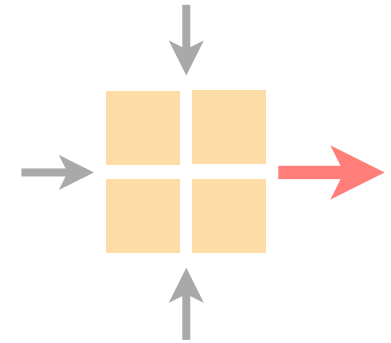# Communication Networks

Spring 2019

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

April 8 2019

Materials inspired from Scott Shenker & Jennifer Rexford

Last week on

Communication Networks

Border Gateway Protocol
policies and more

Building Reliable Networks with the Border Gateway Protocol

BGP

O'REILLY

Iljitsch van Beijnum

1   **BGP Policies**

Follow the Money

2   **Protocol**

How does it work?

3   **Problems**

security, performance, …

Border Gateway Protocol
policies and more

Building Reliable Networks with the Border Gateway Protocol

BGP

O'REILLY

Iljitsch van Beijnum

1   **BGP Policies**
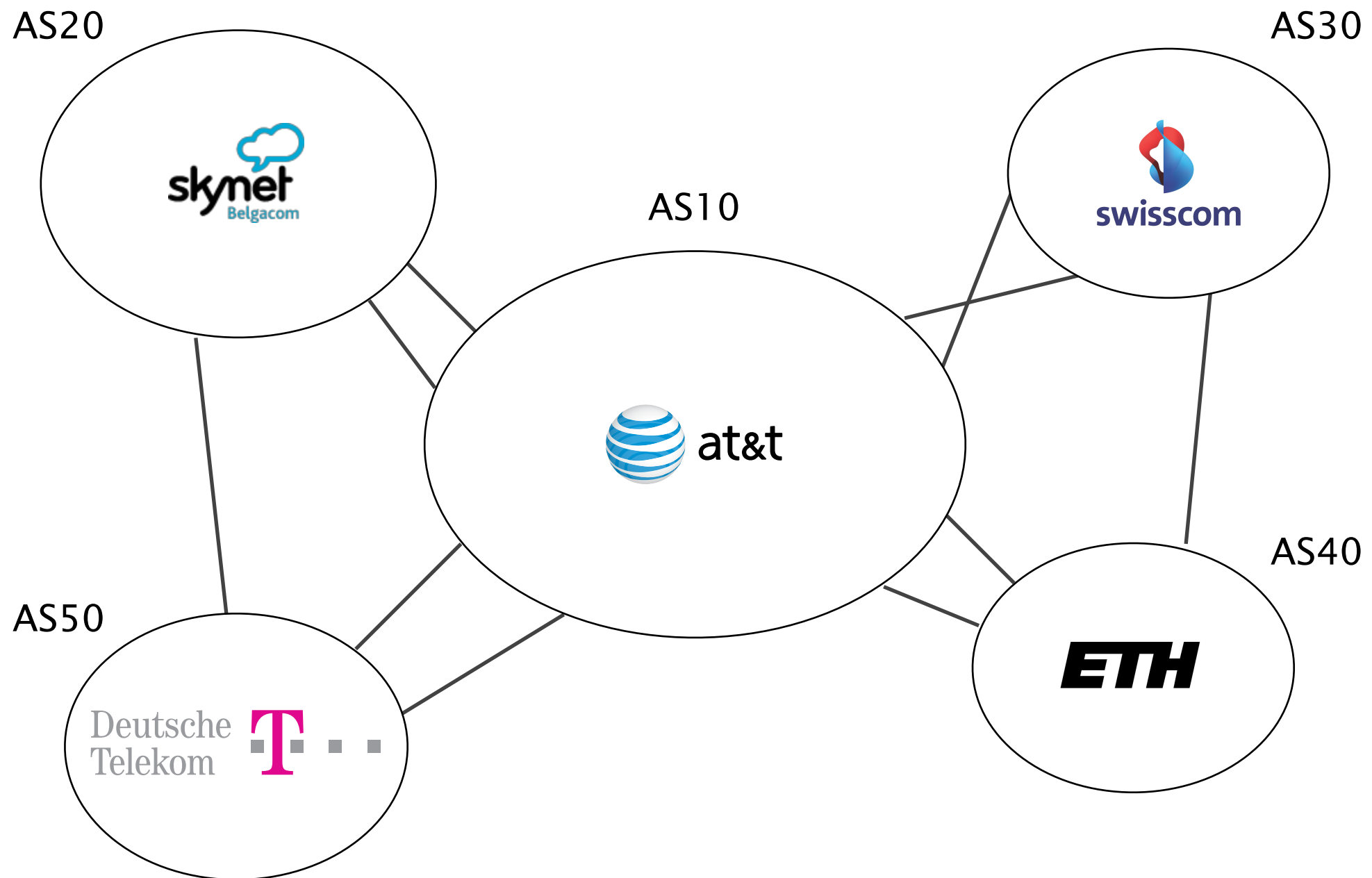
Follow the Money

Protocol

How does it work?

Problems

security, performance, …

The Internet topology is shaped according to *business* relationships

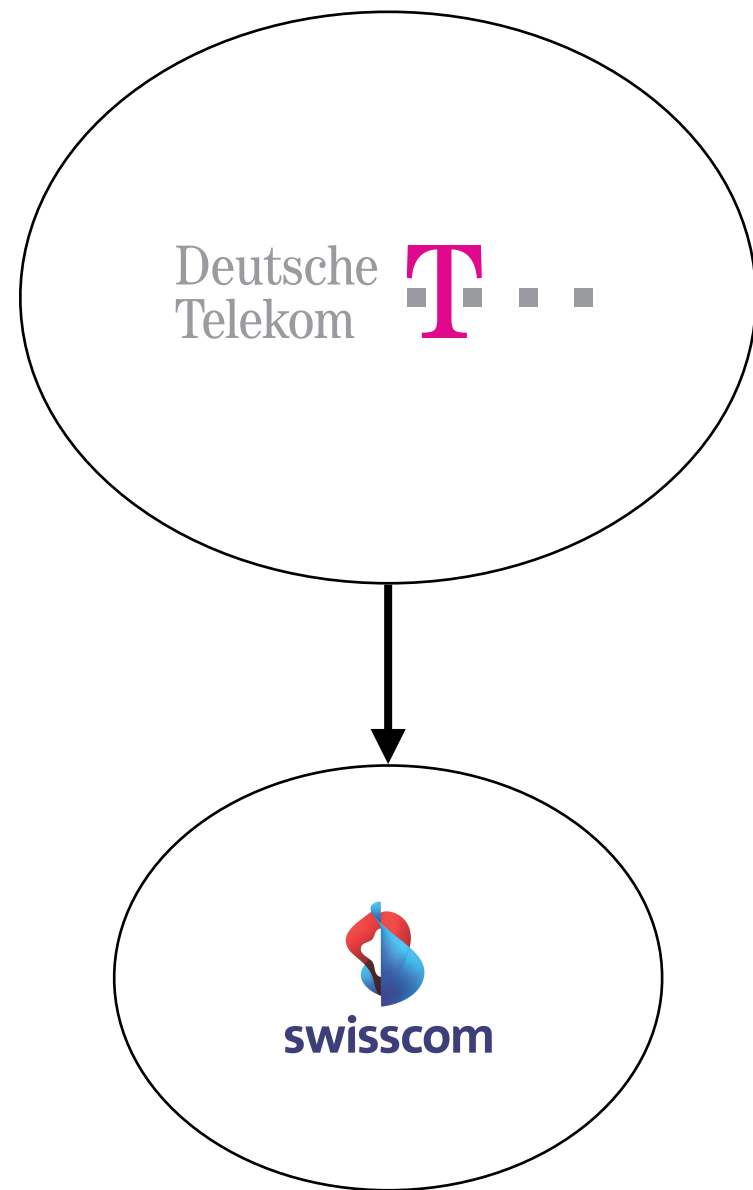There are 2 main business relationships today:

- customer/provider

- peer/peer

many less important ones (siblings, backups,…)

There are 2 main business relationships today:

- customer/provider
- peer/peer

Customers pay providers
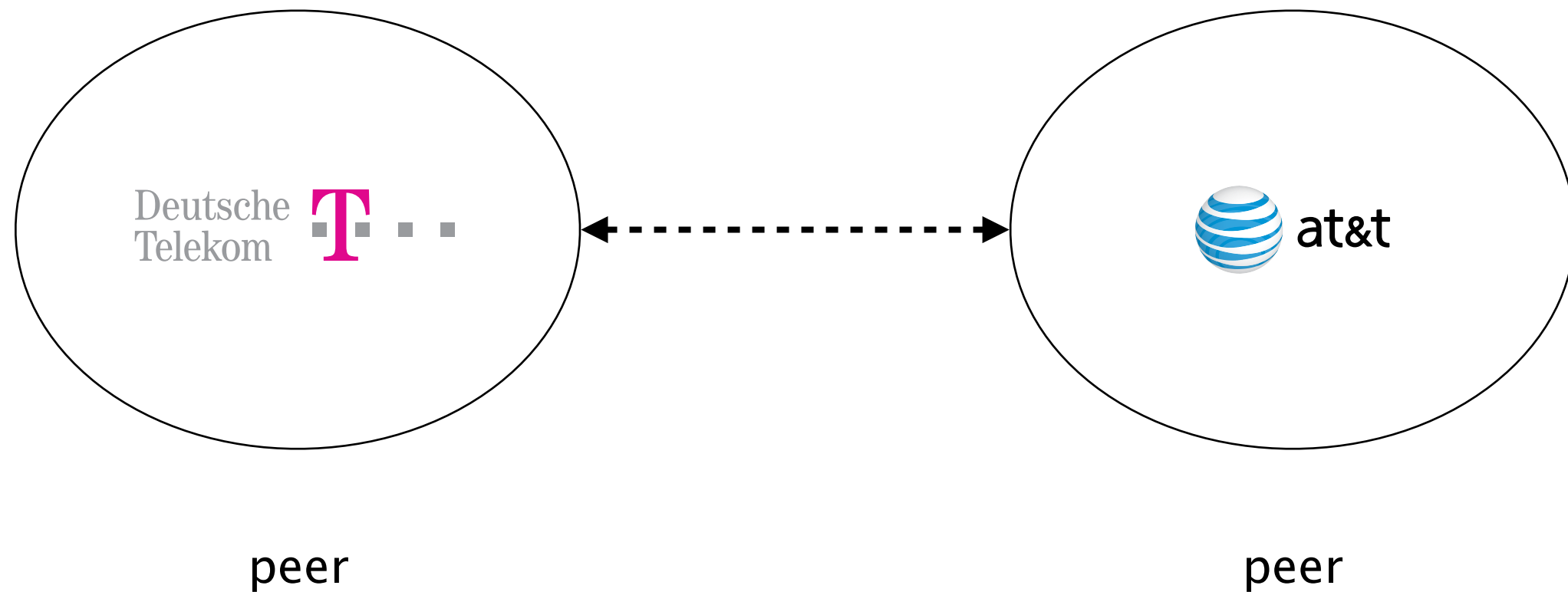## to get Internet connectivity

Deutsche Telekom

swisscom

provider

$$$

customer

Peers don't pay each other for connectivity, they do it *out of common interest*

peer   peer

DT and ATT exchange *tons* of traffic.
they save money by directly connecting to each other

Business relationships conditions
route selection

For a destination p, **prefer routes coming from**

- **customers** over

- **peers** over

- **providers**

route type

Business relationships conditions
route exportation

send to

	customer	peer	provider
from customer			
peer			
provider			

Routes coming from customers
are propagated to everyone else

send to

	customer	peer	provider
customer	✓	✓	✓
from peer			
provider			

Routes coming from peers and providers are only propagated to customers

send to

	customer	peer	provider
customer	✓	✓	✓
from peer	✓	-	-
provider	✓	-	-

Border Gateway Protocol
policies and more

Building Reliable Networks with the Border Gateway Protocol

BGP

O'REILLY

Iljitsch van Beijnum
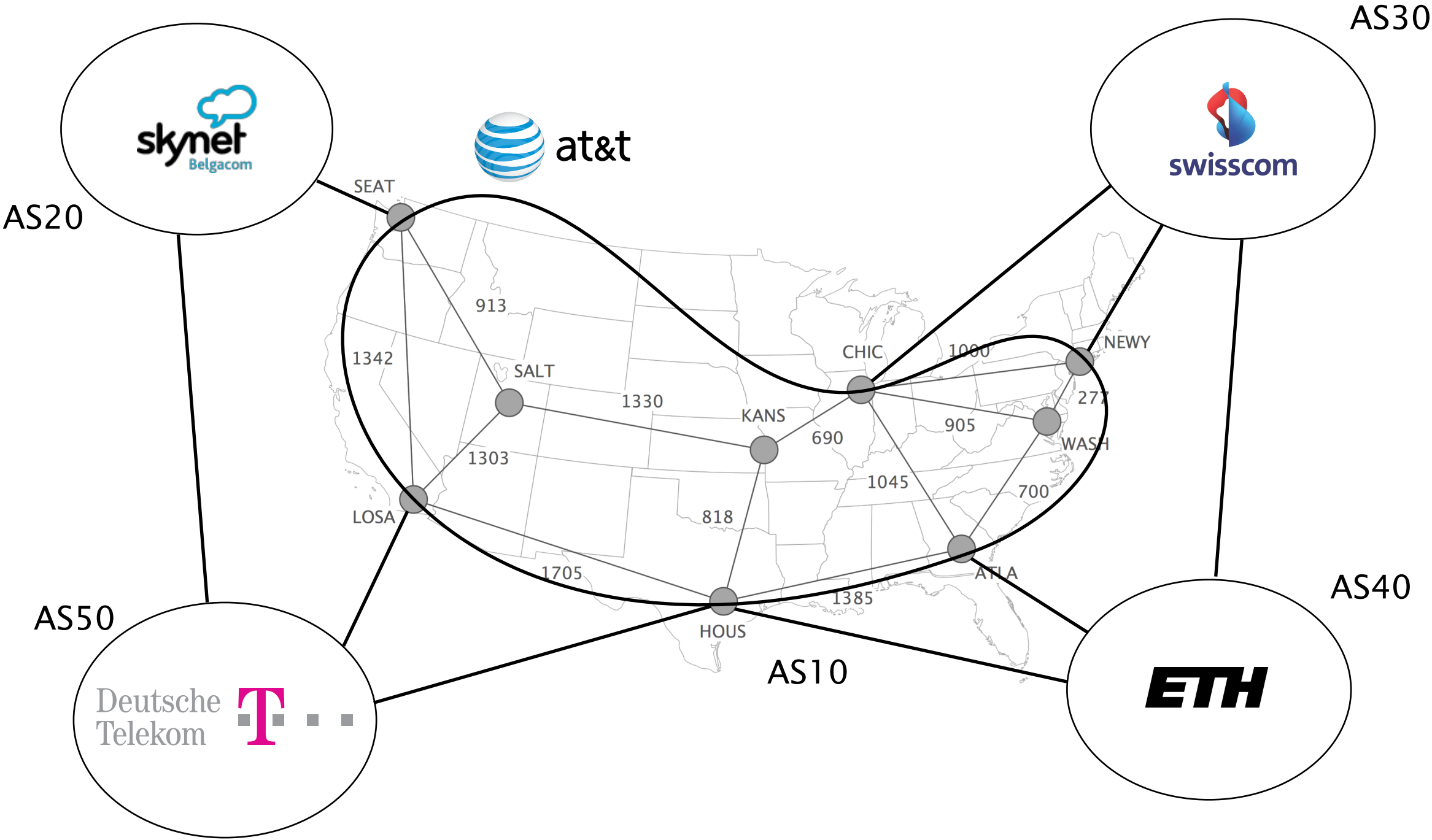
BGP Policies

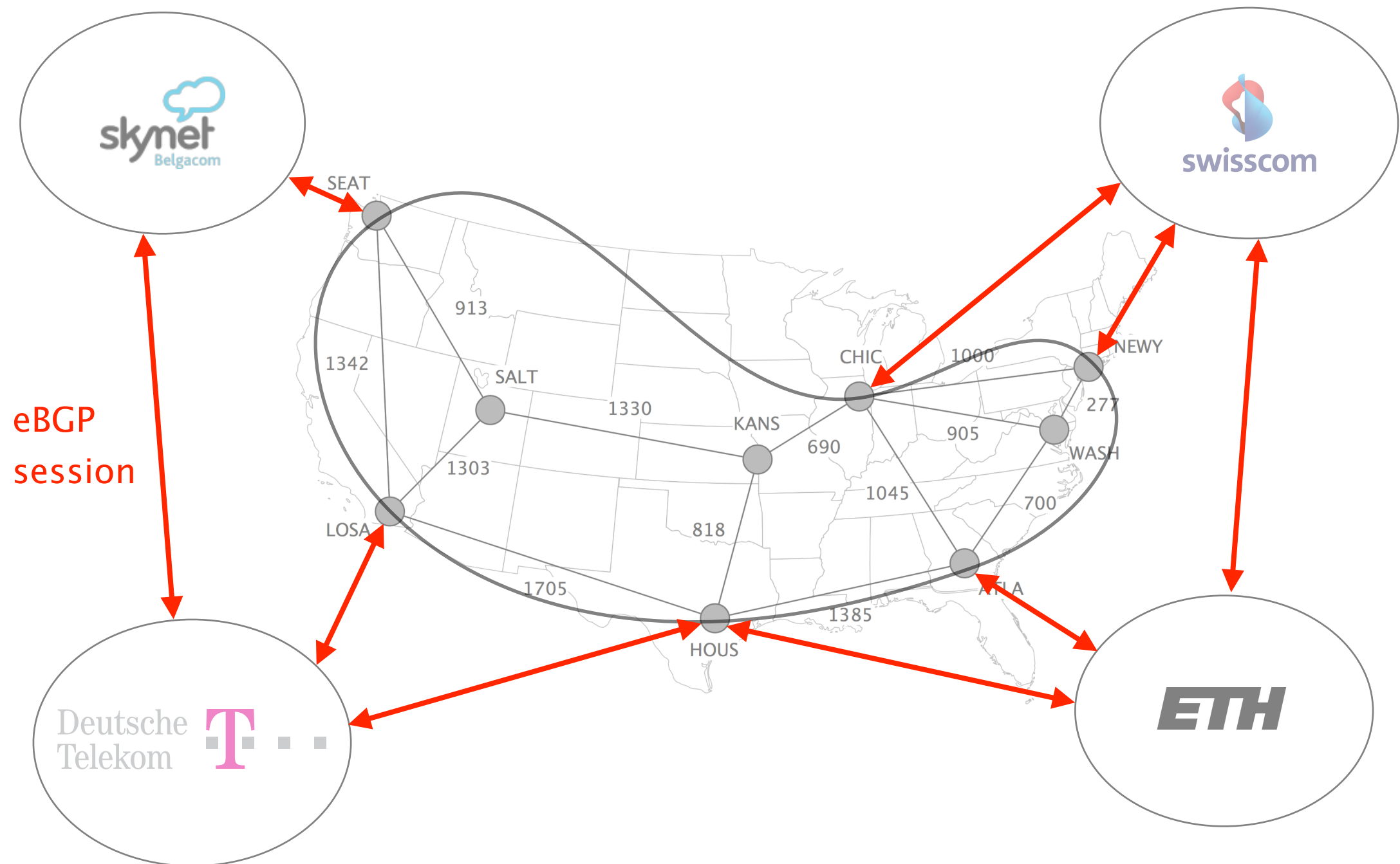Follow the Money

2   Protocol

How does it work?

Problems
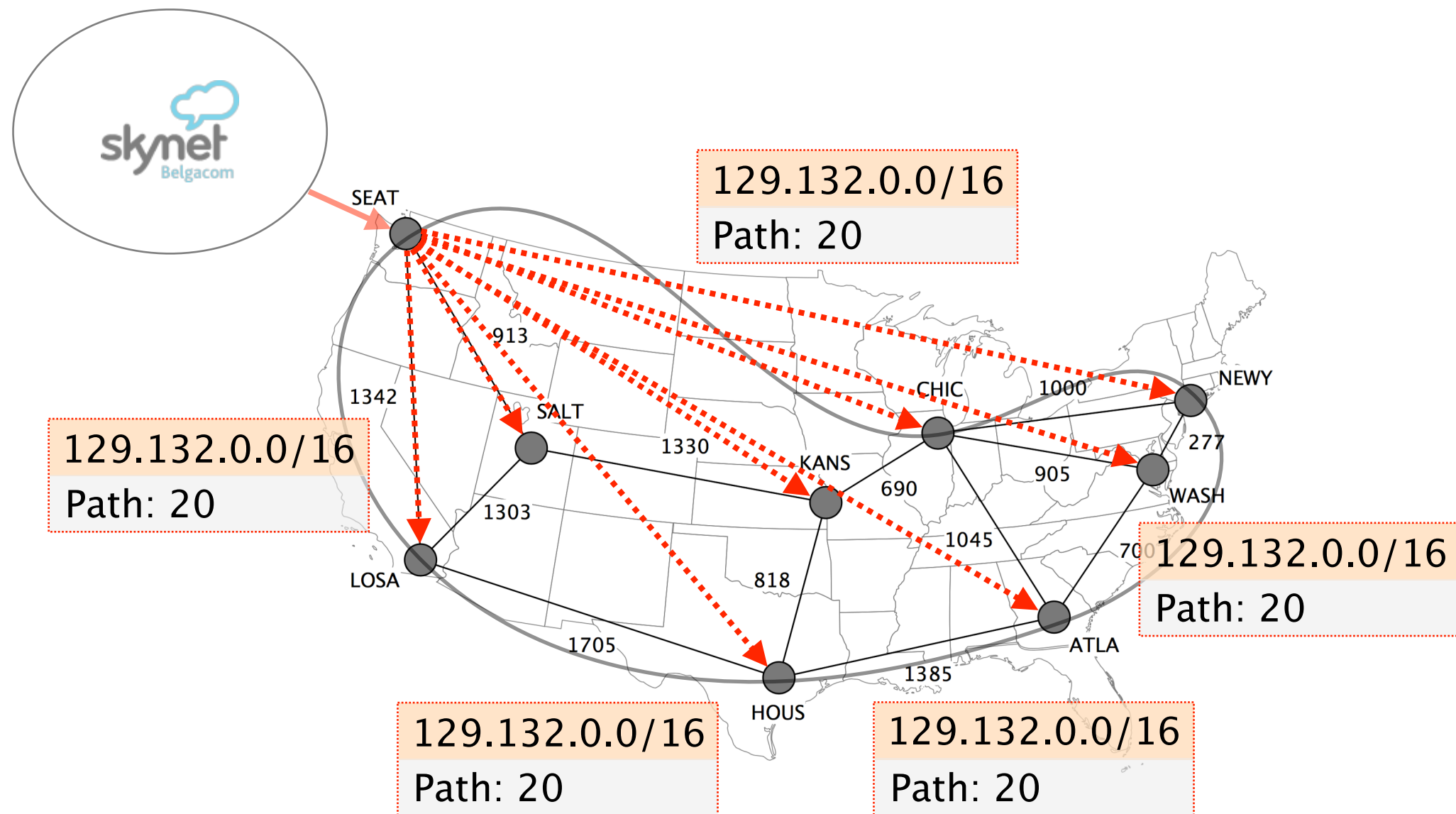
security, performance, …

BGP sessions come in two flavors

external BGP (eBGP) sessions
connect border routers in different ASes

eBGP
session

iBGP sessions are used to disseminate externally-learned routes internally

129.132.0.0/16
Path: 20

129.132.0.0/16
Path: 20

129.132.0.0/16
Path: 20

129.132.0.0/16
Path: 20

129.132.0.0/16
Path: 20

913
1342
1330
1303
1705
818
690
905
1045
1000
277
700
1385

SEAT
SALT
LOSA
KANS
CHIC
HOUS
ATLA
WASH
NEWY

BGP UPDATEs carry an IP prefix together with a set of attributes

IP prefix

Attributes —— Describe route properties

used in route selection/exportation decisions

are either local   (*only* seen on iBGP)

or **global**   (seen on iBGP *and* eBGP)

Attributes	Usage
NEXT-HOP	egress point identification
AS-PATH	loop avoidance
	outbound traffic control
	inbound traffic control
LOCAL-PREF	outbound traffic control
MED	inbound traffic control

Prefer routes…

with higher LOCAL-PREF

with shorter AS-PATH length

with lower MED

learned via eBGP instead of iBGP

with lower IGP metric to the next-hop

with smaller egress IP address (tie-break)

Border Gateway Protocol
policies and more

Building Reliable Networks with the Border Gateway Protocol

BGP

O'REILLY   Iljitsch van Beijnum

BGP Policies

Follow the Money
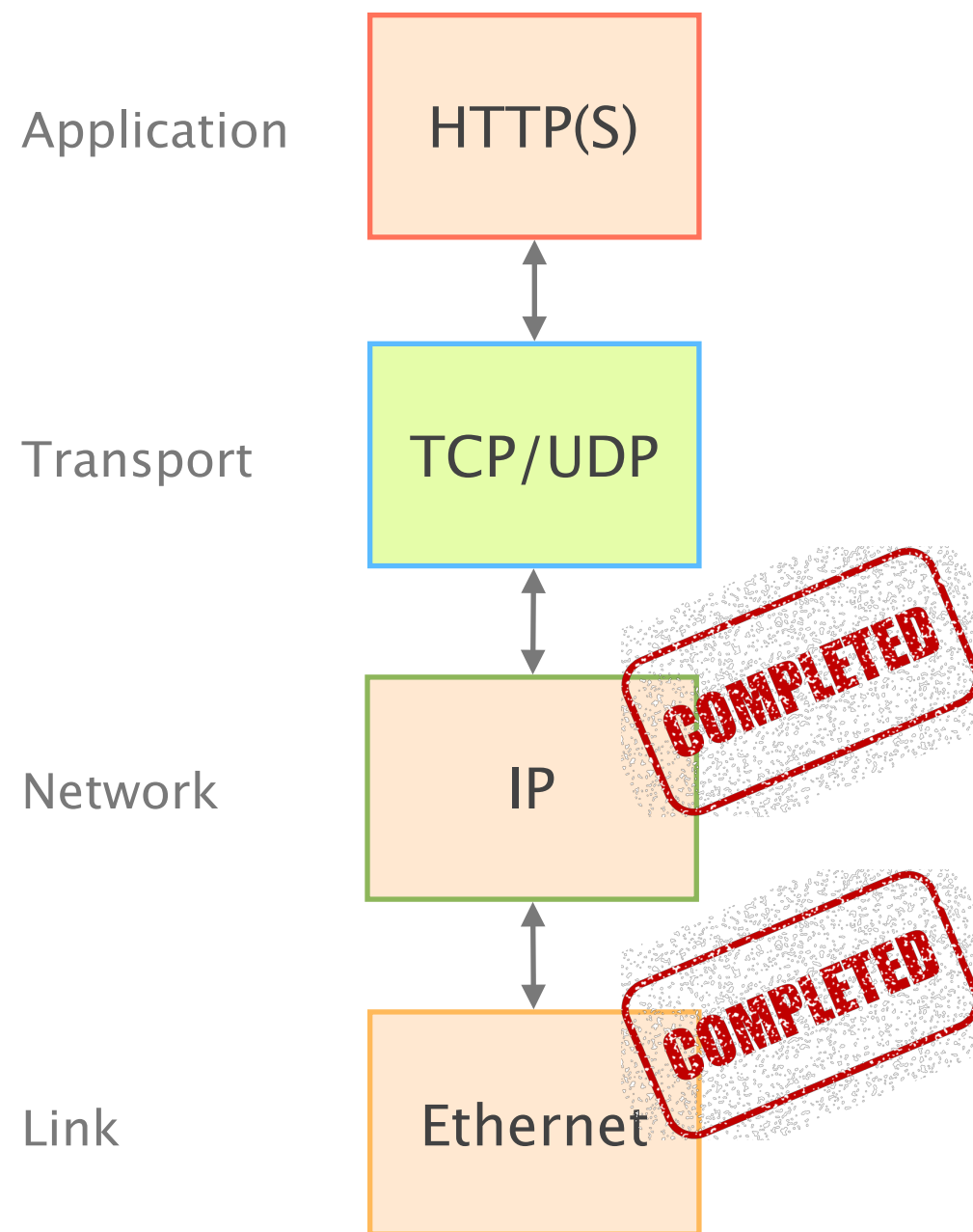
Protocol

How does it work?

3   **Problems**

security, performance, …

This week on

Communication Networks

We're continuing our journey up the layers, now looking at the transport layer

Application   HTTP(S)

Transport   TCP/UDP

Network   IP   COMPLETED

Link   Ethernet   COMPLETED

What do we need in the Transport layer?

Functionality implemented in network

- Keep minimal (easy to build, broadly applicable)

Functionality implemented in the application

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

Functionality implemented in the "network stack"

- The shared networking code on the host
- This relieves burden from both app and network
- The transport layer is a key component here

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
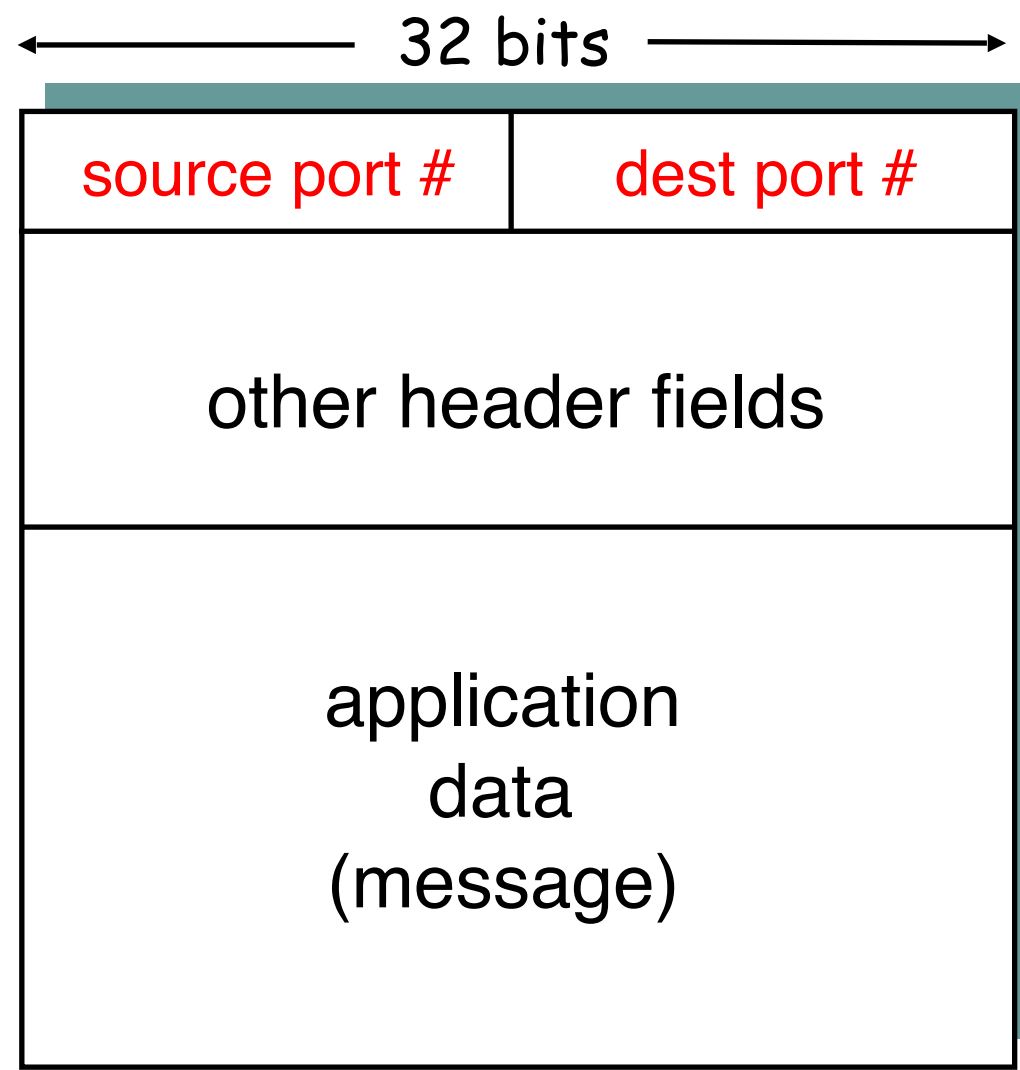
Ephemeral ports (most 1024-65535):
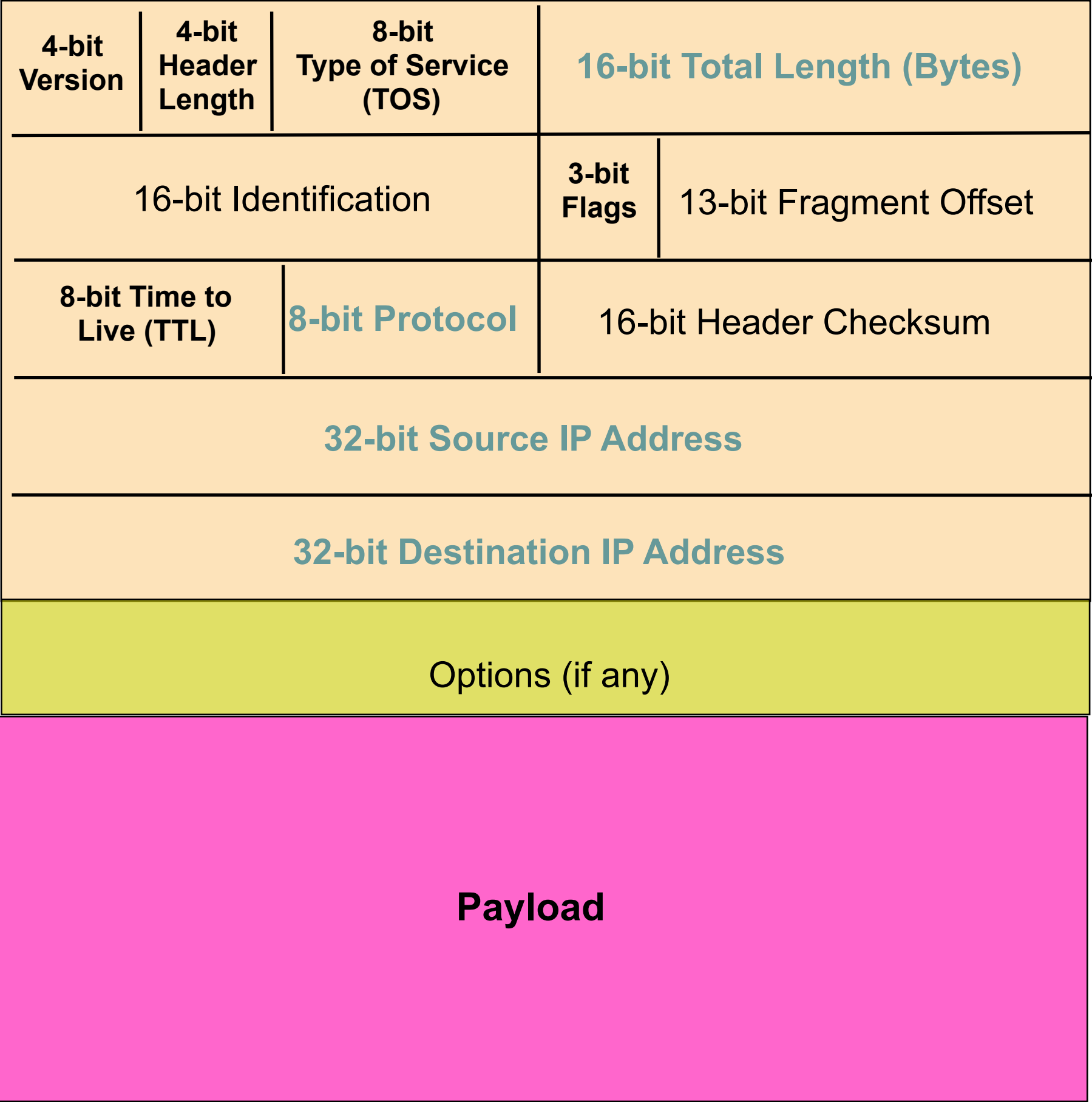
- Given to clients (at random)
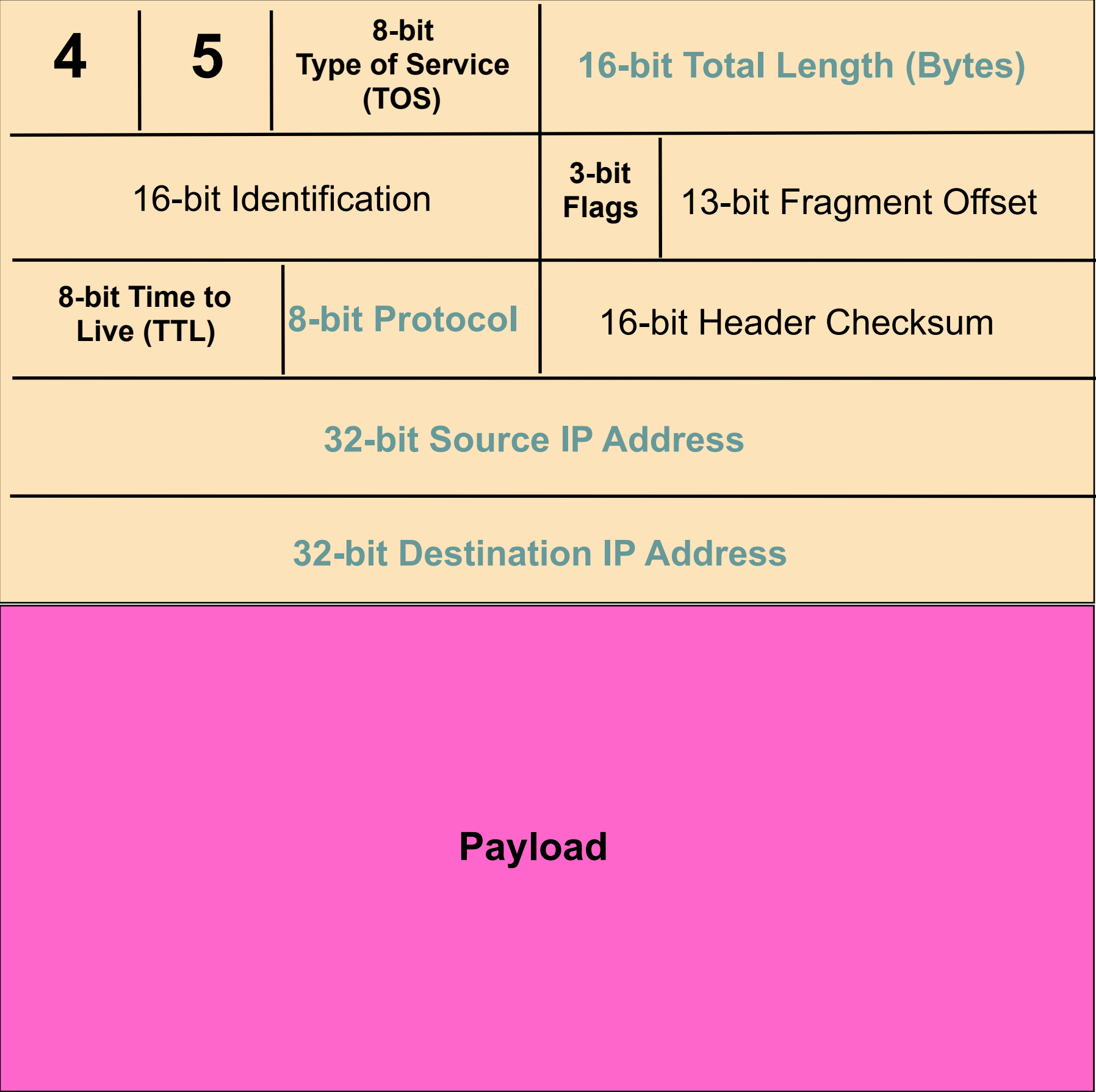
Multiplexing and Demultiplexing
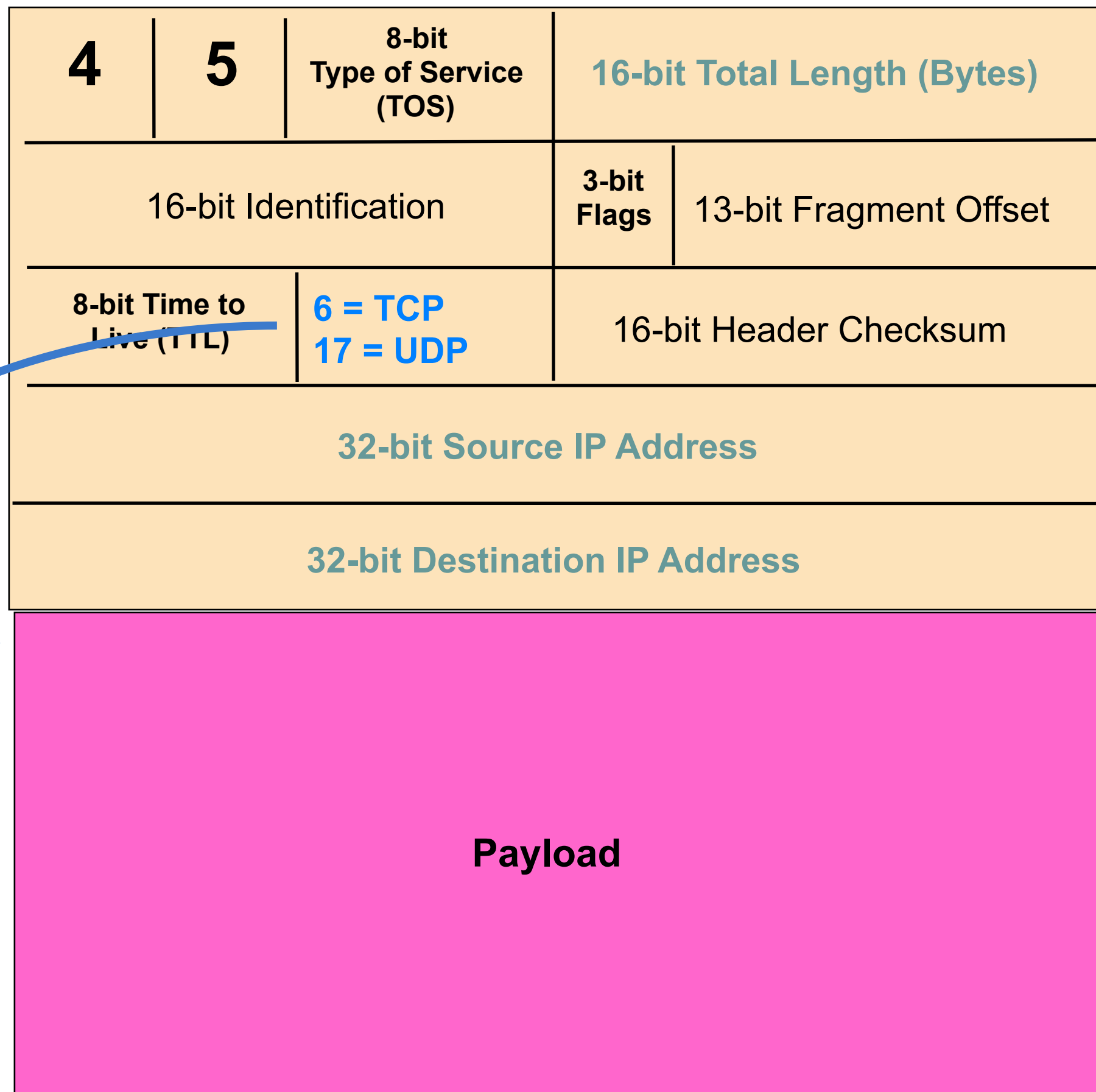
Host receives IP datagrams

- Each datagram has source and destination IP address,

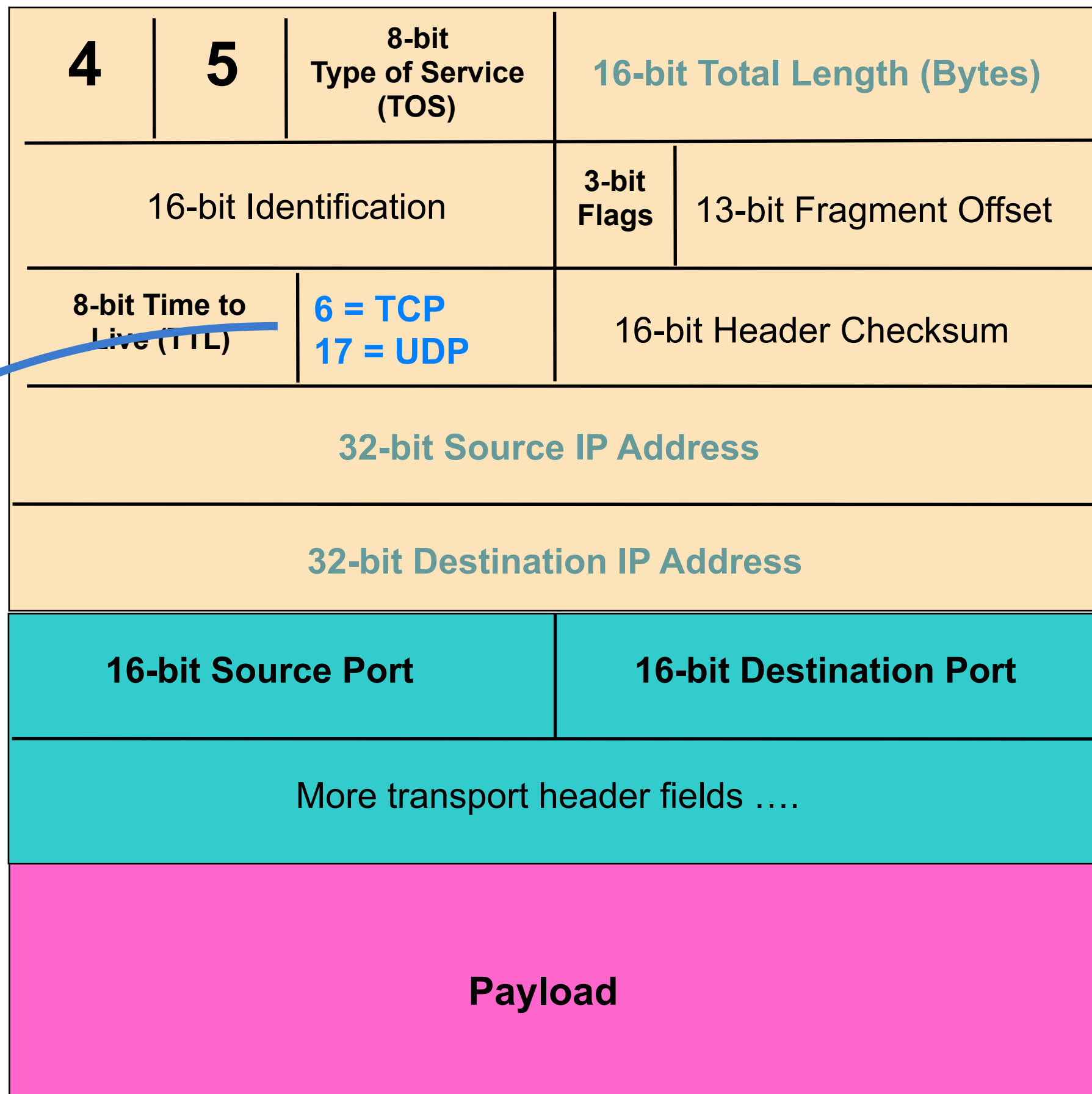- Each segment has source and destination port number

Host uses IP addresses and port numbers to direct the segment to appropriate socket

32 bits
source port # / dest port #
other header fields
application data (message)

4-bit Version	4-bit Header Length	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)		8-bit Protocol	16-bit Header Checksum	
32-bit Source IP Address				
32-bit Destination IP Address				
Options (if any)				
Payload				

4	5	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)		8-bit Protocol	16-bit Header Checksum	
32-bit Source IP Address				
32-bit Destination IP Address				
Payload				

4	5	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)		6 = TCP 17 = UDP	16-bit Header Checksum	
32-bit Source IP Address				
32-bit Destination IP Address				

Payload

4	5	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)		**6 = TCP** **17 = UDP**	16-bit Header Checksum	
32-bit Source IP Address				
32-bit Destination IP Address				
16-bit Source Port			**16-bit Destination Port**	
More transport header fields ….				
Payload				

Connection Mappings

For UDP ports (SOCK_DGRAM)

- OS stores (local port, local IP address) ←→ socket

For TCP ports (SOCK_STREAM)

- OS stores (local port, local IP, remote port, remote IP) ←→ socket

Why the difference?

Implications for mobility

Why do you need to include local IP?

UDP

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")

SRC port	DST port
checksum	length
DATA	

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

Popular Applications That Use UDP

Some interactive streaming apps

- Retransmitting lost/corrupted packets often pointless: by the time the packet is retransmitted, it's too late
  - telephone calls, video conferencing, gaming...
  - *Modern streaming protocols using TCP (and HTTP)*

Simple query protocols like Domain Name System (DNS)

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

"Address for bbc.co.uk?"

"212.58.224.131"

TCP

Transmission Control Protocol (TCP)

Reliable, in-order delivery *(previously, but quick review)*

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

Connection oriented *(today)*

- Explicit set-up and tear-down of TCP session

Full duplex stream-of-bytes service *(today)*

- Sends and receives a stream of bytes, not messages

Flow control *(previously, but quick review)*

- Ensures that sender doesn't overwhelm receiver

Congestion control *(next week)*

- Dynamic adaptation to network path's capacity

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 (ignore)

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	

Segments and Sequence Numbers

TCP "Stream of Bytes" Service...

Application @ Host A

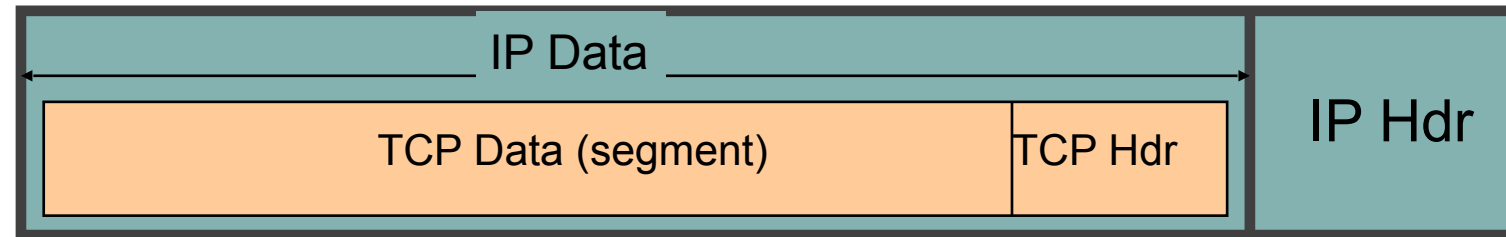Application @ Host B

... Provided Using TCP "Segments"

Host A

Byte 0 | Byte 1 | Byte 2 | Byte 3 | ... | Byte 80

TCP Data

Segment sent when:
1. Segment full (Max Segment Size),
2. Not full, but times out

TCP Data

Host B

Byte 0 | Byte 1 | Byte 2 | Byte 3 | ... | Byte 80

TCP Segment

IP Data		
TCP Data (segment)	TCP Hdr	IP Hdr

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
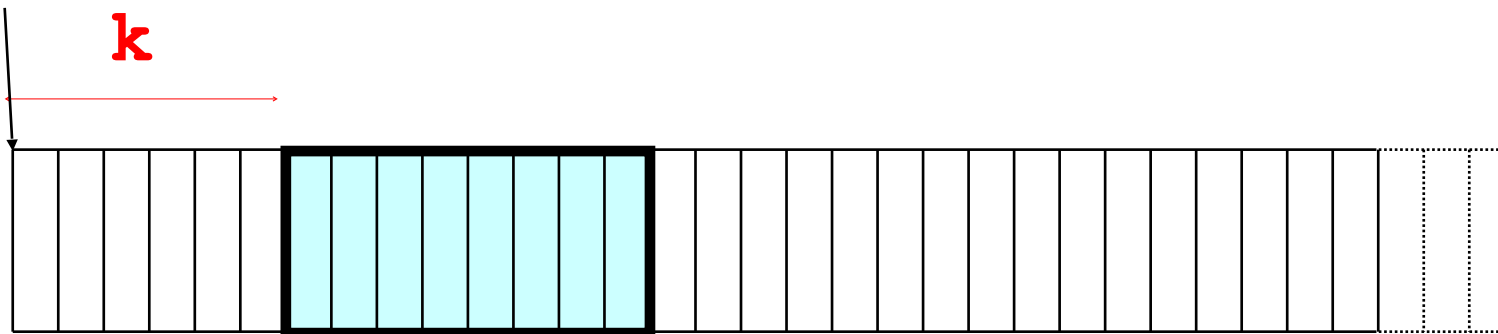- MSS = MTU – (IP header) – (TCP header)

Sequence Numbers

ISN (initial sequence number)

k bytes

Host A

Sequence number
= 1st byte in segment =
ISN + k
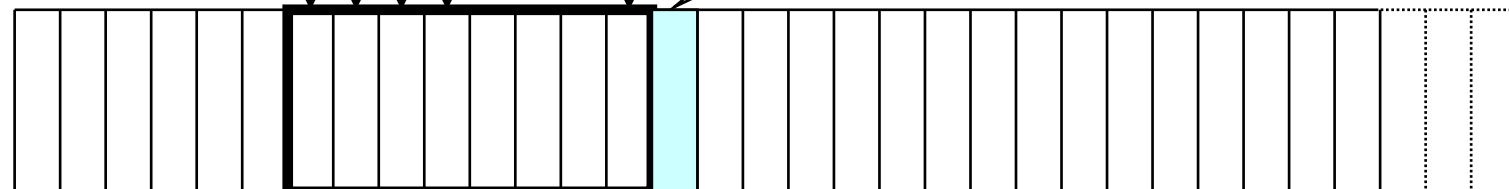
Sequence Numbers

ISN (initial sequence number)

k

Host A

Sequence number
= 1st byte in segment =
ISN + k

TCP Data | TCP HDR

TCP Data | TCP HDR

ACK sequence number
= next expected byte
= seqno + length(data)

Host B

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

Normal Pattern

Sender: seqno=X, length=B

Receiver: ACK=X+B

Sender: seqno=X+B, length=B

Receiver: ACK=X+2B

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

…

Seqno of next packet is same as last ACK field

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

What's missing?

Implementing Sliding Window

Both sender & receiver maintain a **window**
- Sender: not yet ACK'ed
- Receiver: not yet delivered to application

Left edge of window:
- Sender: beginning of unacknowledged data
- Receiver: beginning of undelivered data

For the sender:
- Window size = maximum amount of data in flight

For the receiver:
- Window size = maximum amount of undelivered data

Sliding Window Summary

Sender: window advances when new data ack'd

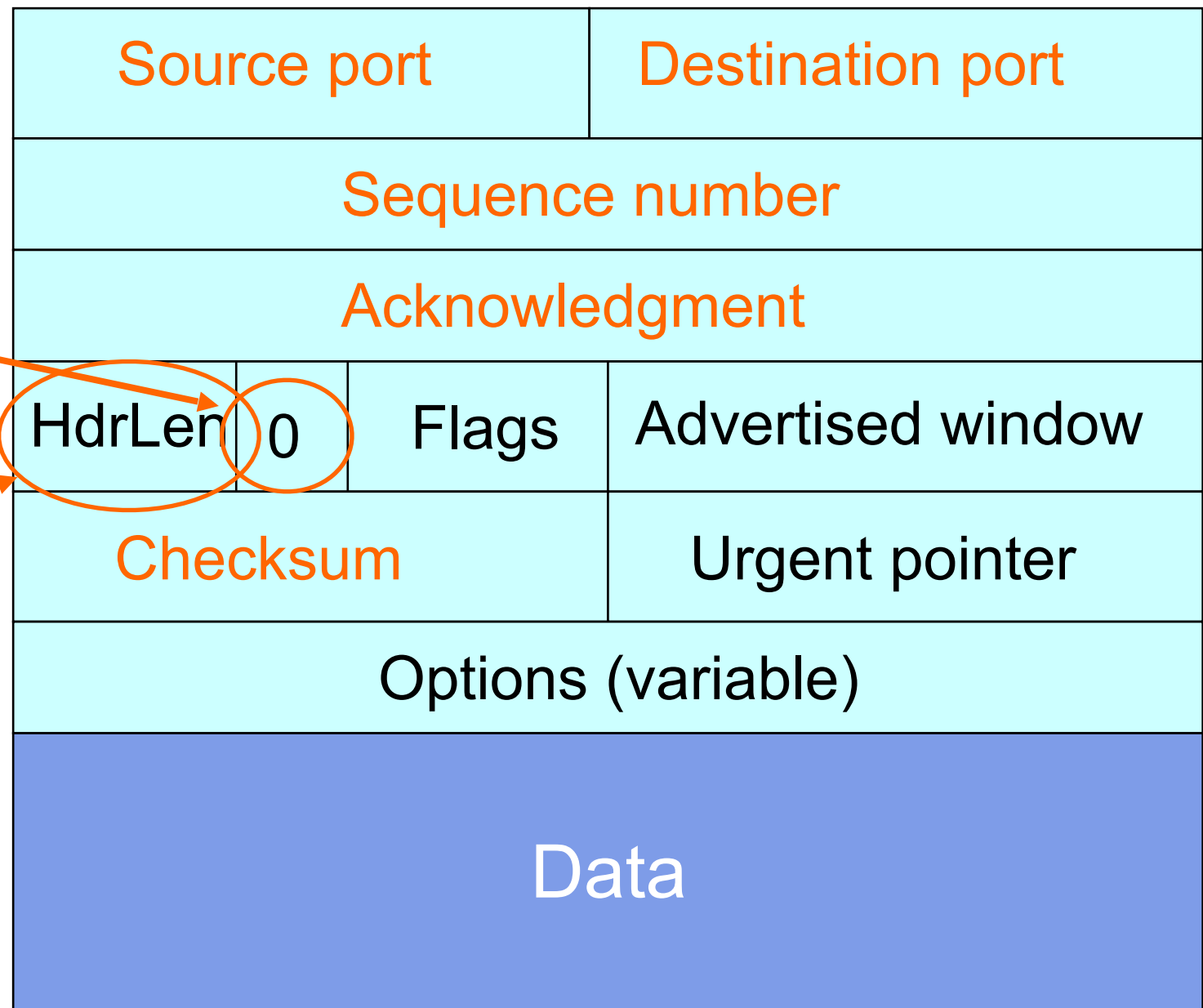Receiver: window advances as receiving process consumes data

Receiver advertises to the 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 can't send beyond the receiver's righthand edge
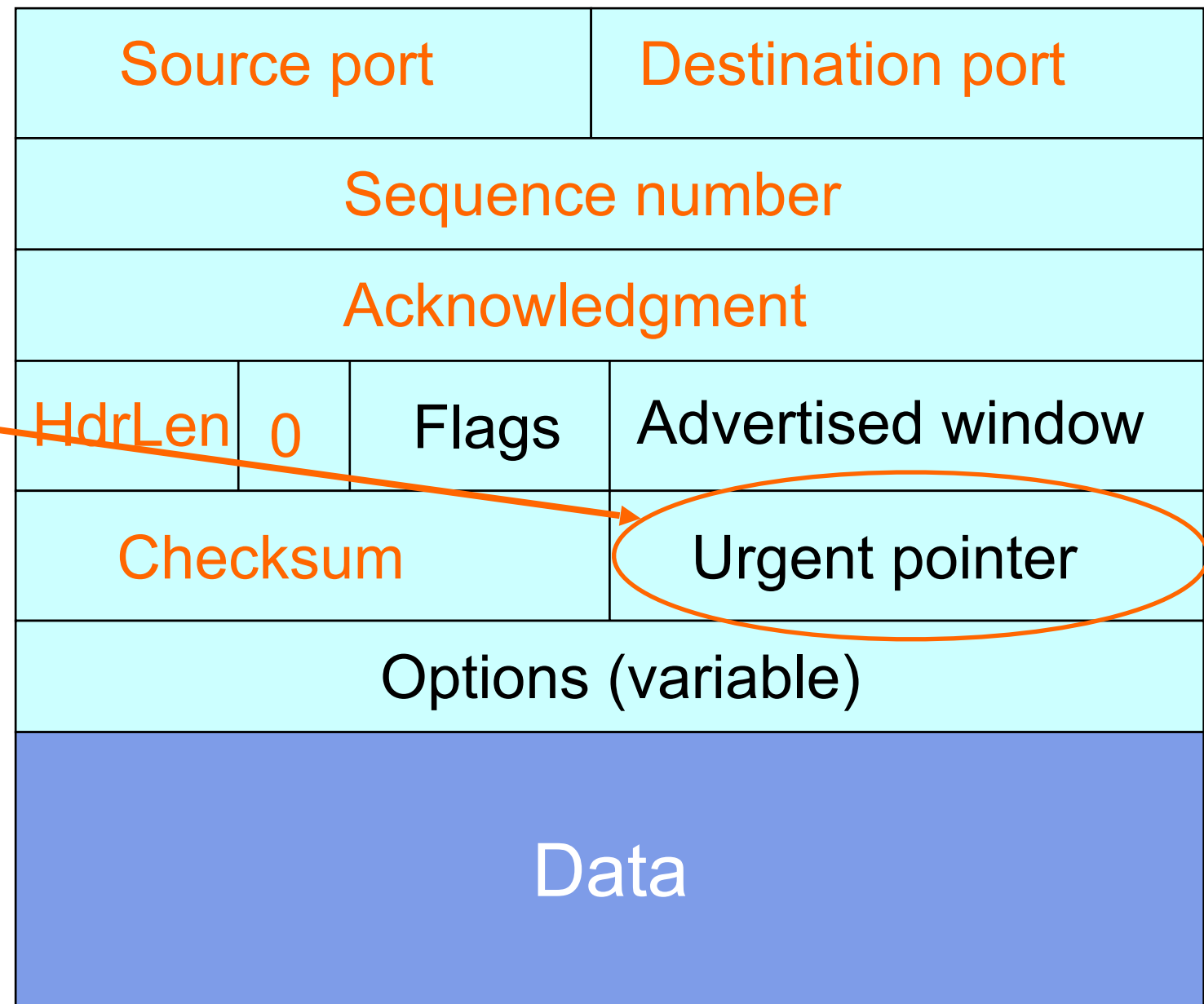
TCP Header: What's left?

"Must Be Zero"
6 bits reserved

Number of 4-byte words in TCP header;
5 = no options

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

TCP Header: What's left?

Used with **URG** flag to indicate urgent data (not discussed further)

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

TCP Header: What's left?

Source port	Destination port
Sequence number	
Acknowledgment	

HdrLen	0	Flags	Advertised window

Checksum	Urgent pointer

Options (variable)

Data

TCP Connection Establishment and Initial Sequence Numbers

Initial Sequence Number (ISN)

Sequence number for the very first byte

- E.g., Why not just use ISN = 0?

Practical issue

- IP addresses and port #s uniquely identify a connection
- Eventually, though, these port #s do get used again
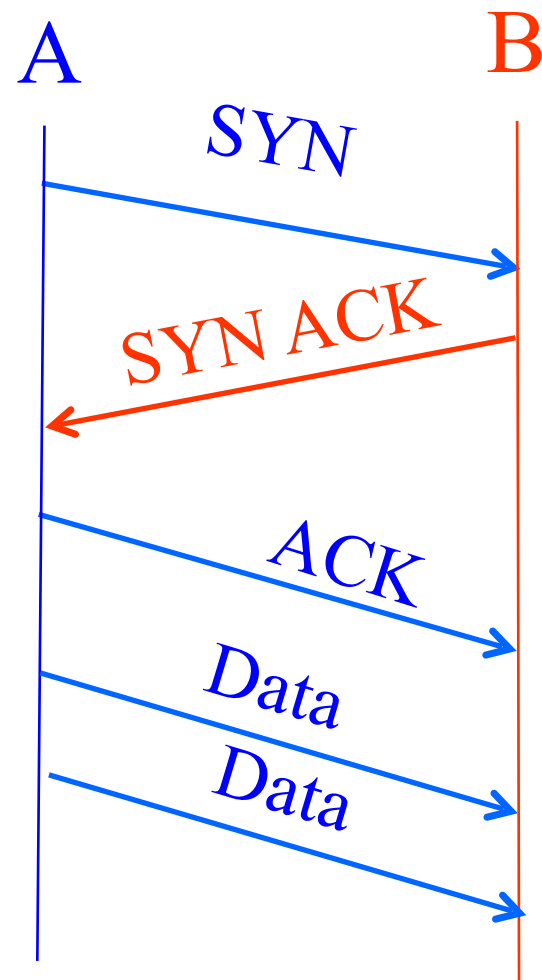- … small chance an old packet is still in flight

TCP therefore requires changing ISN

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

To establish a connection, hosts exchange ISNs

- **How does this help?**

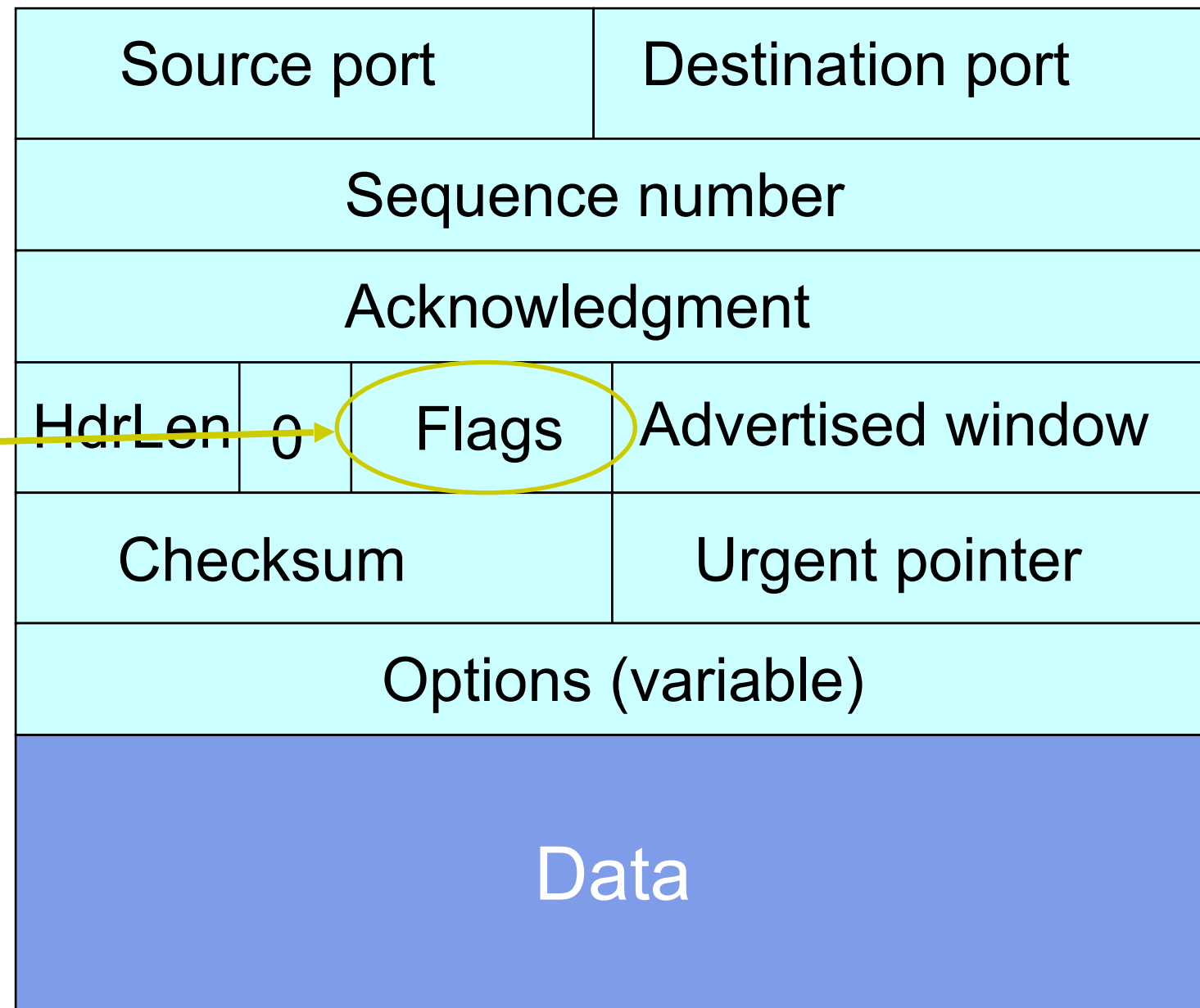# Establishing a TCP Connection

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**)
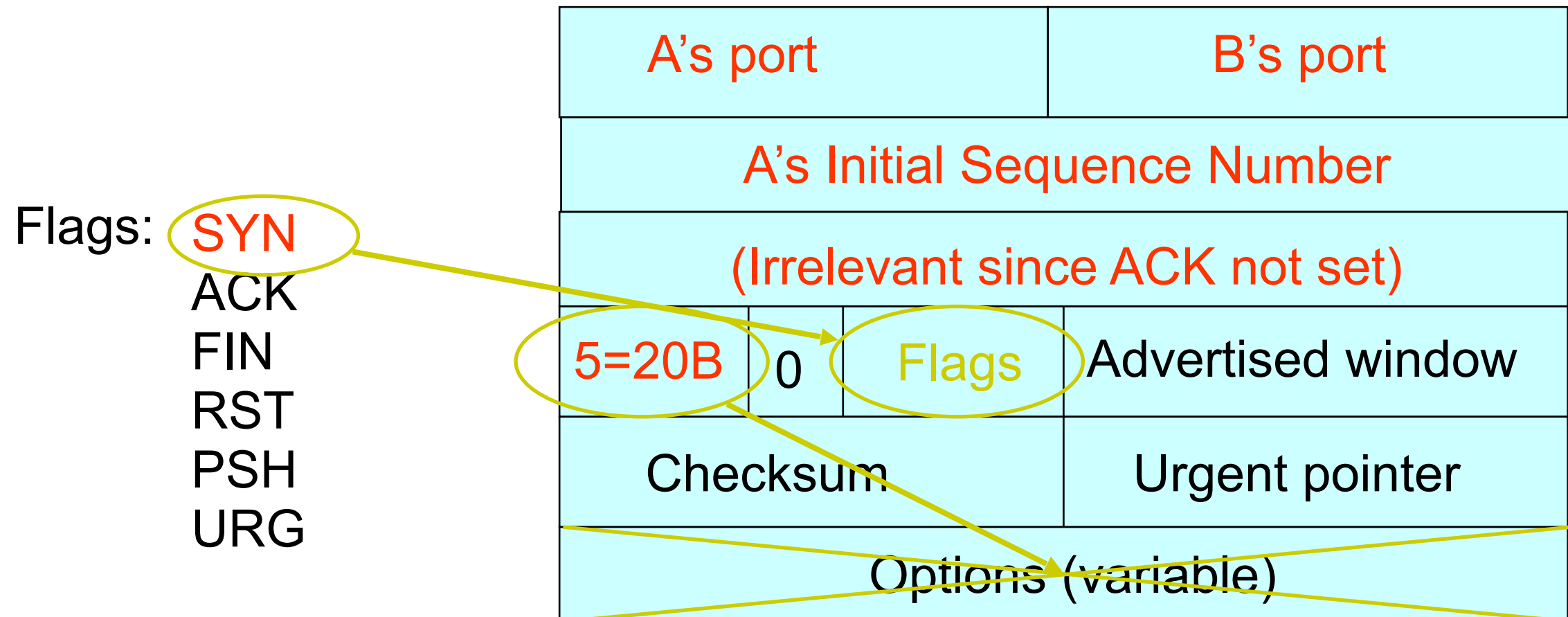- Host A sends an **ACK** to acknowledge the SYN ACK
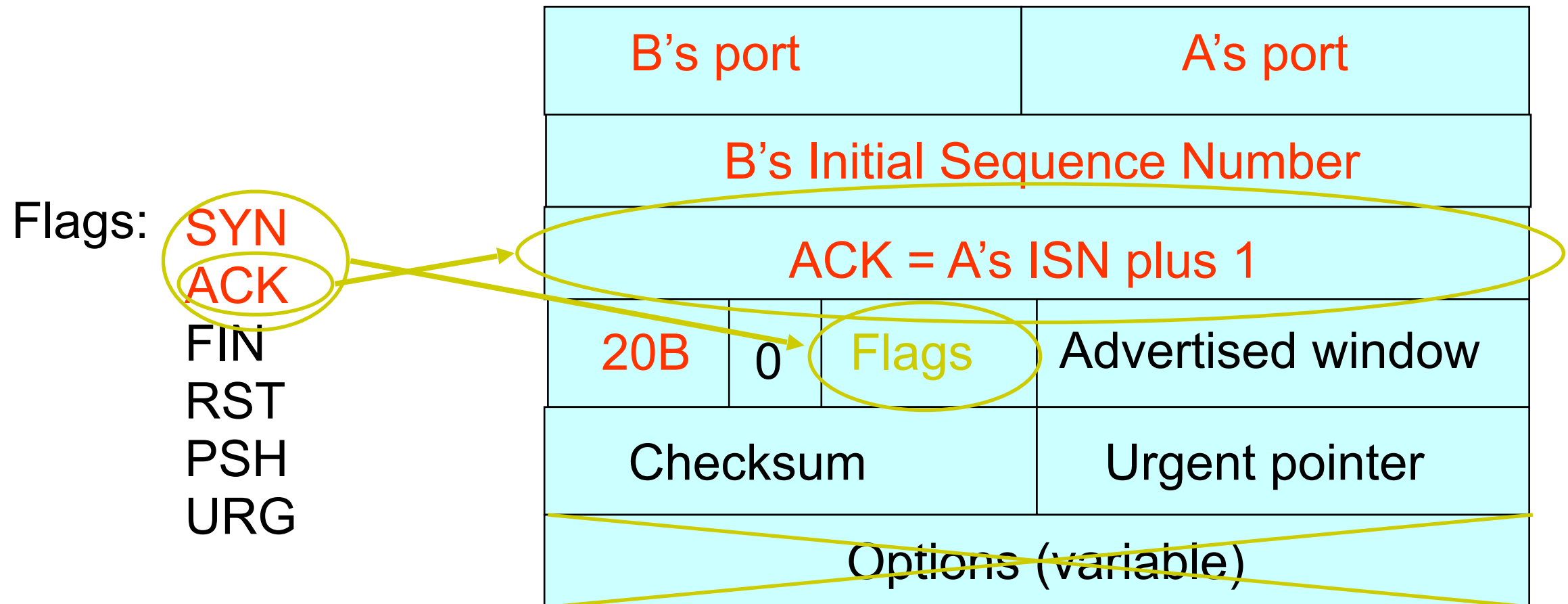
TCP Header

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	

See /usr/include/netinet/tcp.h on Unix Systems

Step 1: A's Initial SYN Packet

Flags: SYN
ACK
FIN
RST
PSH
URG

A's port	B's port
A's Initial Sequence Number	
(Irrelevant since ACK not set)	

5=20B	0	Flags	Advertised window

Checksum	Urgent pointer
Options (variable)	

A tells B it wants to open a connection…

Step 2: B's SYN-ACK Packet

Flags:
SYN
ACK
FIN
RST
PSH
URG

B's port	A's port
B's Initial Sequence Number	
ACK = A's ISN plus 1	

20B	0	Flags	Advertised window

Checksum	Urgent pointer
Options (variable)	

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

… upon receiving this packet, A can start sending data

Step 3: A's ACK of the SYN-ACK

Flags: SYN
ACK
FIN
RST
PSH
URG

A's port	B's port
A's Initial Sequence Number	
B's ISN plus 1	

20B	0	Flags	Advertised window

Checksum	Urgent pointer

Options (variable)

A tells B it's likewise okay to start sending

… upon receiving this packet, B can start sending data

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

SYN Loss and Web Downloads

User clicks on a hypertext link

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

If the SYN is lost…

- 3-6 seconds of delay: can be very long
- User may become impatient
- … and click the hyperlink again, or click "reload"

User triggers an "abort" of the "connect"

- Browser creates a new socket and another "connect"
- Essentially, forces a faster send of a new SYN packet!
- Sometimes very effective, and the page comes quickly

Tearing Down the Connection

Normal 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 the connection, but not B's

- Until B likewise sends a **FIN**
- Which A then acks

Connection now **closed**

Connection now **half-closed**

Timeout:

Avoid *reincarnation* B will retransmit FIN if ACK is lost

Normal Termination, Both Together

Same as before, but B sets **FIN** with their ack of A's **FIN**

B

SYN

SYN ACK

ACK

Data

ACK

• • •

FIN

FIN + ACK

ACK

A

time

Timeout:

Avoid reincarnation
Can retransmit
FIN ACK if ACK lost

Connection
now closed

Abrupt Termination

A sends a RESET (**RST**) to B
- E.g., because app. process on A crashed

That's it
- B does not ack the **RST**
- Thus, **RST** is not delivered reliably
- And: any data in flight is lost
- But: if B sends anything more, will elicit another **RST**

TCP State Transitions

Reliability: TCP Retransmission

Timeouts and Retransmissions

Reliability requires retransmitting lost data

Involves setting timer and retransmitting on timeout

TCP resets timer whenever new data is ACKed

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

Setting the Timeout Value

RTT

Timeout

1

1

Timeout too long → inefficient

Timeout

RTT

1

1

1

Timeout too short → duplicate packets

RTT Estimation

Use exponential averaging of RTT samples

$$SampleRTT = AckRcvdTime - SendPacketTime$$

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT$$
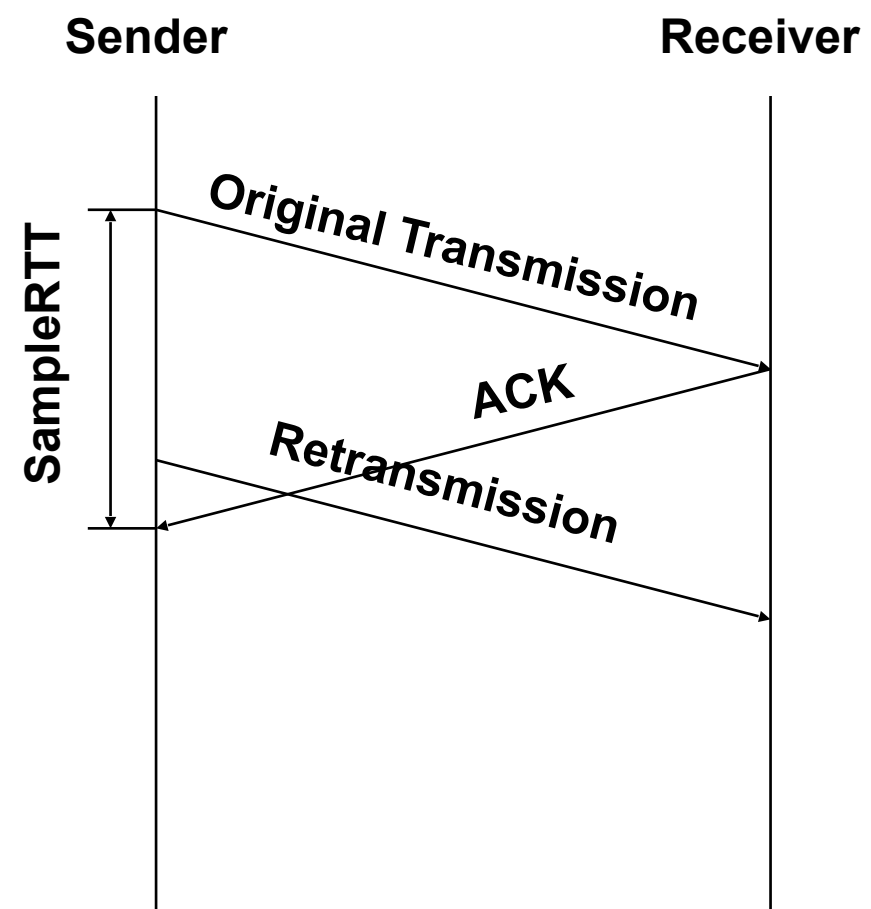
$$0 < \alpha \leq 1$$
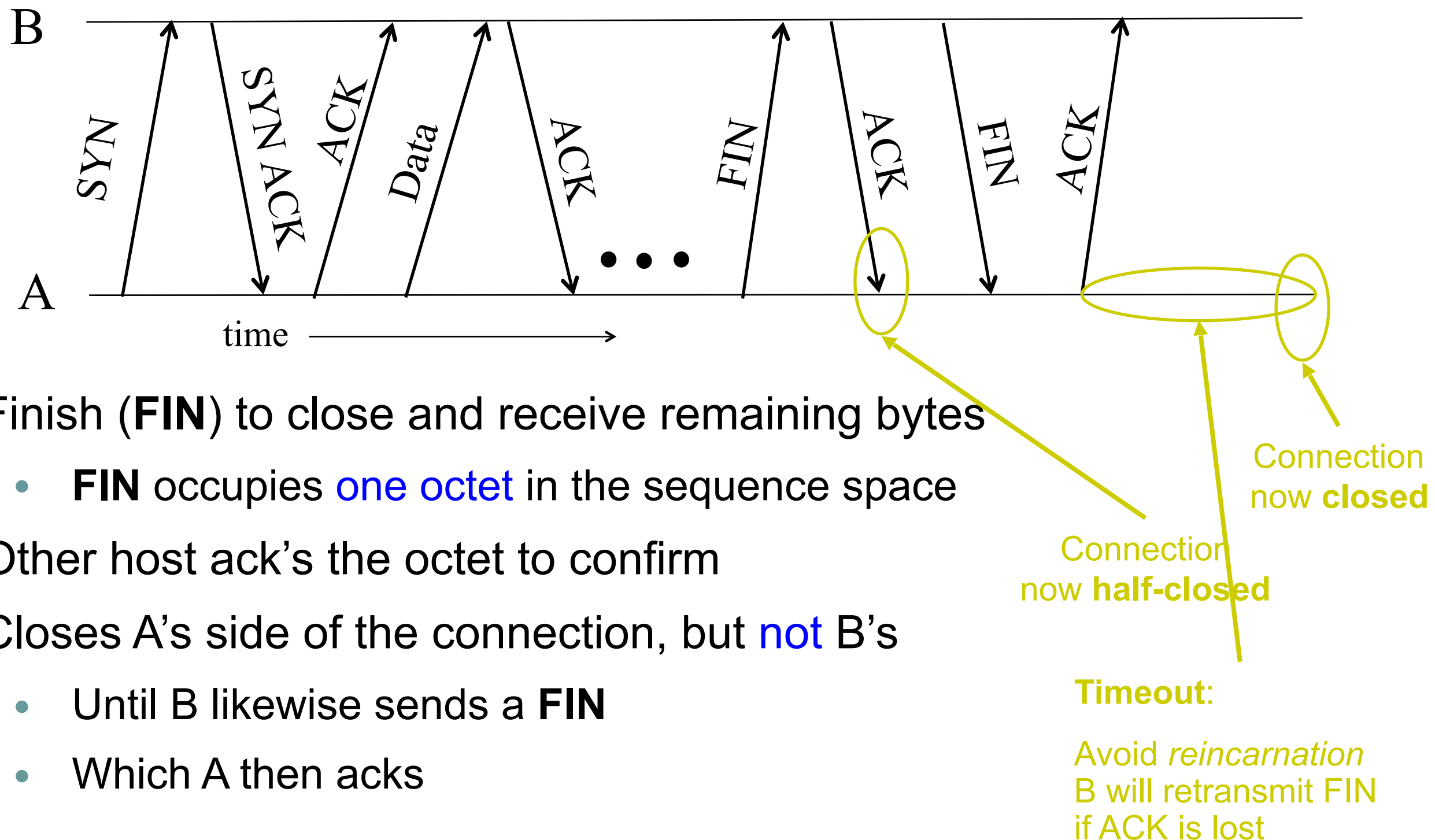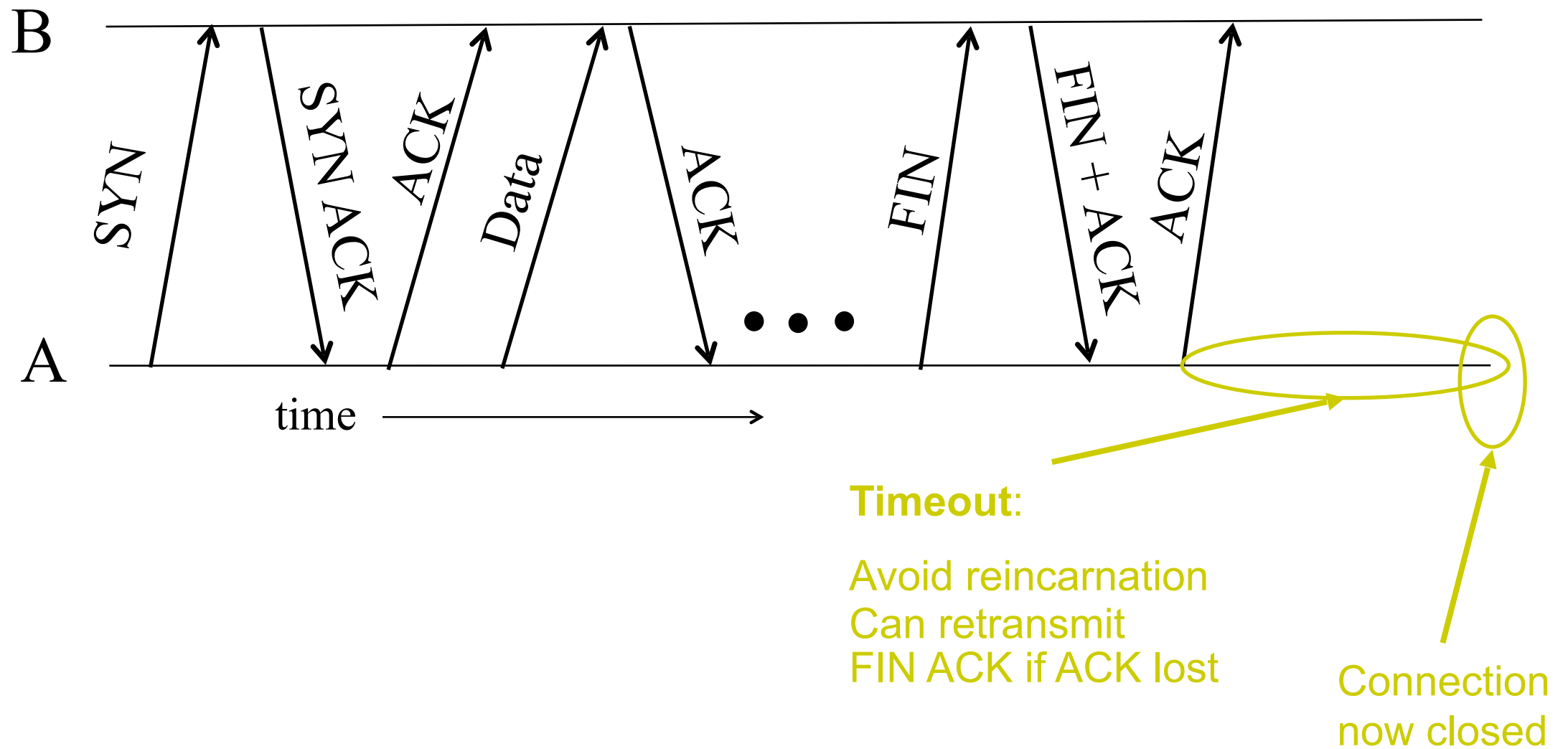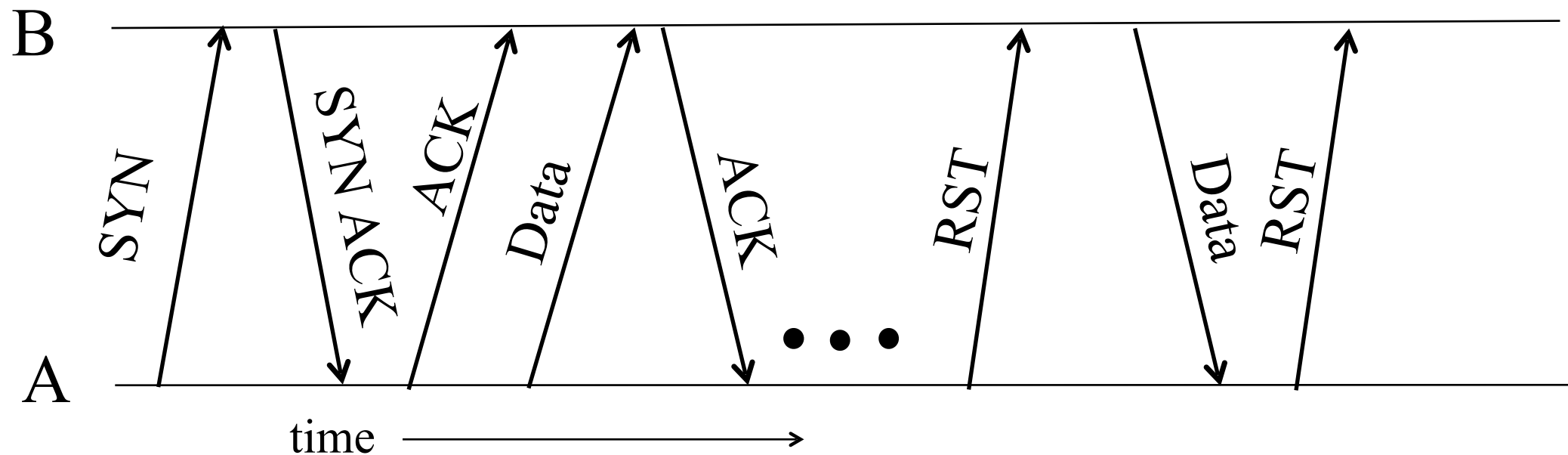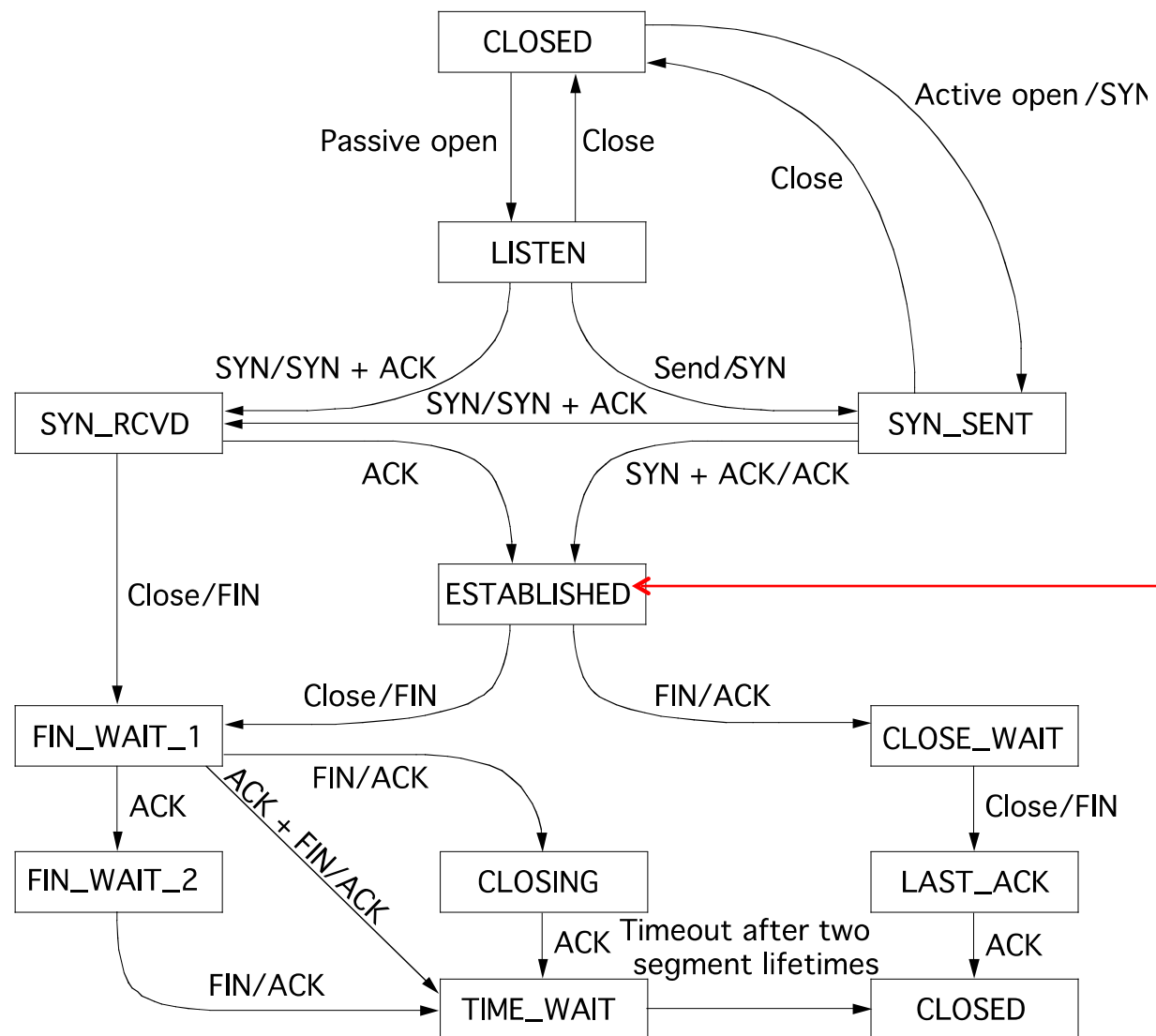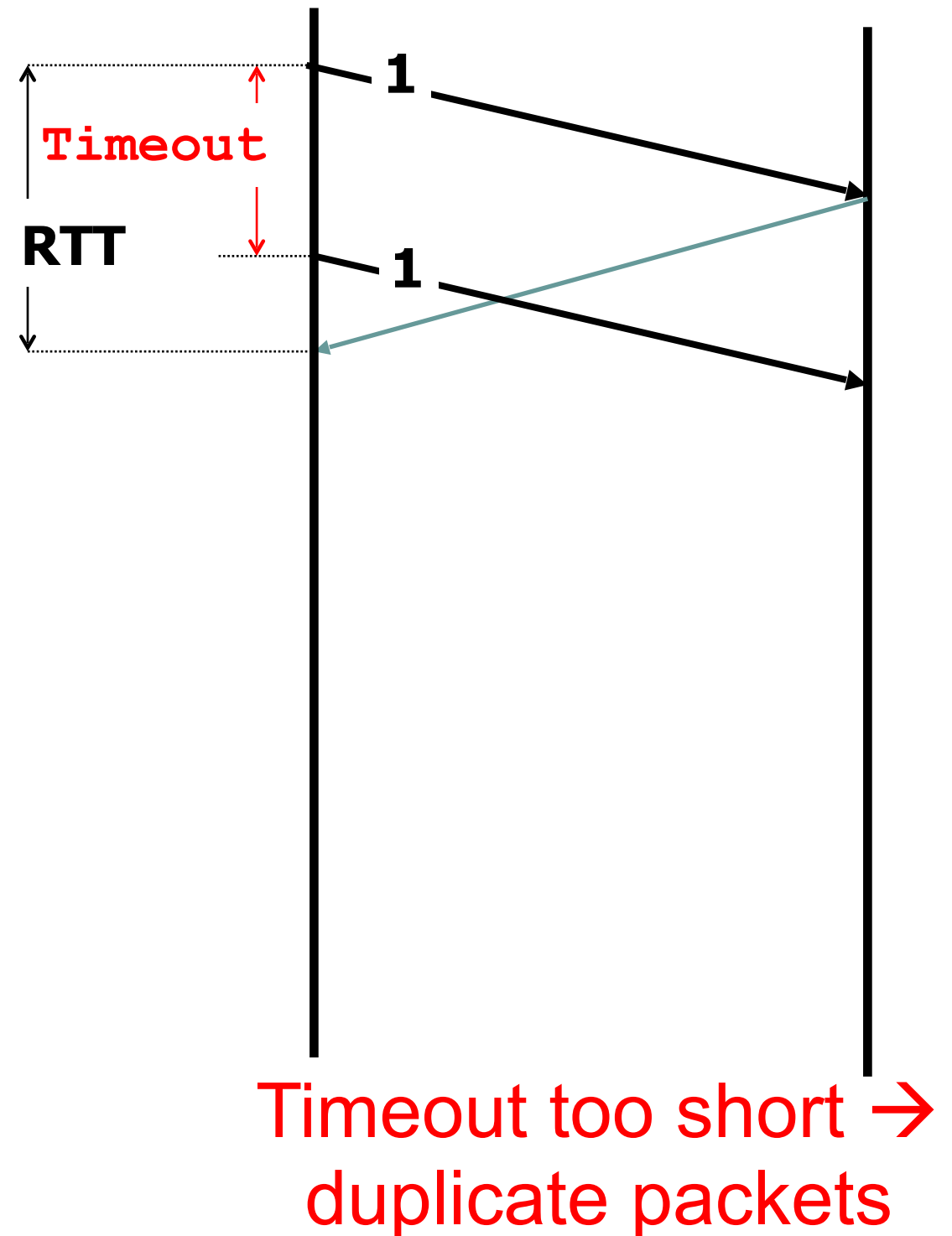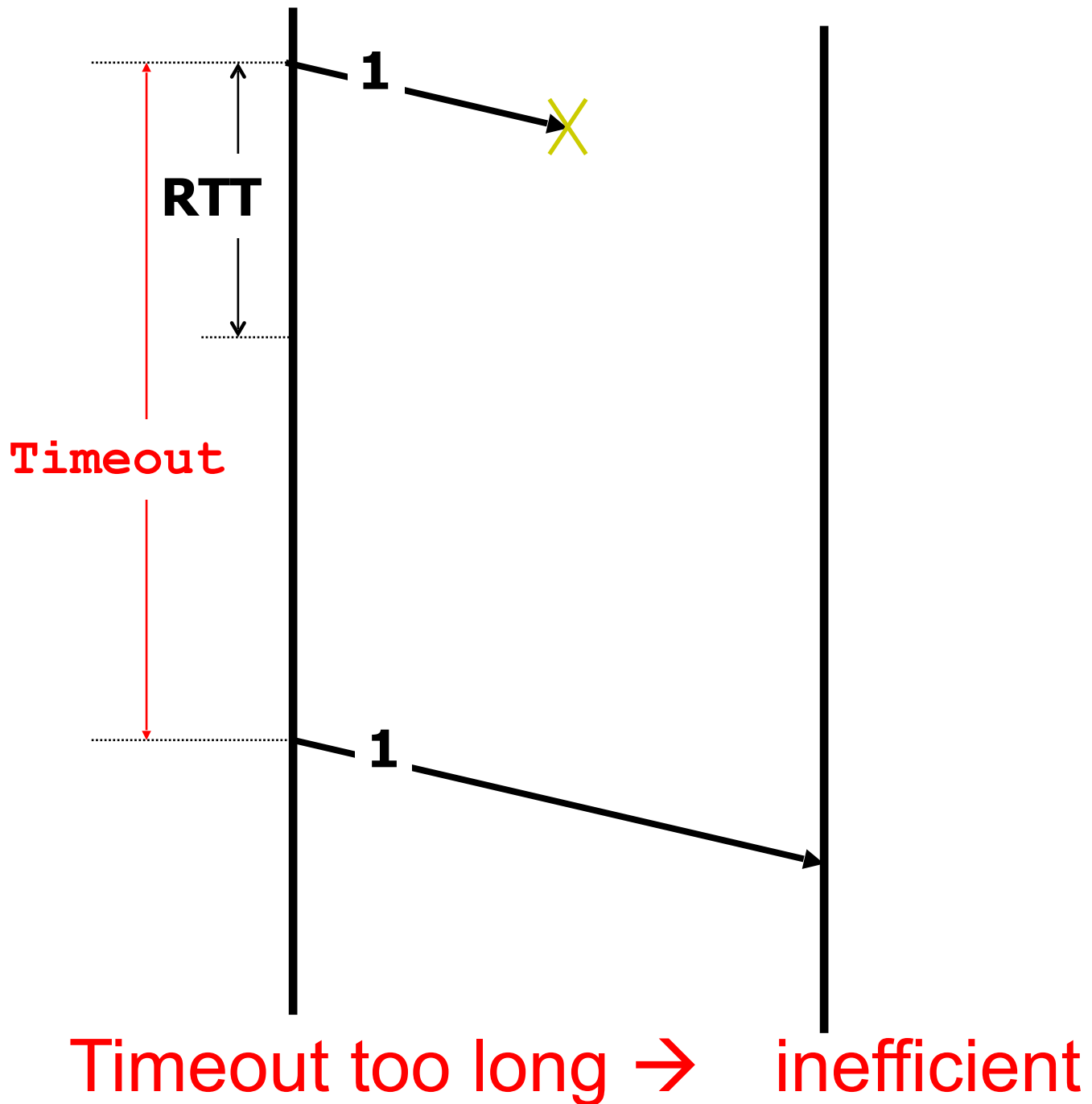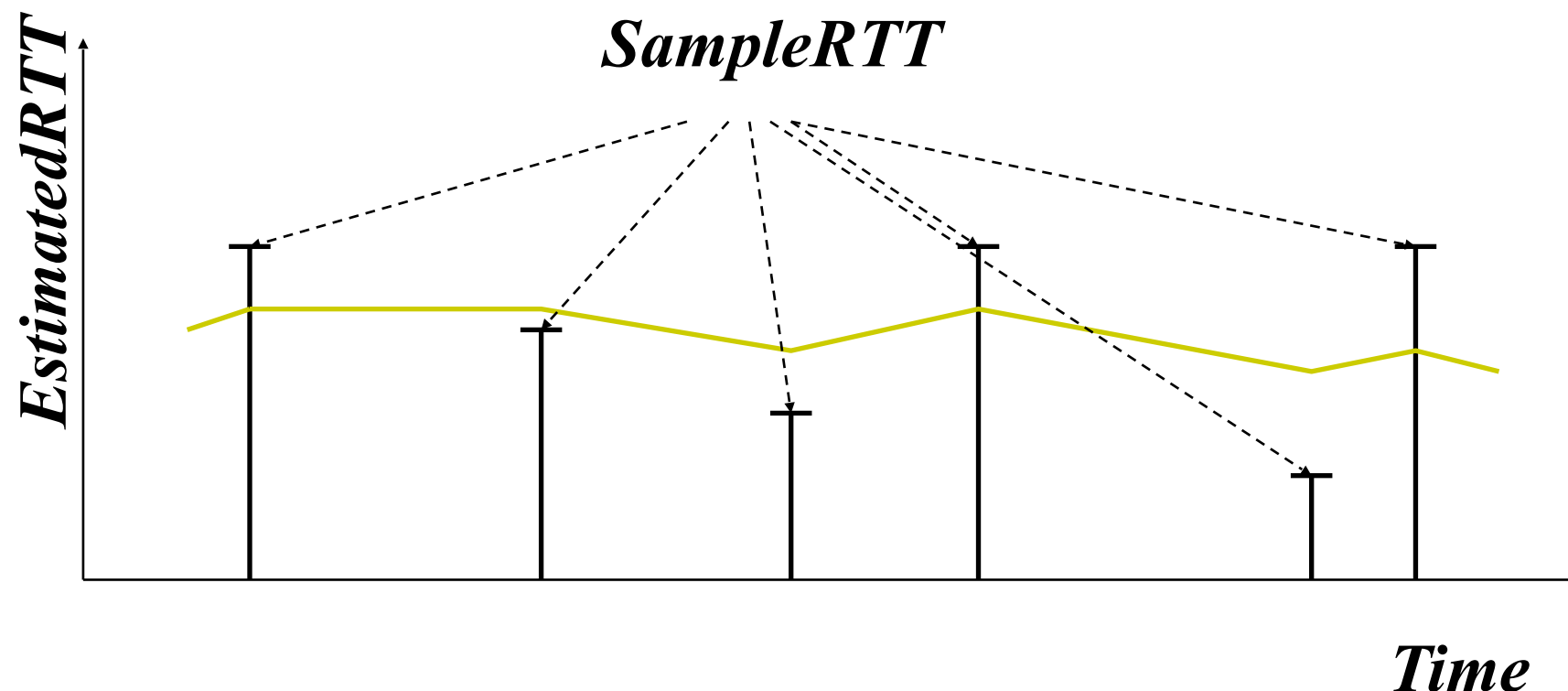
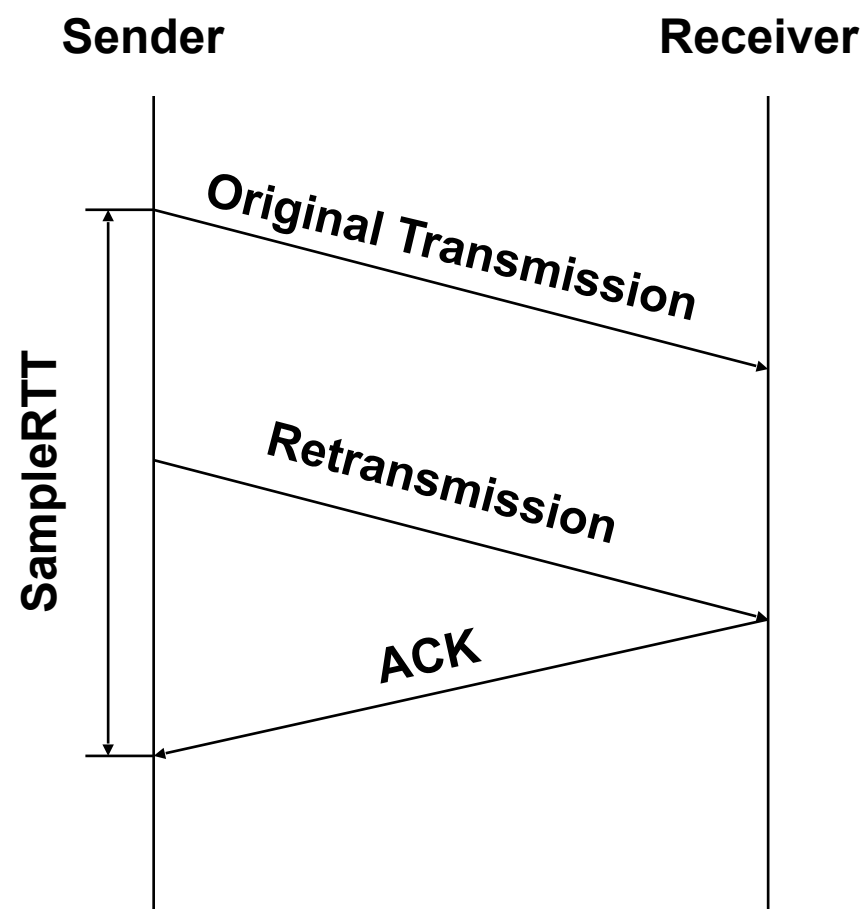# Exponential Averaging Example

EstimatedRTT = α*EstimatedRTT + (1 − α)*SampleRTT

Assume RTT is constant → *SampleRTT* = RTT
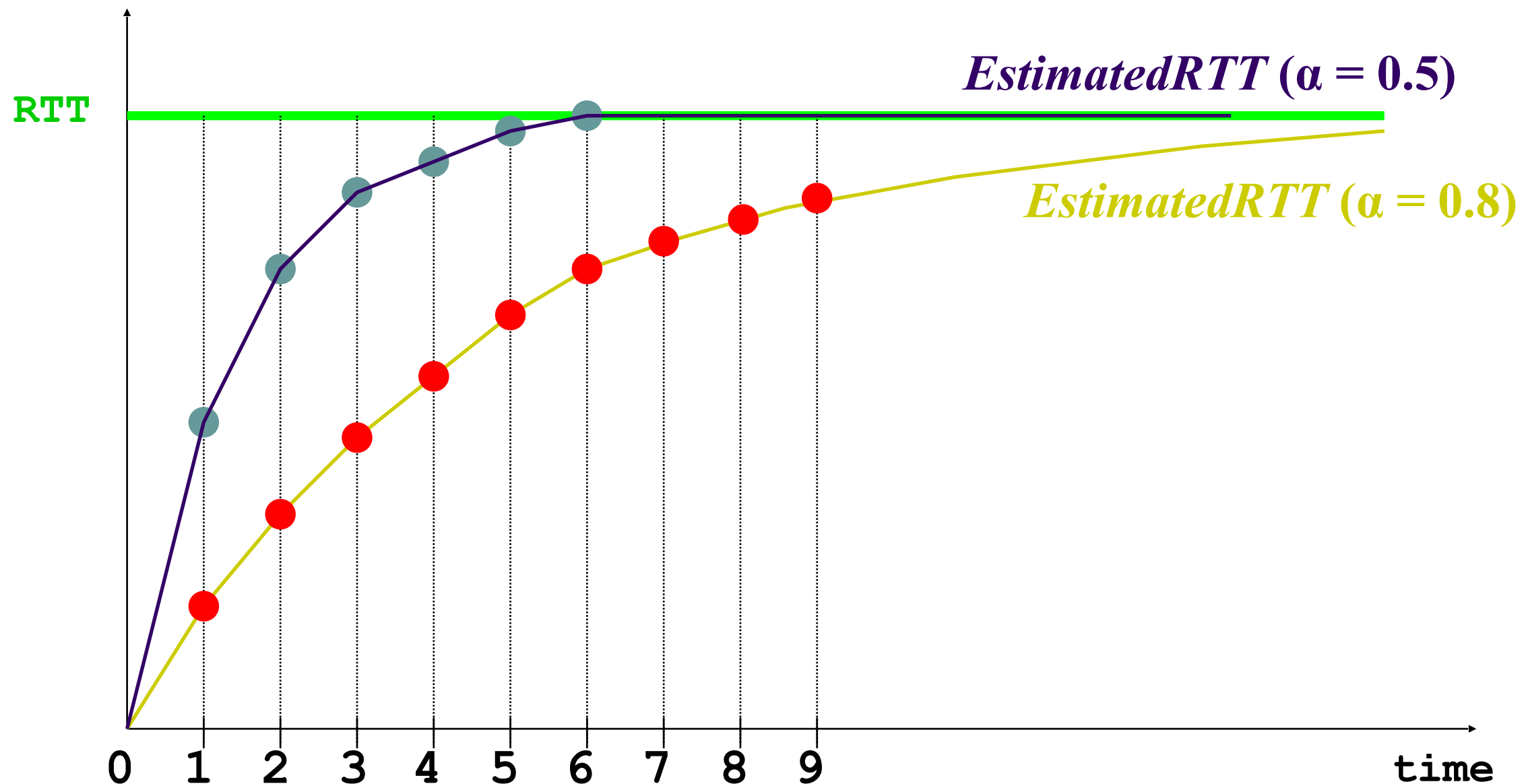
Problem: Ambiguous Measurements

How do we differentiate between the real ACK, and ACK of the retransmitted packet?

Karn/Partridge Algorithm

Measure *SampleRTT* only for original transmissions

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

Timeout value (RTO) = 2 × *EstimatedRTT*

Use exponential backoff for repeated retransmissions

- Every time RTO timer expires, set RTO ← 2·RTO
  - (Up to maximum ≥ 60 sec)
- Every time new measurement comes in (= successful original transmission), collapse RTO back to 2 × EstimatedRTT

This is all very interesting, but…..

Implementations often use a coarse-grained timer

- 500 msec is typical

So what?

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

So we rely on duplicate ACKs

Loss with cumulative ACKs

Sender sends packets with 100B and seqnos.:

- 100, 200, 300, 400, 500, 600, 700, 800, 900, …

Assume the fifth packet (seqno 500) is lost, but no others

Stream of ACKs will be:

- 200, 300, 400, 500, 500, 500, 500,…

Loss with cumulative ACKs

"Duplicate ACKs" are a sign of an *isolated* loss

- The lack of ACK progress means 500 hasn't been delivered
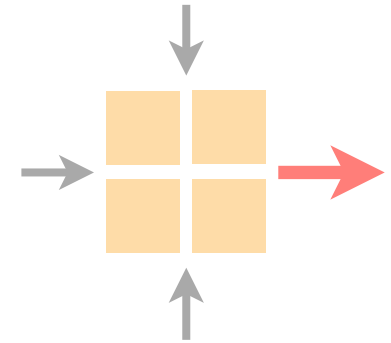- Stream of ACKs means some packets are being delivered

Therefore, could trigger resend upon receiving k duplicate ACKs

- TCP uses k=3

We will revisit this in congestion control

Communication Networks

Spring 2019

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

April 8 2019