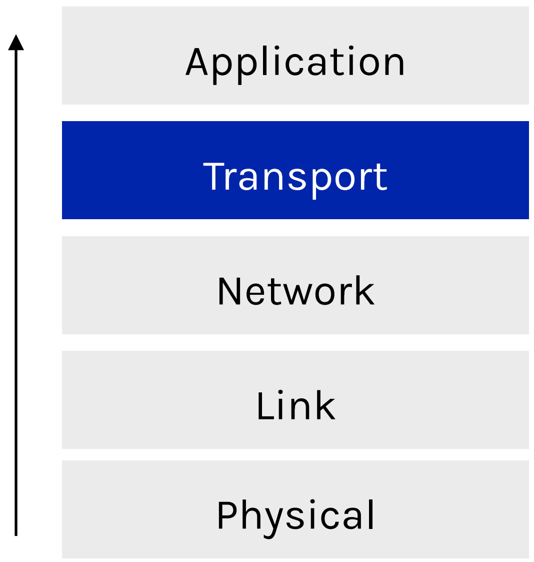
Computer Networks (WS23/24)

L8: The Transport Layer - Part 2

Prof. Dr. Lin Wang
Computer Networks Group (PBNet)
Department of Computer Science
Paderborn University

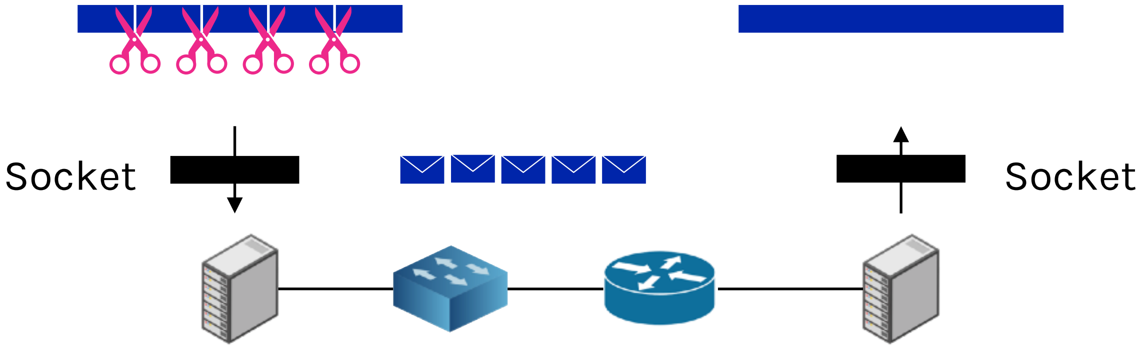
Materials inspired by Laurent Vanbever

Learning objectives



Part 2

- Requirements
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)



Requirements

What do we need in the transport layer?

Functionality implemented in the network

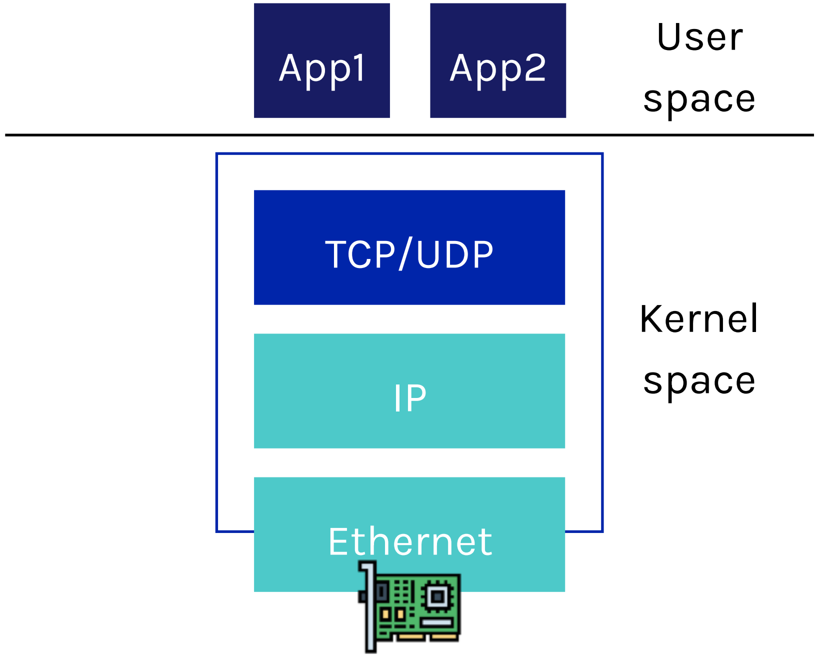
- Keep minimum (easy to build, broadly applicable)

Functionality implemented in the application

- Keep minimum (easy to write)
- Restricted to application-specific functionality

Functionality implemented in the network stack

- Shared networking code on the host
- Relieves burden from both the application and network



What do we need in the transport layer?

Application layer

- Communication for specific applications
- Example: Hyper Text Transfer Protocol (HTTP), File Transfer Protocol (FTP)

Network layer

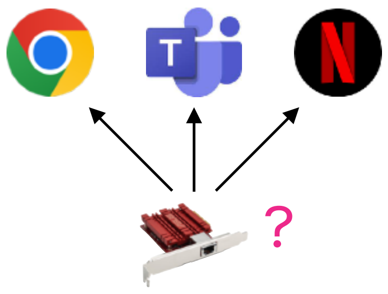
- Global communication between hosts
- Hides details of the link technology
- Example: Internet Protocol (IP)

Transport layer: bridging the gap between the two

What is the gap?

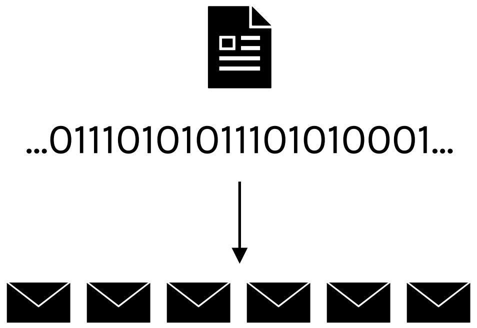
Data delivering, to the correct application

- IP just points towards next protocol
- Transport needs to demultiplex incoming data



Files or bytestreams abstractions for the application

- Network deals with packets
- Transport needs to translate between the two



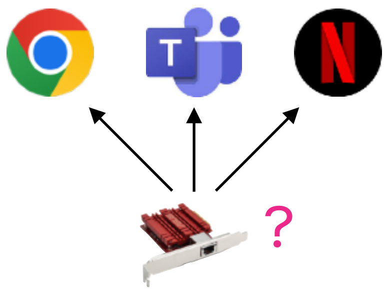
Others

- Reliable transfer (if needed), not overloading anyone

Transport layer functionality

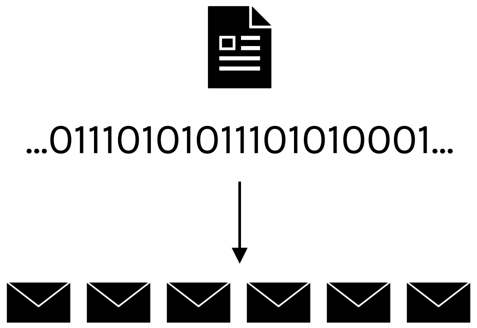
Demultiplexing: identifier for application process

- From host-to-host (IP) to process-to-process



Bytestream - packet translation

- Segmentation and reassembly



Reliability: checksums, ACKs, timeouts

Not overloading the receiver: flow control

Not overloading the network: congestion control

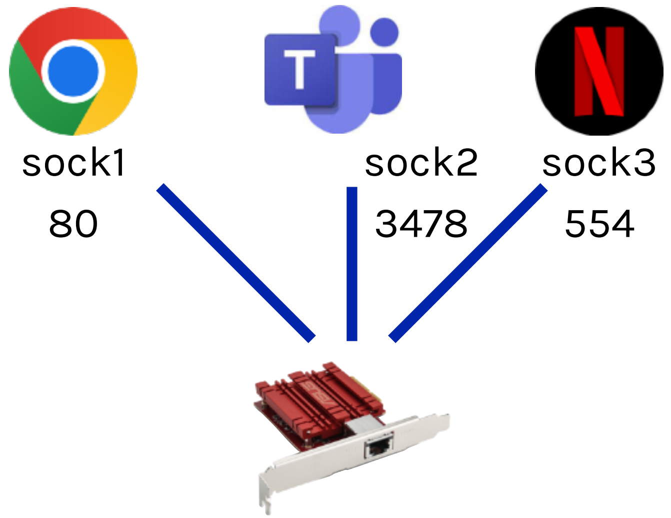
Demultiplexing: sockets and ports

Sockets

- An operating system abstraction

Ports

- A networking abstraction
- Not a physical port on a switch/router (which is a network interface)
- Think of it as a logical interface on a host



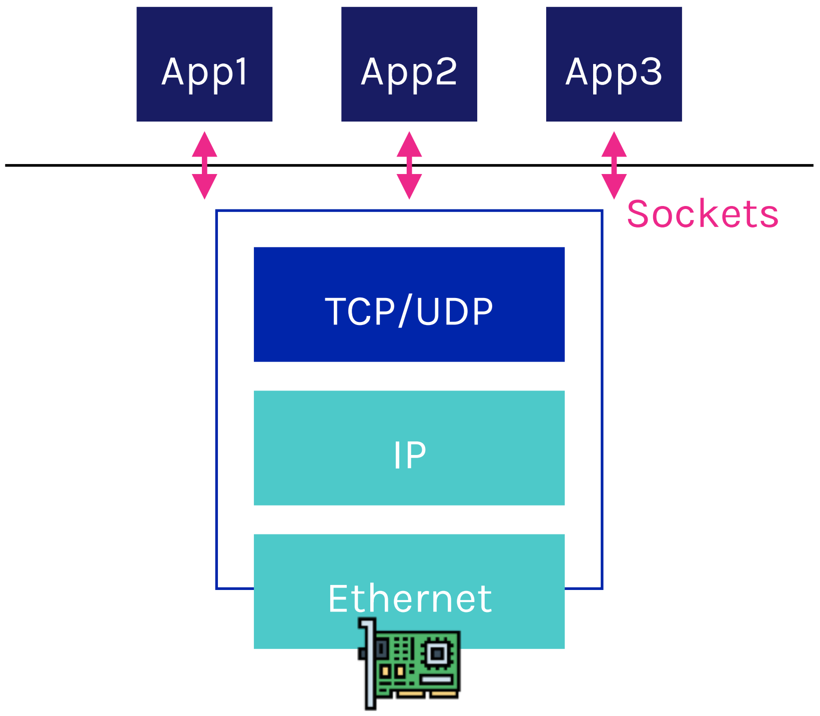
Sockets

A socket is a software abstraction by which an application process exchanges network messages with the (transport layer in the) OS

- socket_id = socket(..., socket.TYPE)
- socket_id.sendto(message, ...)
- socket_id.recvfrom(...)

Two important types of sockets

- UDP socket: TYPE = SOCK_DGRAM
- TCP socket: TYPE = SOCK_STREAM



Ports

Problem to solve

- Which app (socket) gets which packets?

Solution

- Port as transport layer identifier (16 bits)
- Packets carry source/destination port numbers in the transport layer header

Mapping between ports and sockets

- OS stores the mapping

Ports

Seperate 16-bit port address space for UDP, TCP

System or well-known ports (0-1023)

- Agreement on which services run on these ports, e.g., 22 (SSH), 80 (HTTP)
- Client (App) knows appropriate port on server; services can listen on well-known ports

Registered ports (1024-49151)

- Designated for use with a certain protocol or application

Ephemeral (or dynamic, private) ports (49152-65535)

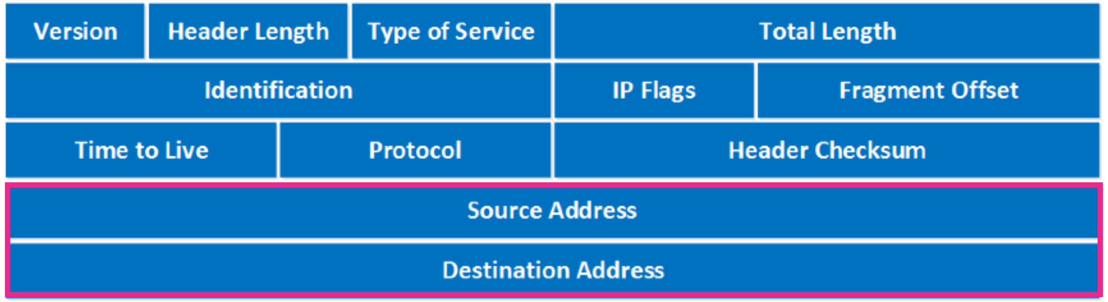
- Given to clients (at random)

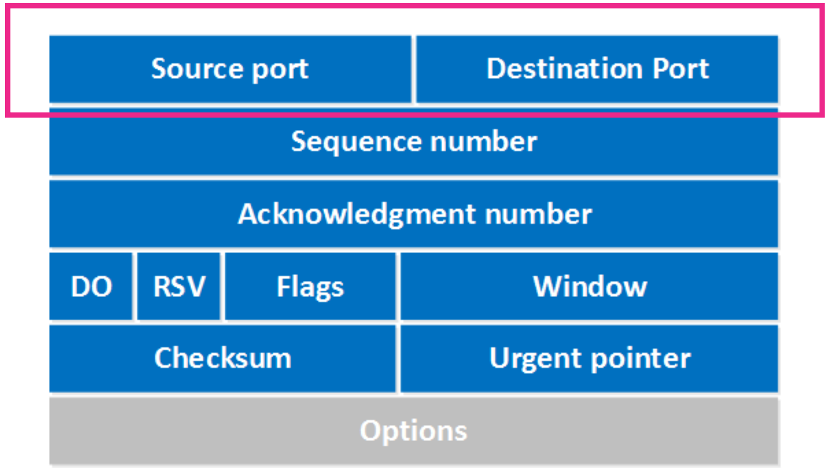
Multiplexing and demultiplexing in TCP

Host receives IP packets

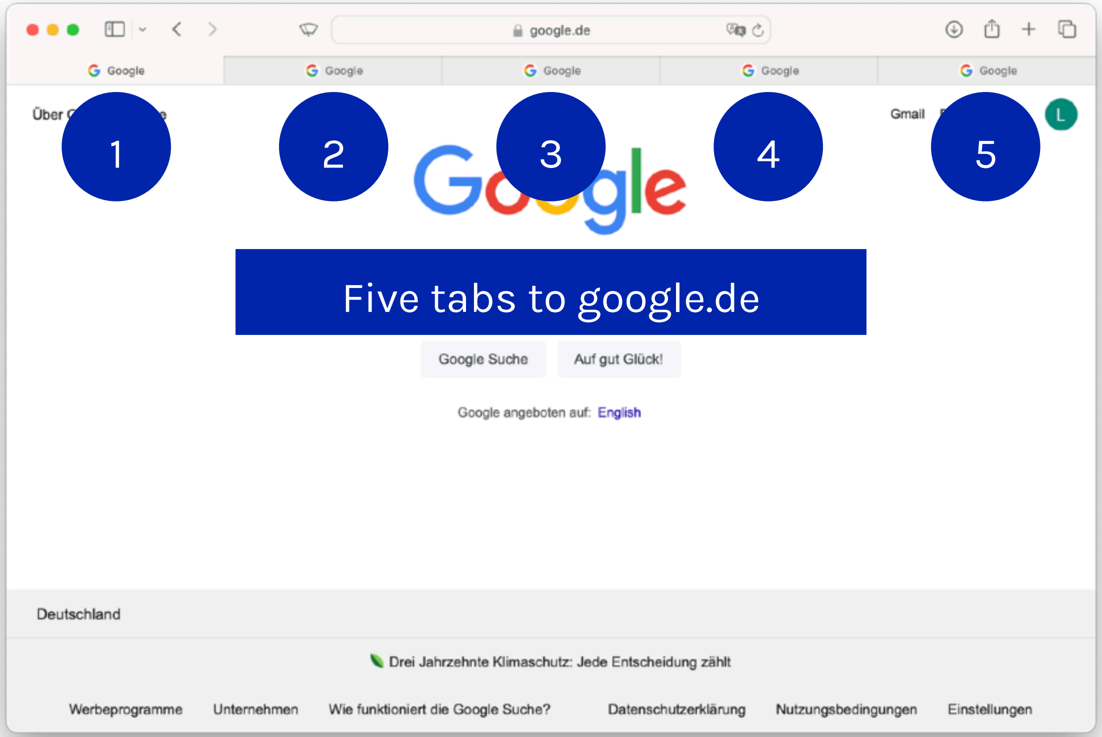
- Each IP packet has source and destination IP addresses
- Each TCP segment has source and destination port number

Host uses IP addresses and port numbers to direct the segment to appropriate socket: a socket is identified by a 4-tuple (SrcIP, SrcPort, DstIP, DstPort)





TCP socket example



Your IP: 131.234.250.184

Google's IP: 142.250.181.206

TCP socket example

Socket

Client OS	Source IP	Source port	Destination IP	Destination port
1	131.234.250.184	54001	142.250.181.206	443
2	131.234.250.184	56320	142.250.181.206	443
3	131.234.250.184	63584	142.250.181.206	443
4	131.234.250.184	55003	142.250.181.206	443
5	131.234.250.184	65076	142.250.181.206	443

Socket

Server OS	Source IP	Source port	Destination IP	Destination port
1	142.250.181.206	443	131.234.250.184	54001
2	142.250.181.206	443	131.234.250.184	56320
3	142.250.181.206	443	131.234.250.184	63584
4	142.250.181.206	443	131.234.250.184	55003
5	142.250.181.206	443	131.234.250.184	65076

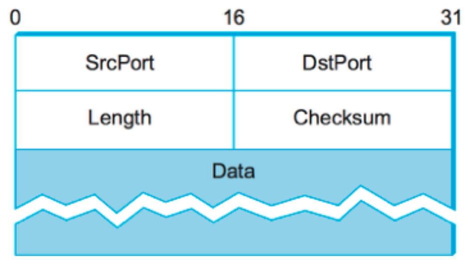
Multiplexing and demultiplxing in UDP

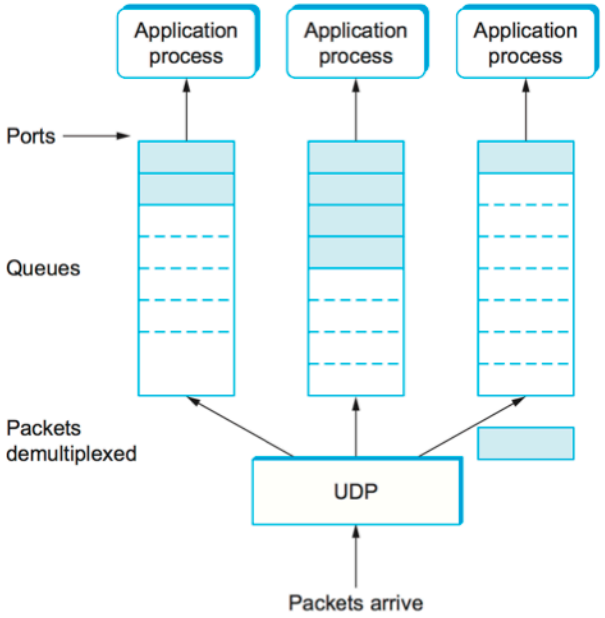
Host receives IP packets

- Each IP packet has the destination port

Host uses the destination port to direct the segment to appropriate socket

Application process distinguishes the UDP datagram with the source IP and/or port





https://book.systemsapproach.org/e2e/udp.html

User Datagram Protocol (UDP)

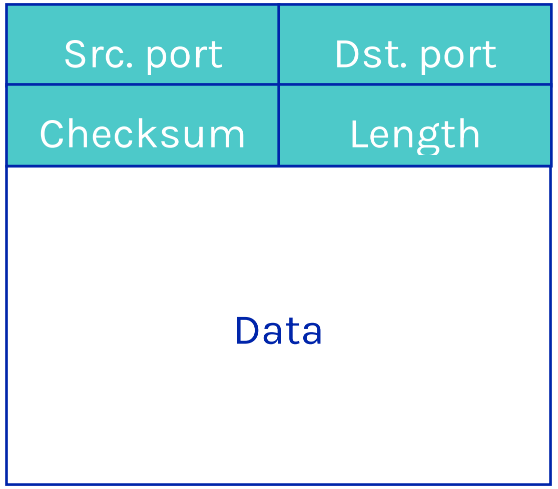
UDP

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 do not verify checksum)



Why would anyone use UDP?

Finer control over what data is sent and when

- As soon as data is written into the socket, UDP will package it and send the packet

No delay for connection establishment

- No formal preliminaries, avoids introducing any unnecessary delays

No connection state

- No allocation of buffers, sequence numbers, timers, etc., easy to handle many clients

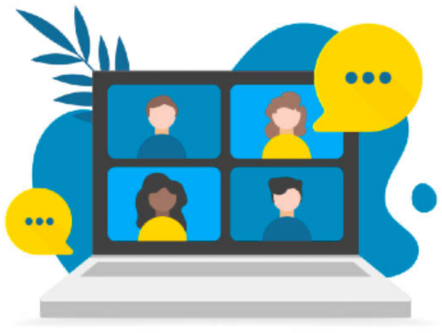
Small packet header overhead

- UDP header is only 8 bytes

Popular applications that use UDP

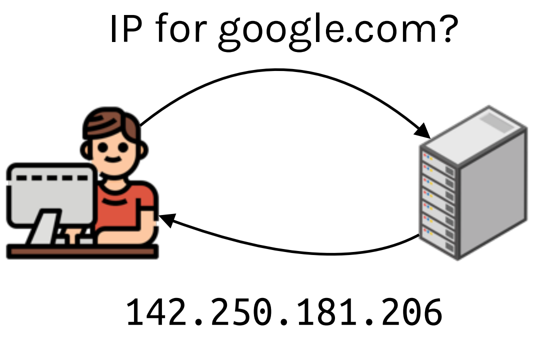
Interactive streaming applications

- Retransmitting lost/corrupted packets often pointless
- By the time the packet is retransmitted, it is too late
- Examples: telephone calls, video conferencing, gaming
- However, modern video streaming protocols use TCP (and HTTP)



Simple query protocols like Domain Name System (DNS)

- Connection establishment overhead would double cost
- Easier to have application retransmit if needed



Transmission Control Protocl
(TCP)

TCP

Reliable, in-order delivery

- Ensure byte stream (eventually) arrives intact
- In the presence of corruption and loss

Connection oriented: explicit set-up and tear-down of TCP session

Fully duplex stream of bytes service: stream of bytes instead of messages

Flow control: ensures that sender does not overwhelm receiver

Congestion control: dynamic adaptation to network path's capacity **(next time)**

Reliability recap

ACKs: cannot be reliable without knowing whether the data has arrived

- TCP uses byte sequence numbers to identify payloads

Checksums: cannot be reliable without knowing whether data is corrupted

- TCP does checksum over TCP and parts of IP header

Timeouts/retransmission: cannot be reliable without retransmitting lost/corrupted data

- TCP retransmits based on timeouts and duplicate ACKs
- Timeout is set based on estimate of RTTs

Other TCP design decisions

Sliding-window flow control

- Allows W contiguous bytes to be in flight

Cumulative ACKs

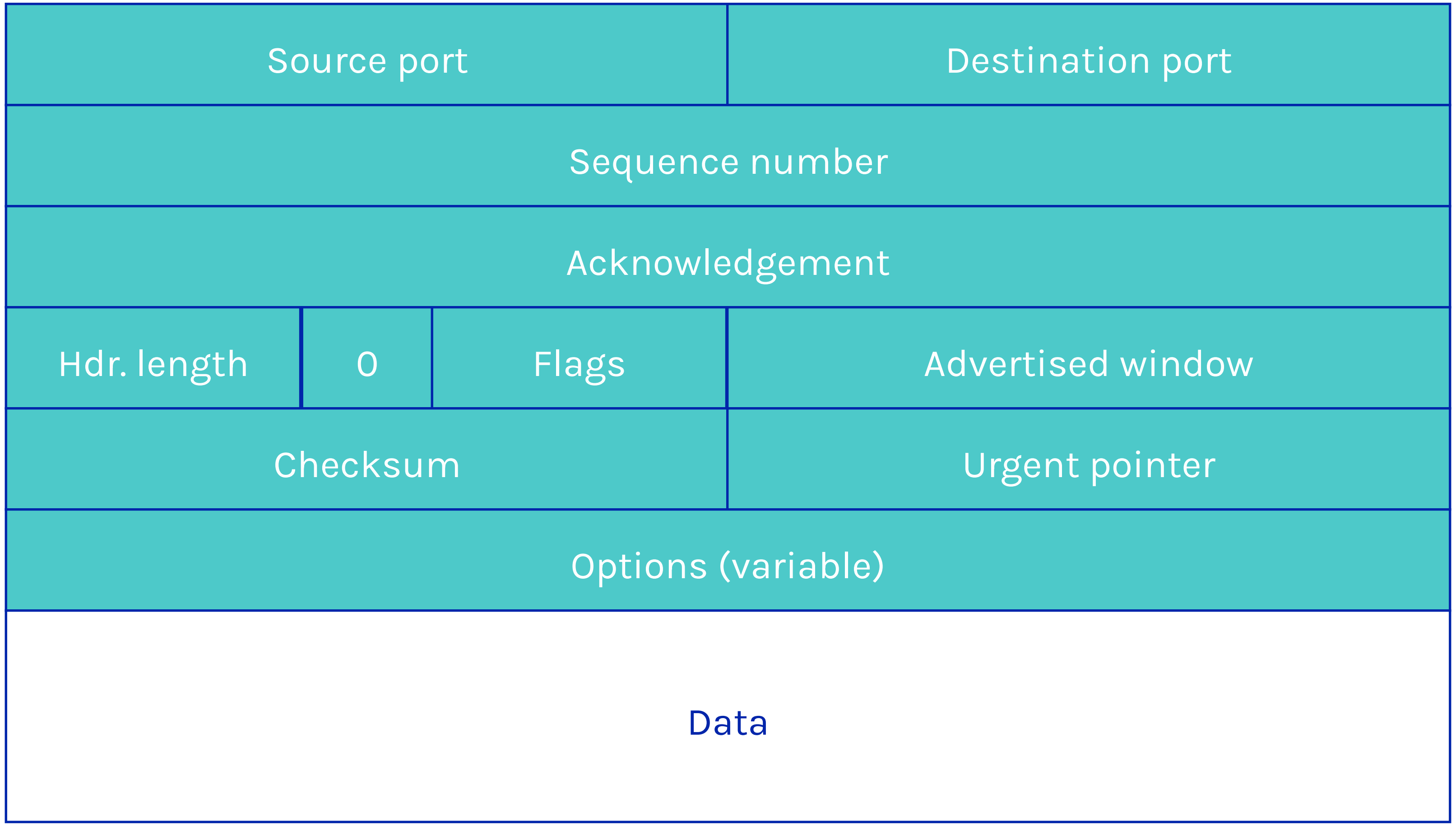
- Selective ACKs (full information) also supported

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)

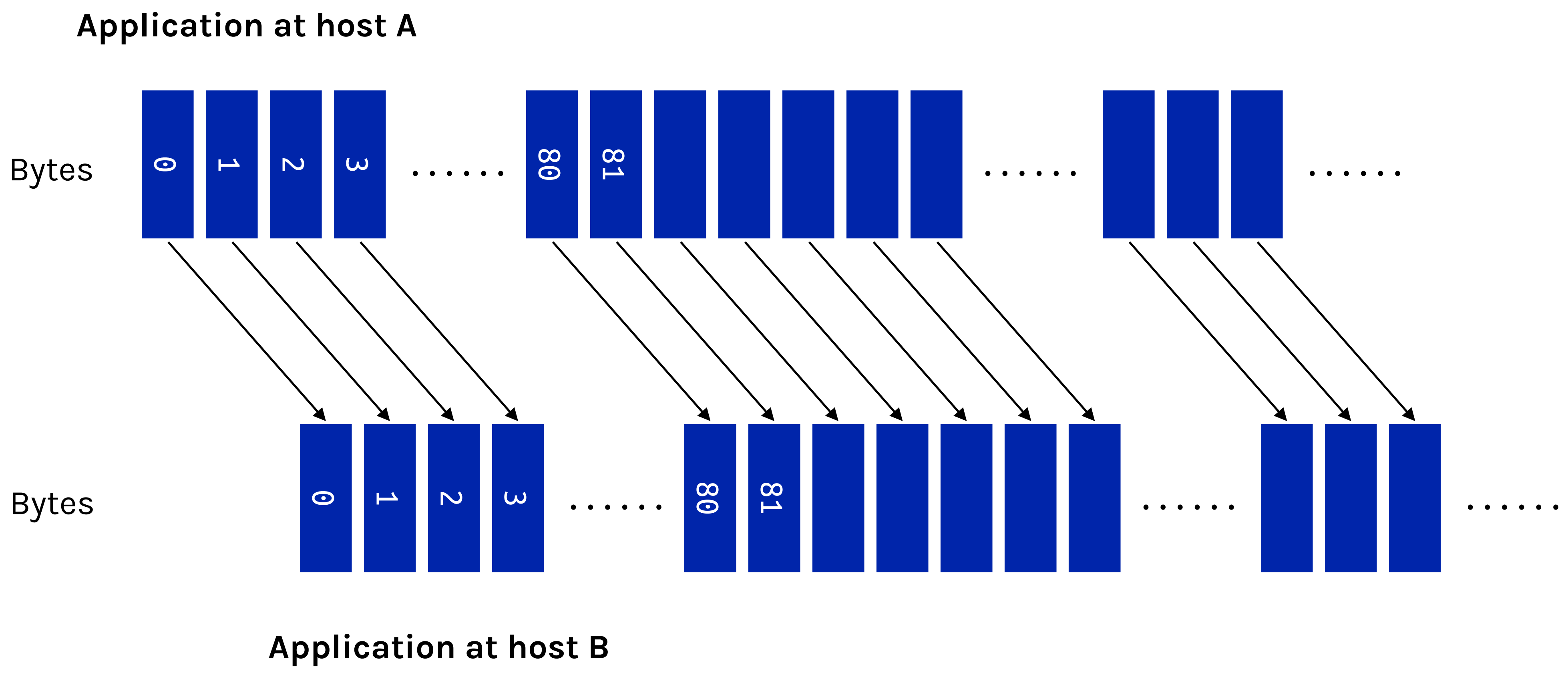
Various tricks for "fast retransmit": using duplicate ACKs to trigger retransmission

TCP header

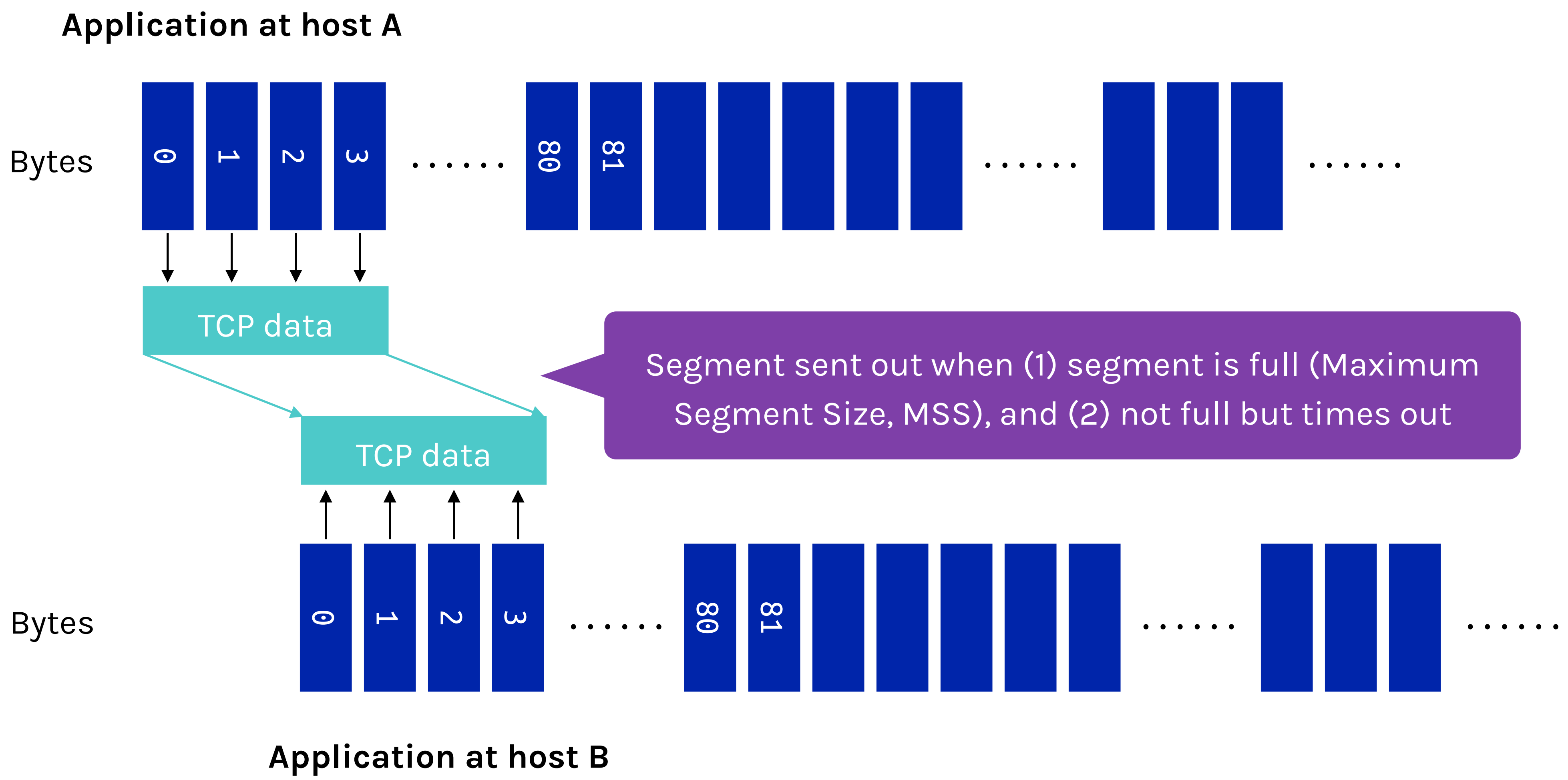


Source port
Destination port
Sequence number
Acknowledgement
Hdr. length
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TCP "stream of bytes" service



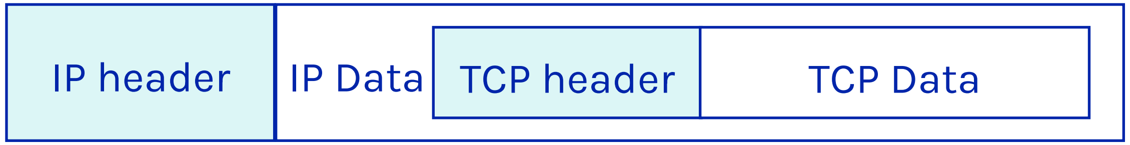
TCP segmentation



TCP segment

IP packet

- No bigger than Maximum Transmission Unit (MTU)
- Example: up to 1500 bytes with Ethernet



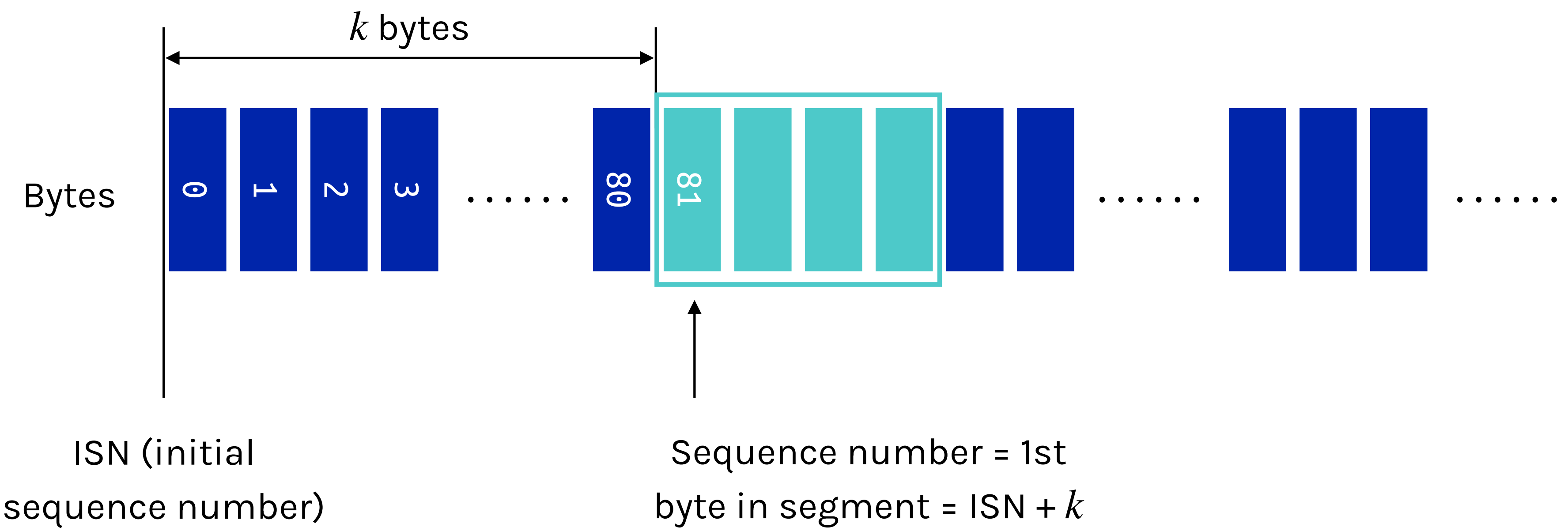
TCP packet

- IP packet with a TCP header (>= 20 bytes long) and data inside

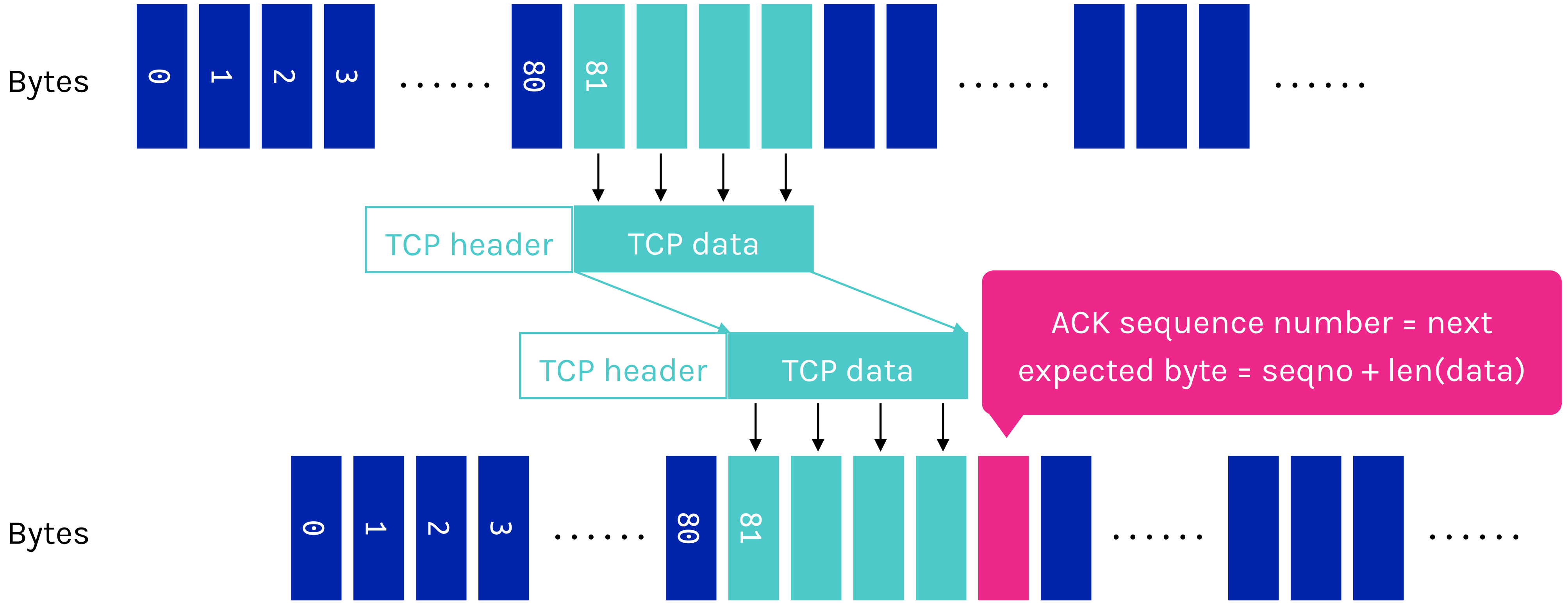
TCP segment

- No more than Maximum Segment Size (MSS) = MTU - IPHdr - TCPHdr
- Example: up to 1460 consecutive bytes from the stream

Sequence number



Acknowledgement number



Sequence and ACK numbers

Sender sends packet

- Data starts with sequence number X
- Packet contains B bytes: $X, X+1, \ldots, X+B-1$

Upon receipt of packet, receiver sends an ACK

- If all data prior to X already received: ACK $X+B$ (next expected byte)
- If highest contiguous byte received is a smaller value Y: ACK $Y+1$ even if it has been ACKed before

Normal pattern

Segment #1

- Sender: seqno = X, length = B
- Receiver: ACK = $X + B$

Segment #2

- Sender: seqno = $X + B$, length = B
- Receiver: ACK = $X + 2B$

Segment #3

- Sender: seqno = $X + 2B$, length = B

Seqno of next packet is the same as last ACK number

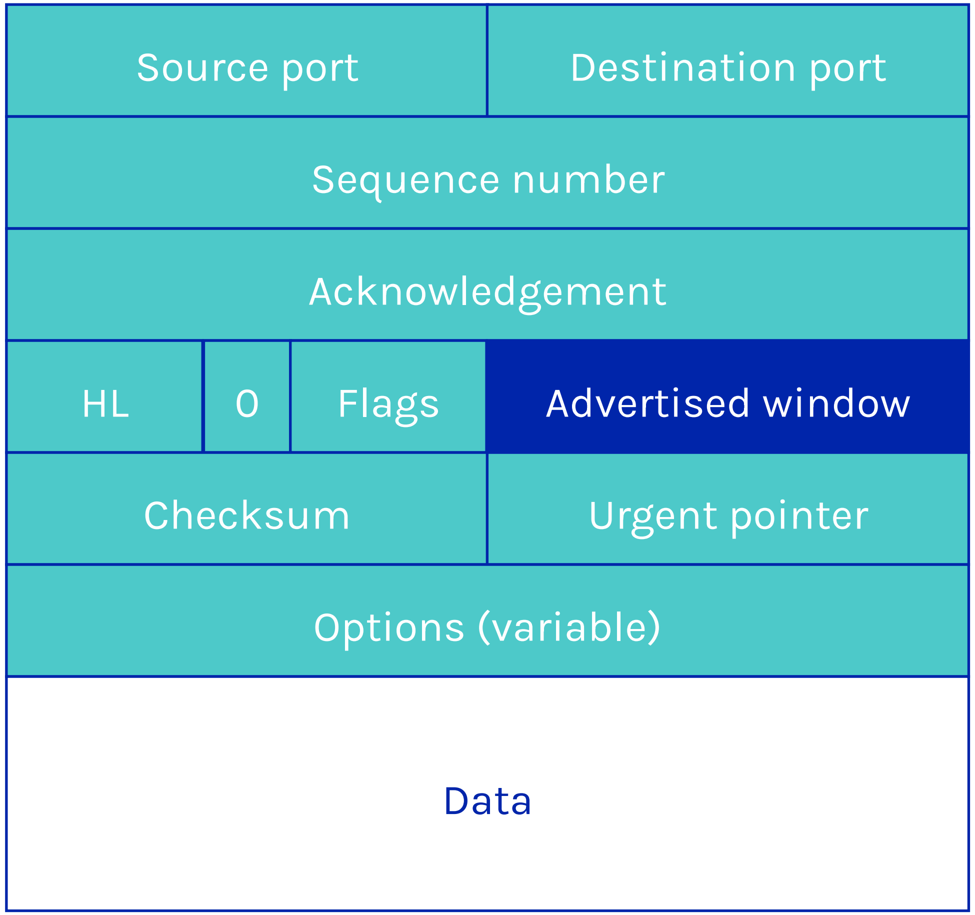
Sliding window flow control

Advertised window W

- Can send W bytes beyond the next expected byte

Receiver uses W to prevent sender from overflowing its buffer

Limites the number of bytes sender can have in flight



Rate limiting with advertised window

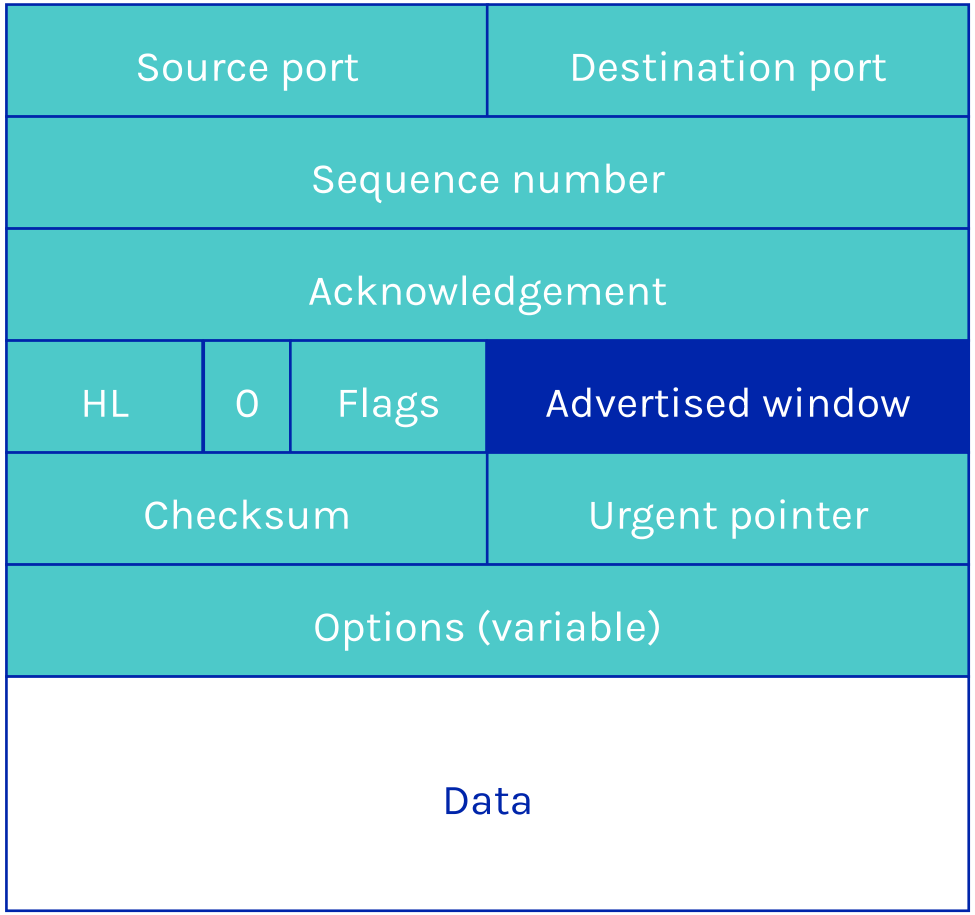
Sender can send no faster than W/RTT bytes per second

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

- Advertises 0 when buffer is full

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

What is missing?



Implementing sliding window

Both sender and receiver maintain a window

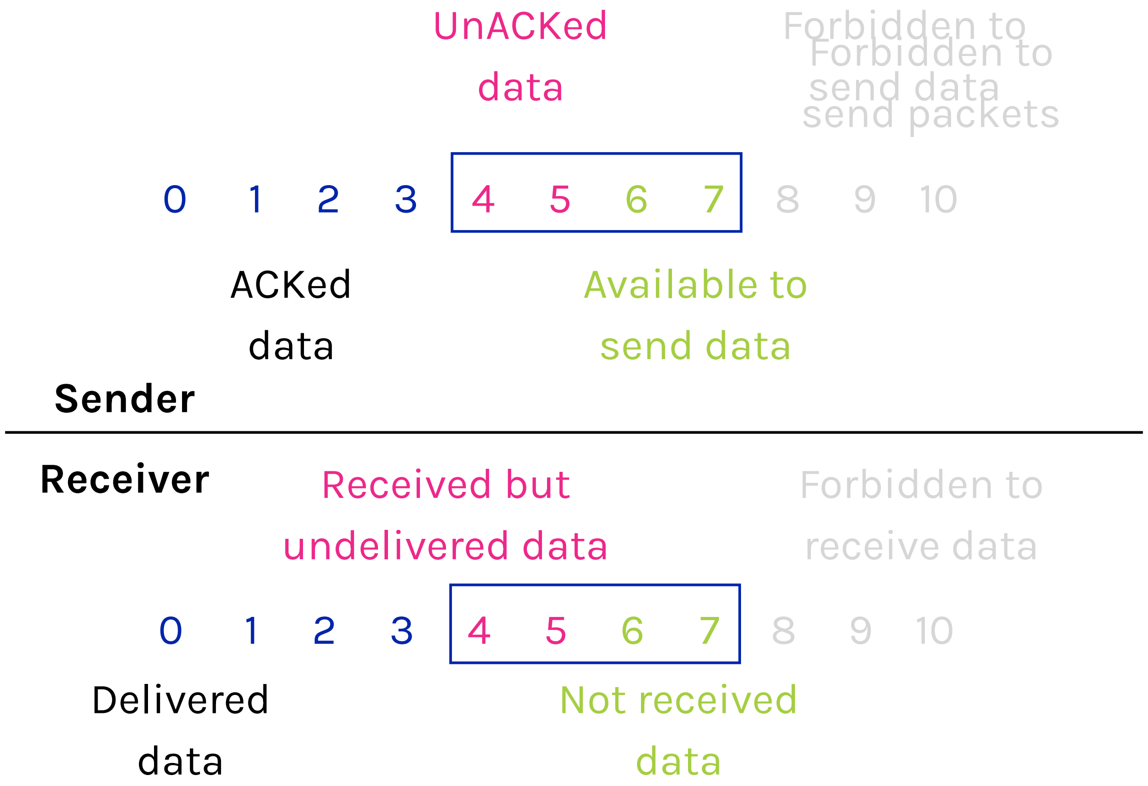
- Sender: not yet ACKed
- Receiver: not yet delivered to application

Left edge of window

- Sender: beginning of unACKed data
- Receiver: beginning of undelivered data

Window size

- Maximum amount of data in flight (sender) and of undelivered data (receiver)



Sliding window summary

Sender

- Window **advances** when new data ACKed

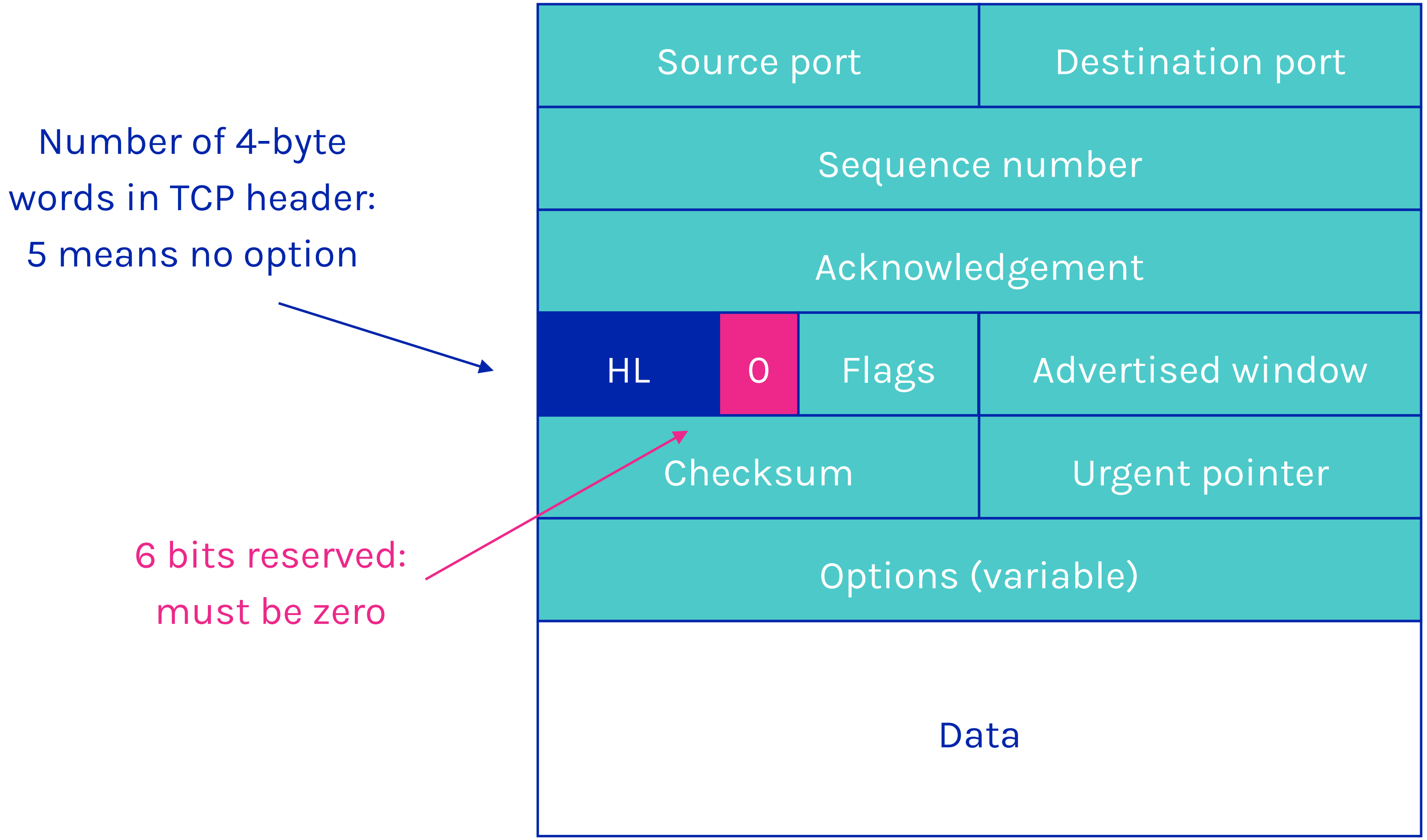
Receiver

- Window **advances** as receiving process consumes data

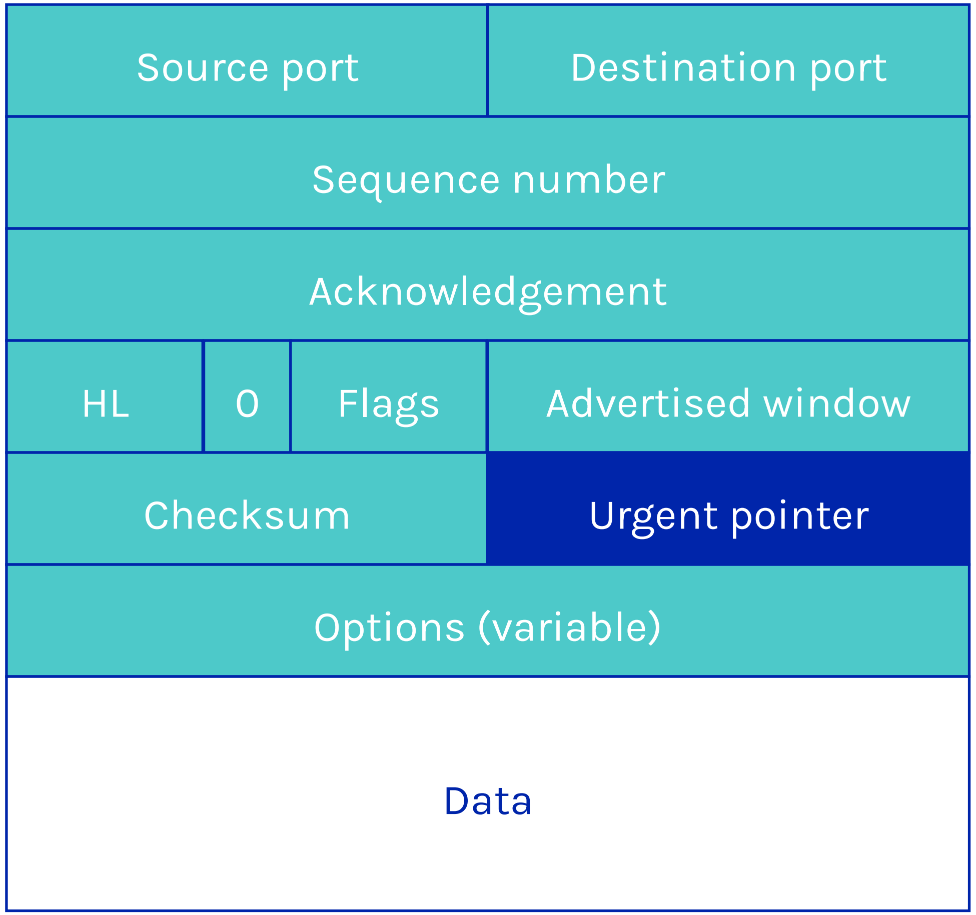
Receiver advertises to sender where the receiver window currently ends (righthand edge)

- Sender agrees not to exceed this amount
- It makes sure by setting its own window size to a value that cannot send beyond the receiver's righthand edge

Other TCP header fields



Other TCP header fields



Used with URG flag to indicate urgent data

Initial sequence number

Sequence number for the very first byte

- Why not just use ISN = 0?

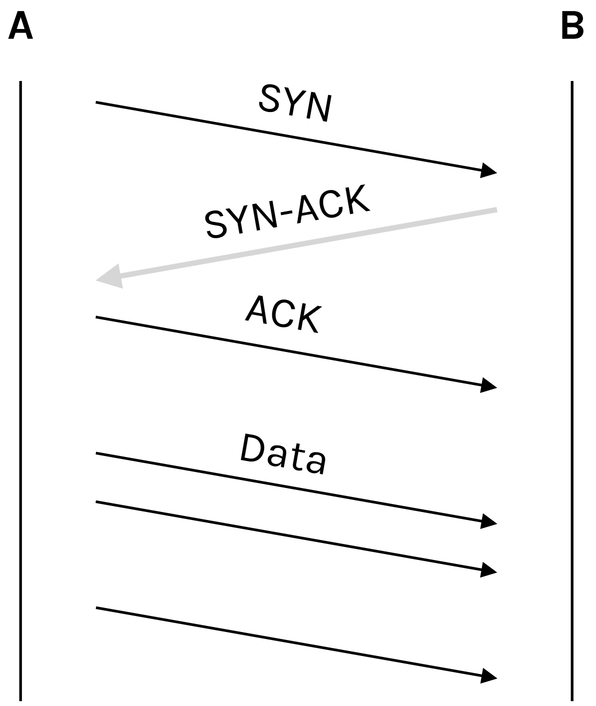
Practical issues

- IP addresses and ports uniquely identify a connection
- Eventually, though, these port numbers do get used again and there is a small chance that a packet from an old connection is still in flight

TCP therefore requires changing ISN

- Initially set from 32-bit clock that ticks every 4 microseconds, now draw from a pseudo random number generator (security)

TCP connection establishment

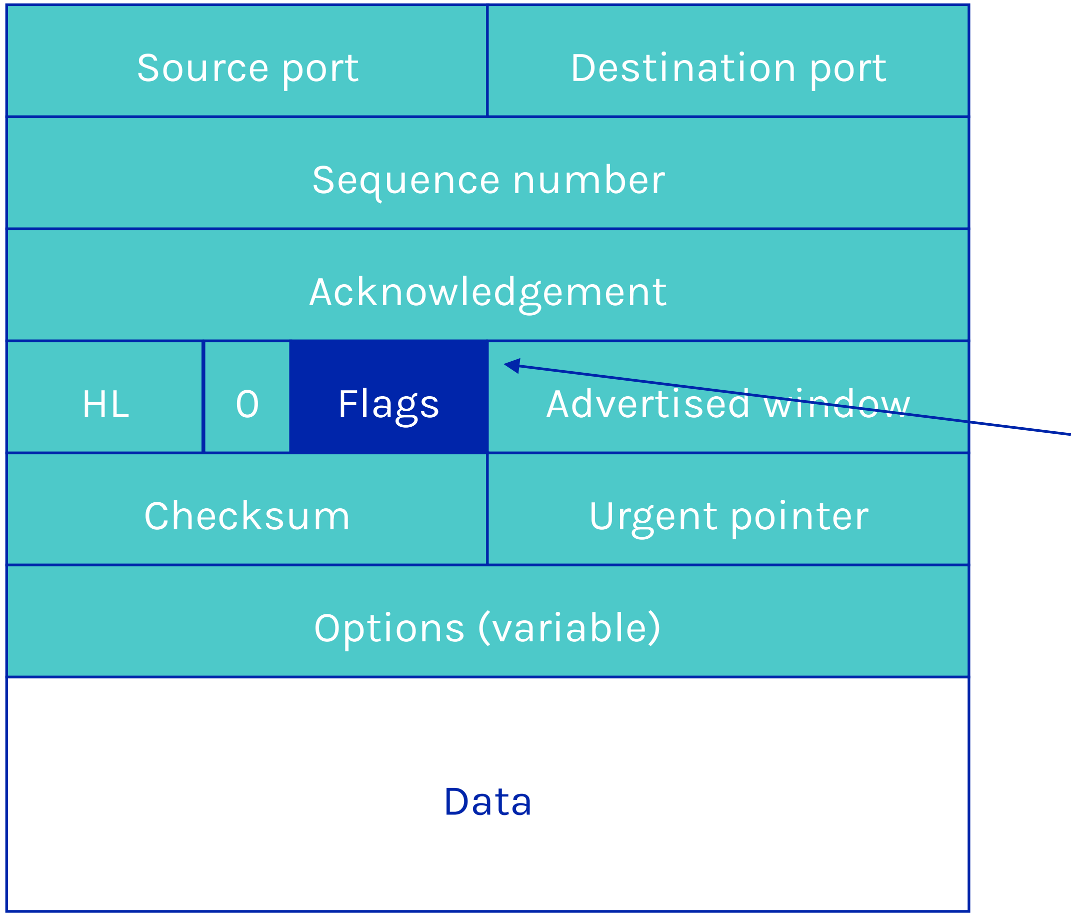


3-way handshake to establish connection

- Host A sends a SYN (open; sychronize seqno)
- Host B returns a SYN acknowledgement (SYN-ACK)
- Host A sends an ACK to acknowledge the SYN-ACK

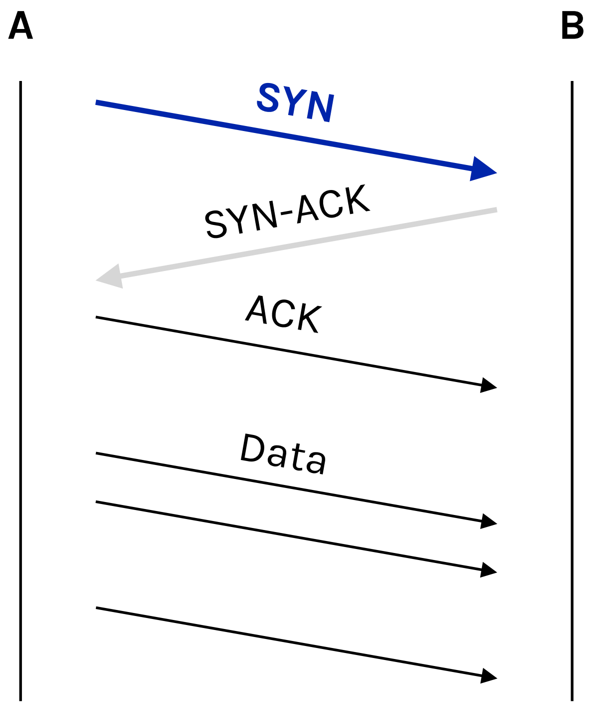
Each host also tells its ISN to the other host

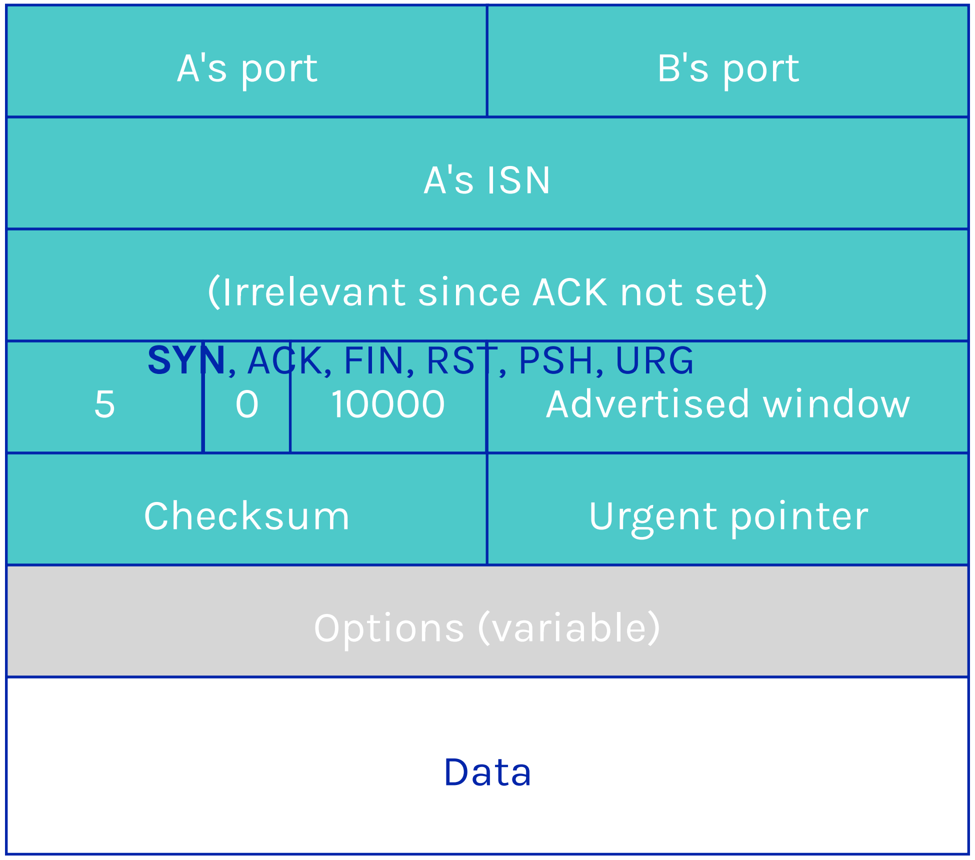
TCP flags



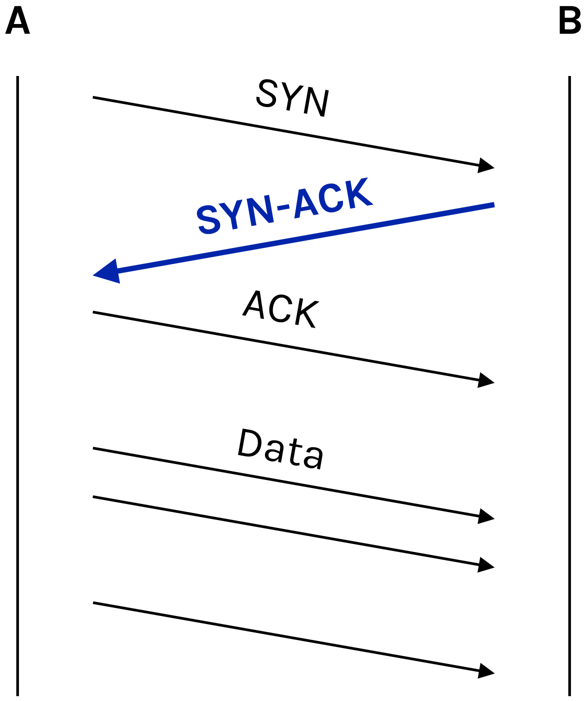
Flags: SYN, ACK, FIN, RST, PSH, URG

TCP connection establishment: SYN





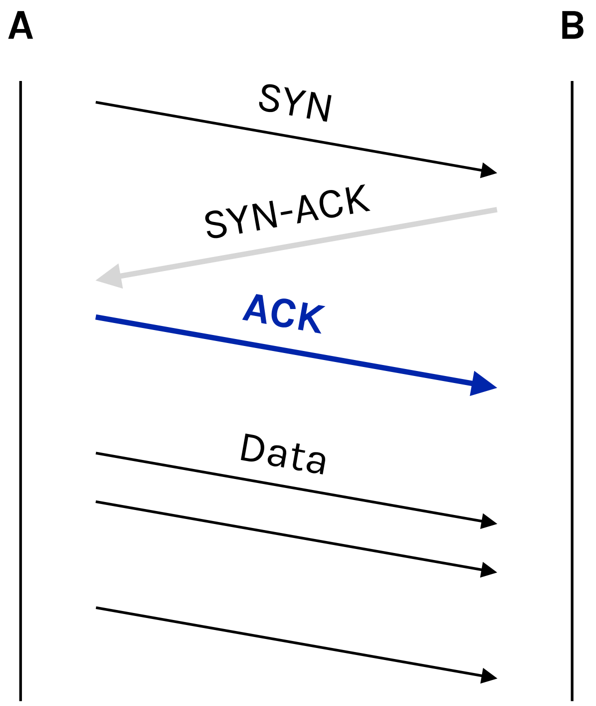
TCP connection establishment: SYN-ACK

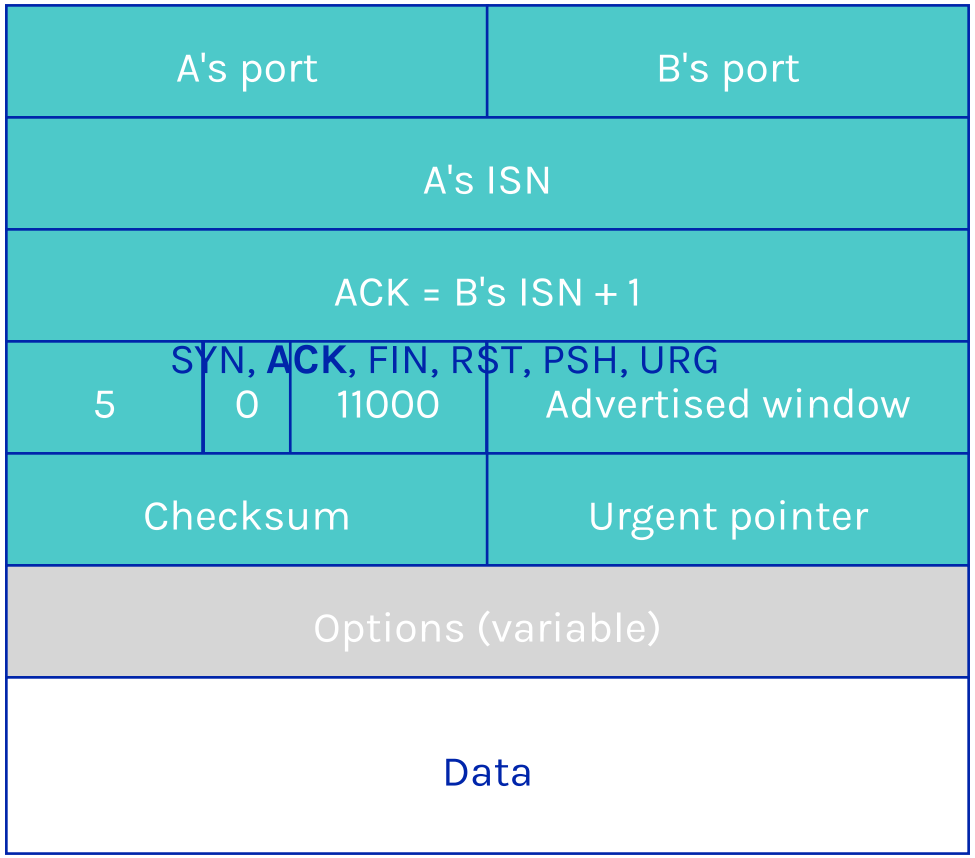


B's port			A's port
B's ISN			
ACK = A's ISN + 1			
5	0	**SYN**, **ACK**, FIN, RST, PSH, URG 11000	Advertised window
Checksum			Urgent pointer
Options (variable)			
Data			

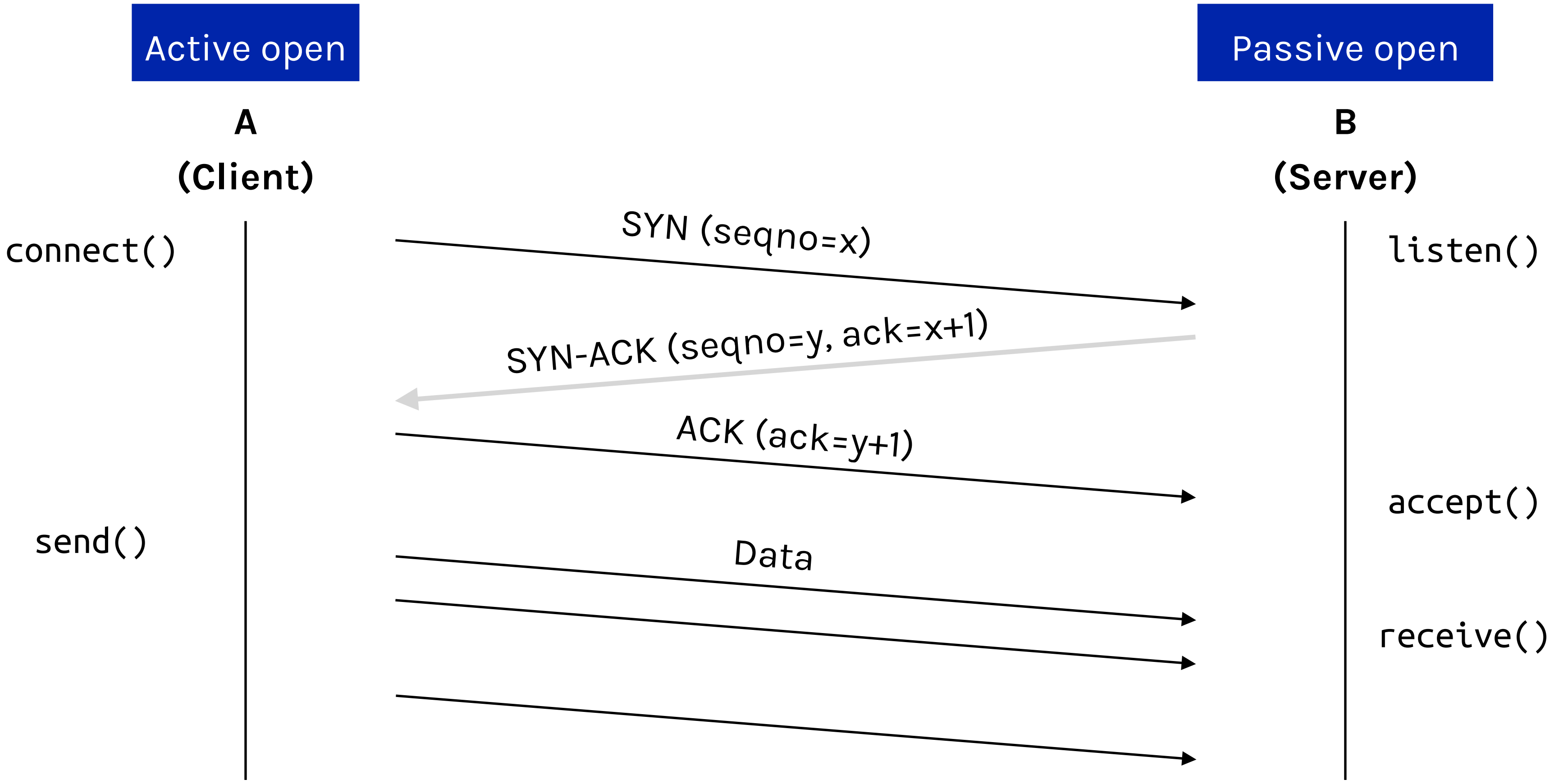
B tells A it accepts, and is ready to hear the next byte; upon receiving this packet, A can start sending data

TCP connection establishment: ACK





3-way handshake



What if SYN gets lost?

Suppose the SYN packet gets lost

- Packet is lost inside the network or server discarded the packet (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, thus hard to guess the time to wait
- SHOULD use default of 3 seconds (RFCs 1122 & 2988)
- Other implementations instead use 6 seconds

SYN loss in web browsing

User clicks on a hypertext link

- Browser creates a socket and calls a "connect"
- The "connect" triggers the OS to transmit a SYN

If the SYN is lost

- 3-6 seconds of delay: too long for impatient users
- User may click the hyperlink again, or click "reload"

User triggers an "abort" of the "connect"

- Browser creates a new socket and another "connect" → a new SYN, and faster!

TCP connection termination: one side at a time

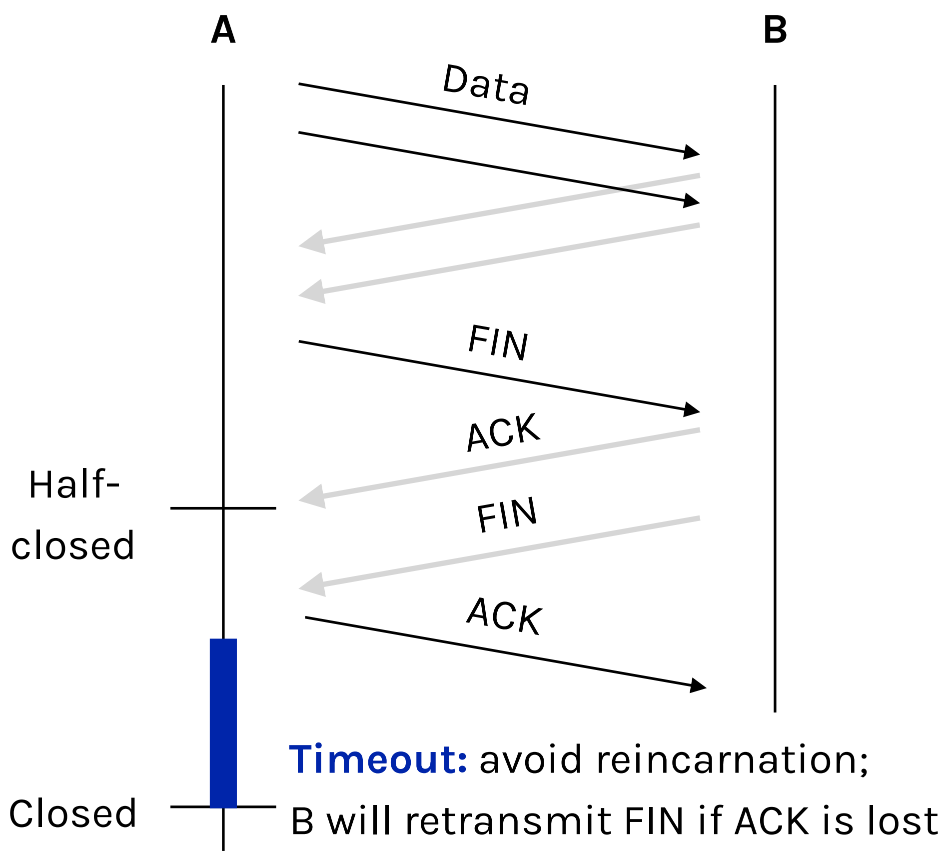
Finish (FIN) to close and receive remaining bytes

- FIN occupies one octet in the sequence space

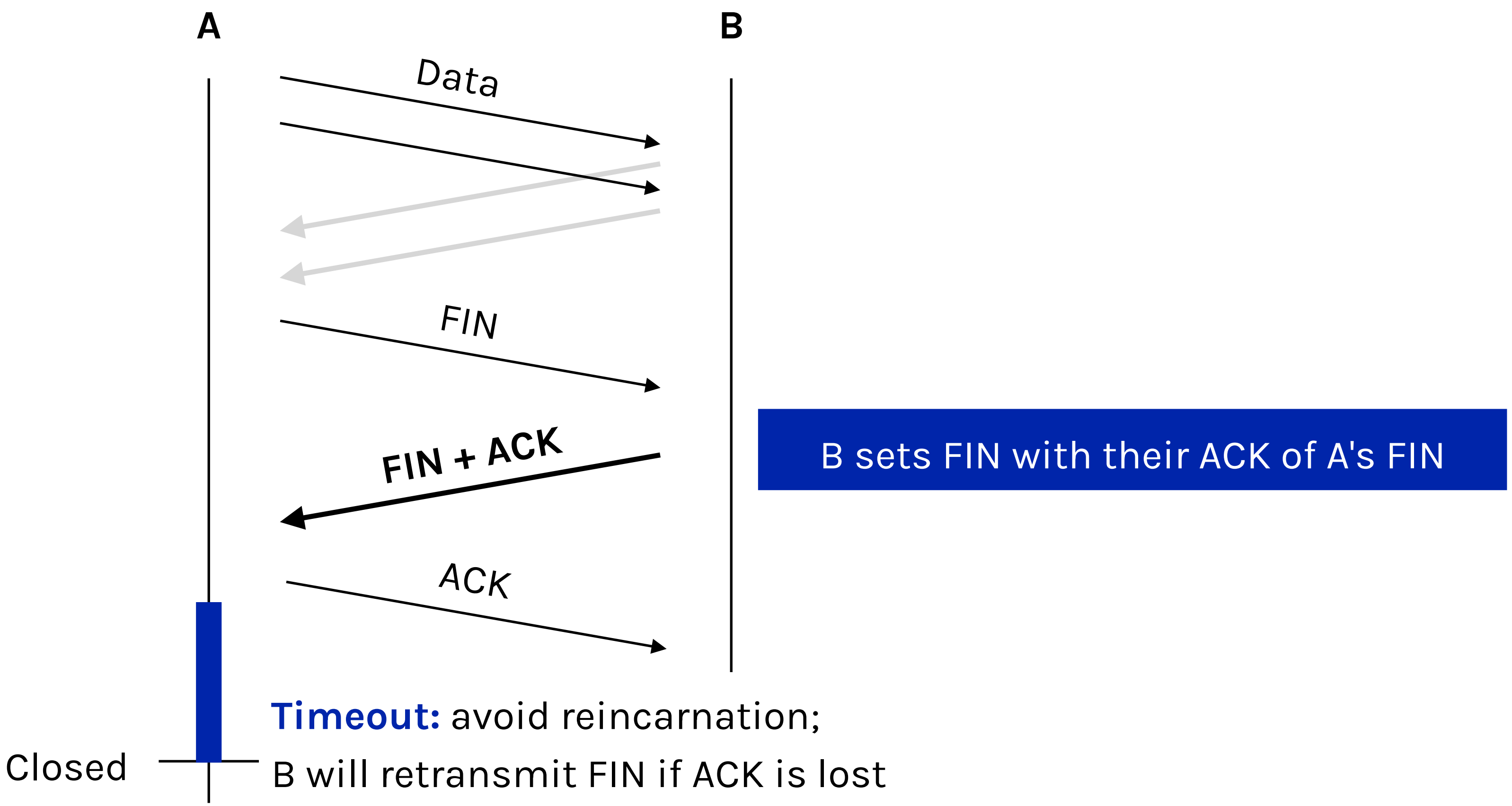
Other host ACK's the octet to confirm

Closes A's side of connection, but not B's side

- Until B likewise sends a FIN
- A ACKs B's FIN



TCP connection termination: both together



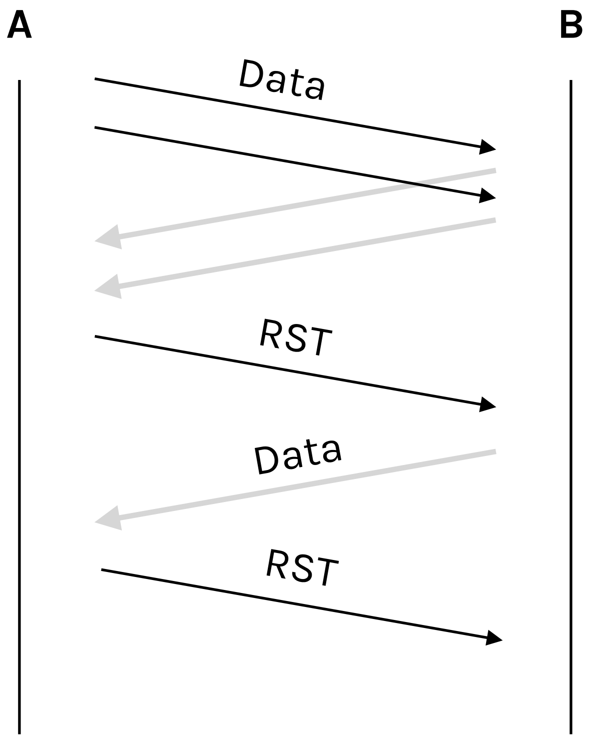
TCP connection termination: abruption

A sends a RESET (RST) to B

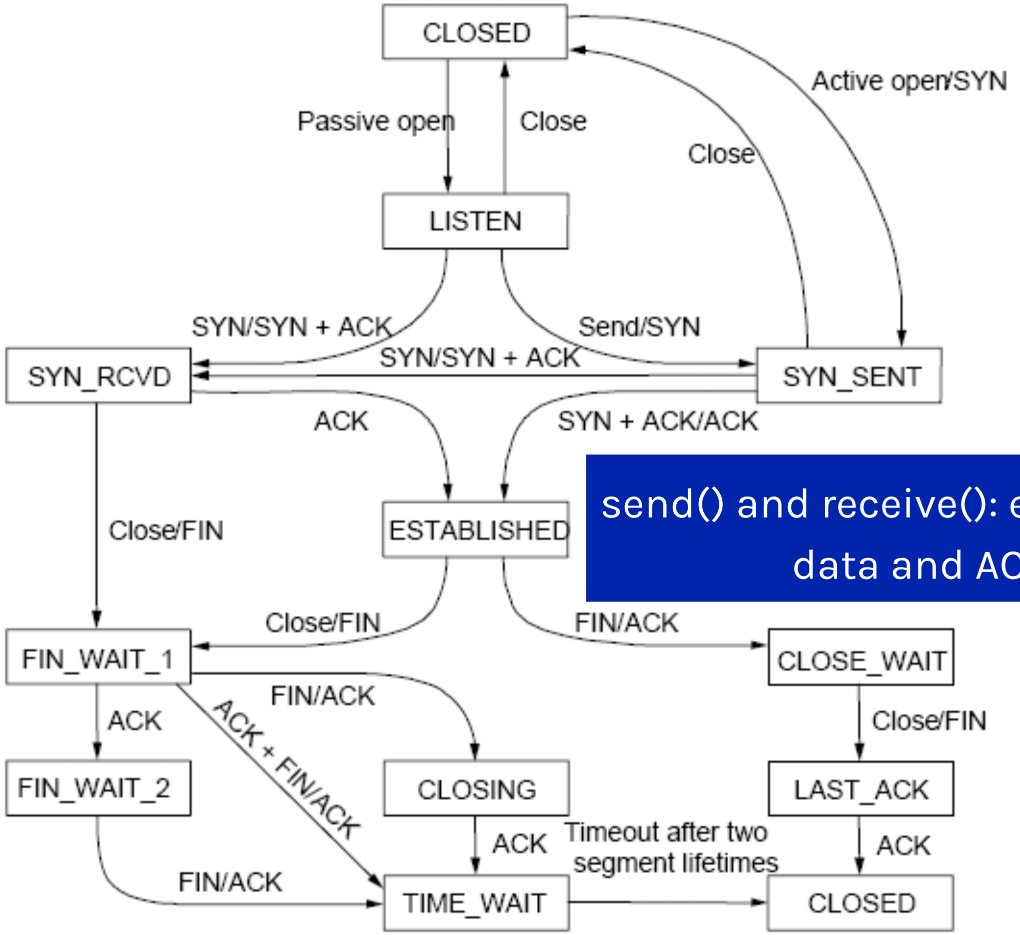
- Example: because application process on A crashed

That is it!

- B does not ACK the RST (so RST is not relivered reliably)
- Any data in flight is lost
- If B sends anything more, A will elicit another RST



TCP state machine



send() and receive(): exchange of data and ACK

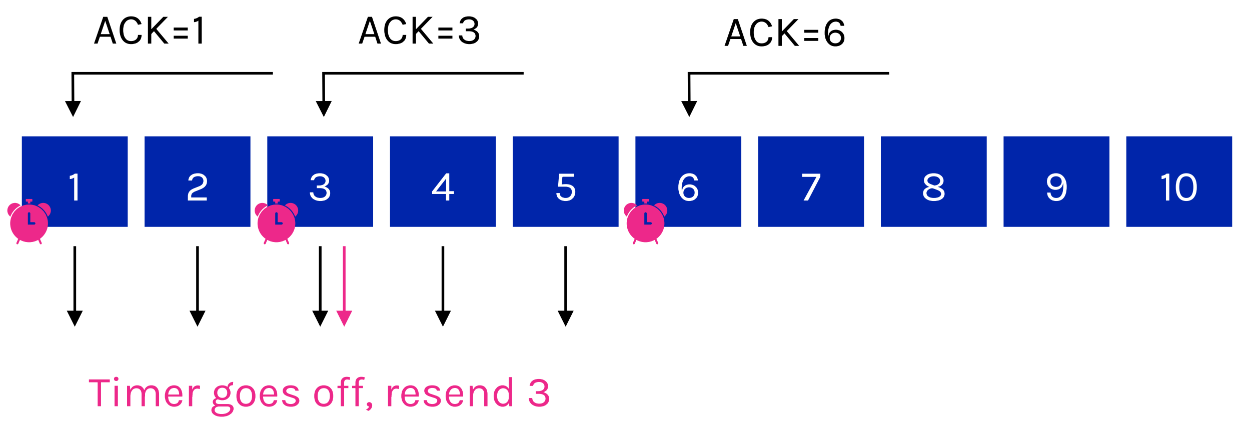
TCP timeouts and retransmission

Reliability requires retransmitting lost data

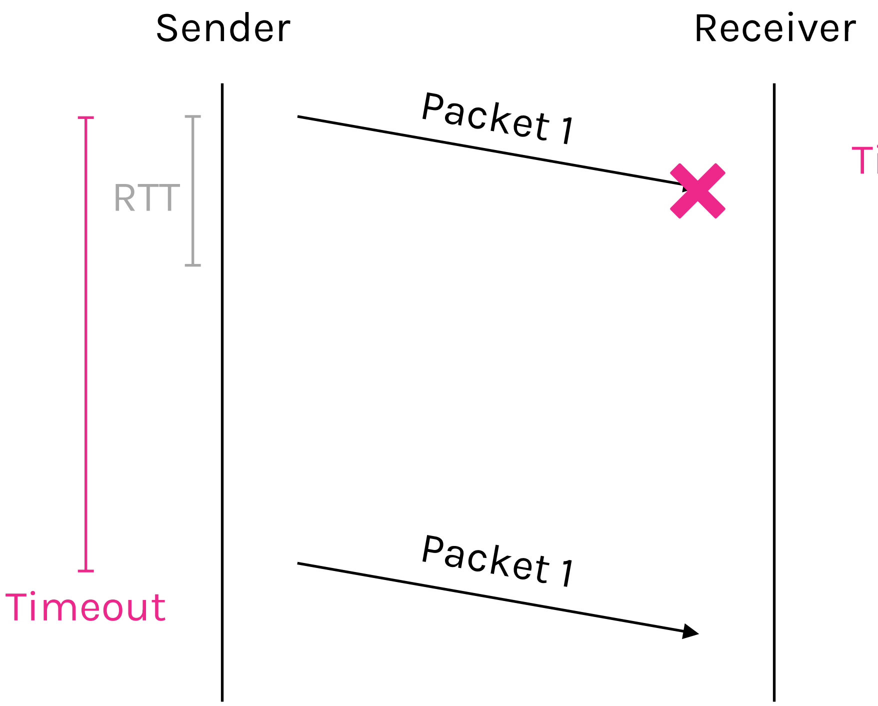
Involves setting timer and retransmitting on timeout

TCP resets timer whenever new data is ACKed

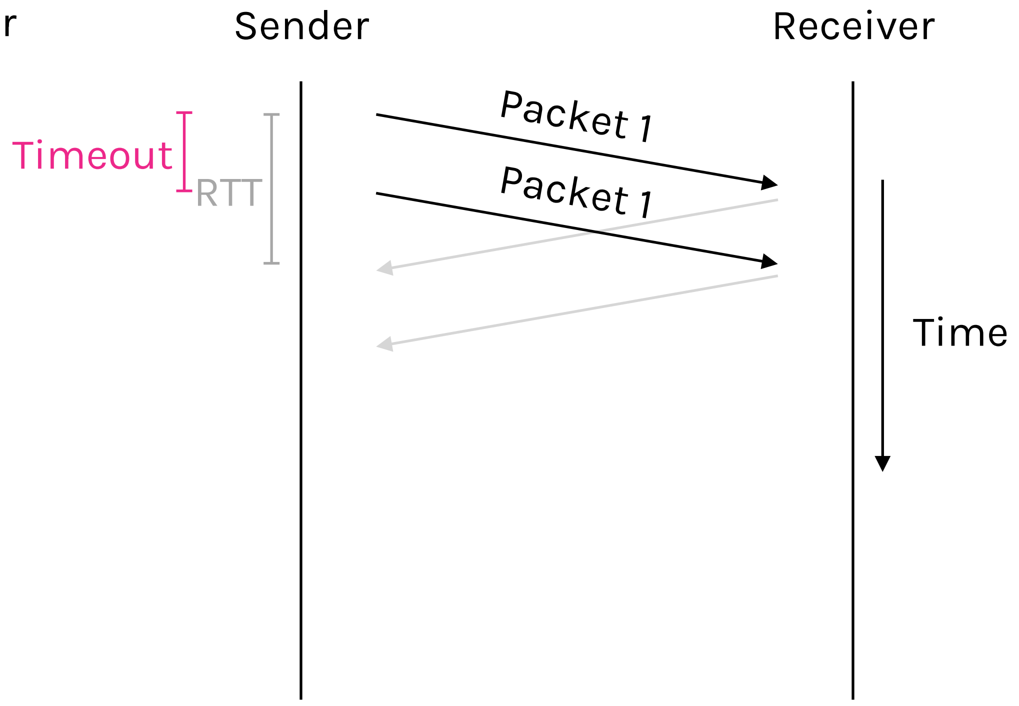
- Retransmission of packet containing "next byte" when timer goes off



Setting TCP timeout



Timeout too long → inefficient



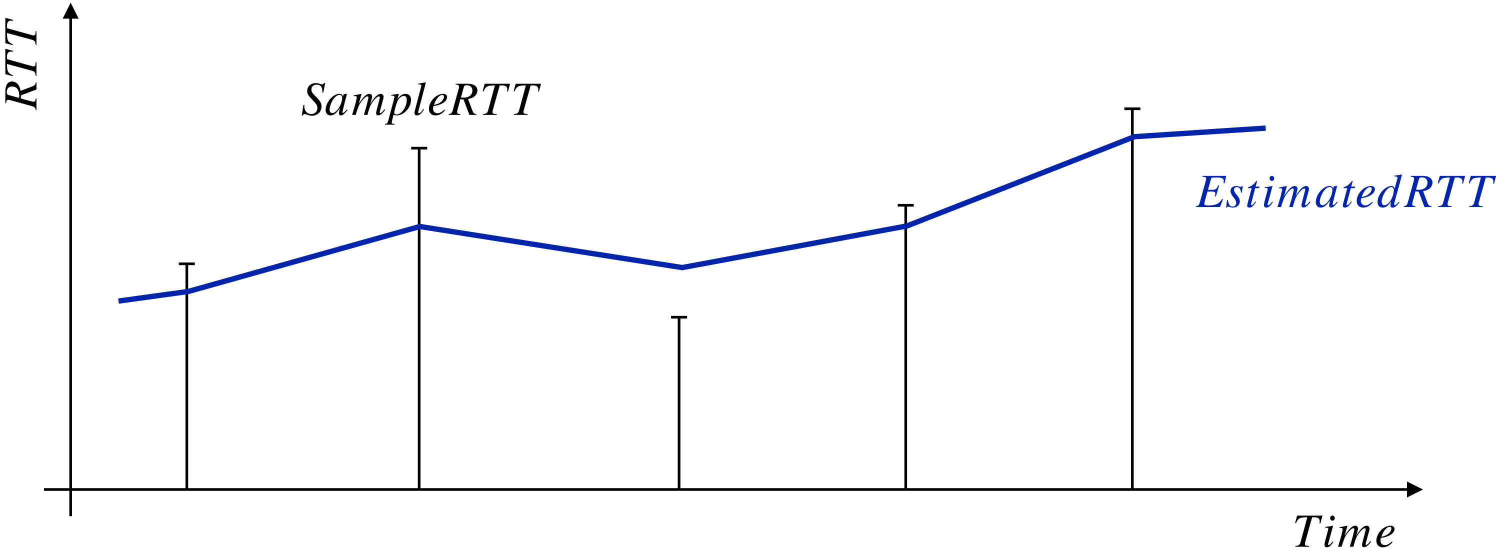
Timeout too short → duplicate packets

RTT estimation

Exponential averaging of RTT samples

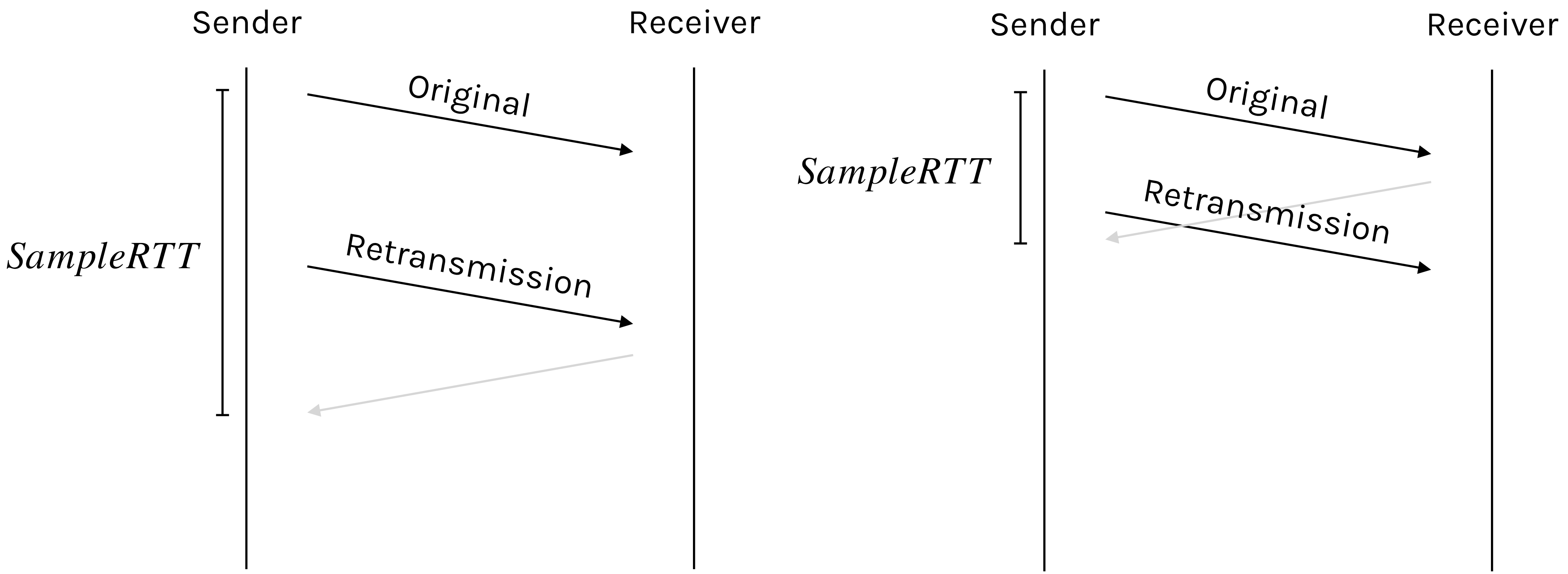
$$SampleRTT = ACKRcvTime - SendPktTime$$

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT$$



Problem: ambiguous measurements

How to differentiate between the real ACK and ACK of the retransmitted packet?



Karn/Patridge algorithm

Measure $SampleRTT$ only for original transmissions

- Once a segment has been retransmitted, do not use it for any further measurements
- Computes $EstimatedRTT$ using $\alpha = 0.875$

Timeout value (RTO) = $2 \times EstimatedRTT$

Use exponential backoff for repeated retransmissions

- Every time RTO timer expires, set $RTO \leftarrow 2 \cdot RTO$ (up to max. 60 seconds)
- Every time new measurement comes in (i.e., successful original transmission), collapse RTO back to $2 \times EstimatedRTT$

However, in practice...

Implementations often use a coarse-grained timer

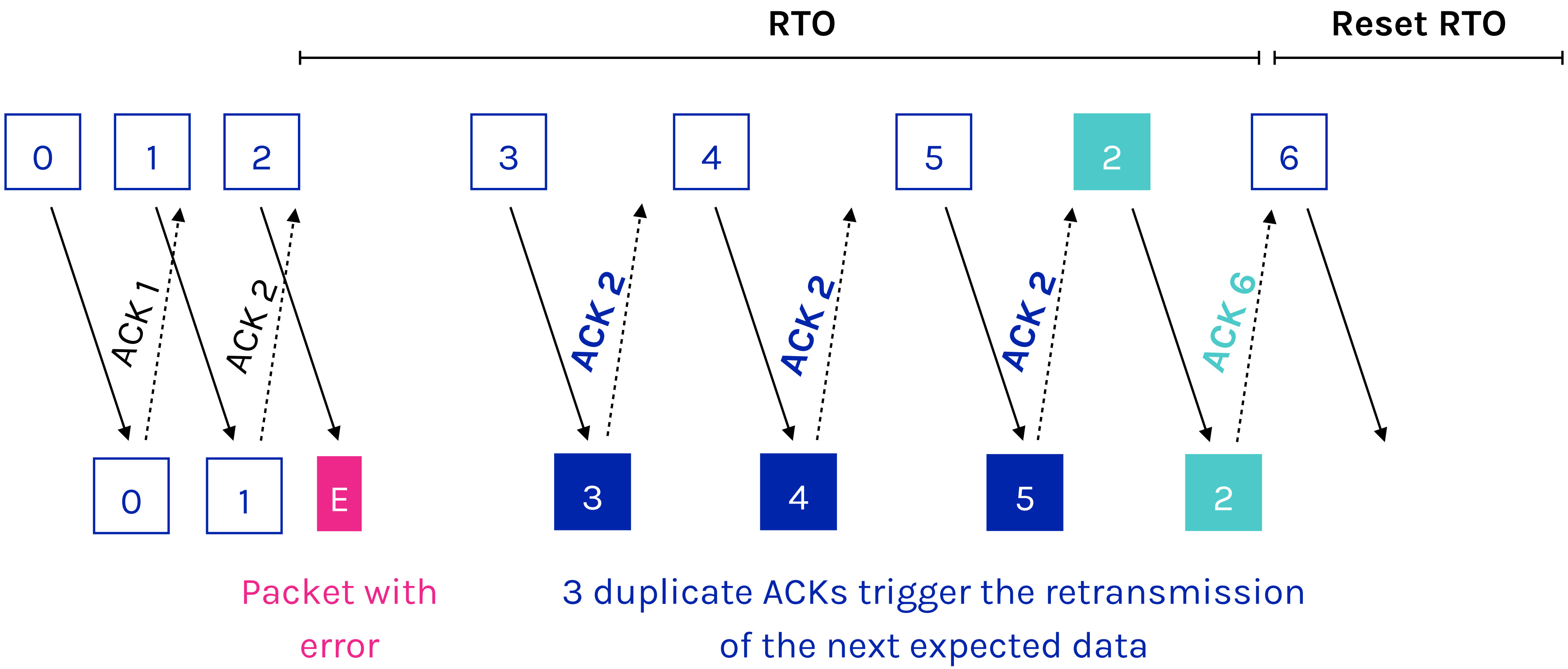
- 500 milliseconds is quite typical

So what?

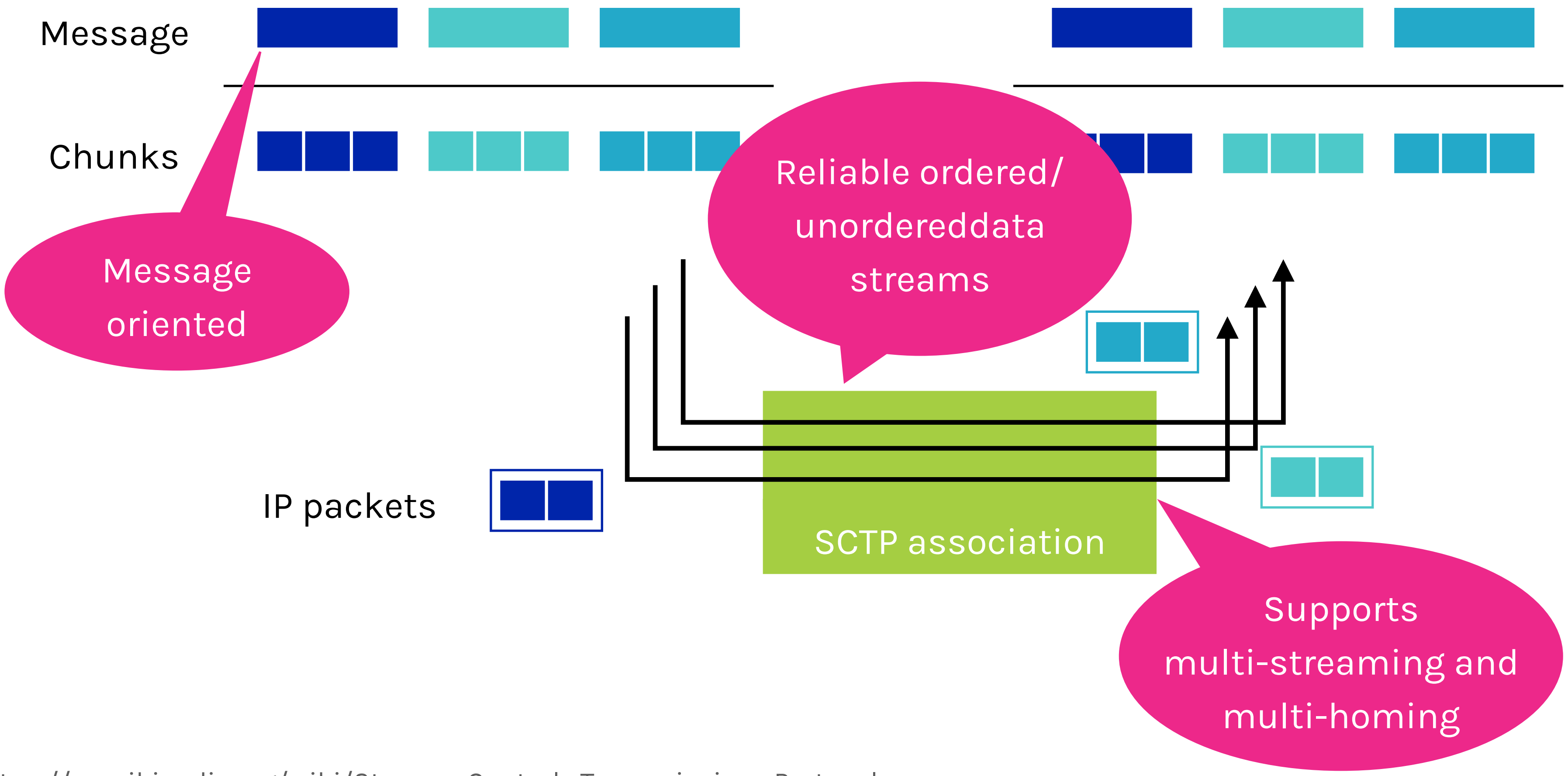
- Above algorithms are largely irrelevant
- Incurring a timeout is expensive

So we rely on duplicate ACKs to detect loss/corruption early on

Loss detection with cumulative ACKs



Stream Control Transmission Protocol (SCTP)

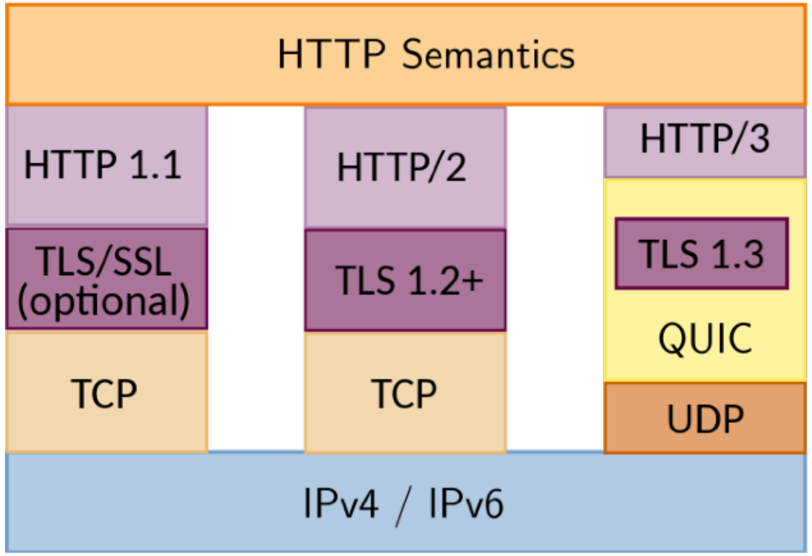


https://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol

QUIC

History

- Experimental protocol deployed at Google since 2013
- Between Google services and Chrome
- Akamai deployment in 2016, Facebook deployment in 2020
- HTTP/3 standardization based on QUIC (RFC 9114) in 2022



Langley et al. The QUIC Transport Protocol: Design and Internet-Scale Deployment. ACM SIGCOMM, 2017.
https://www.rfc-editor.org/rfc/rfc9114.html

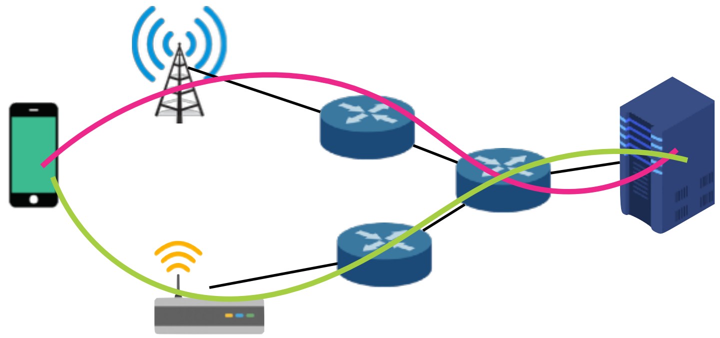
MPTCP

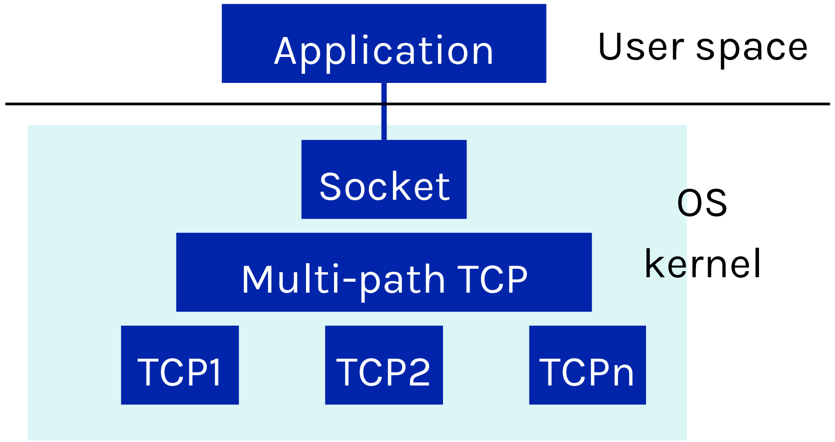
Multi-homed devices become popular

- Mobile devices (with cellular and WiFi at the same time)
- High-end servers (multiple NICs)
- Data centers (rich connectivity with many paths)

Benefits of multi-path

- Higher **throughput**, **failover** from one path to another
- Seamless **mobility**





https://www.multipath-tcp.org/

Summary

Requirements

- Demultiplexing (sockets and ports)
- Byte stream / message
- Reliability
- Flow control
- Congestion control

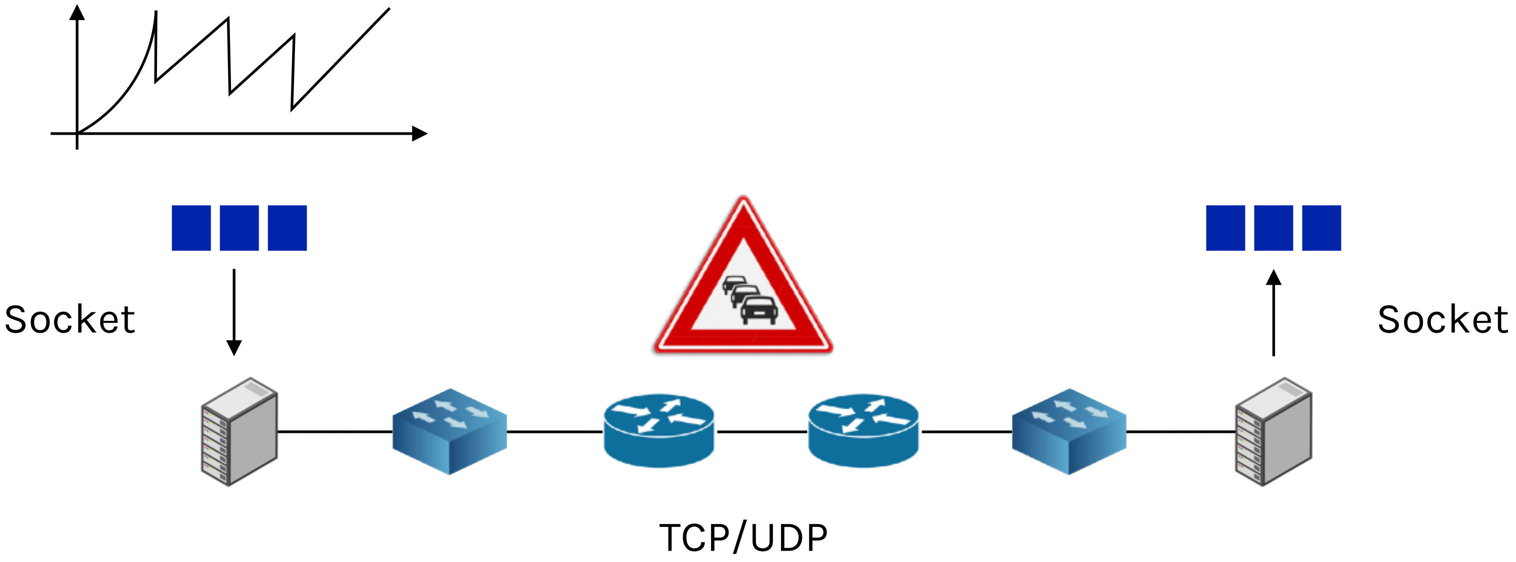
UDP

- Light-weight, low-overhead

TCP

- TCP segmentation
- Sequence and ACK number
- Sliding window flow control
- Connection establishment
- Connection termination
- Timeouts and retransmission
- TCP alternatives: SCTP, QUIC, MPTCP

Next time: transport layer



How fast should the data be sent out?

Further reading material

Andrew S. Tanenbaum, David J. Wetherall. Computer Networks (5th edition).

- Section 6.4: The Internet Transport Protocols: UDP
- Section 6.5: The Internet Transport Protocols: TCP

Larry Peterson, Bruce Davie. Computer Networks: A Systems Approach.

- Section 5.1: Simple Demultiplexor (UDP)
- Section 5.2: Reliable Byte Stream (TCP)