UDP and TCP

What do we need in the transport layer?

- Application layer
  - Communication for specific applications
  - e.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP)

- Network layer
  - Global communication between hosts
  - Hides details of the link technology
  - e.g., Internet Protocol (IP)

What Problems Should Be Solved Here?

- Data delivering, to the correct application
  - IP just points towards next protocol
  - Transport needs to demultiplex incoming data (ports)
- Files or bytestreams abstractions for the applications
  - Network deals with packets
  - Transport layer needs to translate between them
- Reliable transfer (if needed)
- Not overloading the receiver
- Not overloading the network

What Is Needed to Address These?

- Demultiplexing: identifier for application process
  - Going from host-to-host (IP) to process-to-process
- Translating between bytestreams and packets:
  - Do segmentation and reassembly
- Reliability: ACKs and all that stuff
- Corruption: Checksum
- Not overloading receiver: “Flow Control”
  - Limit data in receiver’s buffer
- Not overloading network: “Congestion Control”

UDP: Datagram messaging service

UDP provides a connectionless, unreliable transport service

- No-frills extension of “best-effort” IP

- UDP provides only two services to the App layer
  - Multiplexing/Demultiplexing among processes
  - Discarding corrupted packets (optional)

TCP: Reliable, in-order delivery

TCP provides a connection-oriented, reliable, bytestream transport service

- What UDP provides, plus:
  - Retransmission of lost and corrupted packets
  - Flow control (to not overflow receiver)
  - Congestion control (to not overload network)
  - “Connection” set-up & tear-down

Connections (or sessions)

Reliability requires keeping state

- Sender: packets sent but not ACKed, and related timers
- Receiver: noncontiguous packets

Each bytestream is called a connection or session

- Each with their own connection state
- State is in hosts, not network!

What transport protocols do not provide

- Delay and/or bandwidth guarantees
  - This cannot be offered by transport
  - Requires support at IP level (and let's not go there)

- Sessions that survive change-of-IP-address
  - This is an artifact of current implementations
  - As we shall see....

Important Context: Sockets and Ports

- Sockets: an operating system abstraction

- Ports: a networking abstraction
  - This is not a port on a switch (which is an interface)
  - Think of it as a logical interface on a host

Sockets

- A socket is a software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system
  - socketID = socket(..., socket.TYPE)
  - socketID.sendto(message, ...)
  - socketID.recvfrom(...)

- Two important types of sockets
  - UDP socket: TYPE is SOCK_DGRAM
  - TCP socket: TYPE is SOCK_STREAM

Ports

- Problem: which app (socket) gets which packets

- Solution: port as transport layer identifier (16 bits)
  - Packet carries source/destination port numbers in transport header

- OS stores mapping between sockets and ports
  - Port: in packets
  - Socket: in OS

More on Ports

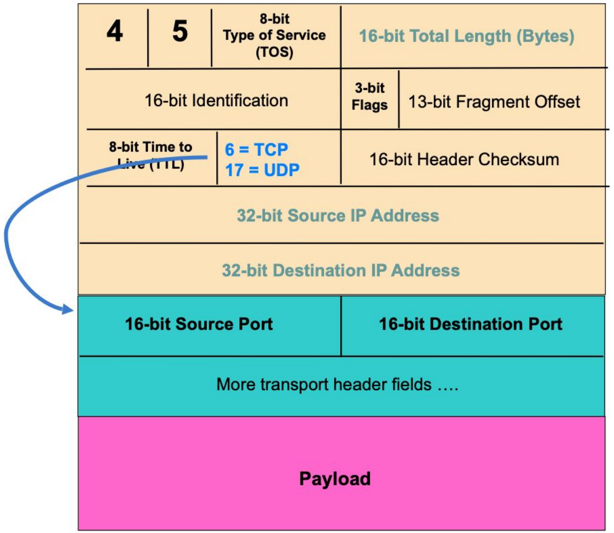
- Separate 16-bit port address space for UDP, TCP

- “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 server
    - Services can listen on well-known port

- Ephemeral ports (most 1024-65535):
    - Given to clients (at random)

Multiplexing and Demultiplexing

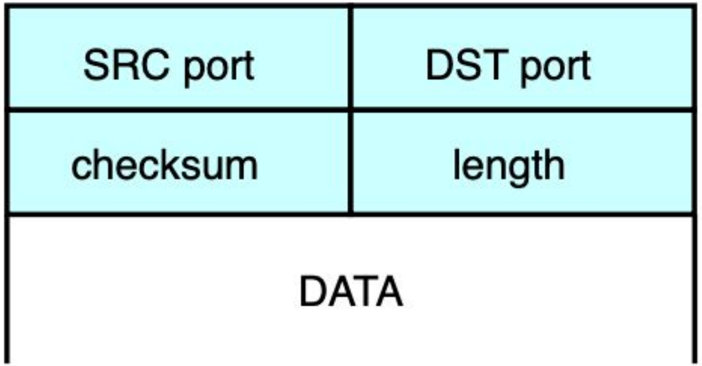
- Host receives IP datagrams
  - Each datagram has source and destination IP address,
  - Each segment has source and destination port number
- Host uses IP addresses and port numbers to direct the segment to appropriate socket



UDP: User Datagram Protocol

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them 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”)



Why Would Anyone Use UDP?

- Finer control over what data is sent and when
  - As soon as an application process writes into the socket
  - … UDP will package the data and send the packet
- No delay for connection establishment
  - UDP just blasts away without any formal preliminaries
  - … which avoids introducing any unnecessary delays
- No connection state
  - No allocation of buffers, sequence #s, timers …
  - … making it easier to handle many active clients at once
- Small packet header overhead
  - UDP header is only 8 bytes

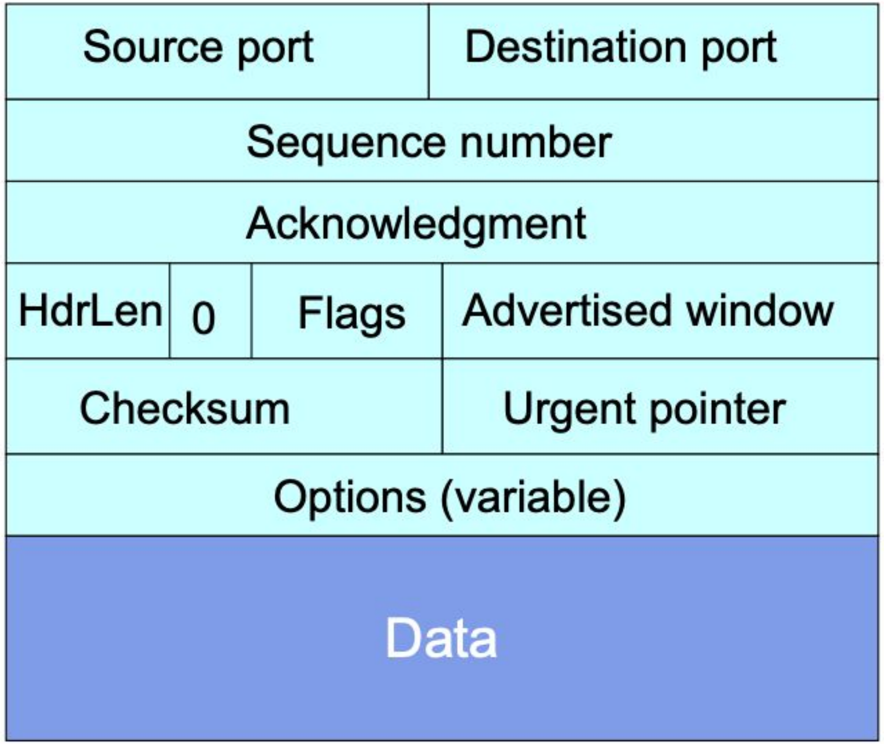
Basic Components of Reliability

- ACKs
  - Can’t be reliable without knowing whether data has arrived
  - TCP uses byte sequence numbers to identify payloads
- Checksums
  - Can’t be reliable without knowing whether data is corrupted
  - TCP does checksum over TCP and pseudoheader
- Timeouts and retransmissions
  - Can’t be reliable without retransmitting lost/corrupted data
  - TCP retransmits based on timeouts and duplicate ACKs
  - Timeout based on estimate of RTT

Other TCP Design Decisions

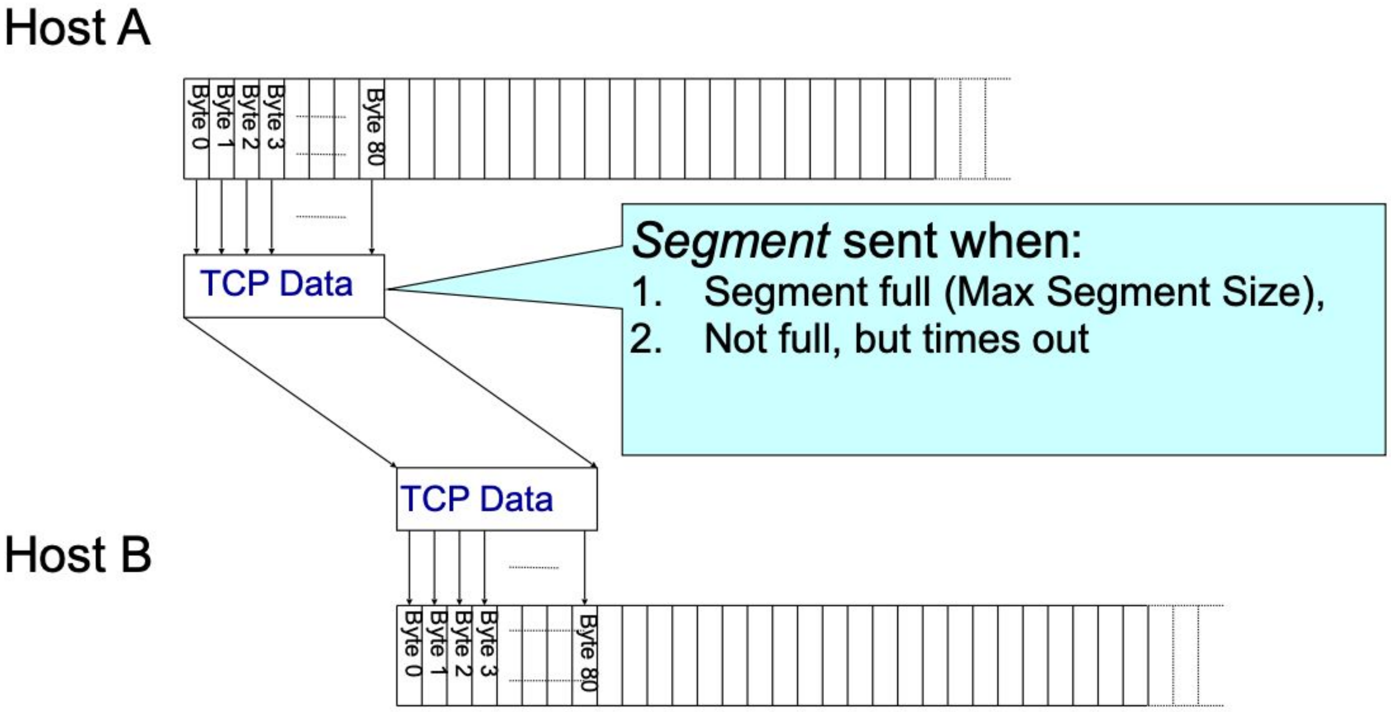
- Sliding window flow control
  - Allow W contiguous bytes to be in flight
- Cumulative acknowledgements
  - Selective ACKs (full information) also supported
- Single timer set after each payload is ACKed
  - Timer is effectively for the “next expected payload”
  - When timer goes off, resend that payload and wait
    - And double timeout period
- Various tricks related to “fast retransmit”
  - Using duplicate ACKs to trigger retransmission

TCP Header



Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

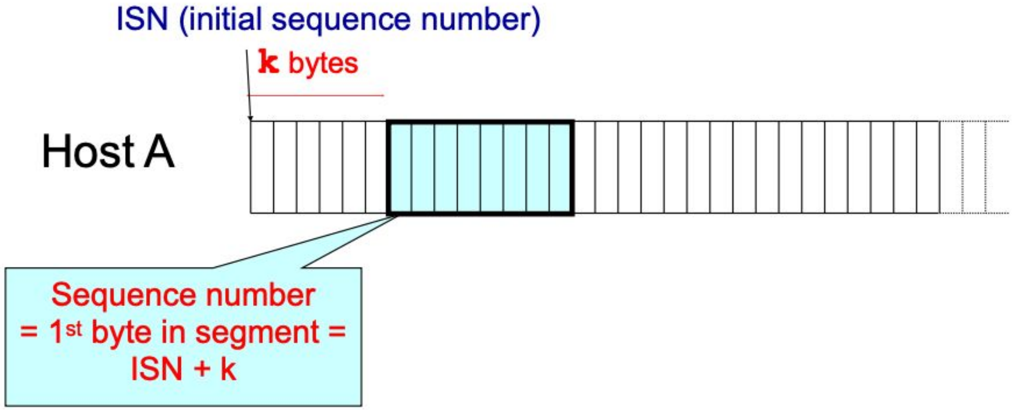
… Provided Using TCP “Segments”



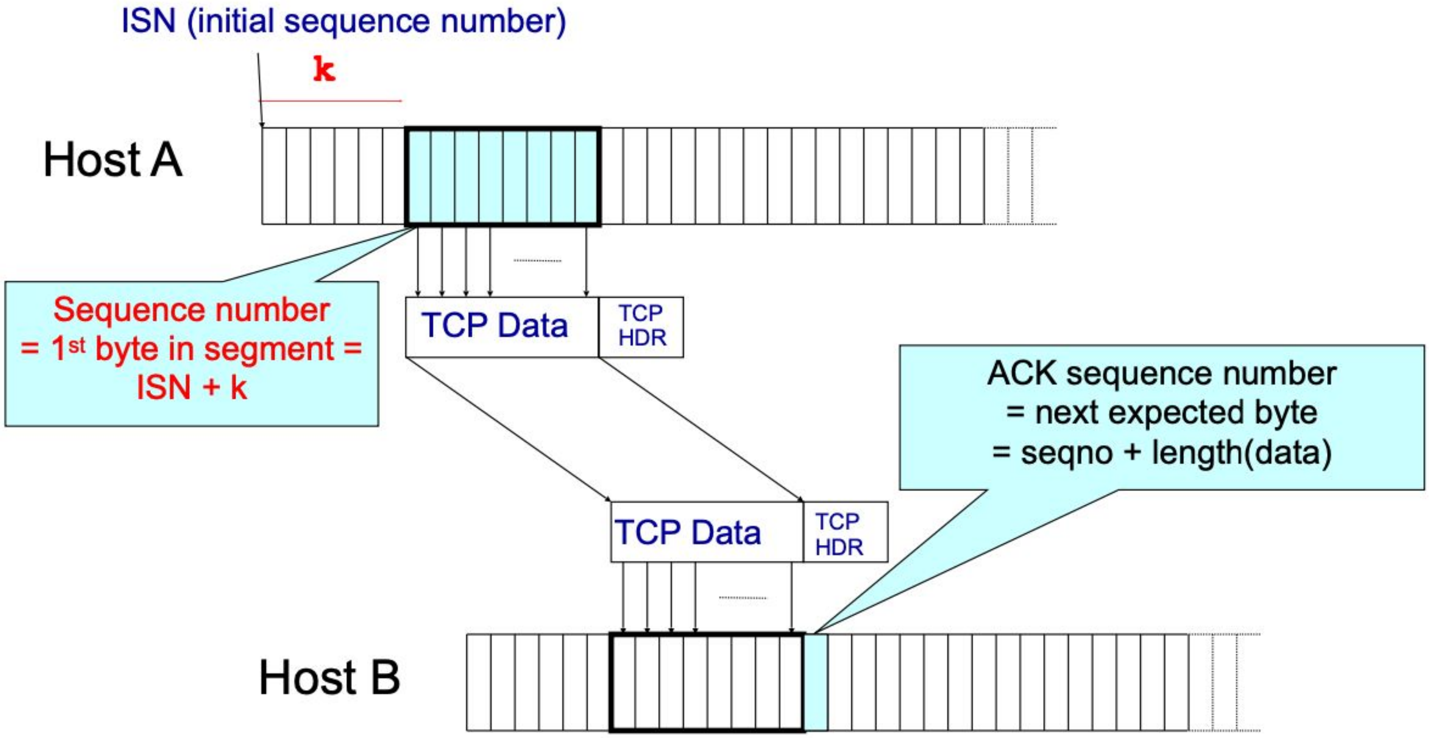
TCP Segment

- 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 Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)

Sequence Numbers



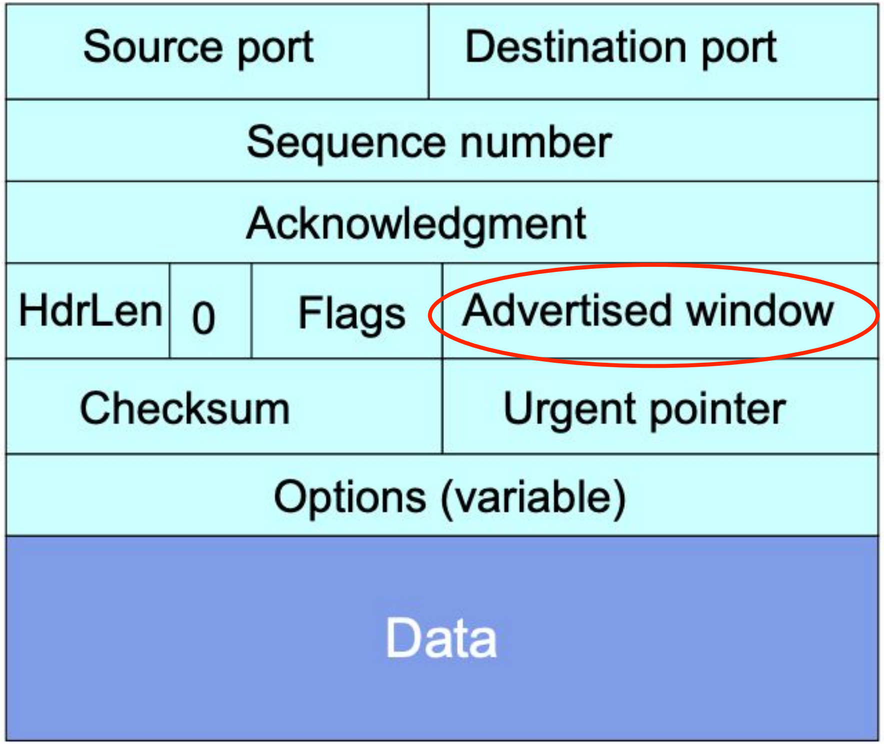
Sequence Numbers



ACKing and Sequence Numbers

- Sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes
    - X, X+1, X+2, ...X+B-1
- Upon receipt of packet, 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

TCP Header


Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

Sliding Window Flow Control

- 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

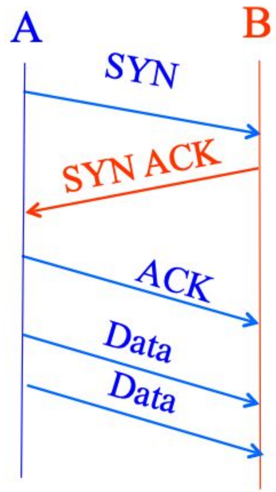
Advertised Window Limits Rate

Sender can send no faster than W/RTT bytes/sec

Receiver only advertises more space when it has consumed old arriving data

In original TCP design, that was the sole protocol mechanism controlling sender's rate

Establishing a TCP Connection

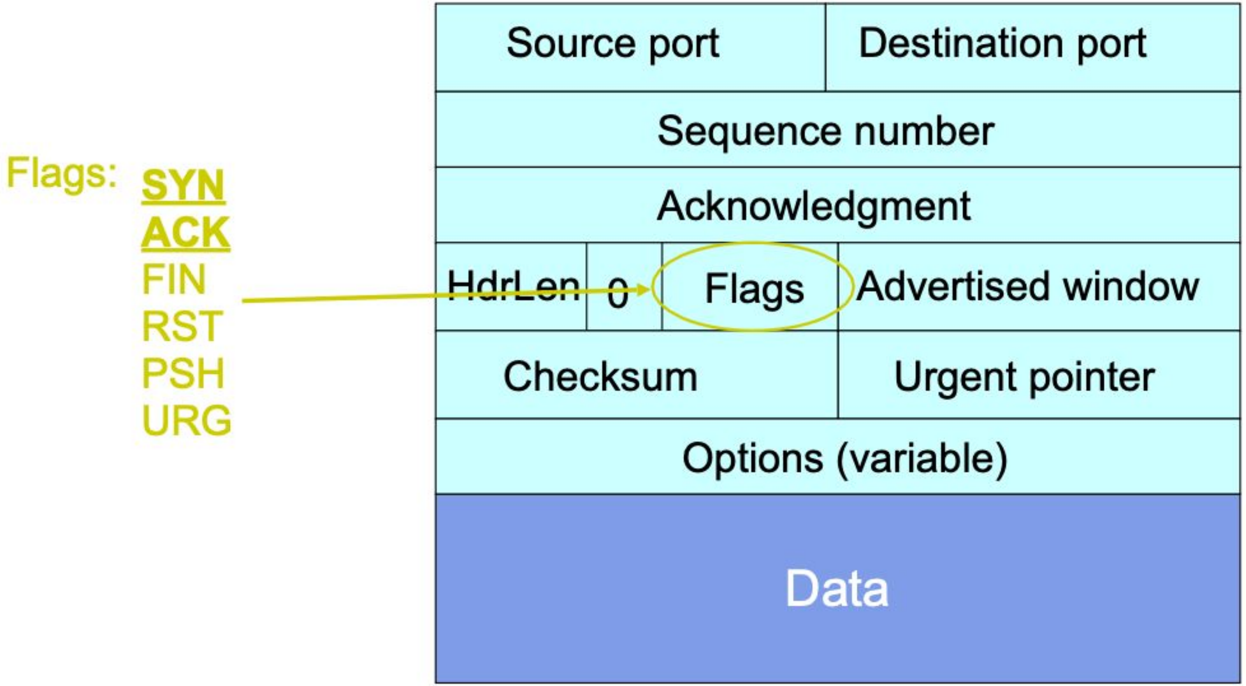


Each host tells its ISN to the other host.

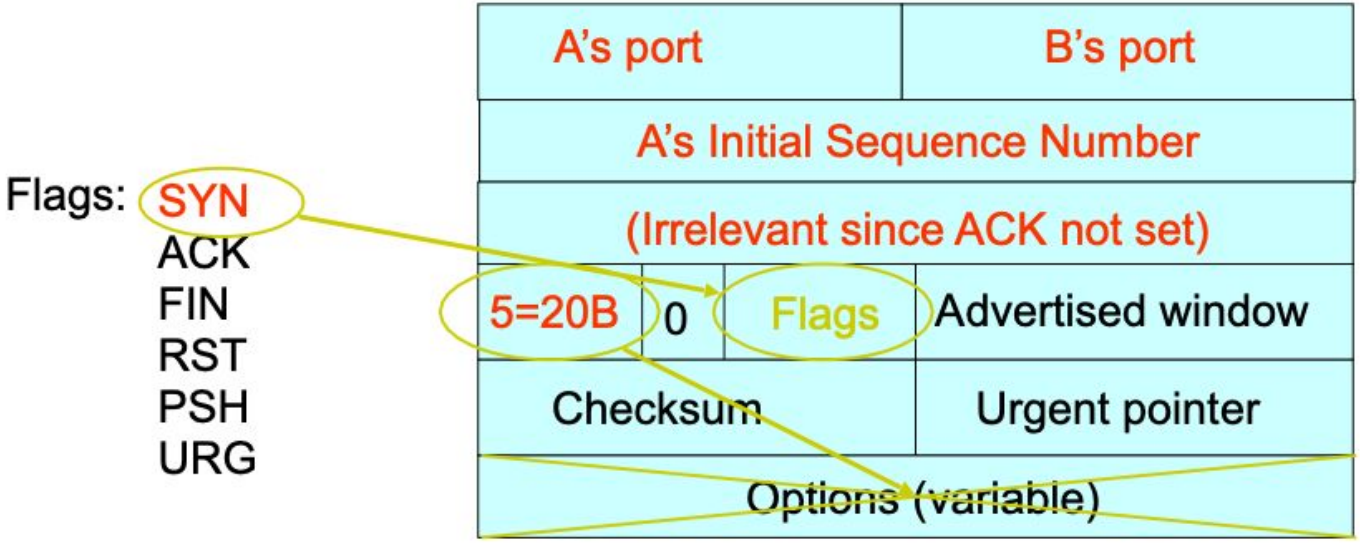
Three-way handshake to establish connection

- Host A sends a **SYN** (open; “synchronize sequence numbers”)
- Host B returns a SYN acknowledgment (**SYN ACK**)
- Host A sends an **ACK** to acknowledge the SYN ACK

TCP Header

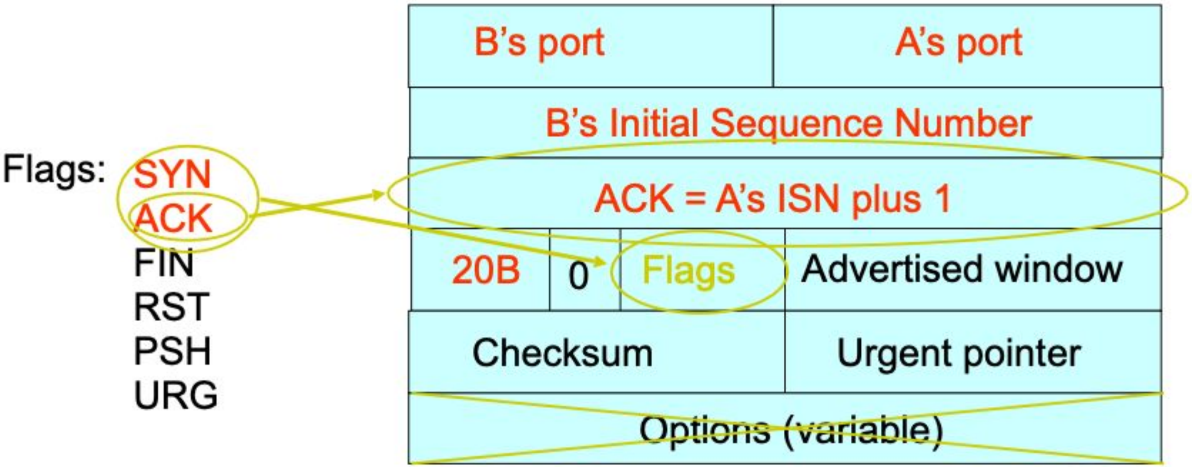

Flags:
SYN
ACK
FIN
RST
PSH
URG
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

Handshake step 1: A's initial SYN packet



A tells B it wants to open a connection…

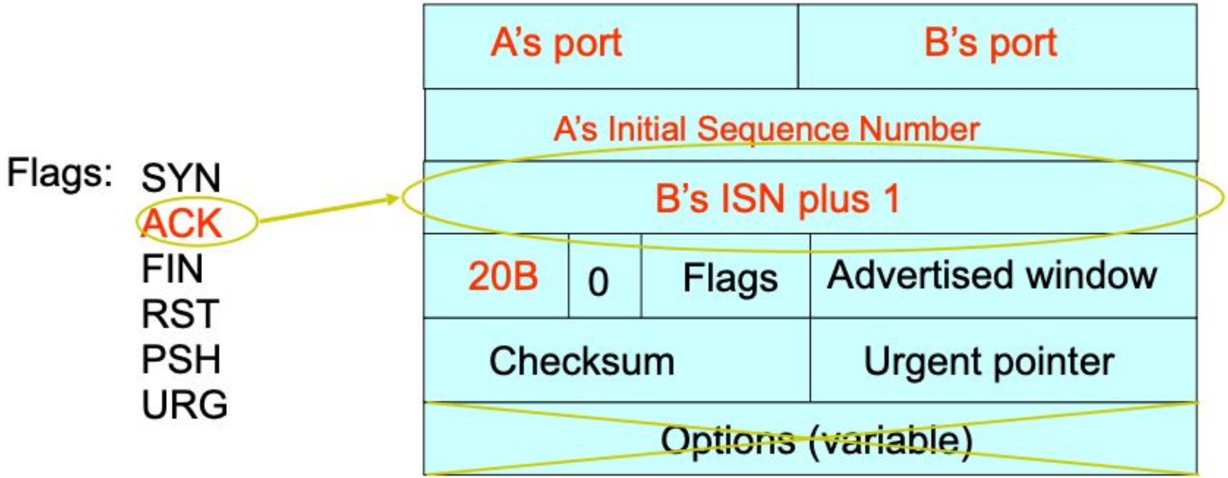
Handshake step 2: B’s SYN-ACK packet



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

… upon receiving this packet, A can start sending data

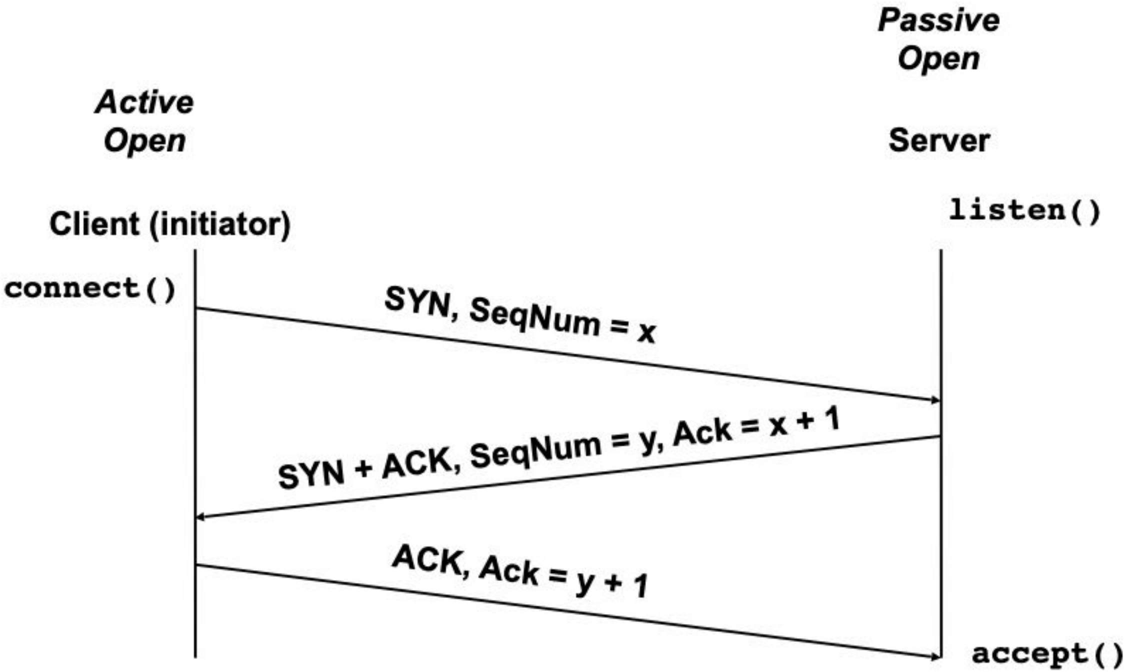
Handshake step 3: A's ACK of the SYN-ACK packet



A tells B it's likewise okay to start sending

… upon receiving this packet, B can start sending data

Timing Diagram: 3-Way Handshaking



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 & 2988) use default of 3 seconds
    - Other implementations instead use 6 seconds