CSCI 466: Networks

Lecture 6: UDP and RDT

Reese Pearsall
Fall 2022

Announcements

PA1 Due Monday September 26th
- Files must be pushed to a PA1 folder on your GitHub Repo
- Video demo is required
- Submit your repo link to D2L when finished

Next Friday will be a work day + help session. No Lecture

Announcements

PA1

Announcements

PA1

UDP Example

OSI Model

Application Layer

Presentation Layer *

Session Layer *

Transport Layer

Network Layer

Data Link Layer

Physical Layer

Application Layer

Messages from Network Applications

Physical Layer

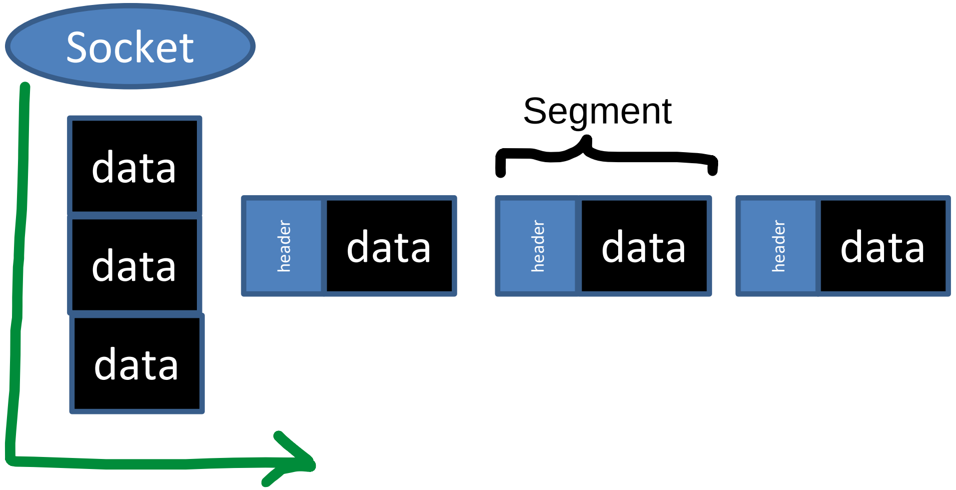
Bits being transmitted over some medium

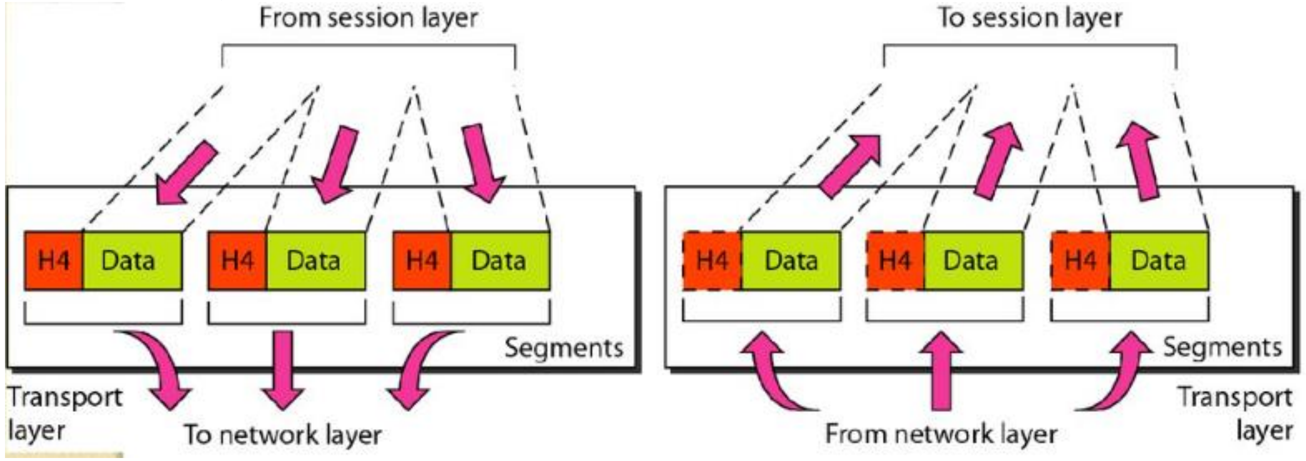
In the textbook, they condense it to a 5-layer model, but 7 layers is what is most used

Transport Layer

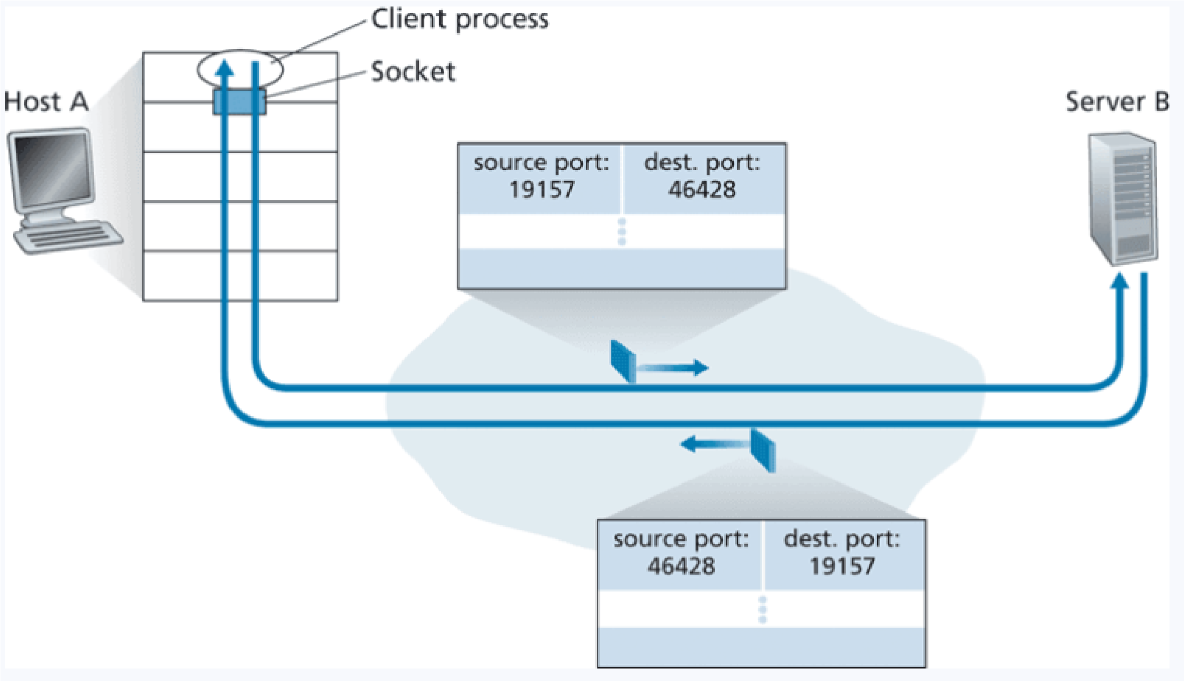
Multiplexing is the process of gathering chunks from sockets, encapsulating chunks with header information, and passing the segment into the network layer

Demultiplexing is the receiving segments from the transport layer and delivering the segment to the correct socket.





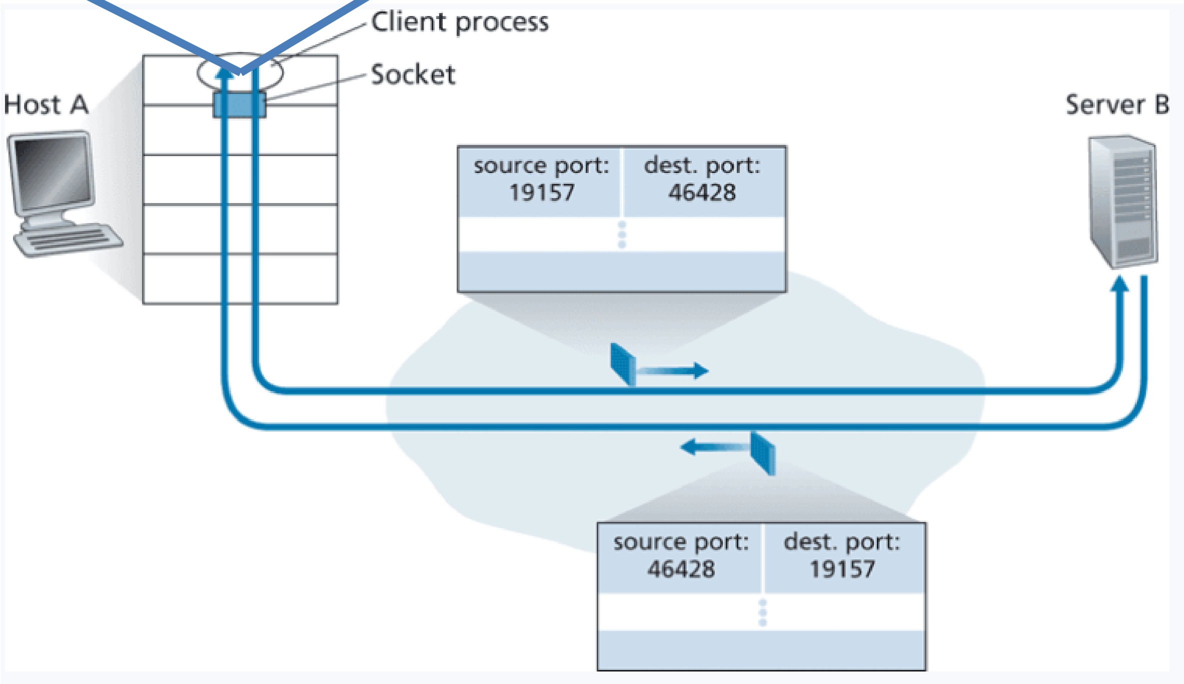
Transport Layer



Transport Layer

UDP sockets are identified by a two-tuple

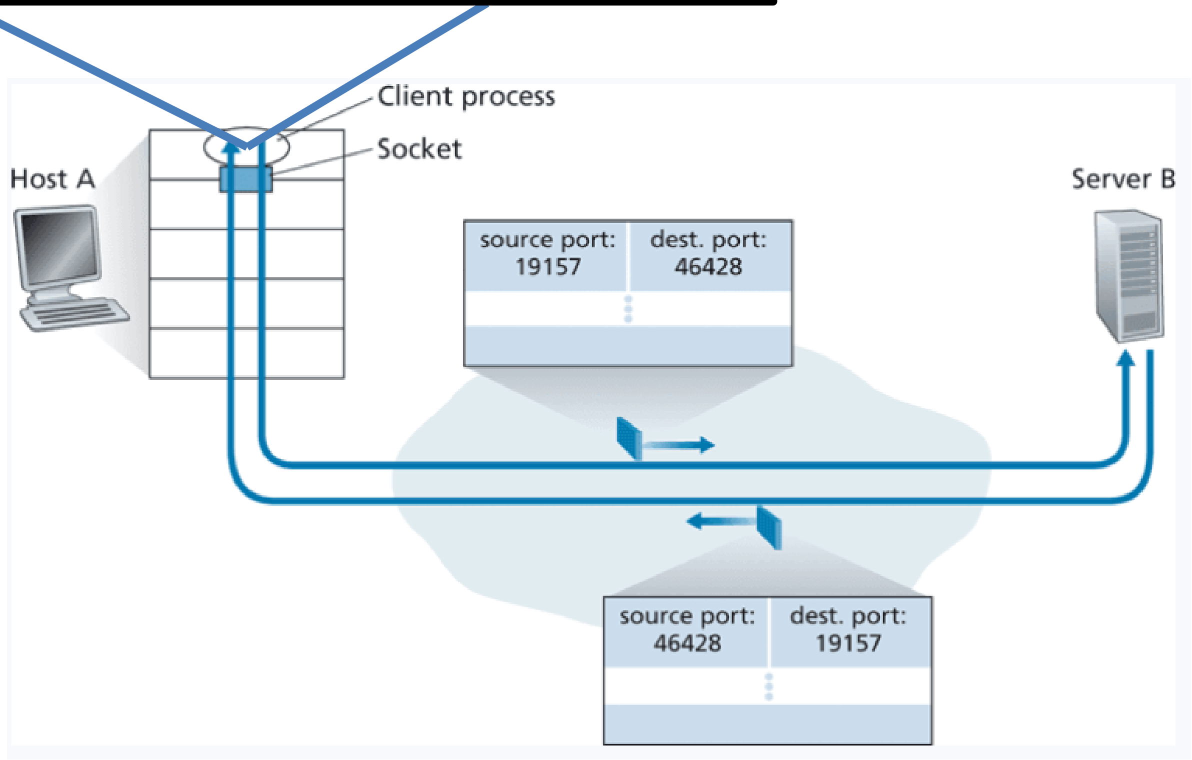
(destination IP address, destination port number)

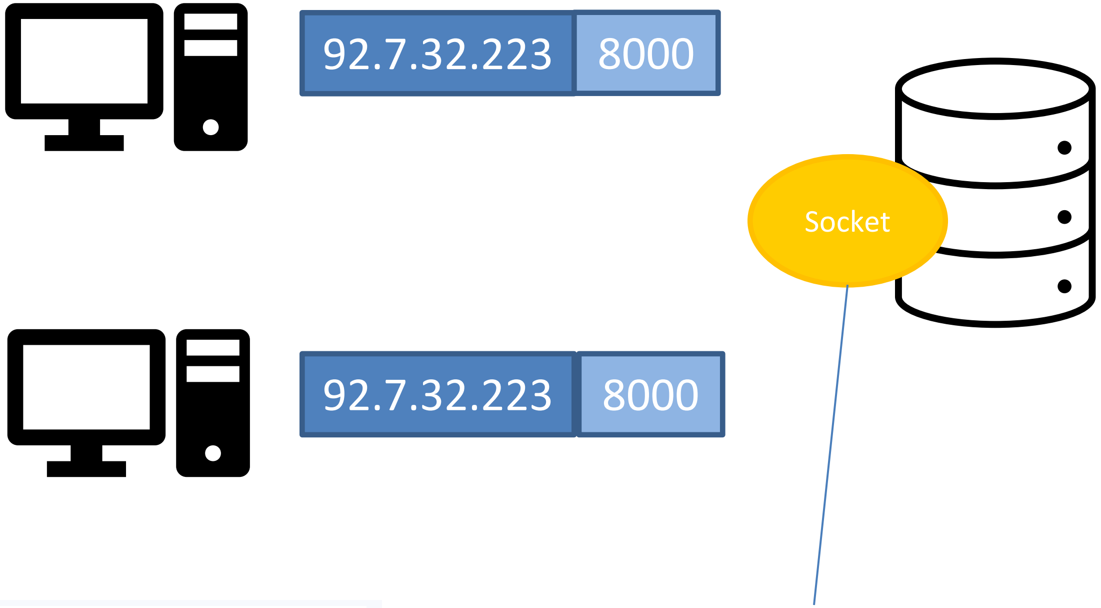


Transport Layer

UDP sockets are identified by a two-tuple

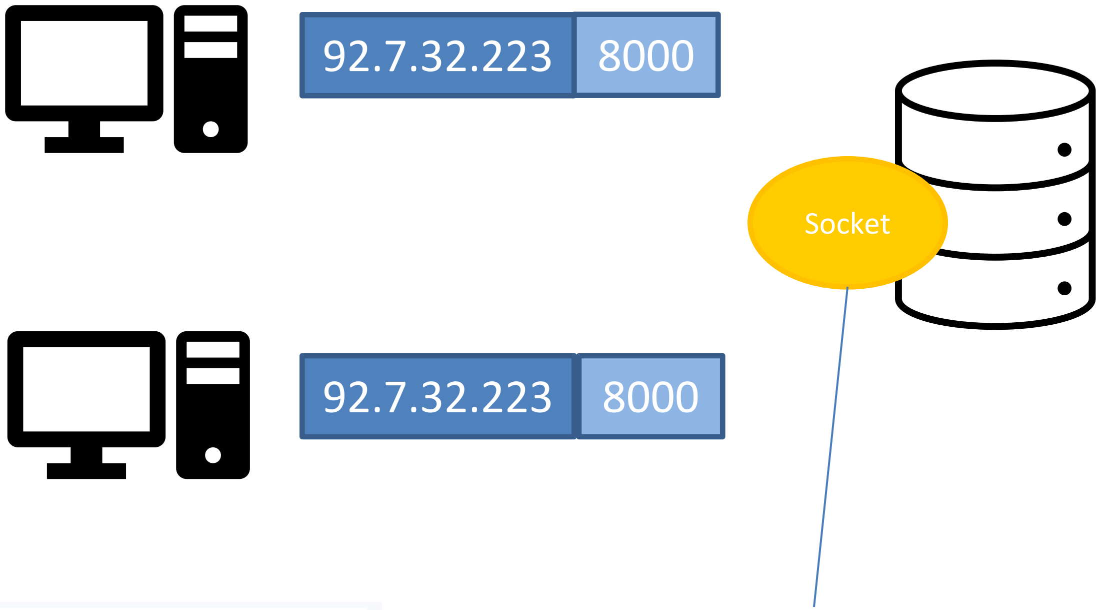
(destination IP address, destination port number)





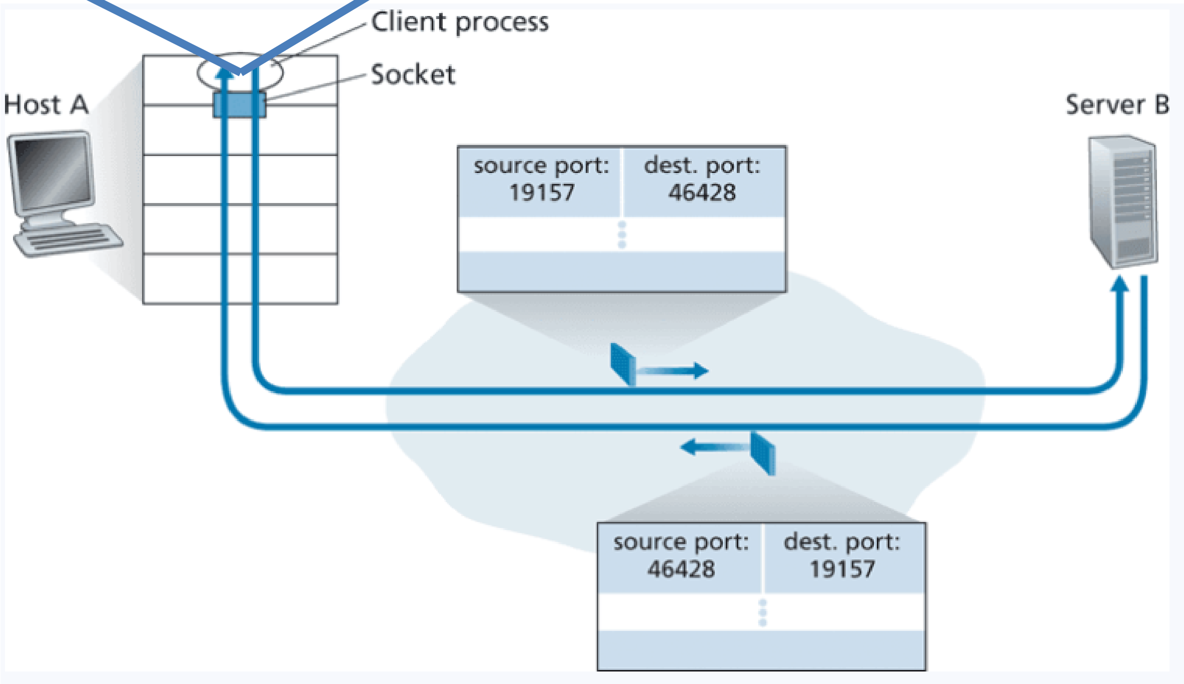
(92.7.32.223, 8000)

Transport Layer



UDP sockets are identified by a two-tuple

(destination IP address, destination port number)



(92.7.32.223, 8000)

The socket won't be able to distinguish between the two endpoints

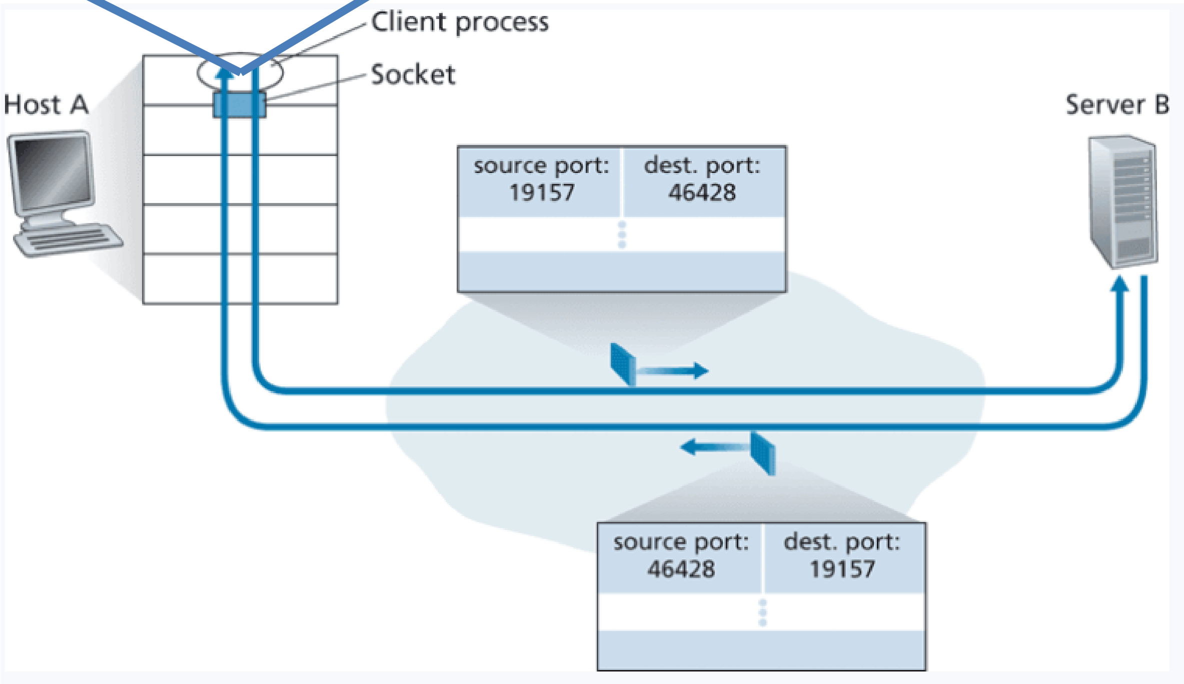
The two segments will be directed to the same process

Transport Layer

TCP sockets are identified by a four-tuple

A host will demultiplex segments using all of these values

(source IP address, source port number, destination IP address, destination port number)

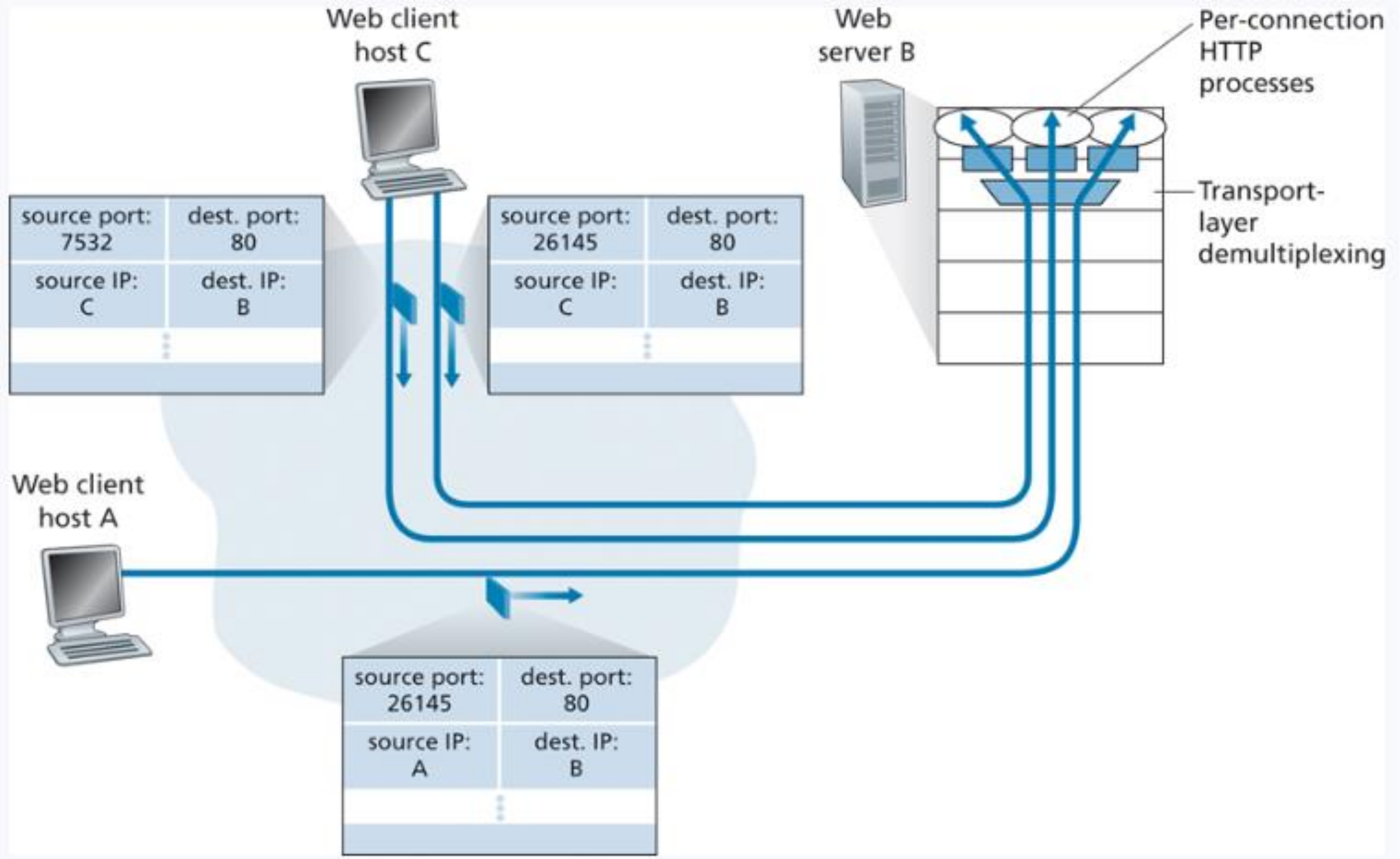


TCP servers will have a "welcoming socket" before creating the processes' socket

Transport Layer

(source IP address, source port number, destination IP address, destination port number)

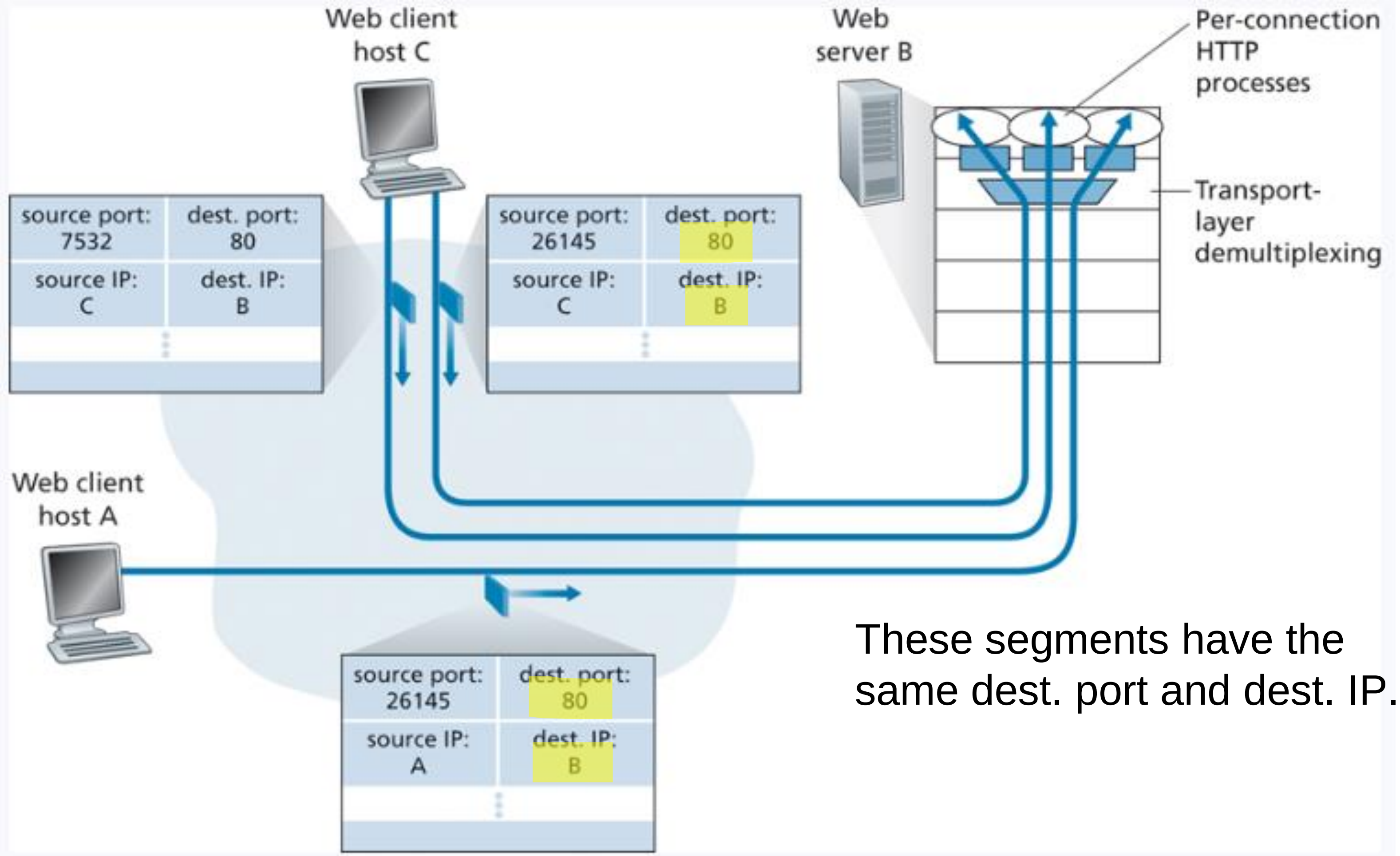
TCP sockets are identified by a four-tuple



Transport Layer

(source IP address, source port number, destination IP address, destination port number)

TCP sockets are identified by a four-tuple

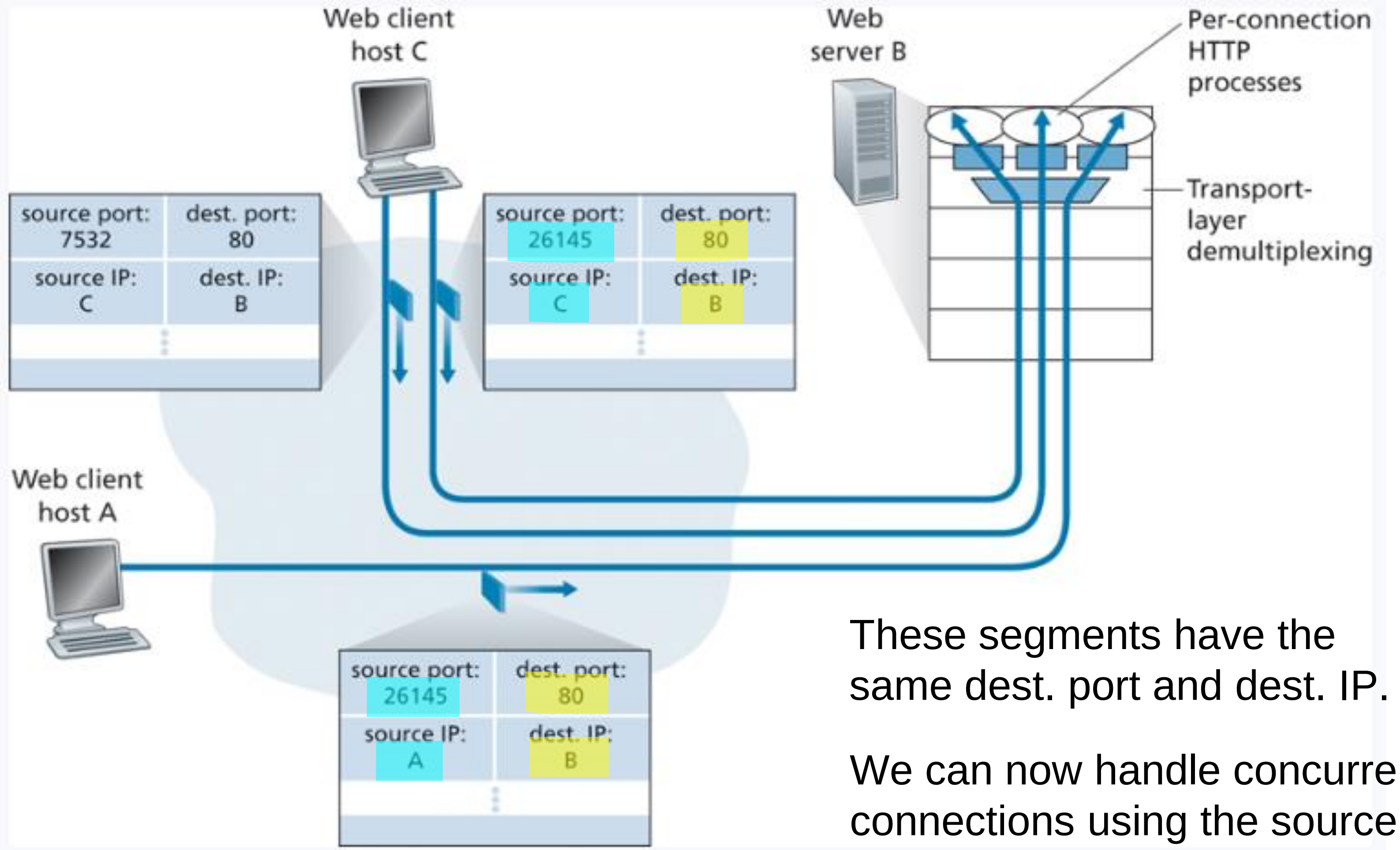


These segments have the same dest. port and dest. IP.

Transport Layer

(source IP address, source port number, destination IP address, destination port number)

TCP sockets are identified by a four-tuple



These segments have the same dest. port and dest. IP.

We can now handle concurrent connections using the source information

Transport Layer

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Secure remote terminal access	SSH	TCP
Web	HTTP, HTTP/3	TCP (for HTTP), UDP (for HTTP/3)
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	DASH	TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

UDP

“Do as little as possible and give it our best effort”

- Bare bones and connectionless

Advantages of UDP:

- Immediate transmission
  - ➢ No connection establishment
- Lower memory requirements
  - ➢ No connection state, 8B for UDP vs 20B for TCP

Disadvantages

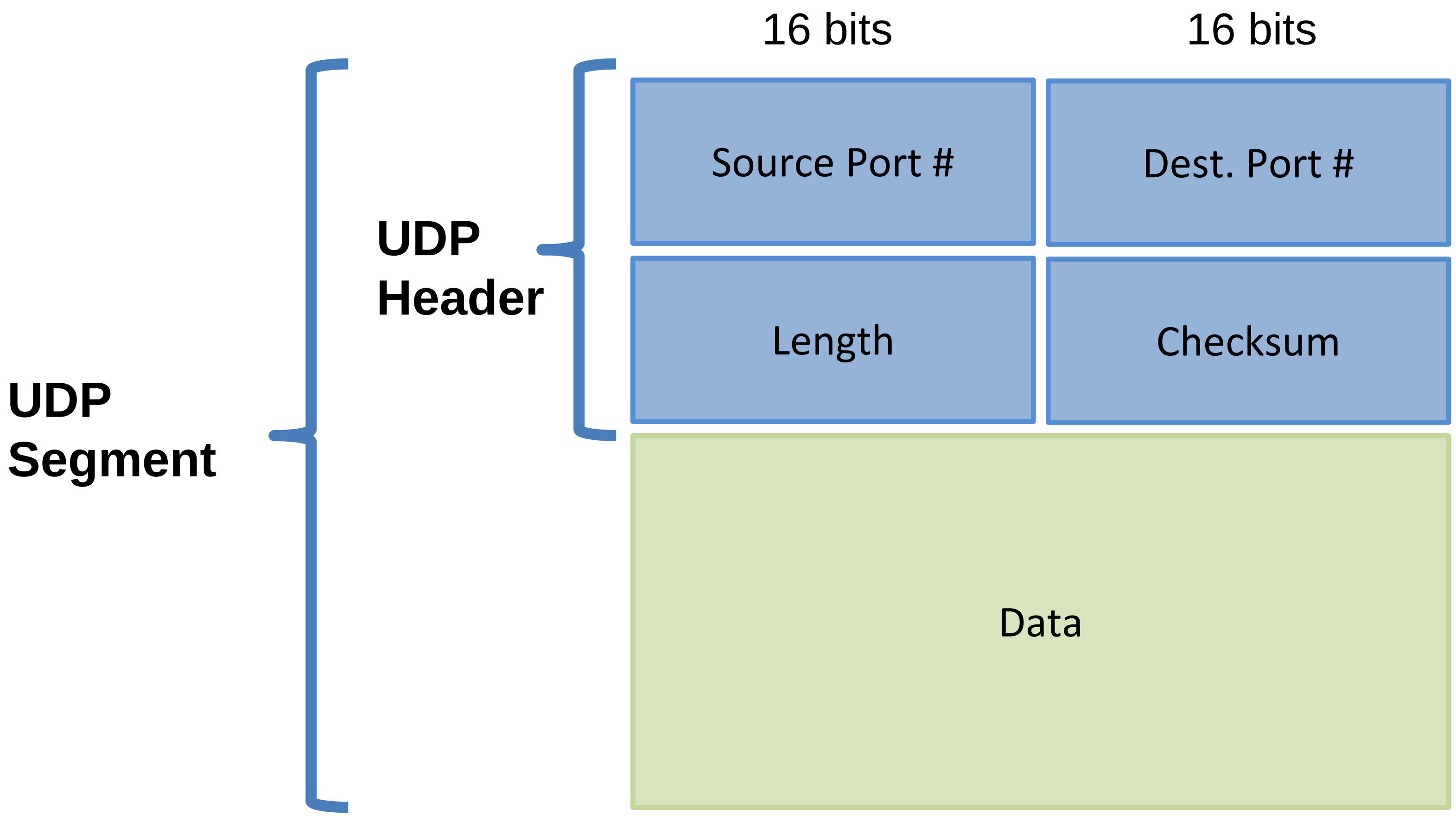
- No congestion control
- No guarantee for in-order delivery
- Reliability and Flow control

QUIC (Quick UDP Internet Connection) is a transport layer protocol that adds reliability to UDP

Transport Layer

UDP

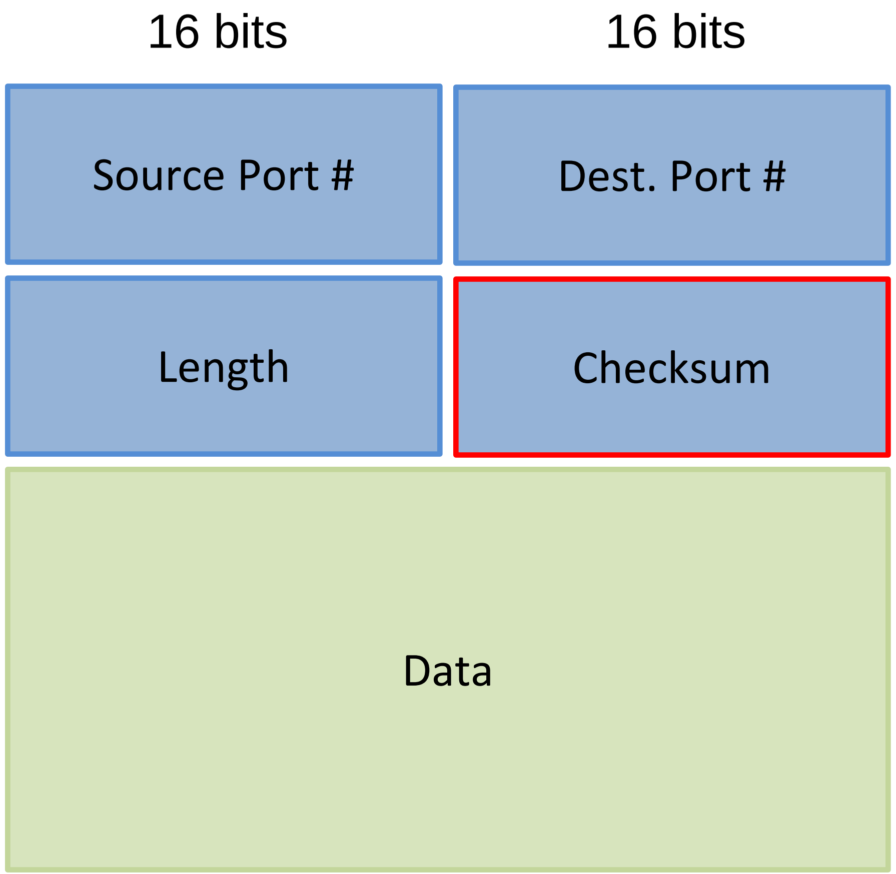
"Do as little as possible and give it our best effort"



UDP “Do as little as possible and give it our best effort”

UDP provides a **checksum** that is used to determine whether bits within the UDP segment have been altered

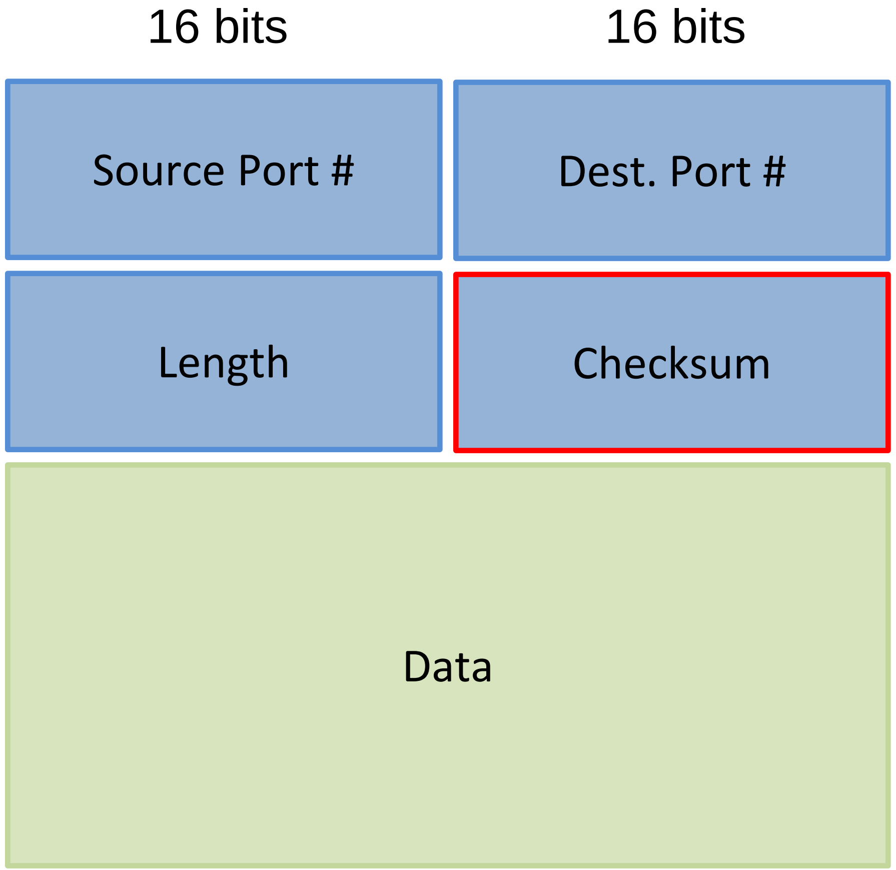
It is the sum of all *n* bit words in the segment followed by the 1s complement



Transport Layer

UDP “Do as little as possible and give it our best effort”

0110011001100000
0101010101010101
1000111100001100



Transport Layer

UDP “Do as little as possible and give it our best effort”

0110011001100000
0101010101010101
1000111100001100

0110011001100000
0101010101010101

1011101110110101

16 bits	16 bits
Source Port #	Dest. Port #
Length	Checksum
Data	

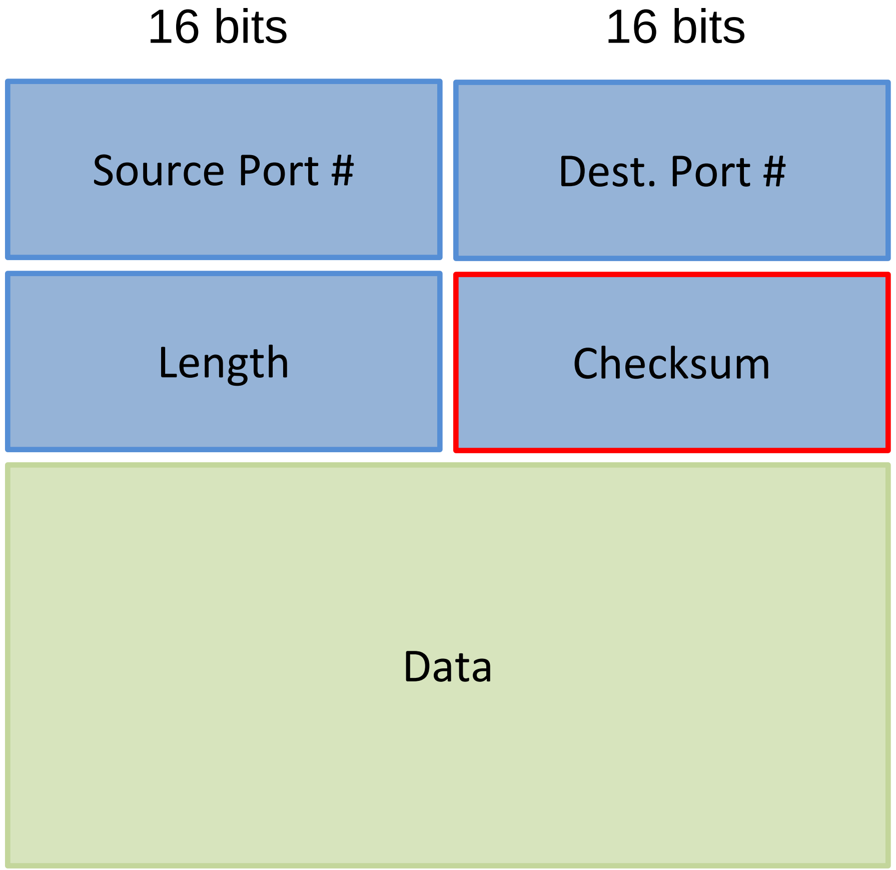
Transport Layer

UDP “Do as little as possible and give it our best effort”

0110011001100000
0101010101010101
1000111100001100

1011101110110101
1000111100001100

<u>0100101011000010</u>



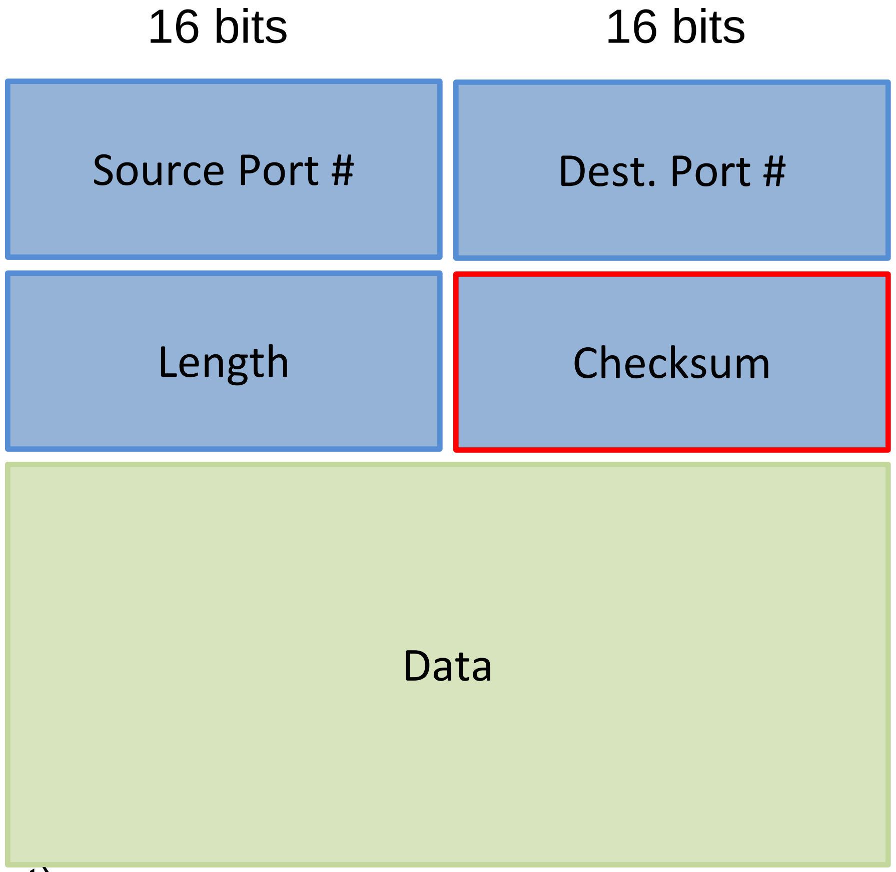
Transport Layer

UDP “Do as little as possible and give it our best effort”

0110011001100000
0101010101010101
1000111100001100

1011101110110101
1000111100001100

0100101011000010 (one’s complement)
1011010100111101 = checksum



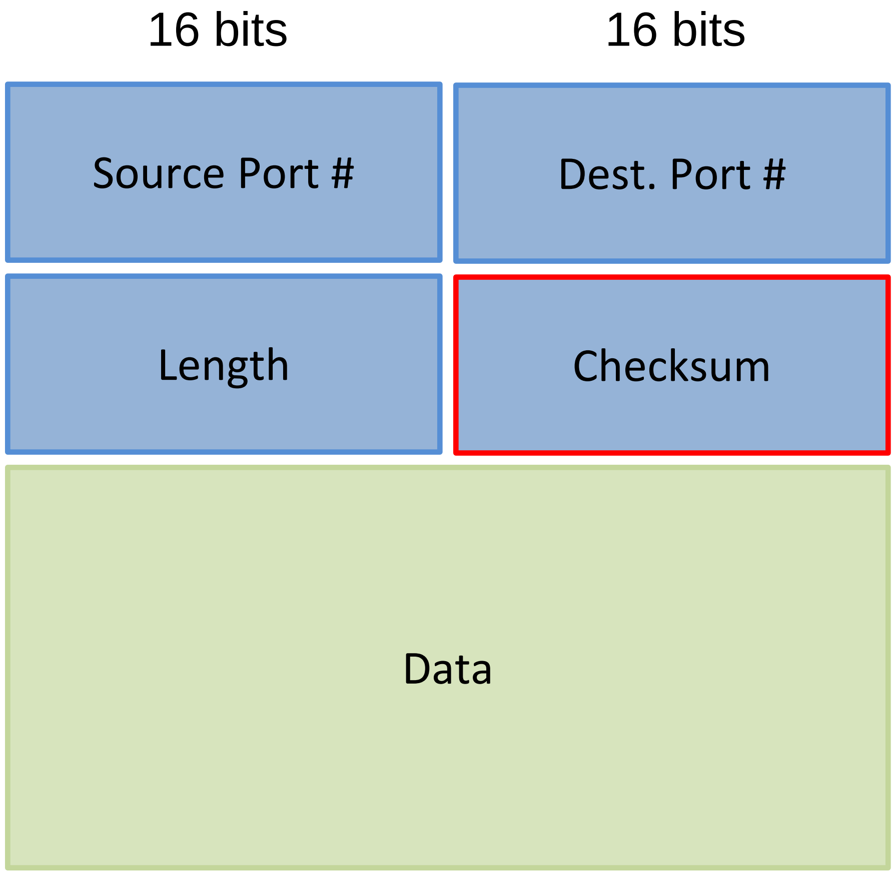
Transport Layer

UDP “Do as little as possible and give it our best effort”

0110011001100000
0101010101010101
1000111100001100
<u>1011010100111101</u>

Receiving packet: Sum up all 16-bit words → **1111111111111111**

If there are zeros, errors were introduced into the packet



Transport Layer

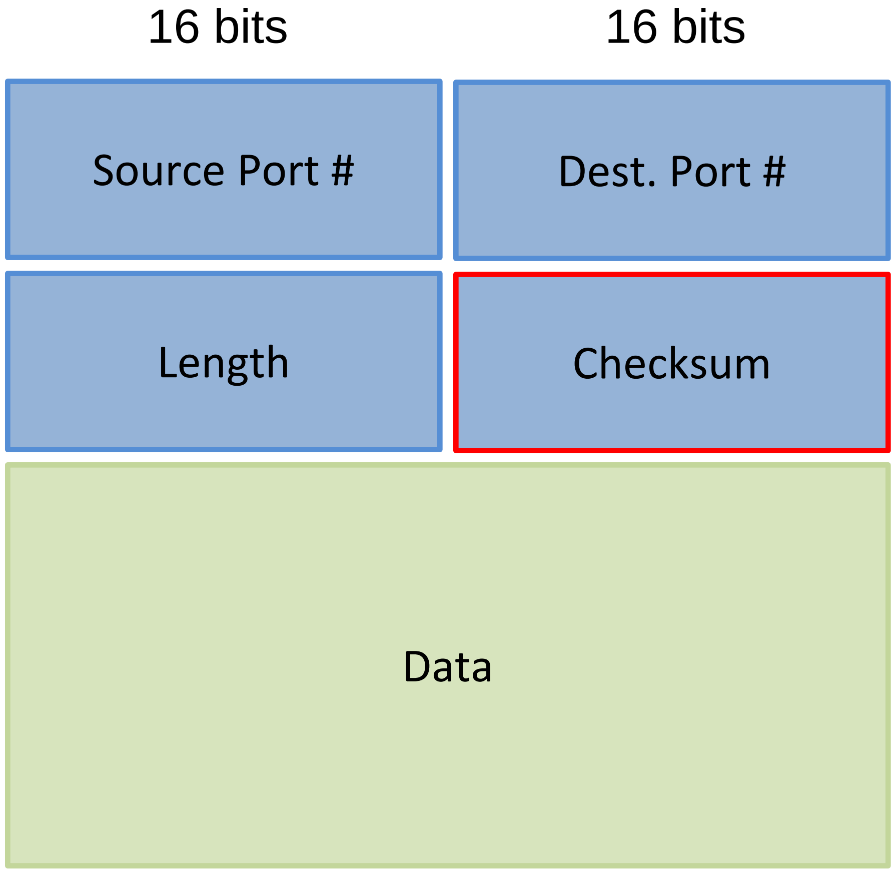
UDP

"Do as little as possible and give it our best effort"

Why do error checking here?

End-to-end principle states that since certain functionality such as error detection, must be implemented on an end-end bases

Functionality places at the lower levels may be redundant or of little value when compared to the cost of providing them at a higher level



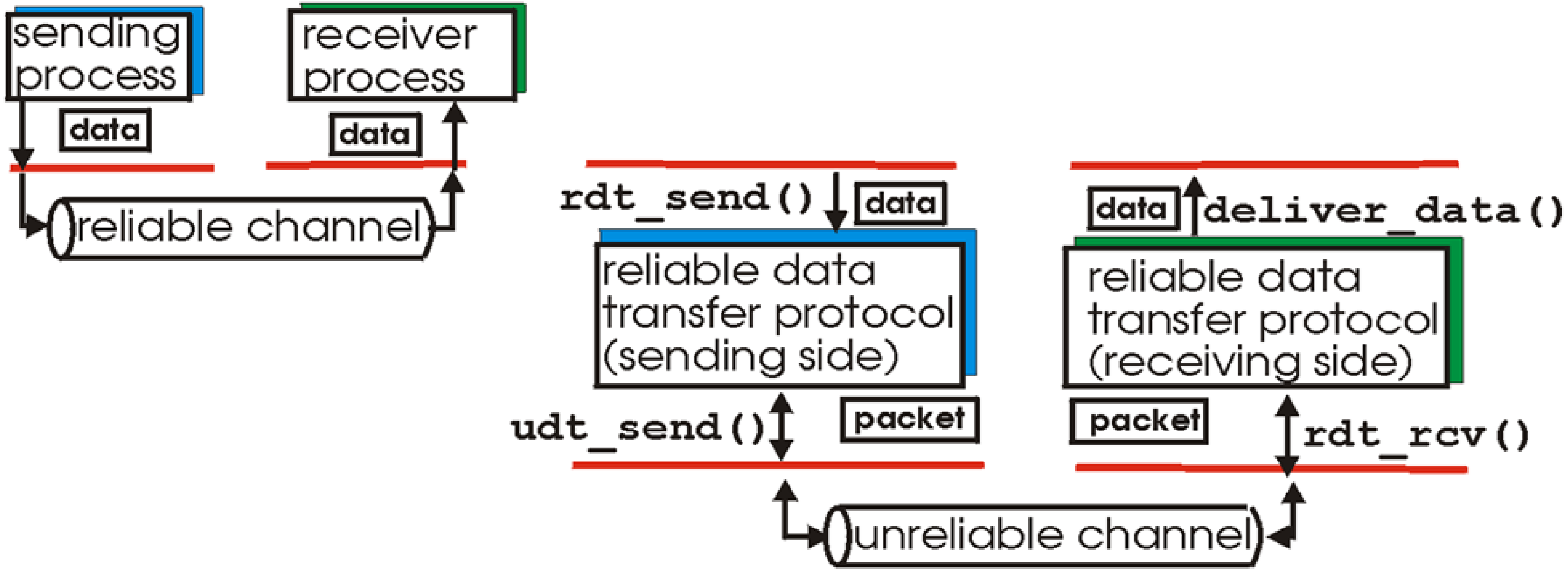
Transport Layer

Reliable Data Transfer

Application

Transport

Network



Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

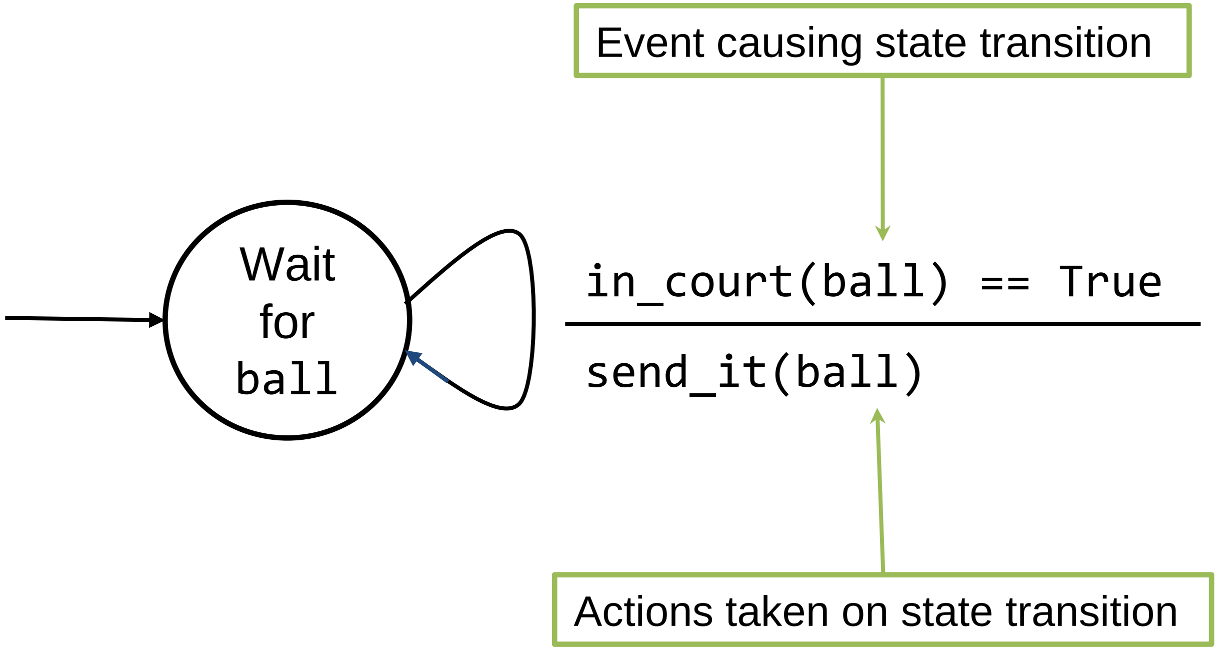
What are some ways in which the network channel can be unreliable?

Things to consider:
Corruption
Loss of packets
Duplicate delivery

Reliable Data Transfer

Reliable Data Transfer

Bruce Lee FSM



Reliable Data Transfer 1.0

RDT 1.0

Assumptions:

- Unidirectional long data flows
- Perfectly reliable channel
- No bit errors
- No packet loss
- No packet reordering

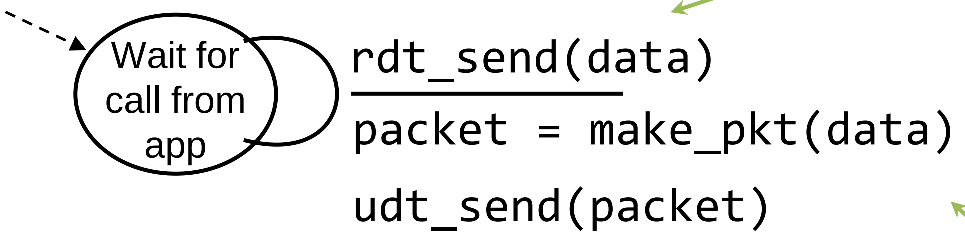
Reliable Data Transfer 1.0

RDT 1.0

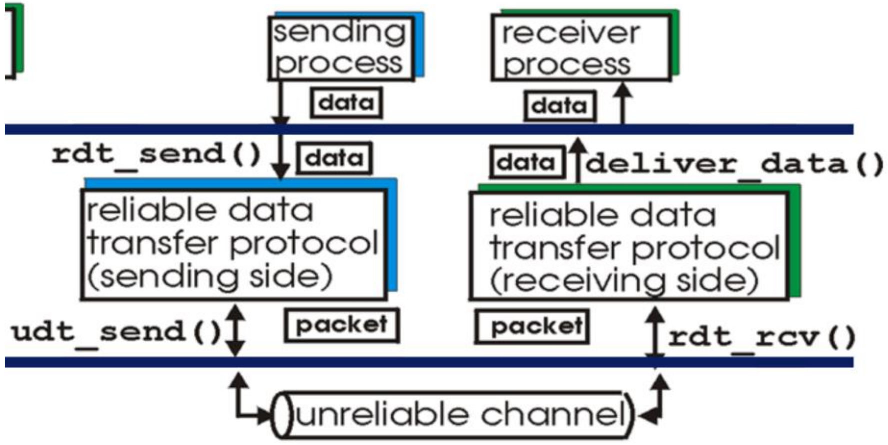
Assumptions:
- Unidirectional long data flows
- Perfectly reliable channel
- No bit errors
- No packet loss
- No packet reordering

Event causing state transition

Sender



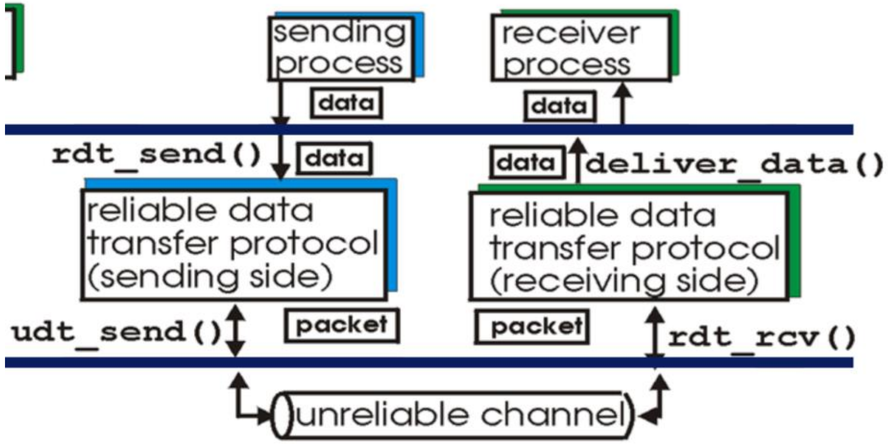
Actions taken on state transition



Reliable Data Transfer 1.0

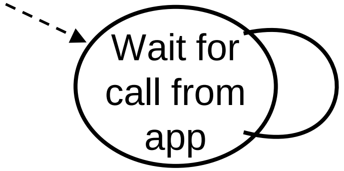
RDT 1.0

Assumptions:
- Unidirectional long data flows
- Perfectly reliable channel
- No bit errors
- No packet loss
- No packet reordering



Event causing state transition

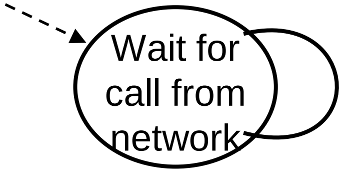
Sender




```
rdt_send(data)
_______________
packet = make_pkt(data)
udt_send(packet)
```

Actions taken on state transition

Receiver



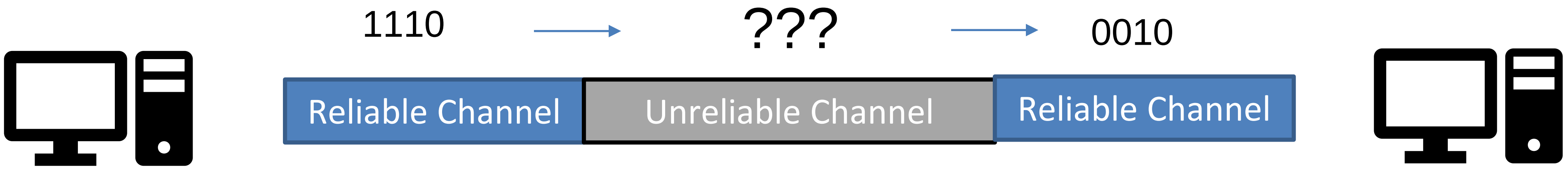

```
rdt_rcv(packet)
_______________
data = extract(packet)
deliver_data(data)
```

Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

How can we detect errors?

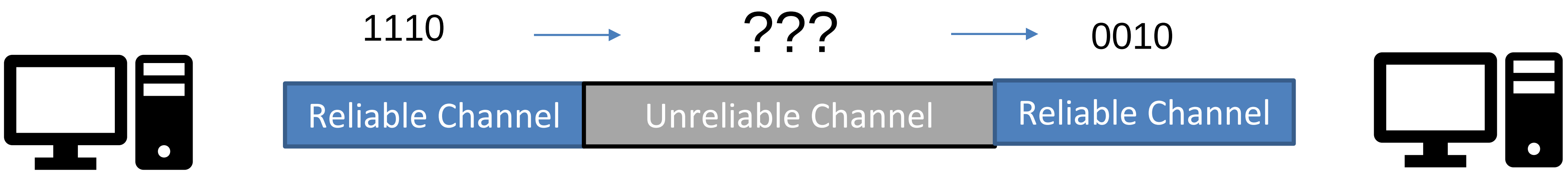


Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

How can we detect errors?
-Checksum



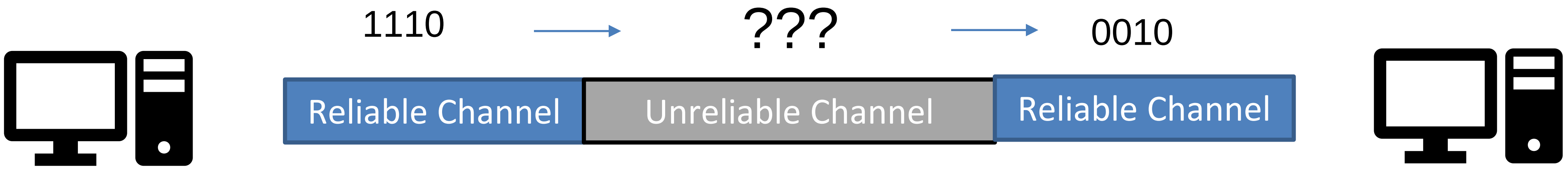
Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

How can we detect errors?
-Checksum

What is a good way to handle and prevent errors?



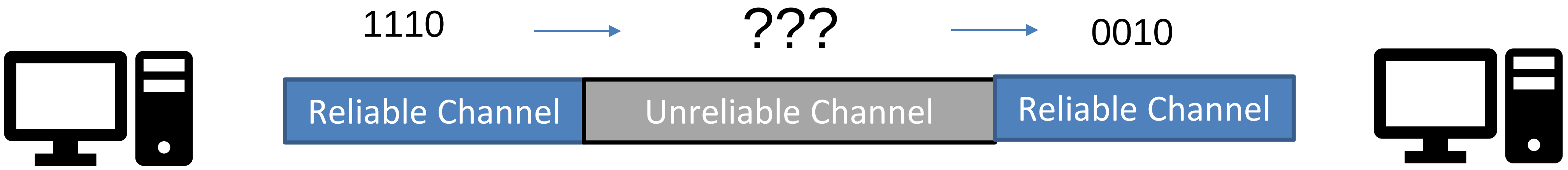
Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

How can we detect errors?
-Checksum

What is a good way to handle and prevent errors?
- Acknowledged packet, Ask for retransmit if needed



Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

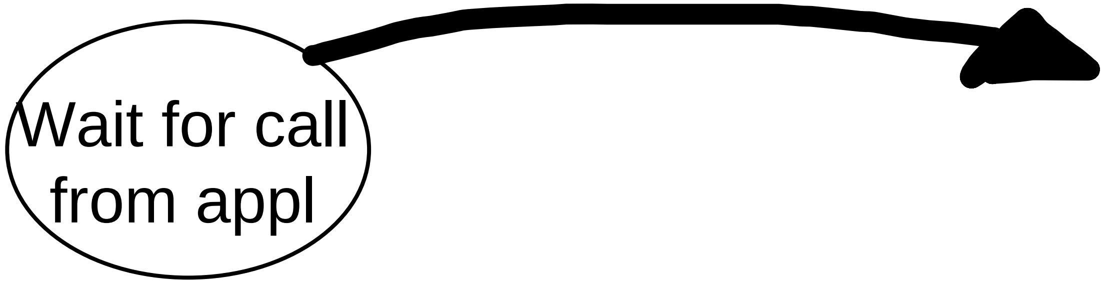
How can we detect errors?
-Checksum

What is a good way to handle and prevent errors?
- Acknowledged packet, Ask for retransmit if needed

Stop-and-wait: sender sends one packet, then waits for receiver response

Sender

```
rdt_send(data)
________________________________
sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)
```



Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

How can we detect errors?
-Checksum

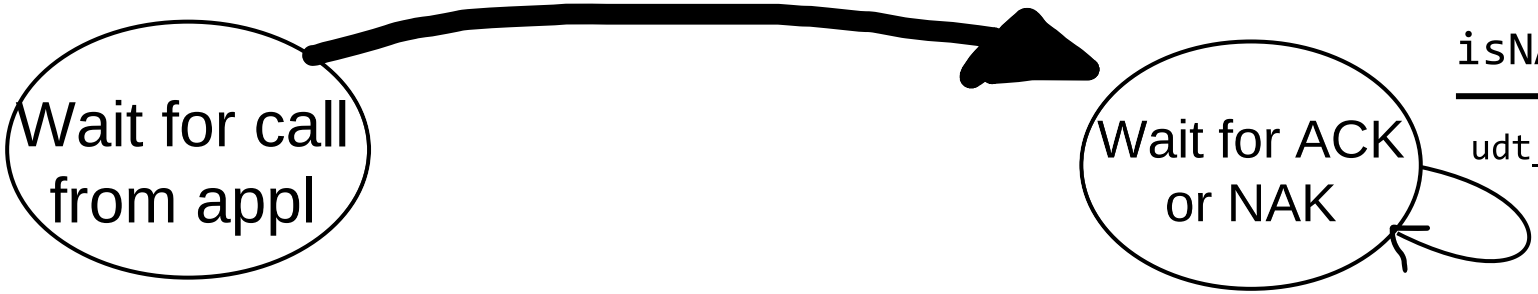
What is a good way to handle and prevent errors?
- Acknowledged packet, Ask for retransmit if needed

Stop-and-wait: sender sends one packet, then waits for receiver response

Sender

```
rdt_send(data)
_____________________________
sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____________________________
udt_send(sndpkt)
```



Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

How can we detect errors?
-Checksum

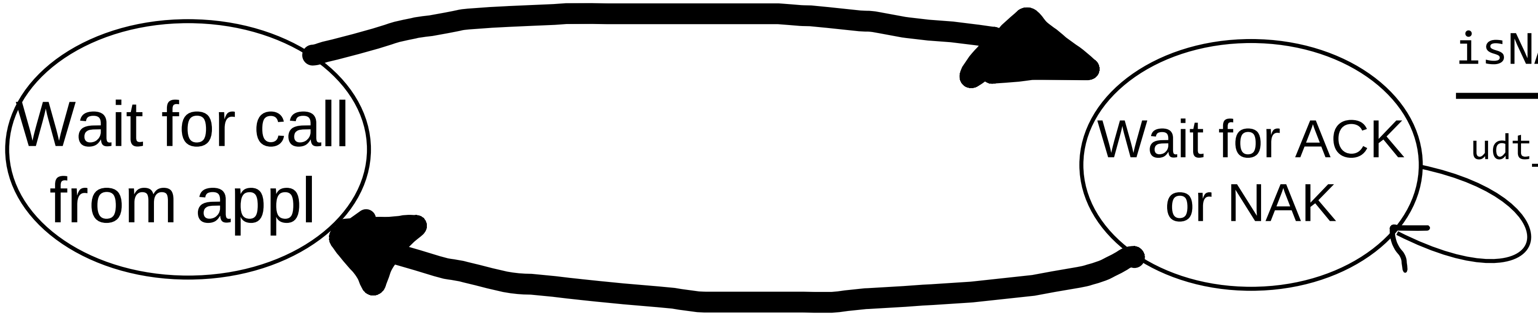
What is a good way to handle and prevent errors?
- Acknowledged packet, Ask for retransmit if needed

Stop-and-wait: sender sends one packet, then waits for receiver response

Sender

```
rdt_send(data)
________________________________
sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
________________________________
udt_send(sndpkt)
```




```
rdt_rcv(rcvpkt) && isACK(rcvpkt)
__________
Λ
```

Reliable Data Transfer 2.0

RDT 2.0

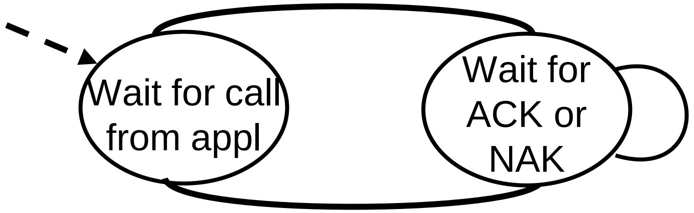
Potential for bit errors

How can we detect errors?
-Checksum

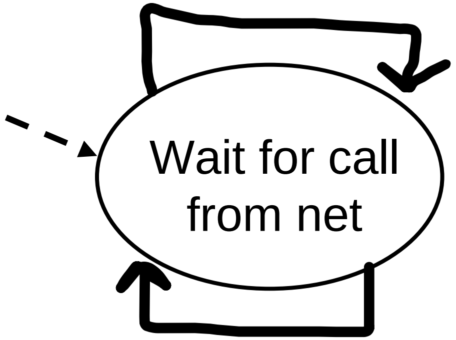
What is a good way to handle and prevent errors?
- Acknowledged packet, Ask for retransmit if needed

Stop-and-wait: sender sends one packet, then waits for receiver response

Sender



Receiver



Transport Layer

Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

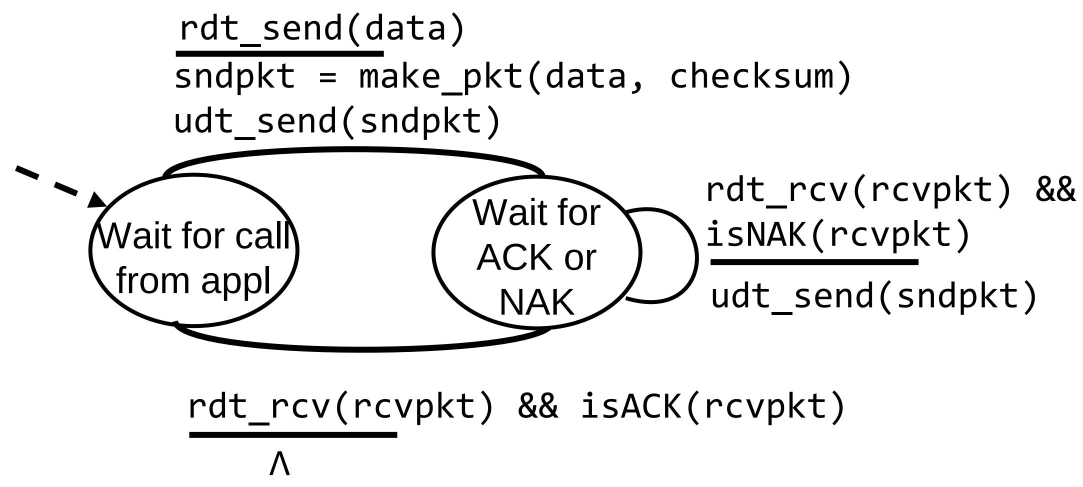
How can we detect errors?
-Checksum

What is a good way to handle and prevent errors?
- Acknowledged packet, Ask for retransmit if needed

Stop-and-wait: sender sends one packet, then waits for receiver response

Receiver

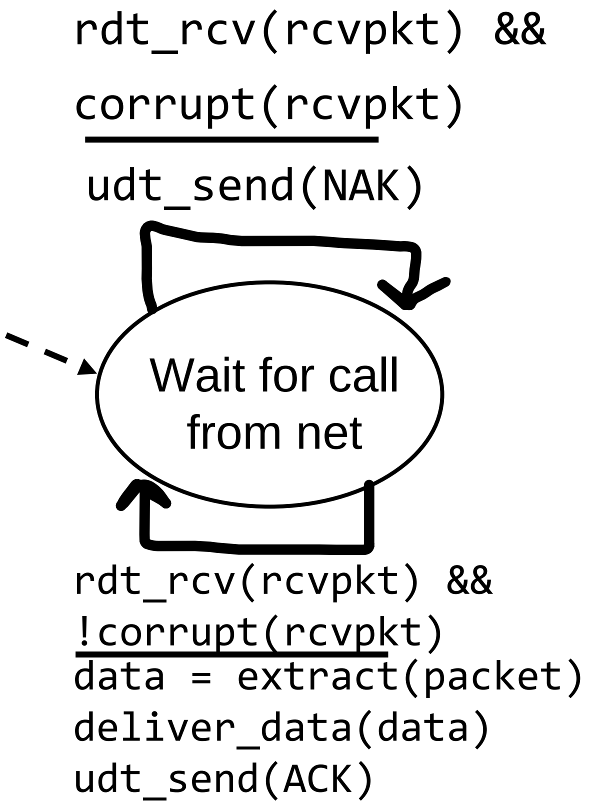
Sender



What happens if ACK/NAK Corrupted?
→ Duplicate delivery, or no retransmission

Solution?
→ Retransmit is CORRUPT packet received

How to deal we duplicate Packets?
→ ???



Reliable Data Transfer 2.0

RDT 2.0

Potential for bit errors

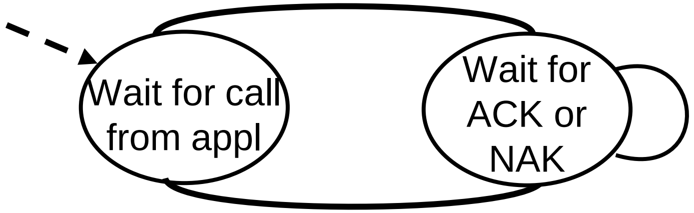
How can we detect errors?
-Checksum

What is a good way to handle and prevent errors?
- Acknowledged packet, Ask for retransmit if needed

Stop-and-wait: sender sends one packet, then waits for receiver response

Sender

rdt_send(data)
sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)



rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ

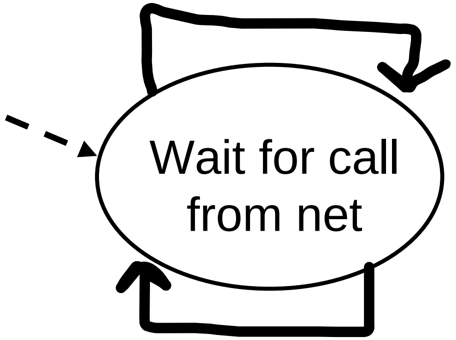
What happens if ACK/NAK Corrupted?
→ Duplicate delivery, or no retransmission

Solution?
→ Retransmit is CORRUPT packet received

How to deal we duplicate Packets?
→ Sequence Number

Receiver

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)



rdt_rcv(rcvpkt) && !corrupt(rcvpkt)
data = extract(packet)
deliver_data(data)
udt_send(ACK)

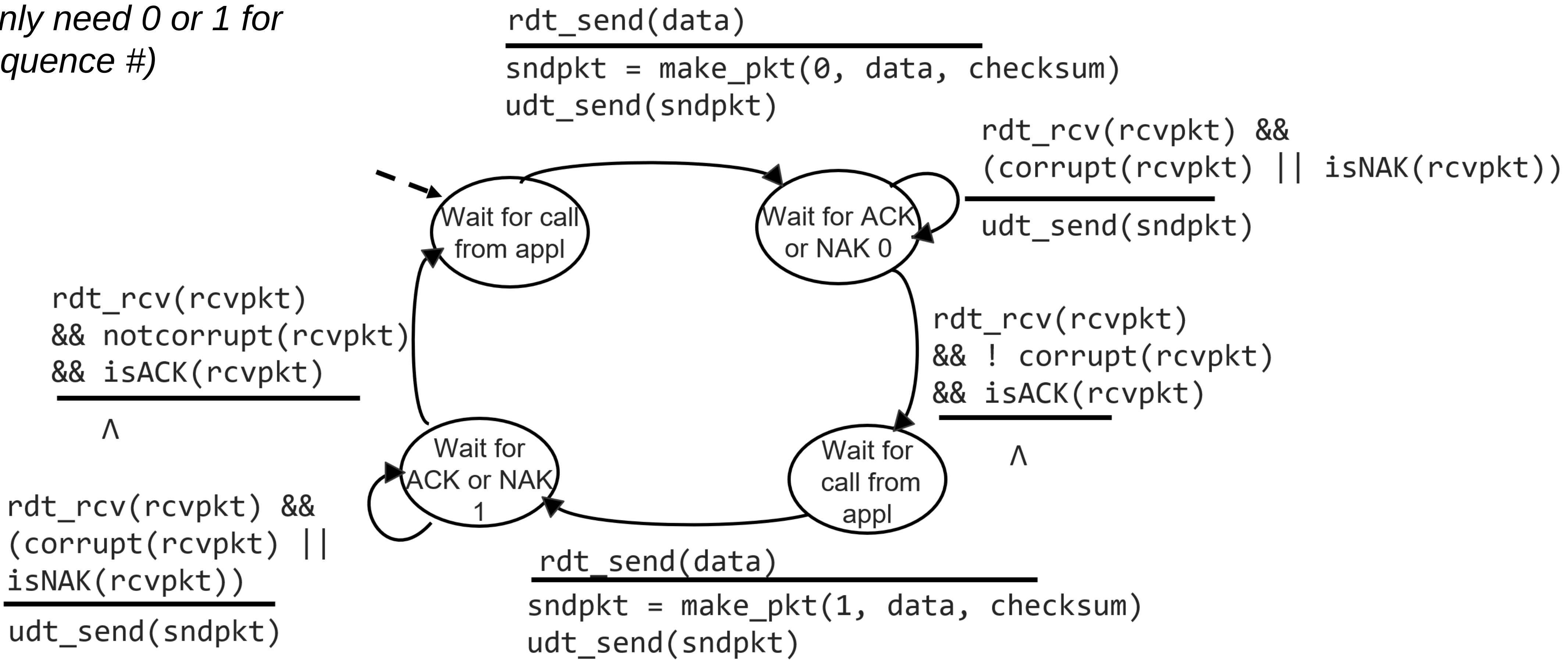
Reliable Data Transfer 2.1

RDT 2.1

Potential for bit errors and garbled ACKs

Stop-and-wait: sender sends one packet, then waits for receiver response

(We only need 0 or 1 for the sequence #)



Reliable Data Transfer 2.2

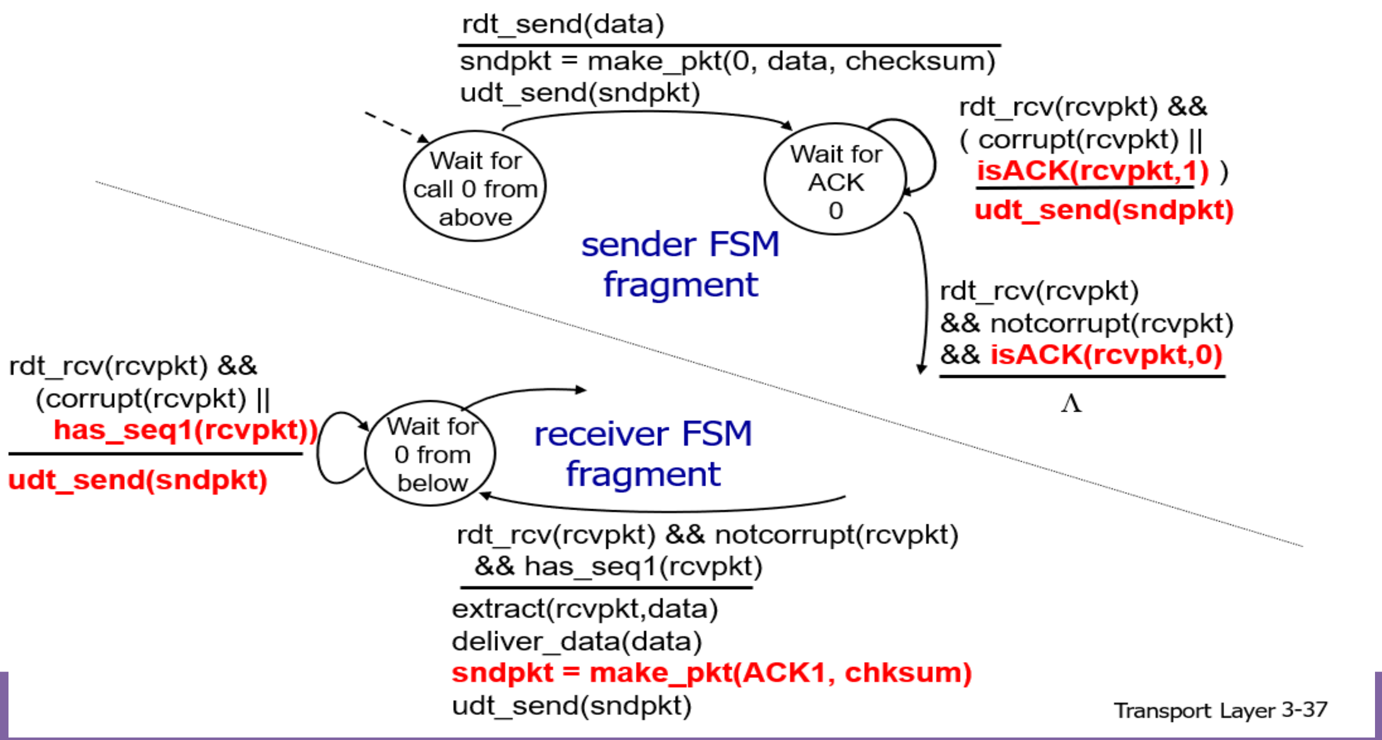
RDT 2.2

Same as rdt2.1, using only ACKs

Instead of NAK, receiver send the ACK for last pkt received successfully.

A.K.A Receiver must explicitly include seq # of pkt being ACKed

Duplicate ACK at sender results in same action as NAK: retransmit current pkt



Reliable Data Transfer 3.0

RDT 3.0

Packets can get lost or dropped

We will still need checksums, seq #, ACKS, but we need more

What if an ACK gets dropped? Sender is stuck waiting forever

Solution?

Reliable Data Transfer 3.0

RDT 3.0

Packets can get lost or dropped

We will still need checksums, seq #, ACKS, but we need more

What if an ACK gets dropped? Sender is stuck waiting forever

Sender should wait a “reasonable” amount of time for an ack

A timer!!!

Retransmit if no ACK received in X amount of time

What if the ACK is just taking a really long time to arrive?

Reliable Data Transfer 3.0

RDT 3.0

Packets can get lost or dropped

We will still need checksums, seq #, ACKS, but we need more

What if an ACK gets dropped? Sender is stuck waiting forever

Sender should wait a “reasonable” amount of time for an ack

A timer!!!

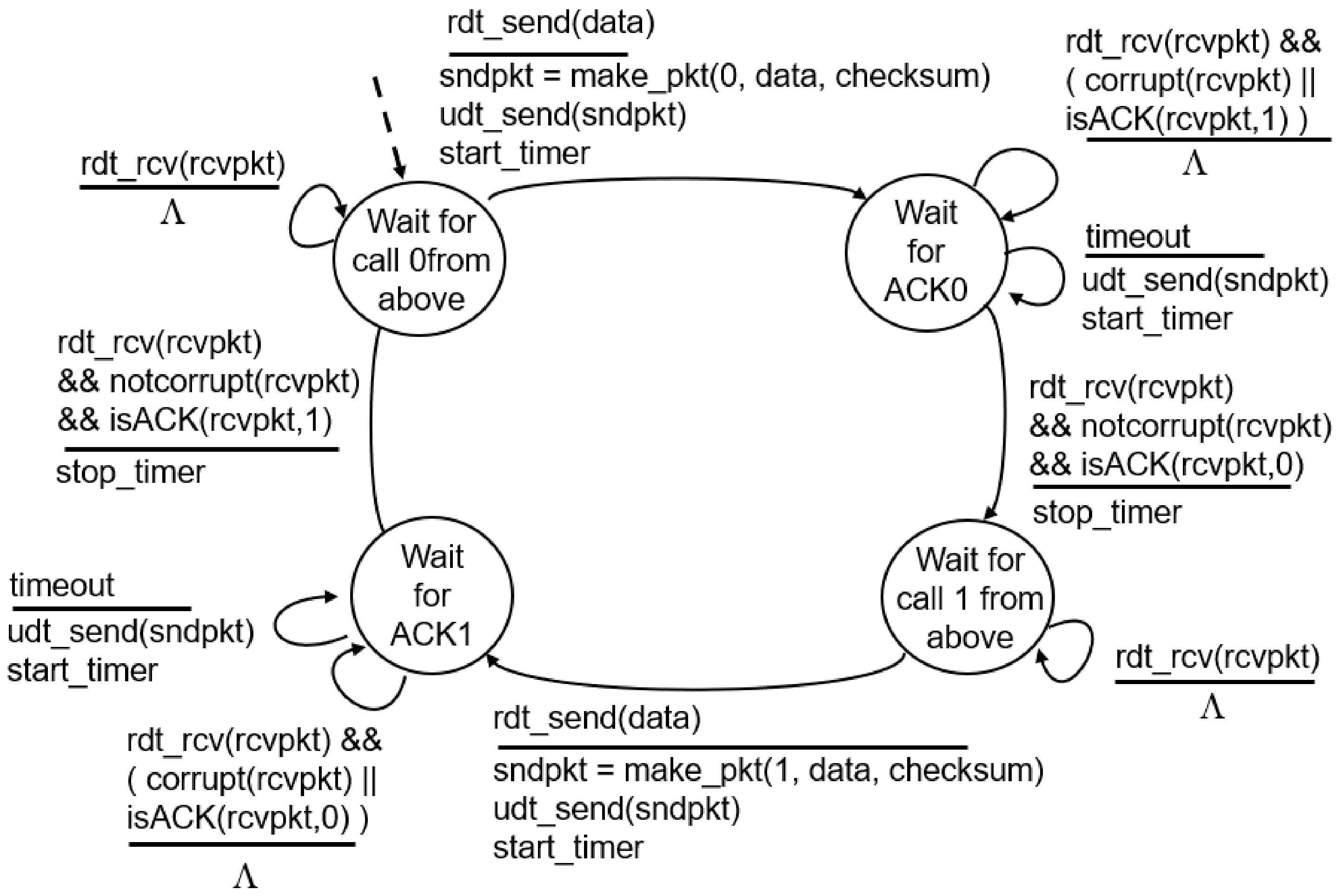
Retransmit if no ACK received in X amount of time

What if the ACK is just taking a really long time to arrive? This will be duplicate data, but we have Seq # to handle that.

Reliable Data Transfer 3.0

RDT 3.0

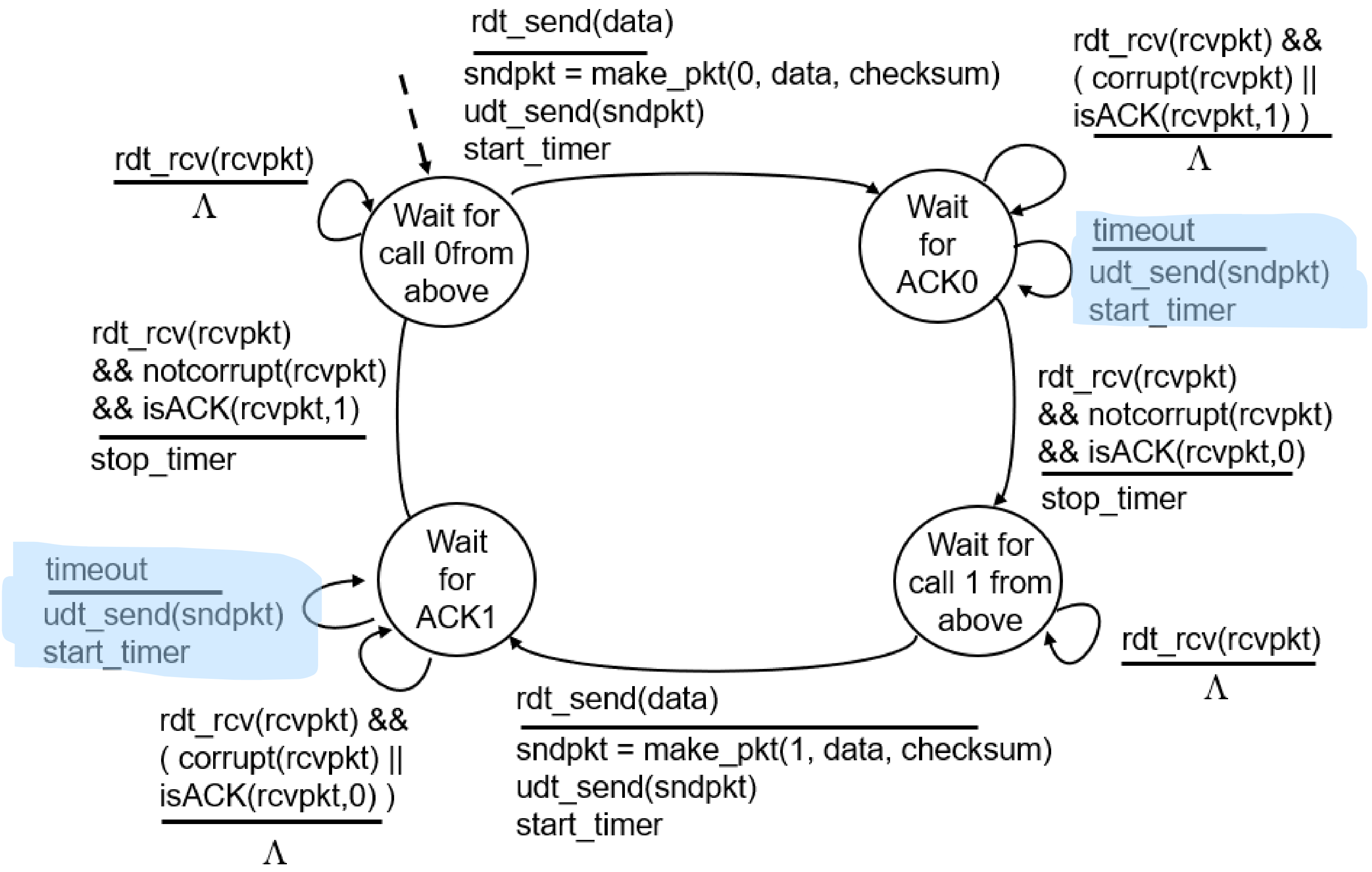
Packets can get lost or dropped



Reliable Data Transfer 3.0

RDT 3.0

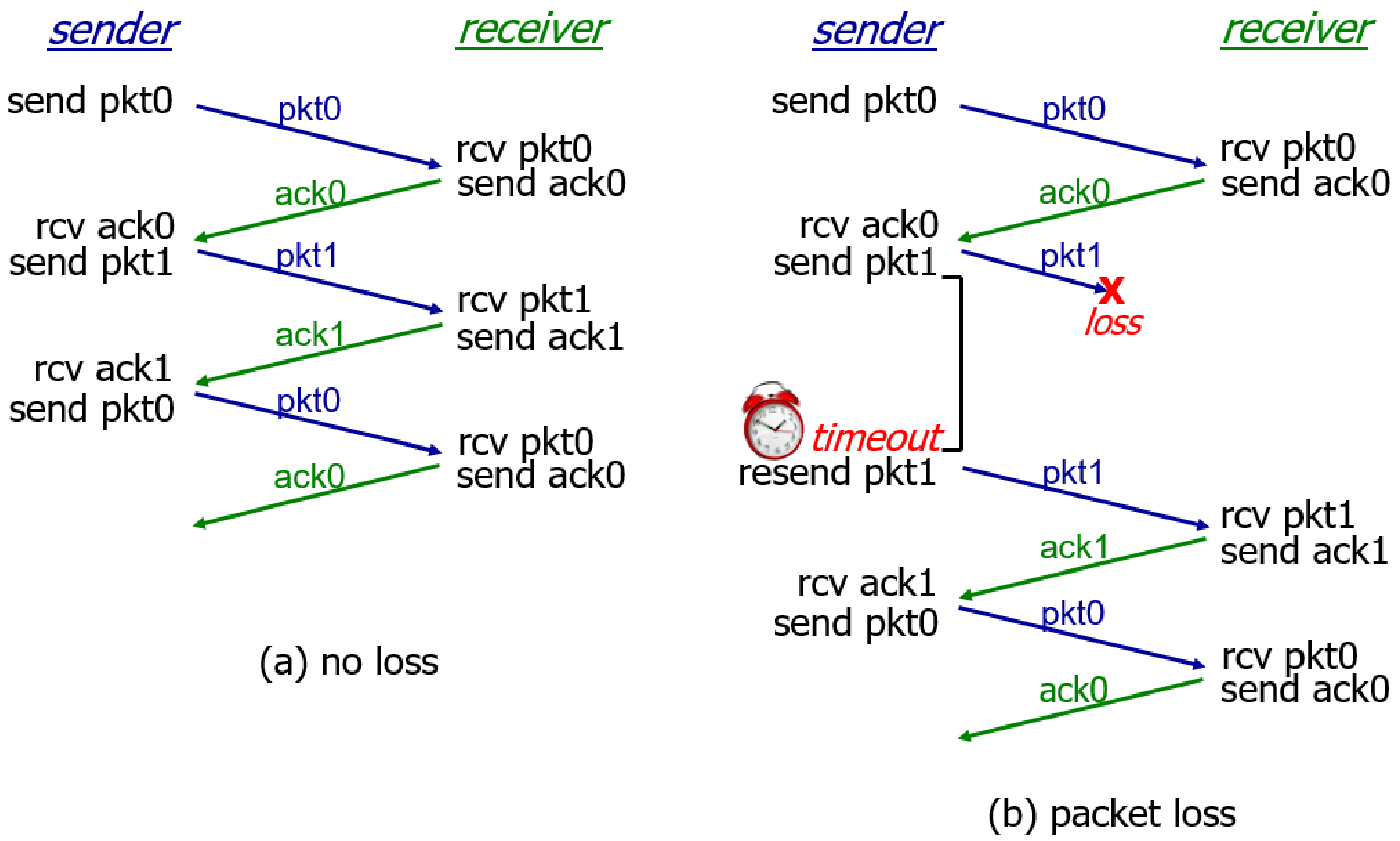
Packets can get lost or dropped



Reliable Data Transfer 3.0

RDT 3.0

Packets can get lost or dropped

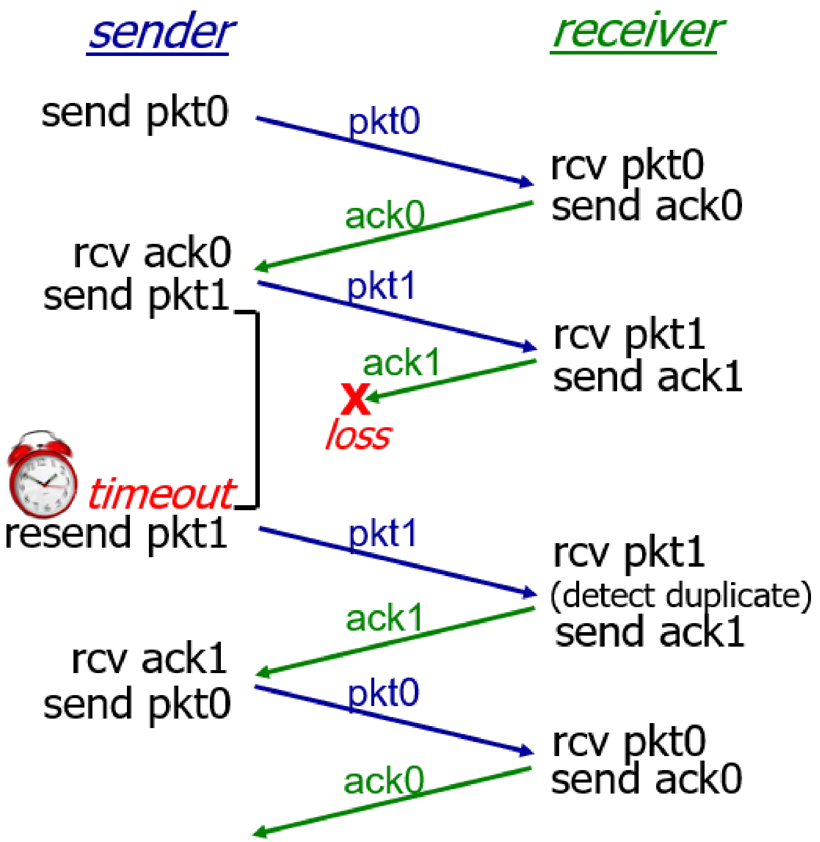


Transport Layer

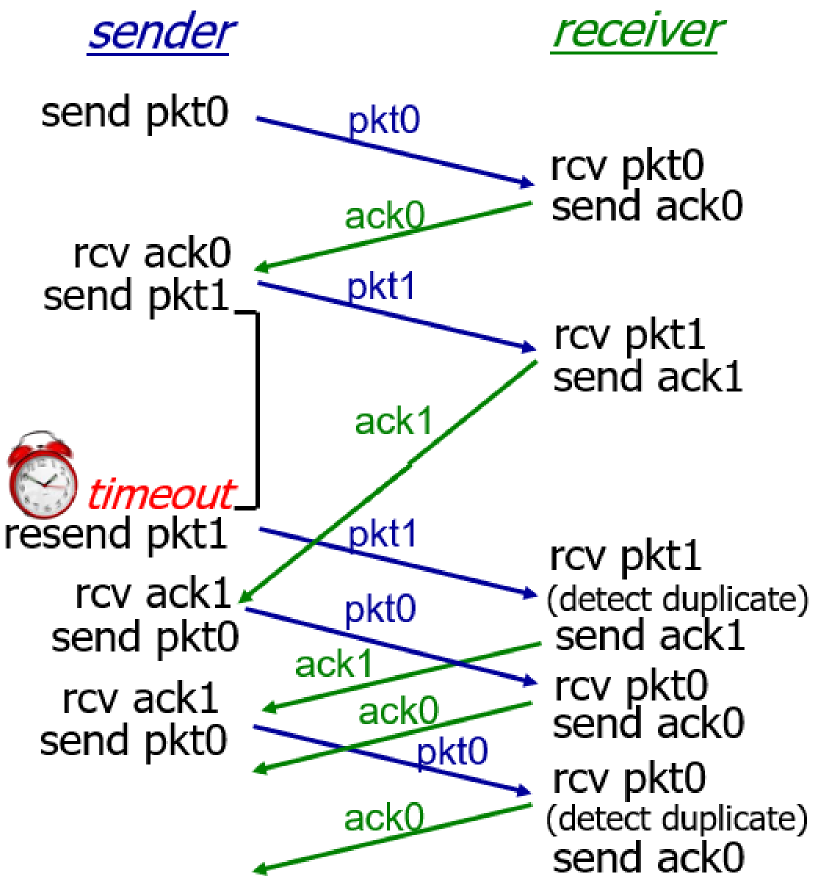
Reliable Data Transfer 3.0

RDT 3.0

Packets can get lost or dropped

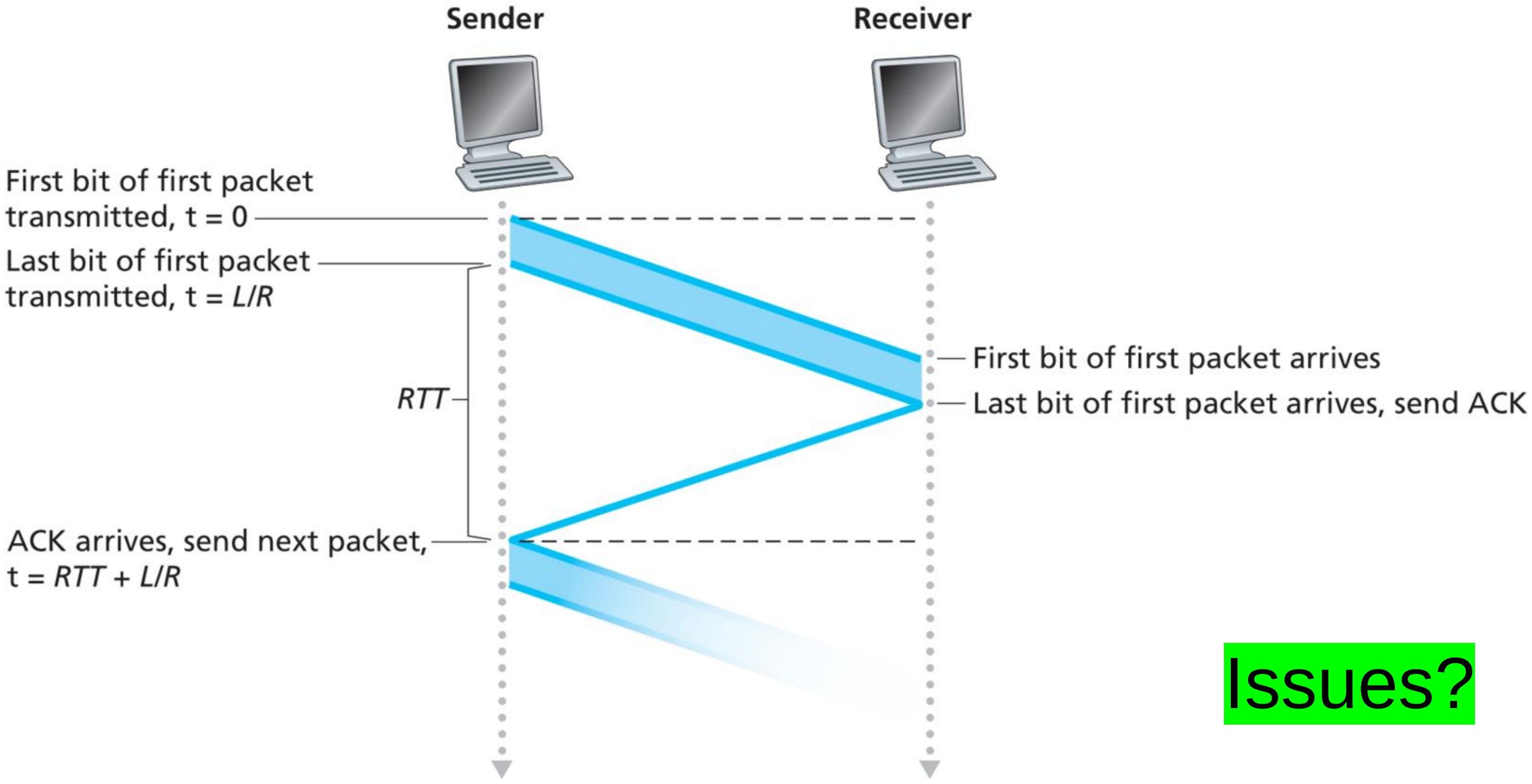


(c) ACK loss



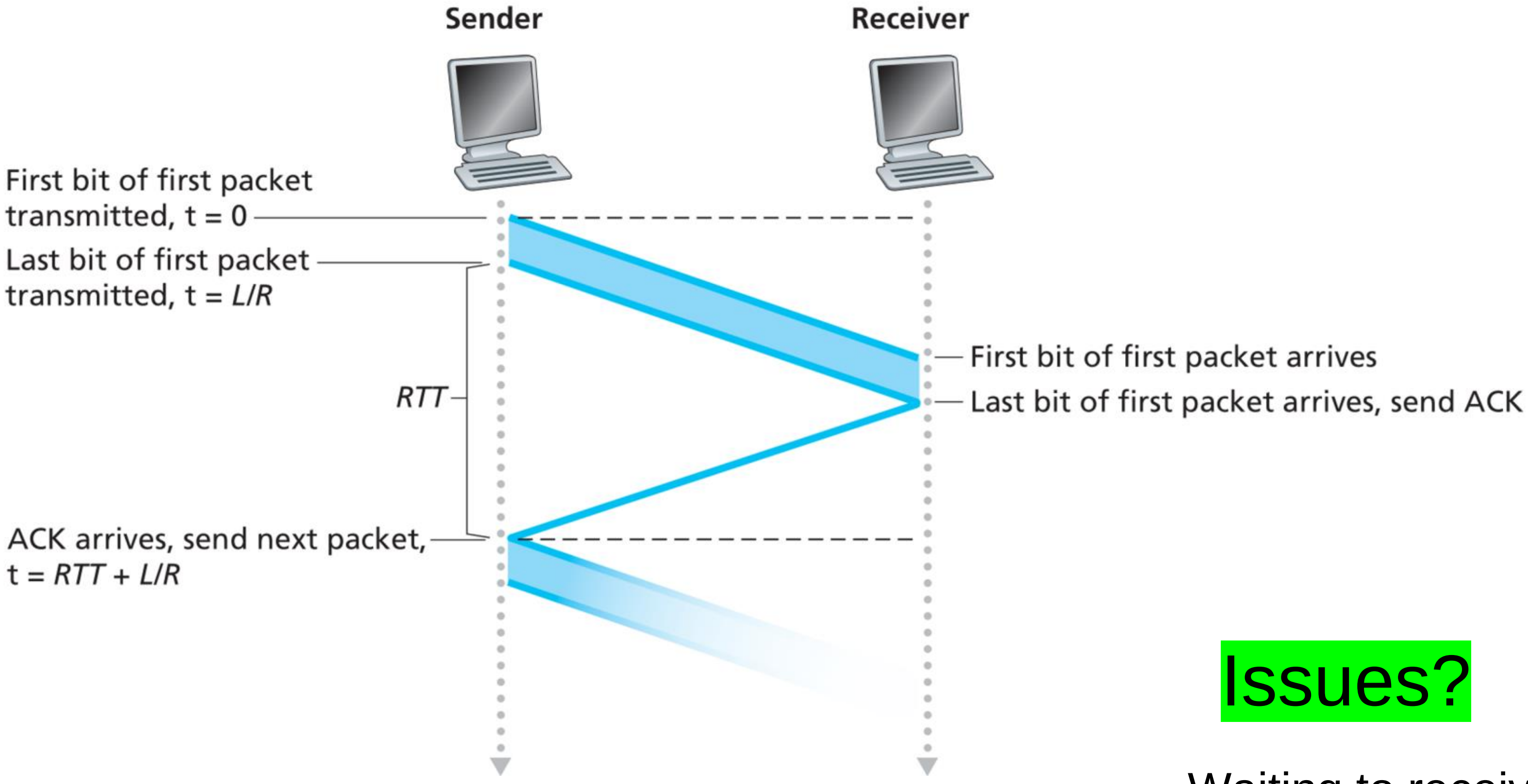
(d) premature timeout/ delayed ACK

Stop-and-wait Protocols?



Issues?

Stop-and-wait Protocols?

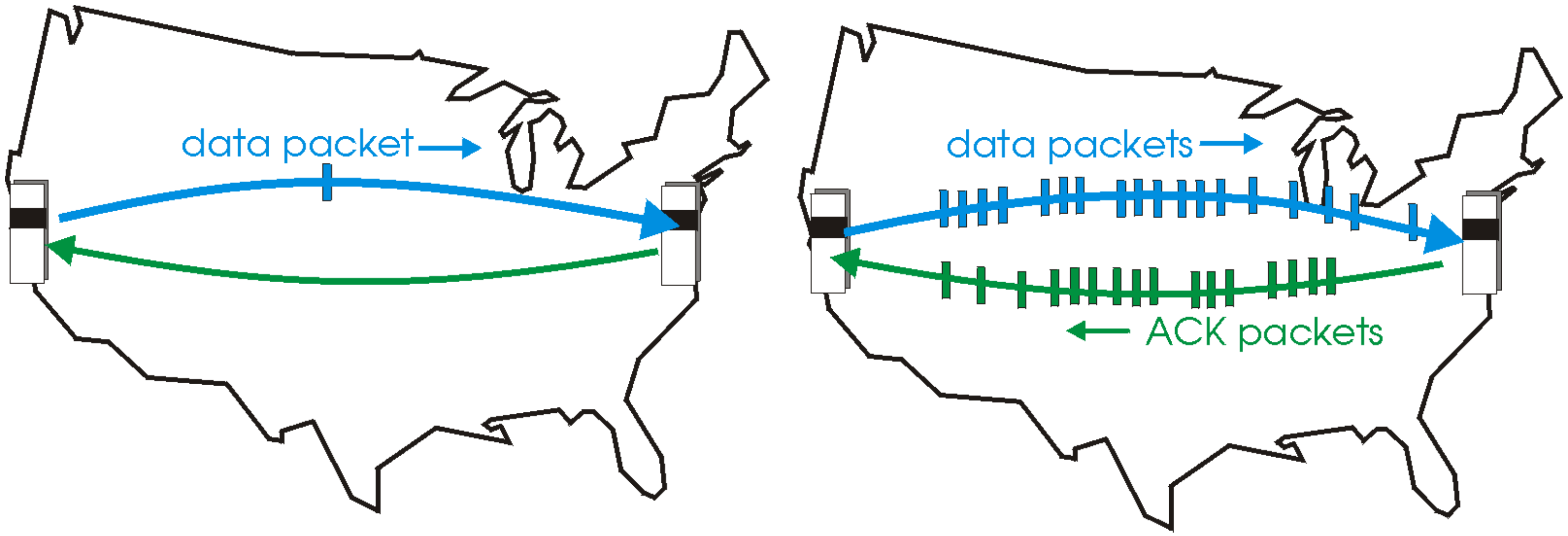


Issues?

Waiting to receive an ACK is inefficient

Pipelining: Increased Utilization

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledges pkts

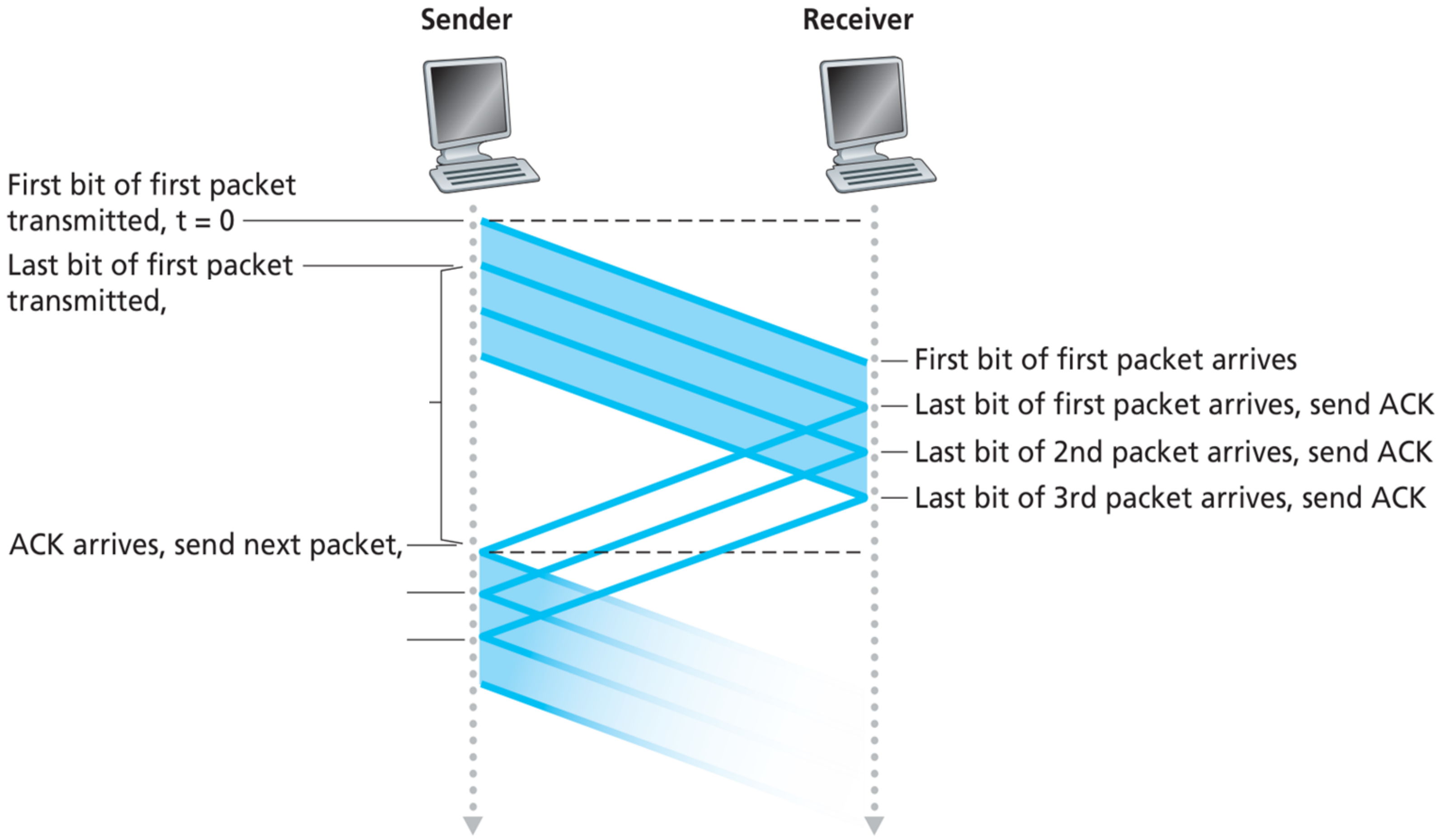


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

Pipelining: Increased Utilization

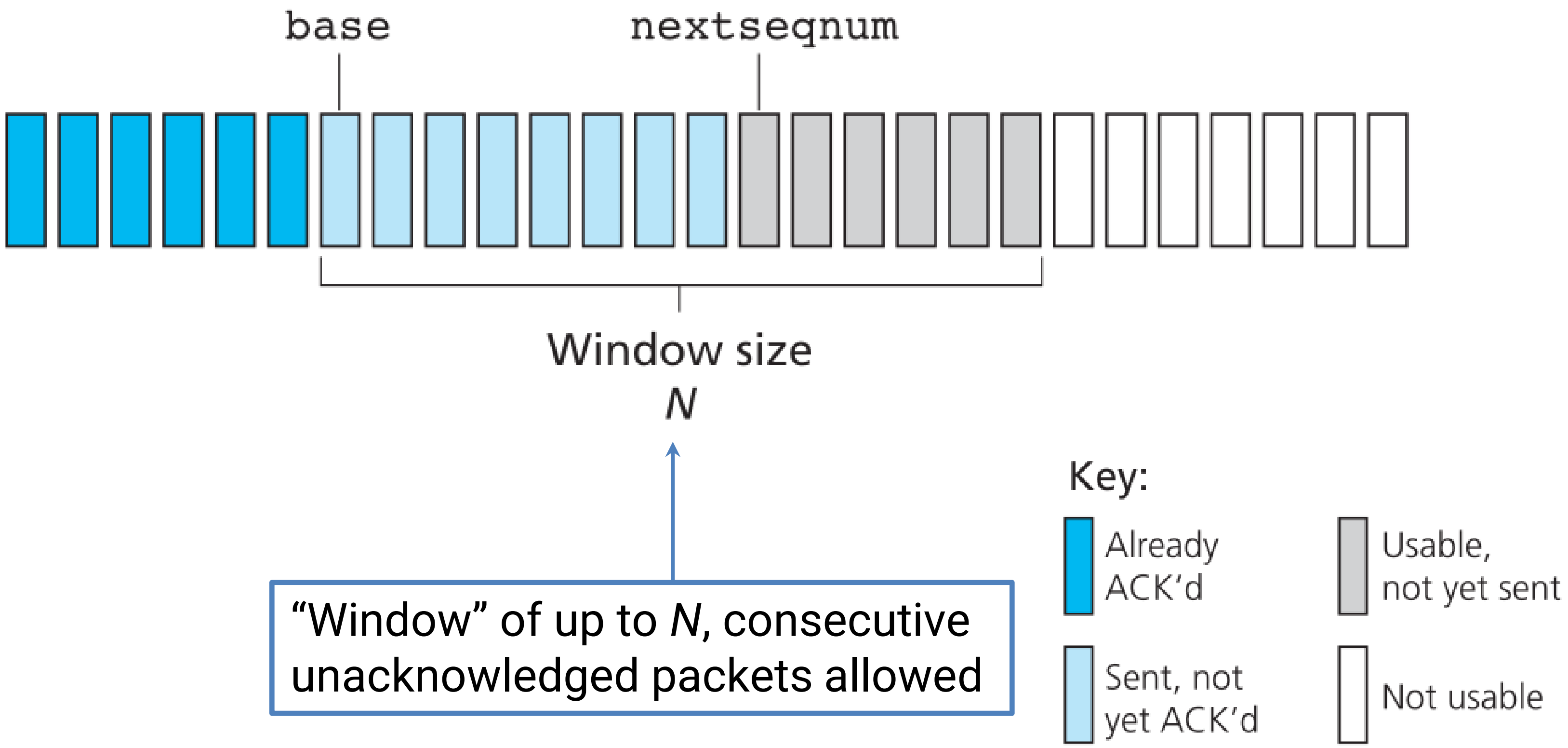
Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledges pkts



Pipelining: Increased Utilization

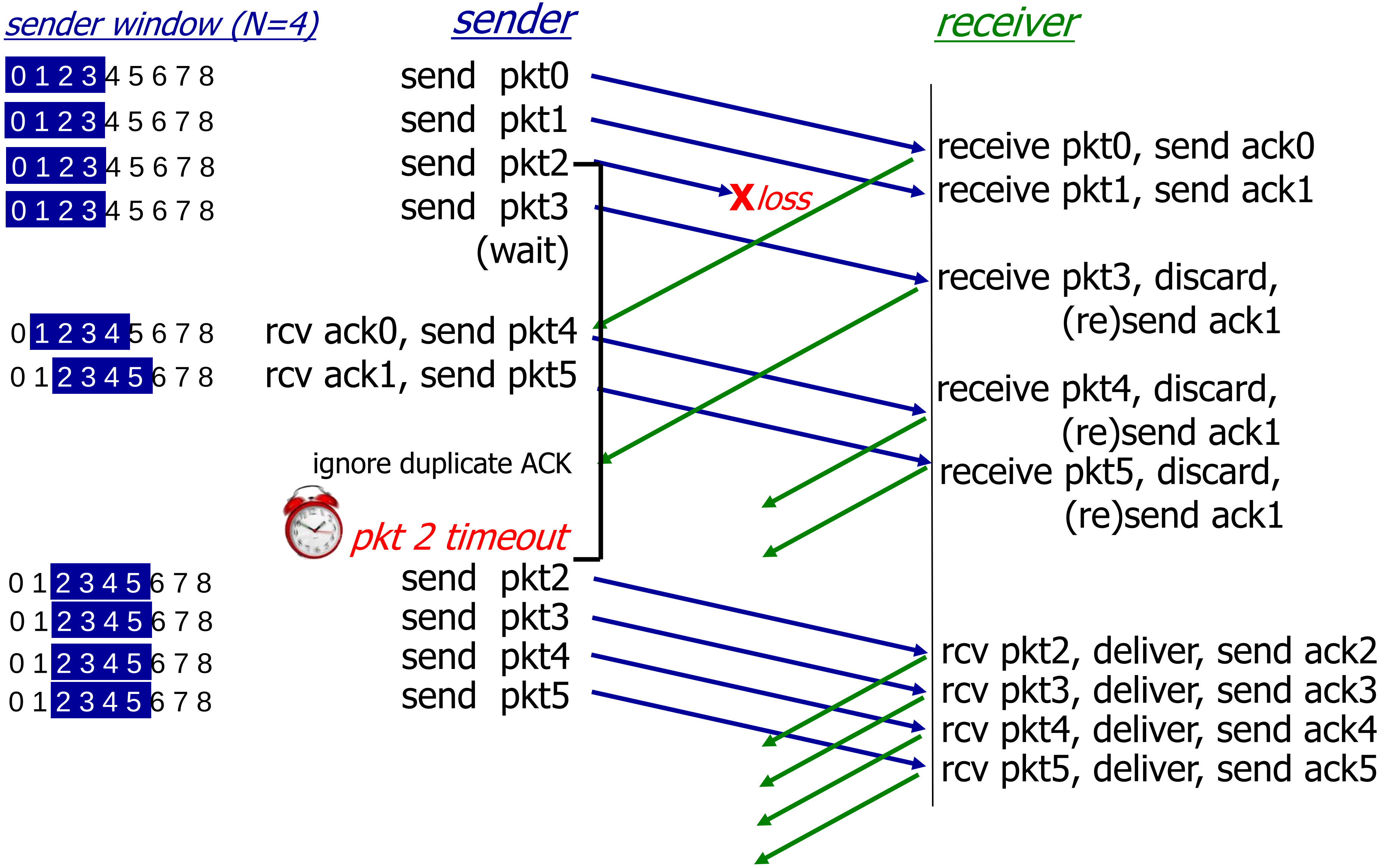
Go-back-N (GBN)

- Sender can have up to N unacked packets in pipeline
- Receiver only sends **cumulative** ack
  - ACK for last contiguous packet
  - No ACK for new packets past a sequence number gap
- Sender has timer for oldest unacked packet
  - When timer expires, retransmit all unacked packets



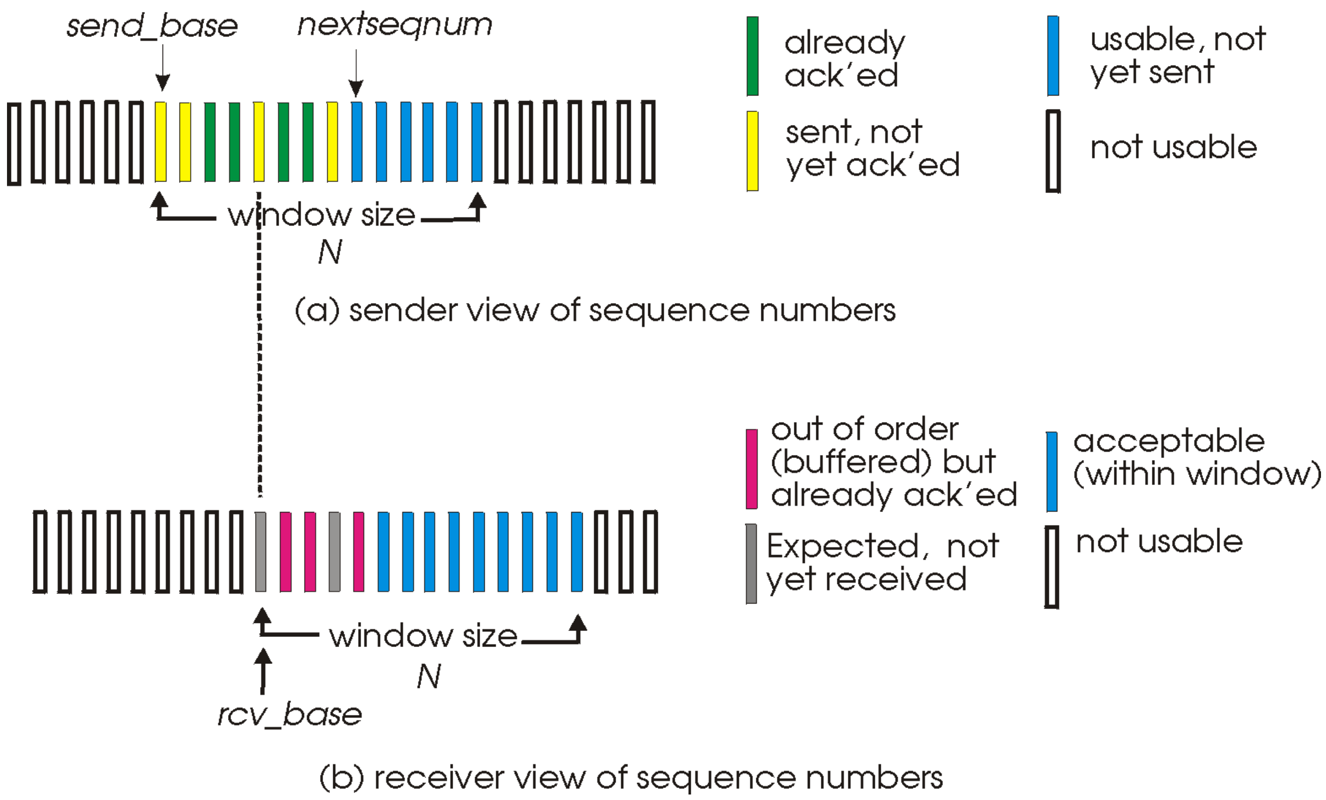
"Window" of up to *N*, consecutive unacknowledged packets allowed

GBN


sender window (N=4)
sender
receiver
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, discard,
(re)send ack1
receive pkt5, discard,
(re)send ack1
ignore duplicate ACK
pkt 2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - *N* consecutive seq #' s
  - limits seq #s of sent, unACKed pkts



Transport Layer

RDT Principles

Checksum- Used to detect bit errors in transmitted pacets

RDT Principles

Checksum- Used to detect bit errors in transmitted pacets

Timer- Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within a channel

RDT Principles

Checksum- Used to detect bit errors in transmitted pacets

Timer- Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within a channel

Sequence Number- Used for sequential numbering of packets of data flowing form sender to receiver. Gaps in sequence number of packets allow the receiver to detect a lost or duplicate packet

RDT Principles

Checksum- Used to detect bit errors in transmitted pacets

Timer- Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within a channel

Sequence Number- Used for sequential numbering of packets of data flowing form sender to receiver. Gaps in sequence number of packets allow the receiver to detect a lost or duplicate packet

Acknowledgement- Used by the receiver to tell the sender that a packet or set of packets has been received correctly. ACKs will typically carry the sequence # of the packet being acknowledged

RDT Principles

Checksum- Used to detect bit errors in transmitted pacets

Timer- Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within a channel

Sequence Number- Used for sequential numbering of packets of data flowing form sender to receiver. Gaps in sequence number of packets allow the receiver to detect a lost or duplicate packet

Acknowledgement- Used by the receiver to tell the sender that a packet or set of packets has been received correctly. ACKs will typically carry the sequence # of the packet being acknowledged

Negative Acknowledgement- Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgements will typically carry the sequence number of the packet that was not received correctly

RDT Principles

Checksum- Used to detect bit errors in transmitted pacets

Timer- Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within a channel

Sequence Number- Used for sequential numbering of packets of data flowing form sender to receiver. Gaps in sequence number of packets allow the receiver to detect a lost or duplicate packet

Acknowledgement- Used by the receiver to tell the sender that a packet or set of packets has been received correctly. ACKs will typically carry the sequence # of the packet being acknowledged

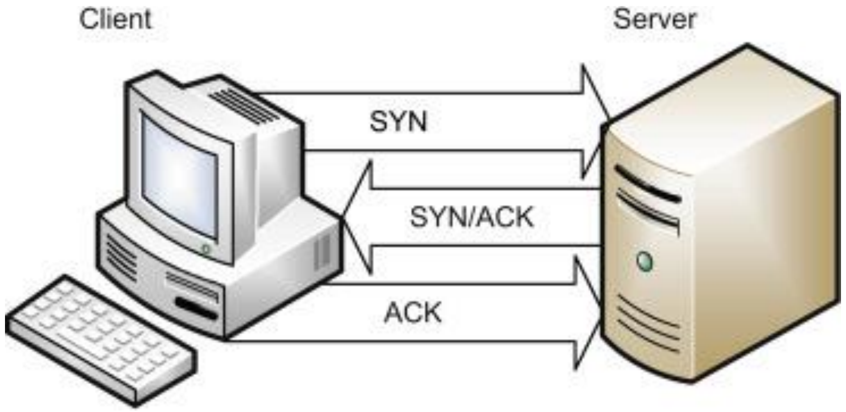
Negative Acknowledgement- Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgements will typically carry the sequence number of the packet that was not received correctly

Window, pipelining- The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation.

Transport Layer

TCP (Transmission Control Protocol)

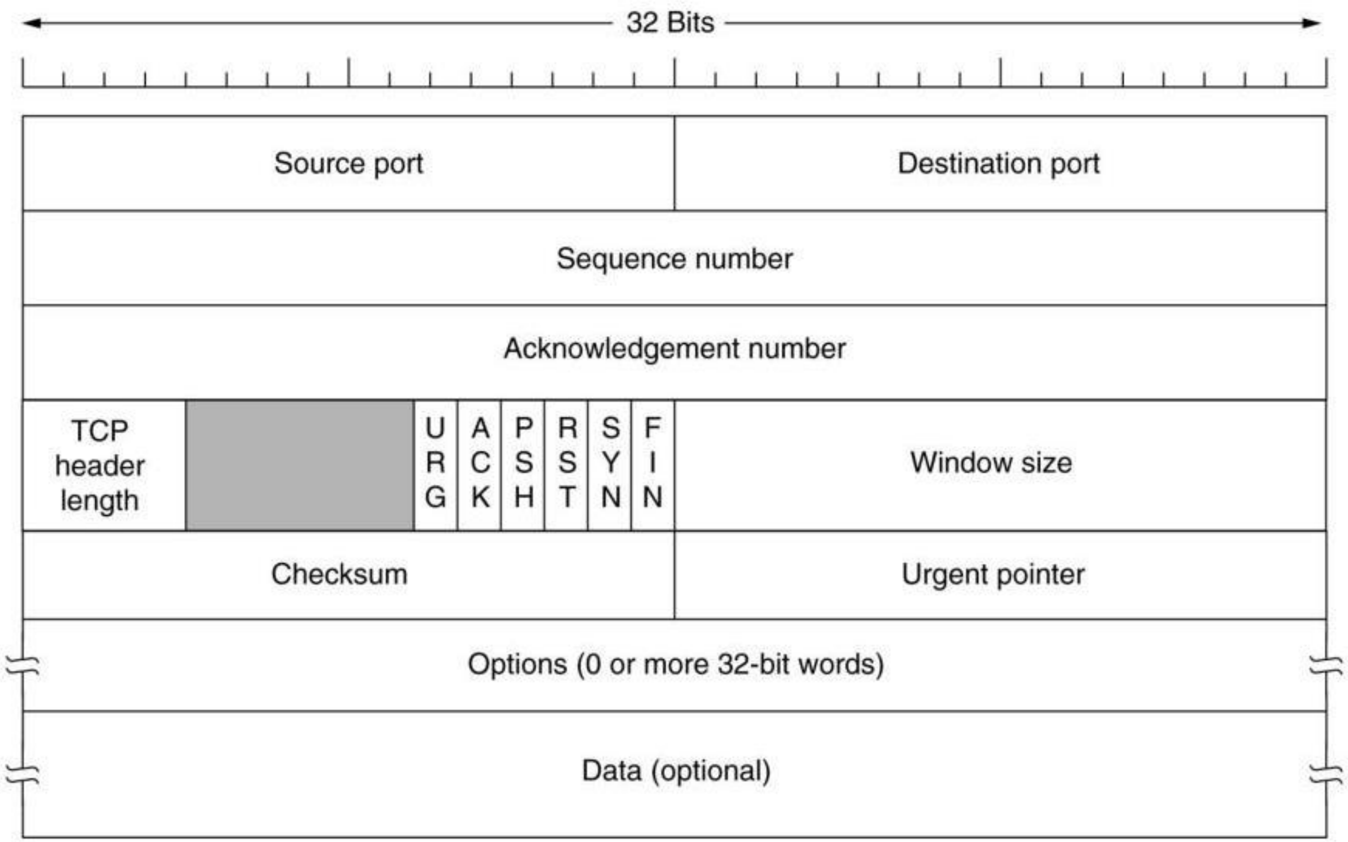
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
- pipelined:
  - TCP congestion and flow control set window size



TCP Handshake

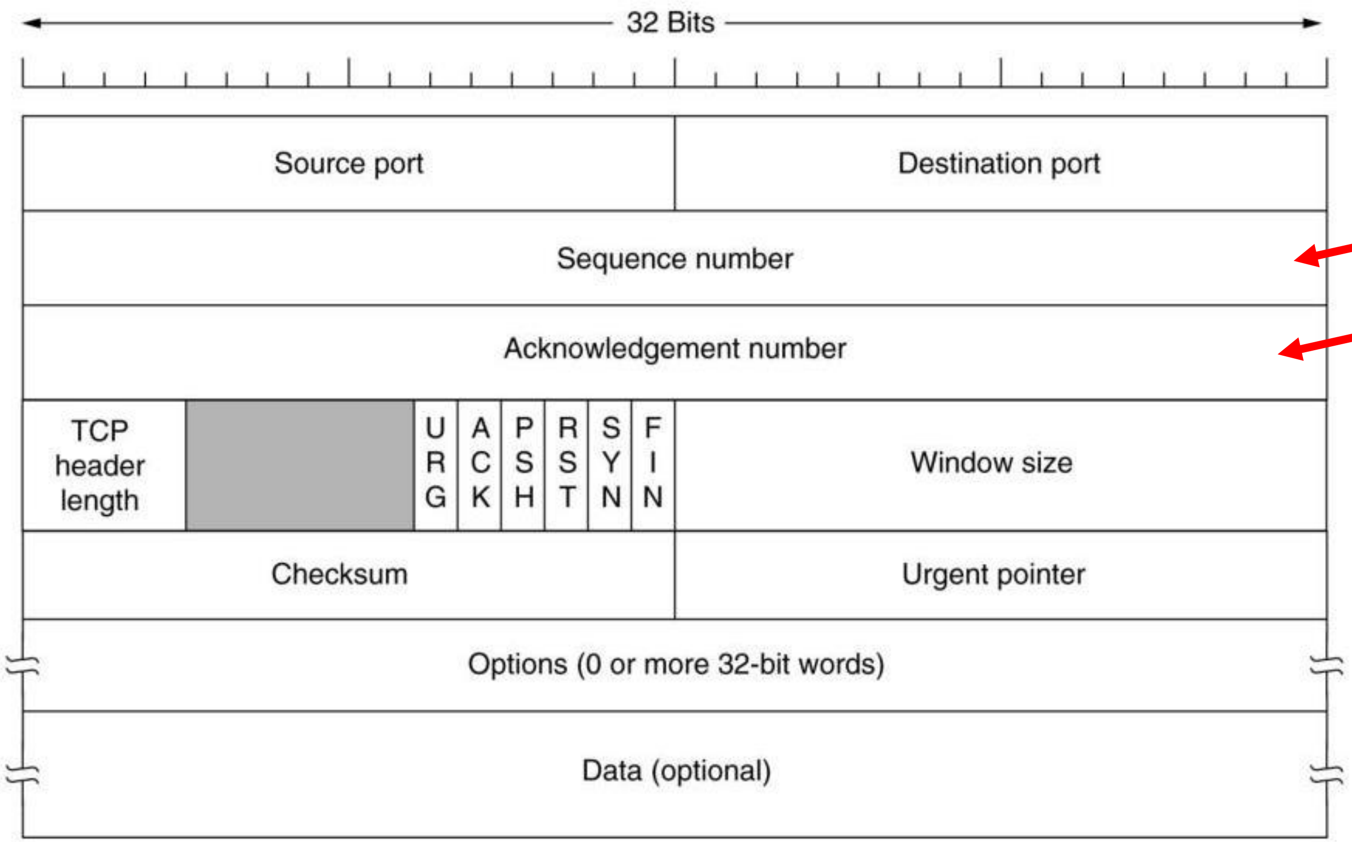
- full duplex data:
  - bi-directional data flow in same connection
- connection-oriented:
  - handshaking (exchange of control msgs) in its sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP Segment Structure



TCP header length in 32 bit words,
URG-urgent, ACK- ack number is valid, PSH-push, RST-reset connection,
SYN-used to establish connection, FIN-used to release connection

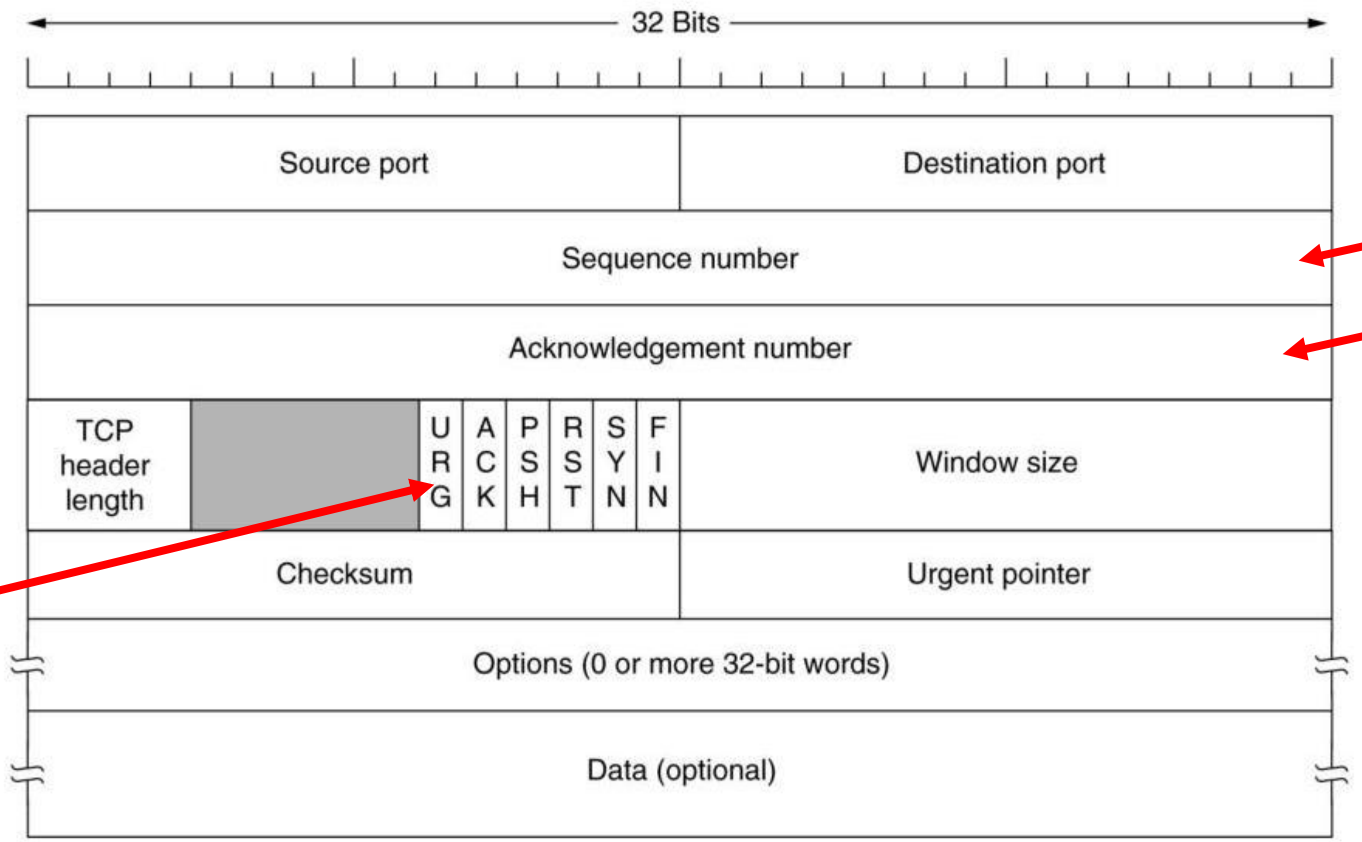
TCP Segment Structure



Counts by bytes, not segment

TCP header length in 32 bit words,
URG-urgent, ACK- ack number is valid, PSH-push, RST-reset connection,
SYN-used to establish connection, FIN-used to release connection

TCP Segment Structure



Counts by bytes, not segment

TCP header length in 32 bit words,
URG-urgent, ACK- ack number is valid, PSH-push, RST-reset connection,
SYN-used to establish connection, FIN-used to release connection

TCP Segment Structure

sequence numbers:

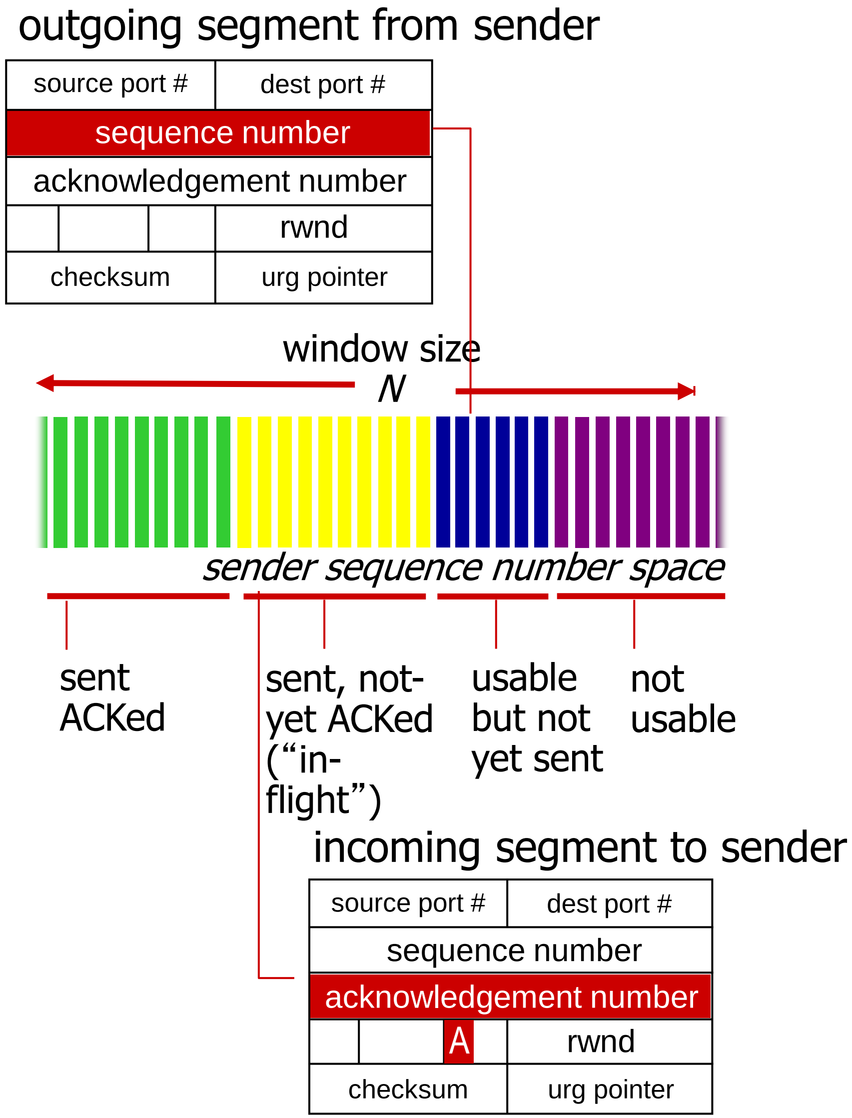
- byte stream "number" of first byte in segment's data

acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



No class on Wednesday 10/5 (go to the career fair)

Fill out the short survey on discord for extra credit

You vs the guy she tells you not to worry about.

UDP	TCP
Unreliable	Reliable
Connectionless	Connection-oriented
No windowing or retransmission	Segment retransmission and flow control through windowing
No sequencing	Segment sequencing
No acknowledgement	Acknowledge segments

TCP v UDP Structure

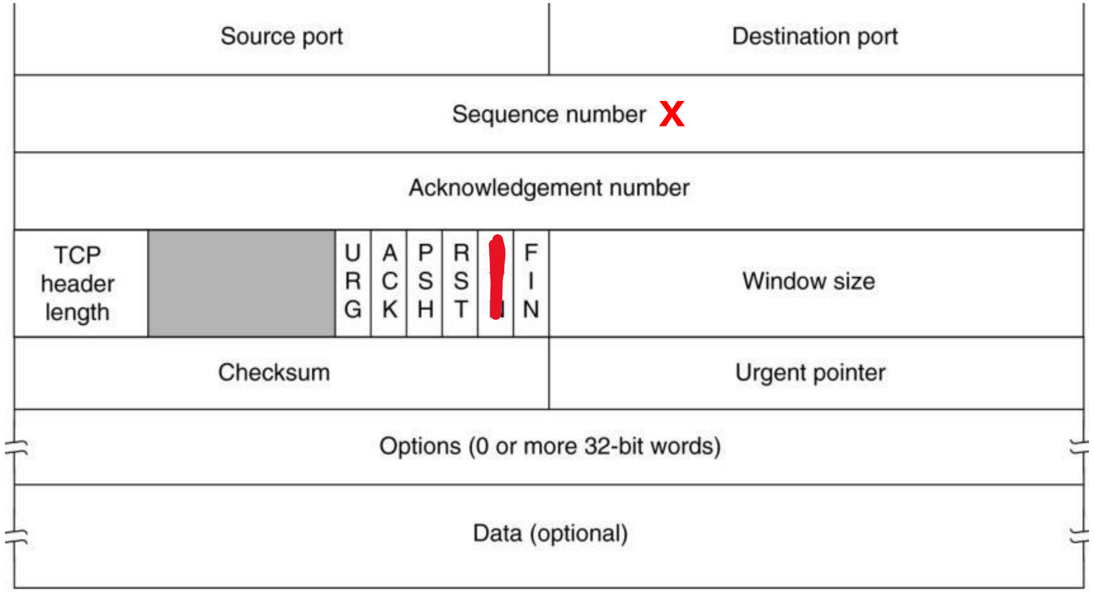
TCP Segment Header Format

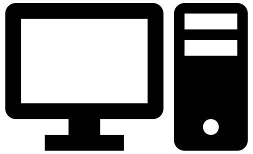
Bit #	0		7	8	15	16	23	24	31
0	Source Port					Destination Port			
32	Sequence Number								
64	Acknowledgment Number								
96	Data Offset	Res	Flags			Window Size			
128	Header and Data Checksum					Urgent Pointer			
160...	Options								

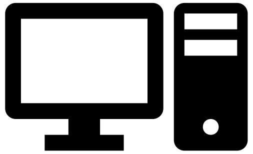
UDP Datagram Header Format

Bit #	0	7	8	15	16	23	24	31
0	Source Port				Destination Port			
32	Length				Header and Data Checksum			

The TCP Handshake (Connection Management)



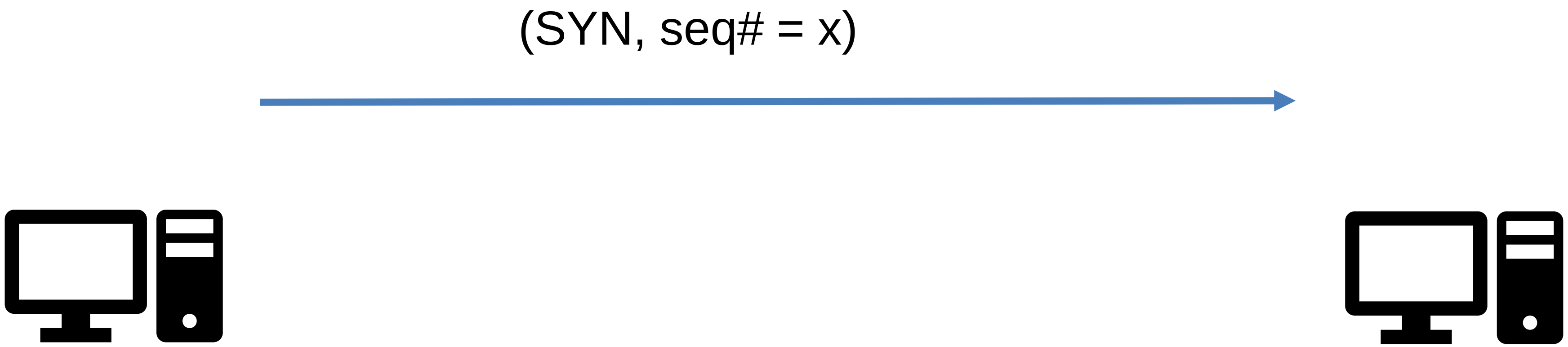




- When establishing the connection, enable the **SYN** flag (set to 1)
- Set an initial sequence number

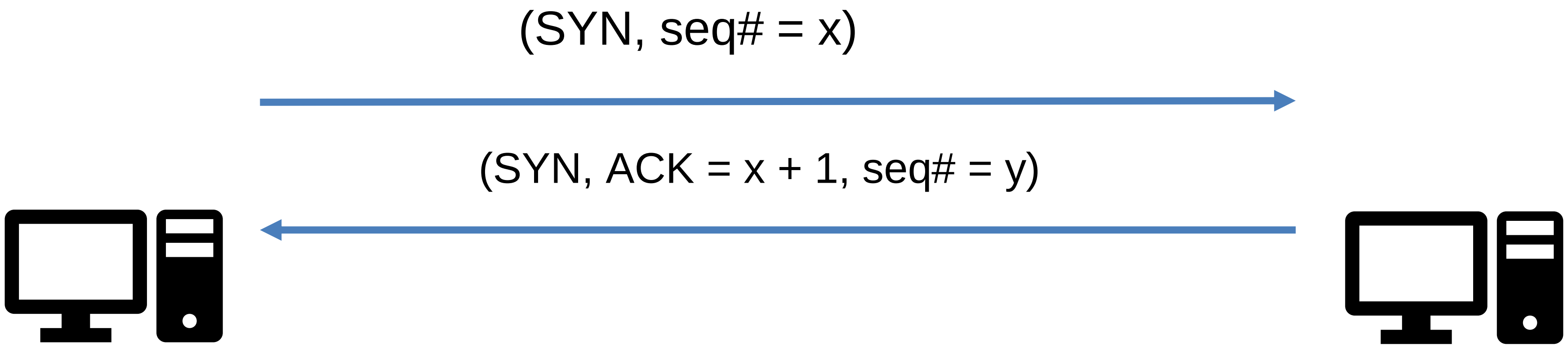
Transport Layer

The TCP Handshake



- When establishing the connection, enable the **SYN** flag (set to 1)
- Set an initial sequence number

The TCP Handshake



- Acknowledge message and increment sequence number

The TCP Handshake

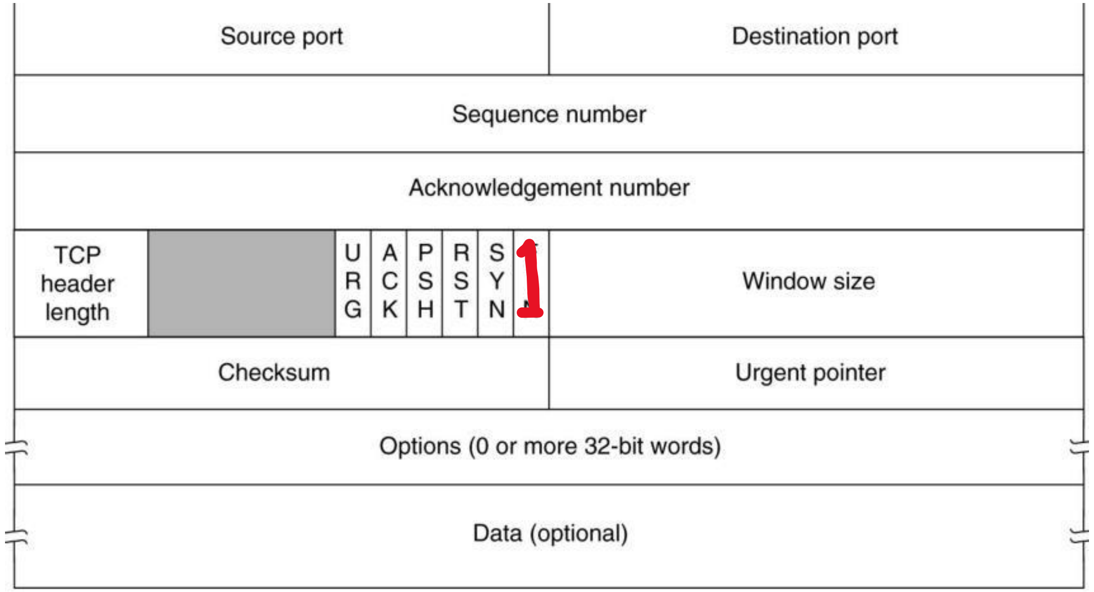
(SYN, seq# = x)

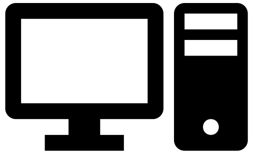
(SYN, ACK = x + 1, seq# = y)

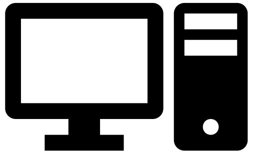
(ACK = y + 1)

- Acknowledge the acknowledgement

The TCP Handshake



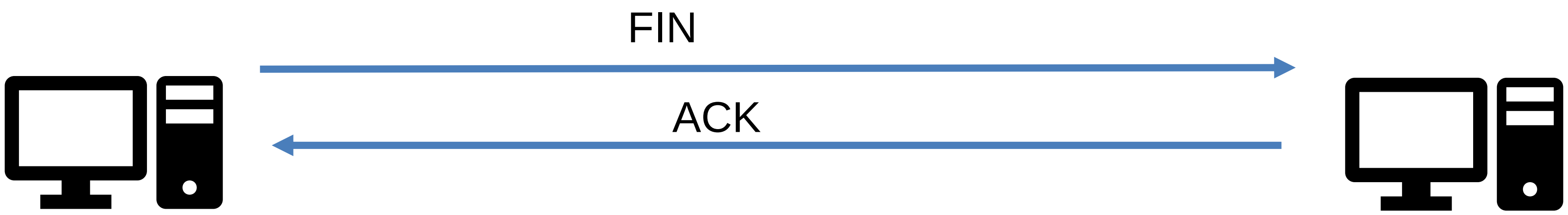




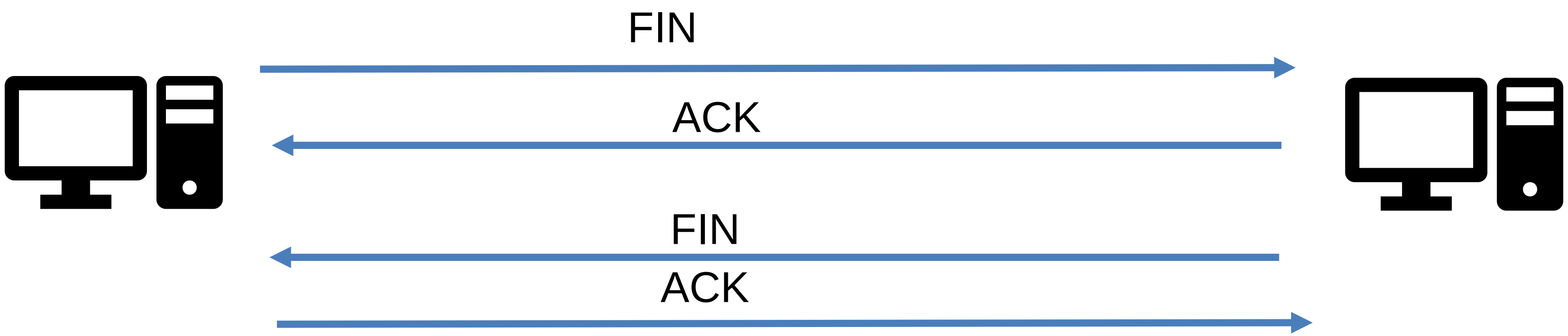
- The end communication, set the FIN flag

Transport Layer

The TCP Handshake



The TCP Handshake

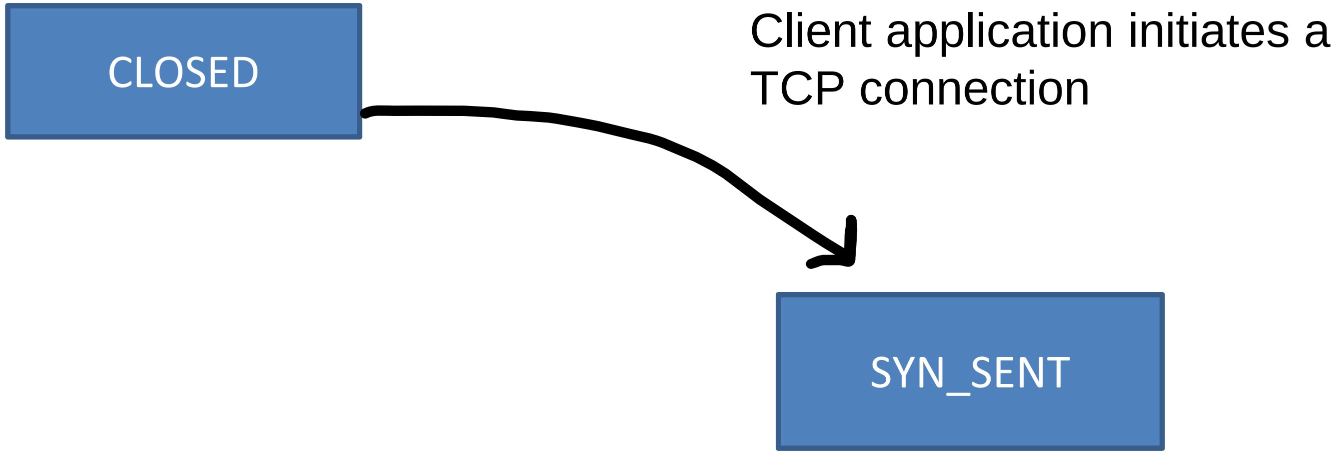


TCP States (Client)

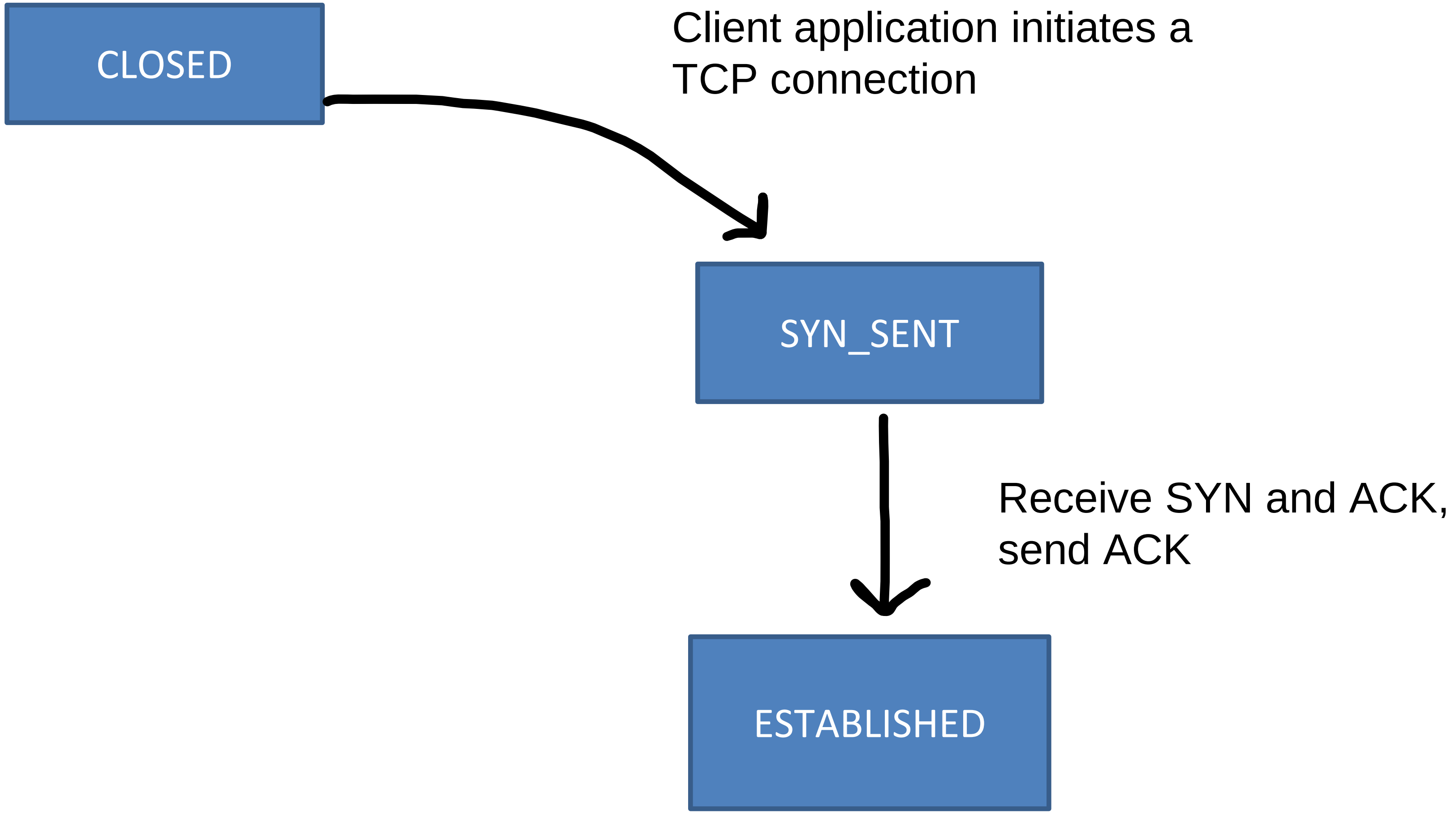
CLOSED

During the life of a TCP connection, the TCP protocol makes transitions through various **TCP states**

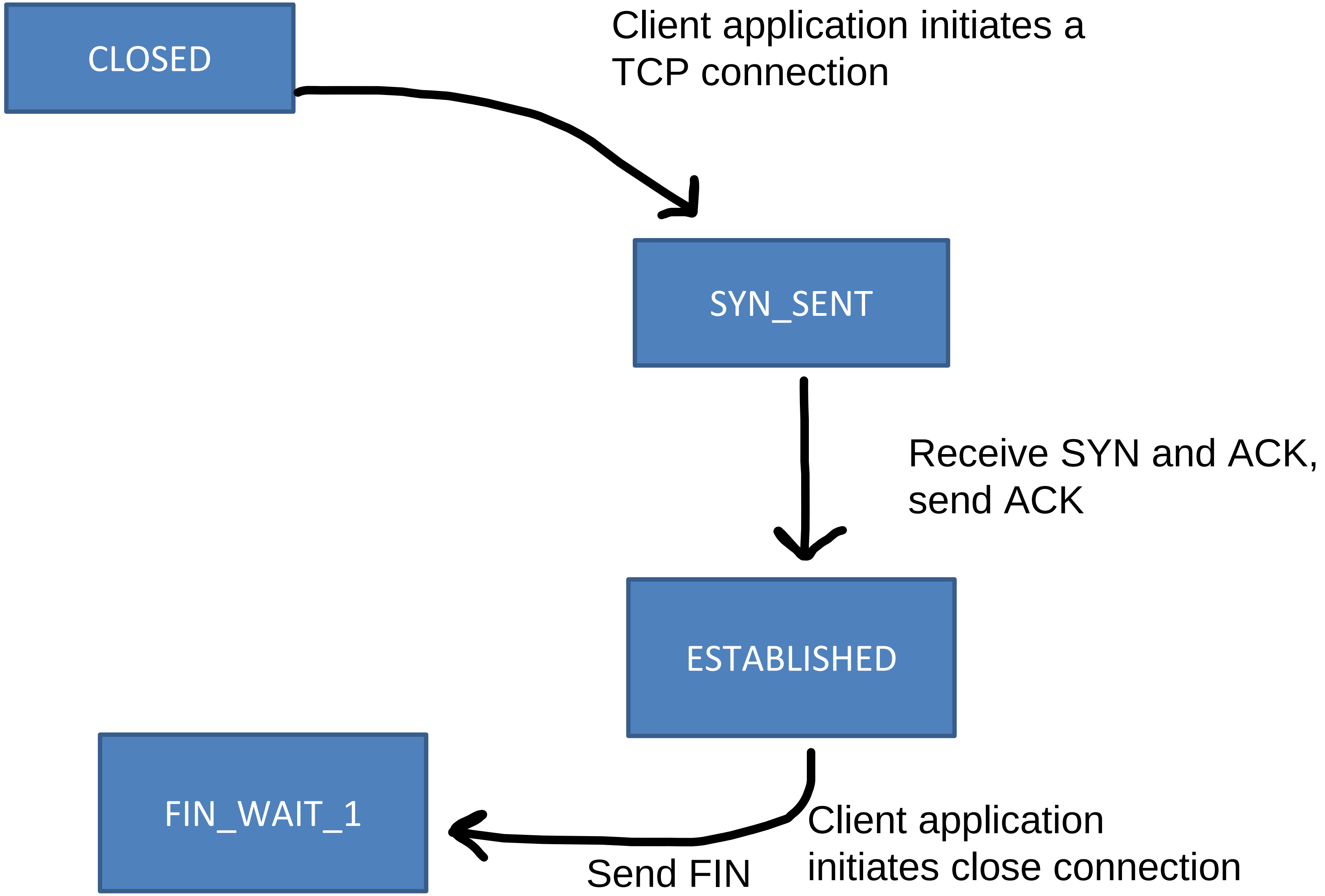
TCP States (Client)



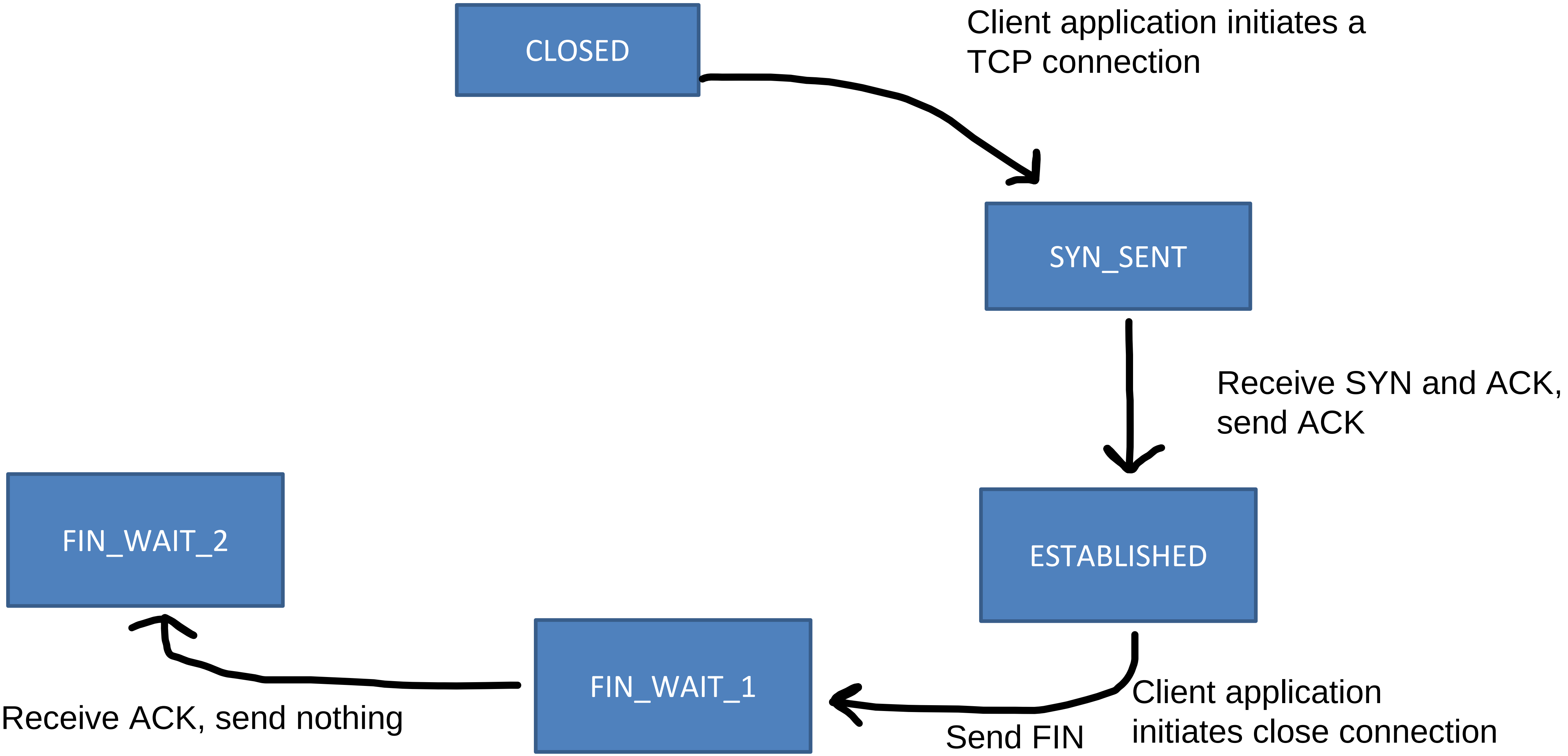
TCP States (Client)



TCP States (Client)

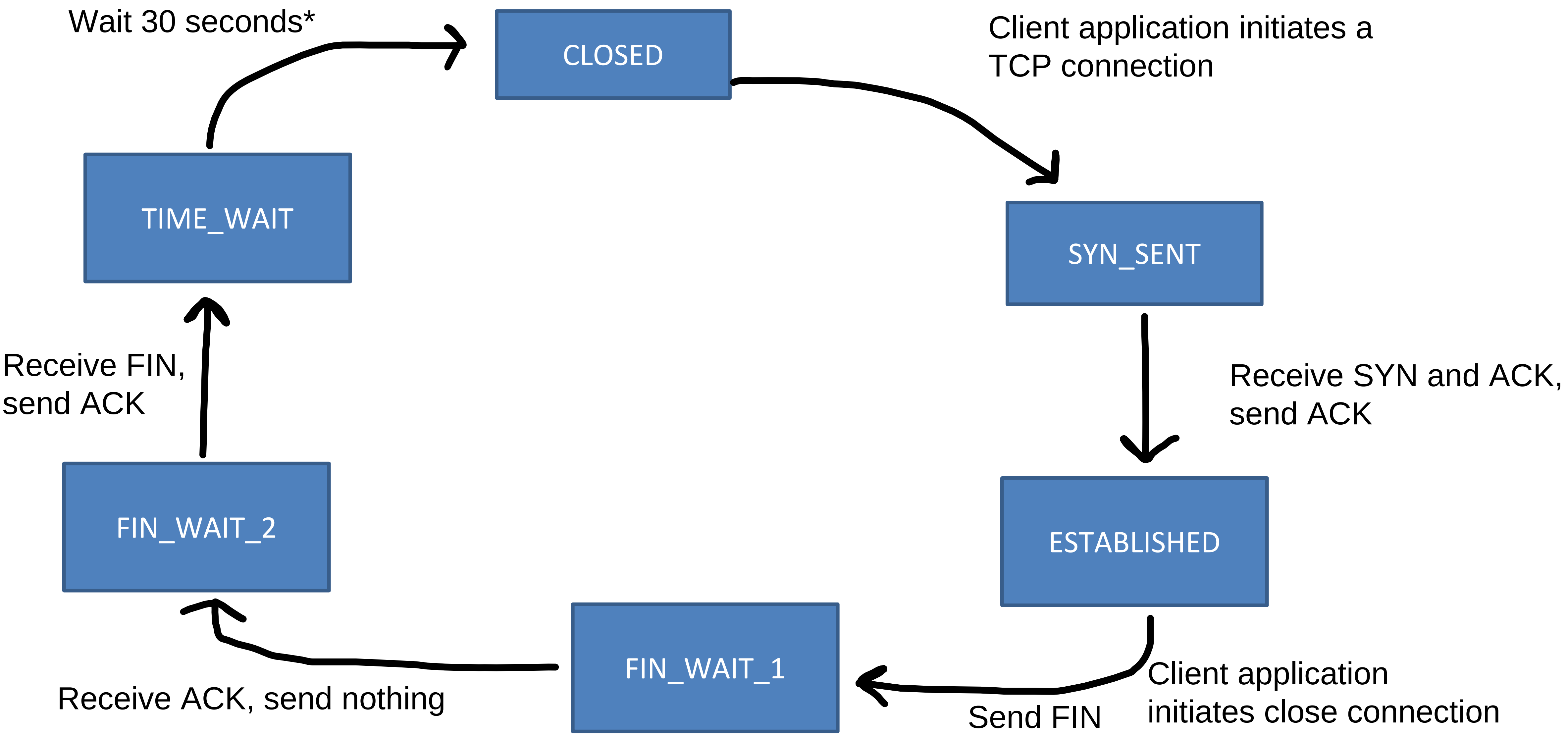


TCP States


CLOSED
Client application initiates a TCP connection
SYN_SENT
Receive SYN and ACK, send ACK
ESTABLISHED
Client application initiates close connection
Send FIN
FIN_WAIT_1
Receive ACK, send nothing
FIN_WAIT_2

Transport Layer

TCP States (Client)



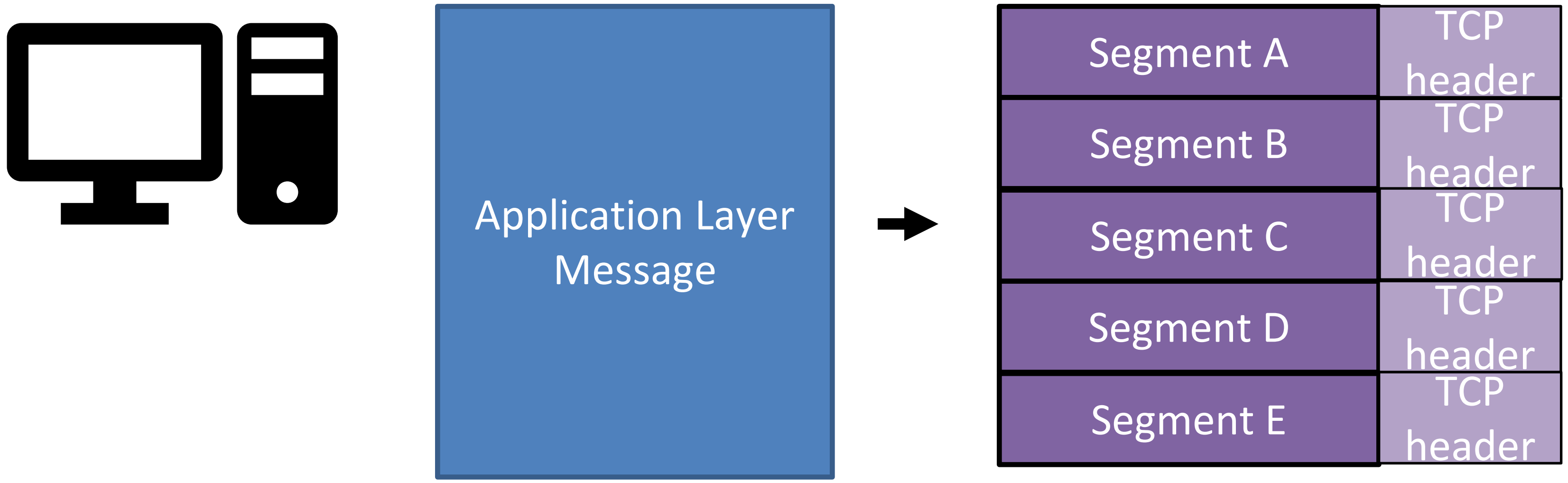
TCP States (Client)

What if we receive a packet that has an invalid port number?

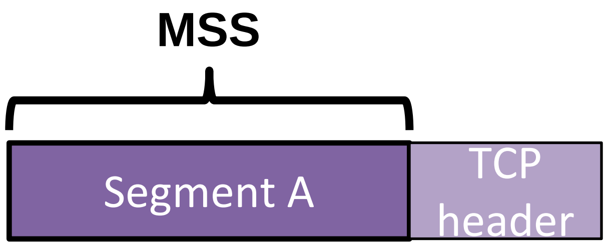
TCP Packet → Send a TCP segment with RST flag on

UDP Packet→ Send an **ICMP** datagram (network layer thing)

TCP Flow Control



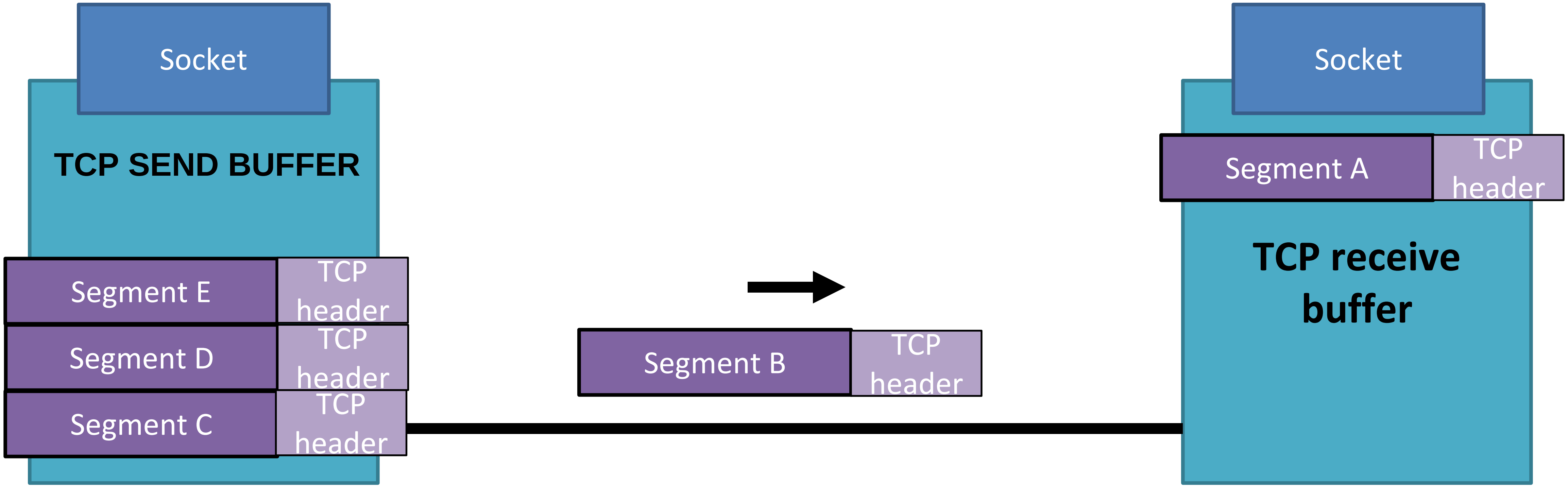
Application layer messages are split into smaller chunks called **segments**



The size of these segments is determined by the **maximum segment size (MSS)**

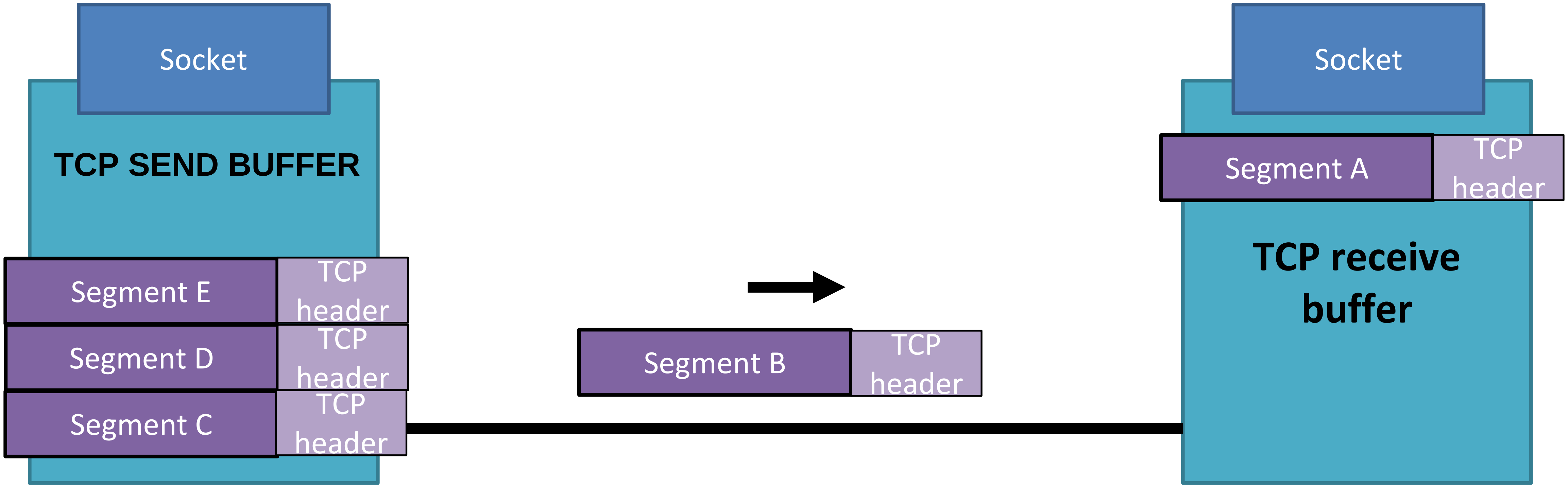
Transport Layer

TCP Flow Control



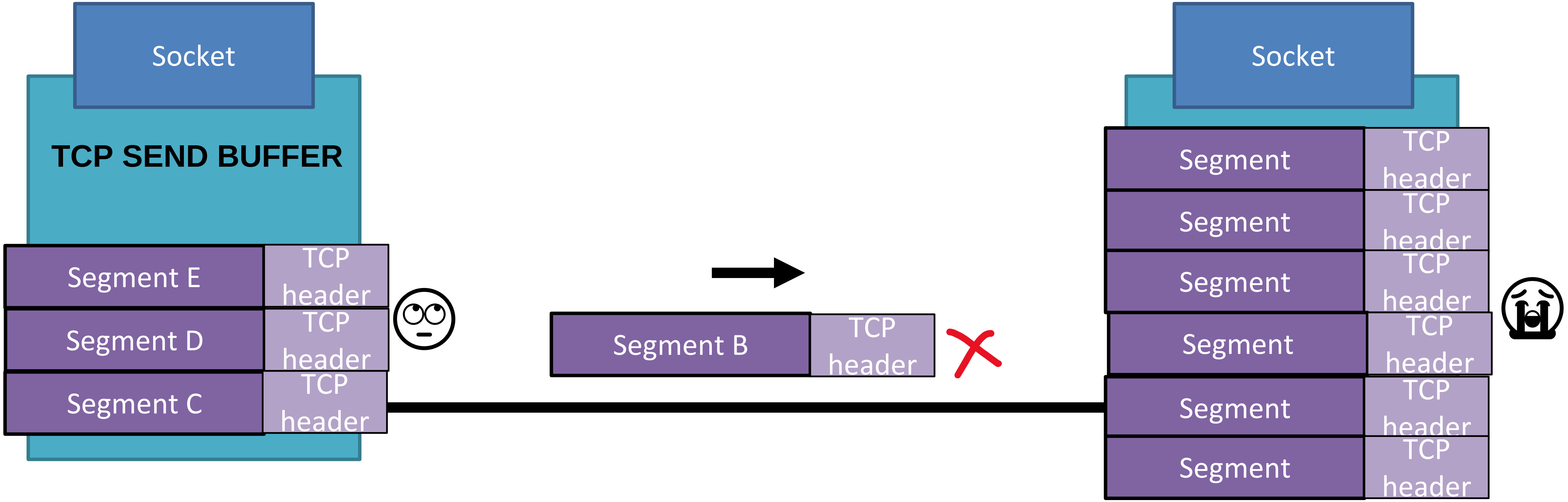
Applications read streams of data from a **TCP buffer**

TCP Flow Control



Applications read streams of data from a **TCP buffer**

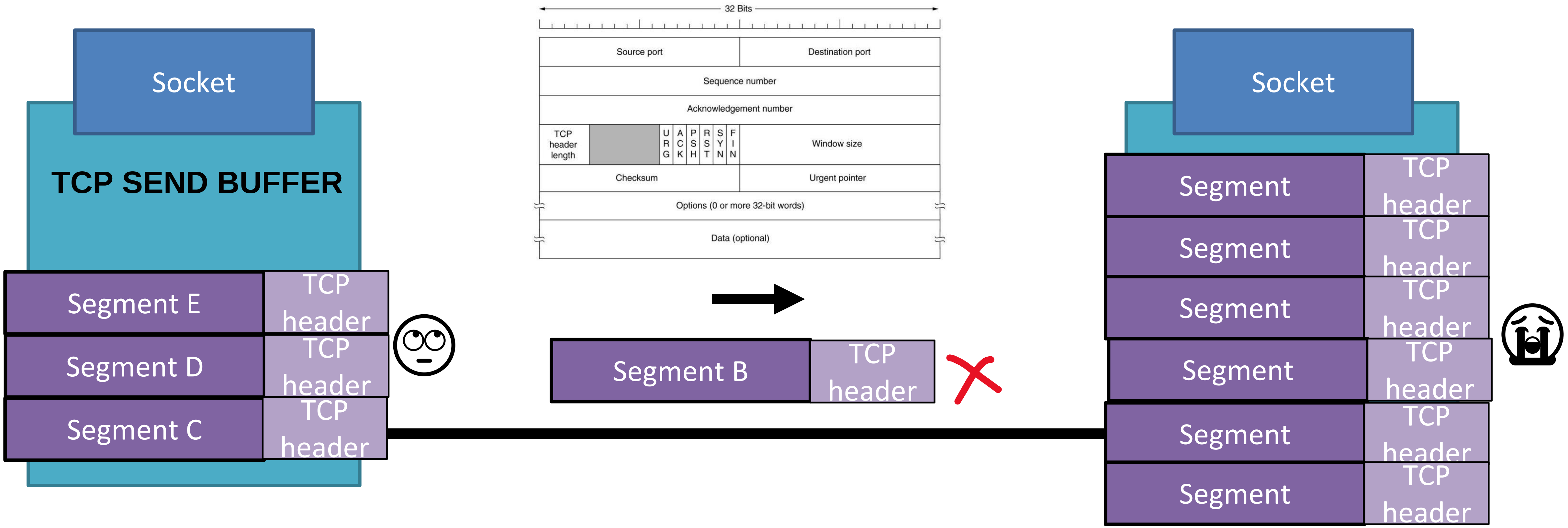
TCP Flow Control



How could we prevent something like this from happening?

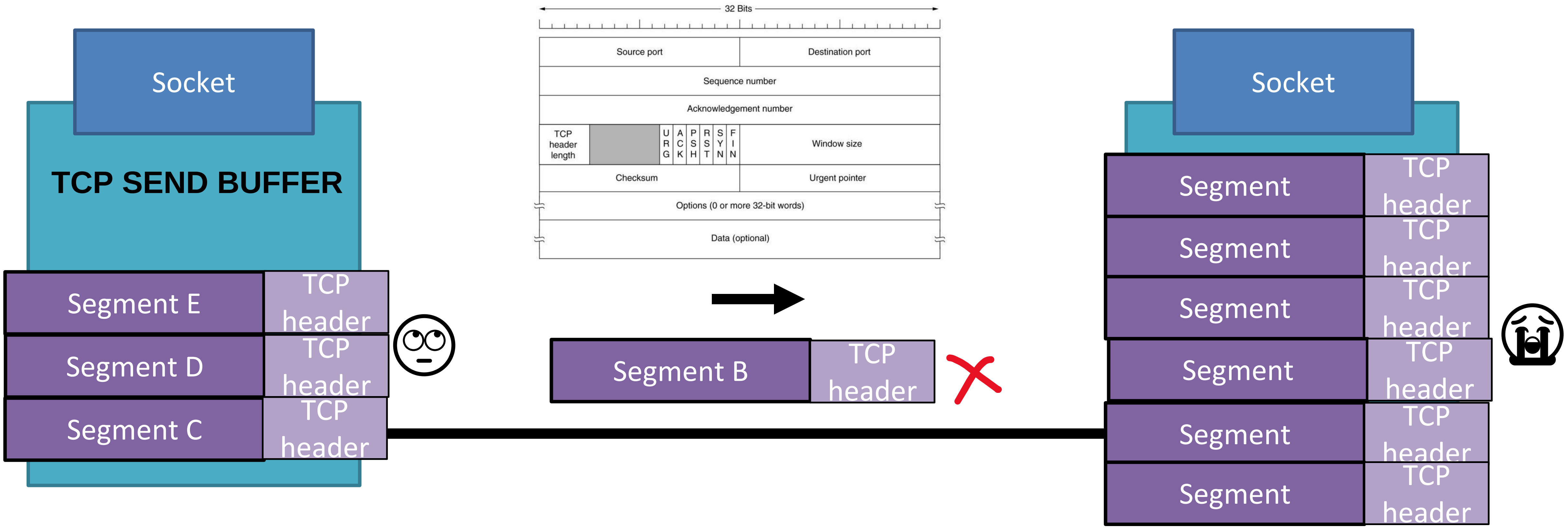
Transport Layer

TCP Flow Control



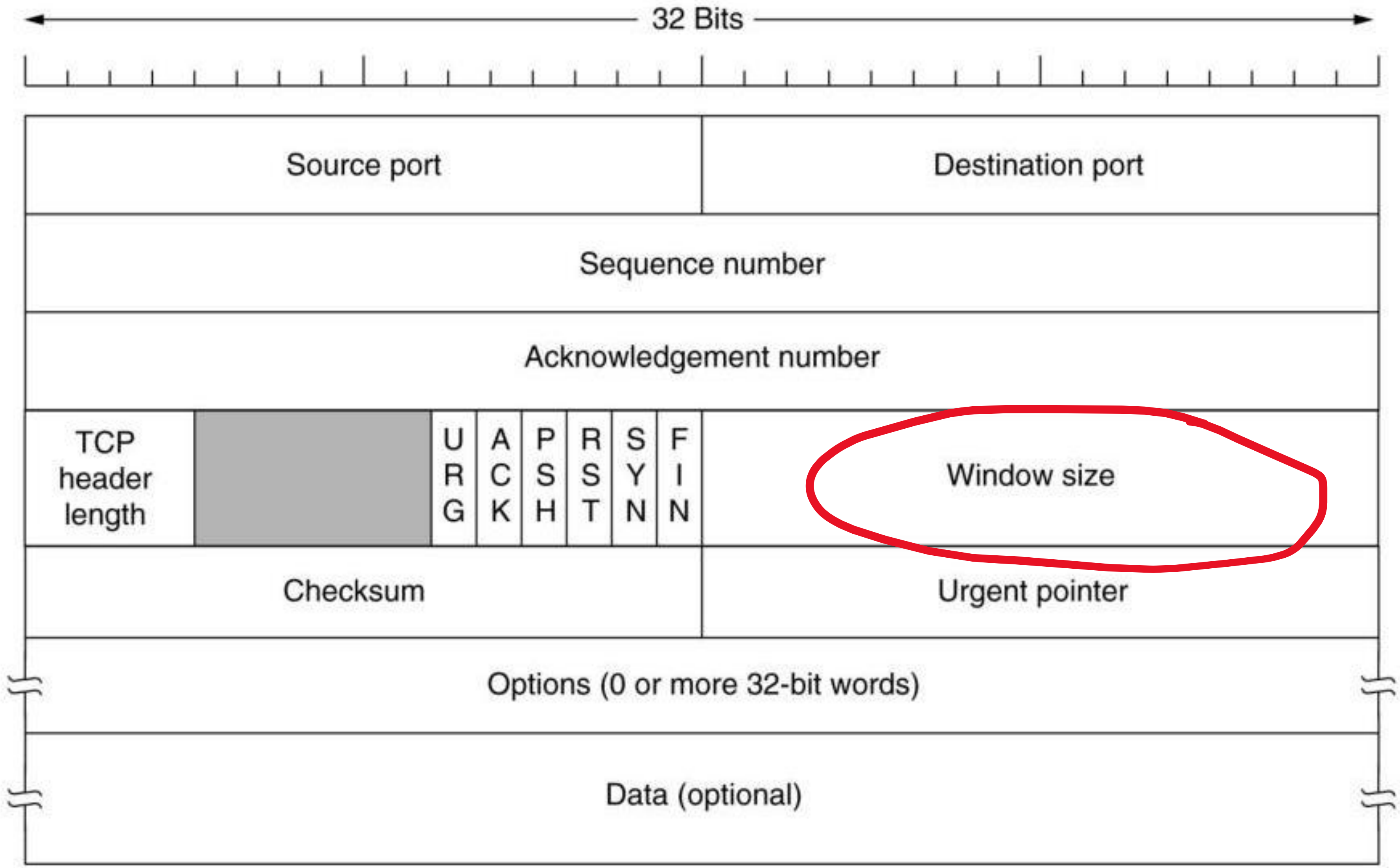
How could we prevent something like this from happening?

TCP Flow Control

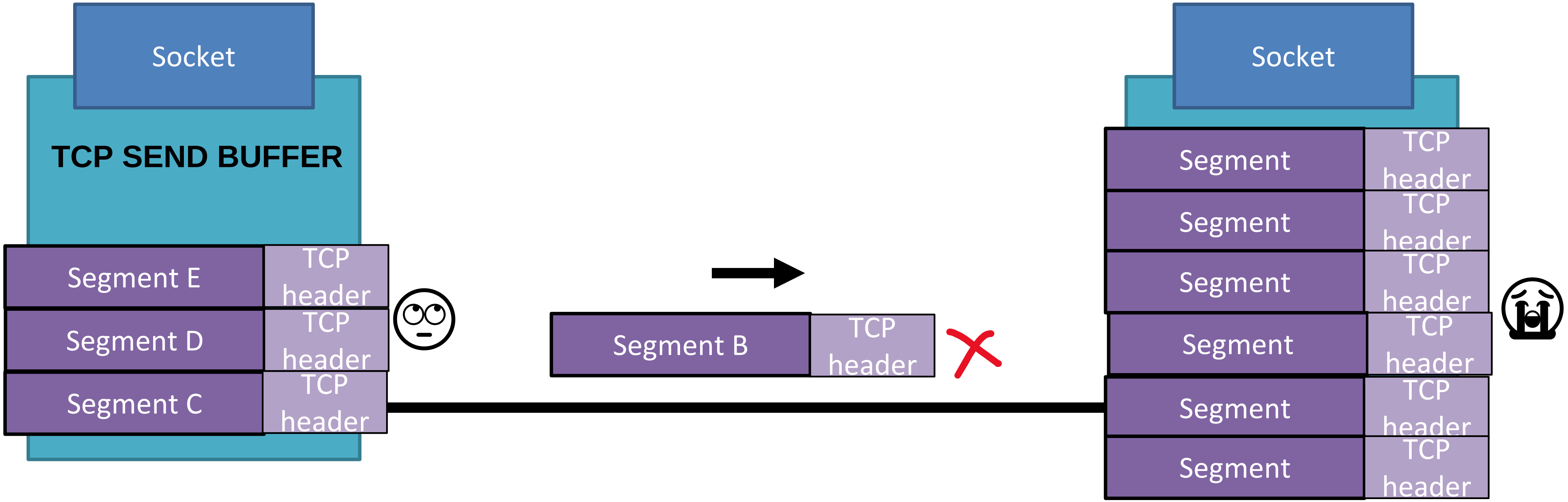


We could send back to the sender how much available space we have in our buffer!

TCP Flow Control



TCP Flow Control



We could send back to the sender how much available space we have in our buffer!

TCP Timeout

https://wps.pearsoned.com/ecs_kurose_compnetw_6/216/55463/14198702.cw/index.html

TCP Timeout

What is a good way to determine when to timeout? (aka the length of timer)

1. Too short: premature timeout, unnecessary retransmissions
2. Too long: slow reaction to segment loss

The TCP timeout value should be around the same time it takes to ……

TCP Timeout

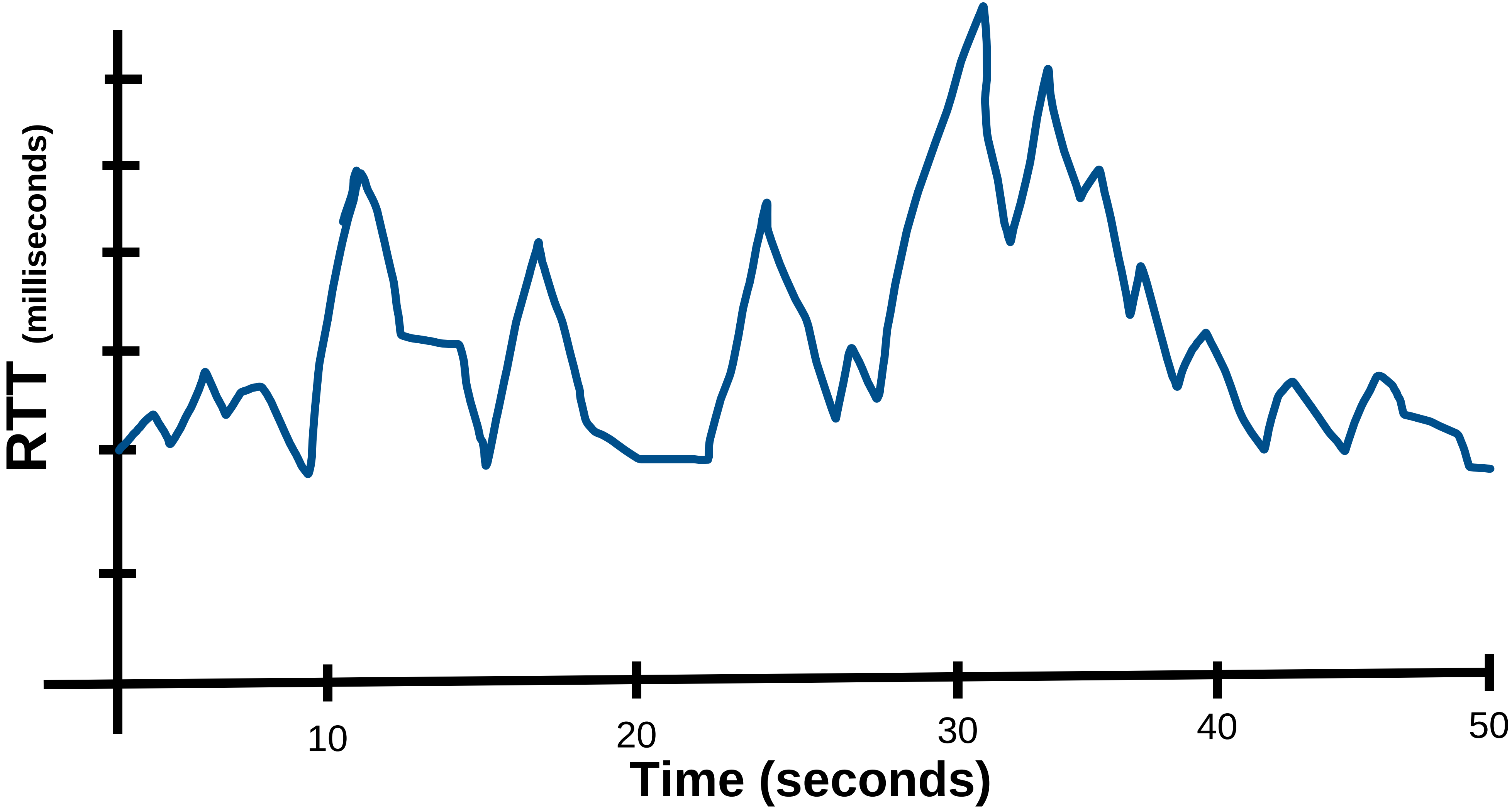
What is a good way to determine when to timeout? (aka the length of timer)

1. Too short: premature timeout, unnecessary retransmissions
2. Too long: slow reaction to segment loss

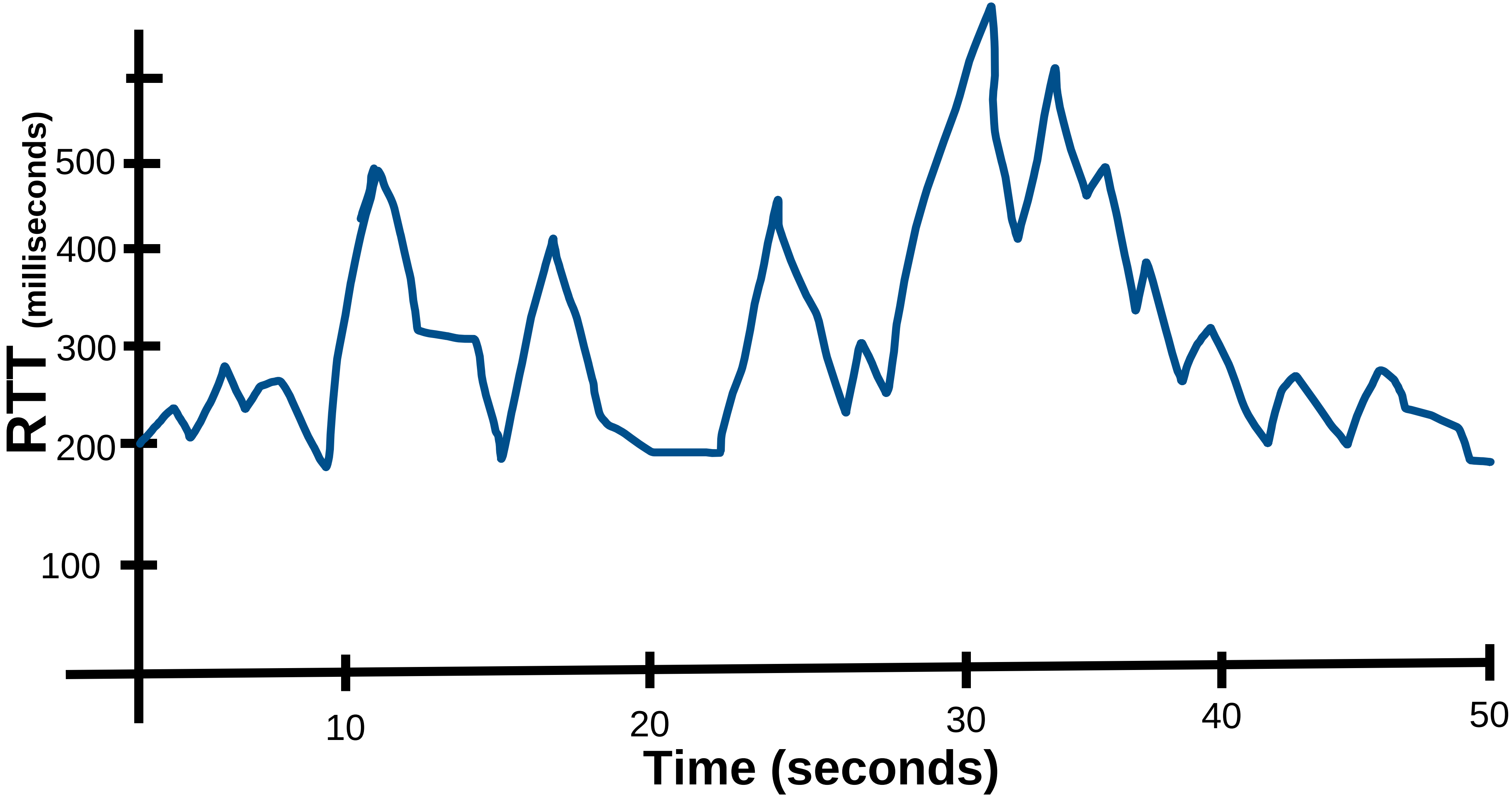
The TCP timeout value should be around the same time it takes to receive an acknowledgement on a sent packet (on average)

Let's consider setting it to be a dynamic value!

TCP Timeout



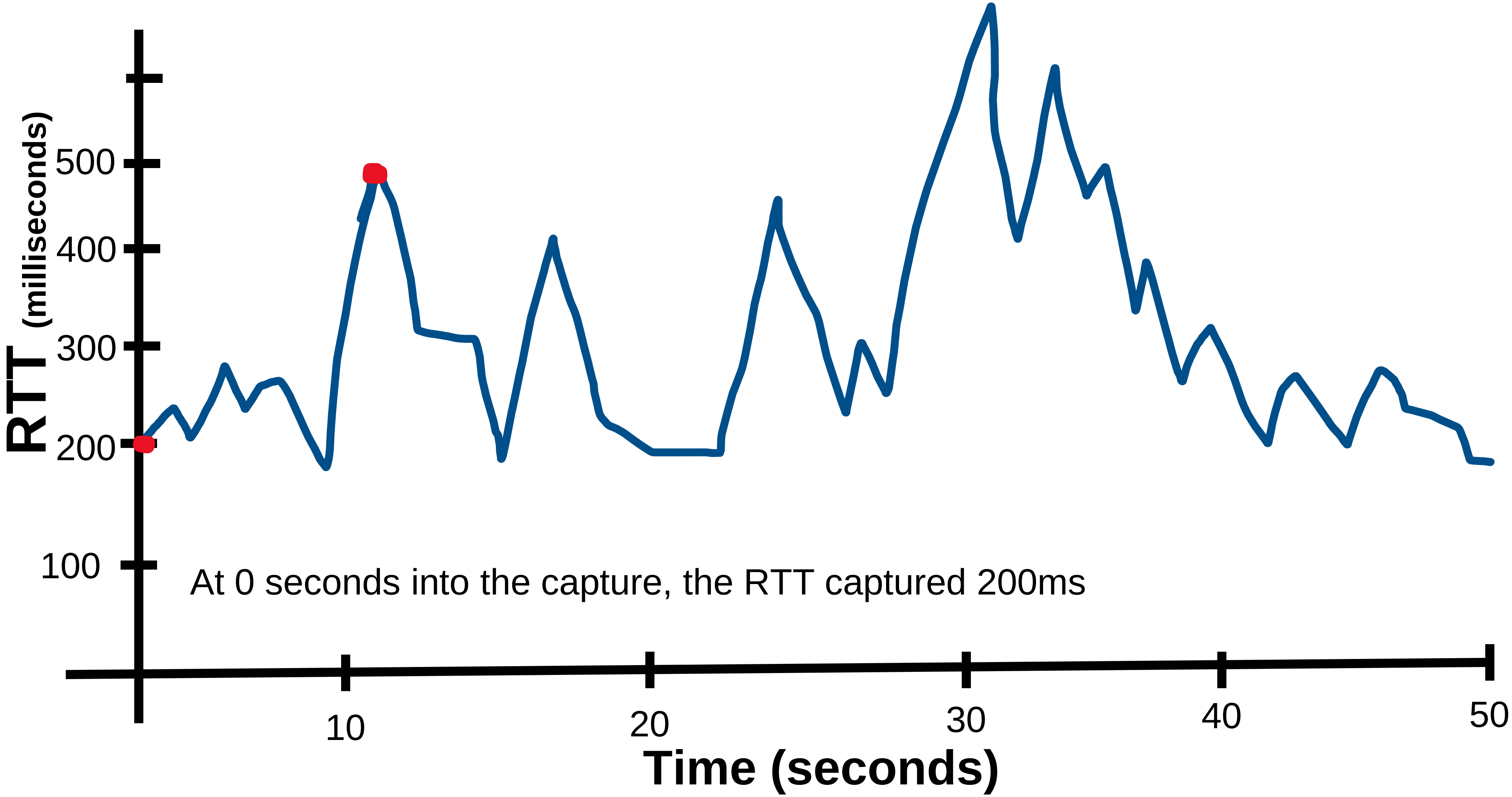
TCP Timeout



RTT (milliseconds)
500
400
300
200
100
10
20
30
40
50
Time (seconds)

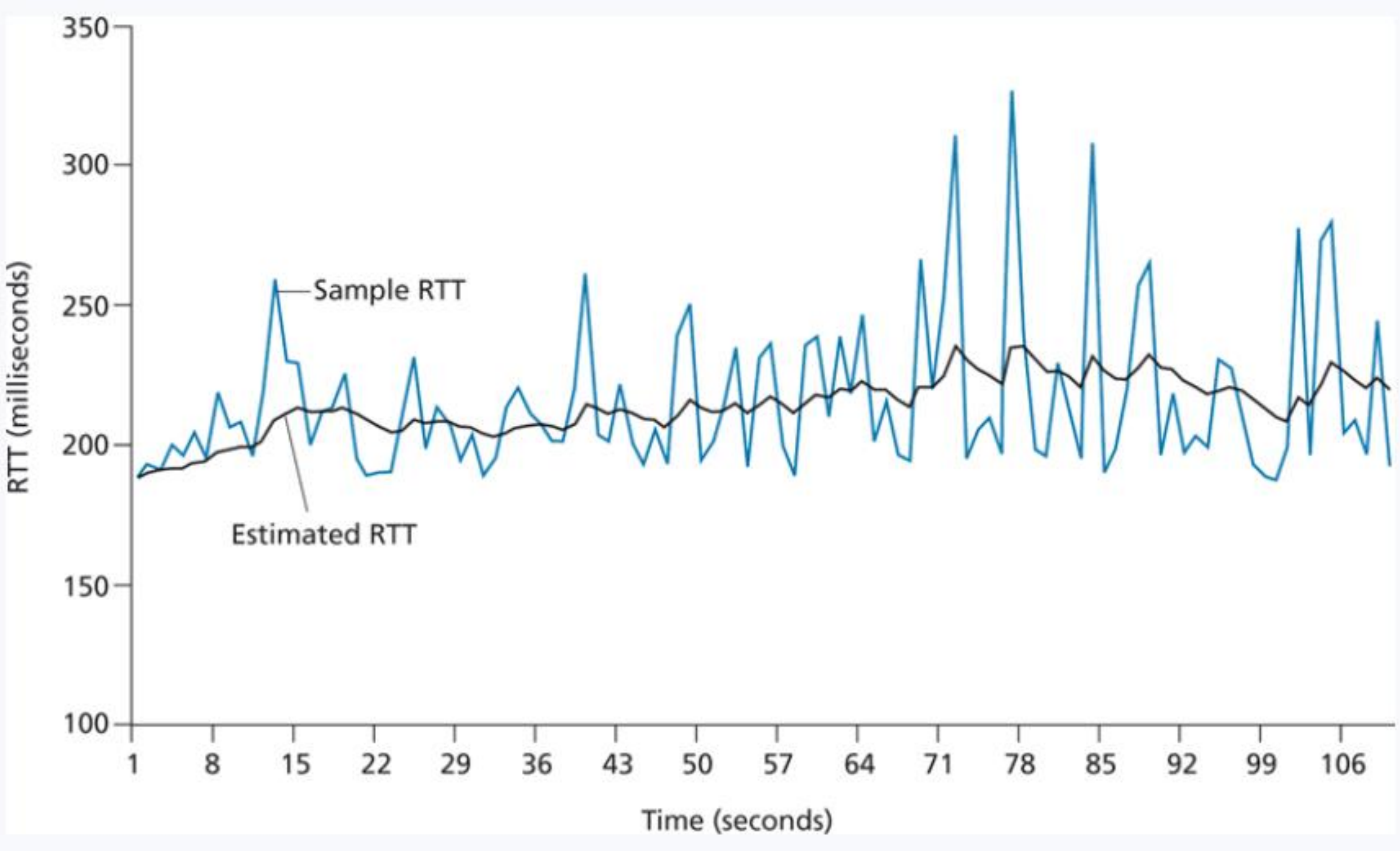
TCP Timeout

11 seconds into the capture, the RTT captured 500ms



At 0 seconds into the capture, the RTT captured 200ms

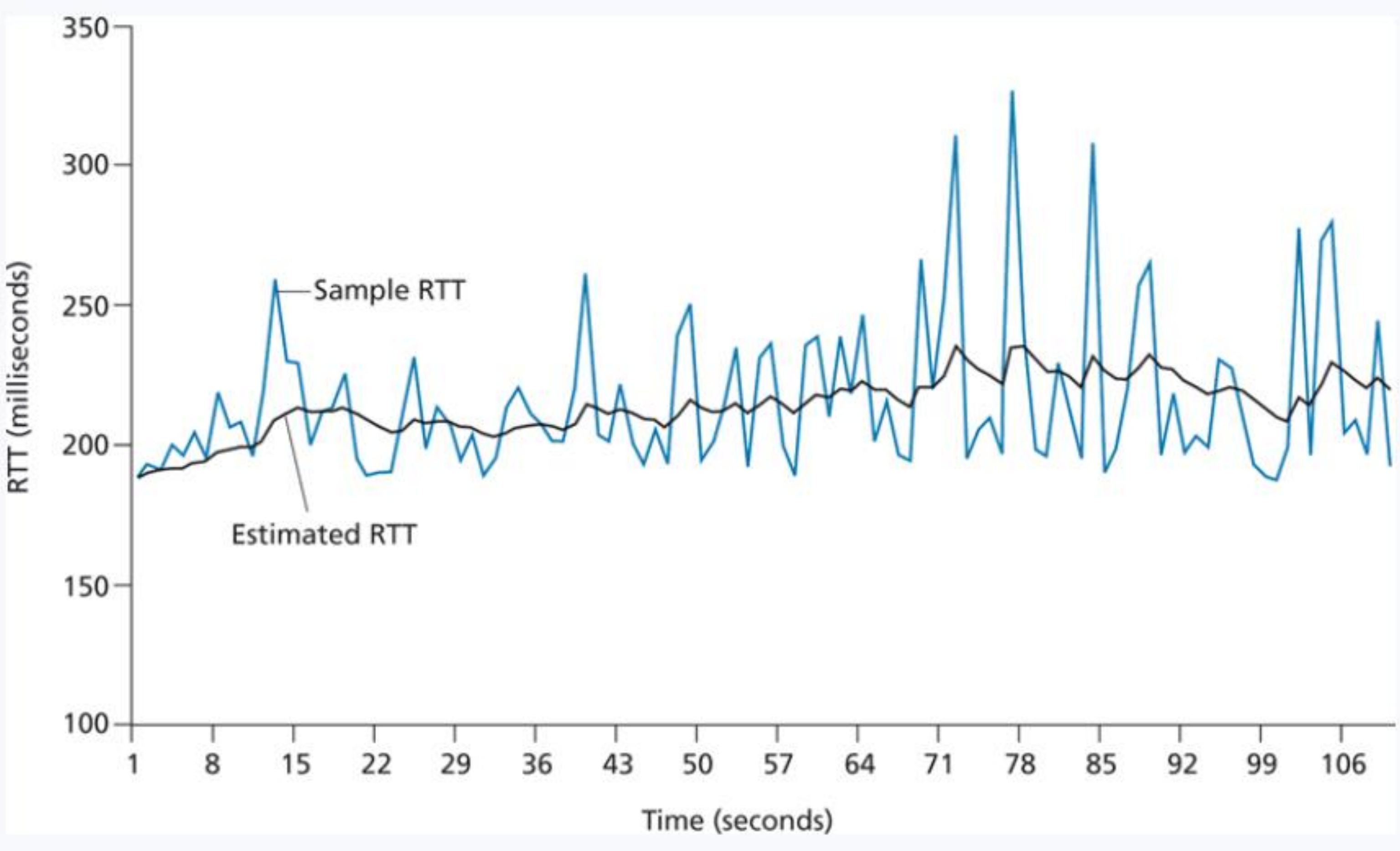
TCP Timeout



$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

a=0.125

TCP Timeout



In addition, we also want some kind of safety margin

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|
             (typically, β = 0.25)
```

TimeoutInterval =

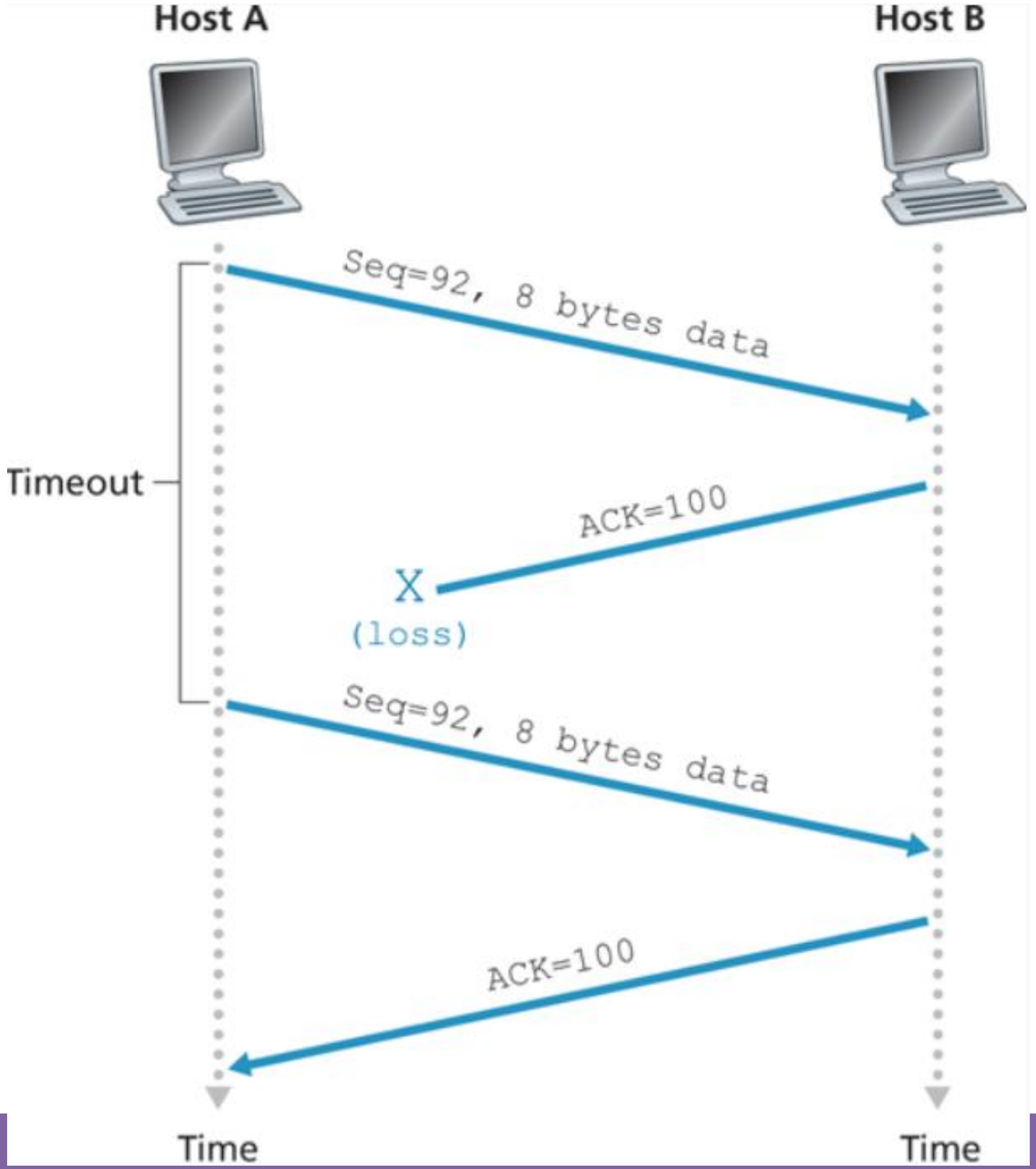
EstimatedRTT + 4*DevRTT

(safety margin)

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

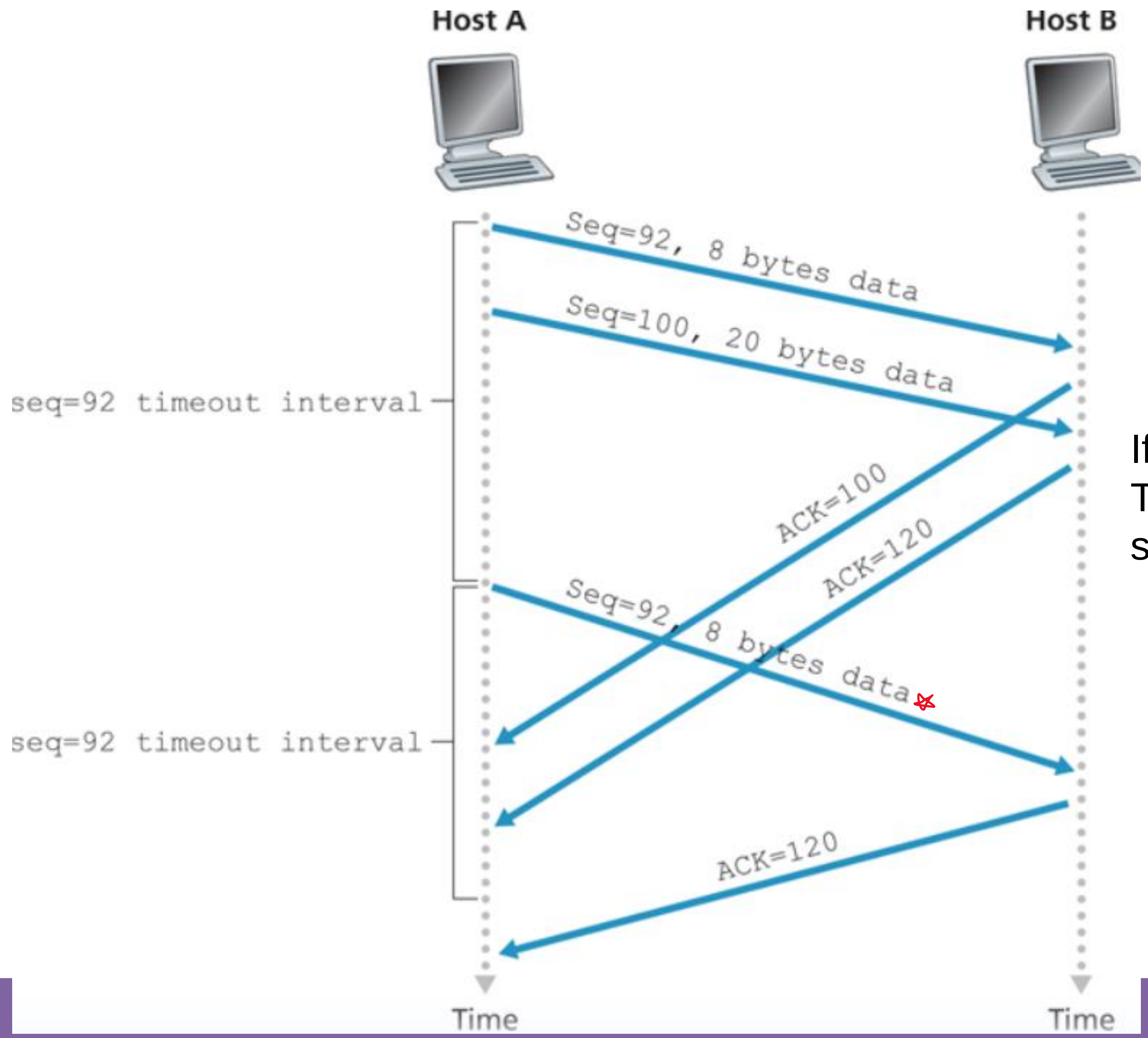
a=0.125

TCP Control Flow



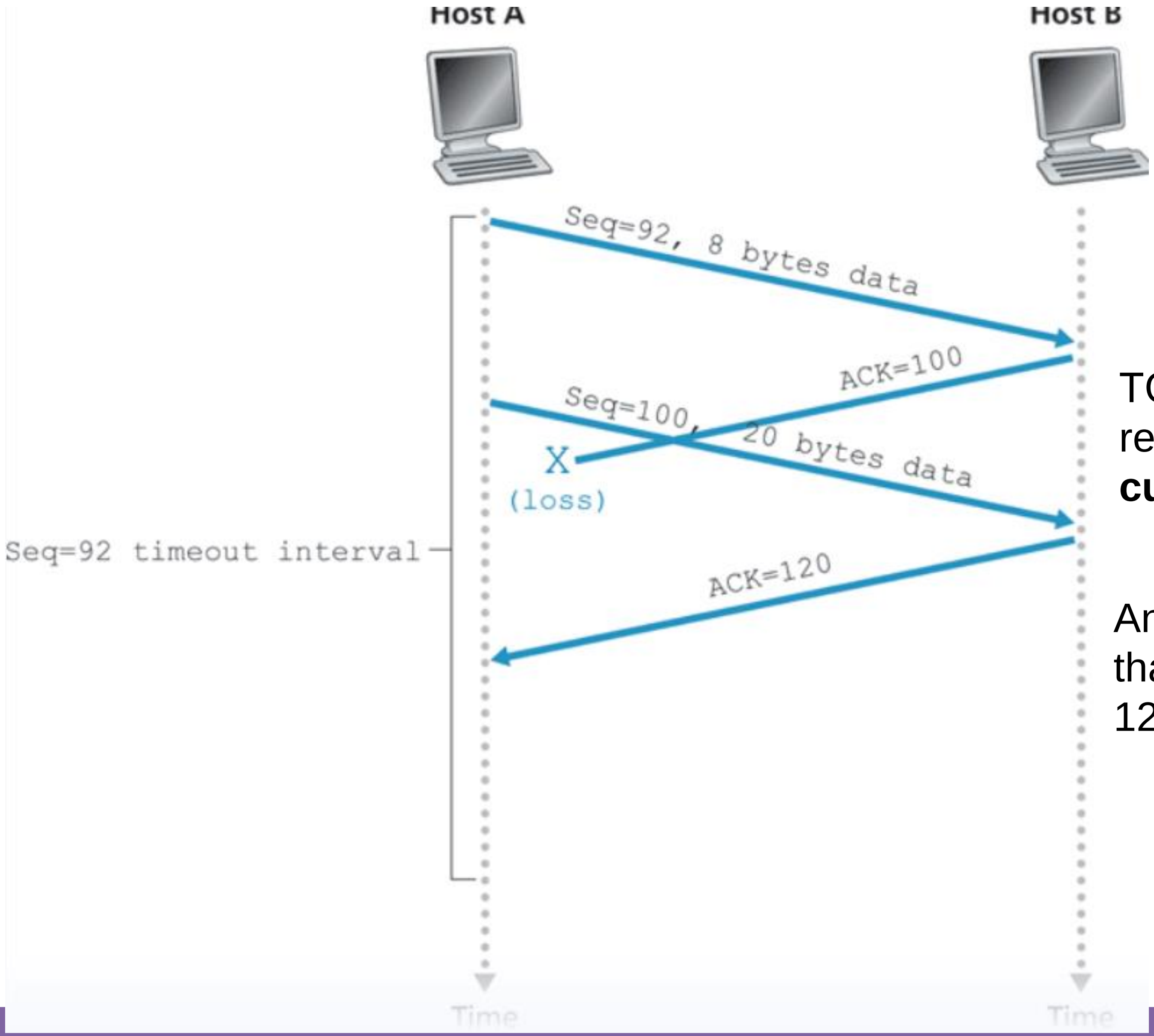
TCP retransmits on ACK loss

TCP Control Flow



If multiple ACKS are lost/late, TCP only resends the first segment in the sequence

TCP Control Flow



TCP won't retransmit if it receives the correct **cumulative knowledge**

An ACK of 120 tells the host that it has *everything* up to 120 bytes

TCP Control Flow

Event	TCP Receiver Action
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged.	Delayed ACK. Wait up to 500 msec for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK.
Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission.	Immediately send single cumulative ACK, ACKing both in-order segments.
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte (which is the lower end of the gap).
Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

Specifics about when/how/why to send ACKs are described in TCP Congestion Control's **RFC (request for comments)**

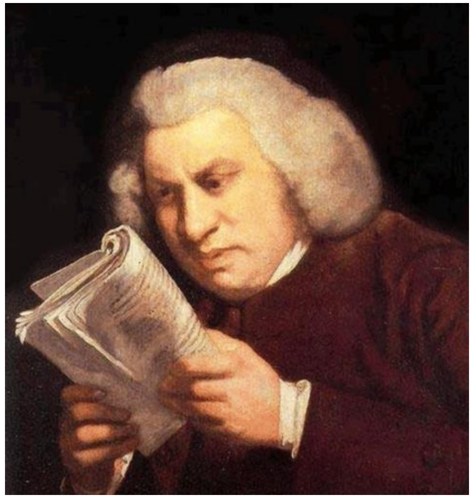
9293

5681

RFCs

```
Network Working Group                                       D. Waitzman
Request for Comments: 1149                                      BBN STC
                                                           1 April 1990


   A Standard for the Transmission of IP Datagrams on Avian Carriers

Status of this Memo

   This memo describes an experimental method for the encapsulation of
   IP datagrams in avian carriers.  This specification is primarily
   useful in Metropolitan Area Networks.  This is an experimental, not
   recommended standard.  Distribution of this memo is unlimited.

Overview and Rational
```

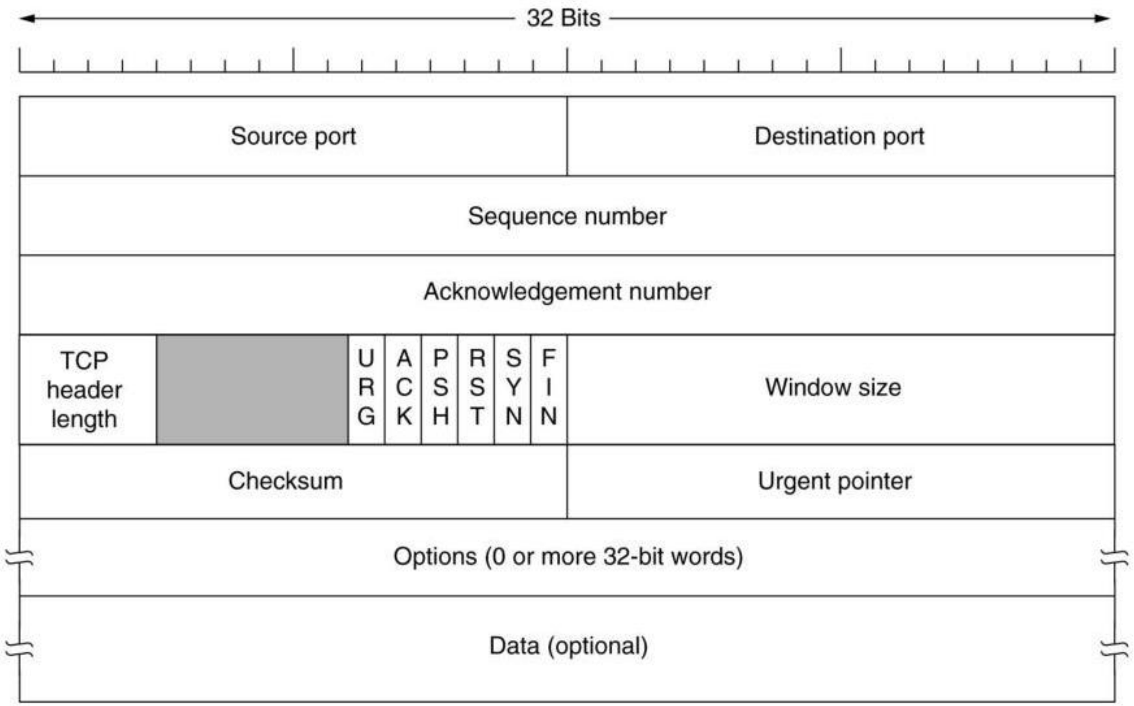
https://www.rfc-editor.org/



TCP Overview

Transmission Control Protocol

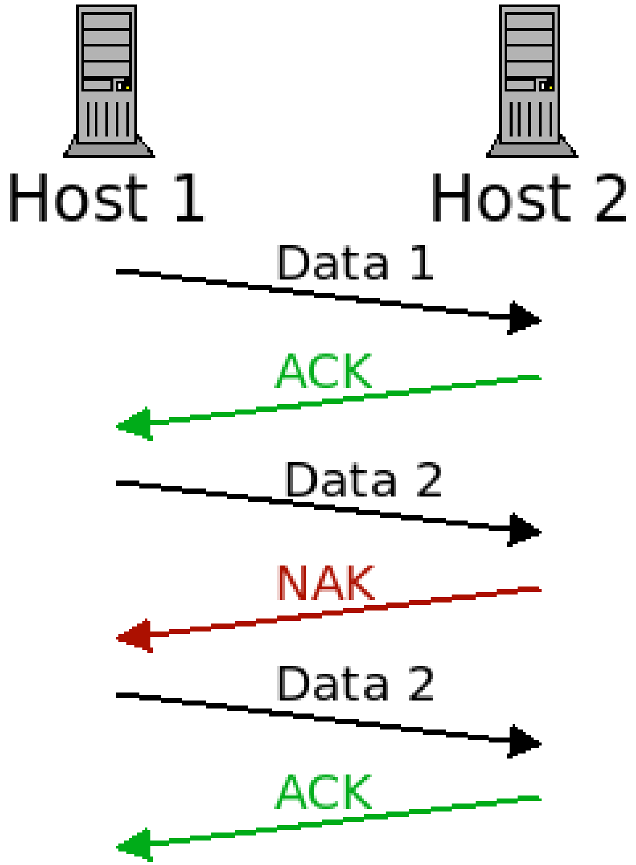
- Segment Structure



TCP header length in 32 bit words,
URG-urgent, ACK- ack number is valid, PSH-push, RST-reset connection,
SYN-used to establish connection, FIN-used to release connection

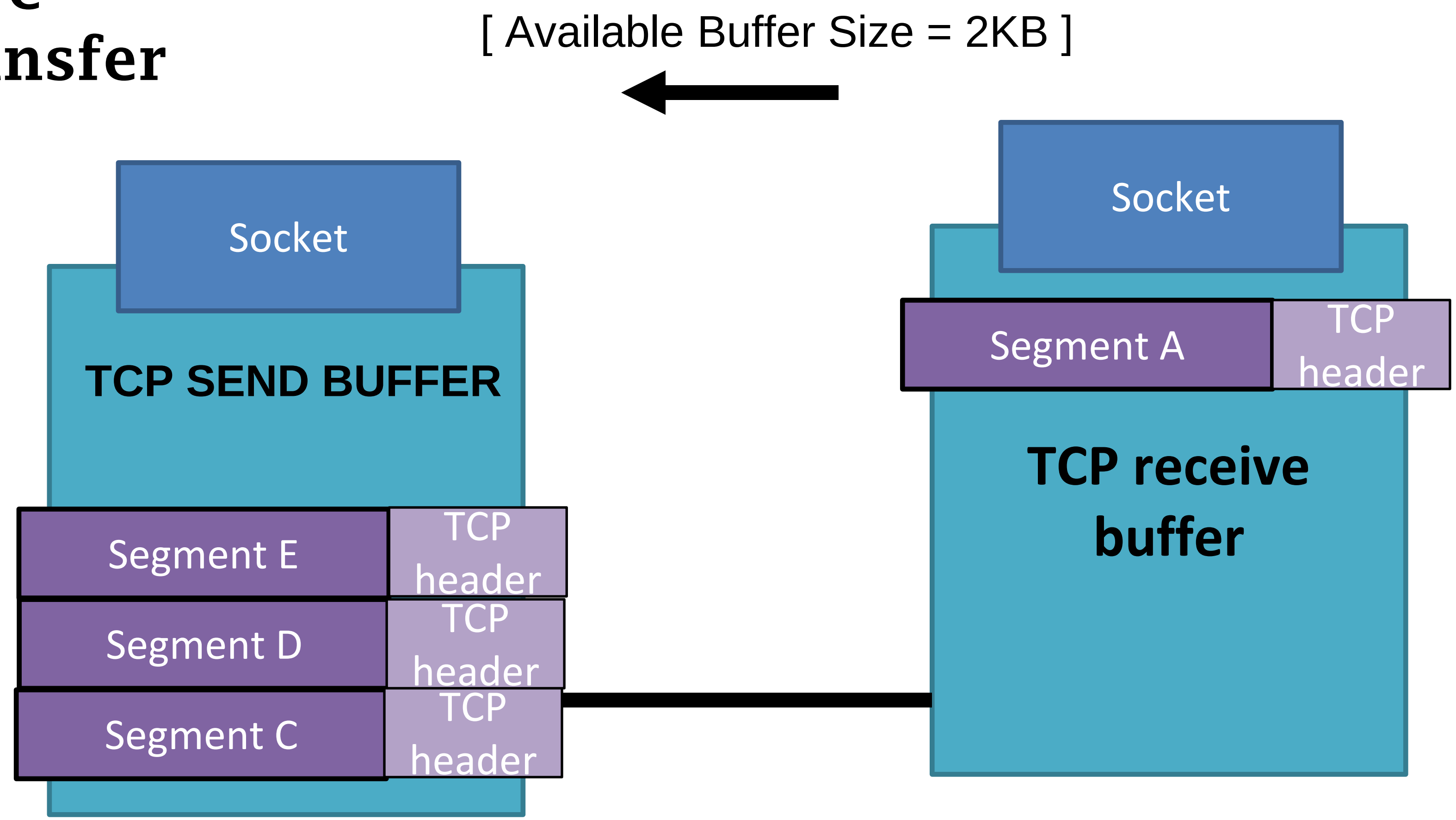
TCP Overview

Transmission Control Protocol
- **Segment Structure**
- **Reliable Data Transfer**



TCP Overview

Transmission Control Protocol
- **Segment Structure**
- **Reliable Data Transfer**
- **Flow Control**



TCP Overview

Transmission Control Protocol

- **Segment Structure**
- **Reliable Data Transfer**
- **Flow Control**
- **Connection Management**