

Computer Networks

Lecture 06

Transport Layer

By
Dr. Khaled ELMenshawy

Lecture

06

LEARNING OUTCOMES

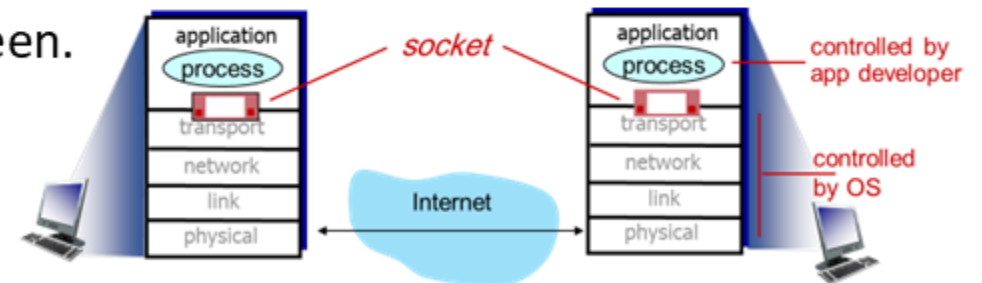
- Define socket and understand socket programming.
- Understand multiplexing / demultiplexing in transport layer.
- Understand UDP and UDP segment structure.
- Understand Checksum and how it works.
- Understand the principles of reliable data transfer.
- Differentiate between different versions of rdt protocol.
- Differentiate between Go-Back-N and selective repeat protocols
- Understand Connection-Oriented Transport: TCP and segment structure.
- Understand congestion and congestion control.

What is socket?

- A socket is the interface between the application layer and the transport layer, also referred to as the **API** (Application Programming Interface) between the applications and the network.
- Two socket types for two transport services:
 - **UDP**: unreliable datagram.
 - **TCP**: reliable, byte stream-oriented.

Application Example (UDP connection):

- Client reads a line of characters (data) from its keyboard and sends data to server.
- Server receives the data and converts characters to uppercase.
- Server sends modified data to client.
- Client receives modified data and displays line on its screen.



Server side:

Import socket library from python.

→ `import socket`

Set socket number

→ `serverPort=10000`

Create UDP socket

→ `serverSocket=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`

Bind socket to local port no. 10000

→ `serverSocket.bind(('',serverPort))`

`print("The server is ready to receive")`

Loop forever (listen to port no. 100000)

→ `while True:`

Read data from UDP socket, getting client's address (client IP and port)

→ `data,clientAddress=serverSocket.recvfrom(2048)`

Decode data and put it in message

→ `message=data.decode("UTF-8")`

`print (message)`

Convert message into uppercase letters

→ `modifiedMessage = message.upper()`

Send modified message to client's address

→ `serverSocket.sendto(modifiedMessage.encode("UTF-8"), clientAddress)`

Client side:

Import socket library from python.

→ `import socket`

Set server name

→ `serverName='localhost'`

Set socket number

→ `serverPort=10000`

Create UDP socket

→ `clientSocket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)`

Get user keyboard input

→ `message=input('Input lowercase sentence: ')`

Attach server name, port to message; send into socket

→ `clientSocket.sendto(message.encode('UTF-8'), (serverName, serverPort))`

Read reply characters from socket into string

→ `data, clientAddress=clientSocket.recvfrom(2048)`

Print data from server

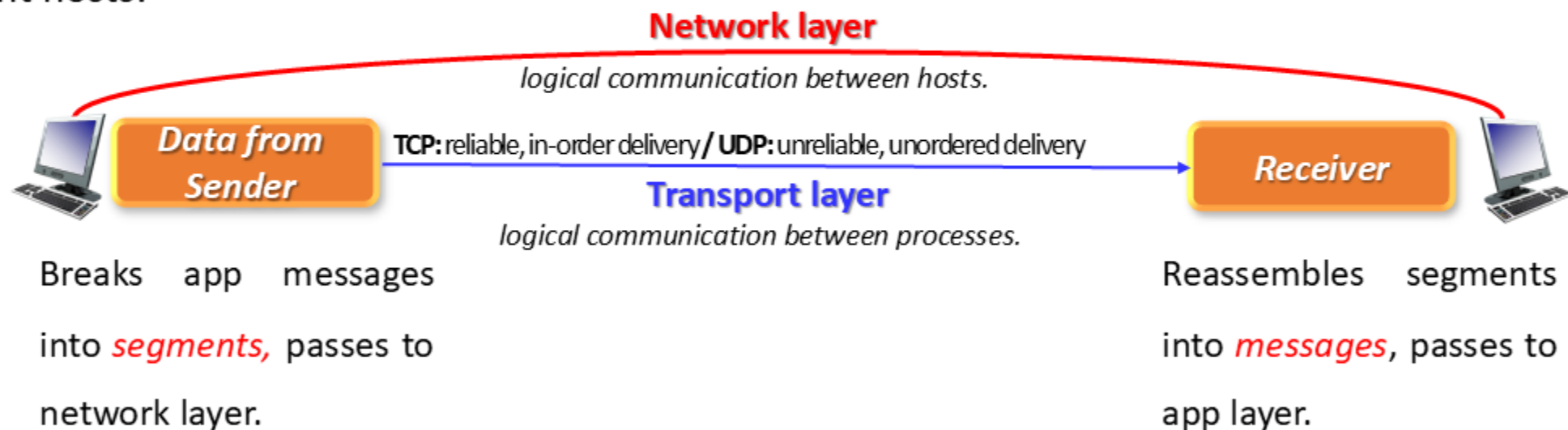
→ `print (data.decode("UTF-8"))`

Close socket

→ `clientSocket.close()`

`message=input('Press enter to exit')`

Transport layer provides end to end logical communication between applications processes running on different hosts.

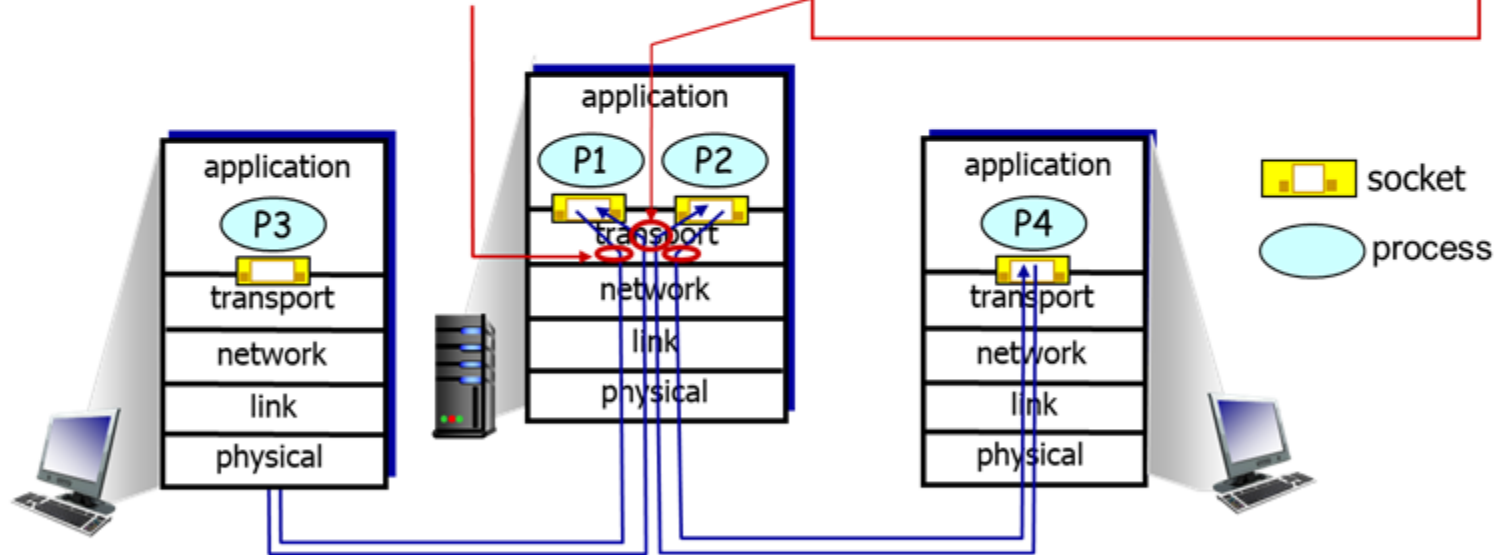


— multiplexing at sender: —

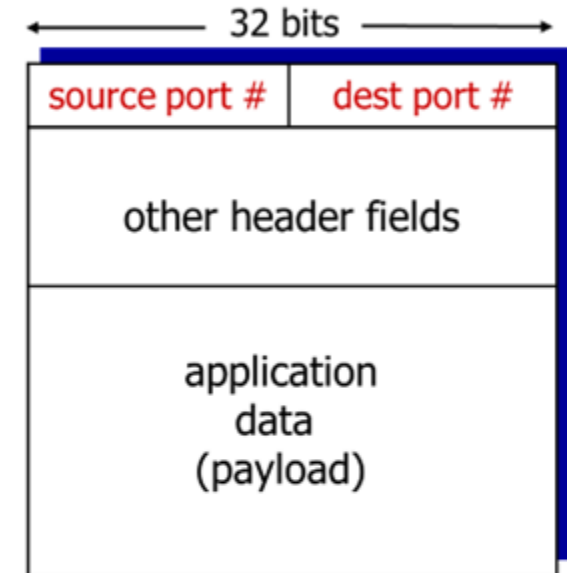
handle data from multiple sockets, add transport header (later used for demultiplexing)

— demultiplexing at receiver: —

use header info to deliver received segments to correct socket



host uses **IP addresses & port numbers** to direct segment to appropriate socket



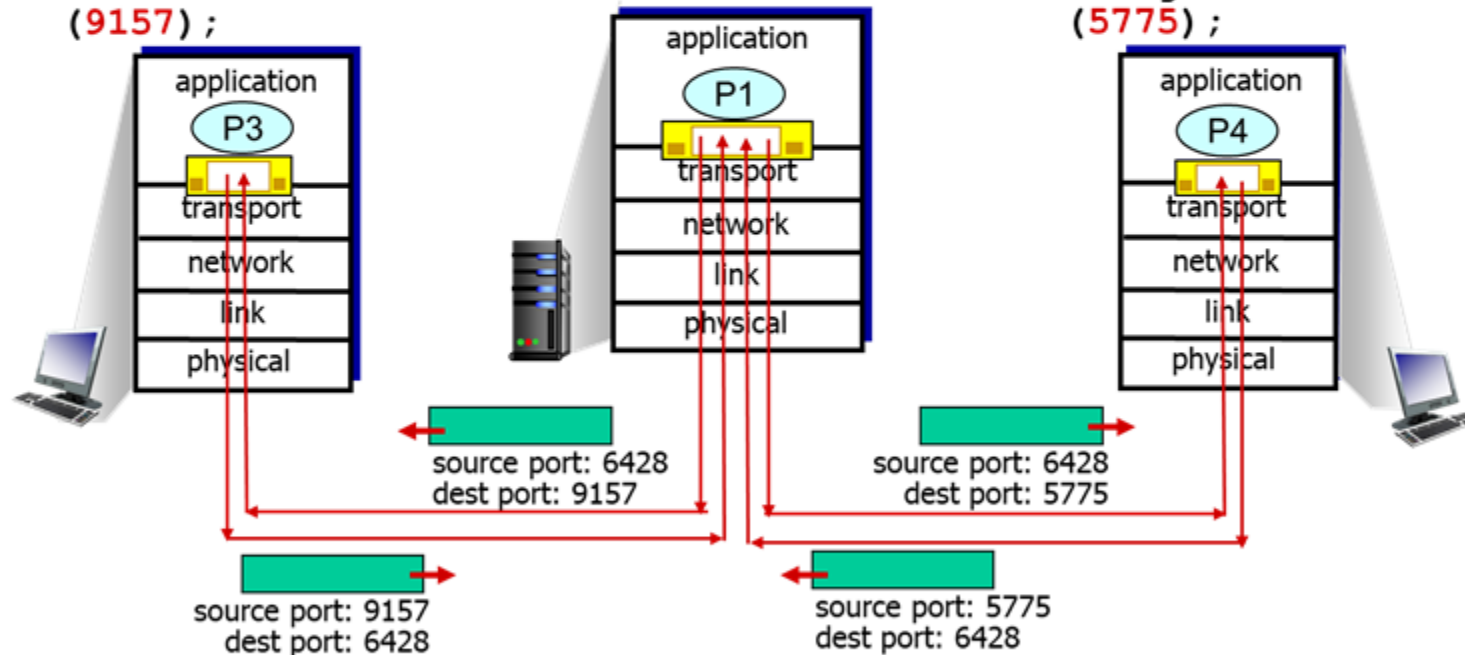
- Each IP packet has source IP address, destination IP address and carries one transport-layer segment.
- Each segment has source, destination port number.

Connectionless demultiplexing

```
DatagramSocket  
mySocket2 = new  
DatagramSocket  
(9157);
```

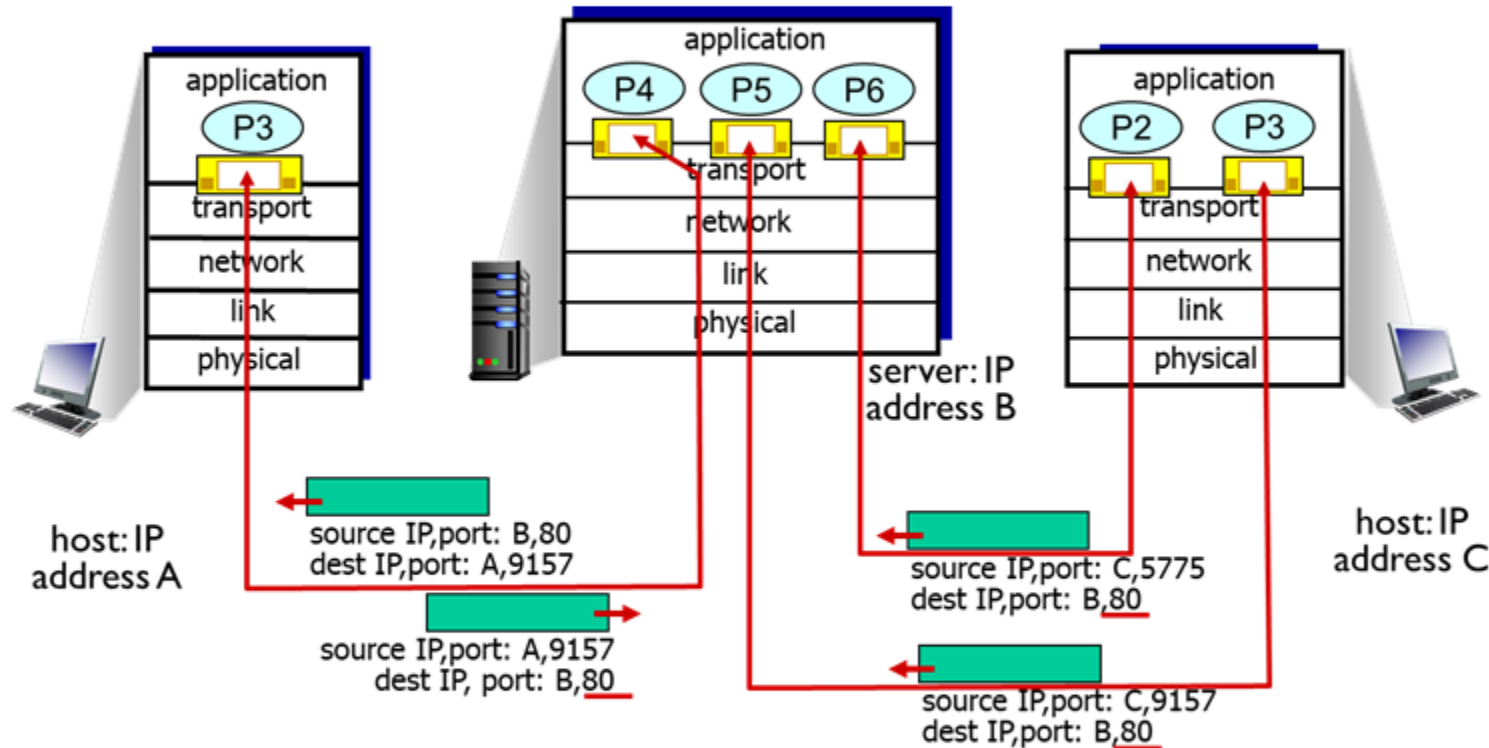
```
DatagramSocket  
serverSocket = new  
DatagramSocket  
(6428);
```

```
DatagramSocket  
mySocket1 = new  
DatagramSocket  
(5775);
```



- *recall*: created socket has host-local port #:
 - `DatagramSocket mySocket1 = new DatagramSocket(12534);`
- *recall*: when creating datagram to send into UDP socket, must specify:
 - destination IP address
 - destination port #
- When host receives UDP segment:
 - Checks destination port # in segment
 - Directs UDP segment to socket with that port #.
 - IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket at dest.*

Connection-oriented demultiplexing



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

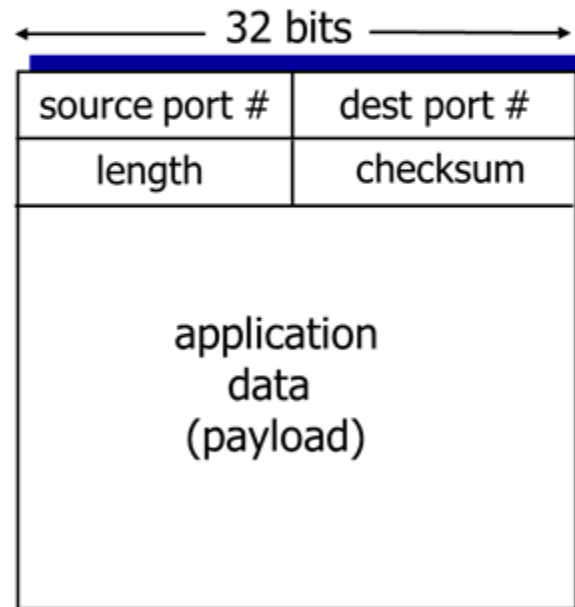
- TCP socket identified by 4-tuple:
 - Source IP address.
 - Source port number.
 - Dest IP address.
 - Dest port number.
- Demux: receiver uses all four values to direct segment to appropriate socket.
- Server host may support many simultaneous TCP sockets each socket identified by its own 4-tuple.
- Web servers have different sockets for each connecting client.

UDP

Characteristics

➤ **Connectionless:**

- No handshaking between UDP sender, receiver.
- Each UDP segment handled independently of others.
- Small header size
- No congestion control: UDP



UDP segment format

Uses

➤ **UDP use:**

- Streaming multimedia
- DNS
- SNMP

Goal: detect “errors” in transmitted segment

➤ **Sender:**

- Treat segment contents, including header fields, as sequence of 16-bit integers (2-Bytes).
- **Checksum:** addition (one's complement sum) of segment contents.
- Sender puts checksum value into UDP checksum field.

➤ **Receiver:**

- Compute checksum of received segment.
- Check if computed checksum equals checksum field value.
- Check If the resultant of the addition of checksum value to the sum of received data equals all one's:
 - **NO** - error detected
 - **YES** - no error detected.

Review of Binary Addition

Rule 1	0 + 0 = 0	
Rule 2	0 + 1 = 1	
Rule 3	1 + 0 = 1	
Rule 4	1 + 1 = 0	Carry 1
Rule 5	1 + 1 + 1 = 1	Carry 1

	1			
	1	1	1	0
+	1	1	0	0
1	1	0	1	0

Result; $(1110)_2 + (1100)_2 = (11010)_2$

- To confirm the binary addition result, simply convert the binary numbers to decimal values and perform decimal addition. If the resultant decimal number from decimal addition equals the resultant decimal number from conversion of resultant binary number, then the addition is correct. For example:
 - $(1110)_2 = (14)_{10}$, and $(1100)_2 = (12)_{10}$. So, the result of decimal addition should be; $(14)_{10} + (12)_{10} = (26)_{10}$.

Example: add two 16-bit integersSender side:

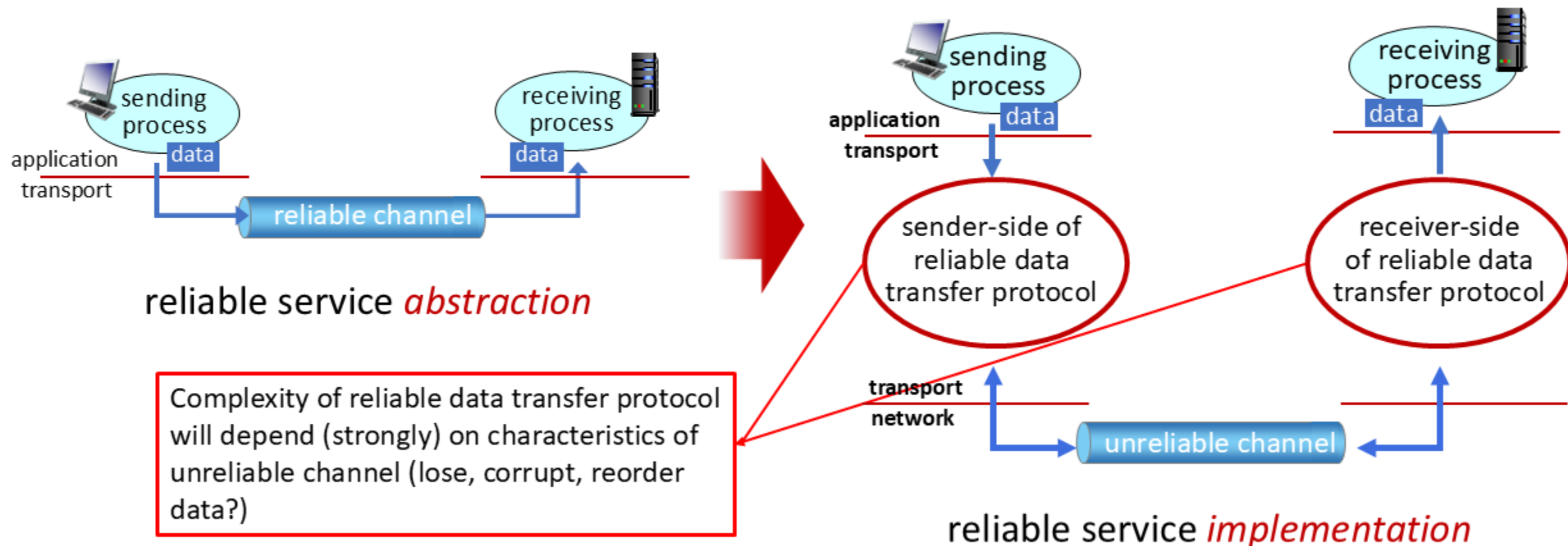
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
																	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

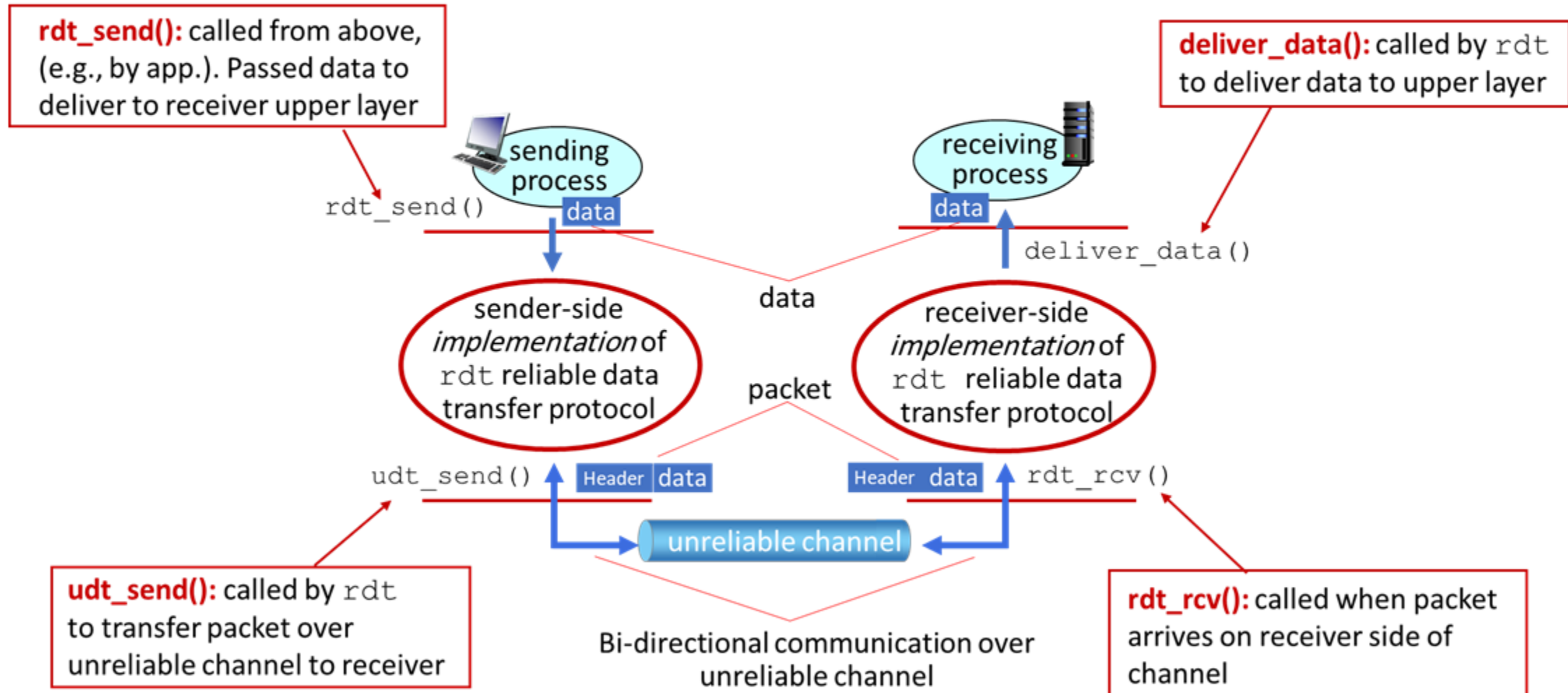
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Example: add two 16-bit integers

Receiver side:

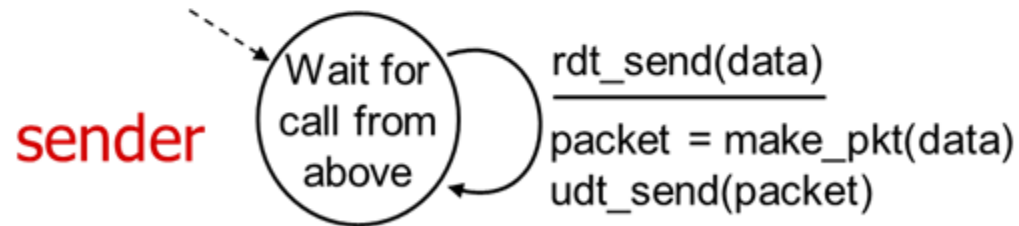
[illegible]





rdt1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
 - No bit errors
 - No loss of packets
- *Separate* Finite State Machines (FSMs) for sender, receiver:
 - Sender sends data into underlying channel
 - Receiver reads data from underlying channel



The question is:

How to recover from errors?



rdt2.0

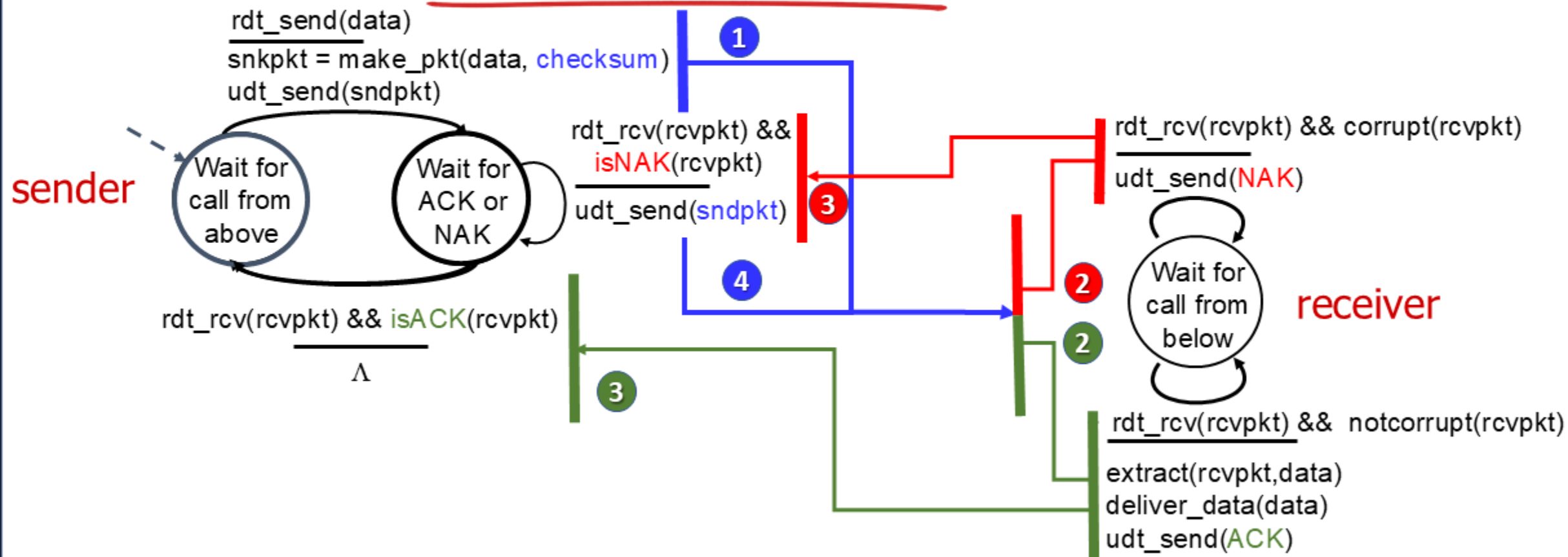


rdt2.0: channel with bit errors**➤ How to recover from errors?**

- **Checksum:** If the underlying channel causes flip bits in packet.
- **Acknowledgments (ACKs):** receiver explicitly tells sender that pkt received OK.
- **Negative Acknowledgments (NAKs):** receiver explicitly tells sender that pkt had errors.
- **Retransmits** pkt by the sender on receipt of **NAK**.

Stop and Wait

Sender sends one packet, then waits for receiver response

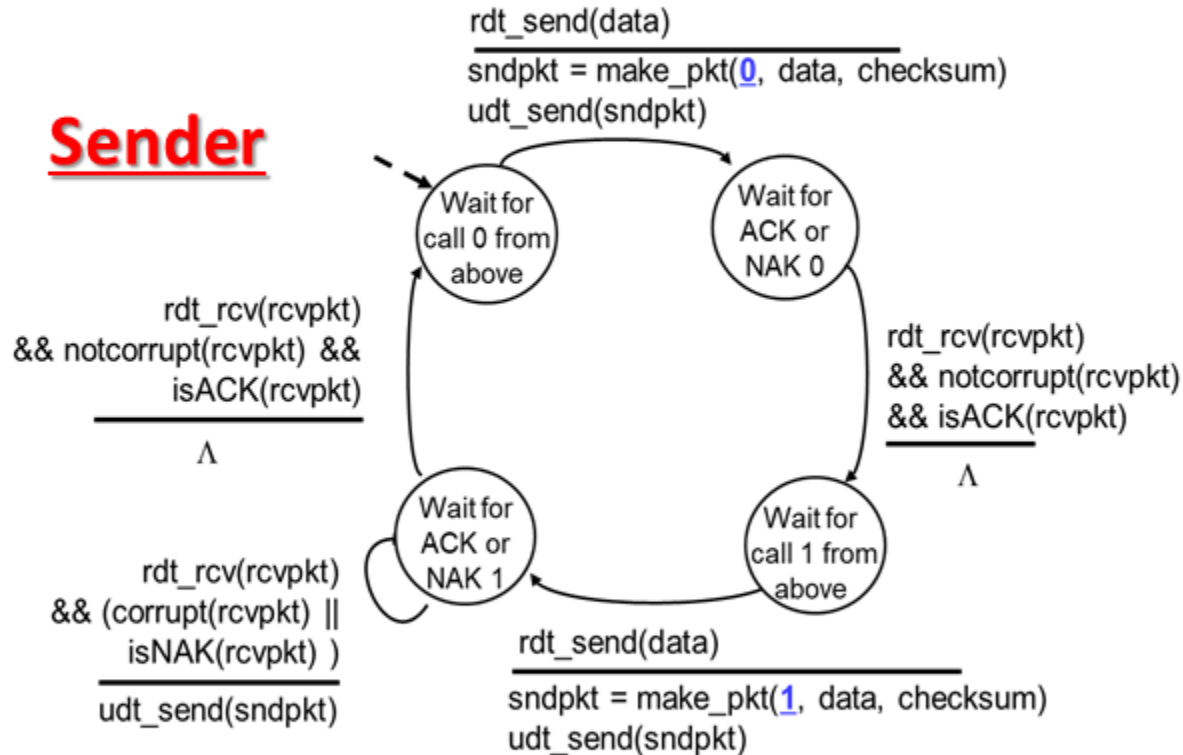
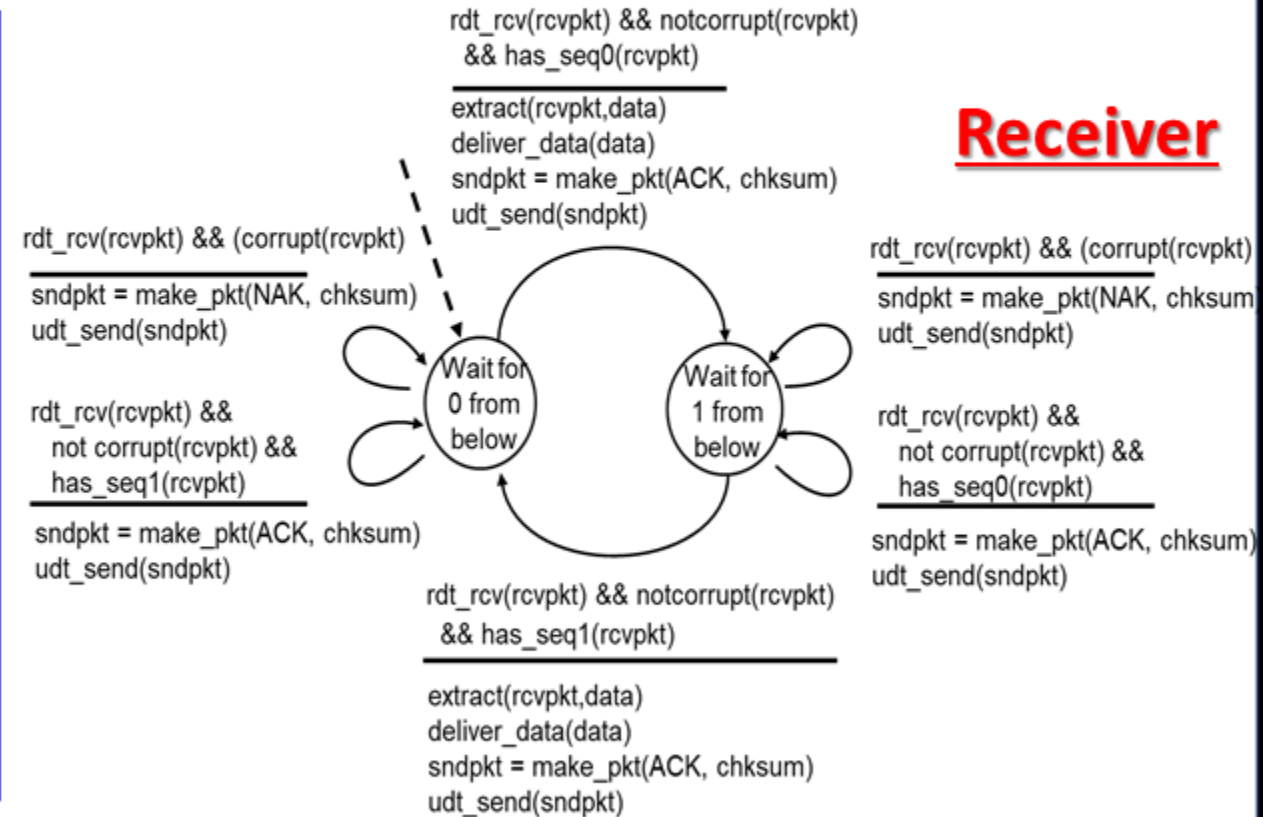
rdt2.0: channel with bit errors

rdt2.0 has a fatal flaw➤ **What happens if ACK/NAK corrupted?**

- Sender doesn't know what happened at receiver!
- Can't just retransmit; possible duplicate.

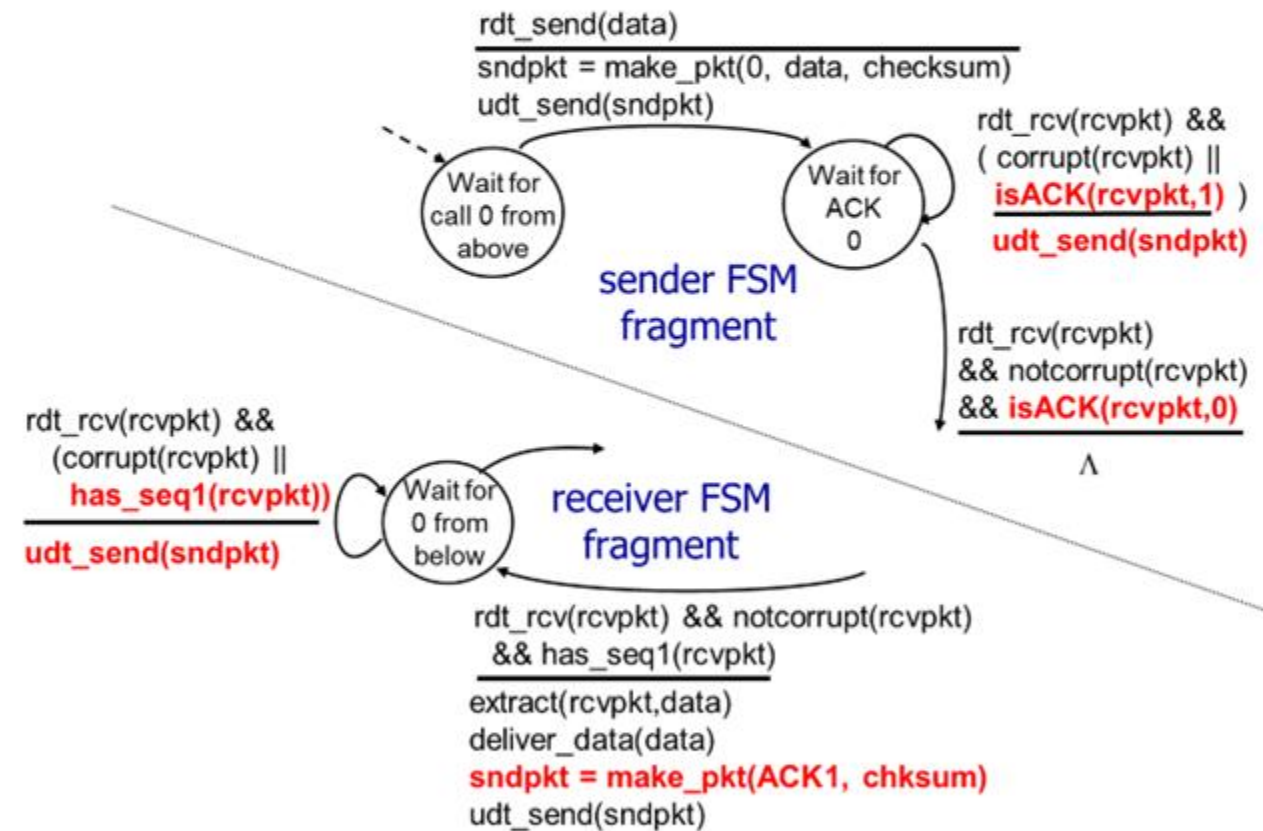
**rdt2.1**➤ **handling duplicates:**

- Sender retransmits current pkt if ACK/NAK corrupted.
- Sender adds *sequence number* to each pkt.
- Receiver discards (doesn't deliver up) duplicate pkt.

rdt2.1: sender, receiver handling garbled ACK/NAKsSenderReceiver

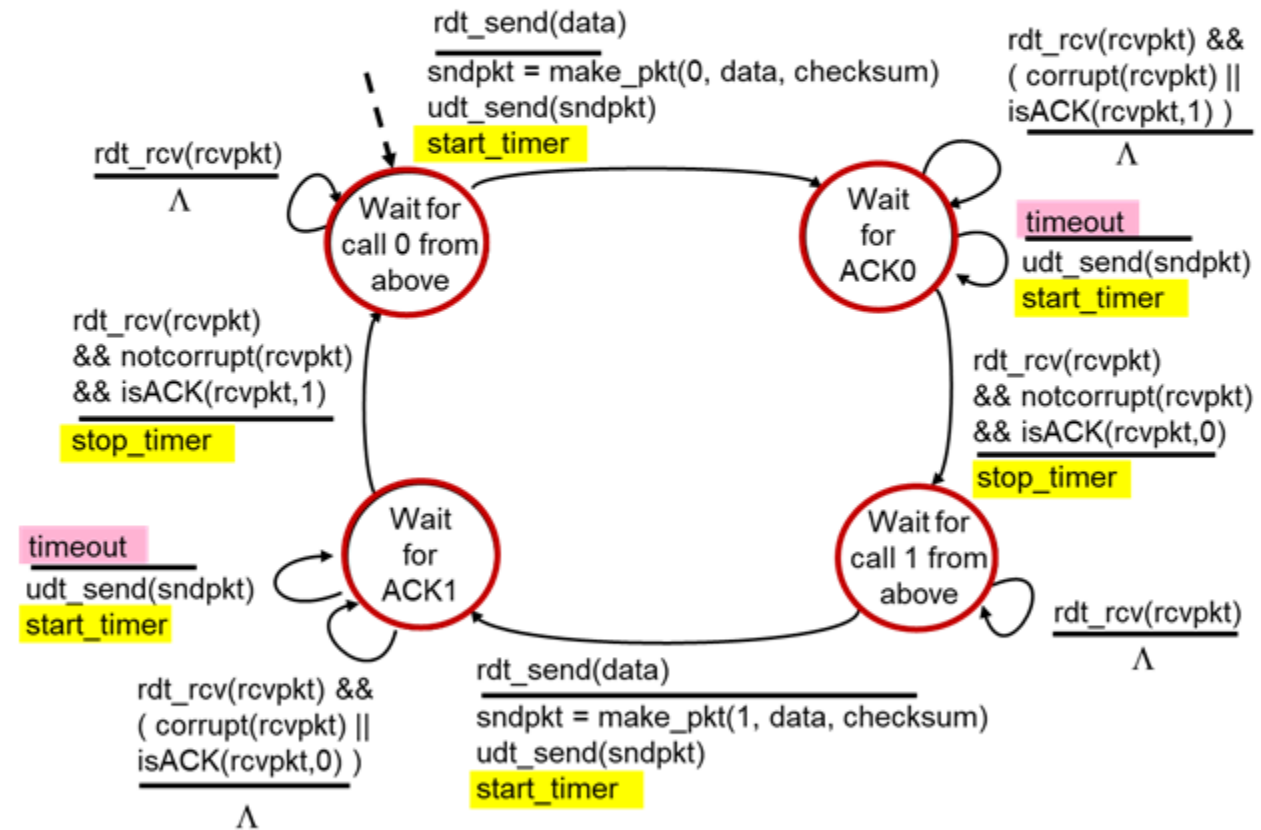
rdt2.2: a NAK-free protocol

- Same functionality as rdt2.1, *using ACKs only*
- Instead of NAK, receiver sends ACK for last pkt received OK
 - Receiver *must include seq #* of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: retransmit current pkt

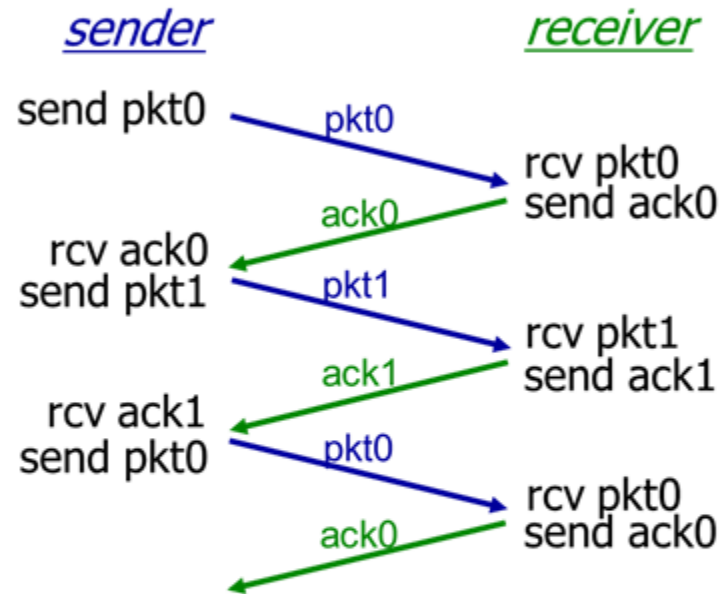


How to recover from data loss?

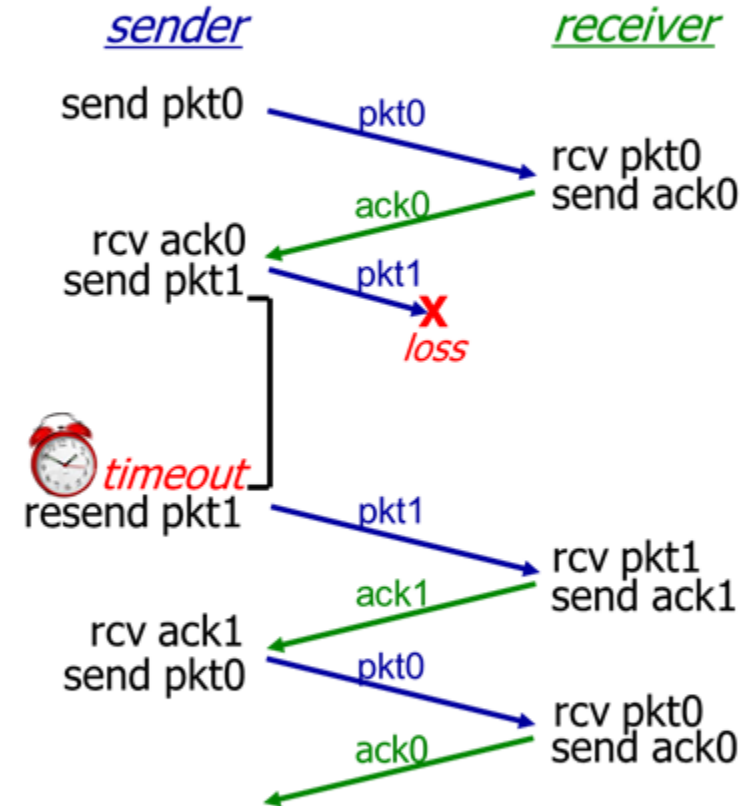
- **Timer:** sender waits "reasonable" amount of time for ACK Retransmits if no ACK received in this time.
- If pkt (or ACK) just delayed (not lost):
 - Retransmission will be duplicate, but **seq #s** already handles this!
 - Receiver must specify seq # of packet being ACKed



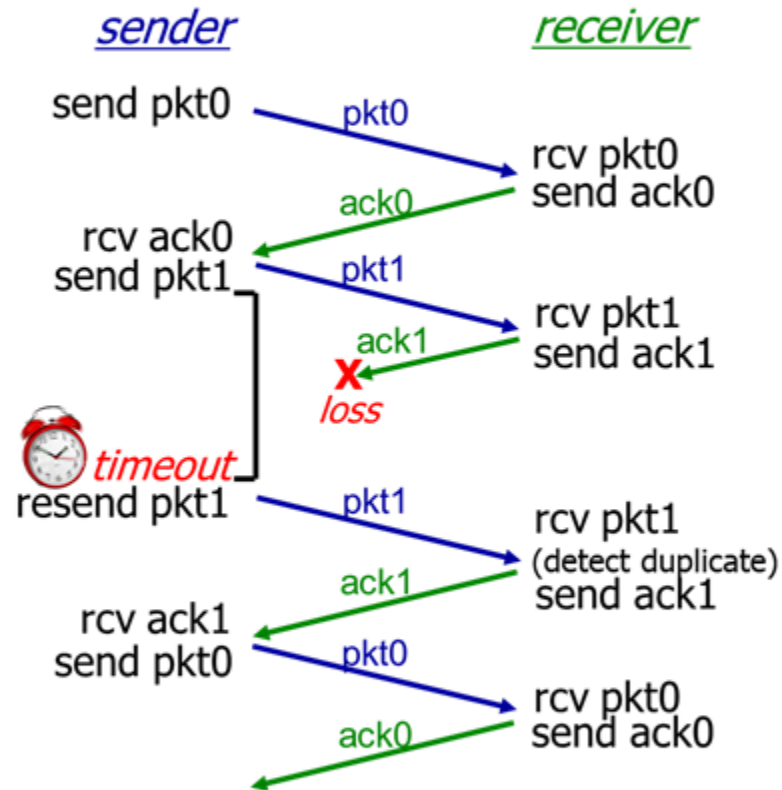
Principles of reliable data transfer

rdt3.0 scenarios

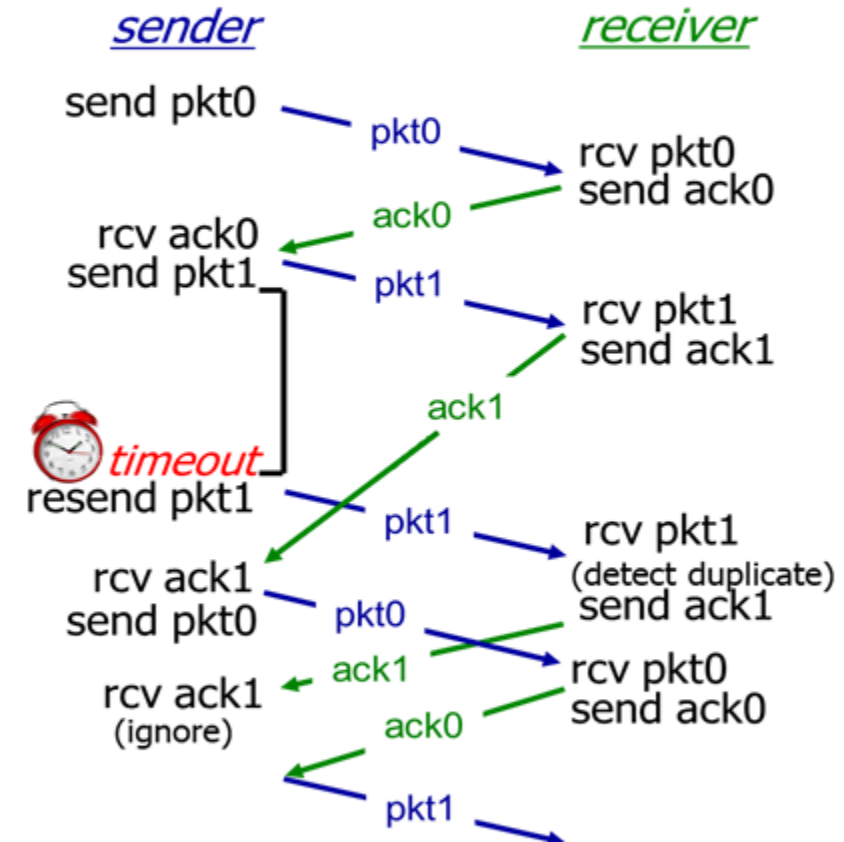
(a) no loss



(b) packet loss

rdt3.0 scenarios

(c) ACK loss



(d) premature timeout/ delayed ACK

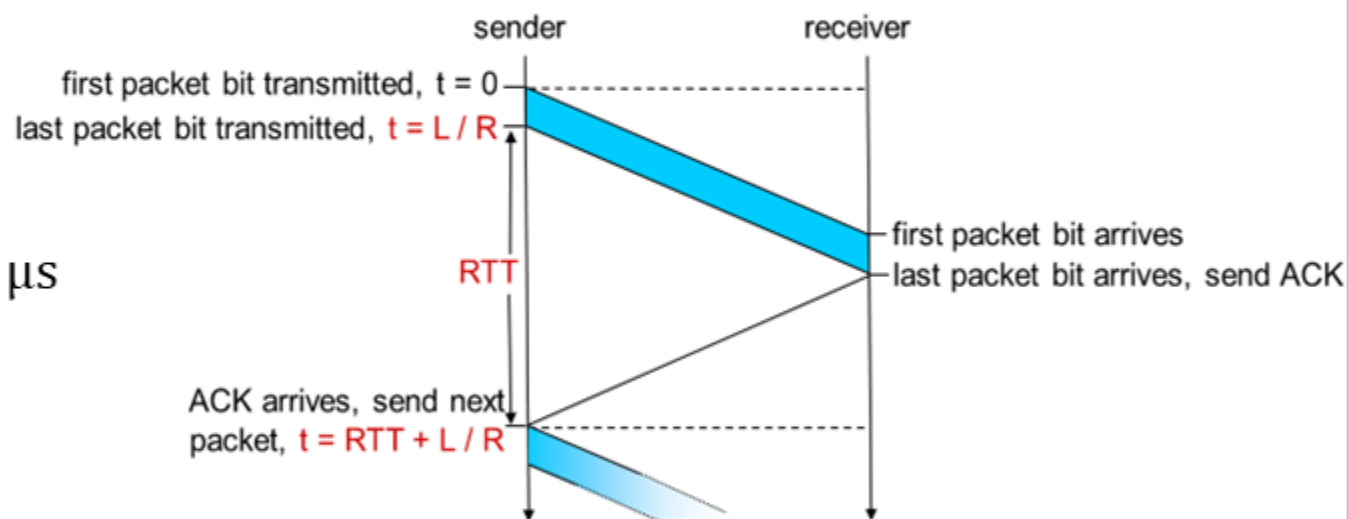
Performance of rdt3.0

- Link speed = 1 Gbps
- propagation delay = 15 ms. (i.e., RTT = 30 ms.).
- Packet size = 8000-bit

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 0.000008 = 8 \mu\text{s}$$

Utilization of the sender:

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30 + 0.008} = 0.00027$$



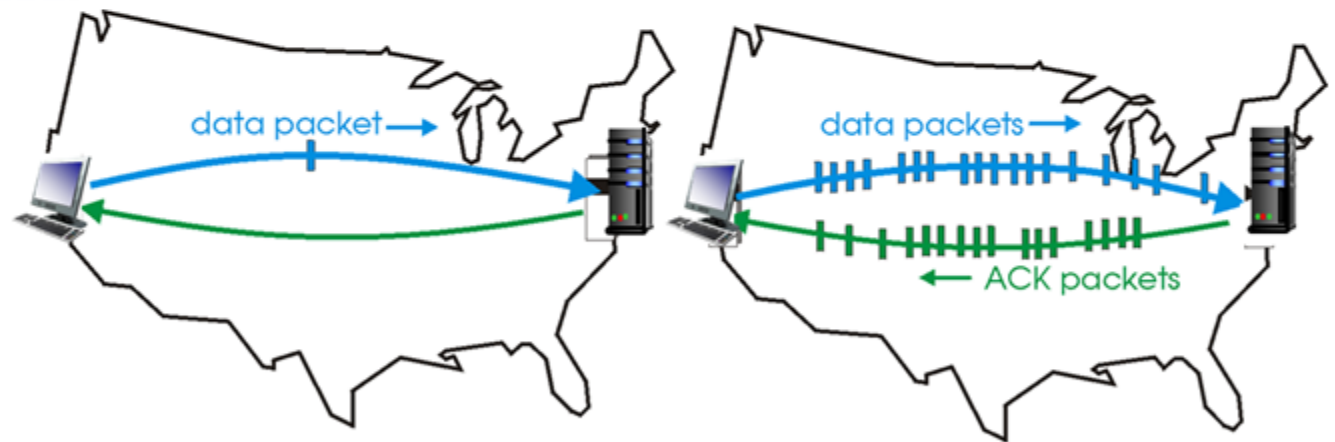
rdt3.0 has a very low performance

Pipelined protocols

Pipelining is the method of sending multiple data units without waiting for an acknowledgment for the first frame sent. Pipelining ensures better utilization of network resources and also increases the speed of delivery, particularly in situations where a large number of data units make up a message to be sent. In this case *range of sequence numbers must be increased*.

Two generic forms of pipelined protocols:

- Go-Back-N.
- Selective repeat.

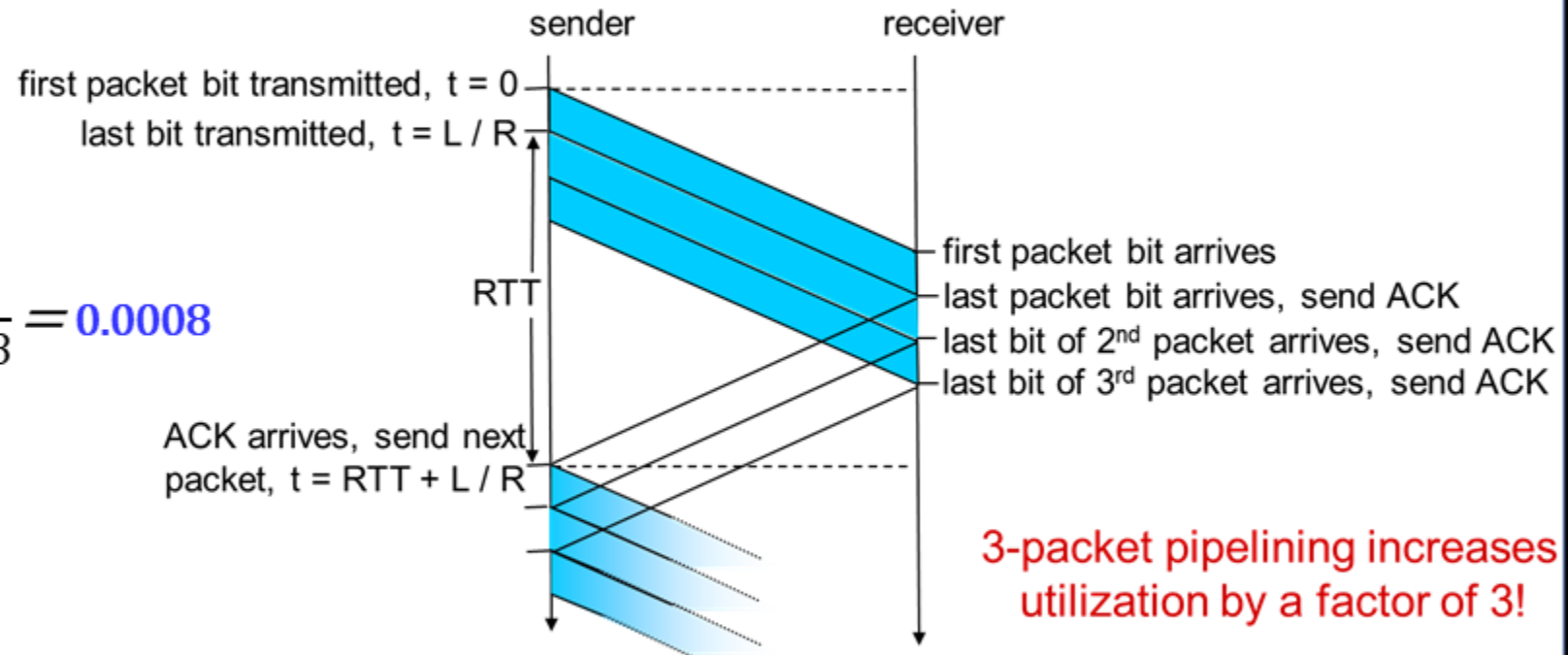


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

Pipelined protocolsUtilization of the sender:

$$U_{\text{sender}} = \frac{3L/R}{RTT + L/R} = \frac{0.024}{30 + 0.008} = 0.0008$$

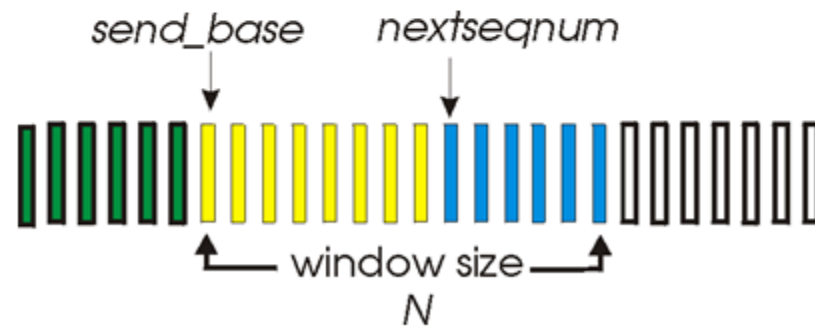


rdt3.0 performance increases

Go-Back-N protocol

- Sender can have up to N- unacknowledged packets in pipeline.
- Receiver only sends *cumulative ack*.
 - Doesn't acknowledge packet if there's a gap.
- Sender has timer for oldest unacknowledged packet.
 - When timer expires, retransmit all unacknowledged packets

- **k-bit** seq # in pkt header.
- "window" of up to N, consecutive unacknowledged packets allowed.
- **Cumulative ACK**: ACK(n): ACKs all packets up to, including seq # n .
 - On receiving ACK(n): move window forward to begin at n+1.
- **Timeout(n)**: retransmit packet n and all higher seq # pkts in window.



Go-Back-N protocolsender window (N=4)

0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8

sender

send pkt0
 send pkt1
 send pkt2
 send pkt3
 (wait)

rcv ack0, send pkt4
 rcv ack1, send pkt5

ignore duplicate ACK



pkt 2 timeout

send pkt2
 send pkt3
 send pkt4
 send pkt5

receiver

receive pkt0, send ack0
 receive pkt1, send ack1

receive pkt3, discard,
 (re)send ack1

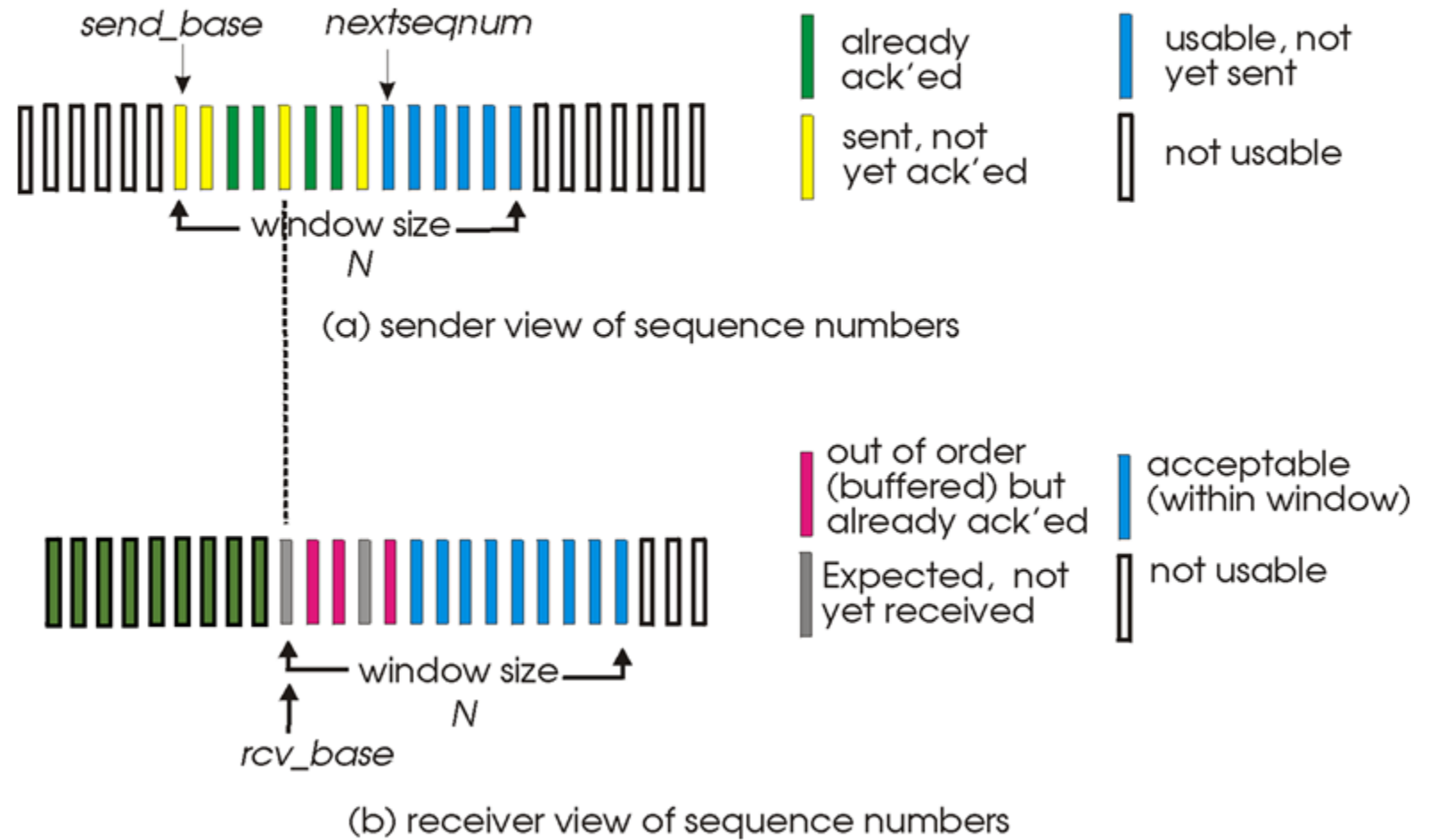
receive pkt4, discard,
 (re)send ack1

receive pkt5, discard,
 (re)send ack1

rcv pkt2, deliver, send ack2
 rcv pkt3, deliver, send ack3
 rcv pkt4, deliver, send ack4
 rcv pkt5, deliver, send ack5

Selective Repeat protocol

- Sender can have up to N- unacknowledged packets in pipeline.
 - Receiver only sends *individual ack* for each packet.
 - buffers packets, as needed, for eventual in-order delivery to upper layer.
 - Sender maintains timer for each unacknowledged packet.
 - When timer expires, retransmit only that unacknowledged packets.
- N consecutive seq #'s
 - Limits seq #'s of sent, unacknowledged packet.
 - ACK: receiver individually acknowledges all correctly received packets.
 - Timeout(n): retransmit only packet n.

Selective Repeat protocol

Principles of reliable data transfer

Selective Repeat protocolsender window (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

record ack3 arrived



pkt 2 timeout

send pkt2

record ack4 arrived

record ack5 arrived

receiver

receive pkt0, send ack0

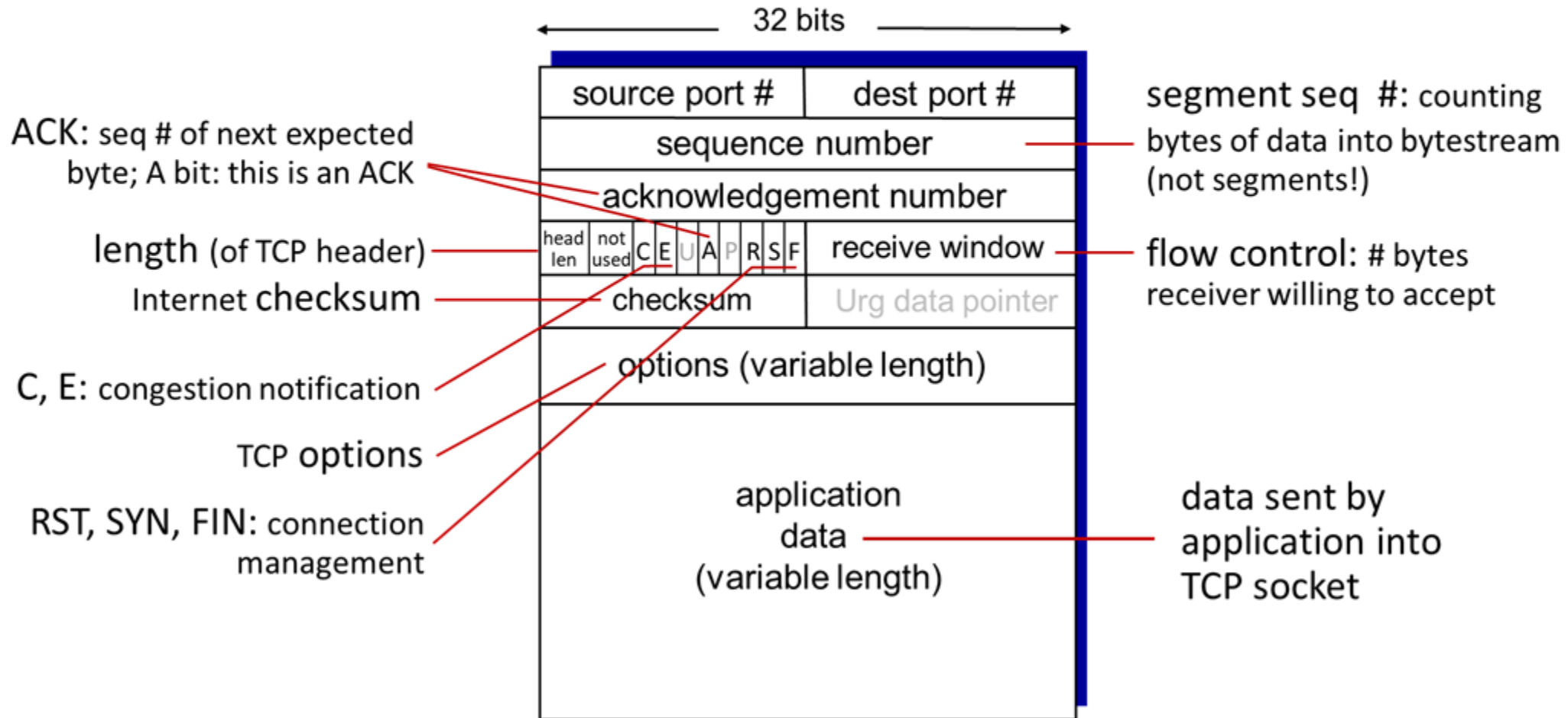
receive pkt1, send ack1

receive pkt3, buffer,
send ack3receive pkt4, buffer,
send ack4receive pkt5, buffer,
send ack5rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2

X loss

TCP: Overview

- **Point-to-Point:**
 - One sender, one receiver
- **Reliable, in-order *byte stream*:**
 - No “message boundaries”
- **Full duplex data:**
 - Bi-directional data flow in same connection
 - MSS: maximum segment size
- **Cumulative ACKs**
- **Pipelining:**
 - TCP congestion and flow control set window size
- **Connection-oriented:**
 - Handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- **Flow controlled:**
 - Sender will not overwhelm receiver

TCP: segment structure

TCP: sequence numbers, ACKs➤ Sequence numbers:

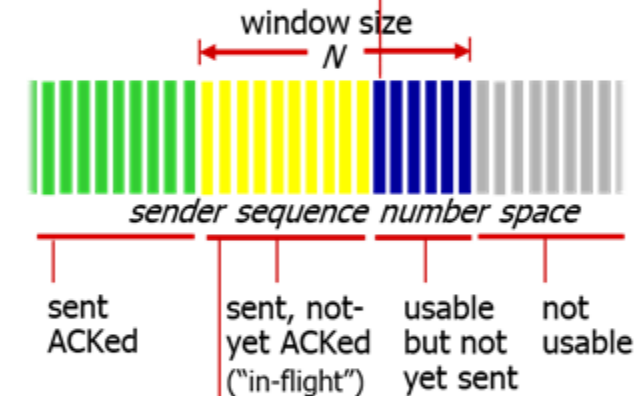
- Byte stream “number” of first byte in segment’s data.

➤ Acknowledgments:

- Seq # of next byte expected from other side
- Cumulative ACK

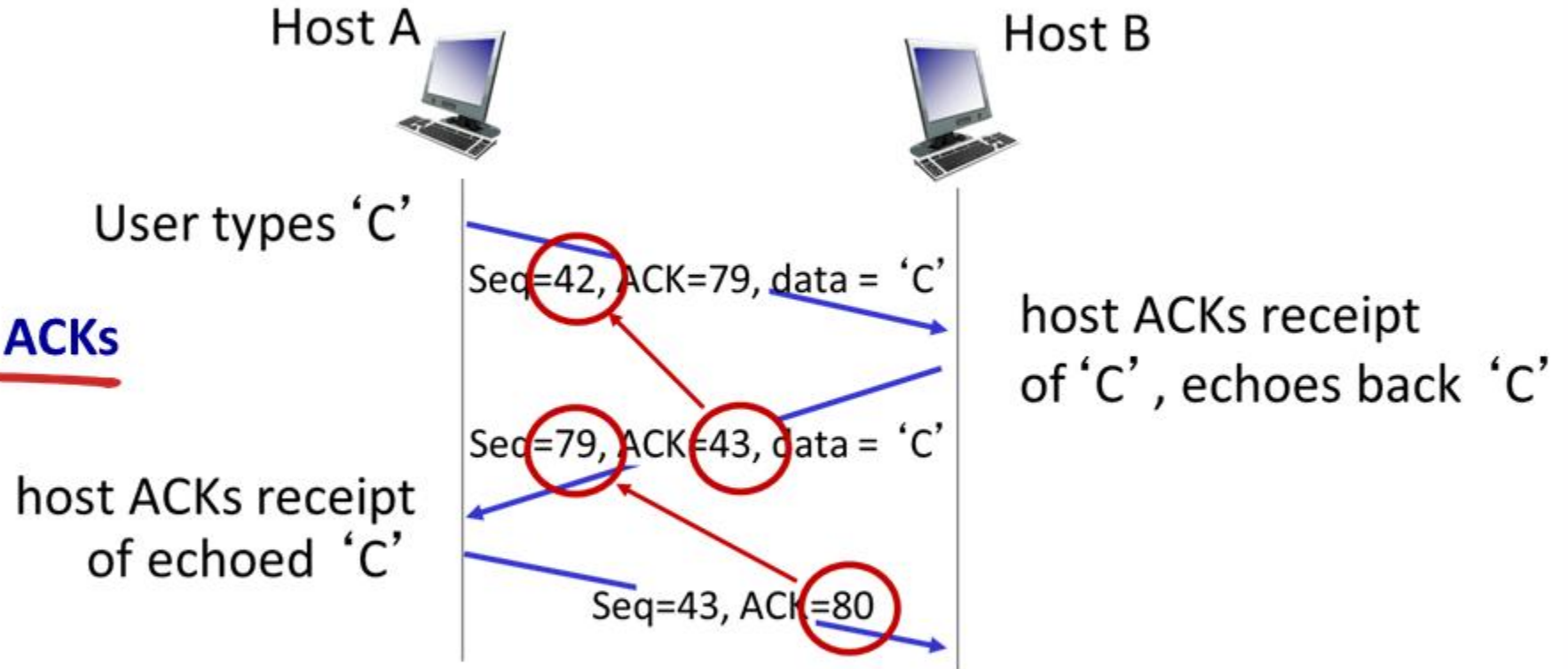
outgoing segment from sender

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



outgoing segment from receiver

source port #	dest port #
sequence number	
acknowledgement number	
	A
checksum	urg pointer

TCP: sequence numbers, ACKs

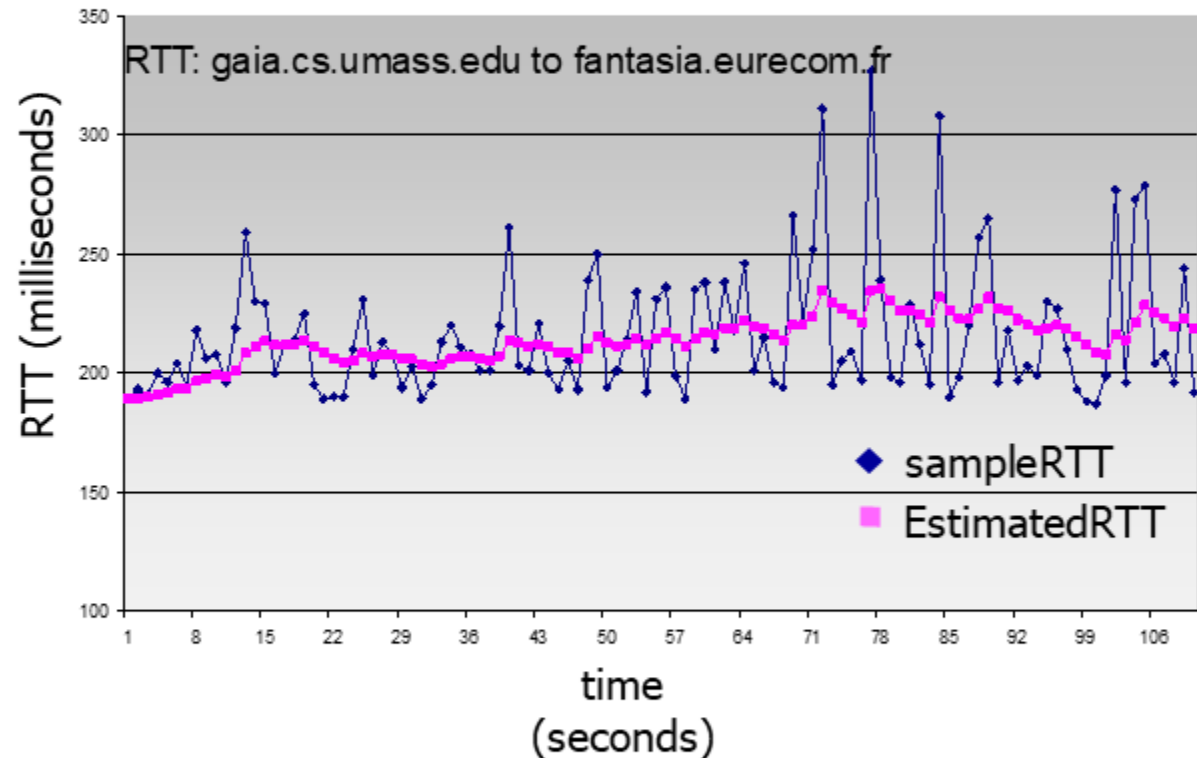
simple telnet scenario

TCP: round trip time, timeout➤ SampleRTT:

- Measured time from segment transmission until ACK receipt.

➤ Estimated RTT :

- Average several recent measurements (last eight), not just current SampleRTT.



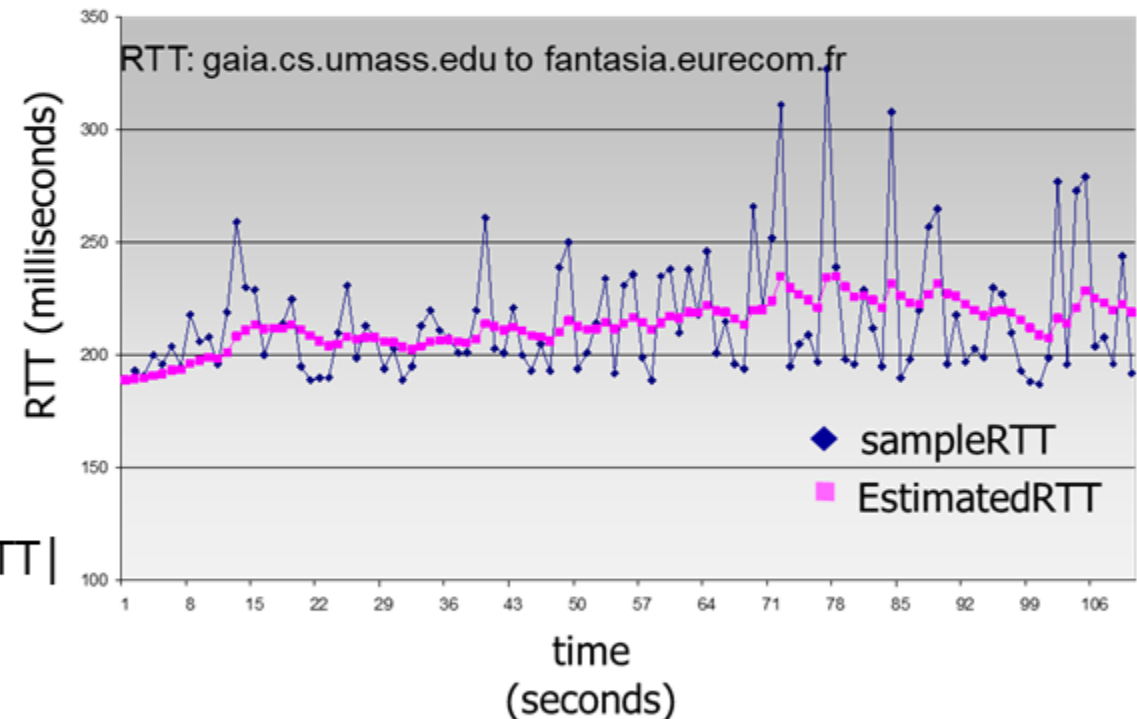
$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT} \quad (\text{typical value: } \alpha = 0.125)$$

TCP: round trip time, timeout➤ Timeout:

- Interval: EstimatedRTT plus “safety margin”.
- TimeoutInterval = EstimatedRTT + 4*DevRTT

➤ DevRTT:

- Dynamic average of last four deviation.
- $\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$
- Typically, $\beta = 0.25$



TCP reliable data transfer➤ TCP creates rdt service on top of IP's unreliable service:

- Pipelined segments.
- Cumulative acks.
- Single retransmission timer.

➤ Retransmissions triggered by:

- Timeout events, long delay before resending lost packet.
- Detect lost segments via duplicate ACKs.

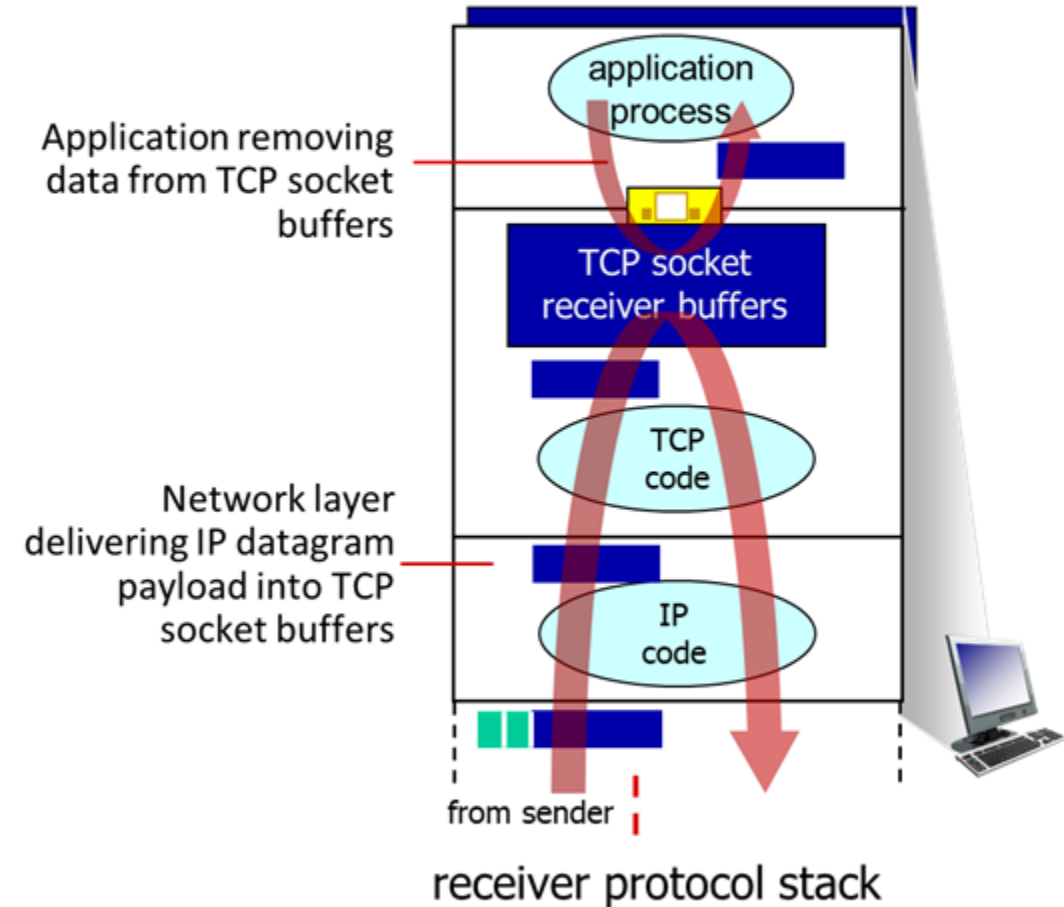
TCP Flow Control

➤ Overflow:

- If network layer delivers data faster than application layer removes data from socket buffers.

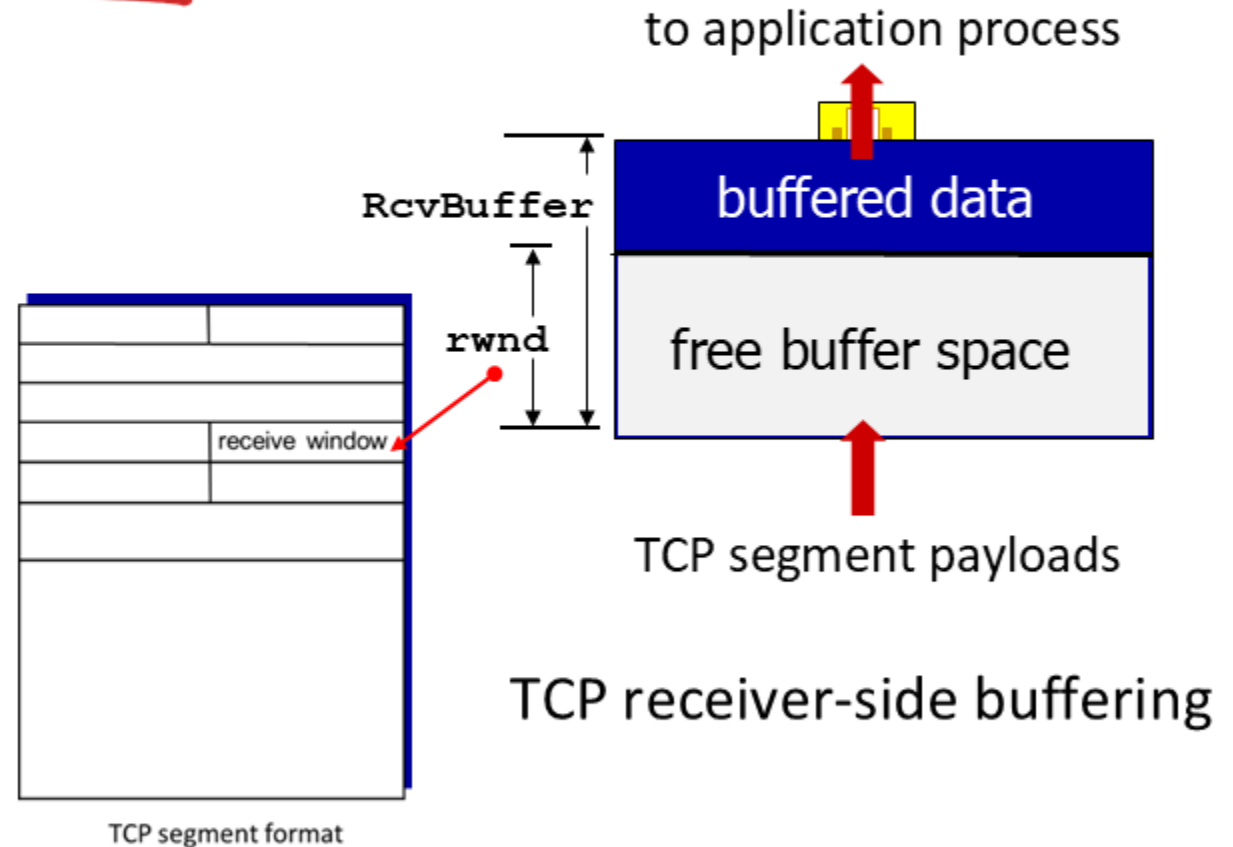
➤ Flow Control:

- Receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast.

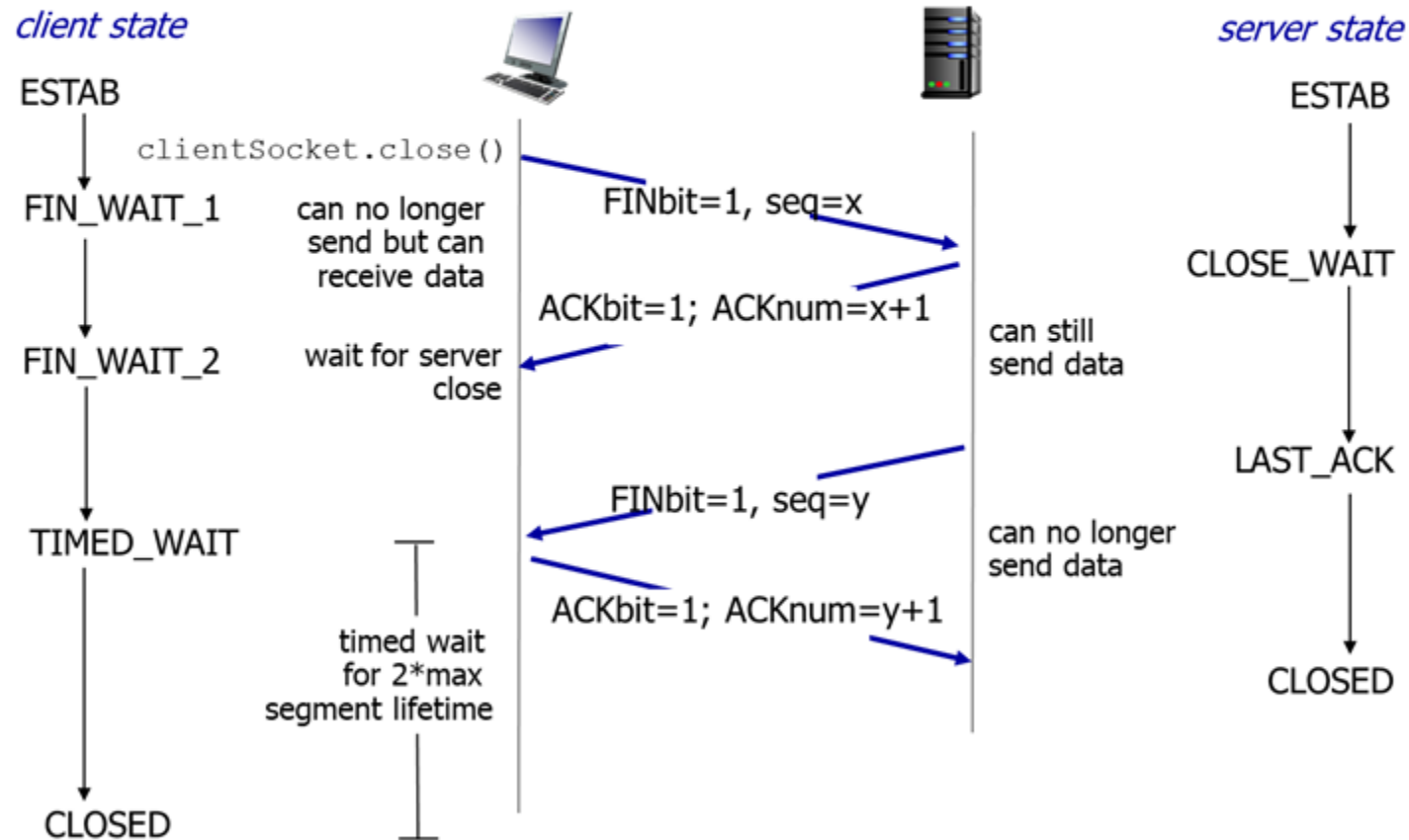


TCP Flow Control

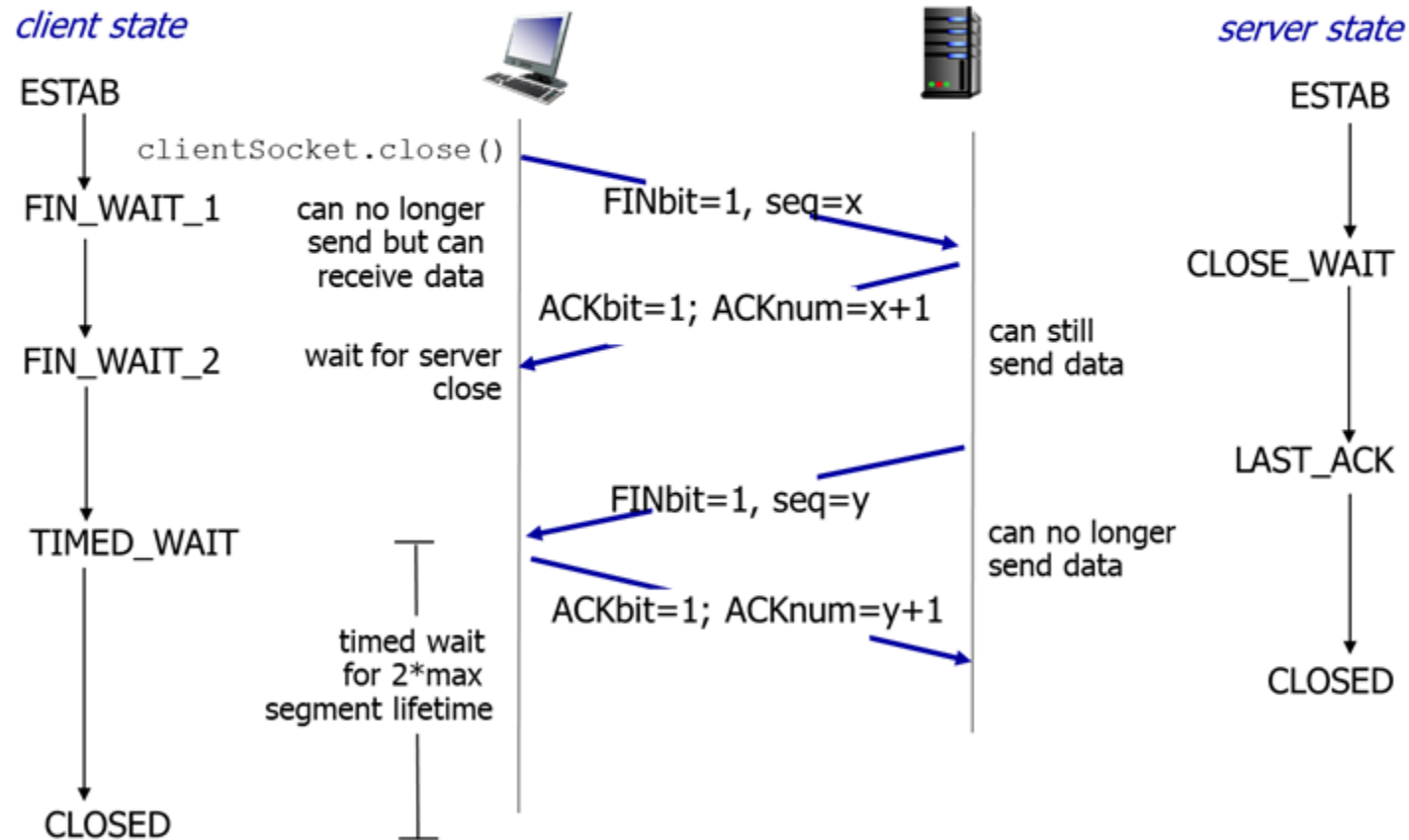
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header.
- **RcvBuffer** size set via socket options (typical default is 4096 bytes).
- Many operating systems auto adjust **RcvBuffer**
- Sender limits amount of unacknowledged data to received **rwnd**.
- Guarantees receive buffer will not overflow.



TCP Connection management; closing a connection



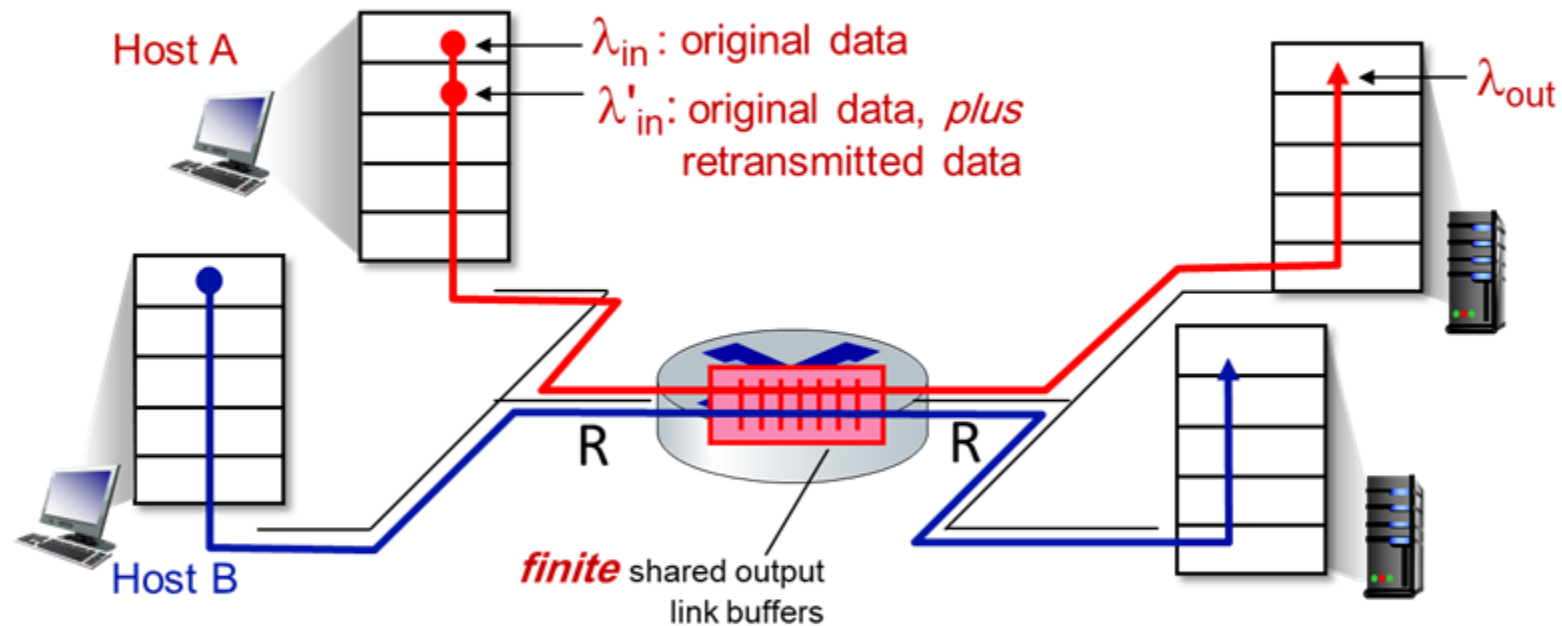
TCP Connection management; closing a connection



What is congestion?

- Simply: “too many sources sending too much data too fast for *network* to handle”
- Different from flow control!
- Manifestations:
 - Lost packets (buffer overflow at routers)
 - Long delays (queueing in router buffers)

- Two senders, two receivers.
- One router, infinite buffers.
- Output link capacity: R .
- No retransmission.



TCP congestion control

- Sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs.
- **Additive increase**: increase **cwnd** by 1 MSS every RTT until loss detected
- **Multiplicative decrease**: cut **cwnd** in half after loss.

