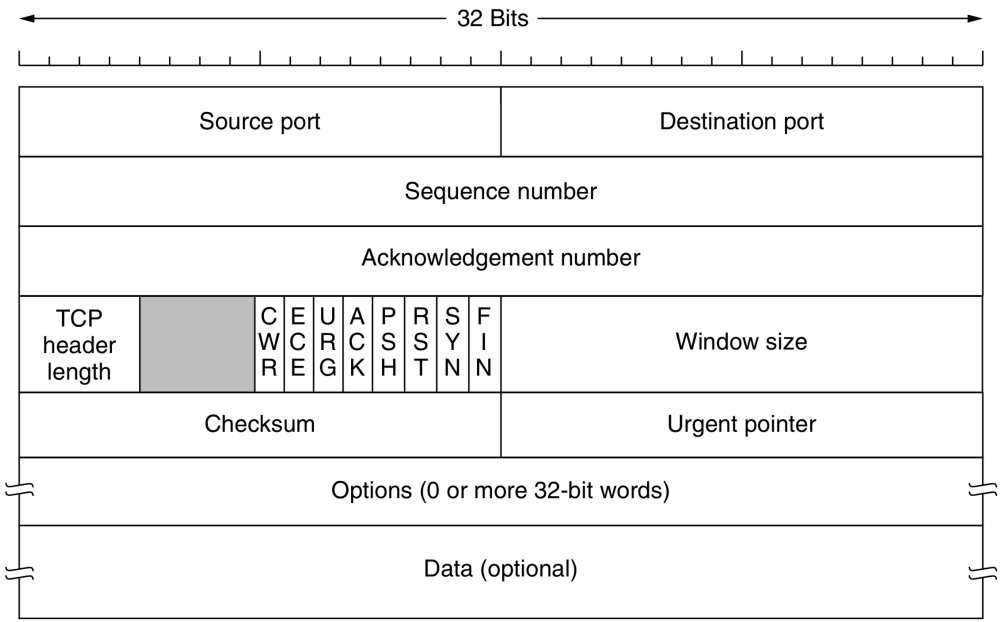
# Eight Flags

- FIN -> release a connection
- sender has no more data to *transmit*.
- After closing a connection, the closing process may continue to *receive* data indefinitely.
- Both *SYN* and *FIN* segments have sequence numbers and are thus guaranteed to be processed in the correct order.



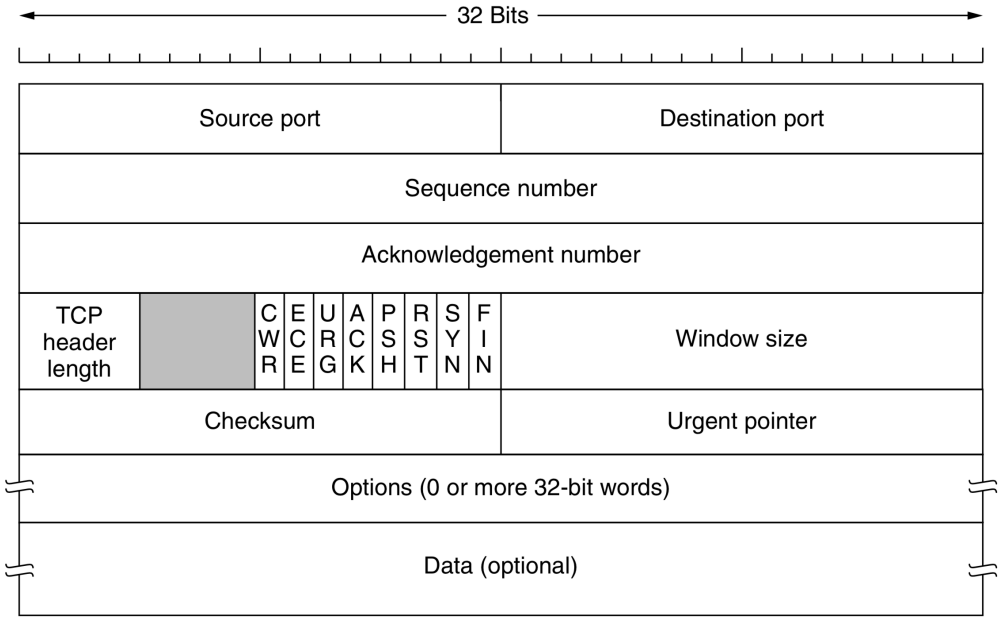
# Window size

- **Flow control** in TCP is handled using a variable-sized sliding window.
- tells how many bytes may be sent starting at the byte acknowledged.
- A *Window size* field of 0 is legal and says that the bytes up to and including *Acknowledgement number* − 1 have been received. But no more data could be received by receiver.



# Checksum and Options

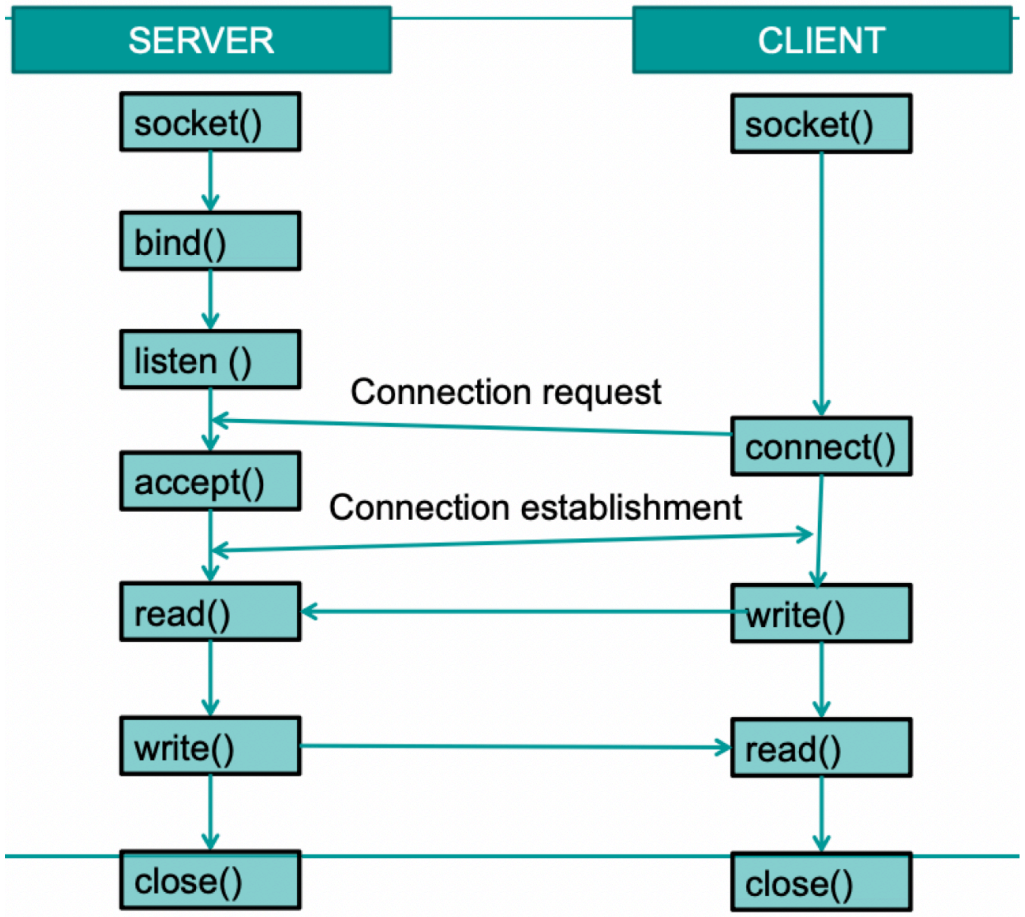
- Checksum: Extra reliability
- Options: provides a way to add extra facilities not covered by the regular header
  - allows each host to specify the **MSS** (**Maximum Segment Size**) it is willing to accept



# Question 6

- Describe with a simple flowchart how a single socket-based client-server communication works?

# Solution 6


SERVER
CLIENT
socket()
socket()
bind()
listen ()
Connection request
connect()
accept()
Connection establishment
read()
write()
write()
read()
close()
close()