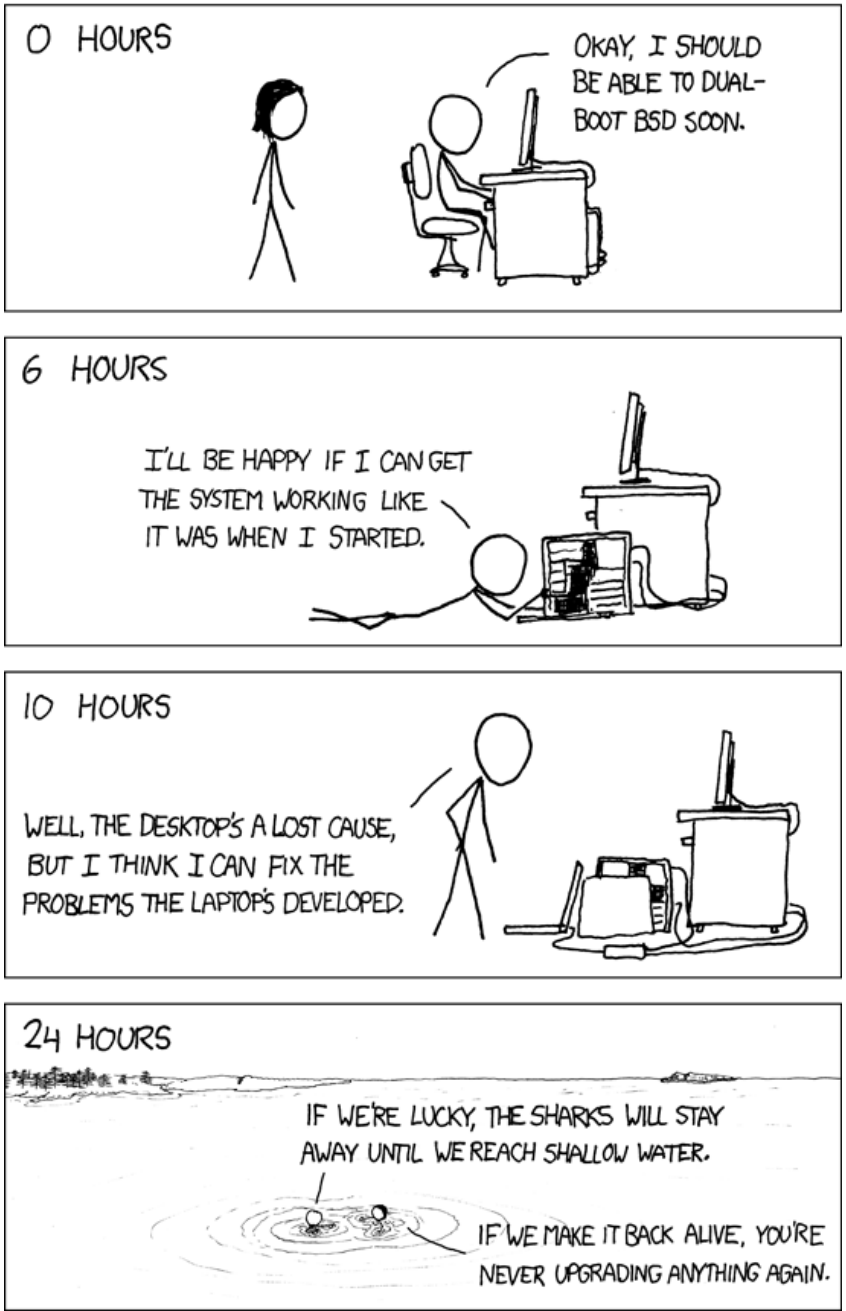
AS A PROJECT WEARS ON, STANDARDS FOR SUCCESS SLIP LOWER AND LOWER.



https://xkcd.com/349/

CS 60:
Computer Networks

Transport layer: TCP

Review from last class

Two options at Layer 4 (Transport Layer)

- UDP – not connection oriented, missing or corrupt data dropped, faster, 8-byte header
- TCP – connection oriented, data delivered in order with no missing packets, but slower

Layer 4 ports connect processes, mux/demux (Layer 3 IP address connect hosts)

UDP

- A UDP socket is identified by the destination IP address and destination port number
- UDP is "connectionless", no persistent connection is established
- Socket listens for and receives all datagrams sent to its specific port by any sender
- Calling *recvfrom* reads bytes from only one datagram
- Can reply to the correct sender because each received datagram contains the sender's IP address (Layer 3 header) and port (Layer 4 header)

TCP

- A server creates a "listening socket" bound to a specific port (e.g., 80 for a web server)
- When a client initiates a connection, the server's OS creates a *new*, dedicated socket for that specific connection
- This new socket is uniquely identified by the **4-tuple**: (src IP, src Port, dst IP, dst Port)
- A single server can handle multiple simultaneous connections by de-muxing each incoming packet to the unique socket
- Bytes from multiple segments are concatenated into a byte stream (no segment boundaries)
- Calling *recv* may read bytes from multiple segments or only part of one segment

Review: Transport layer moves segments (or datagrams) across a network

Conceptual network layers

7) Application	Interacts with application programs to send ***messages*** Applications assigned a port, multiple instances can run (many browser pages) Examples: HTTP, SSH, FTP, SMTP, DNS
4) Transport	Moves ***segments*** *or datagrams* May provide error control, flow control, application addressing (ports) Examples: **TCP** (connection-oriented), UDP (connectionless) TCP provides sequencing, dropped packet resend, traffic congestion routing
3) Network (IP)	Moves ***packets*** between local area networks (routing) Each computer on the Internet identified by an IP address (IP v4 or v6) Also called Layer 3 or IP layer (ICMP Ping is here)
2) Link (MAC)	Moves ***frames*** within a local area network (switching) Each computer identified by a MAC address on its Network Interface Card (NIC) Also called Layer 2, MAC layer, Data Link layer, or Ethernet layer
1) Physical	How data is physically transmitted • Transmitter converts logical 1 and 0 ***bits*** to electrical/light pulses or phase/amplitude of radio frequency (RF) and sends down wire or over air • Receiver converts electrical/light or RF back to logical 1 and 0 bits

https://www.ibm.com/think/topics/osi-model

Agenda



1. TCP connections
2. TCP header bytes
3. TCP code
4. Handling multiple clients
5. Supplemental material

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 defines max payload size
    - Max Link Layer size (1500 bytes in Ethernet)
    - Minus TCP and IP header size (normally 20 bytes each)
    - MSS typically = 1460 bytes
  - Large messages broken into pieces of MSS 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

Layer 4 TCP is a logical connection between *processes*

Layer 3 is a logical connection between *hosts*

Layer 3 routers (mainly) do not see Layer 4 information

RFCs: 9293, 1122, 2018, 5681, 7323

What could go wrong when a sender transmits a packet to a receiver?

Things that might go wrong

1. Data could be corrupted (bit flips)
2. Packet could get lost
3. Packets could get delivered out of order

Reliable Data Transfer (Layer 4) attempts to deal with each of these possibilities

But the channel (Layer 3) between sender and receiver is unreliable!

TCP attempts to provide reliable data transfer over an unreliable network

TCP handles the three big things that can go wrong

Issue	Solution
Data could be corrupted (bit flips)	Use checksum to detect errors • Drop packet if errors detected • Dropped packet now the same as a lost packet

TCP handles the three big things that can go wrong

Issue	Solution
Data could be corrupted (bit flips)	Use checksum to detect errors • Drop packet if errors detected • Dropped packet now the same as a lost packet
Packets could get lost	Receiver acknowledges receipt of bytes (not packet number!) Sender starts a timer when sending bytes If no ACK received for bytes, resend if timer expires

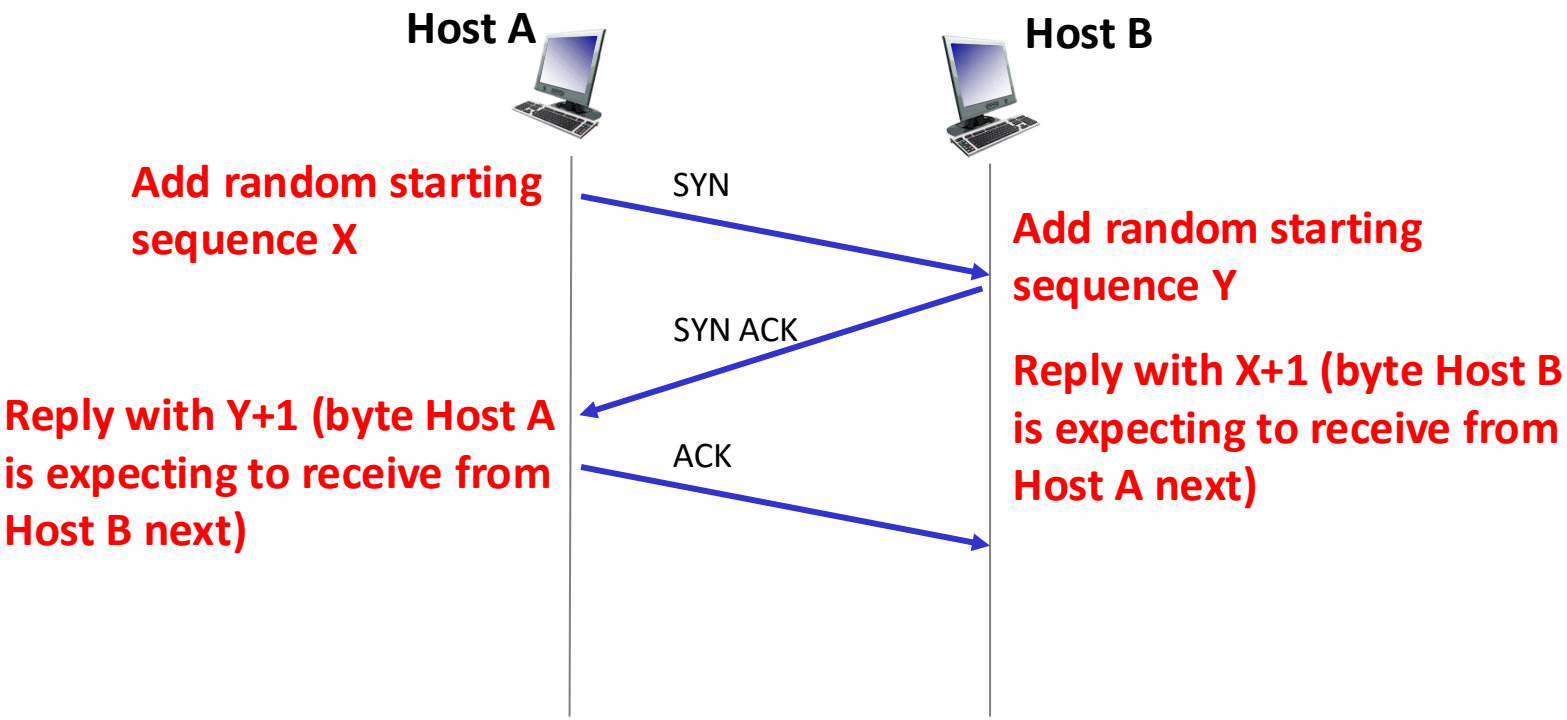
TCP handles the three big things that can go wrong

Issue	Solution
Data could be corrupted (bit flips)	Use checksum to detect errors • Drop packet if errors detected • Dropped packet now the same as a lost packet
Packets could get lost	Receiver acknowledges receipt of bytes (not packet number!) Sender starts a timer when sending bytes If no ACK received for bytes, resend if timer expires
Packets could arrive out of order	Receiver allocates a buffer • Fills bytes as they arrive • May create gaps if packets arrive out of order • TCP will not send bytes to Layer 7 with gaps

"Tools" we will use to solve big three issues:

- **Checksum**
- **ACK bytes received**
- **Timer**
- **Receive buffer**

TCP uses a three-way handshake to establish a connection between hosts



Key point:
The handshake exchanges starting sequence numbers and establishes a TCP connection

Why start at random sequence numbers? Why not always start at 0?

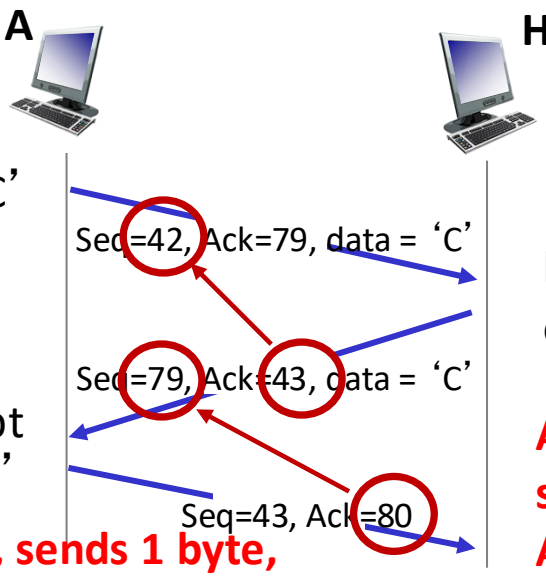
Might be packets floating around from prior connections

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

While exchanging data Segments include Sequence and Acknowledgement numbers

Key points:
Ack number starts one greater than other side's seq number

When bytes are received Ack number is the next byte the receiver is expecting



A starts a Seq 41, sends 1 byte, so this is byte 42
ACK from B to A is one more byte than Seq from A to B
A now knows B got Seq 42 and is waiting for Seq 43

B starts at 78, sends 1 byte, so this is byte 79
ACK from A to B is one more byte than Seq from B to A

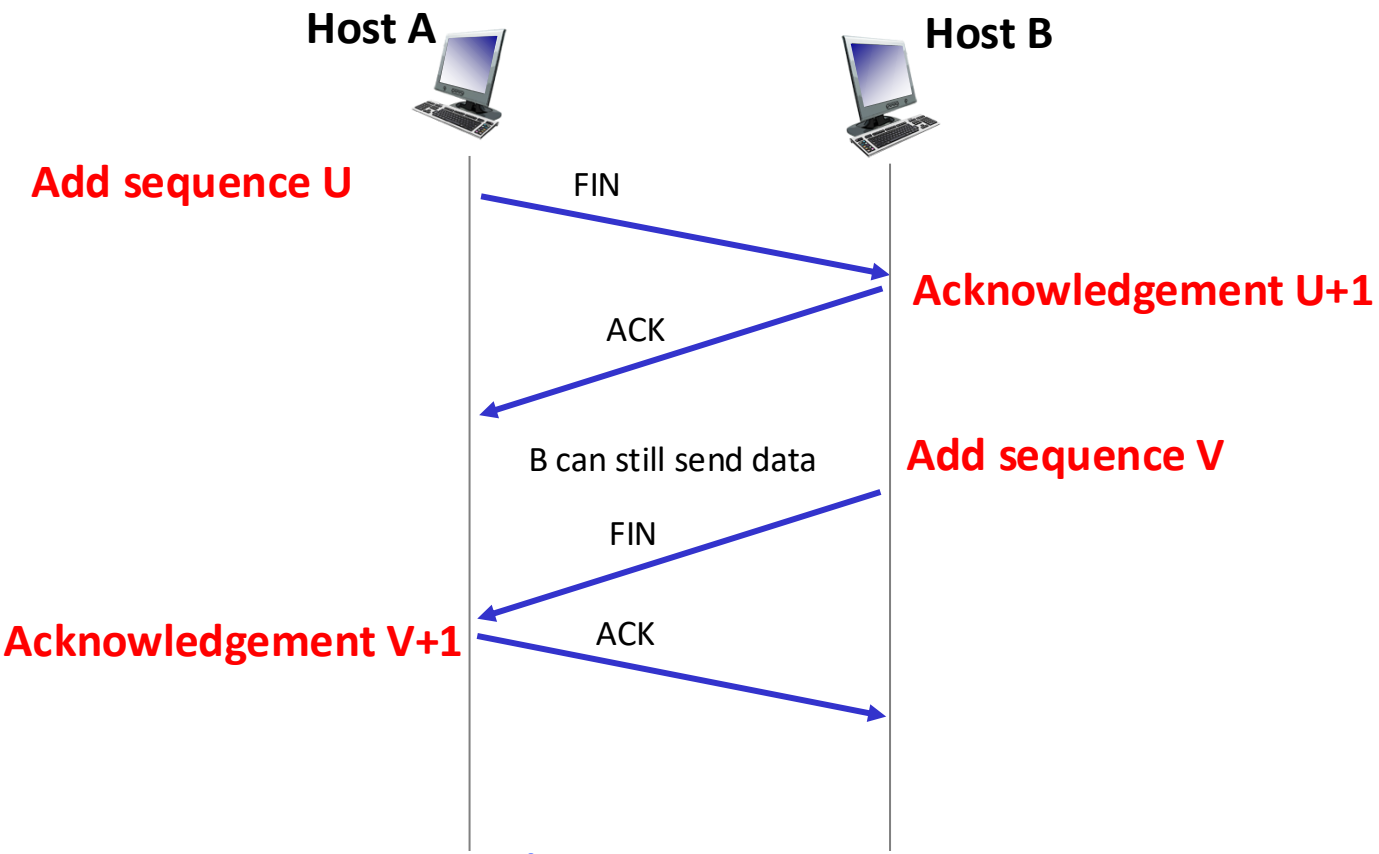
B now knows A got Seq 79 and is waiting for Seq 80

simple telnet scenario

Recall, telnet sends one byte each time a key is pressed
The telnet receiver echoes the key back to the sender
Assume A starts with Seq 41 and B starts with Seq 78

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

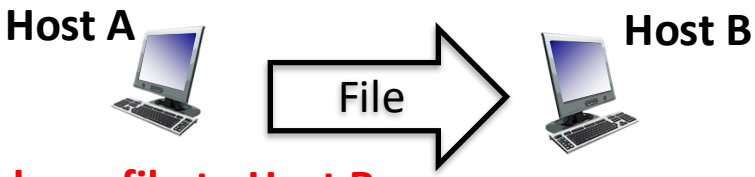
Closing a connection takes four segments, a FIN and ACK in each direction



Key point:
Both sides close the connection by sending a FIN

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP views data as a stream of bytes

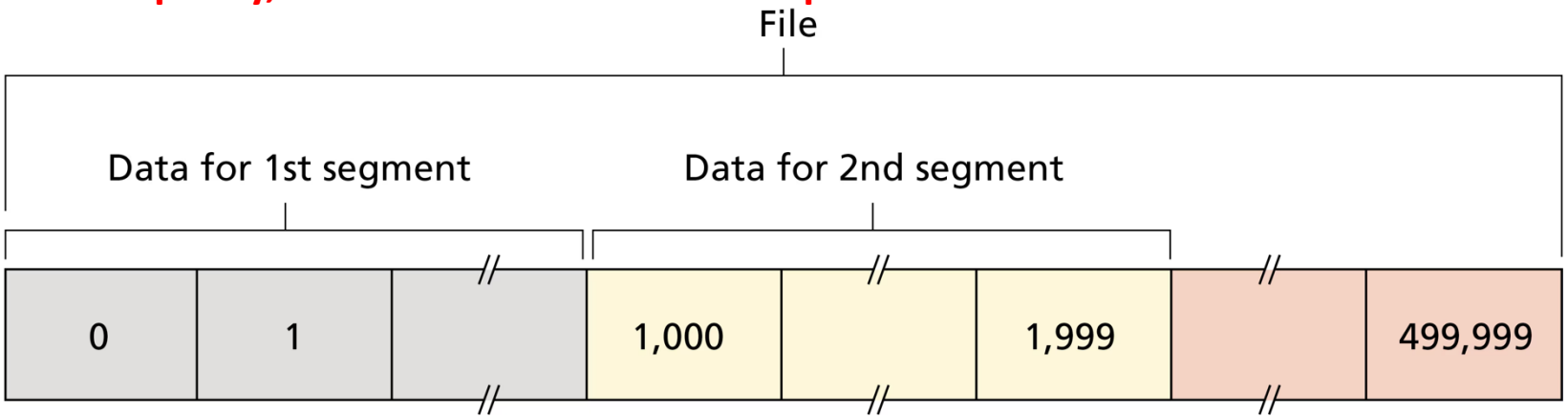


Host A wants to send a large file to Host B

- **File contains 500,000 bytes**
- **MSS is 1,000 bytes (to make math easy for me!)**

Host A creates 500 Segments each of 1,000 bytes for file

For simplicity, assume Host A starts at Seq 0



Segment 0 has 1000 bytes
Numbered 0 to 999

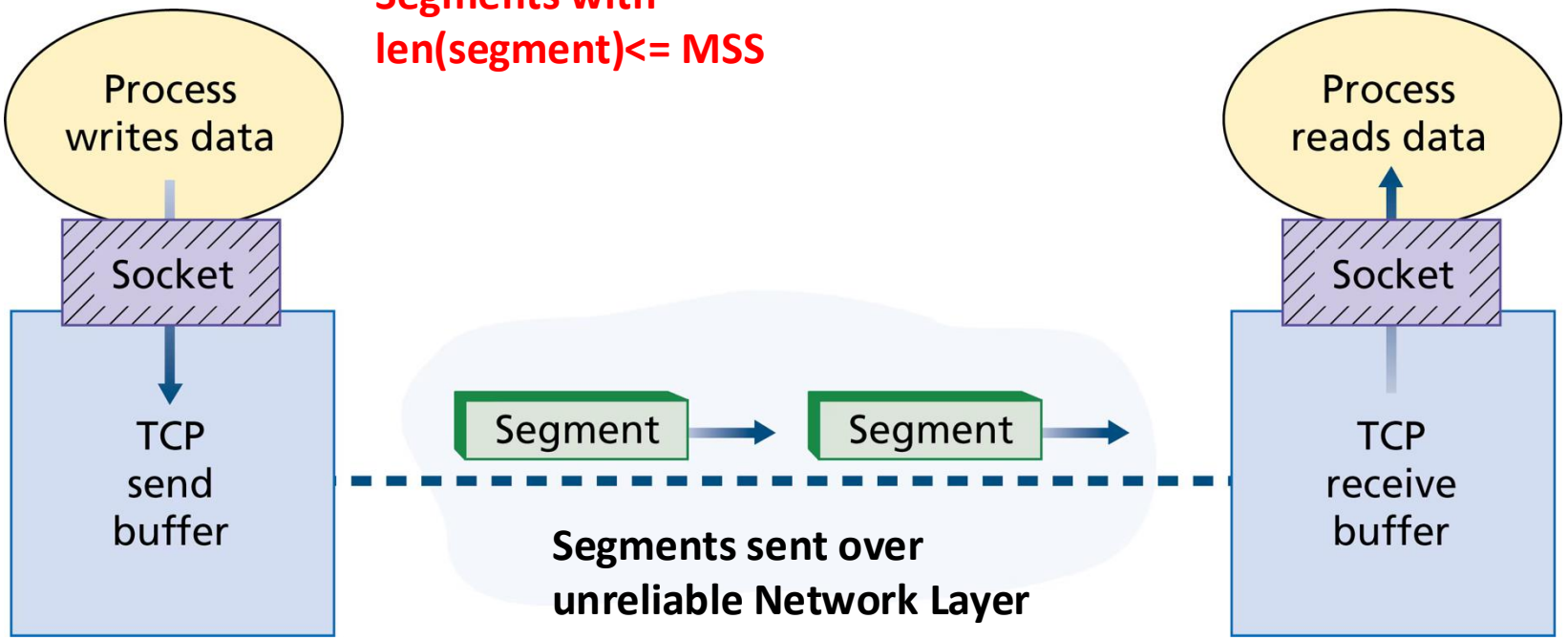
Segment 1 has 1000 bytes
Numbered 1,000 to 1,999

Segment 499 has 1000 bytes
Numbered 499,000 to 499,999

After Segment 0 arrives, receiver ACKs back 1000
letting sender know it is expecting byte 1000 next

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Sender and receiver each allocate a send and a receiver buffer, then send Segments



TCP: **Sender** **Receiver**

- Creates a stream of bytes
- Does not identify boundaries between messages (so if multiple Segments arrive, their bytes are concatenated until they are read from the socket, *unlike UDP recvfrom*)
- Will not pass bytes to Layer 7 if there are gaps

Receiver puts bytes in order in a receive buffer

Creates gaps if packets missing or out of order

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP Sender (simplified)

Event: data received from application

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- Start timer if not already running
  - Think of timer as for oldest unACKed segment
  - Expiration interval: **`TimeOutInterval`**

Event: timeout

- Retransmit segment that caused timeout
- Restart timer

Event: ACK received

- If ACK acknowledges previously unACKed segments
  - Update what is known to be ACKed
  - Start timer if there are still unACKed segments

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

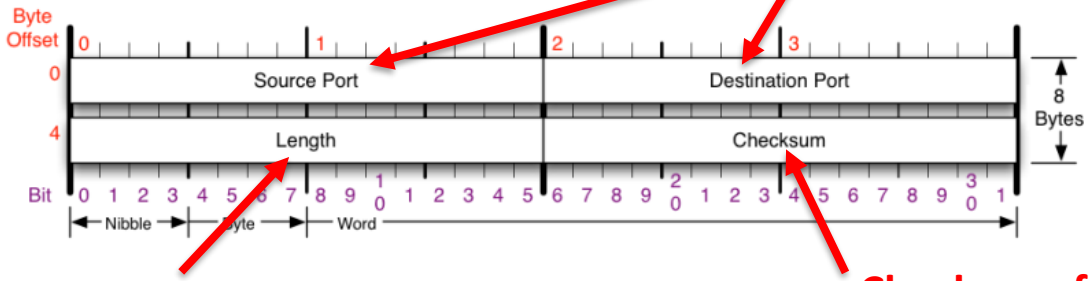
Agenda



1. TCP connections
2. TCP header bytes
3. TCP code
4. Handling multiple clients
5. Supplemental material

UDP and TCP Transport Layer headers share some commonality

UDP header



sport and dport for demuxing to process

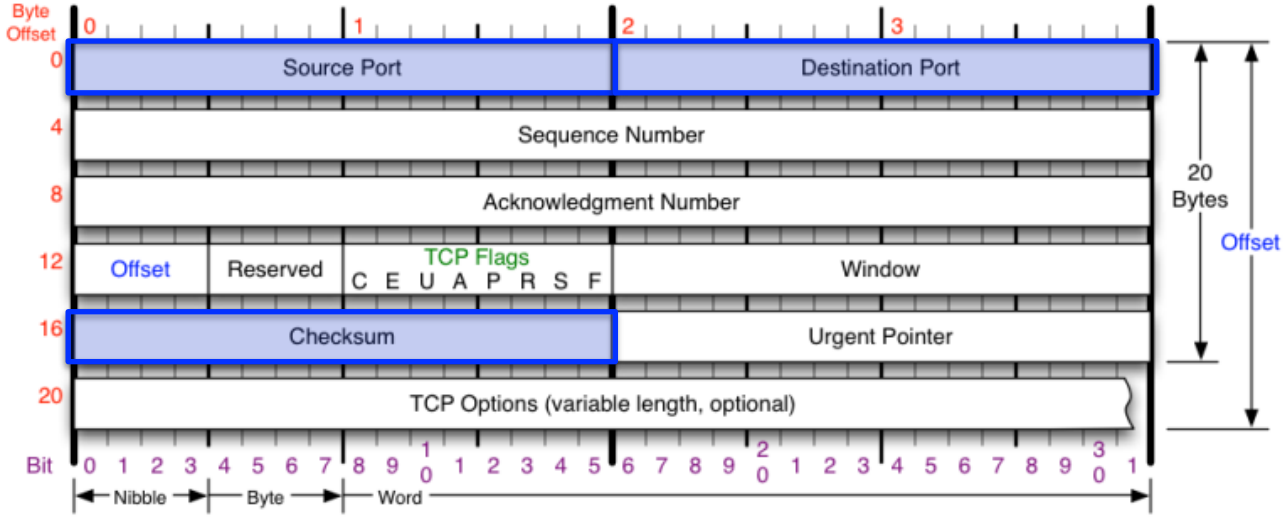
UDP header is always 8 bytes long

UDP does not deal with lost, out of order, or packets sent too fast

Header length + Layer 7 message length

Checksum for detecting bit flips

TCP header



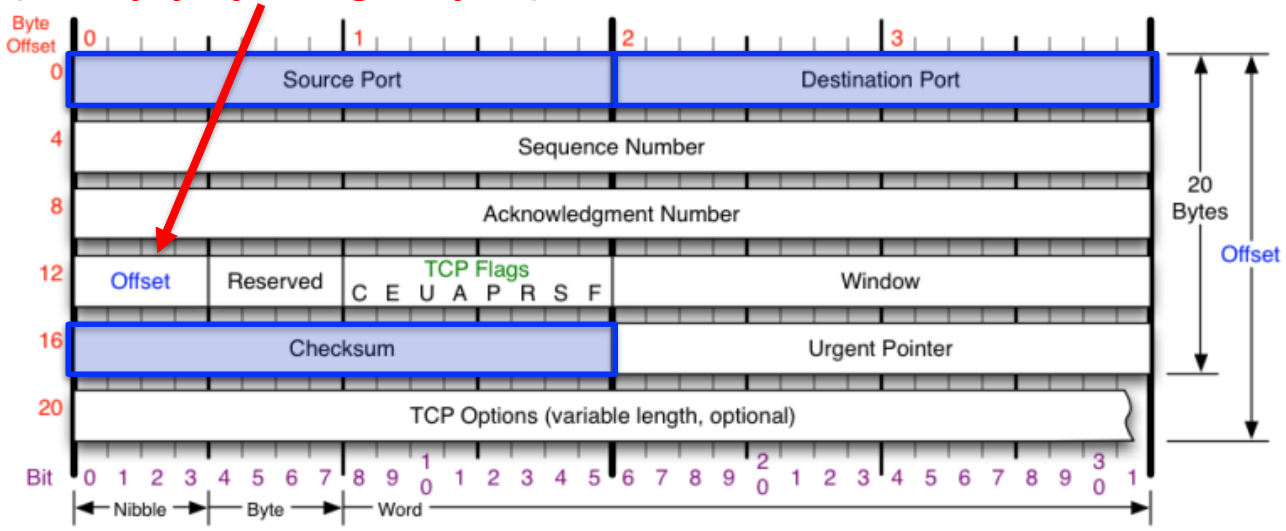
TCP header may contain optional components

Header is normally 20 bytes long, but that is not guaranteed (due to optional part)

Has sport, dport, and checksum like UDP

TCP has an offset to calculate the header length (due to optional components)

Header length in 32-bit words (multiply by 4 to get bytes)



Question: what is the TCP packet's payload?

Application Layer (Layer 7) message

Does TCP add data to the packet other than these header fields?

No

How do we know where the Layer 7 message begins?

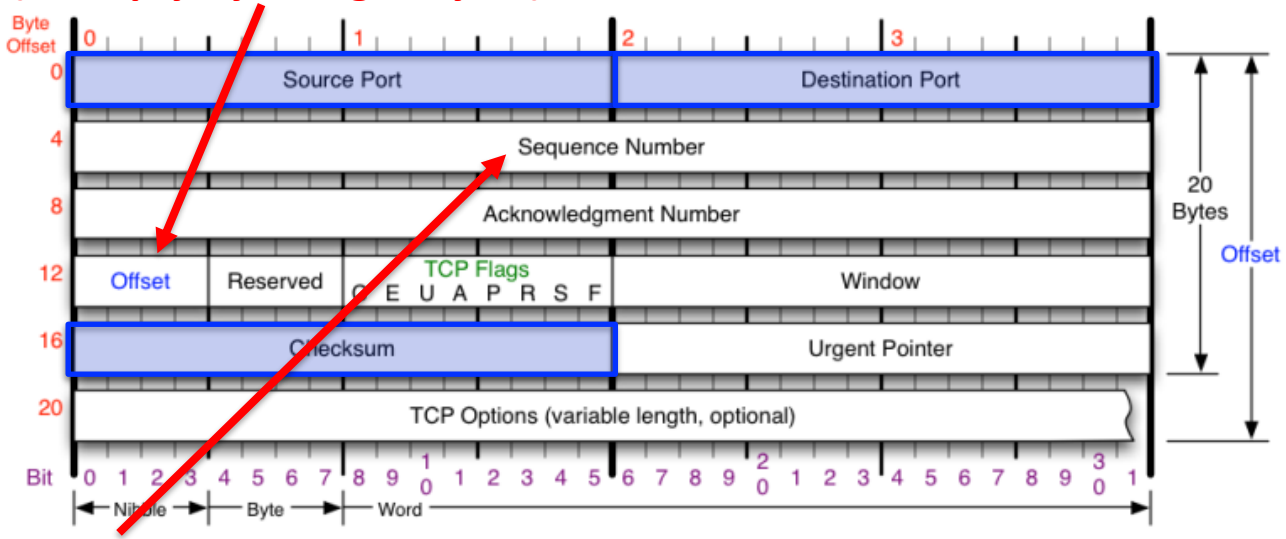
Offset *4 bytes from the start of the TCP header

With options, TCP header length varies between 20 and 60 bytes (so we need offset)

Unlike UDP, where the payload always begins 8 bytes after the header start

TCP adds sequence number that UDP lacks

Header length in 32-bit words (multiply by 4 to get bytes)



Sender and receiver each pick random starting seq numbers

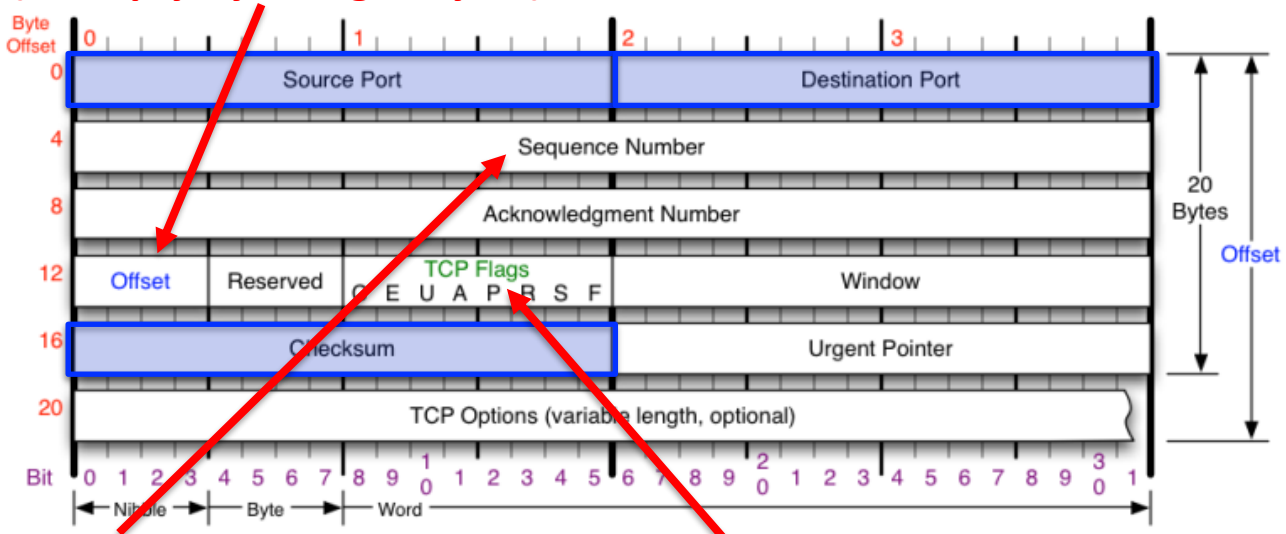
Send starting seq number to each other in 3-way handshake (SYN, SYN ACK, ACK)

Increment seq by number of bytes sent (not number of Segments)

Flags identify type of segment

Header length in 32-bit words (multiply by 4 to get bytes)

TCP header



Sender and receiver each pick random starting seq numbers

Send starting seq number to each other in 3-way handshake (SYN, SYN ACK, ACK)

Increment seq by number of bytes sent (not number of Segments)

Flags:

C: congestion

E: ECN

U: urgent

A: ack

P: push

R: reset

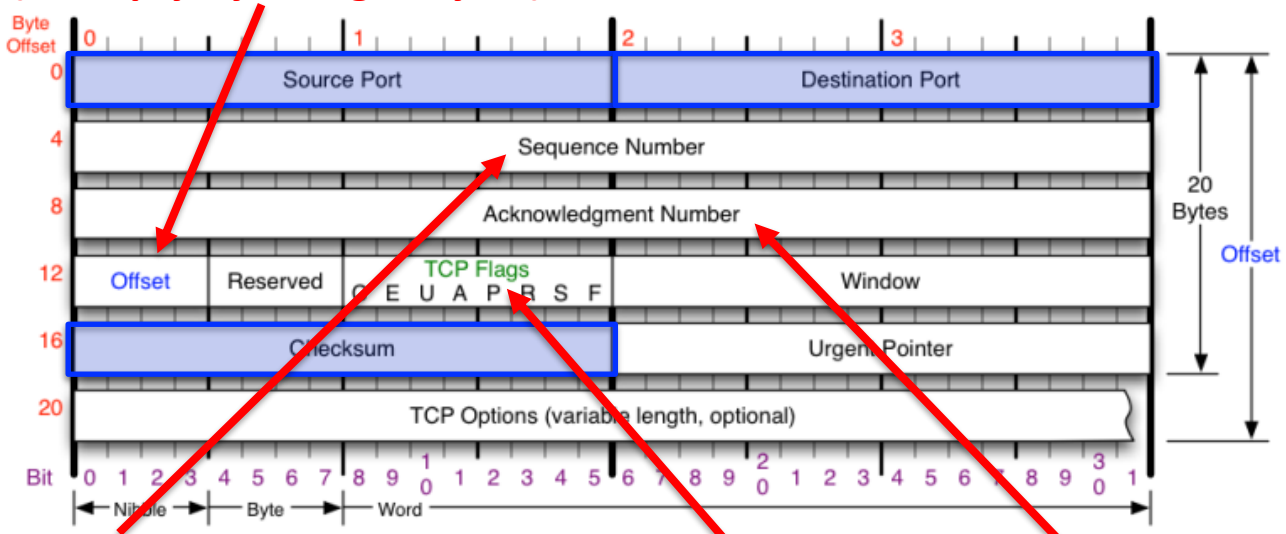
S: syn

F: fin

Acknowledgement number tracks bytes received from starting number

Header length in 32-bit words (multiply by 4 to get bytes)

TCP header



Sender and receiver each pick random starting seq numbers

Send starting seq number to each other in 3-way handshake (SYN, SYN ACK, ACK)

Increment seq by number of bytes sent (not number of Segments)

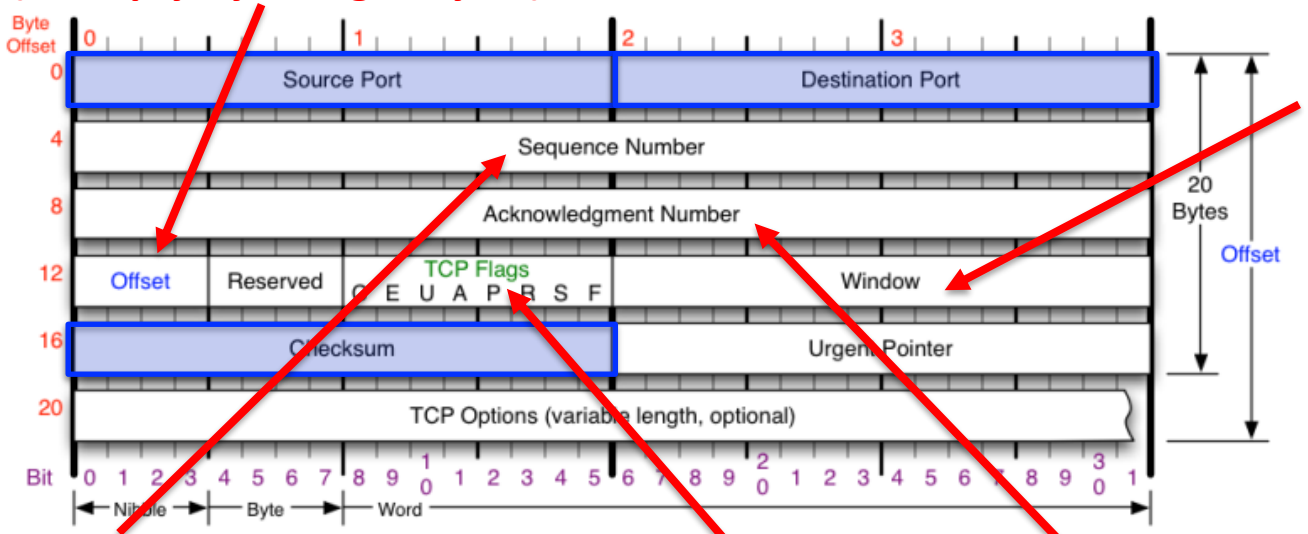
Flags:
C: congestion
E: ECN
U: urgent
A: ack
P: push
R: reset
S: syn
F: fin

Sender and receiver each track other side's starting seq + 1 + bytes received in Acknowledgement number field

Used by Go-Back-N or Selective Repeat

TCP keeps senders and receivers in sync with receive window

Header length in 32-bit words (multiply by 4 to get bytes)



Tell sender how many bytes are available in the receiver's buffer to prevent sender from going too fast

This is "flow control"
TCP also has network congestion control (we won't cover today)

Sender and receiver each pick random starting seq numbers

Send starting seq number to each other in 3-way handshake (SYN, SYN ACK, ACK)

Increment seq by number of bytes sent (not number of Segments)

Flags:
C: congestion
E: ECN
U: urgent
A: ack
P: push
R: reset
S: syn
F: fin

Sender and receiver each track other side's starting seq + 1 + bytes received in Acknowledgement number field

Used by Go-Back-N or Selective Repeat

TCP 3-way handshake states and socket calls in detail

No notion of connect in UDP

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

CLOSED

```
clientSocket.connect((serverName,serverPort))
```

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

SYNSENT

ESTAB

Server state

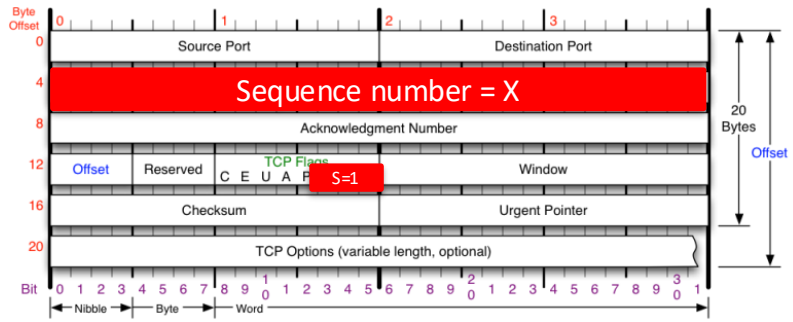
```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen()
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD

ESTAB

SYN



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP 3-way handshake states and socket calls in detail

Each connection gets its own socket

Identified by src/dst IP and src/dst Port

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

CLOSED

```
clientSocket.connect((serverName,serverPort))
```

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ESTAB

Server state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen()
connectionSocket, addr = serverSocket.accept()
```

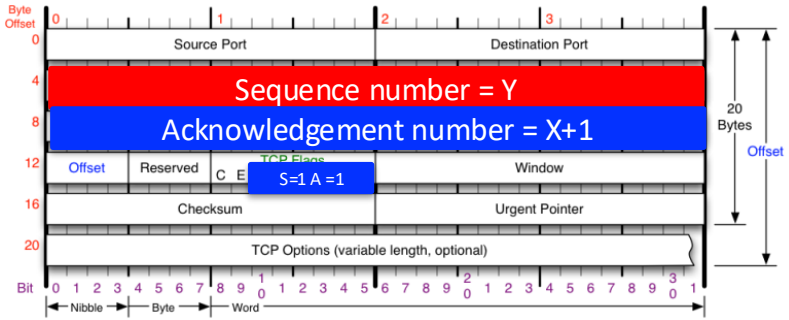
LISTEN

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYN RCVD

ESTAB

SYN ACK



TCP 3-way handshake states and socket calls in detail

Server state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen()
connectionSocket, addr = serverSocket.accept()
```

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

CLOSED

```
clientSocket.connect((serverName,serverPort))
```





LISTEN

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYNSENT

SYN RCVD

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

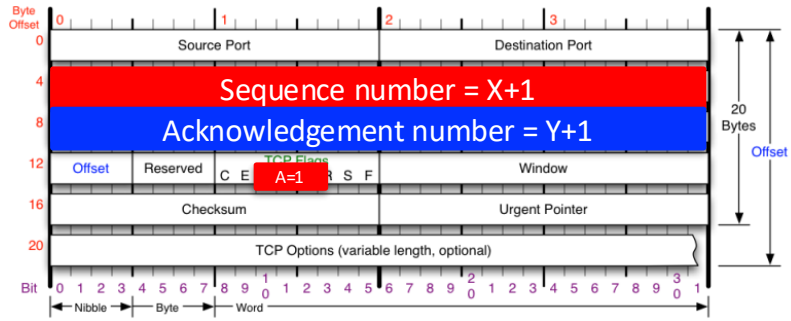
ESTAB

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

ESTAB

ACK



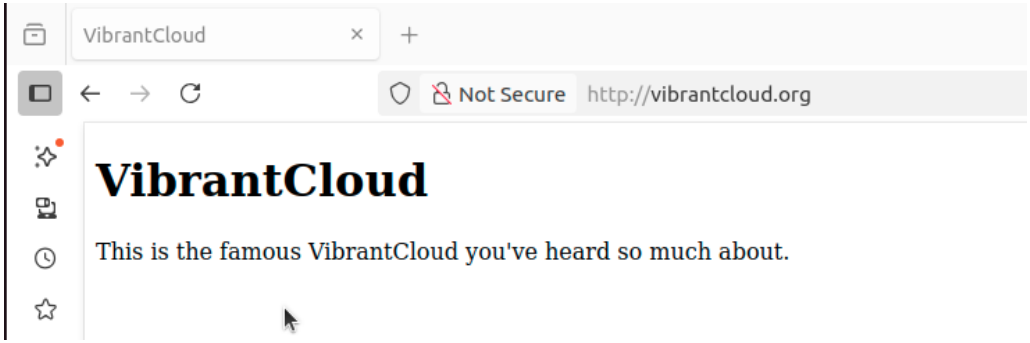
Future packets:

- **Sequence number increases by # bytes sent**
- **Acknowledgement number by # bytes received +1**

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Example: get a web page

Watch web browser download web page
Layer 4 = TCP
Layer 7 = HTTP
Start wireshark
Use vibrantcloud.org in browser (my server)



Browser renders the html as a page

Browser sends

GET /index.html HTTP/1.1 \r\n
Host: vibrantcloud.org \r\n
\r\n
Gets index.html file

index.html

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
  <title>VibrantCloud</title>
</head>

<body>
  <h1>VibrantCloud</h1>
  <p>This is the famous VibrantCloud you've heard so
much about.</p>
</body>
</html>
```

Getting web page using HTTP over TCP



Find vibrantcloud.org IP address

```
$ nslookup -type=A vibrantcloud.org
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
Name:   vibrantcloud.org
Address: 173.201.179.249
```

Getting web page using HTTP over TCP



Filter on ip address of vibrantcloud

How to do get it?

nslookup –type=A vibrantcloud.org

gives 173.201.179.249

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Le
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=19
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=42228402
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=1942573
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=42228408
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=1942578
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=42228413

```
▸ Frame 5: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_a8:d0:96 (00:0c:29:a8:d0:96), Dst: VMware_f5:9e:21 (00:50:56:f5:9e:21)
▸ Internet Protocol Version 4, Src: 192.168.159.129, Dst: 173.201.179.249
▾ Transmission Control Protocol, Src Port: 43786, Dst Port: 80, Seq: 194256941, Len: 0
    Source Port: 43786
    Destination Port: 80
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 194256941
    [Next Sequence Number: 194256942]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1010 .... = Header Length: 40 bytes (10)
  ▾ Flags: 0x002 (SYN)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...0 .... = Acknowledgment: Not set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    ▸ .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ··········S·]
    Window: 64240
    [Calculated window size: 64240]
    Checksum: 0xc21b [unverified]
    [Checksum Status: Unverified]
```

Step 1: TCP SYN



SYN

192.168.159.129
Port 43786

173.201.179.249
Port 80

VM to vibrantcloud

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Le
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=19
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=42228402
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=1942573
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=42228408
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=1942578
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=42228413

```
▸ Frame 5: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_a8:d0:96 (00:0c:29:a8:d0:96), Dst: VMware_f5:9e:21 (00:50:56:f5:9e:21)
▸ Internet Protocol Version 4, Src: 192.168.159.129, Dst: 173.201.179.249
▾ Transmission Control Protocol, Src Port: 43786, Dst Port: 80, Seq: 194256941, Len: 0
    Source Port: 43786
    Destination Port: 80
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 194256941
    [Next Sequence Number: 194256942]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1010 .... = Header Length: 40 bytes (10)
  ▾ Flags: 0x002 (SYN)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...0 .... = Acknowledgment: Not set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    ▸ .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ··········S·]
    Window: 64240
    [Calculated window size: 64240]
    Checksum: 0xc21b [unverified]
    [Checksum Status: Unverified]
```

Layer 4
sport and dport

Random seq #
194256941

Syn flag set

Step 1: TCP SYN



SYN

192.168.159.129 Port 43786

173.201.179.249 Port 80

MSS 1460

- **1500 Ethernet max**
- **- 20-byte TCP header**
- **-20-byte IP header**
- **Equals 1460 MSS**

Passed in optional header component

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Le
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=19
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=42228402
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=1942573
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=42228408
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=1942578
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=42228413

```
▸ Frame 5: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_a8:d0:96 (00:0c:29:a8:d0:96), Dst: VMware_f5:9e:21 (00:50:56:f5:9e:21)
▸ Internet Protocol Version 4, Src: 192.168.159.129, Dst: 173.201.179.249
▾ Transmission Control Protocol, Src Port: 43786, Dst Port: 80, Seq: 194256941, Len: 0
    Source Port: 43786
    Destination Port: 80
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 194256941
    [Next Sequence Number: 194256942]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1010 .... = Header Length: 40 bytes (10)
  ▾ Flags: 0x002 (SYN)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...0 .... = Acknowledgment: Not set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    ▸ .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ··········S·]
    Window: 64240
    [Calculated window size: 64240]
    Checksum: 0xc21b [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▾ Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
    ▸ TCP Option - Maximum segment size: 1460 bytes
    ▸ TCP Option - SACK permitted
```

Step 2: SYN ACK



SYN ACK

192.168.159.129 Port 43786

173.201.179.249 Port 80

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len=
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=194
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=422284028
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=194257385
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=422284083
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=194257828
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=422284136

```
▸ Frame 6: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_f5:9e:21 (00:50:56:f5:9e:21), Dst: VMware_a8:d0:96 (00:0c:29:a8:d0:96)
▸ Internet Protocol Version 4, Src: 173.201.179.249, Dst: 192.168.159.129
▾ Transmission Control Protocol, Src Port: 80, Dst Port: 43786, Seq: 4222840285, Ack: 194256942, Len: 0
    Source Port: 80
    Destination Port: 43786
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 4222840285
    [Next Sequence Number: 4222840286]
    Acknowledgment Number: 194256942
    0110 .... = Header Length: 24 bytes (6)
  ▾ Flags: 0x012 (SYN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    ▸ .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····A··S·]
    Window: 64240
    [Calculated window size: 64240]
    Checksum: 0x9a8a [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
```

Layer 4 sport and dport

Random seq # 4222840285

Ack # one bigger than random 194256941 chosen by VM

Syn and Ack flags set

Step 3: ACK



ACK

192.168.159.129
Port 43786

173.201.179.249
Port 80

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len=
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=1942
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=4222840286
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=194257385
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=4222840839
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=194257828
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=4222841366

```
▸ Frame 7: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_a8:d0:96 (00:0c:29:a8:d0:96), Dst: VMware_f5:9e:21 (00:50:56:f5:9e:21)
▸ Internet Protocol Version 4, Src: 192.168.159.129, Dst: 173.201.179.249
▾ Transmission Control Protocol, Src Port: 43786, Dst Port: 80, Seq: 194256942, Ack: 4222840286, Len: 0
    Source Port: 43786
    Destination Port: 80
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 194256942
    [Next Sequence Number: 194256942]
    Acknowledgment Number: 4222840286
    0101 .... = Header Length: 20 bytes (5)
  ▾ Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····A····]
    Window: 64240
    [Calculated window size: 64240]
    [Window size scaling factor: -2 (no window scaling used)]
    Checksum: 0xc207 [unverified]
    [Checksum Status: Unverified]
```

Seq # one bigger than randomly chosen 194256941

Ack # one bigger than random 4222840285 chosen by server

Ack flag set

TCP connection is now established (SYN, SYN ACK, ACK complete)

Step 4: HTTP GET



HTTP GET

192.168.159.129 Port 43786

173.201.179.249 Port 80

VM asks for web page using HTTP Get

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=194
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=422284028
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=19425738
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=422284083
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=19425782
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=422284136

```
▸ Frame 8: 497 bytes on wire (3976 bits), 497 bytes captured (3976 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_a8:d0:96 (00:0c:29:a8:d0:96), Dst: VMware_f5:9e:21 (00:50:56:f5:9e:21)
▸ Internet Protocol Version 4, Src: 192.168.159.129, Dst: 173.201.179.249
▾ Transmission Control Protocol, Src Port: 43786, Dst Port: 80, Seq: 194256942, Ack: 4222840286, Len: 443
    Source Port: 43786
    Destination Port: 80
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 443]
    Sequence Number: 194256942
    [Next Sequence Number: 194257385]
    Acknowledgment Number: 4222840286
    0101 .... = Header Length: 20 bytes (5)
  ▾ Flags: 0x018 (PSH, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 1... = Push: Set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····AP···]
    Window: 64240
    [Calculated window size: 64240]
    [Window size scaling factor: -2 (no window scaling used)]
    Checksum: 0xc3c2 [unverified]
    [Checksum Status: Unverified]
```

Seq # one larger than originally chosen

Step 4: HTTP GET



HTTP GET

192.168.159.129 Port 43786

173.201.179.249 Port 80

VM asks for web page using HTTP Get

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len=0
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=19425
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=4222840286
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	[illegible]	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=194257385
10	2025-09-30 19:1…	[illegible]	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=4222840839
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=194257828
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=4222841366

```
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 1... = Push: Set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
    .... .... ...0 = Fin: Not set
    [TCP Flags: ·····AP···]
  Window: 64240
  [Calculated window size: 64240]
  [Window size scaling factor: -2 (no window scaling used)]
  Checksum: 0xc3c2 [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  [Timestamps]
  [SEQ/ACK analysis]
  TCP payload (443 bytes)
Hypertext Transfer Protocol
  GET / HTTP/1.1\r\n
  Host: vibrantcloud.org\r\n
  User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:143.0) Gecko/20100101 Firefox/143.0\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
  Accept-Language: en-US,en;q=0.5\r\n
  Accept-Encoding: gzip, deflate\r\n
  Connection: keep-alive\r\n
  Upgrade-Insecure-Requests: 1\r\n
  If-Modified-Since: Fri, 16 Apr 2021 15:17:30 GMT\r\n
  If-None-Match: "1b210a5-137-5c0187a783680-gzip"\r\n
  Priority: u=0, i\r\n
  \r\n
  [Full request URI: http://vibrantcloud.org/]
```

HTTP GET request is 443 bytes long

Step 5: ACK HTTP GET



ACK

192.168.159.129 Port 43786

173.201.179.249 Port 80

Server ACKs VM's HTTP request

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len=
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=1942
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=4222840286
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=194257385
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=4222840839
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=194257828
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=4222841366

```
▸ Frame 9: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_f5:9e:21 (00:50:56:f5:9e:21), Dst: VMware_a8:d0:96 (00:0c:29:a8:d0:96)
▸ Internet Protocol Version 4, Src: 173.201.179.249, Dst: 192.168.159.129
▾ Transmission Control Protocol, Src Port: 80, Dst Port: 43786, Seq: 4222840286, Ack: 194257385, Len: 0
    Source Port: 80
    Destination Port: 43786
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 4222840286
    [Next Sequence Number: 4222840286]
    Acknowledgment Number: 194257385
    0101 .... = Header Length: 20 bytes (5)
  ▾ Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····A····]
    Window: 64240
    [Calculated window size: 64240]
    [Window size scaling factor: -2 (no window scaling used)]
    Checksum: 0xb08c [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
```

Ack # previous + 443 bytes from HTTP request

194256942 + 443 = 194257385

Step 6: HTTP OK (reply to GET)



HTTP OK

192.168.159.129 Port 43786

173.201.179.249 Port 80

Server returns web page 553 bytes

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len=0
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=19425
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=4222840286
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=194257385
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=4222840839
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=194257828
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=4222841366

```
▸ Frame 10: 607 bytes on wire (4856 bits), 607 bytes captured (4856 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_f5:9e:21 (00:50:56:f5:9e:21), Dst: VMware_a8:d0:96 (00:0c:29:a8:d0:96)
▸ Internet Protocol Version 4, Src: 173.201.179.249, Dst: 192.168.159.129
▾ Transmission Control Protocol, Src Port: 80, Dst Port: 43786, Seq: 4222840286, Ack: 194257385, Len: 553
    Source Port: 80
    Destination Port: 43786
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 553]
    Sequence Number: 4222840286
    [Next Sequence Number: 4222840839]
    Acknowledgment Number: 194257385
    0101 .... = Header Length: 20 bytes (5)
  ▾ Flags: 0x018 (PSH, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 1... = Push: Set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····AP···]
    Window: 64240
    [Calculated window size: 64240]
    [Window size scaling factor: -2 (no window scaling used)]
    Checksum: 0x6638 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
```

No new data from VM so Ack stays at 194257385

Step 6: HTTP OK (reply to GET)



HTTP OK

192.168.159.129 Port 43786

173.201.179.249 Port 80

Server returns web page 553 bytes

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len=
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=1942
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=4222840286
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=194257385
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=4222840839
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=194257828
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=4222841366

```
ETag: "1b210a5-137-5c0187a783680-gzip"\r\n
Accept-Ranges: bytes\r\n
Vary: Accept-Encoding\r\n
Content-Encoding: gzip\r\n
Content-Length: 207\r\n
Keep-Alive: timeout=5\r\n
Content-Type: text/html\r\n
\r\n
[HTTP response 1/2]
[Time since request: 0.080321815 seconds]
[Request in frame: 8]
[Next request in frame: 12]
[Next response in frame: 14]
[Request URI: http://vibrantcloud.org/]
Content-encoded entity body (gzip): 207 bytes -> 311 bytes
File Data: 311 bytes
Line-based text data: text/html (17 lines)
<!doctype html>\n
\n
<html lang="en">\n
<head>\n
  <meta charset="utf-8">\n
\n
  <title>VibrantCloud</title>\n
  <meta name="description" content="VibrantCloud">\n
  <meta name="author" content="Anon">\n
\n
</head>\n
\n
<body>\n
```

Reply contains HTML for web page

Step 7: ACK HTTP OK (reply to GET)



ACK

192.168.159.129
Port 43786

173.201.179.249
Port 80

VM Acks web page

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len=0
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=19425
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=4222840286
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=194257385
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=4222840839
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=194257828
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=4222841366

```
▸ Frame 11: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_a8:d0:96 (00:0c:29:a8:d0:96), Dst: VMware_f5:9e:21 (00:50:56:f5:9e:21)
▸ Internet Protocol Version 4, Src: 192.168.159.129, Dst: 173.201.179.249
▾ Transmission Control Protocol, Src Port: 43786, Dst Port: 80, Seq: 194257385, Ack: 4222840839, Len: 0
    Source Port: 43786
    Destination Port: 80
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 194257385
    [Next Sequence Number: 194257385]
    Acknowledgment Number: 4222840839
    0101 .... = Header Length: 20 bytes (5)
  ▾ Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····A····]
    Window: 63687
    [Calculated window size: 63687]
    [Window size scaling factor: -2 (no window scaling used)]
    Checksum: 0xc207 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
```

Ack# = previous + 553 bytes

4222840286 + 553 = 4222840839

Step 7: ACK HTTP OK (reply to GET)



ACK

192.168.159.129
Port 43786

173.201.179.249
Port 80

[SEED Labs] *eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr == 173.201.179.249

No.	Time	Source	Destination	Protocol	Length	Info
5	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	74	43786 → 80 [SYN] Seq=194256941 Win=64240 Len=0
6	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [SYN, ACK] Seq=4222840285 Ack=19425
7	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194256942 Ack=4222840286
8	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
9	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840286 Ack=194257385
10	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	607	HTTP/1.1 200 OK (text/html)
11	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257385 Ack=4222840839
12	2025-09-30 19:1…	192.168.159.129	173.201.179.249	HTTP	497	GET / HTTP/1.1
13	2025-09-30 19:1…	173.201.179.249	192.168.159.129	TCP	60	80 → 43786 [ACK] Seq=4222840839 Ack=194257828
14	2025-09-30 19:1…	173.201.179.249	192.168.159.129	HTTP	581	HTTP/1.1 200 OK (text/html)
15	2025-09-30 19:1…	192.168.159.129	173.201.179.249	TCP	54	43786 → 80 [ACK] Seq=194257828 Ack=4222841366

```
▸ Frame 11: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface eth0, id 0
▸ Ethernet II, Src: VMware_a8:d0:96 (00:0c:29:a8:d0:96), Dst: VMware_f5:9e:21 (00:50:56:f5:9e:21)
▸ Internet Protocol Version 4, Src: 192.168.159.129, Dst: 173.201.179.249
▾ Transmission Control Protocol, Src Port: 43786, Dst Port: 80, Seq: 194257385, Ack: 4222840839, Len: 0
    Source Port: 43786
    Destination Port: 80
    [Stream index: 2]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 194257385
    [Next Sequence Number: 194257385]
    Acknowledgment Number: 4222840839
    0101 .... = Header Length: 20 bytes (5)
  ▾ Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····A····]
    Window: 63687
    [Calculated window size: 63687]
    [Window size scaling factor: -2 (no window scaling used)]
    Checksum: 0xc207 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
```

I hit refresh on my browser to fetch the page again

Notice the pattern:

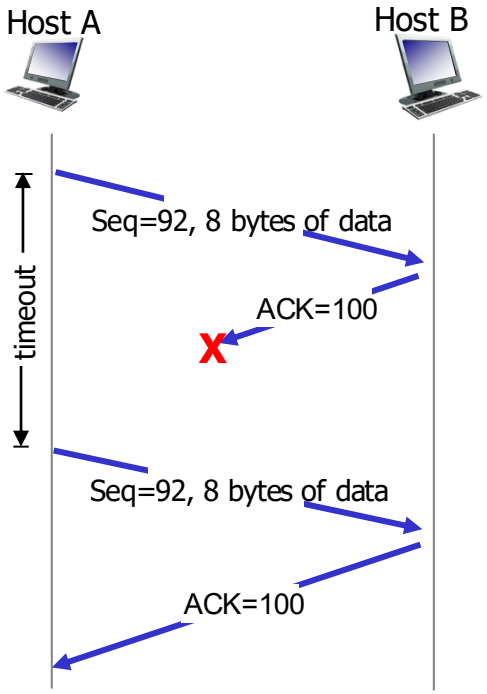
VM->Server: HTTP GET

Server->VM: ACK (GET)

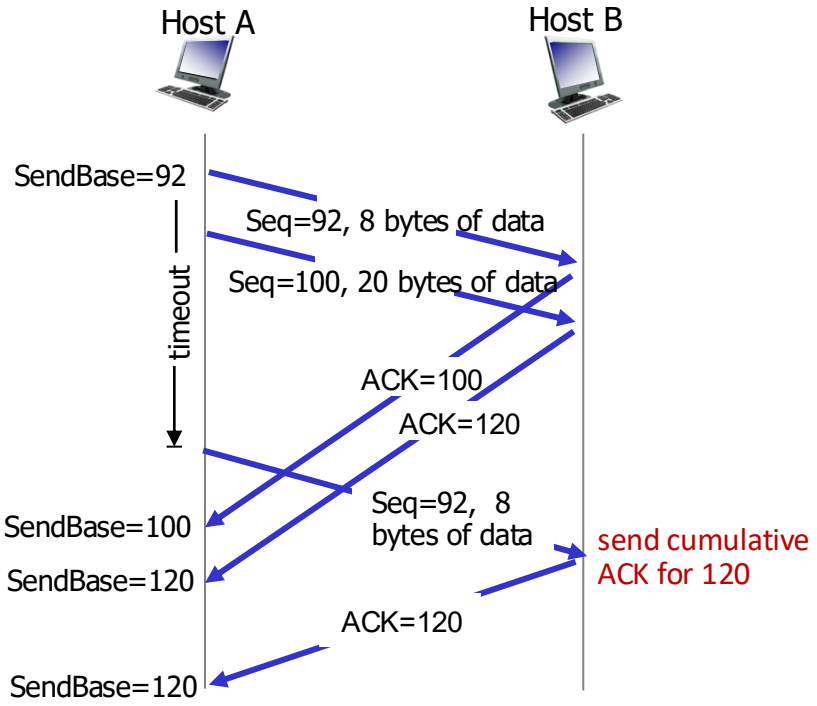
Server->VM: HTTP OK (reply to GET)

VM->Server: ACK (HTTP OK)

TCP: retransmission scenarios



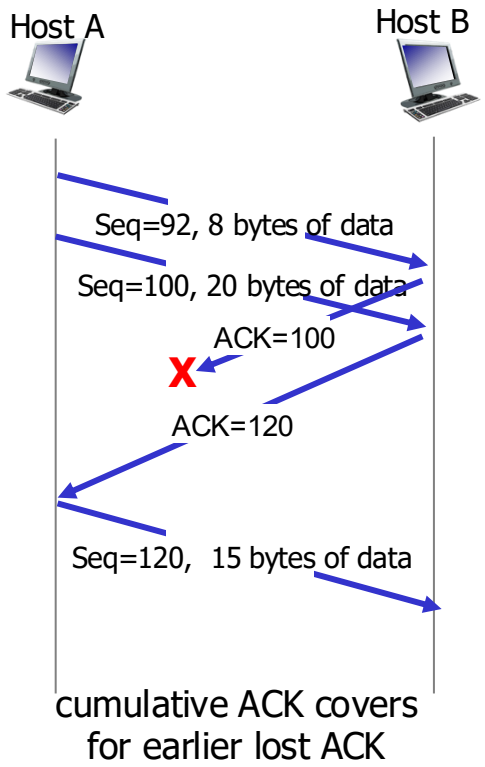
lost ACK scenario



premature timeout

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP: retransmission scenarios



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP fast retransmit

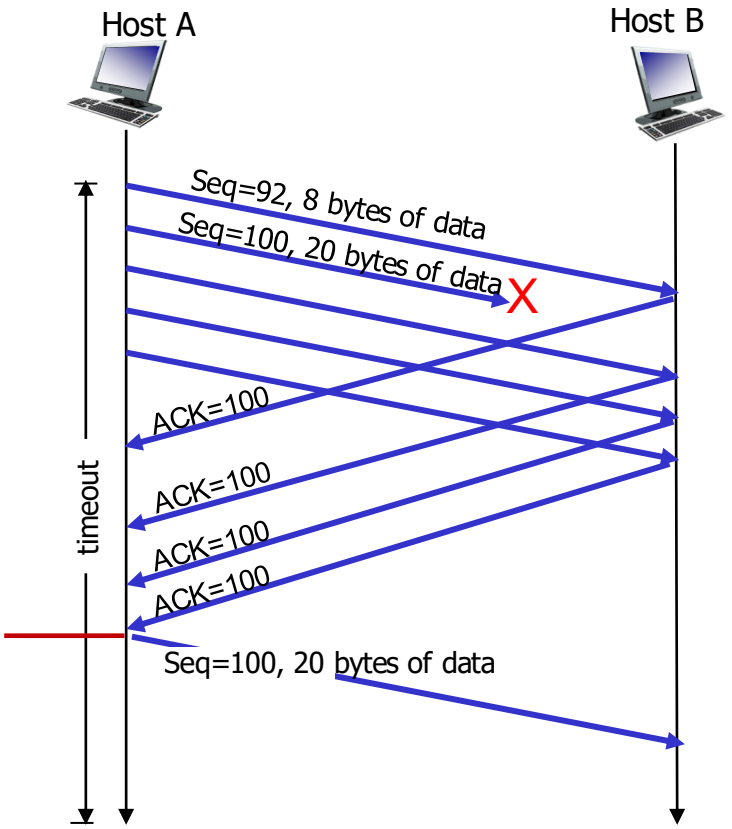
TCP fast retransmit

If sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- Likely that unACKed segment lost, so don't wait for timeout



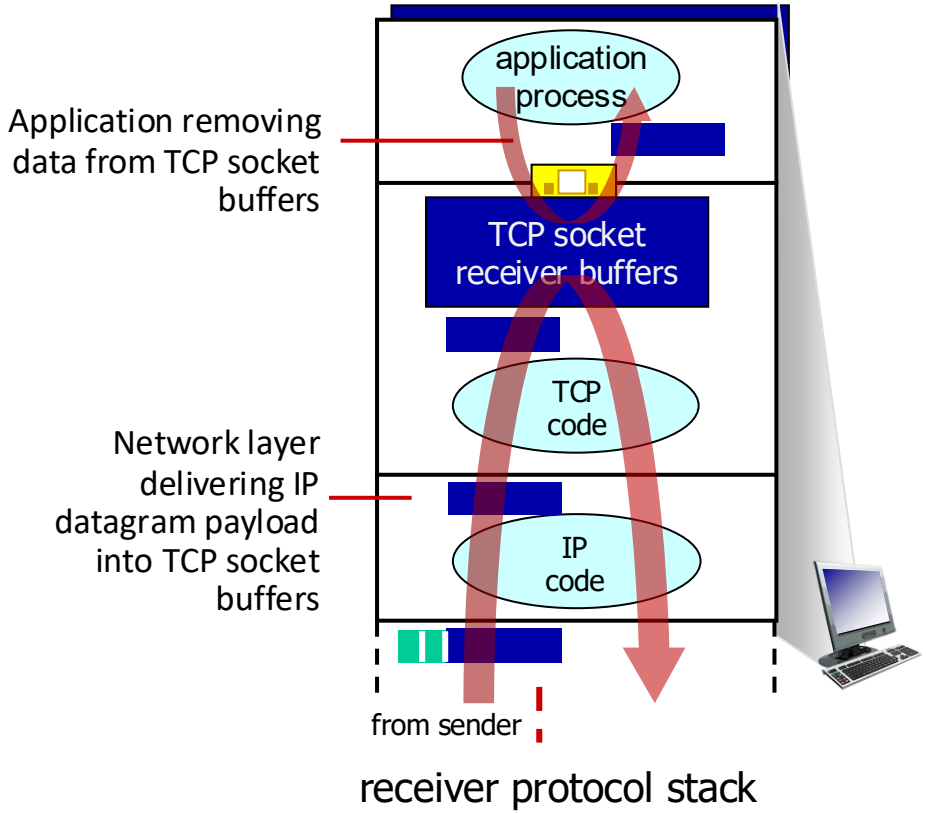
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

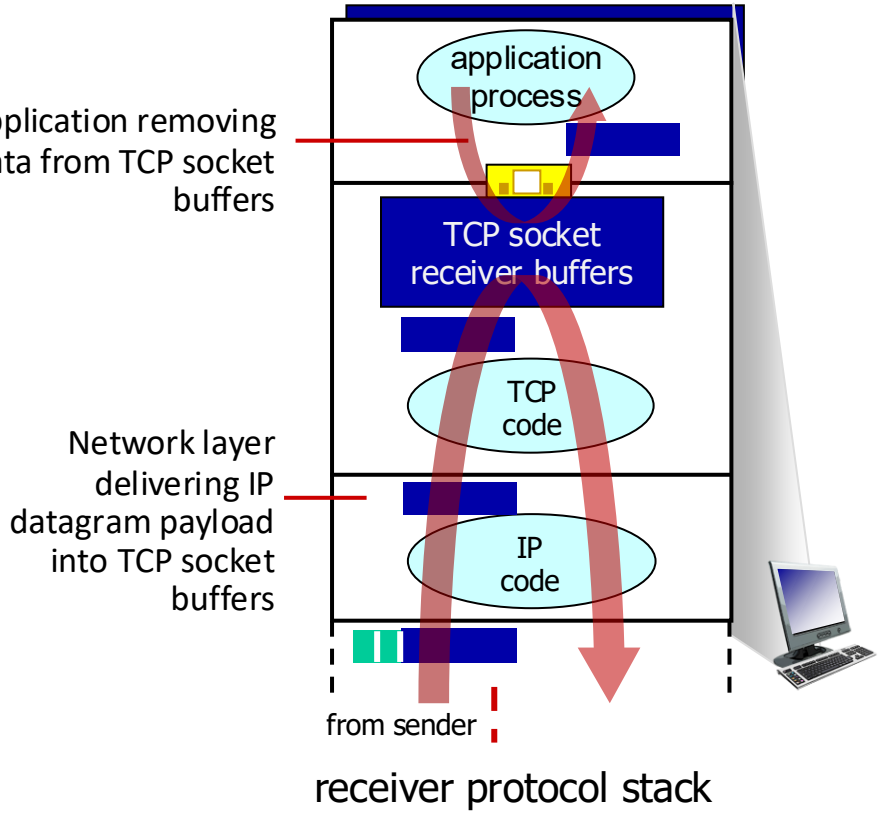


Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

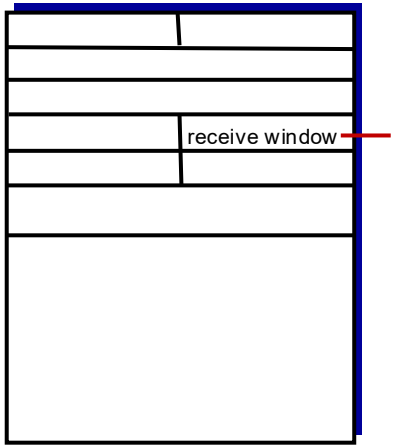




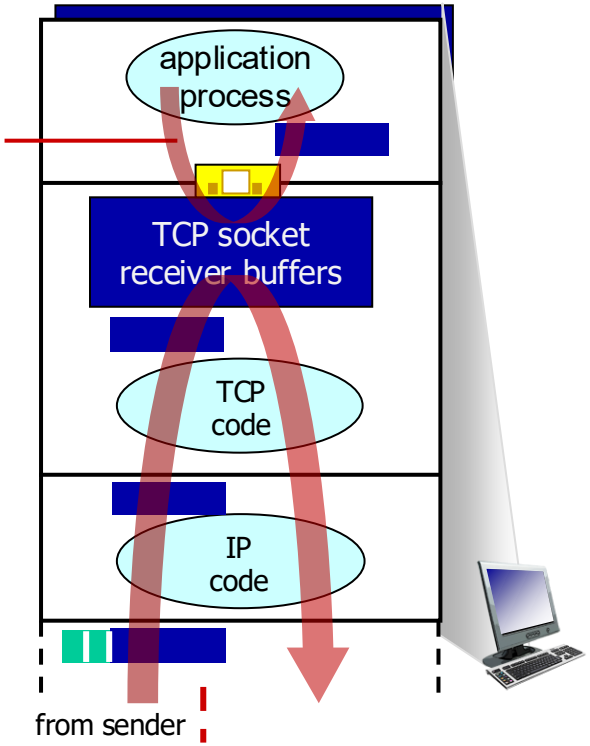
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control: # bytes receiver willing to accept



Application removing data from TCP socket buffers

receiver protocol stack

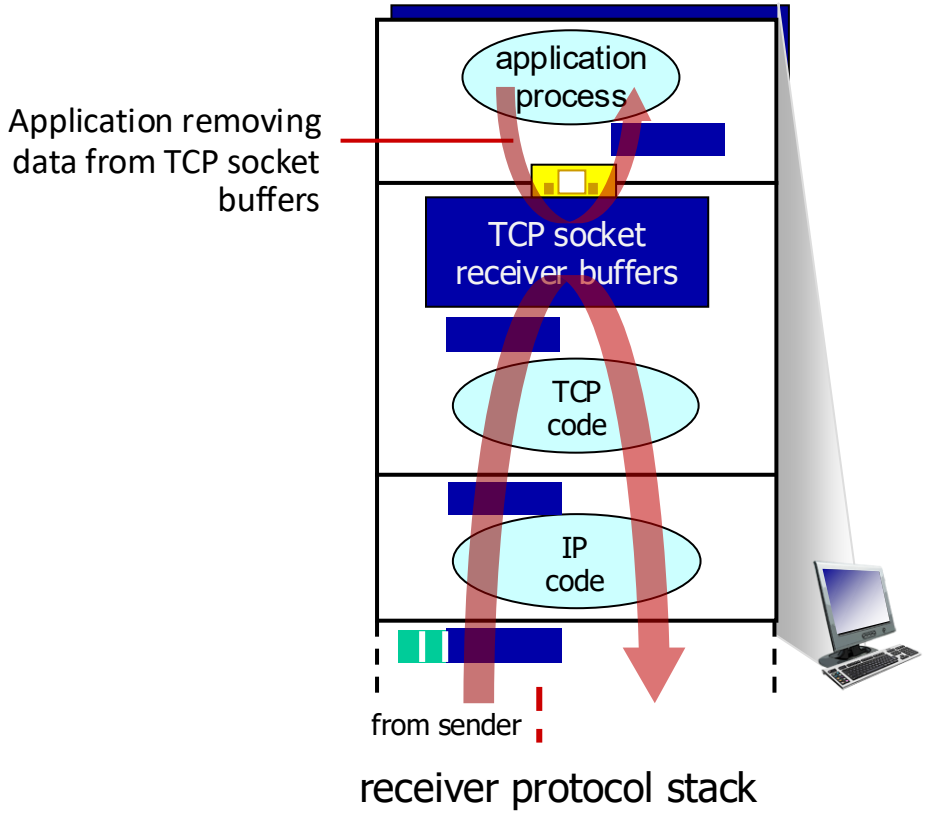
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

flow control

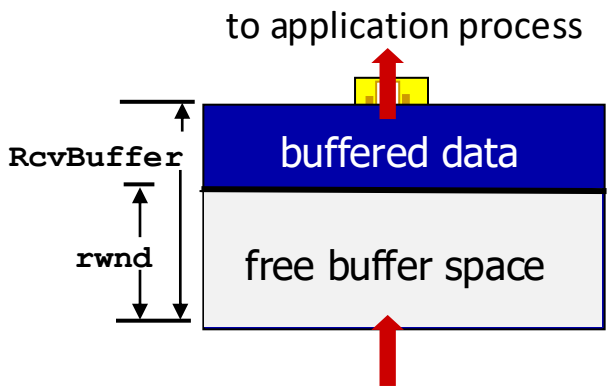
Receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 128 KB but can increase to MB)
  - Many operating systems auto-adjust **`RcvBuffer`**
- Sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- Guarantees receive buffer will not overflow



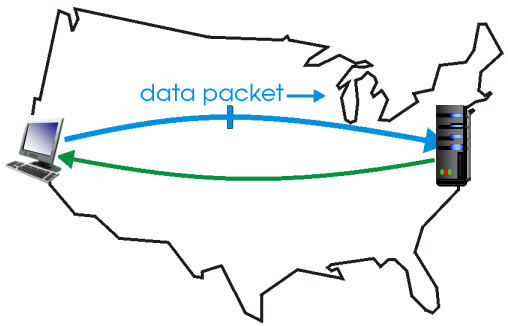
TCP receiver-side buffering

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Pipelined protocols operation

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

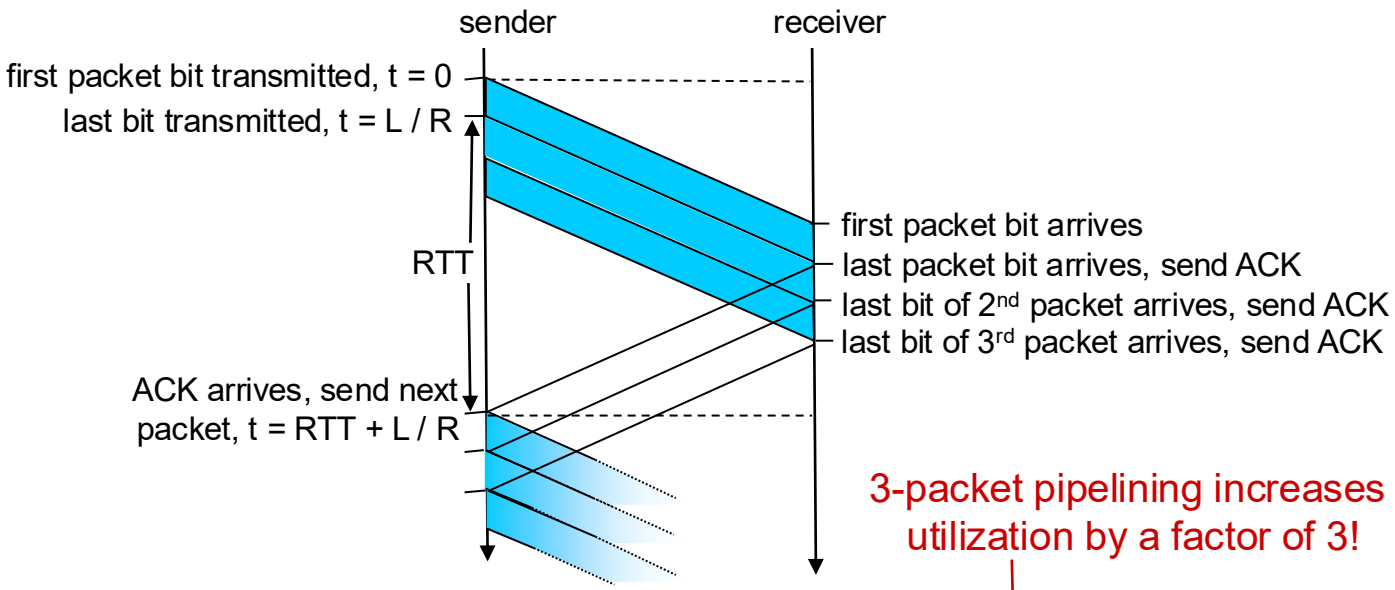
- Range of sequence numbers must be increased
- Buffering at sender and/or receiver



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

Stop-and-wait protocol is inefficient!
Better to send multiple packets without stopping for ACK

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

(U = 0.00027 if stop and wait)

L = number of bits
R = rate (bits/second)

See supplemental material for how to deal with multiple packets in flight

- **Go Back N**
- **Selective Repeat**

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Agenda

1. TCP connections
2. TCP header bytes
3. TCP code
4. Handling multiple clients
5. Supplemental material

Sending with TCP involves a connection step that was not part of UDP

send_tcp.py

```
# server's ip and port
HOST = '127.0.0.1'
PORT = 9090


# create TCP socket to connect to server
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((HOST, PORT))



try:
    message = "Hello, Server!"
    client_socket.sendall(message.encode()) # encode converts to bytes

        # get server's response
    data = client_socket.recv(1024)
    print(f"Received from server: {data.decode()}")
finally:
    client_socket.close()
```

***connect* to server starts**
3-way handshake
Not part of UDP

Use SOCK_STREAM for TCP
SOCK_DGRAM for UDP

***sendall* loops until all Segments are sent**
***send* returns number of bytes sent**
Must loop yourself to send all bytes

Receive TCP from a single client

receive_tcp.py

```
# host ip and port
HOST = '0.0.0.0' #'127.0.0.1' #localhost
PORT = 9090

# create TCP socket listening on ip address and port
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))

# listen for connections
server_socket.listen()
print(f"Server listening on {HOST}:{PORT}...")
client_socket = None

try:
    client_socket, addr = server_socket.accept() # accept returns a new socket for client
    print(f"Connected by {addr}")

    while True:
        data = client_socket.recv(1024)
        if not data:
            break
        print(f"Received: {data.decode()}") # decode converts bytes to string
        client_socket.sendall(b"Message received: " + data) # echo back to client
finally:
    if client_socket:
        client_socket.close()
    server_socket.close()
```

Create socket and bind as with UDP

Listen for connections (unlike UDP which has no connections)

Block and wait for connections

Each connection gets its own socket (like in CS10)

Call *recv*, not *recvfrom* in TCP to get client's message

If data empty, client hung up

Close sockets

- **Listening socket**
- **Client-specific socket**

Exercise

1. Alter **send_tcp.py**
   - Send 5 extra messages quickly after the “Hello Server” message
   - I called them “message{i}” for i=0..4

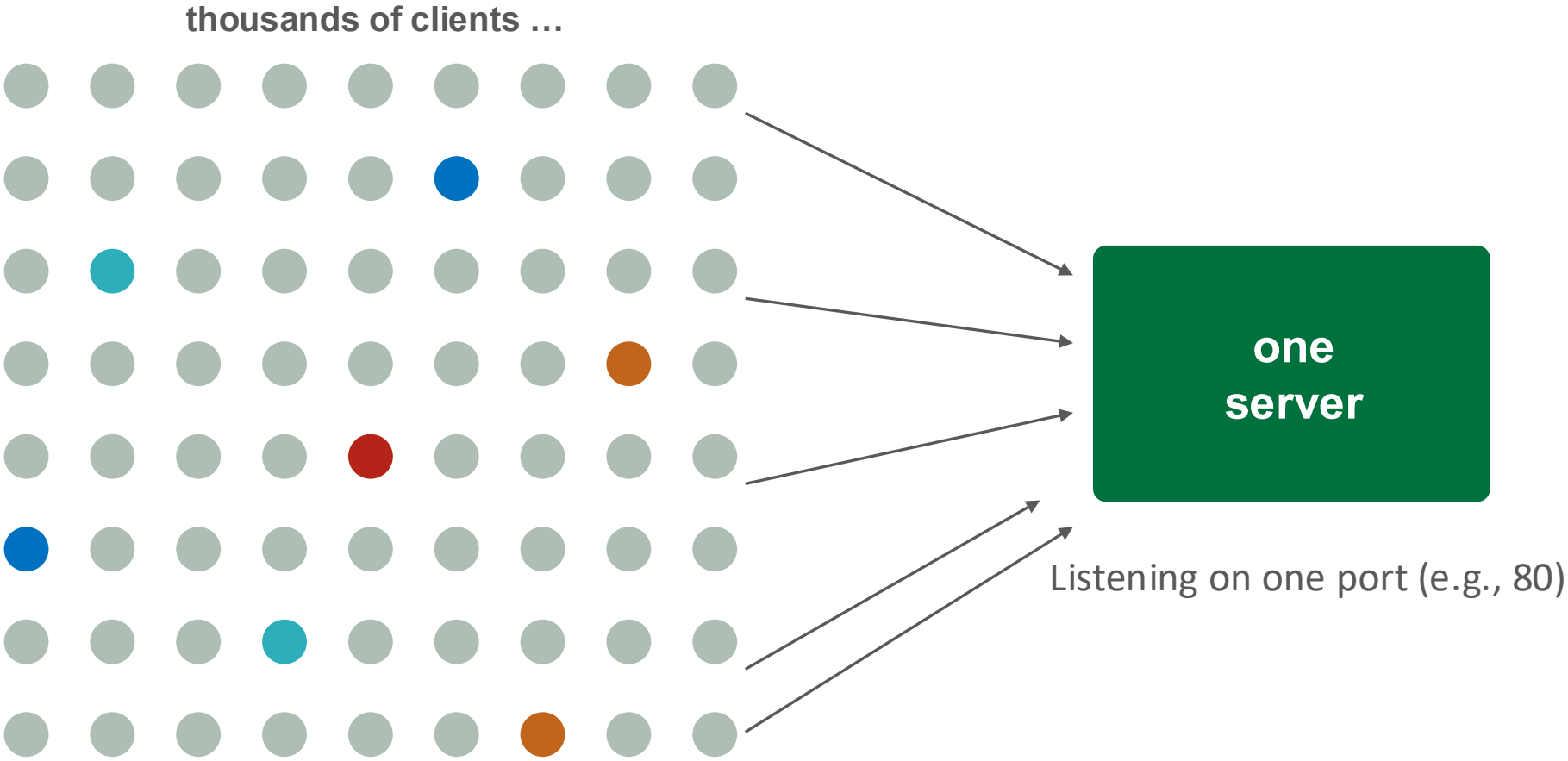
2. Alter **receive_tcp.py**
   - Sleep for 3 seconds after a client connects
   - Sleep allows multiple segments to arrive from client
   - Call *recv* and print the bytes received
   - Describe what happened
   - If bytes from multiple segments get concatenated in the stream, how can you deal with this issue?

Watch this exchange with Wireshark

Agenda

1. TCP connections
2. TCP header bytes
3. TCP code
4. Handling multiple clients
5. Supplemental material

Servers typically handle thousands of connections, two common approaches



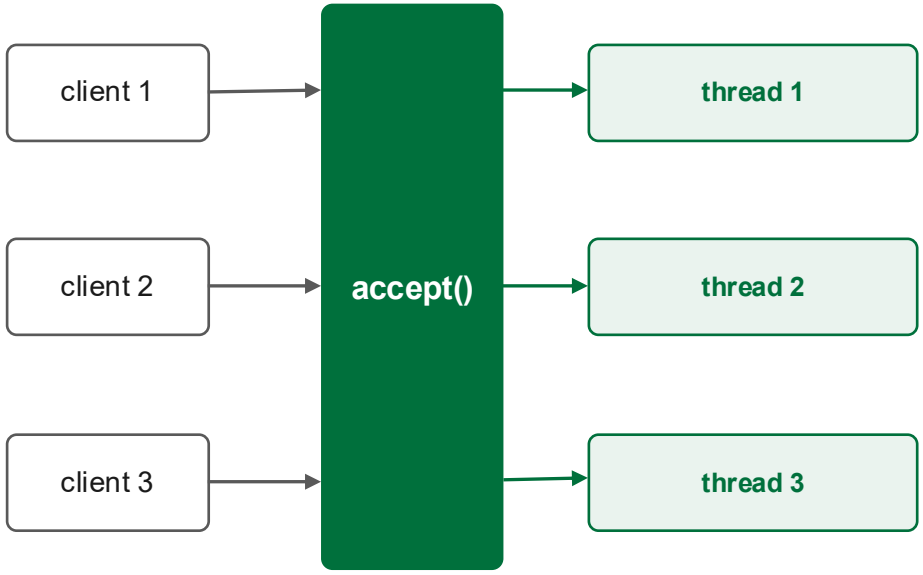
A real server must handle many connections at once

We will examine the two common approaches:

1. **A new worker for each connection (Apache)**
2. **A few workers sharing many connections with an event loop (nginx)**

Approach 1: one thread per connection

One thread per connection



Each new connection gets its own thread running the blocking recv loop

How it works

- *accept* returns a new socket for the client that just connected
- Create a new thread and pass the new socket to it
- Return to listening for new connections on the main socket
- Each thread blocks in a *recv* loop
- The OS scheduler switches between threads, so a slow client blocks only its own thread
- Thread ends when client hangs up

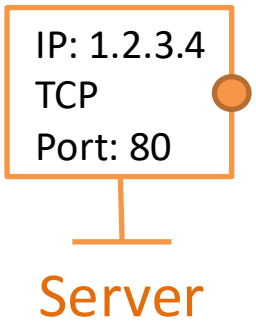
\+ Simple: blocking code, no callbacks

– Each worker costs a stack/process

– 10,000 clients = 10,000 stacks

– Called the “C10k problem”

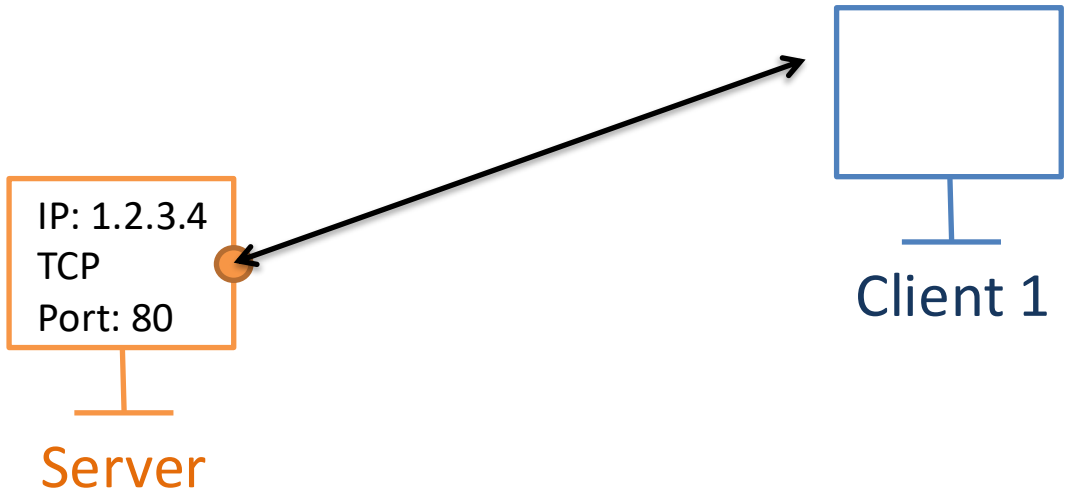
Approach 1: server listens on a socket



Server is listening on
a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

Approach 1: First client connects to socket

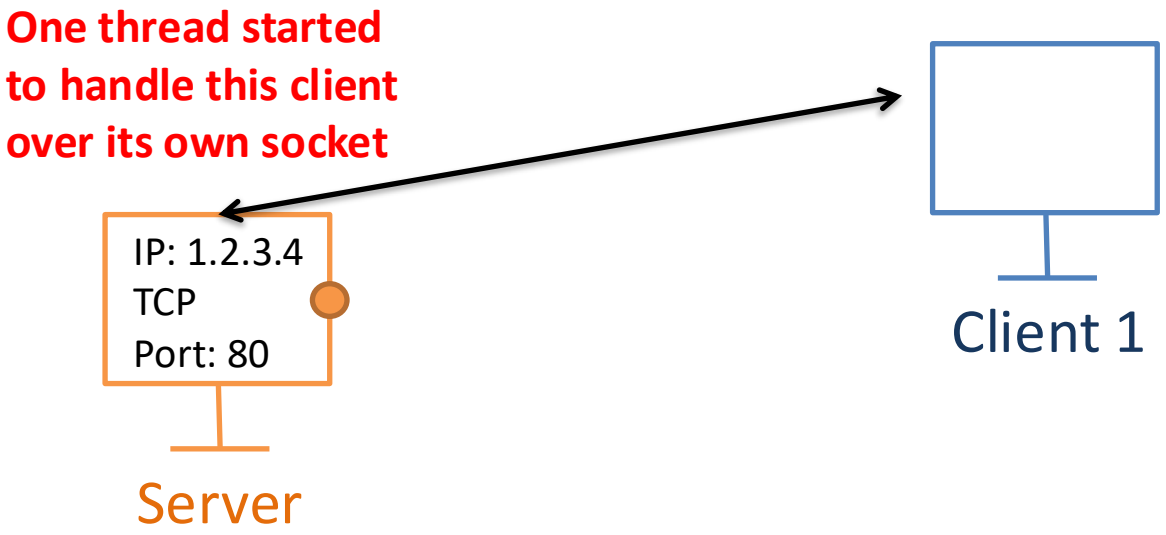


- Client 1 makes connection over socket
- Server receives connection, moves communications to own socket

Server is listening on
a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

Approach 1: First client gets its own socket (server goes back to listening)

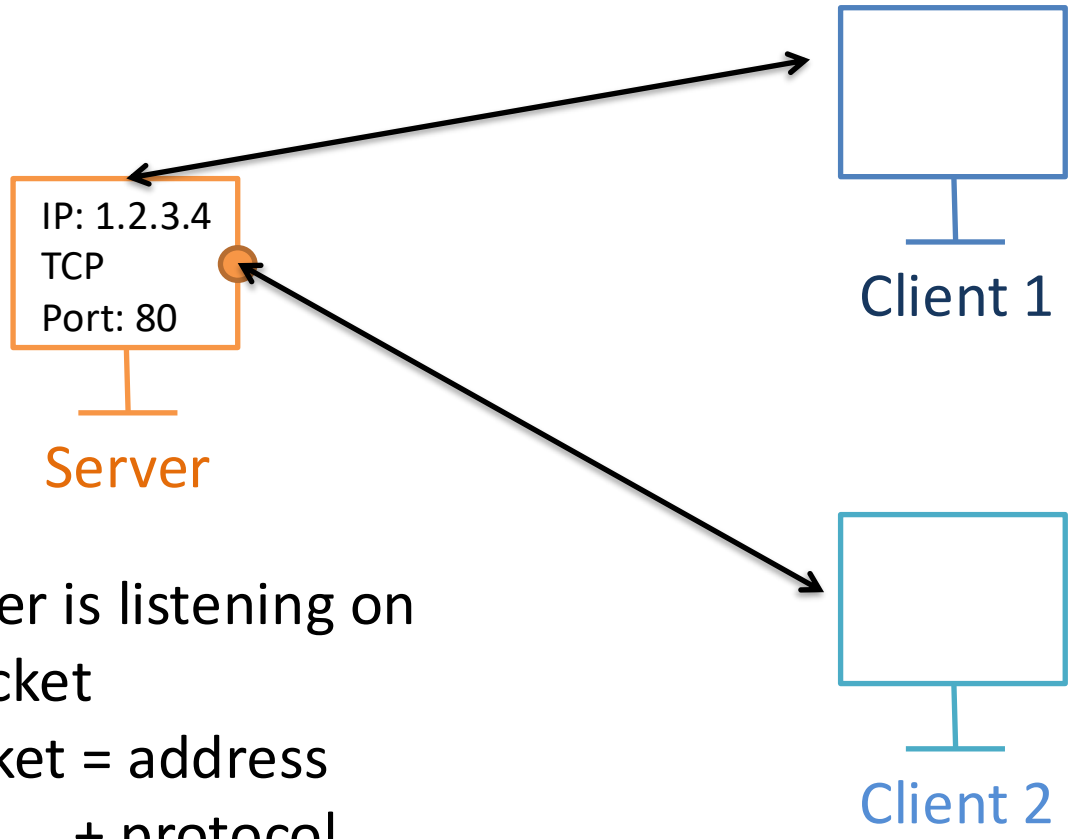


- Client 1 makes connection over socket
- Server receives connection, moves communications to own socket
- Server returns to listening on main socket
- Server talking to Client 1 and ready for others

Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

Approach 1: second client connects

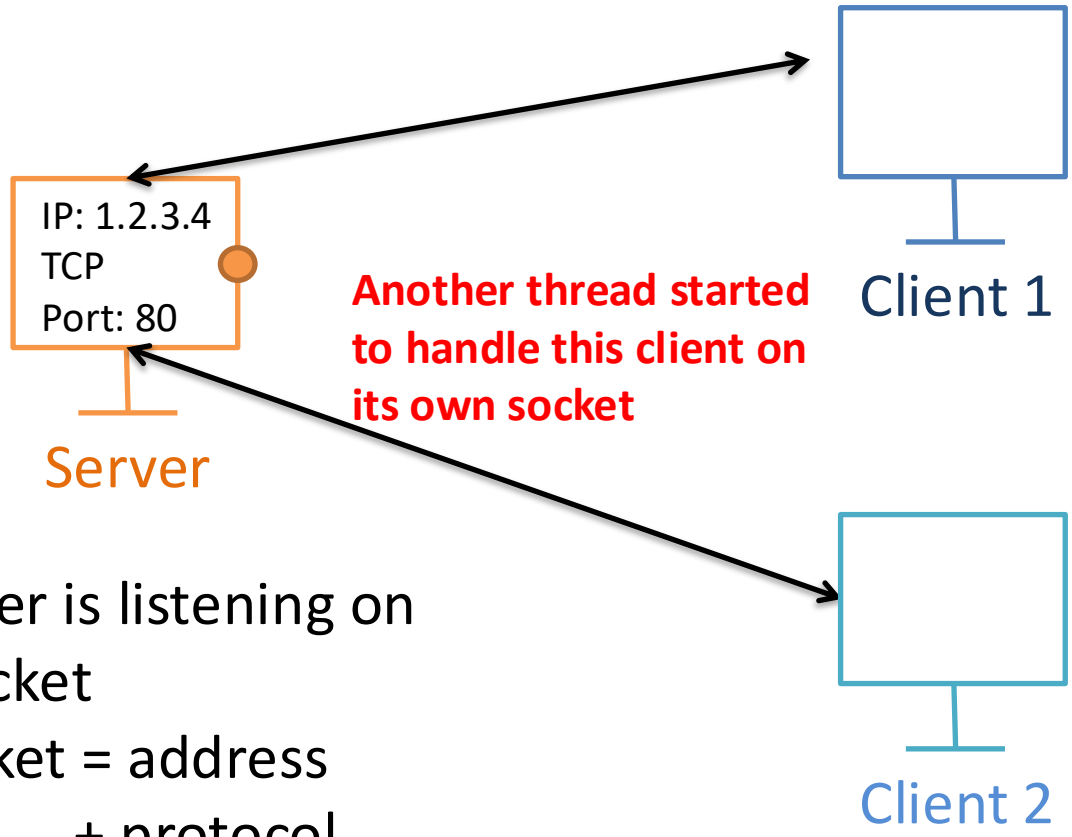


- Client 2 makes connection over socket

Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

Approach 1: second client gets its own socket (server goes back to listening)



Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

- Client 2 makes connection over socket
- Server receives connection, moves communications to own socket
- Server returns to listening
- Server talking to client 1 and 2 ready for others

Approach 1 in Python works like it did in CS10 in Java

receive_tcp_multithreaded.py

```
# host ip and port
HOST = '0.0.0.0' #'127.0.0.1' #localhost
PORT = 9090

# everything that used to be the body of the while loop now lives in here.
# one thread runs this function per connected client, so the loop below is
# free to go straight back to accept() and pick up the next client
def handle_client(client_socket, addr):
    print(f"Connected by {addr}")
    try:
        while True:
            data = client_socket.recv(1024)
            if not data: # empty bytes means the client sent FIN, not "no data yet"
                break
            print(f"Received from {addr}: {data.decode()}") # decode converts bytes to string
    except ConnectionError: # one client crashing should not take down the server
        print(f"Connection lost: {addr}")
    finally:
        client_socket.close()
        print(f"Disconnected: {addr}")

# create TCP socket listening on ip address and port
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # restart right away instead of waiting TIME_WAIT
server_socket.bind((HOST, PORT))


# listen for connections
server_socket.listen()
print(f"Server listening on {HOST}:{PORT}...")
try:
    while True:
        client_socket, addr = server_socket.accept() # accept returns a new socket to the client and the clients ip/port
        # daemon=True so ctrl-c kills the server even with clients still connected
        t = threading.Thread(target=handle_client, args=(client_socket, addr), daemon=True)
        t.start()
except KeyboardInterrupt:
    print("\nShutting down.")
finally:
    server_socket.close()
```

***handle_client* called when a thread started**

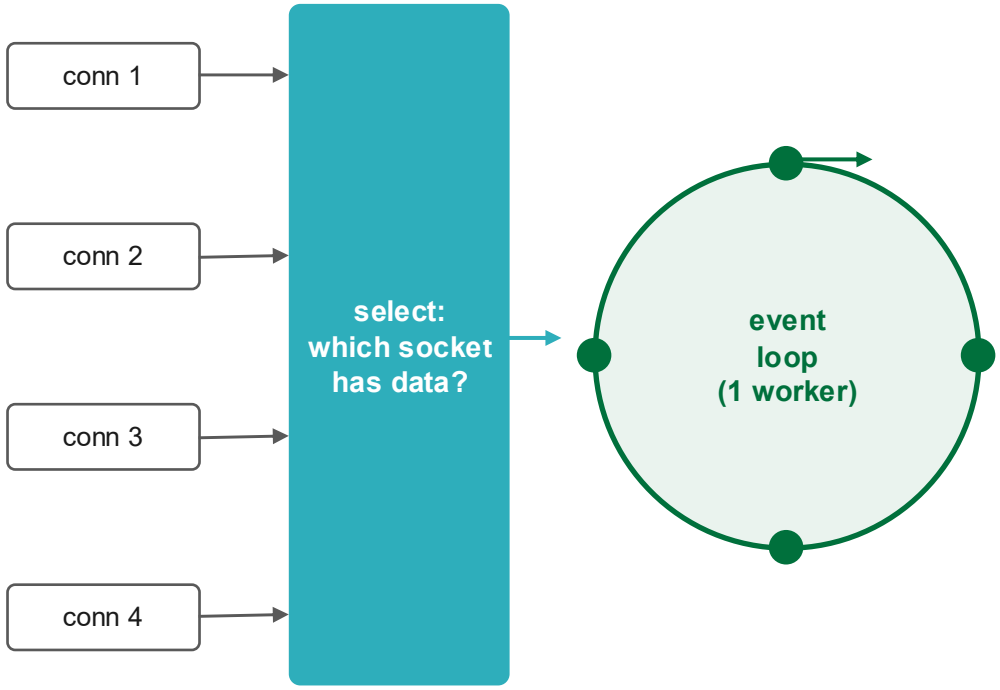
Thread handles client as if it were the single client case

Block until client connects

Start new thread for each connection

Approach 2: a few workers each handle thousands of connections in event loop

One worker with an event loop



Asks the OS which sockets are ready, then services just those — no blocking

You can choose either approach in Lab 2

How it works

- One worker per CPU core, not per connection
- Each worker handles thousands of sockets
- Instead of blocking on one *recv*, the worker asks the OS which sockets have data and services those
- The readiness call is *epoll* (Linux), *kqueue* (BSD/macOS), or *select/poll*
- Sockets are non-blocking: a *recv* with no data returns at once

\+ A few workers serve tens of thousands of connections

– Harder to write, must keep track of each connection's state

– One CPU-bound task stalls every connection the worker handles

Approach 2 in Python

receive_tcp_polling.py

```
#define host ip and port
HOST = '0.0.0.0' #'127.0.0.1' #localhost
PORT = 9090


#the selector is the kernel's "tell me which sockets are ready" service.
#DefaultSelector picks epoll on linux, kqueue on mac, select on windows
sel = selectors.DefaultSelector()
```

<snip>

```
#create TCP socket listening on ip address and port
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #lets us restart right away instead of waiting out TIM
server_socket.bind((HOST, PORT))


#listen for connections
server_socket.listen()
server_socket.setblocking(False)
sel.register(server_socket, selectors.EVENT_READ, lambda: on_acceptable(server_socket))
print(f"Server listening on {HOST}:{PORT}...")


#the event loop. select() blocks until at least one socket is ready, then hands
#back only those, and we run whatever callback we registered for each
try:
    while True:
        for key, mask in sel.select():
            key.data() #key.data is the lambda we registered above
except KeyboardInterrupt:
    print("\nShutting down.")
finally:
    sel.close()
    server_socket.close()
```

Process:

- **Set up a *selector* to watch sockets**
- **Call *on_acceptable* whenever someone connects (but unlike multithreaded, don't block)**
- **When someone connects *on_acceptable* is called back**
  - **Get a socket for that connection**
  - **Set up a callback called *on_readable* when data is available on the new socket**

Approach 2 in Python

receive_tcp_polling.py

```
#define host ip and port
HOST = '0.0.0.0' #'127.0.0.1' #localhost
PORT = 9090


#the selector is the kernel's "tell me which sockets are ready" service.
#DefaultSelector picks epoll on linux, kqueue on mac, select on windows
sel = selectors.DefaultSelector()

<snip>

#create TCP socket listening on ip address and port
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #lets us restart right away instead of waiting out TIME_WAIT
server_socket.bind((HOST, PORT))


#listen for connections
server_socket.listen()
server_socket.setblocking(False)
sel.register(server_socket, selectors.EVENT_READ, lambda: on_acceptable(server_socket))
print(f"Server listening on {HOST}:{PORT}...")


#the event loop. select() blocks until at least one socket is ready, then hands
#back only those, and we run whatever callback we registered for each
try:
    while True:
        for key, mask in sel.select():
            key.data() #key.data is the lambda we registered above
except KeyboardInterrupt:
    print("\nShutting down.")
finally:
    sel.close()
    server_socket.close()
```

Set up a *selector* which will notify when a socket has received bytes

***DefaultSelector* set things up depending on the OS**

Create server socket as before

Note: *setsockopt* allows this program to access a socket right away on a restart otherwise must wait TIME_WAIT (60 secs) until socket released

Set socket to not block

Calls *on_acceptable* callback from lambda

Watch the *selector* on *server_socket* for data
Call *on_acceptable* when a client connects (see next slide), does not block

Approach 2 in Python

receive_tcp_polling.py

```
#a client socket is readable: either data arrived or the client hung up
def on_readable(client):
    data = client.socket.recv(1024)
    if not data: #empty bytes means the client sent FIN, not "no data yet"
        print(f"Disconnected: {client.addr}")
        sel.unregister(client.socket) #stop watching before closing
        client.socket.close()
        return
    print(f"Received from {client.addr}: {data.decode()}") #decode converts bytes to string
    # could echo back to client here if we wanted

#the listening socket is readable: a new client is waiting in the accept queue
def on_acceptable(server_socket):
    client_socket, addr = server_socket.accept() #accept returns a new socket to the client and the clients ip/port
    print(f"Connected by {addr}")
    client_socket.setblocking(False) #never block the one thread we have
    client = Client(client_socket, addr)
    sel.register(client_socket, selectors.EVENT_READ, lambda: on_readable(client))
```

on_readable **reads data from the socket, but only when there is data to be read**

on_acceptable **is called back when someone connects**

Create a non-blocking socket for that connection by calling accept (as we did in the multithreaded version)

Set up the new socket to call back *on_readable* when it has data

If the socket has no data, it does not block

Agenda

1. TCP connections
2. TCP header bytes
3. TCP code
4. Handling multiple clients
5. Supplemental material

Consider a reliable channel between a sender and receiver



reliable service *abstraction*

We will begin with an imaginary situation where data is always delivered

- **Without bit errors**
- **Without dropped packets**
- **In order**

What have we seen in CS10 that keeps two processes in sync? Semaphore!
We will use something somewhat like a semaphore in TCP

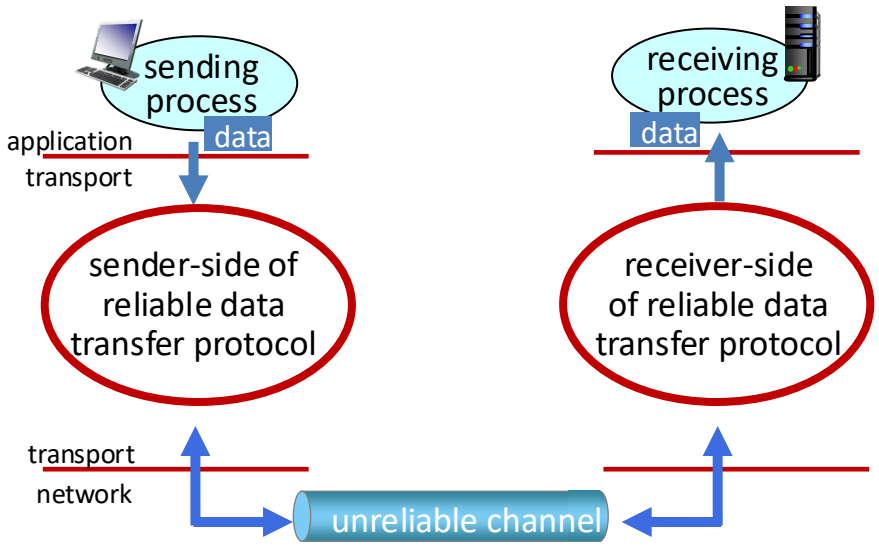
Reliable means:

1. Packets are not corrupted in transit from sender to receiver (no bit flips)
2. No packets are dropped between sender and receiver
3. At the receiver, packets are delivered to the Application Layer in order they were sent
4. Receiver can receive data as fast as sender sends it

A real network does not have all these properties, but this is our starting point
TCP attempts to deliver these properties to the Application Layer (UDP does not)

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

In the real world, the channel between sender and receiver is unreliable

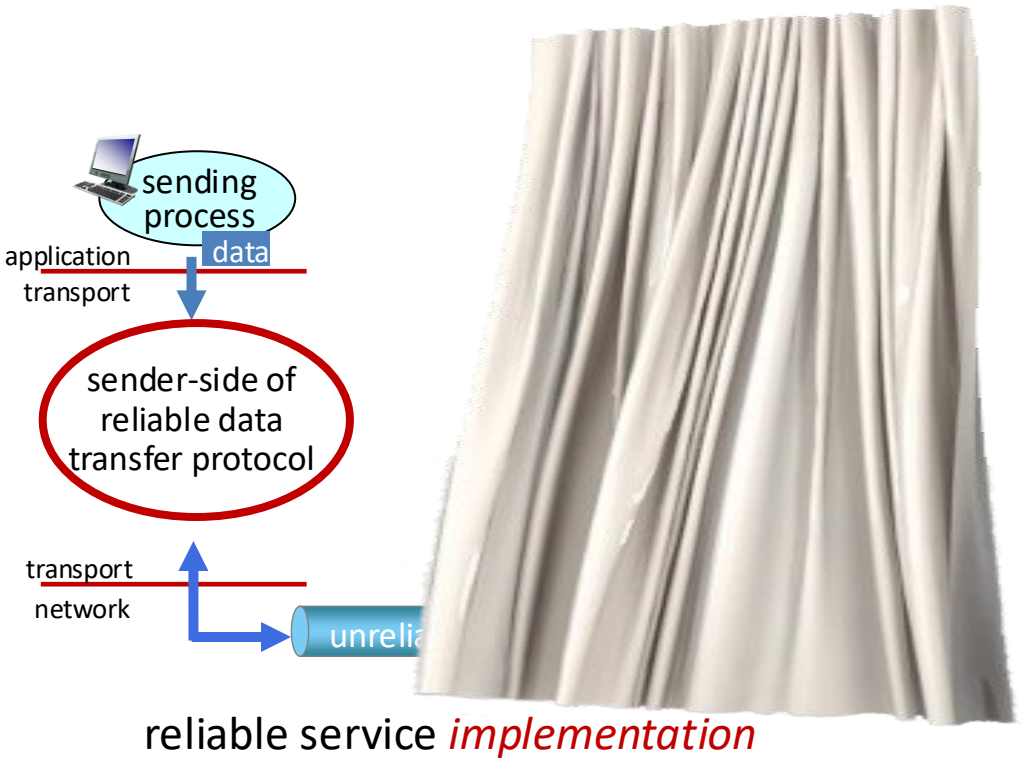


reliable service *implementation*

We want reliable TWO-WAY communication (**reliable data transfer = rdt**)
Sender and receiver will each implement ONE-WAY reliable communication
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Problem: each side does not know the state of the other side



reliable service *implementation*

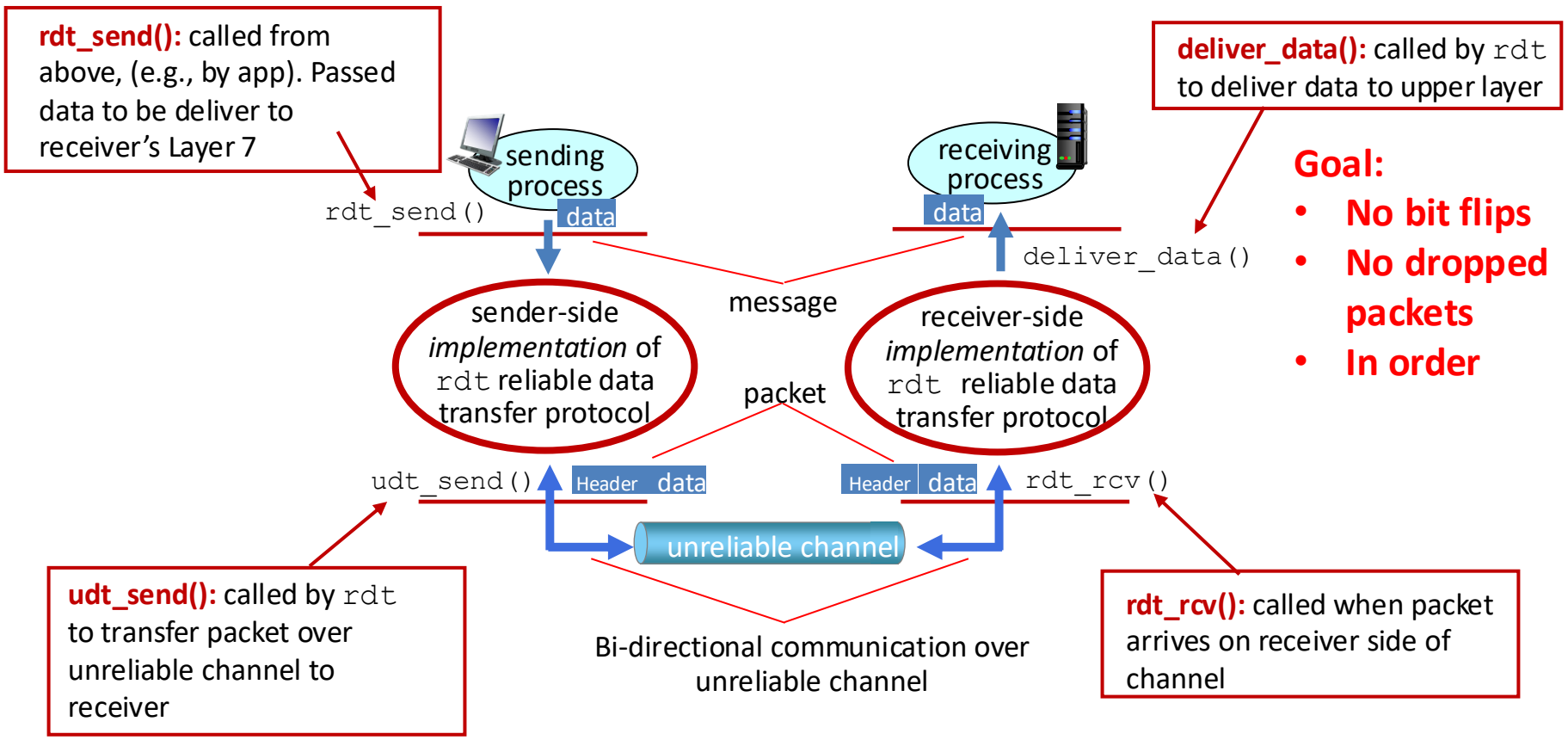
Sender, receiver do *not* know the "state" of each other, e.g., was a sent message received?

- Unless receiver communicates reception to sender via a message

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Reliable data transfer (rdt) over an unreliable channel needs more functions

rdt = reliable data transfer (Layer 4)
udt = unreliable data transfer (Layer 3)



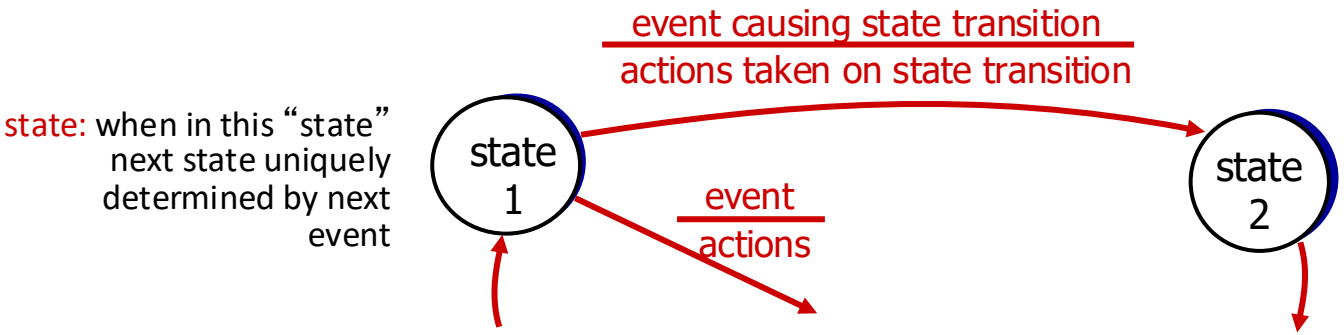
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

We can model reliable data transfer (rdt) using Finite State Machines (FSM)

We will:

- Incrementally develop sender and receiver sides of reliable data transfer protocol (`rdt`)
- Consider only unidirectional data transfer
  - But control info will flow in both directions!
- Use finite state machines (FSM) to specify sender and receiver

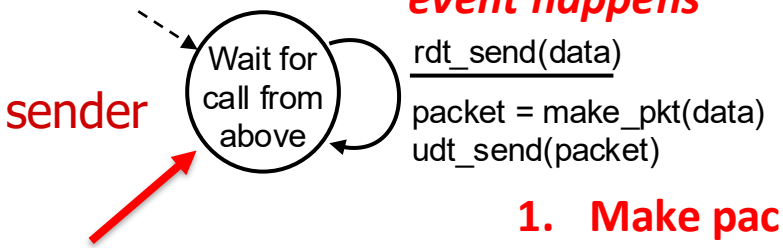
Recall FSMs from CS10
They are a way to formalize rdt specification



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt1.0: reliable transfer over a reliable channel is easy

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
  - In order
- *Separate* FSMs for sender and receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



When an *rdt_send(data)* *event happens*

Call from above is rdt_send from Application (Layer 7) trying to send message to a process on another host

1. **Make packet**
2. **Pass packet to unreliable Network Layer (Layer 3) by calling *udt_send(packet)***

Then

Pass packet to Link Layer (Layer 2)

Pass packet to PHY layer (Layer 1) to receiver

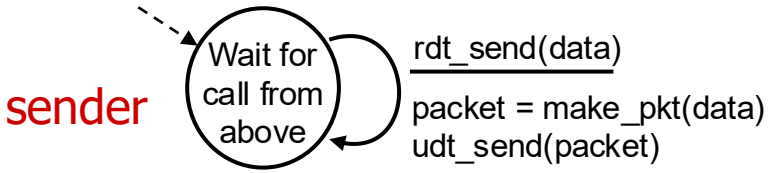
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

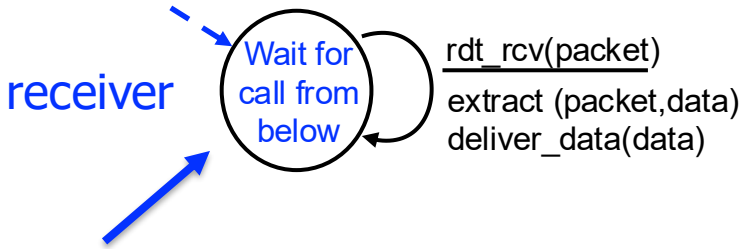
rdt1.0: reliable transfer over a reliable channel is easy

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
  - In order
- *Separate* FSMs for sender and receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



rdt extracts packet and data
Delivers data to App Layer





Call from below is the Network Layer (Layer 3) calling *rdt_rcv(packet)* when packet arrives over unreliable channel
Recall Layer 3 got packet from Layer 2, which got the packet from Layer 1

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0: channel with possible bit errors

For now, still assuming no packet loss

- More realistic, underlying channel may flip bits in packet
  - How to detect bit flips?
  - Add checksum to packet!
- The question: how to recover from bit errors?

How do humans recover from "errors" during conversation?

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0: channel with possible bit errors

For now, still assuming no packet loss

- More realistic, underlying channel may flip bits in packet
  - How to detect bit flips?
  - Checksum!
- The question: how to recover from bit errors?
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - Sender *retransmits* pkt on receipt of NAK

Stop and wait

Sender sends one packet, then waits for receiver response

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0: FSM specification

Start by waiting for Application Layer to send data to rdt layer

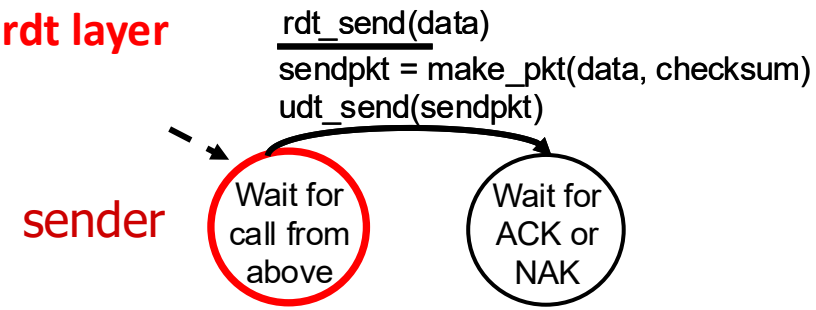


rdt2.0: FSM specification

Start by waiting for Application Layer to send data to rdt layer

App layer calls *rdt_send*, passing data

Make packet and send to Network Layer below with *udt_send*



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

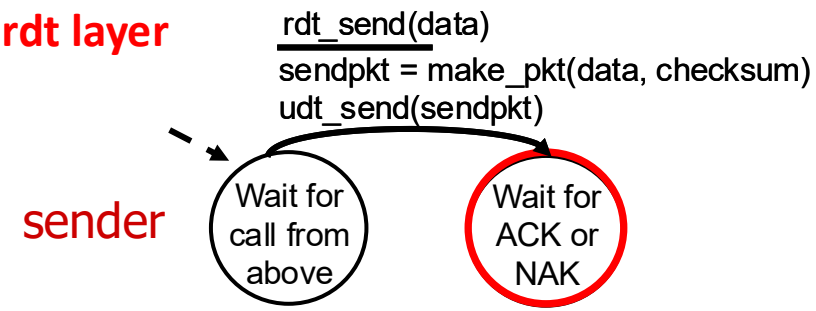
rdt2.0: FSM specification

Start by waiting for Application Layer to send data to rdt layer

App layer calls *rdt_send*, passing data

Make packet and send to Network Layer below with *udt_send*

Transition state



Sender cannot accept more data from the Application Layer until an ACK or NAK is received (stop-and-wait protocol)

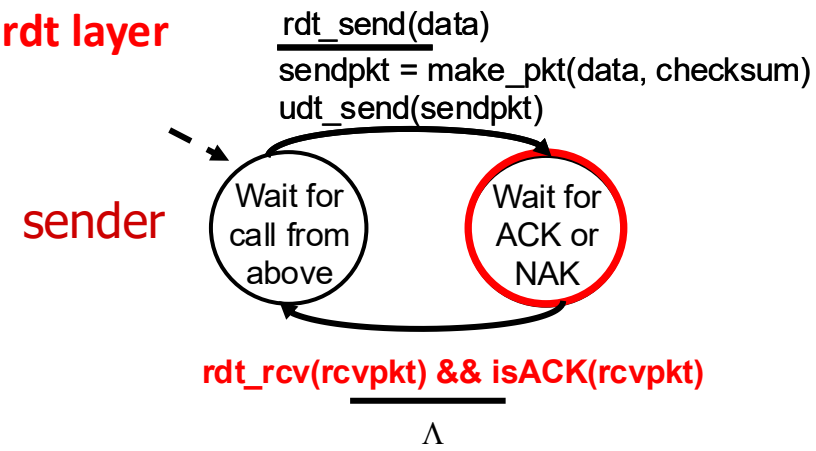
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0: FSM specification

Start by waiting for Application Layer to send data to rdt layer

App layer calls *rdt_send*, passing data
Make packet and send to Network Layer below with *udt_send*

Transition state



Sender cannot accept more data from the Application Layer until an ACK or NAK is received (stop-and-wait protocol)

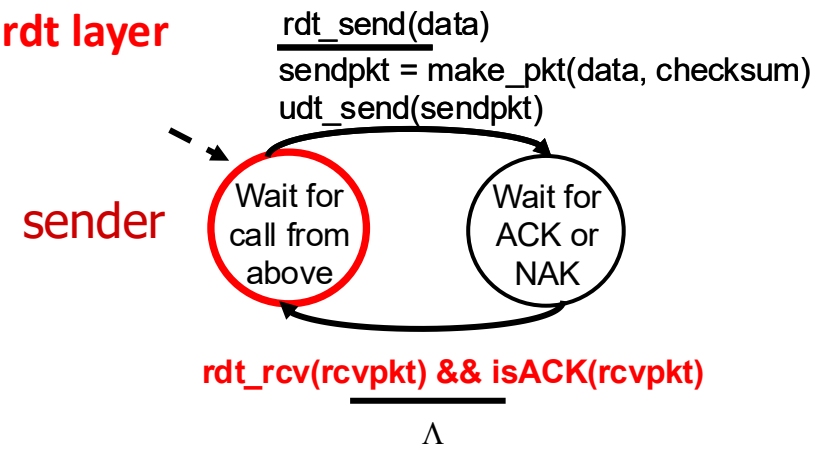
If sender receives ACK from receiver, then done (for this packet)

rdt2.0: FSM specification

Start by waiting for Application Layer to send data to rdt layer

App layer calls *rdt_send*, passing data
Make packet and send to Network Layer below with *udt_send*

Transition state



Sender cannot accept more data from the Application Layer until an ACK or NAK is received (stop-and-wait protocol)

If sender receives ACK from receiver, then done (for this packet)

- **Transition state to wait for App to send more data**

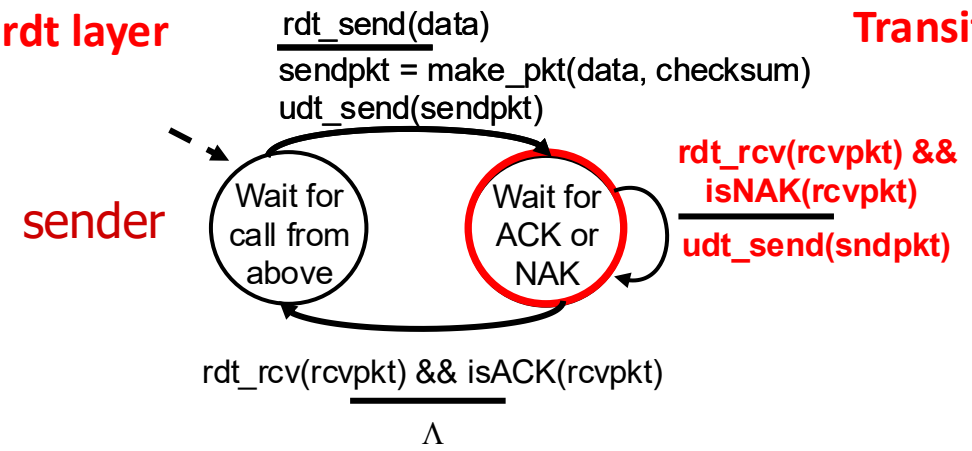
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0: FSM specification

Start by waiting for Application Layer to send data to rdt layer

App layer calls *rdt_send*, passing data
Make packet and send to Network Layer below with *udt_send*

Transition state



Sender cannot accept more data from the Application Layer until an ACK or NAK is received (stop-and-wait protocol)

If sender receives ACK from receiver, then done (for this packet)

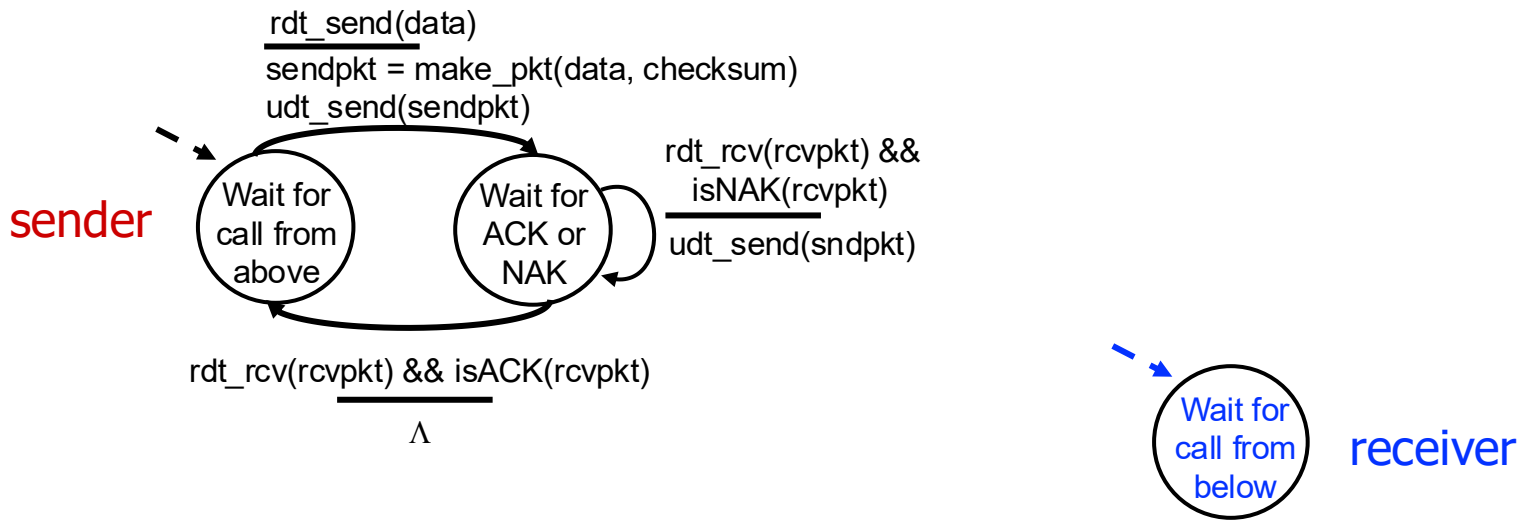
- **Transition state to wait for App to send more data**

Else if sender receives NAK from receiver
Then receiver had bit errors, what to do?

- **Resend packet**
- **Wait for ACK or NAK reply**
- ***Do not transition state***

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

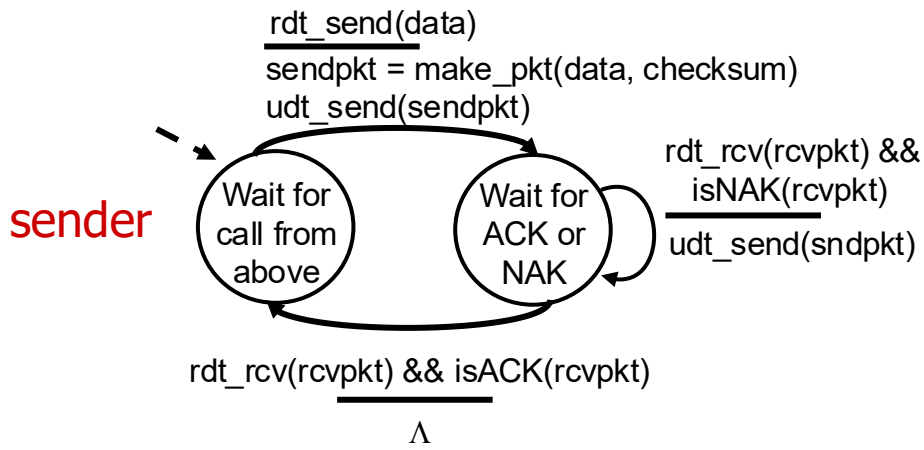
rdt2.0: FSM specification



Start by waiting for Network Layer receive packet and send to rdt layer

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0: FSM specification





receiver

If checksum passes

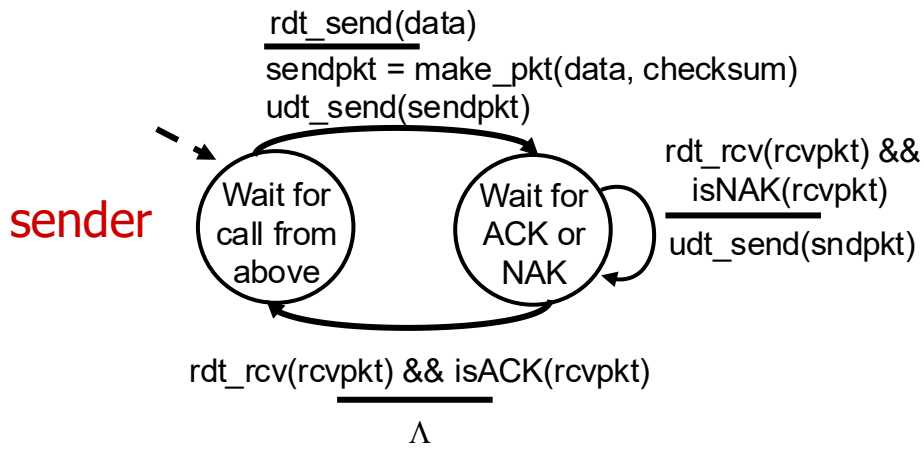
- **Extract data**
- **Pass data to App Layer**
- **Send ACK to sender**

rdt_rcv(rcvpkt) && **notcorrupt(rcvpkt)**
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

Start by waiting for Network Layer receive packet and send to rdt layer

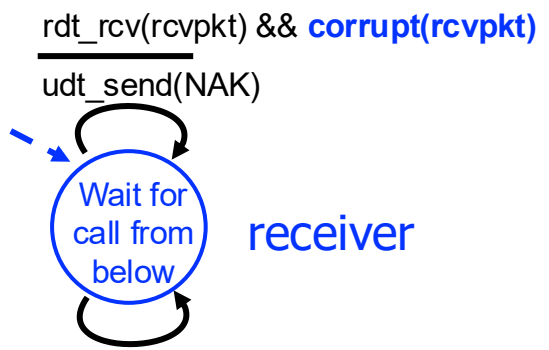
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0: FSM specification



If checksum fails

- **Send NAK to sender**



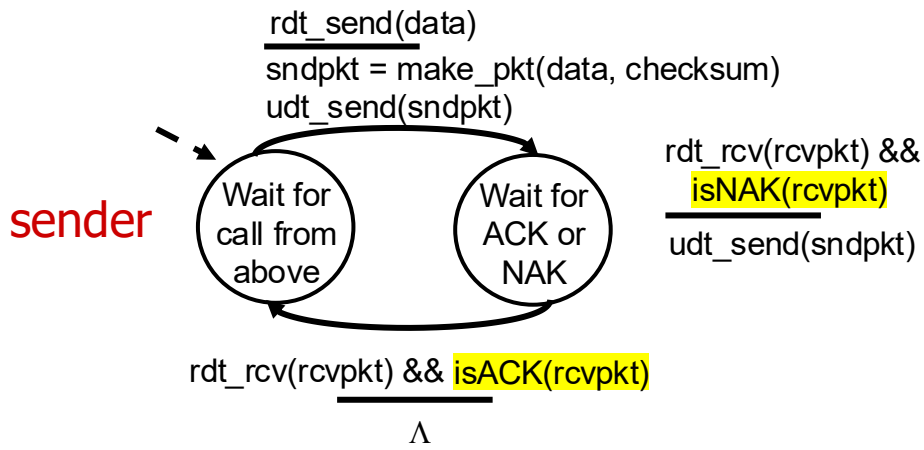
If checksum passes

- **Extract data**
- **Pass data to App Layer**
- **Send ACK to sender**

Start by waiting for Network Layer receive packet and send to rdt layer

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Receiver's state is not known by the sender (unless the receiver tells the sender)



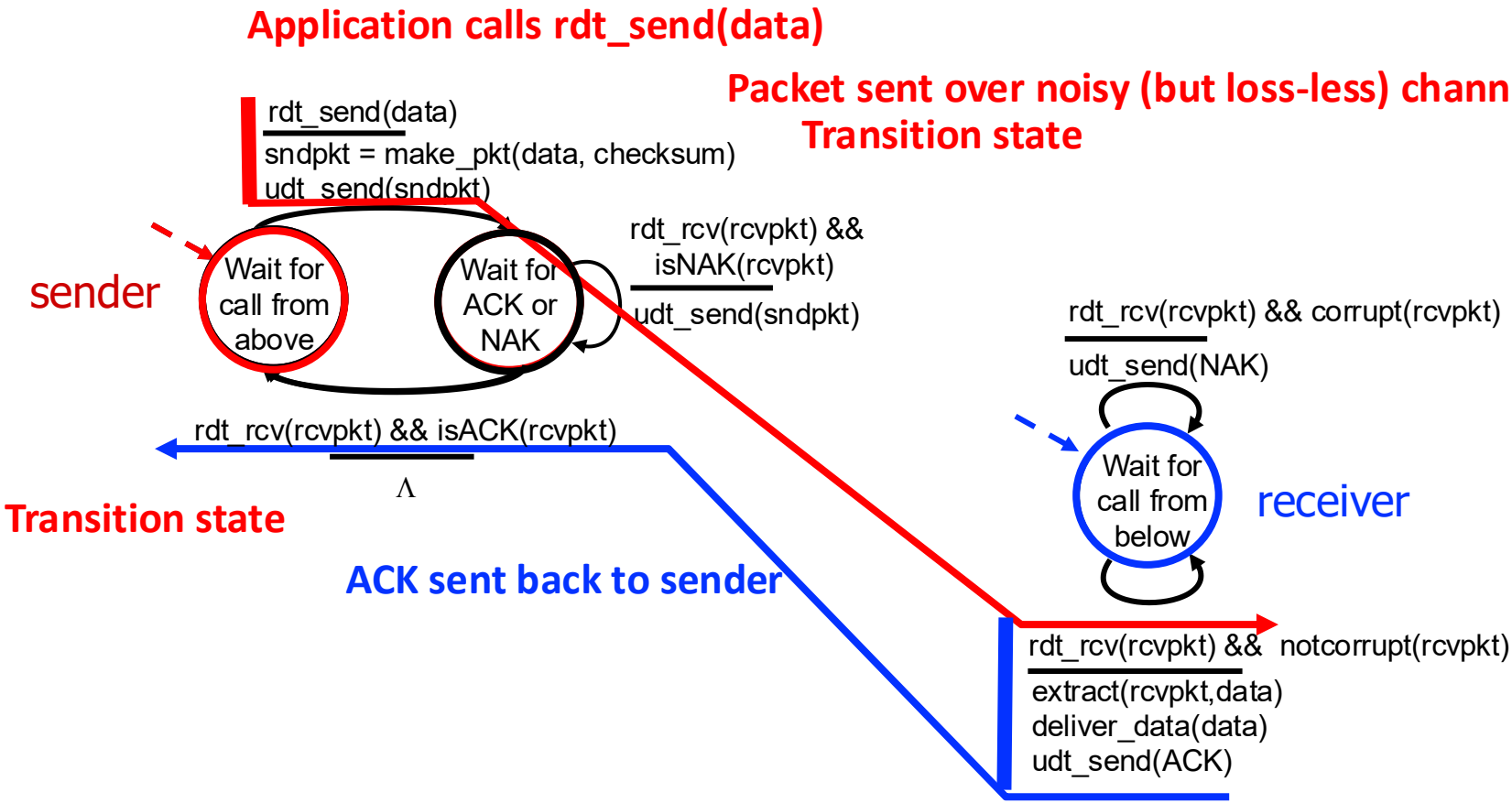
Recall we are still assuming no packet loss



Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless the state somehow communicated from receiver to sender

- That's why we need a protocol that defines how the receiver will inform the sender!

rdt2.0: operation with no errors



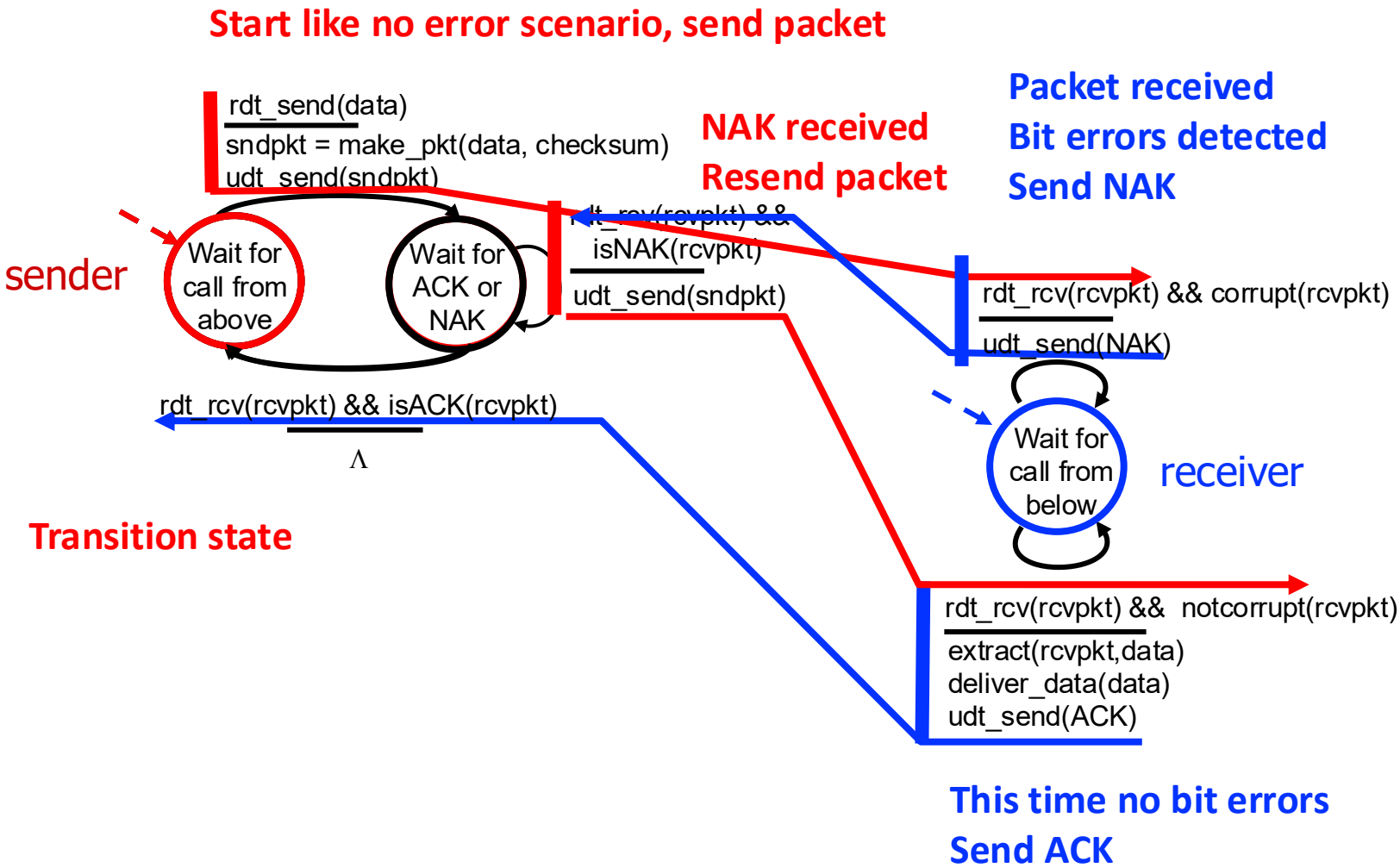
Receiver layers 1 - 3 process packet

Not corrupted

- **Pass data to rdt (Layer 4)**
- **Send ACK to sender**

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0: corrupted packet scenario



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- Sender doesn't know what happened at receiver!
- Data was received, but reply was garbled (must add checksum to ACK/NAK to detect errors at sender)
- ACK the ACK?
  - What if ACKACK not received or ACKACKACK reply not received?!?
- Just resend the packet?
  - Possible duplicate packets at the receiver
  - Apps are not excepting duplicate packets
  - They expect data without errors in order, each packet delivered once!

Handling duplicates:

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

How many seq numbers will we need?
Just two with stop and wait!

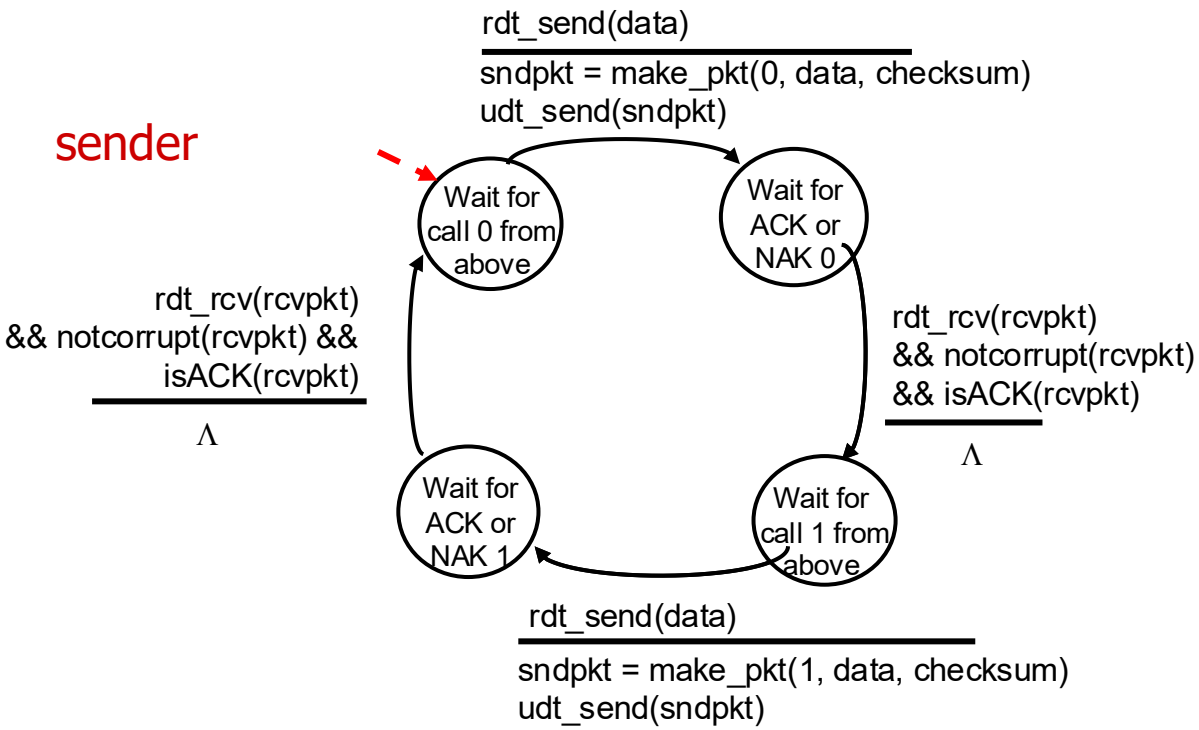
Stop and wait

Sender sends one packet, then waits for receiver response

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: sender, handling garbled ACK/NAKs

A 1-bit sequence number will suffice for now (channel does not lose packets at this point)

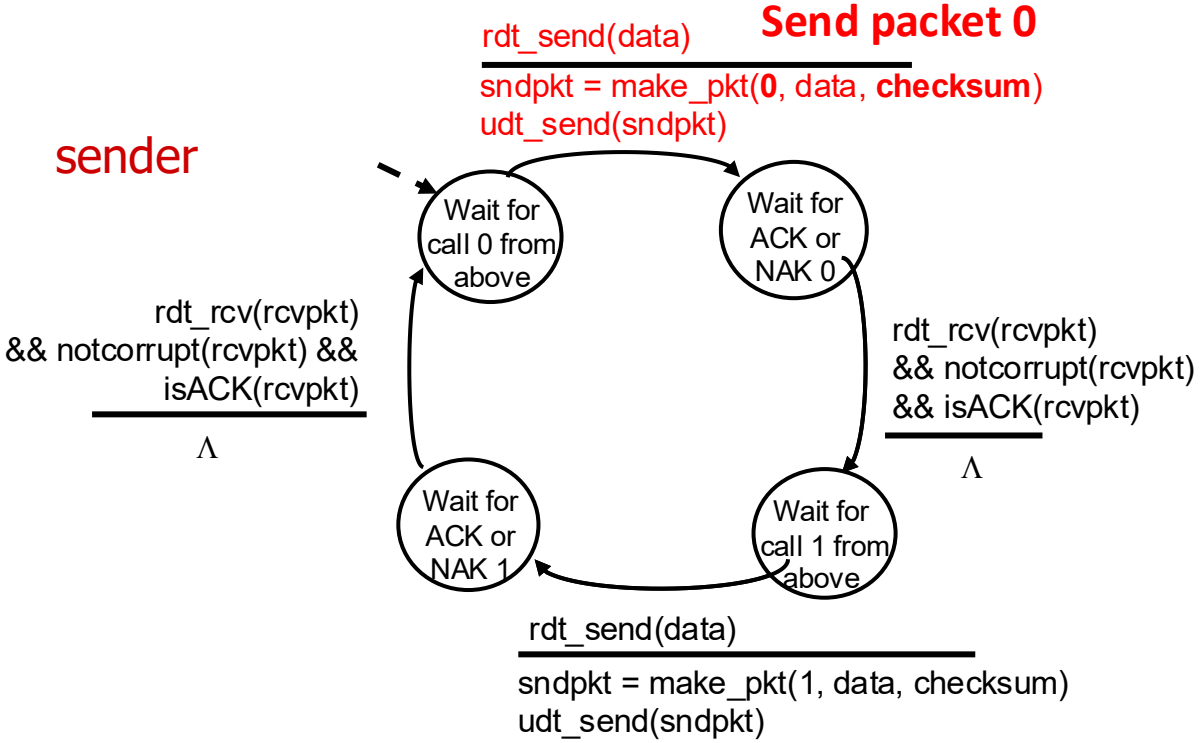


Same states as before, but now one set of states for packet 0 and another set of states for packet 1

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: sender, handling garbled ACK/NAKs

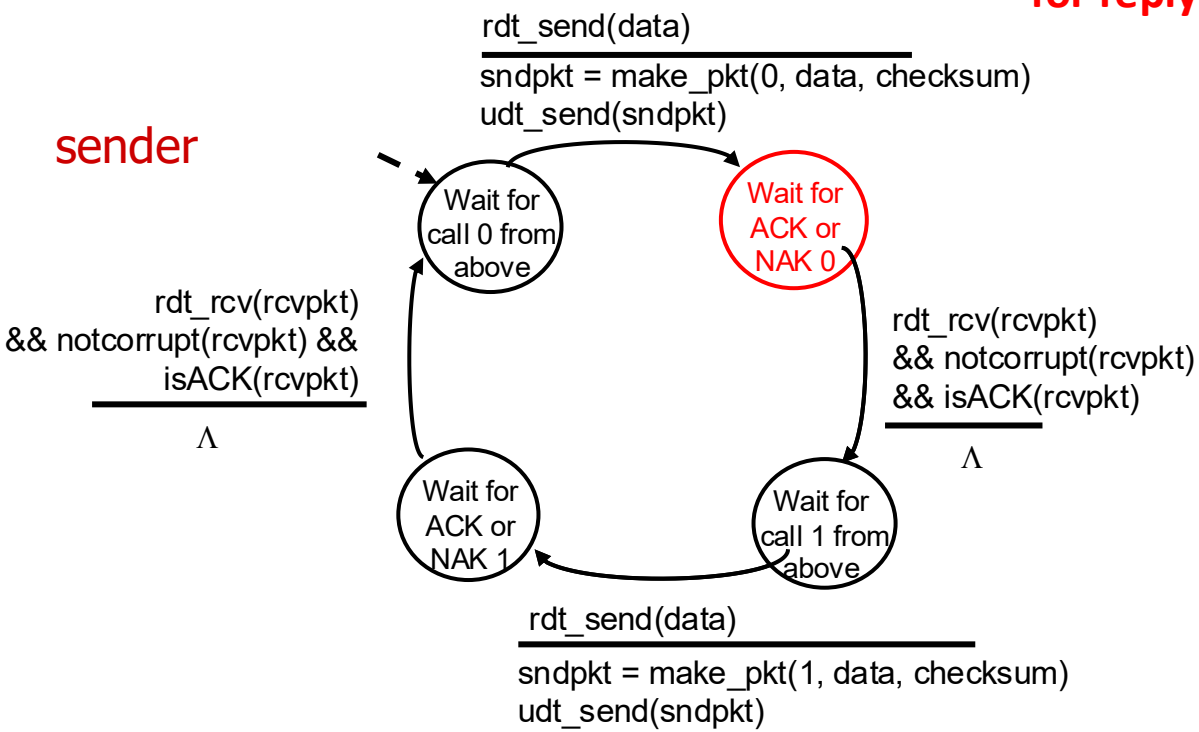
Application Layer sends data to rdt layer
Make packet with seq=0 and checksum
Send packet 0



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

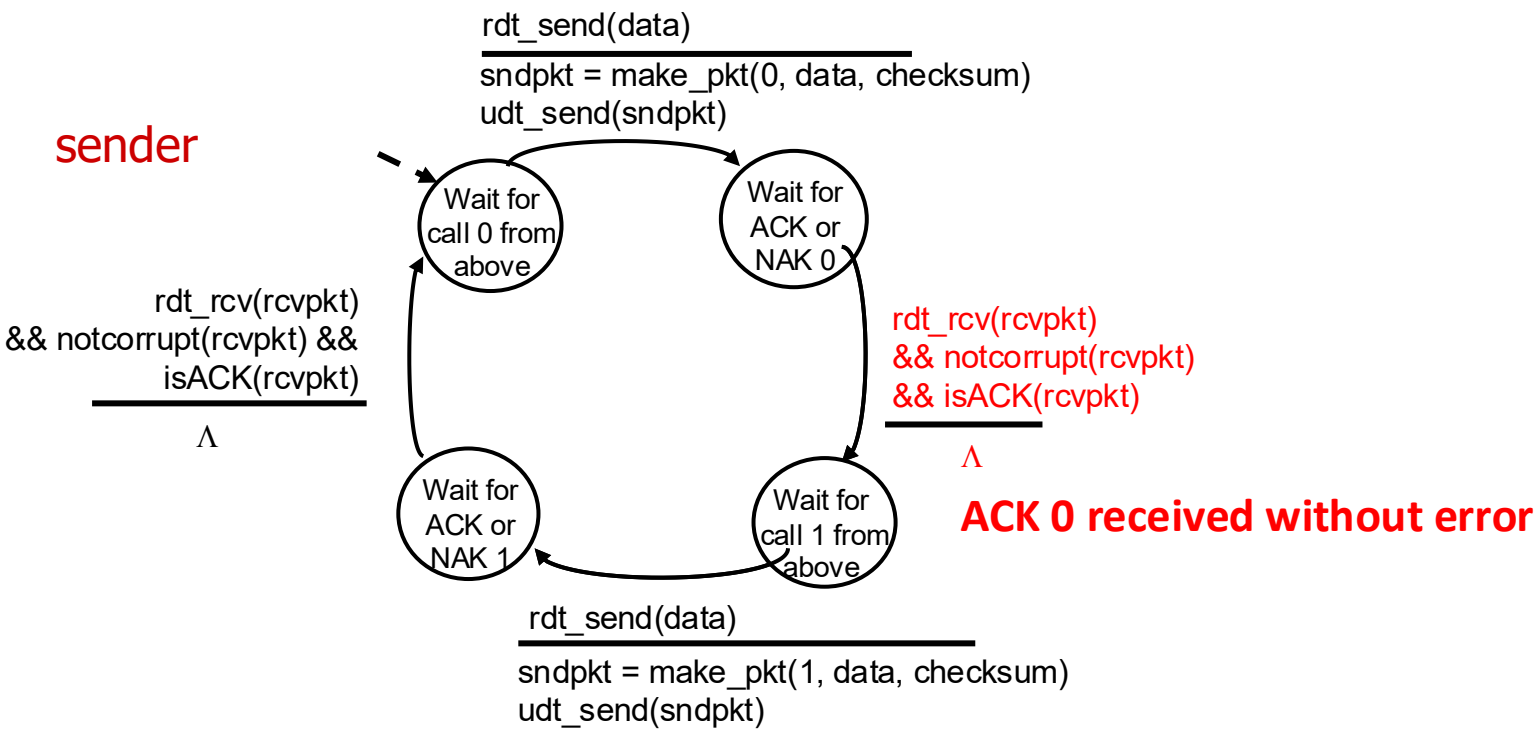
rdt2.1: sender, handling garbled ACK/NAKs

Transition state to wait for reply for packet 0



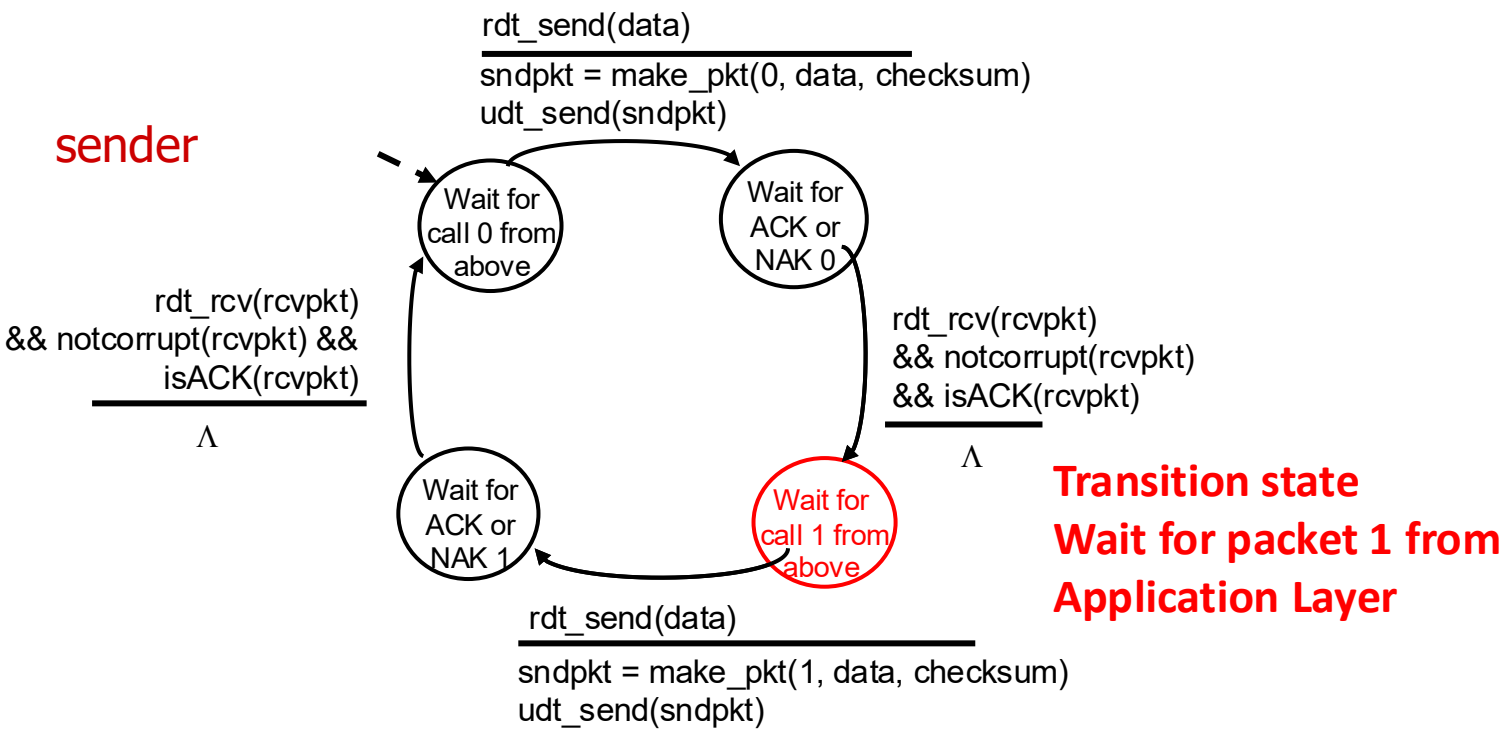
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: sender, handling garbled ACK/NAKs



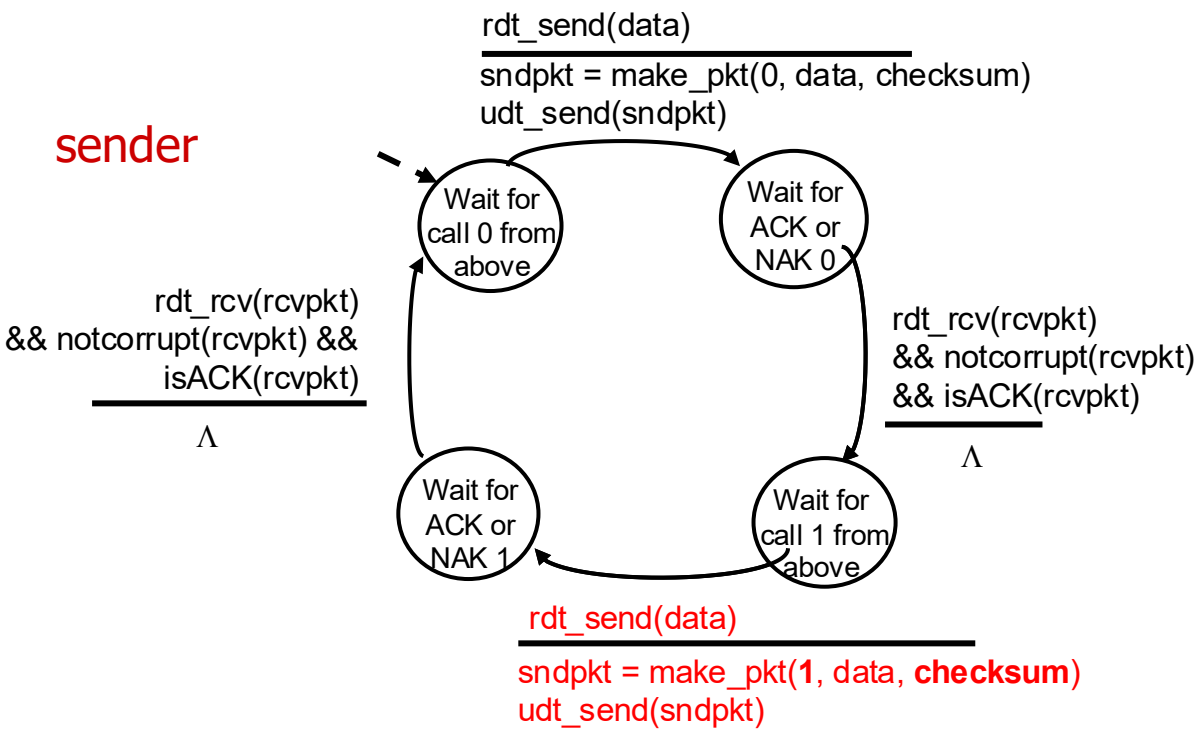
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: sender, handling garbled ACK/NAKs



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

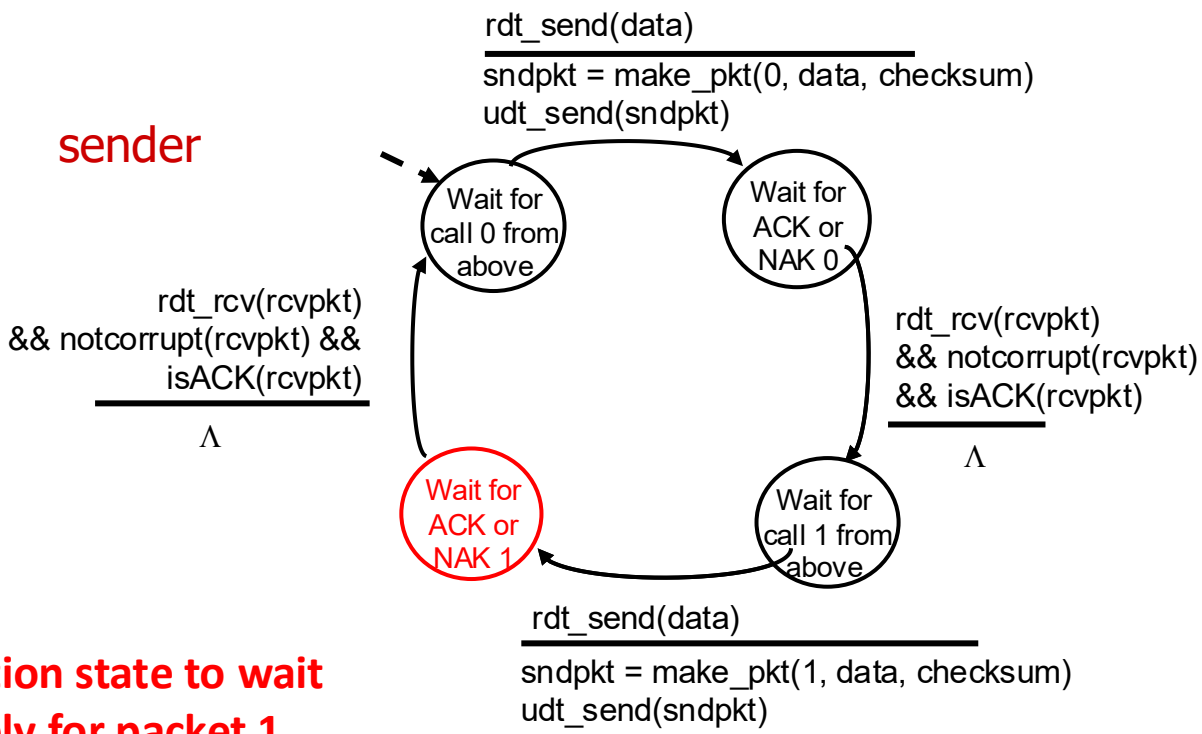
rdt2.1: sender, handling garbled ACK/NAKs



Packet 1 received from Application Layer
Make packet with seq=1 and checksum
Send packet 1

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

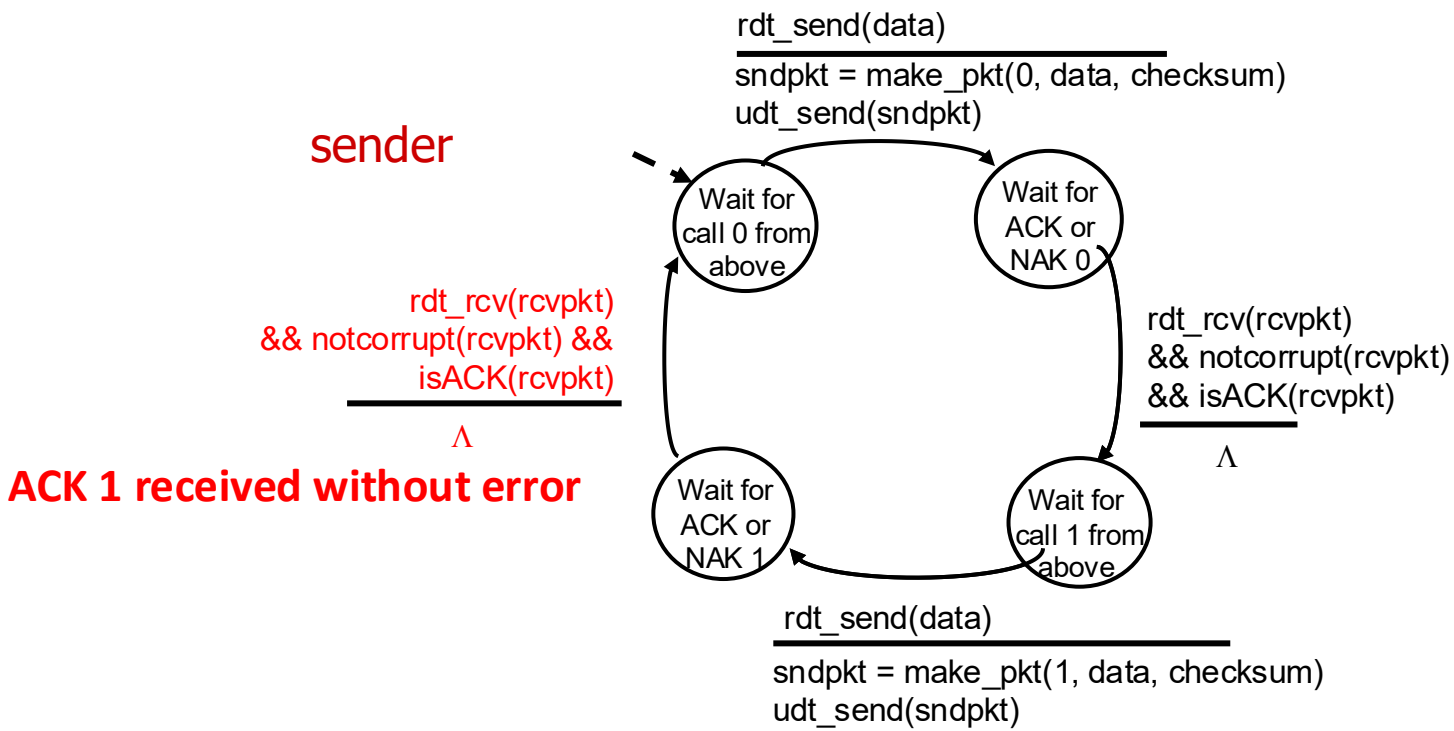
rdt2.1: sender, handling garbled ACK/NAKs



Transition state to wait for reply for packet 1

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

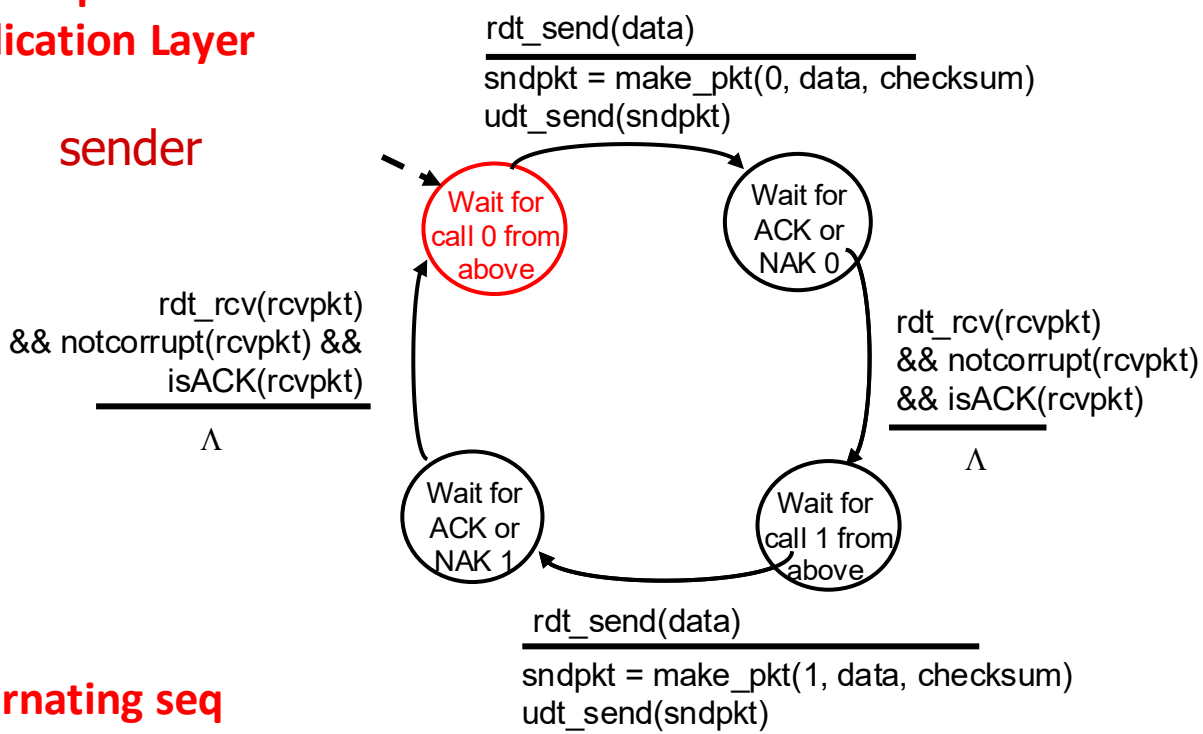
rdt2.1: sender, handling garbled ACK/NAKs



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: sender, handling garbled ACK/NAKs

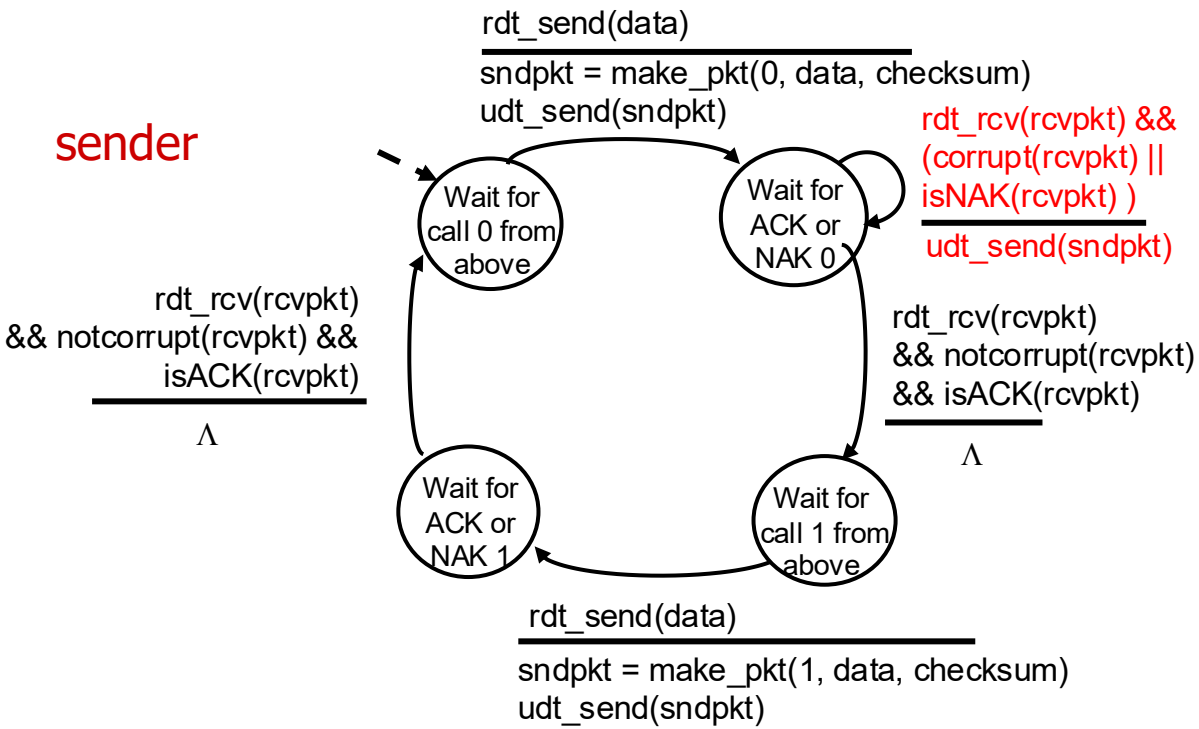
Transition state
Wait for next packet 0
from Application Layer



Note: alternating seq numbers between 0 and 1 works with Stop and Wait

rdt2.1: sender, handling garbled ACK/NAKs

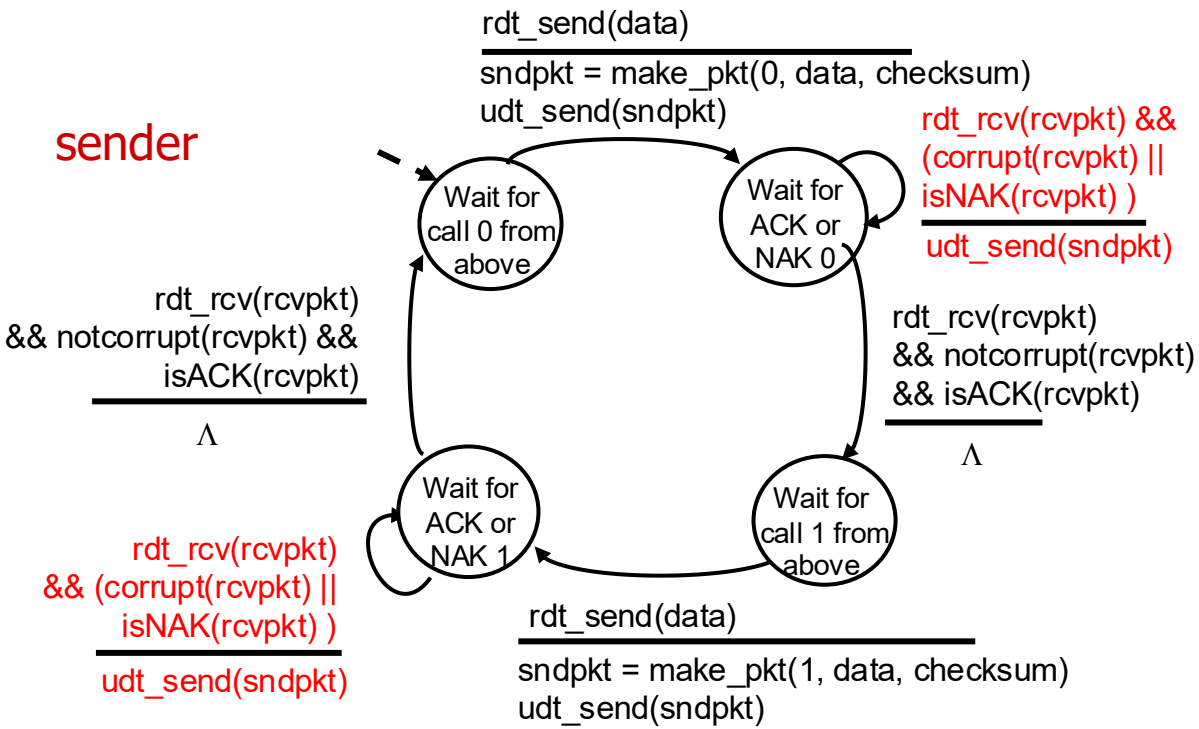
Add step to resend packet 0 if reply corrupt or NAK



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: sender, handling garbled ACK/NAKs

Add step to resend packet 0 if reply corrupt or NAK

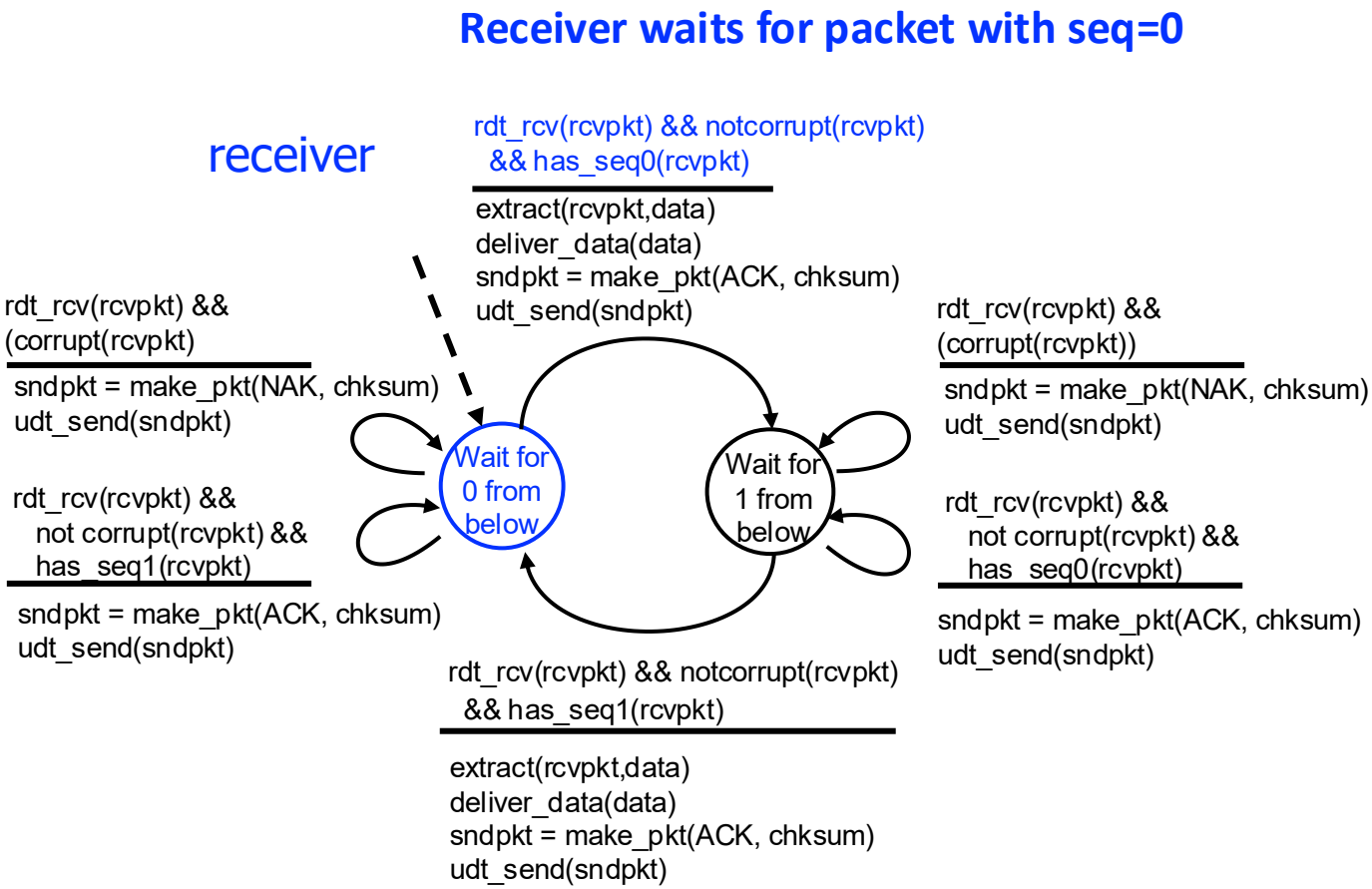


Same for packet 1

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: receiver

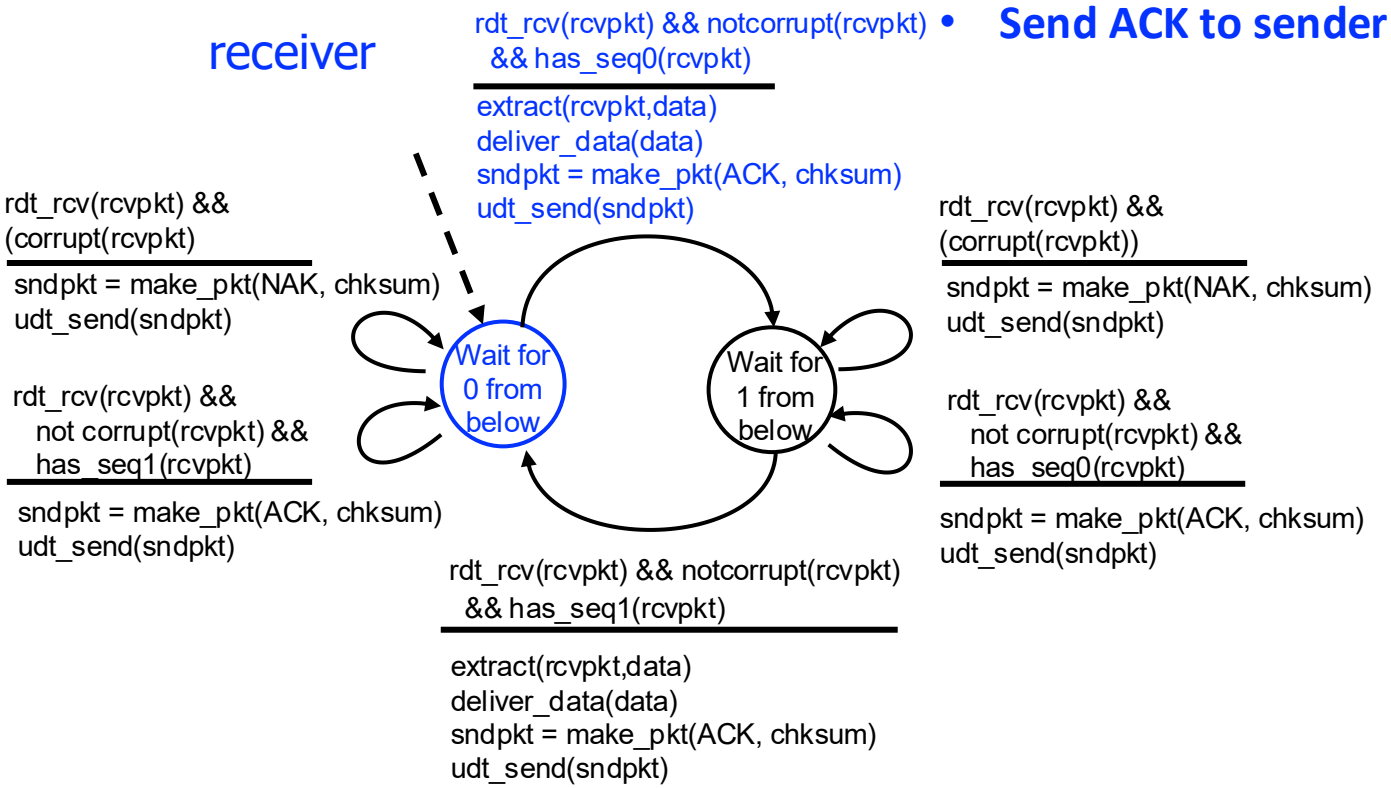
rdt2.1: receiver, handling garbled ACK/NAKs



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

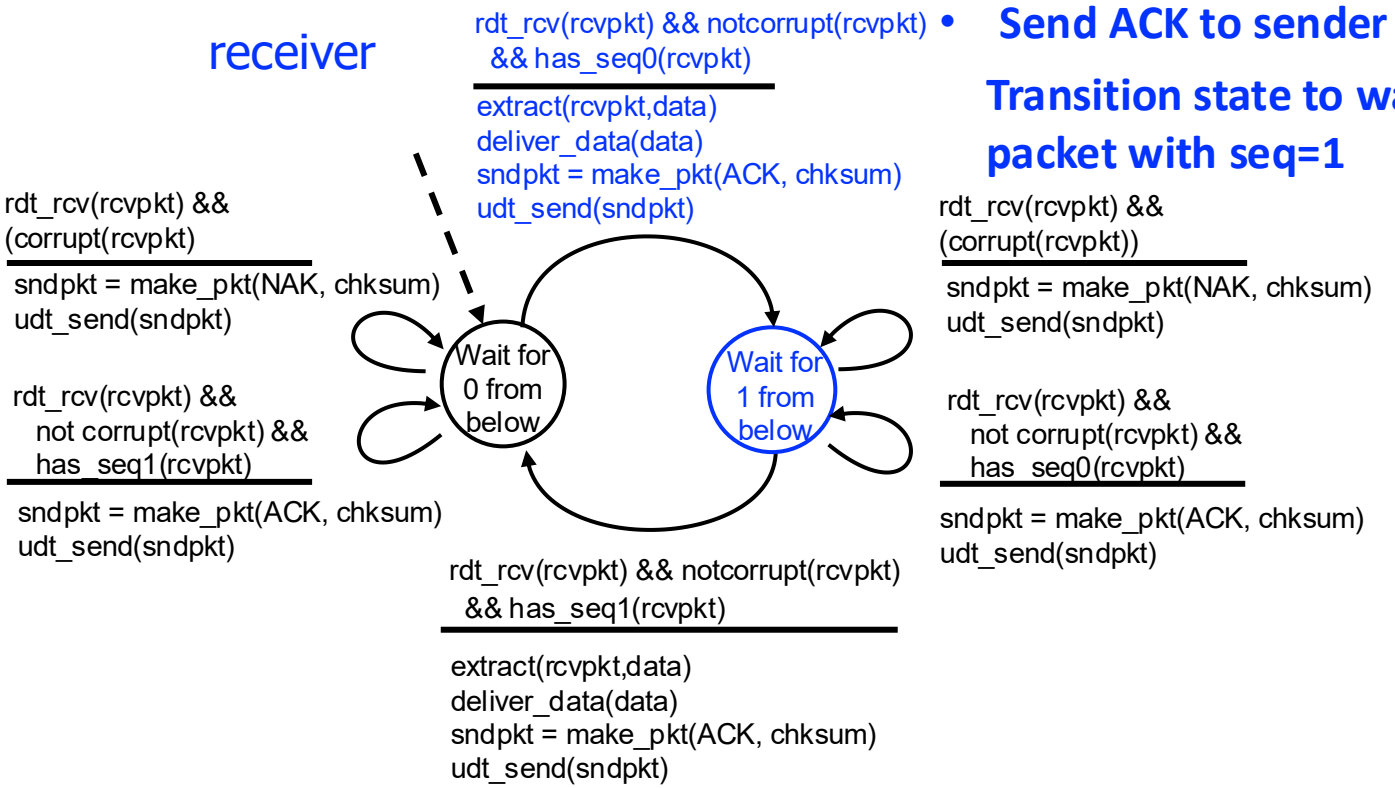
rdt2.1: receiver, handling garbled ACK/NAKs

If seq=0 and not corrupt
- **Send data to Application Layer**
- **Send ACK to sender**



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: receiver, handling garbled ACK/NAKs



If seq=0 and not corrupt

- **Send data to Application Layer**
- **Send ACK to sender**

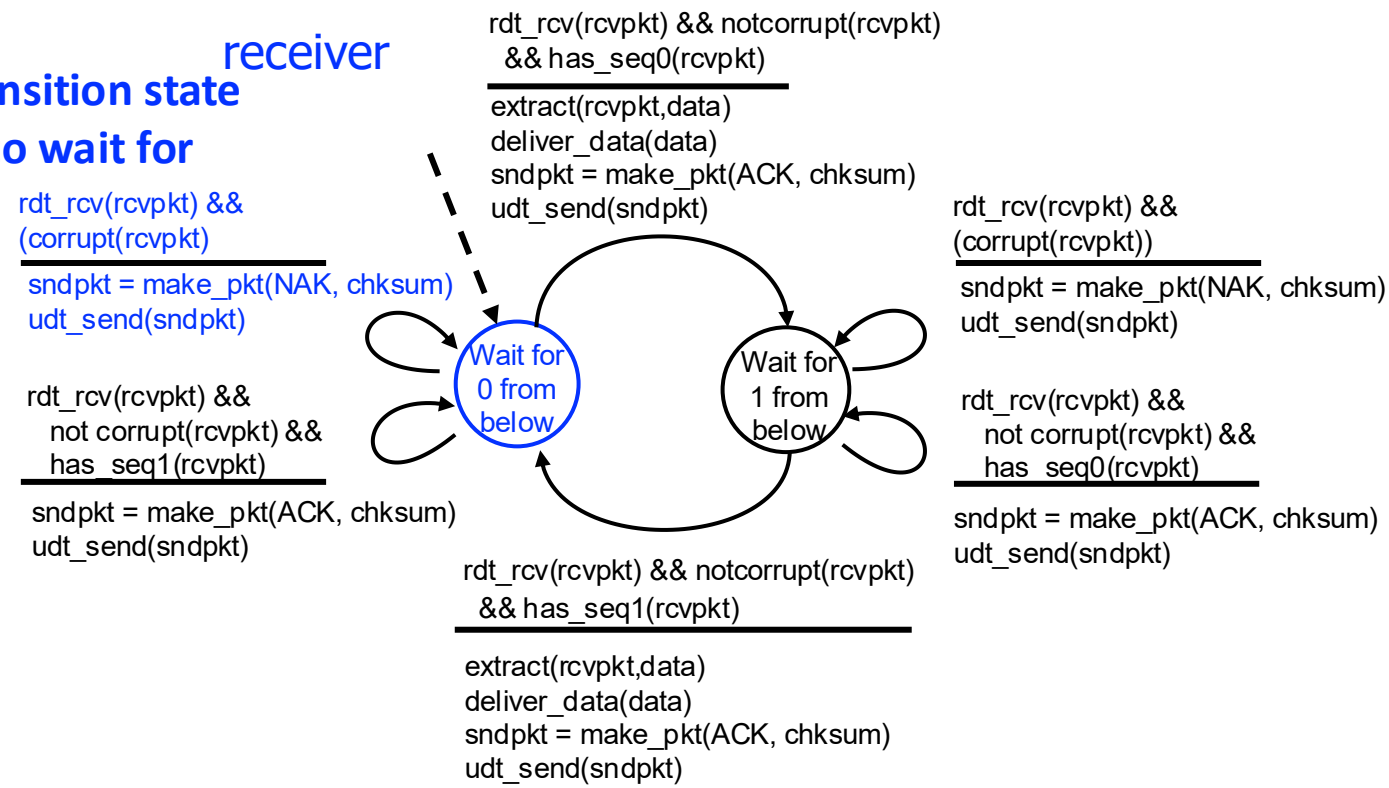
Transition state to wait for packet with seq=1

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: receiver, handling garbled ACK/NAKs

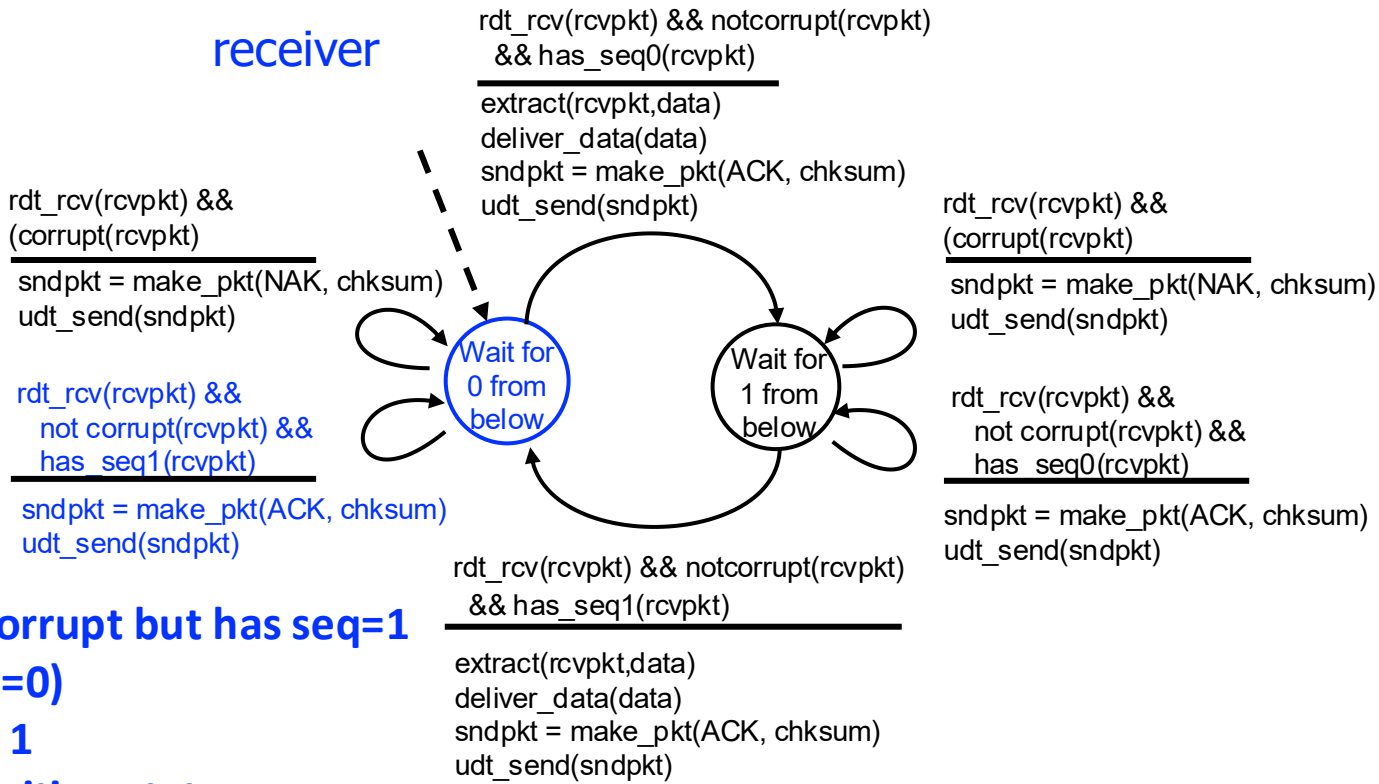
If packet was corrupt

- **Send NAK**
- **Do not transition state**
- **Continue to wait for seq=0**



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: receiver, handling garbled ACK/NAKs

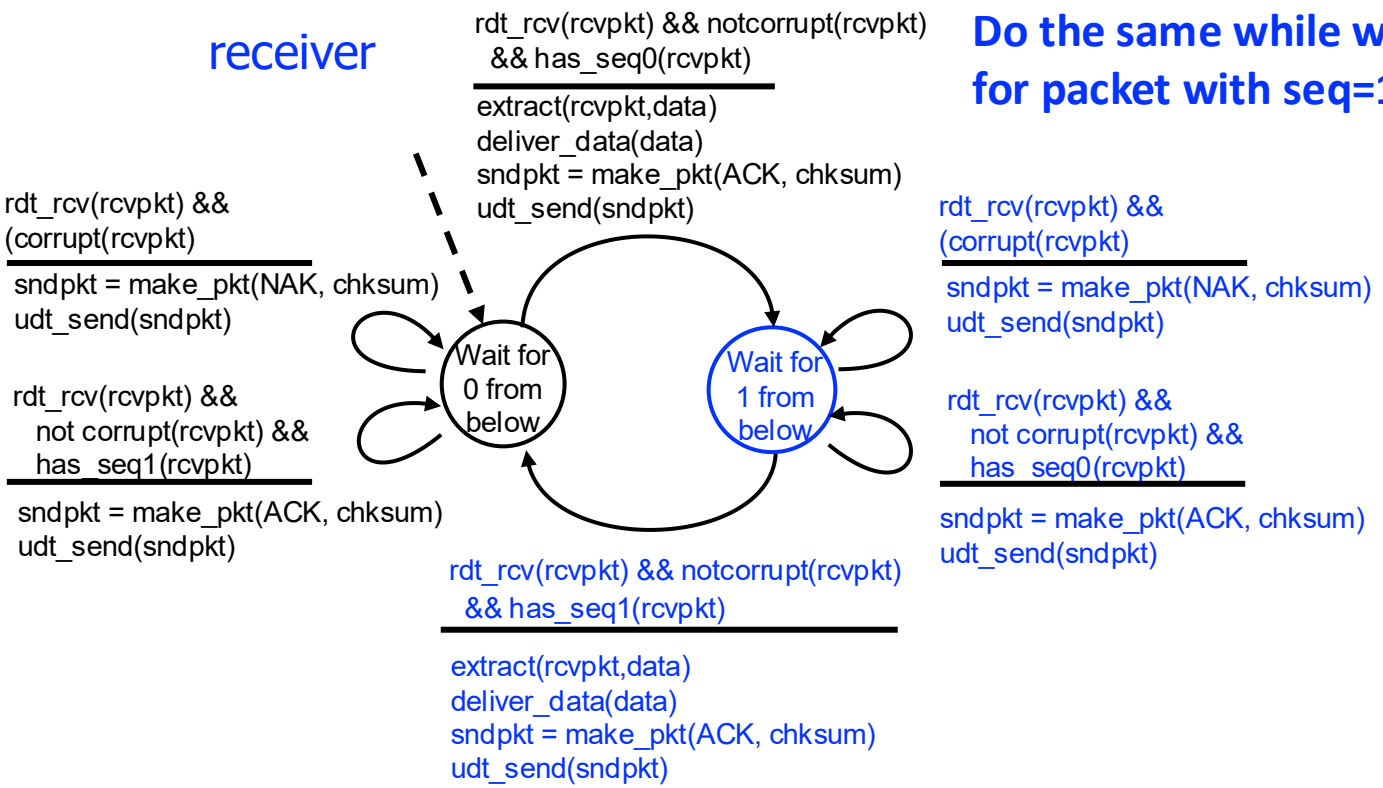


If packet not corrupt but has seq=1 (expecting seq=0)

- **ACK packet 1**
- **Do not transition state**
- **Continue to wait for seq=0**

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: receiver, handling garbled ACK/NAKs



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.1: discussion

sender:

- Seq # added to pkt
- Two seq. #s (0,1) will suffice Why?
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must "remember" whether "expected" pkt should have seq # of 0 or 1

receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected pkt seq #
- Note: receiver can *not* know if its last ACK/NAK was successfully received at sender

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

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 *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

How can sender get two ACKs with same sequence number?

- **First packet sent to receiver and ACK received by sender**
- **Second packet sent, but garbled at receiver**
- **Receiver sends ACK with previous sequence number**
- **Sender receives the same sequence number twice in a row**
- **Knows the receiver did not receive the second packet, resends**

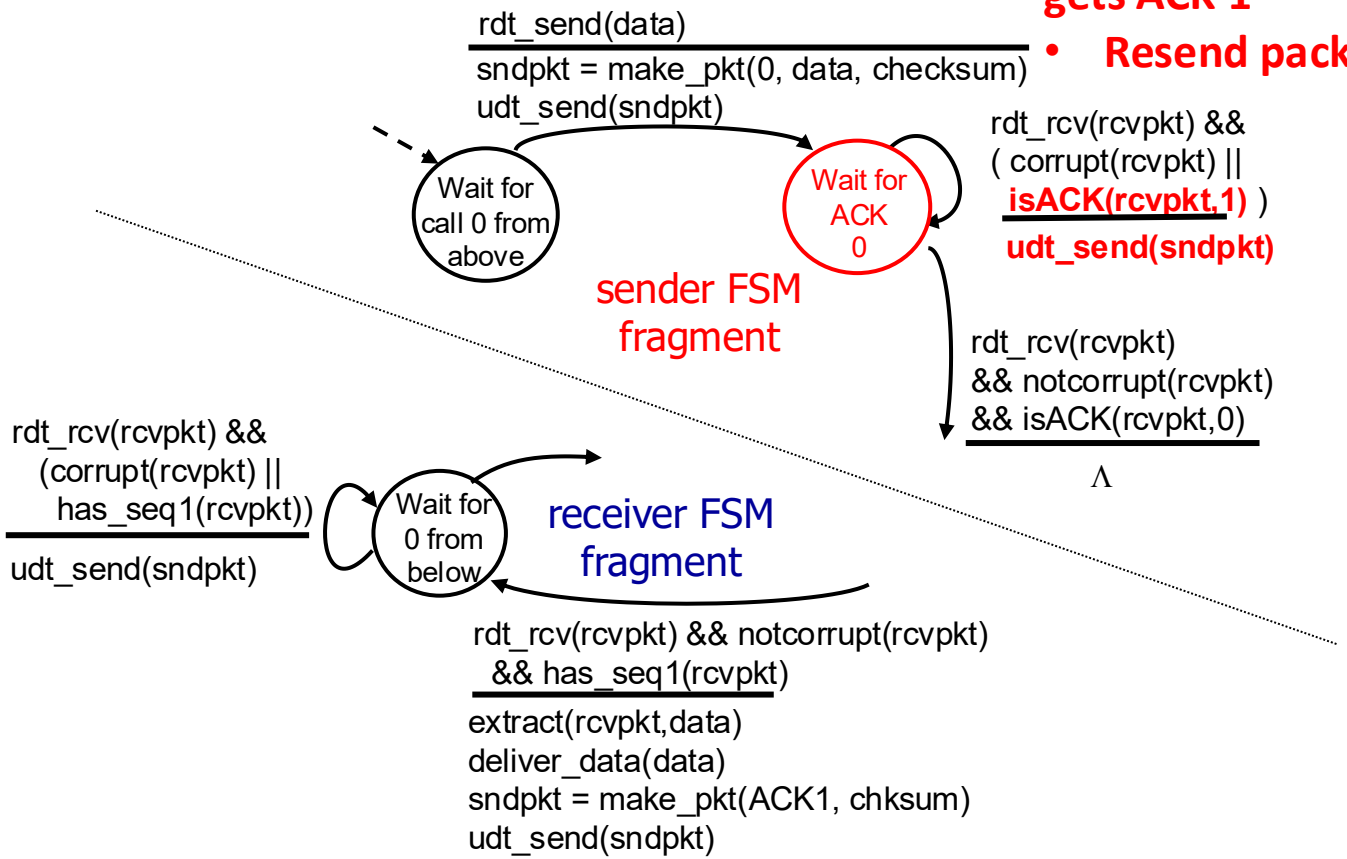
As we will see, TCP uses this approach to be NAK-free

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.2: sender fragment

If sender waiting for ACK 0 but gets ACK 1

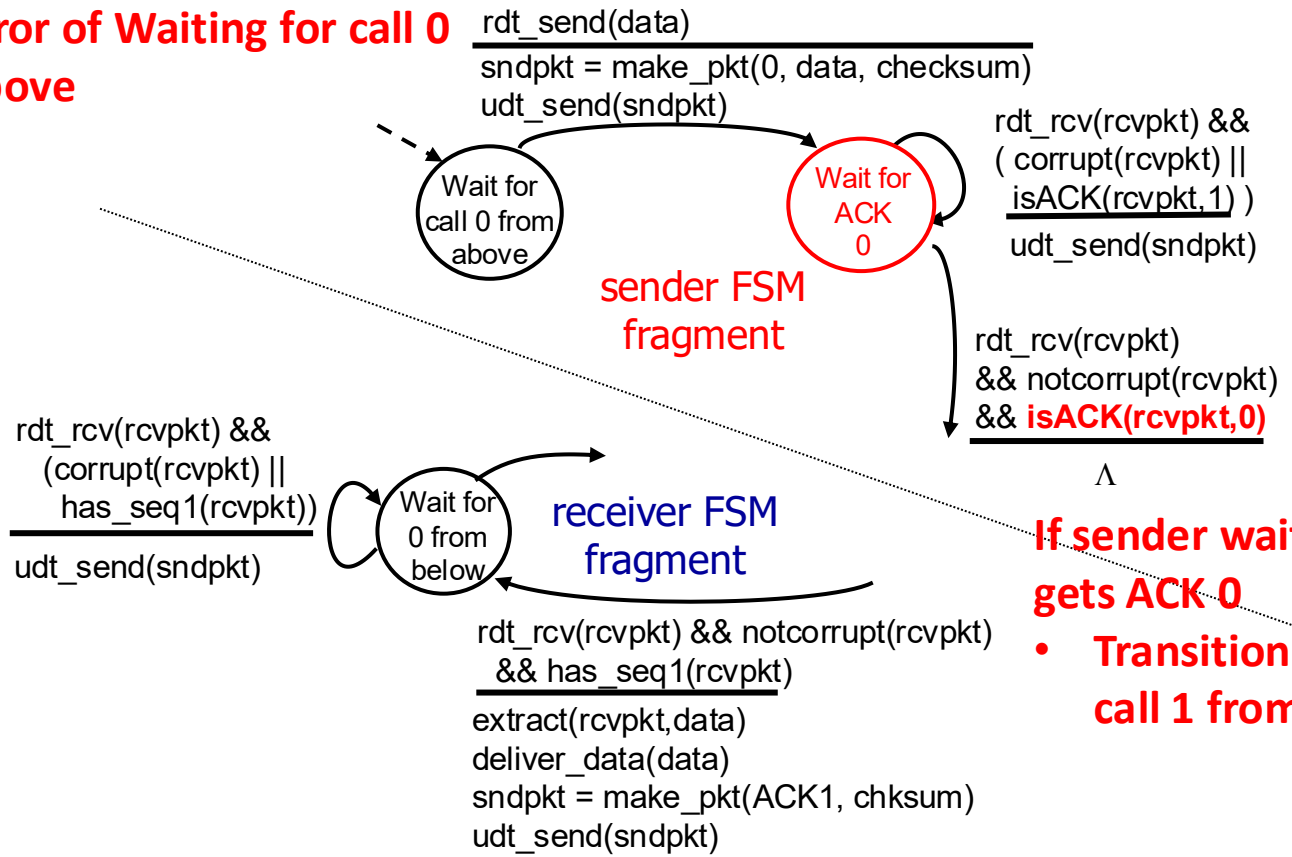
- **Resend packet 0**



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.2: sender fragment

Waiting for call 1 from above is the mirror of Waiting for call 0 from above

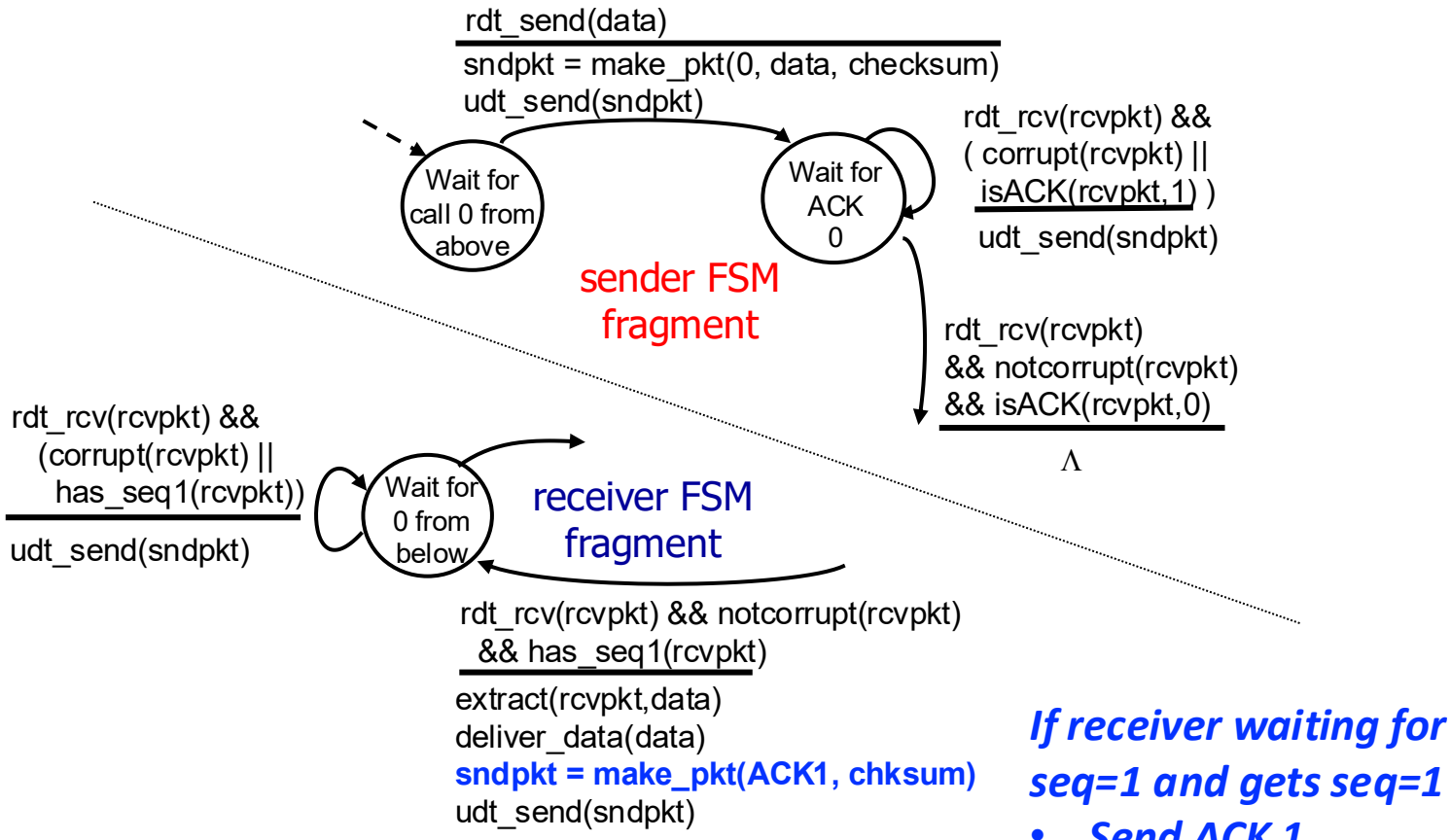


If sender waiting for ACK 0 and gets ACK 0

- **Transition to Waiting for call 1 from above**

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.2: receiver fragment

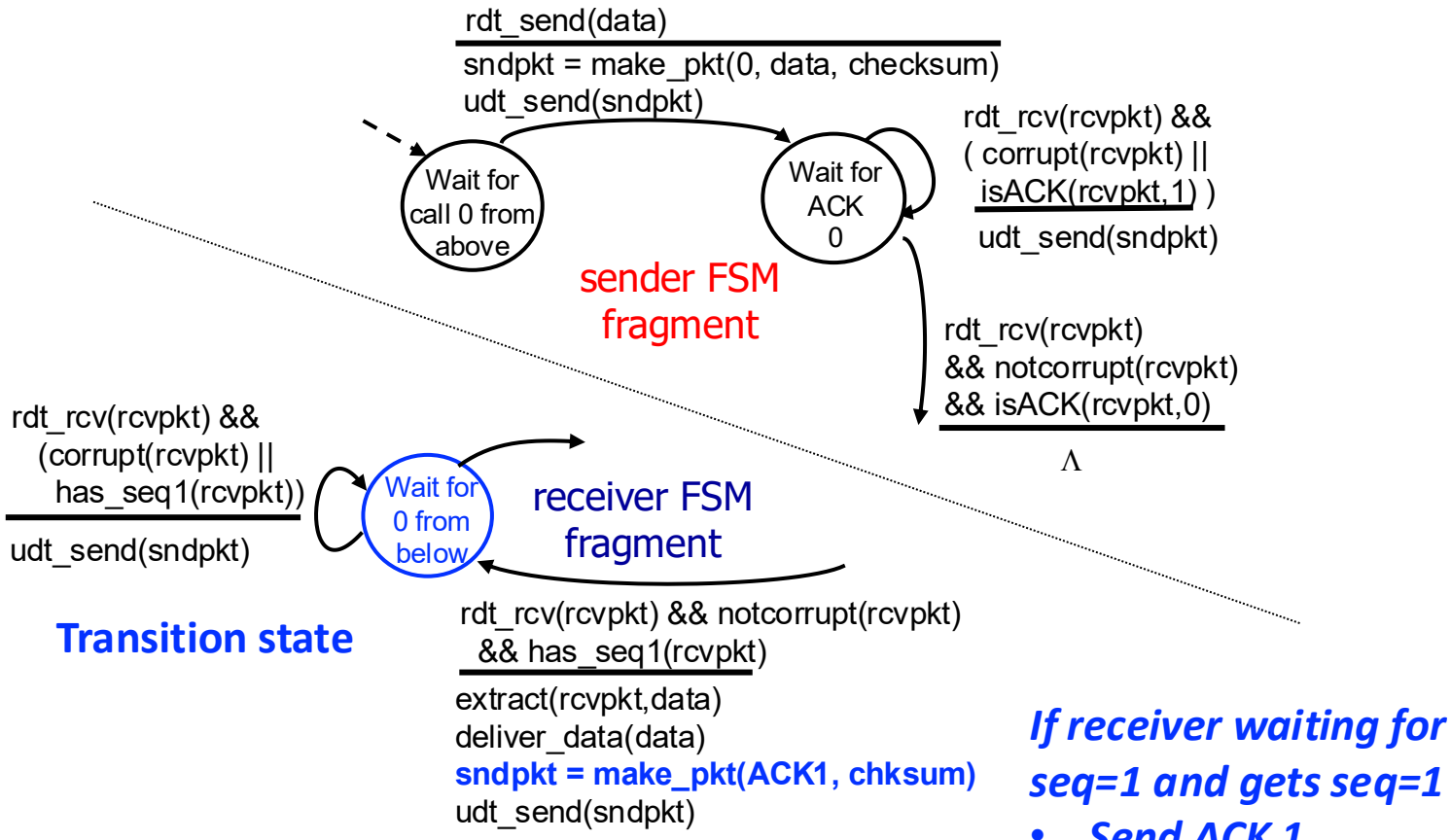


If receiver waiting for seq=1 and gets seq=1

- ***Send ACK 1***

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.2: receiver fragment

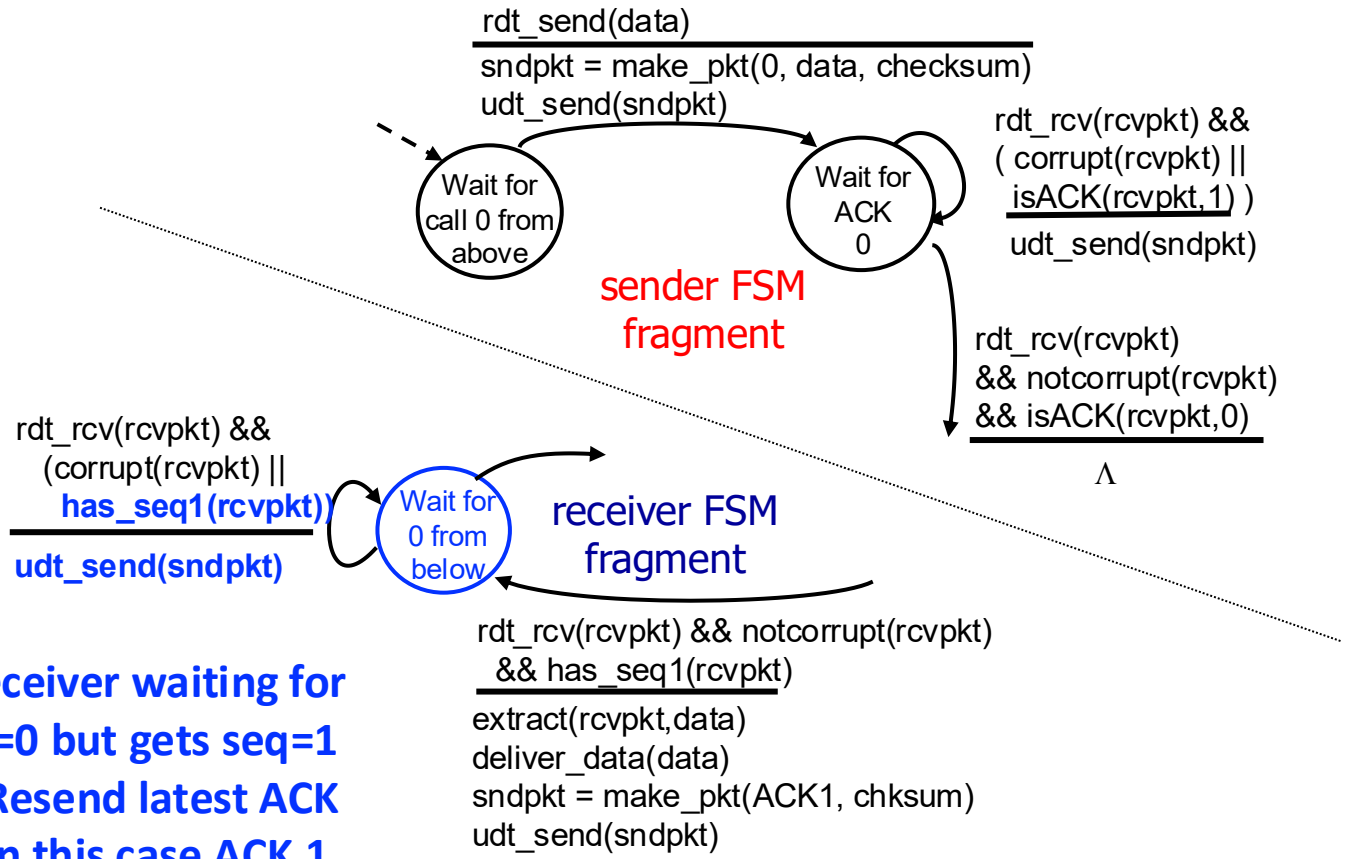


If receiver waiting for seq=1 and gets seq=1

- ***Send ACK 1***

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.2: receiver fragment

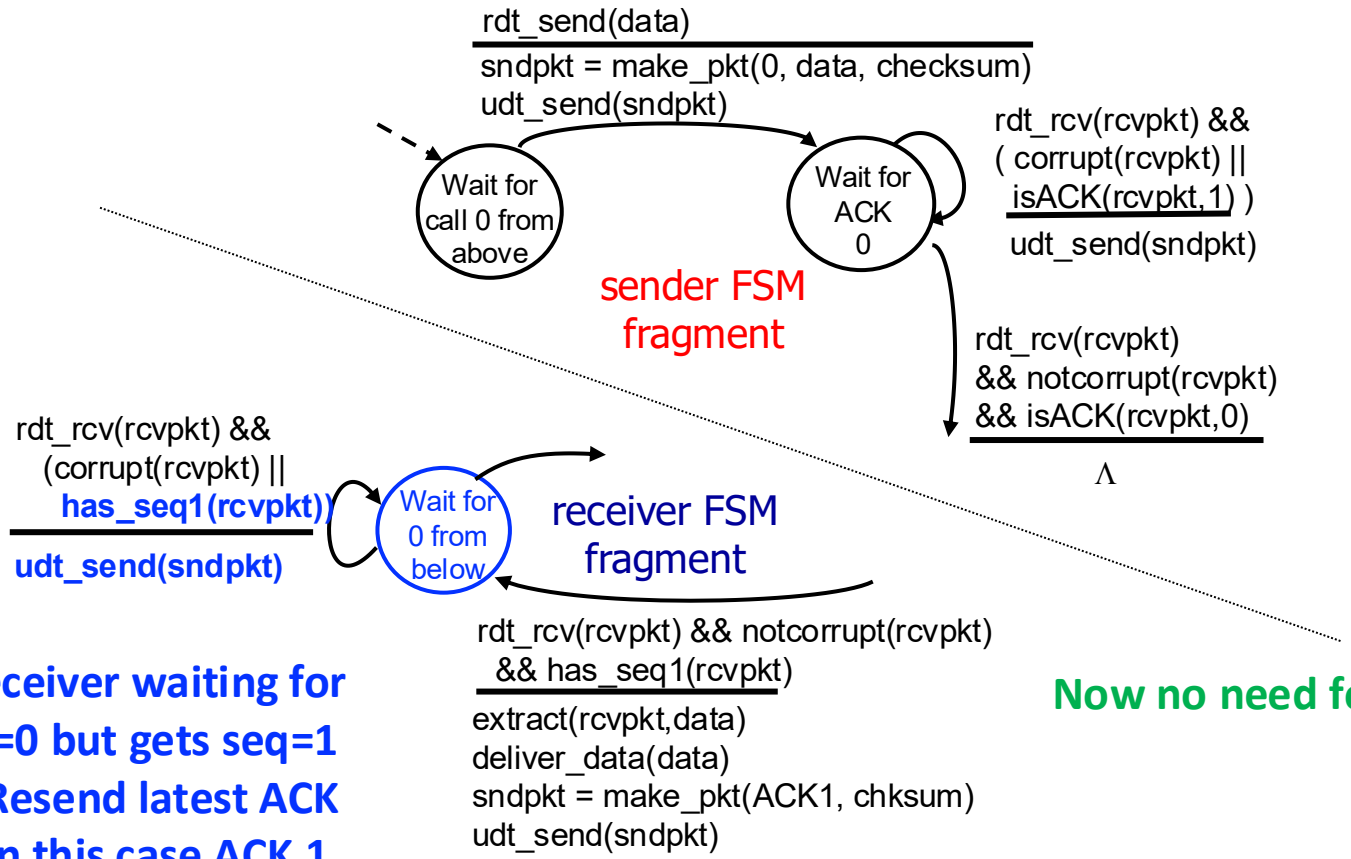


If receiver waiting for seq=0 but gets seq=1

- **Resend latest ACK**
- **In this case ACK 1**
- **Not ACK 0**

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt2.2: receiver fragment



If receiver waiting for seq=0 but gets seq=1

- **Resend latest ACK**
- **In this case ACK 1**
- **Not ACK 0**

Now no need for NAK

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt3.0: channels with errors *and* loss

New channel assumption: underlying channel can also *lose* or *delay* packets (data, ACKs)

- Checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

Two new problems

1. **How to detect missing packets (either data or ACK)**
2. **What to do about them**

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt3.0: channels with errors *and* loss

Approach: sender waits "reasonable" amount of time for ACK

- Retransmits packet 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

- Use countdown timer to interrupt after "reasonable" amount of time

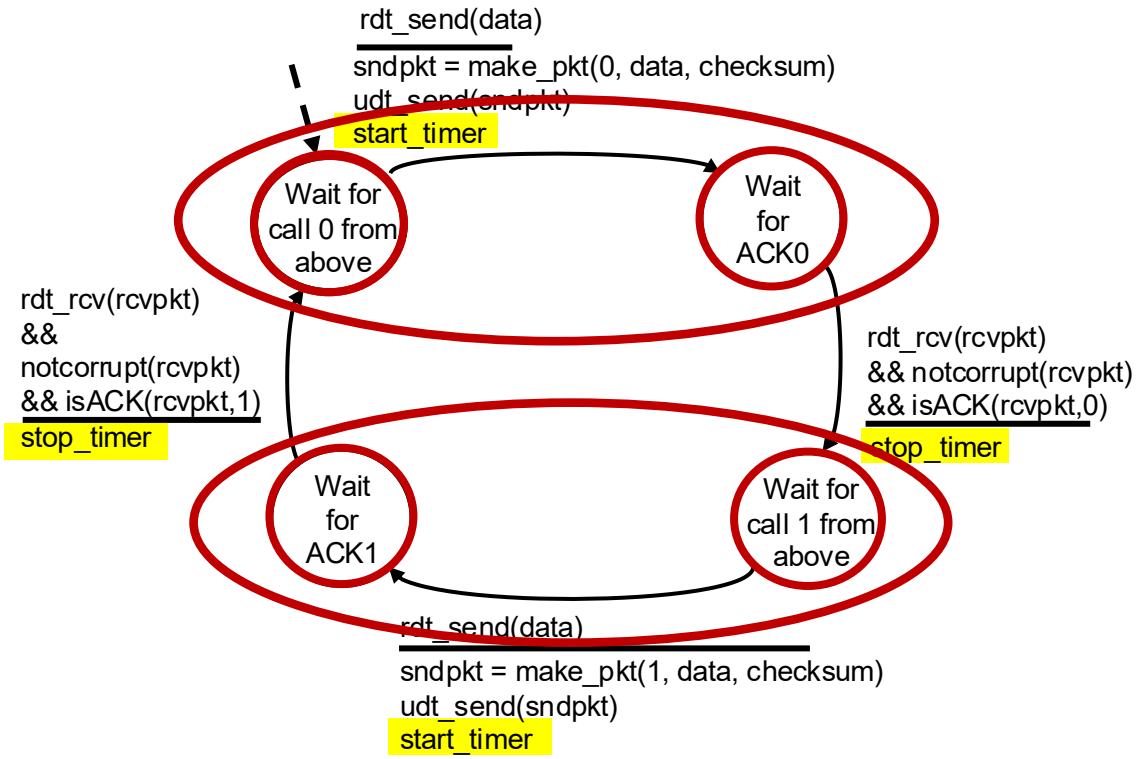
What is a "reasonable" amount of time?
At least round-trip time plus time receiver needs to process packet!



timeout

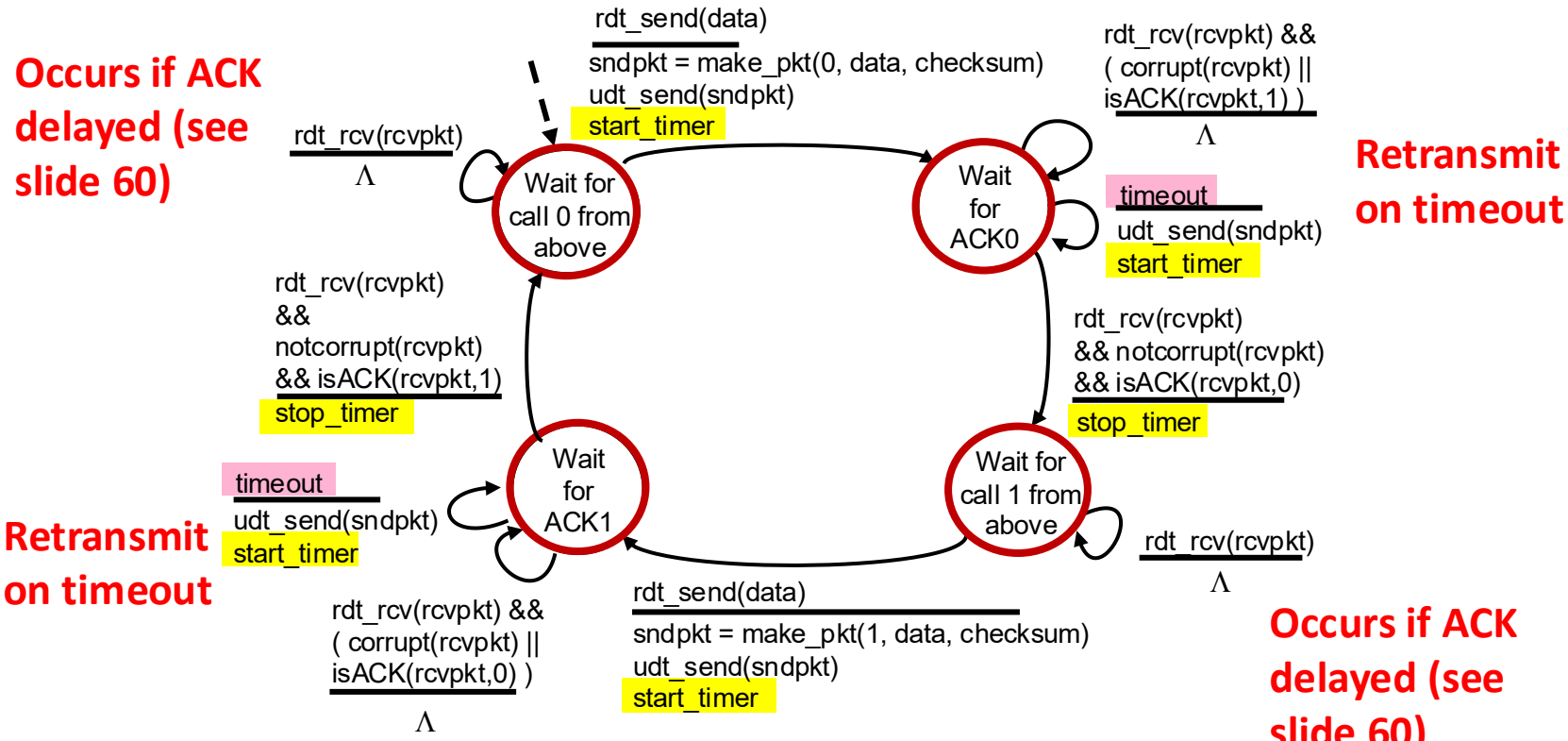
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt3.0 sender



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

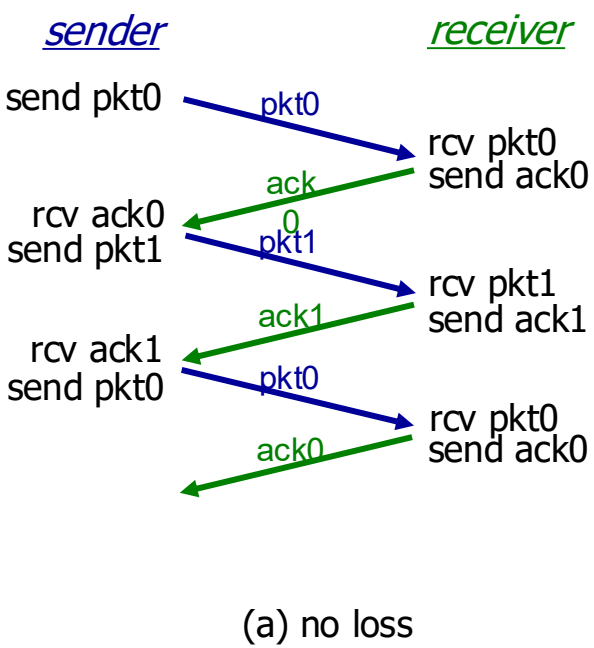
rdt3.0 sender



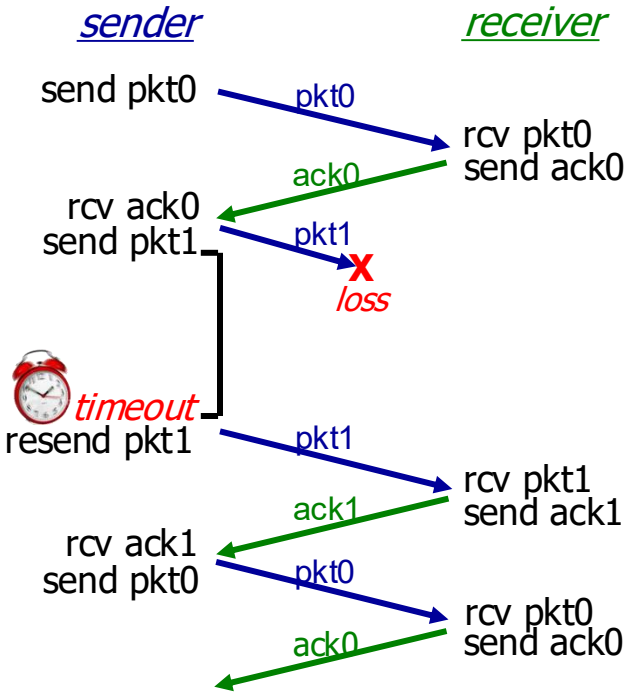
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt3.0 with no loss and with packet loss

Note: sequence numbers are still just 0 or 1



(a) no loss

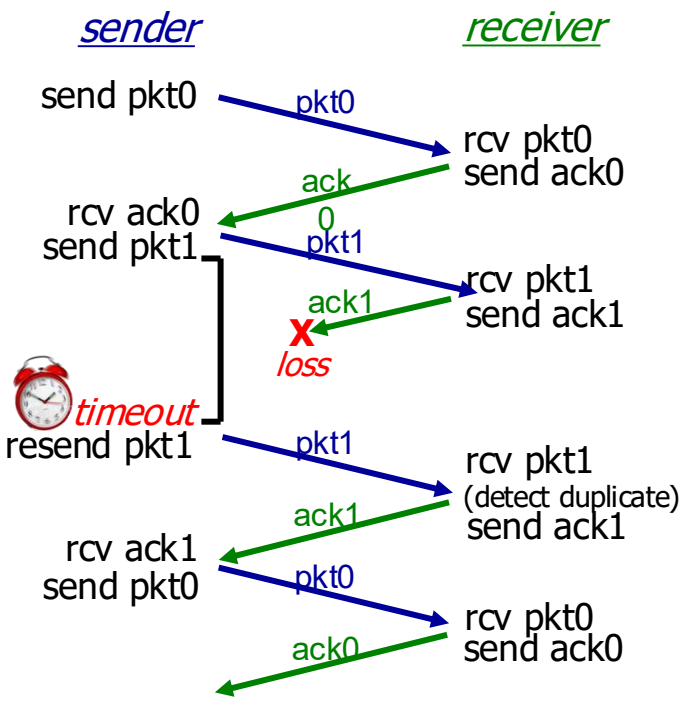


(b) packet loss

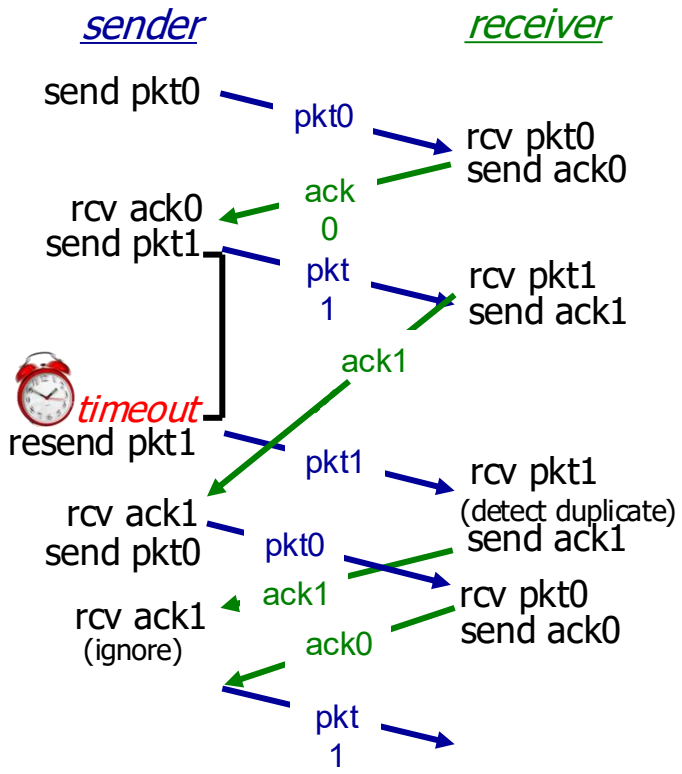
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt3.0 with ACK loss and delayed ACK

Note: sequence numbers are still just 0 or 1



(c) ACK loss



(d) premature timeout or delayed ACK

Duplicate packets at receiver

Duplicate packets at receiver

Duplicate ACKs received by sender

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

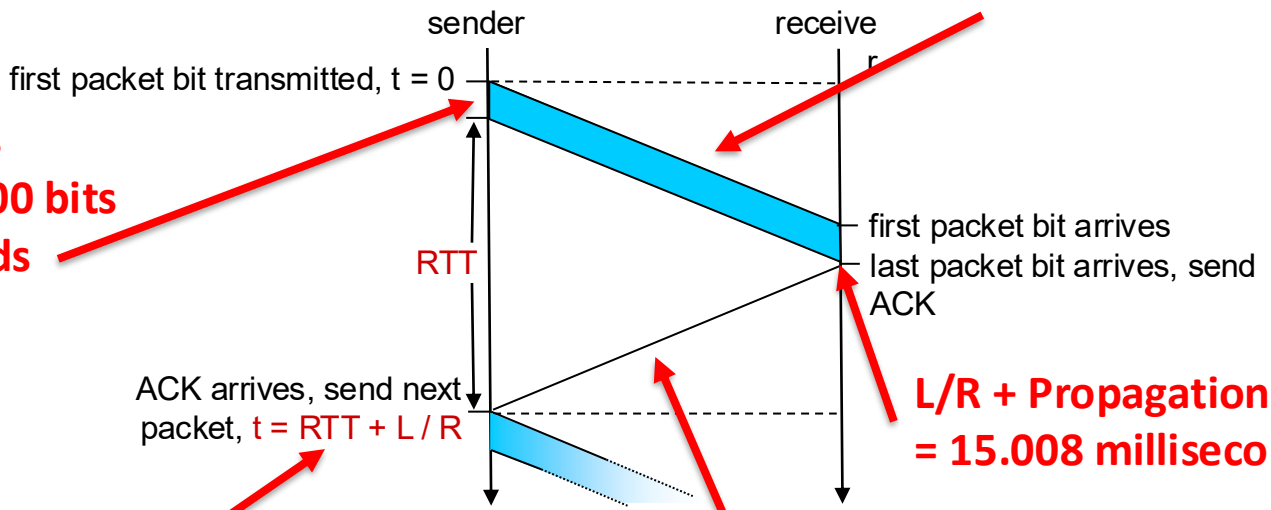
Performance of rdt3.0 with Stop-and-Wait operation

- U_{sender}: *utilization* – fraction of time sender busy sending
- Example: 1 Gbps link, 15 ms prop. delay, 8000-bit packets
  - Link rate R = 1 Gbps (=10^9 bits per second)
  - Packet length L = 1,000 bytes (=8,000 bits)
  - Propagation delay = 15 milliseconds
  - Time to transmit packet into channel:

$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs\ to\ send\ 1\ packet$$

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Performance of rdt3.0 with Stop-and-Wait operation



Propagation delay = 15 milliseconds

Link rate R = 1 Gbps
Packet length = 8,000 bits
L/R = 8 microseconds

L/R + Propagation = 15.008 milliseconds

t = L/R + 2* Propagation = 30.008 milliseconds

Propagation Assuming receiver responds instantly!

Also assuming ACK is of negligible size

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

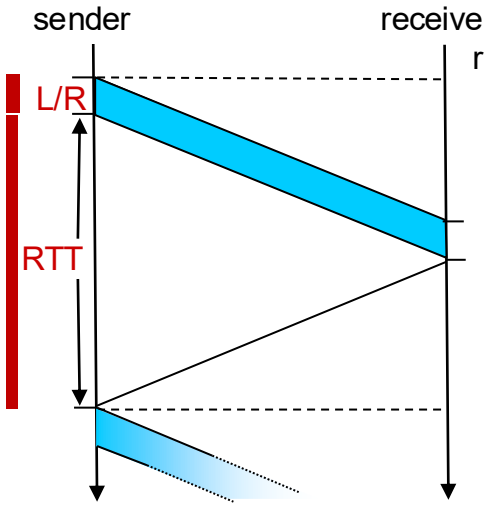
Performance of rdt3.0 with Stop-and-Wait operation

Utilization = fraction of time the sender is sending

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

$$= \frac{.008}{30.008}$$

$$= 0.00027$$



- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

Effective rate is 267 kbps, even though we have a 1 Gbps link!
(ignoring other layer delays and processing time)

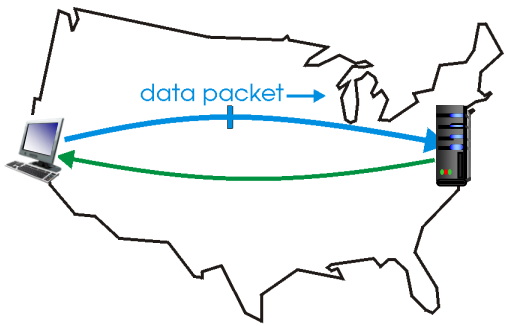
If RTT is such a big factor, what can we do?
Pipeline

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

rdt3.0: pipelined protocols operation

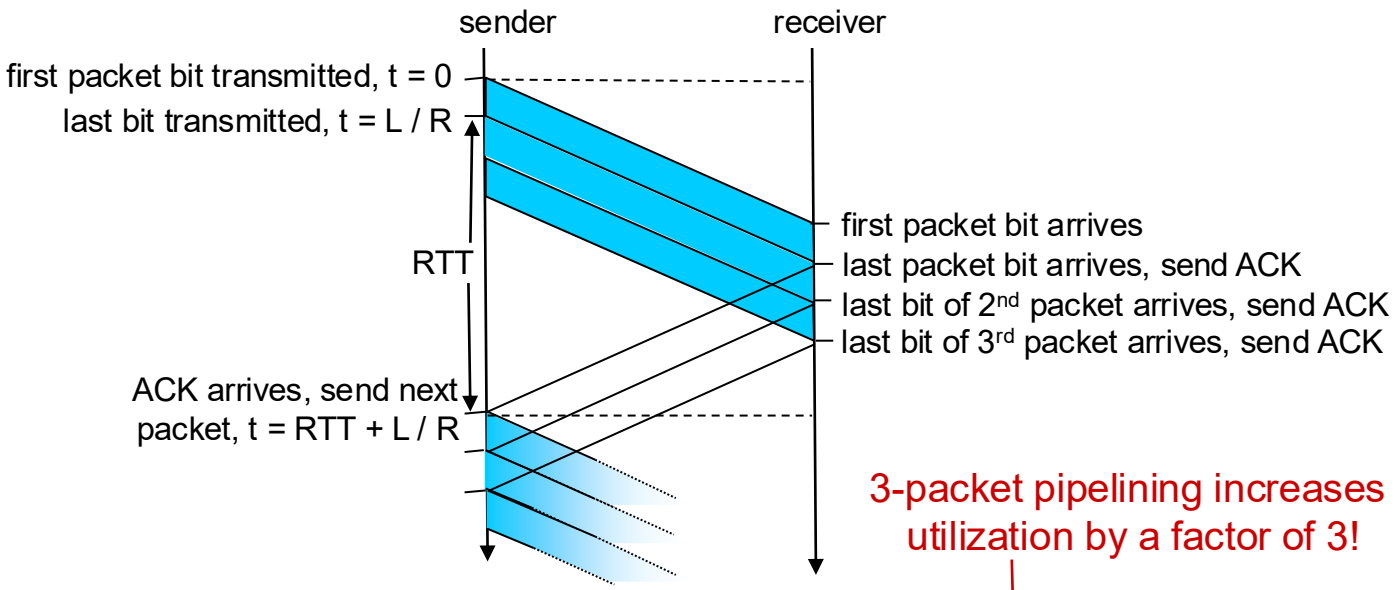
Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- Range of sequence numbers must be increased (not just 0/1)
- Buffering at sender and/or receiver



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

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

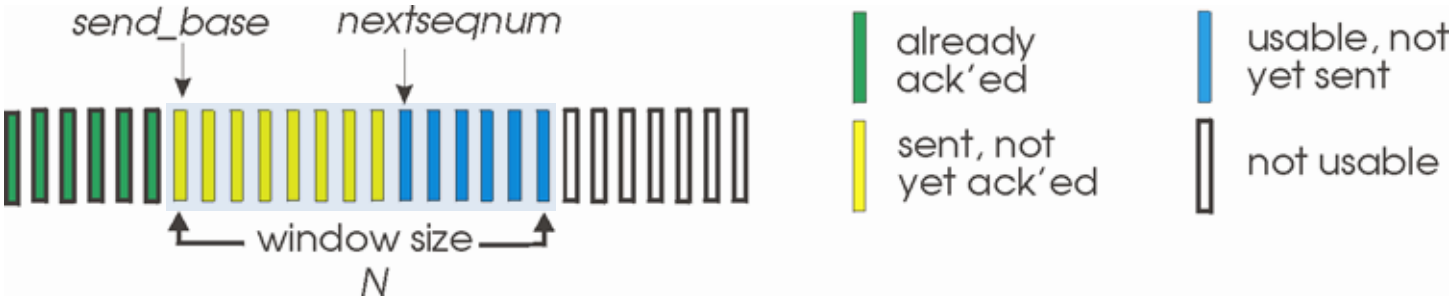
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$ (was 0.00027)

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Go-Back-N: sender

Also known as GBN (Go-Back-N)

- Sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - Use k-bit seq # in pkt header



- *Cumulative ACK:* ACK(*n*): receiver ACKs all packets up to, including seq # *n*
  - On receiving ACK(*n*): sender moves window forward to begin at *n+1*
- Timer for oldest in-flight packet
- If *Timeout(i):* retransmit packet *i* and all higher seq # packets in window

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Go-Back-N: receiver

Also known as GBN (Go-Back-N)

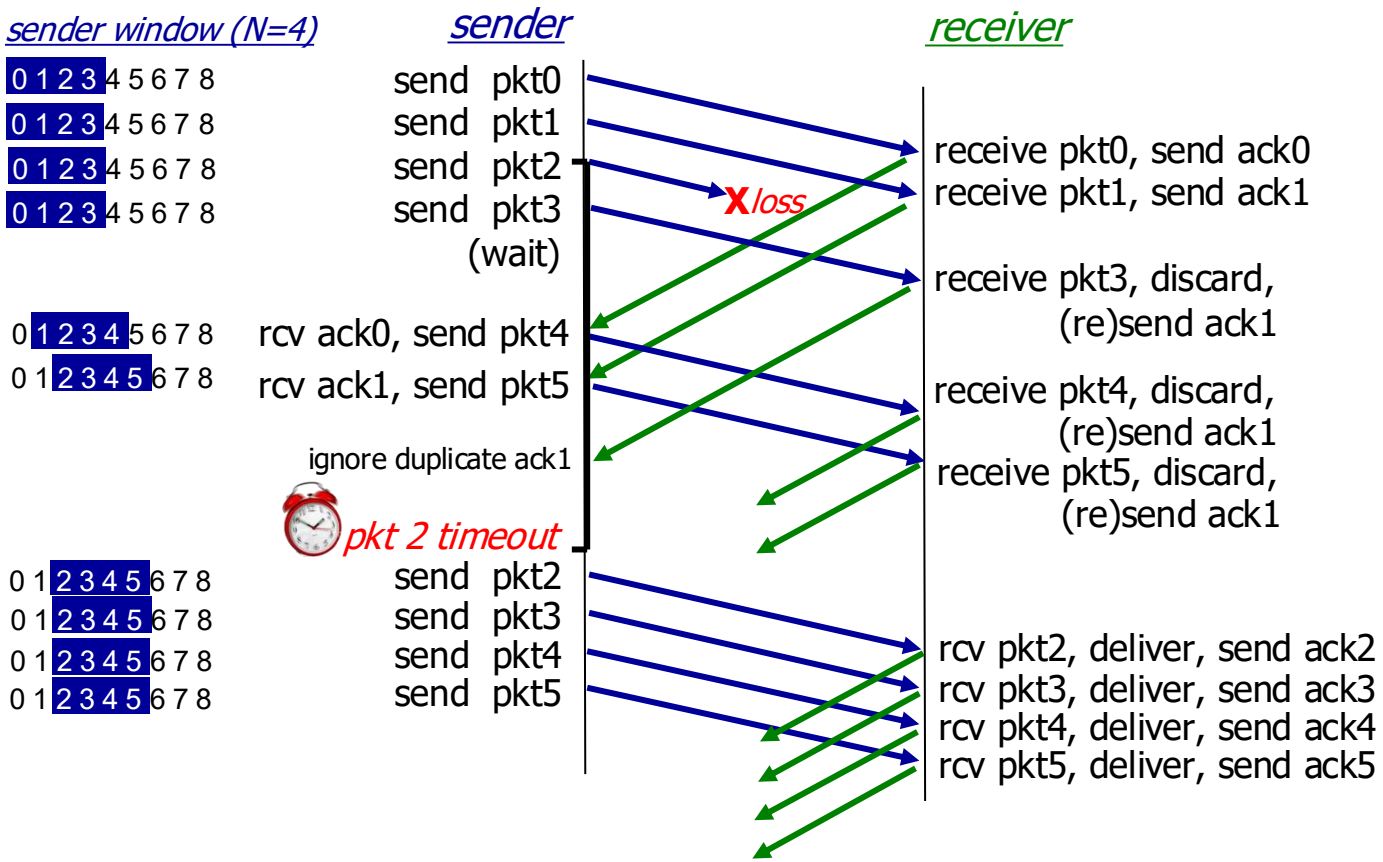
- ACK-only: receiver always sends ACK for correctly-received packet so far, with highest *in-order* seq #
  - May generate duplicate ACKs
  - Need only remember rcv_base
- On receipt of out-of-order packet:
  - Can discard (don't buffer) or buffer: an implementation decision
  - Re-ACK pkt with highest in-order seq #



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Go-Back-N in action

Also known as GBN (Go-Back-N)



GBN can cause lots of retransmissions if N is large and delay is large

***Selective repeat* is an alternative approach**

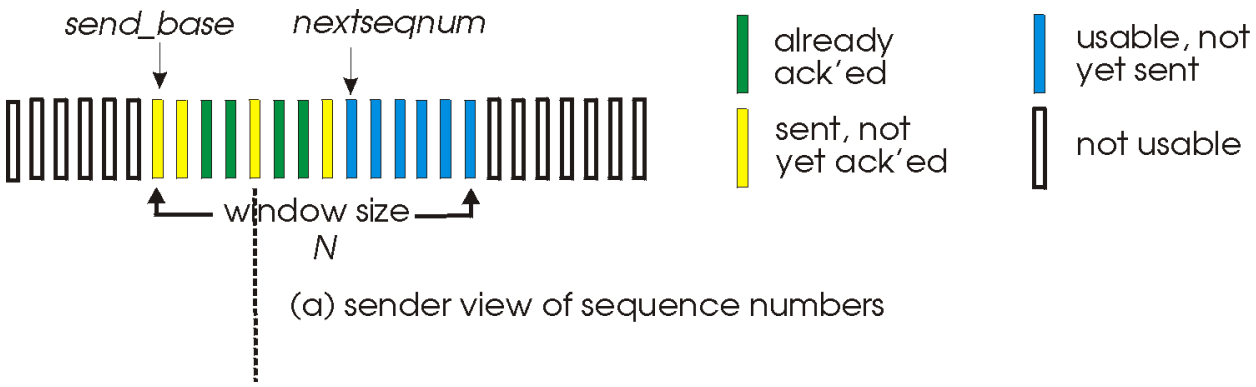
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Selective repeat

- Receiver *individually* acknowledges all correctly received packets
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender times-out/retransmits individually for unACKed packets
  - Sender maintains timer for each unACKed pkt
- Sender window
  - *N* consecutive seq #s
  - Limits seq #s of sent, unACKed packets

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Selective repeat: sender, receiver windows



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Selective repeat: sender and receiver

Sender

Data from above:

- If next available seq # in window, send packet, else wait

Timeout(*i*):

- Resend packet *i*, restart timer

ACK(*i*) in [sendbase,sendbase+N-1]:

- Mark packet *i* as received
- If *i* smallest unACKed packet, advance window base to next unACKed seq #

Receiver

Packet *i* in [rcvbase, rcvbase+N-1]

- Send ACK(*i*)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

Packet *i* in [rcvbase-N,rcvbase-1]

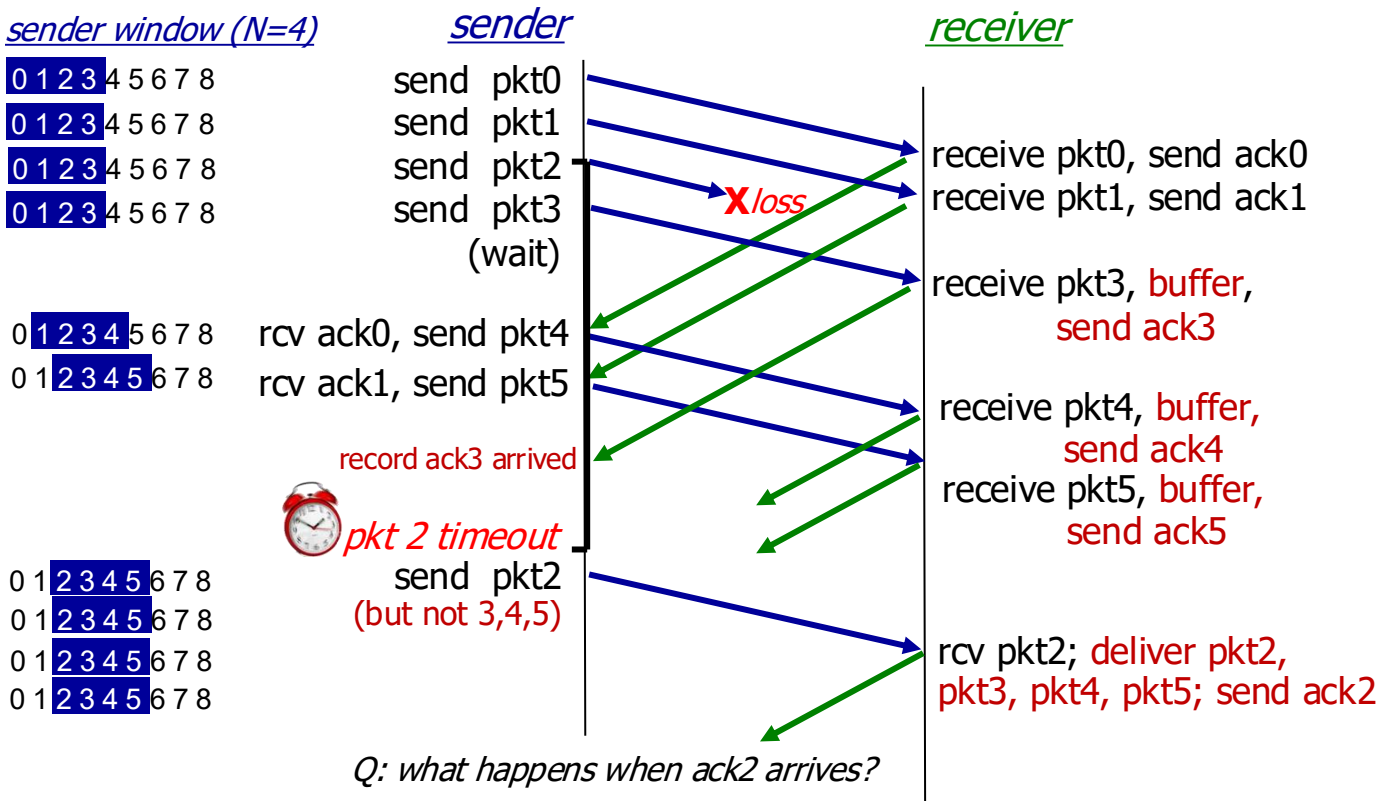
- ACK(*i*)

Otherwise:

- ignore

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Selective Repeat in action



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Principles of congestion control

Principles of congestion control

Congestion:

- Informally: "too many sources sending too much data too fast for *network* to handle"
- Manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- Different from flow control!
- A top-10 problem!





congestion control: too many senders, sending too fast

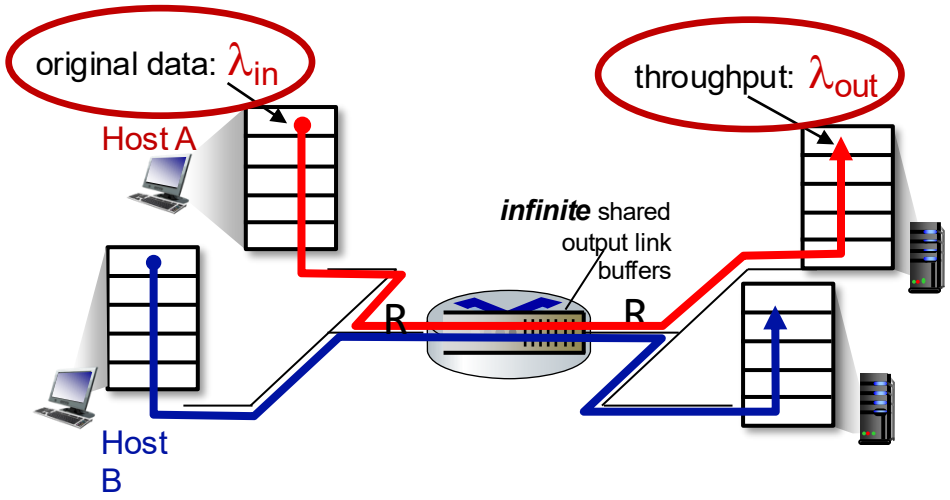
flow control: one sender too fast for one receiver

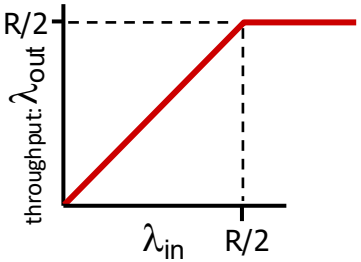
Causes/costs of congestion: scenario 1

Simplest scenario:

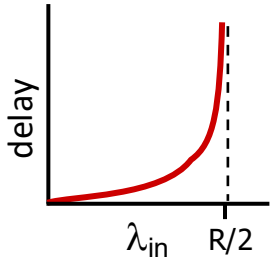
- One router, infinite buffers
- Input, output link capacity: R
- Two flows: A and B
- No retransmissions needed

Q: What happens as arrival rate λ_{in} approaches R/2?





maximum per-connection throughput: R/2

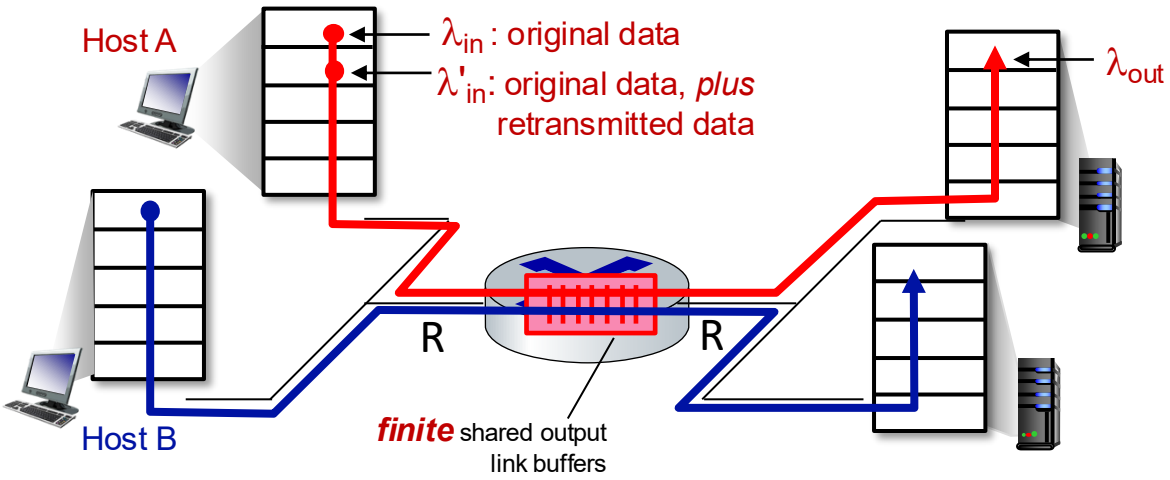


large delays as arrival rate λ_{in} approaches capacity

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: scenario 2

- One router, *finite* buffers
- Sender retransmits lost, timed-out packet
  - Application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - Transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

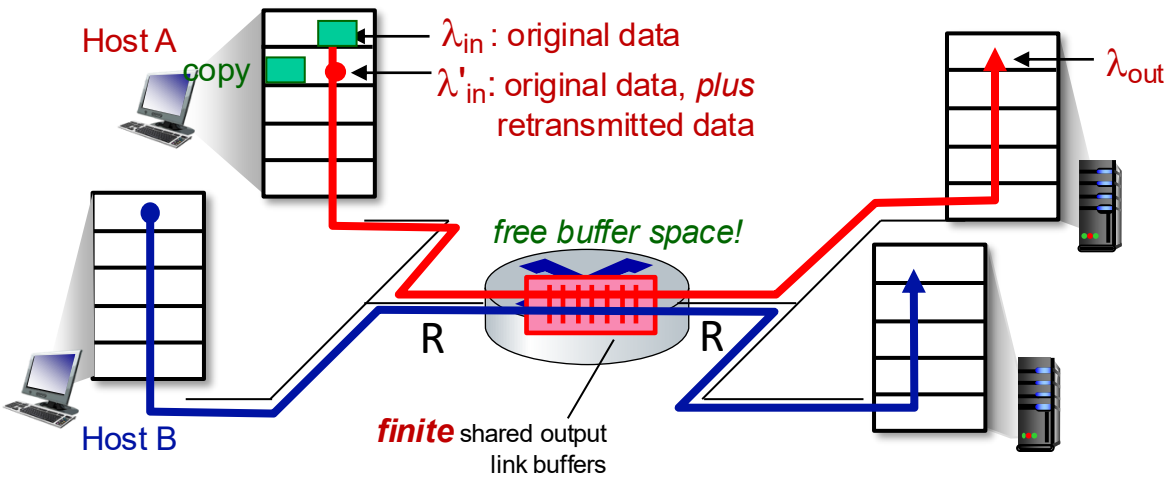


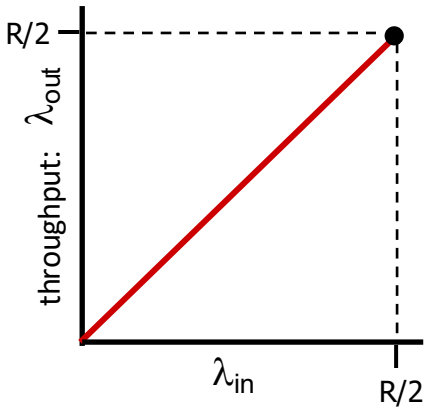
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- Sender sends only when router buffers available



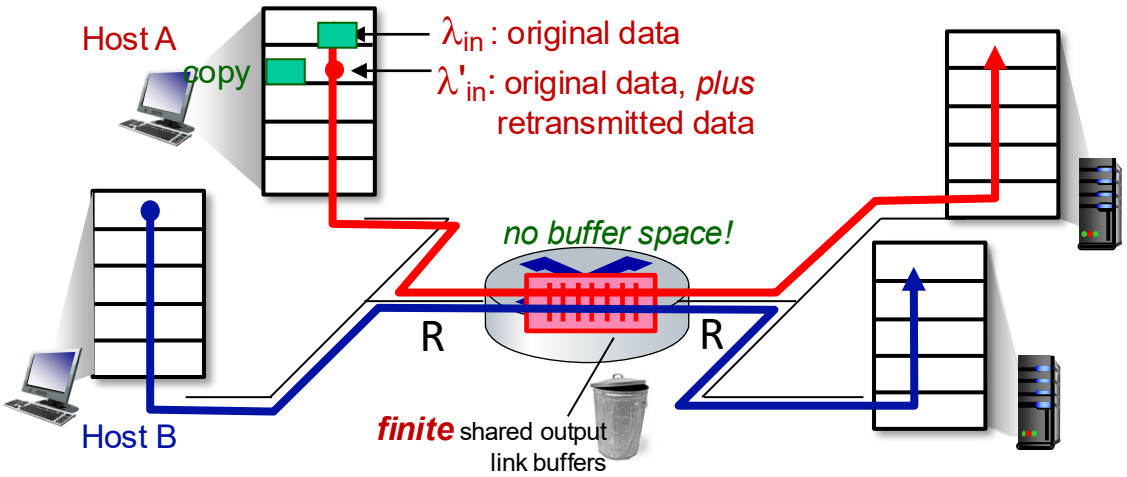


Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- Packets can be lost (dropped at router) due to full buffers
- Sender knows when packet has been dropped: only resends if packet *known* to be lost

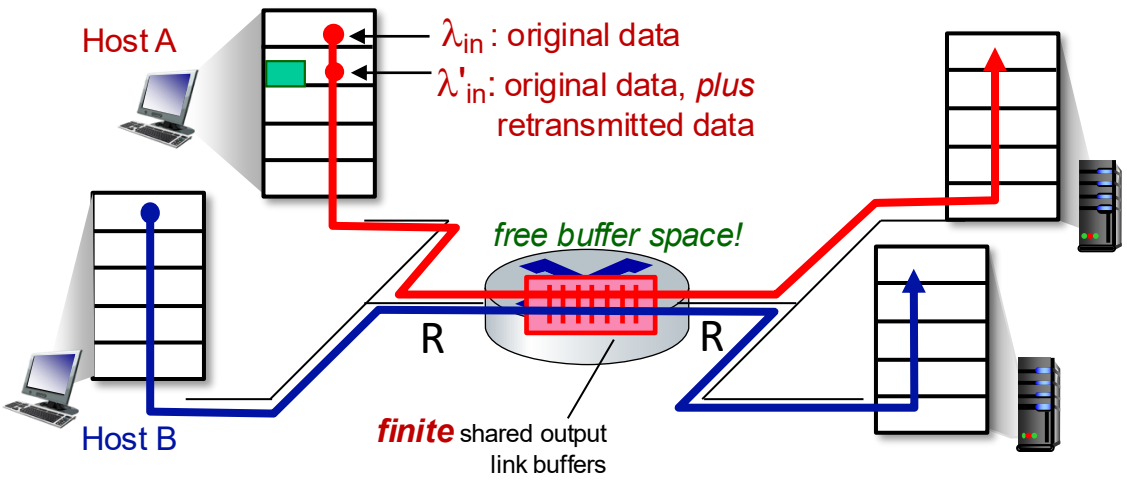


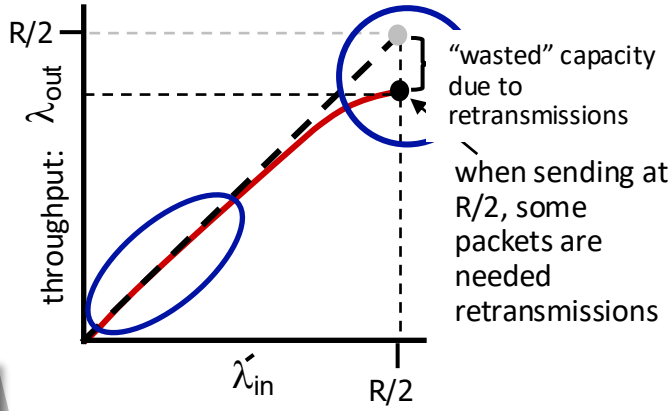
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- Packets can be lost (dropped at router) due to full buffers
- Sender knows when packet has been dropped: only resends if packet *known* to be lost



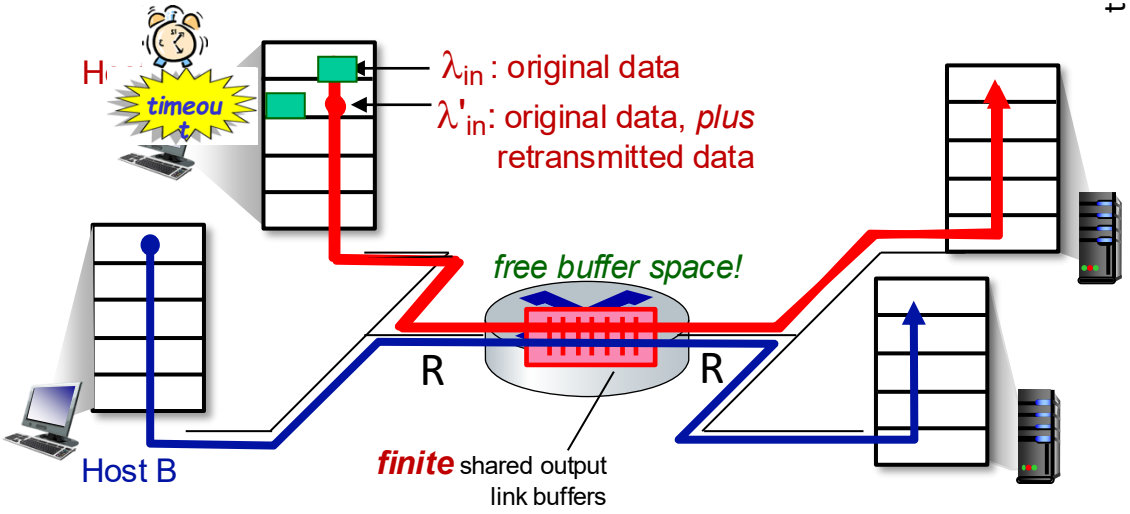


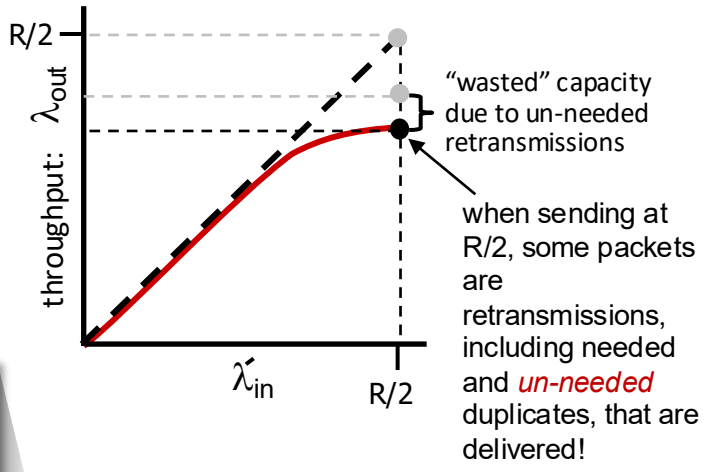
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- Packets can be lost, dropped at router due to full buffers – requiring retransmissions
- But sender times can time out prematurely, sending *two* copies, *both* of which are delivered



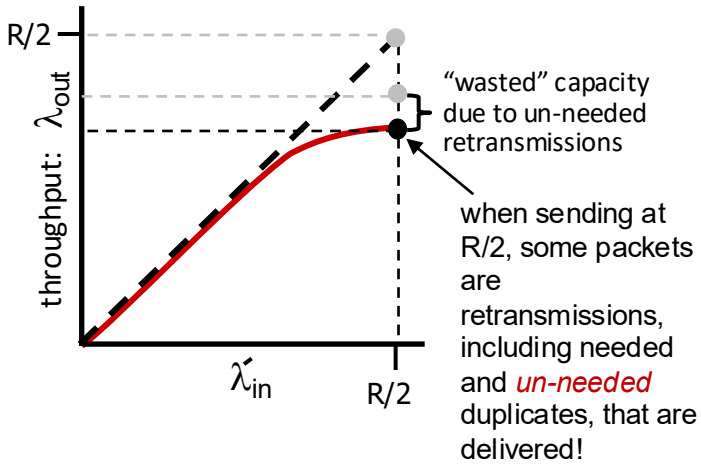


Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- Packets can be lost, dropped at router due to full buffers – requiring retransmissions
- But sender times can time out prematurely, sending *two* copies, *both* of which are delivered



"costs" of congestion:

- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

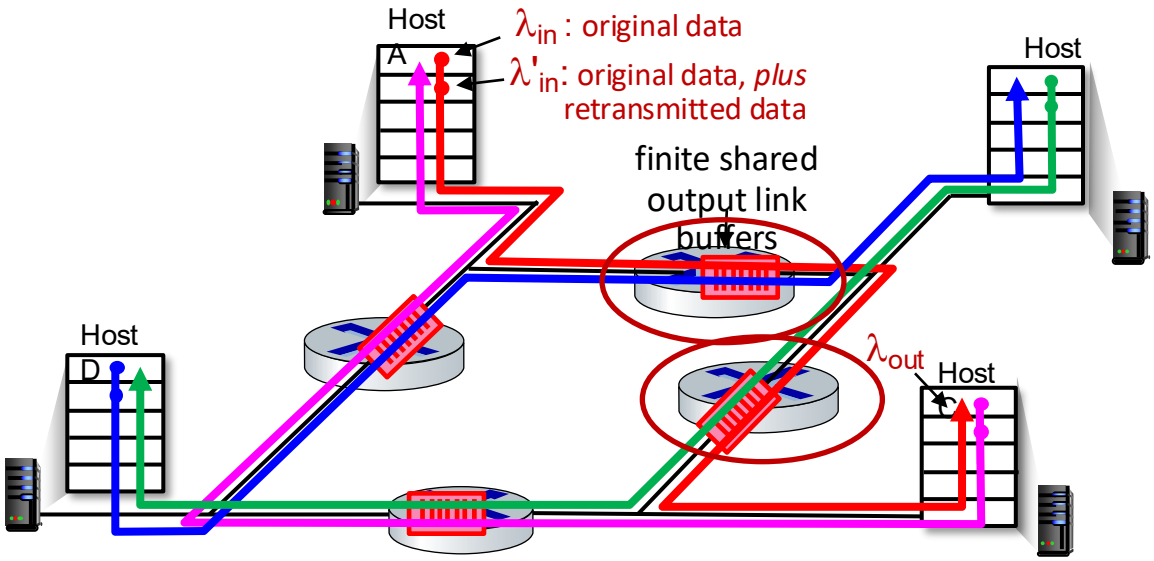
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: scenario 3

- *four* senders
- *multi-hop* paths
- timeout/retransmit

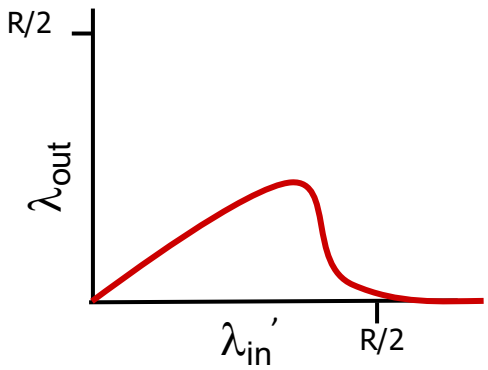
Q: what happens as λ_{in} and λ_{in}' increase ?

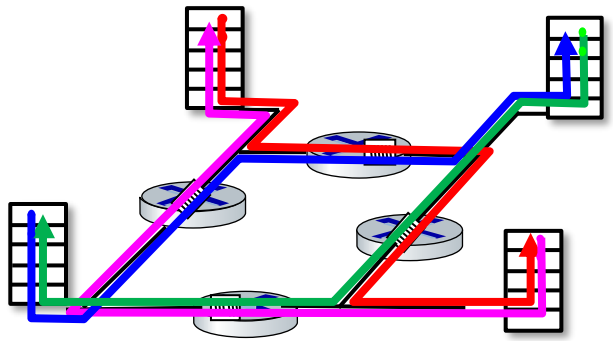
A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: scenario 3





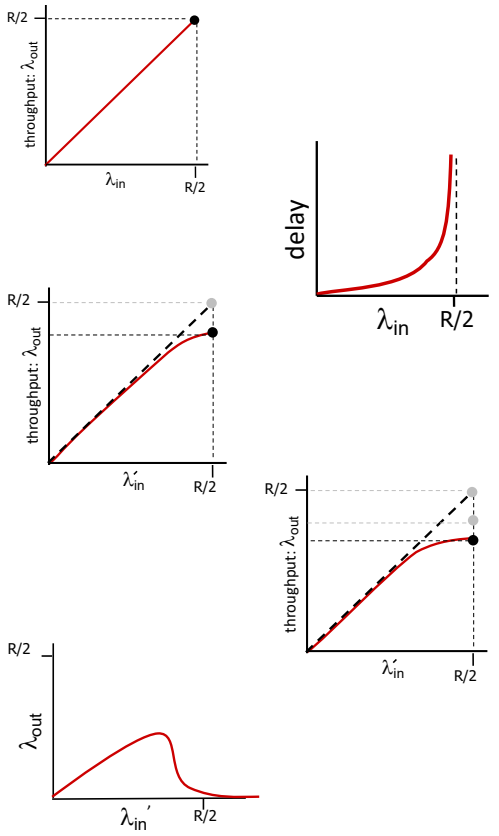
another "cost" of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Causes/costs of congestion: insights

- Throughput can never exceed capacity
- Delay increases as capacity approached
- Loss/retransmission decreases effective throughput
- Un-needed duplicates further decreases effective throughput
- Upstream transmission capacity / buffering wasted for packets lost downstream

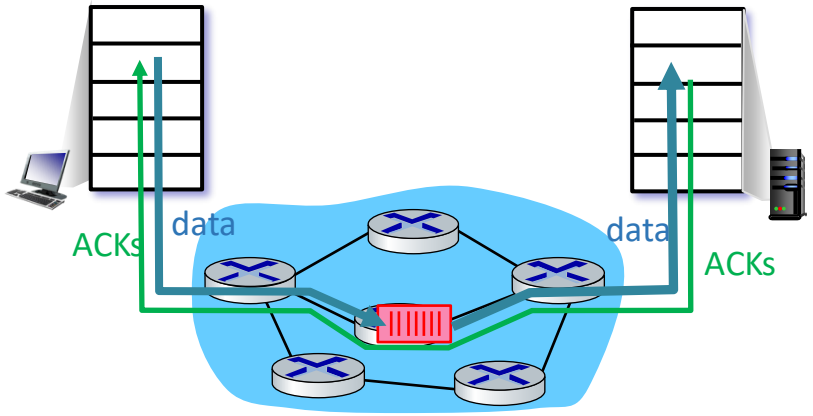


Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Approaches towards congestion control

End-end congestion control:

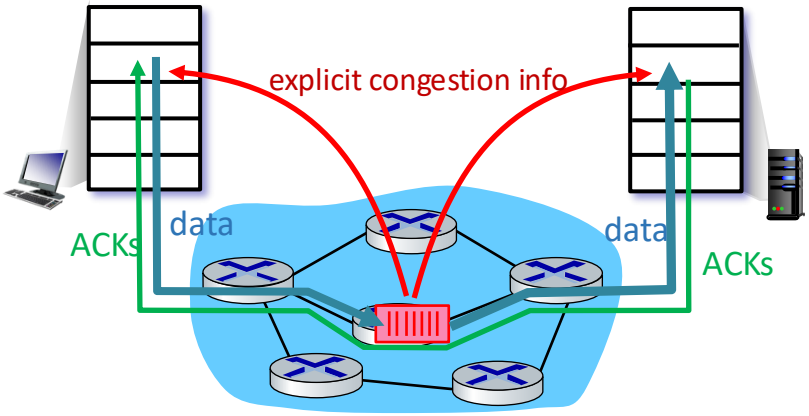
- No explicit feedback from network
- Congestion *inferred* from observed loss, delay
- Approach taken by TCP



Approaches towards congestion control

Network-assisted congestion control:

- Routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- May indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



TCP congestion control

TCP limits its transmission rate during congestion

Open questions

1. How does TCP limit its transmission rate

2. How does TCP perceive there is congestion

3. What algorithm to use to limit transmission?

It turns out there are several approaches used by hosts on the Internet

TCP congestion control: AIMD

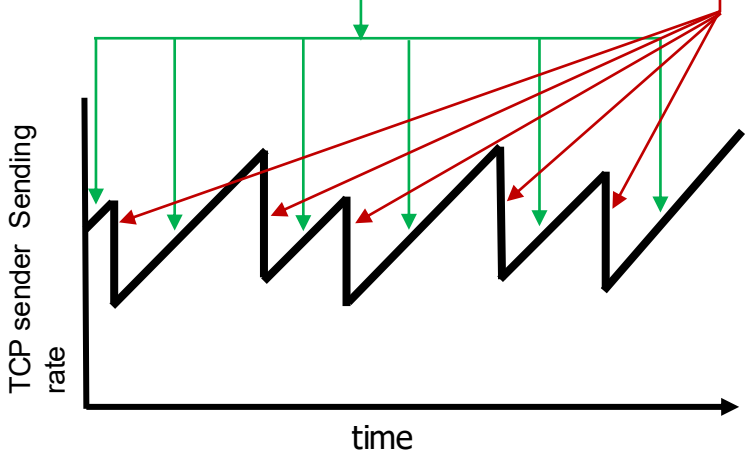
- *Approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP AIMD: more

Multiplicative decrease detail: sending rate is

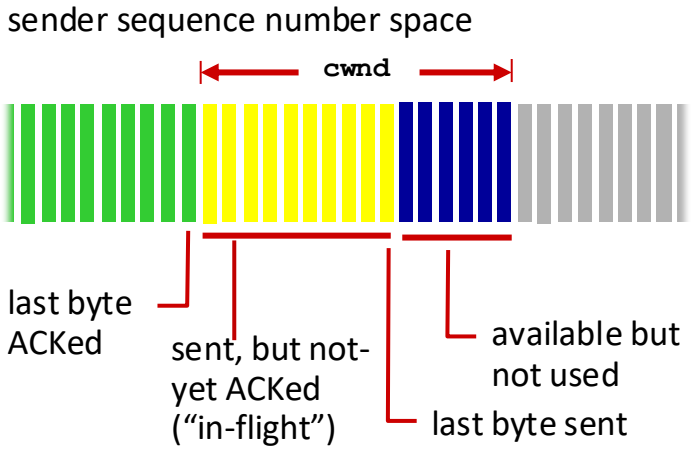
- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP congestion control: details



TCP sending behavior:

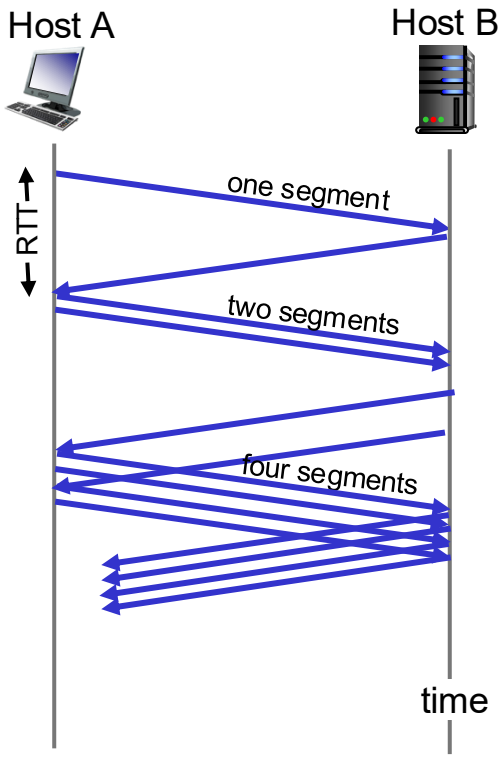
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP sender limits transmission: **`LastByteSent- LastByteAcked ≤ cwnd`**
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

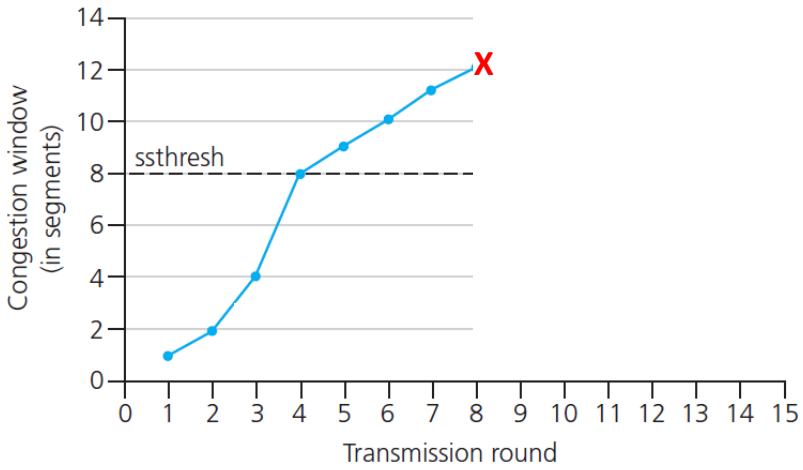
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

A: when **`cwnd`** gets to 1/2 of its value before timeout.

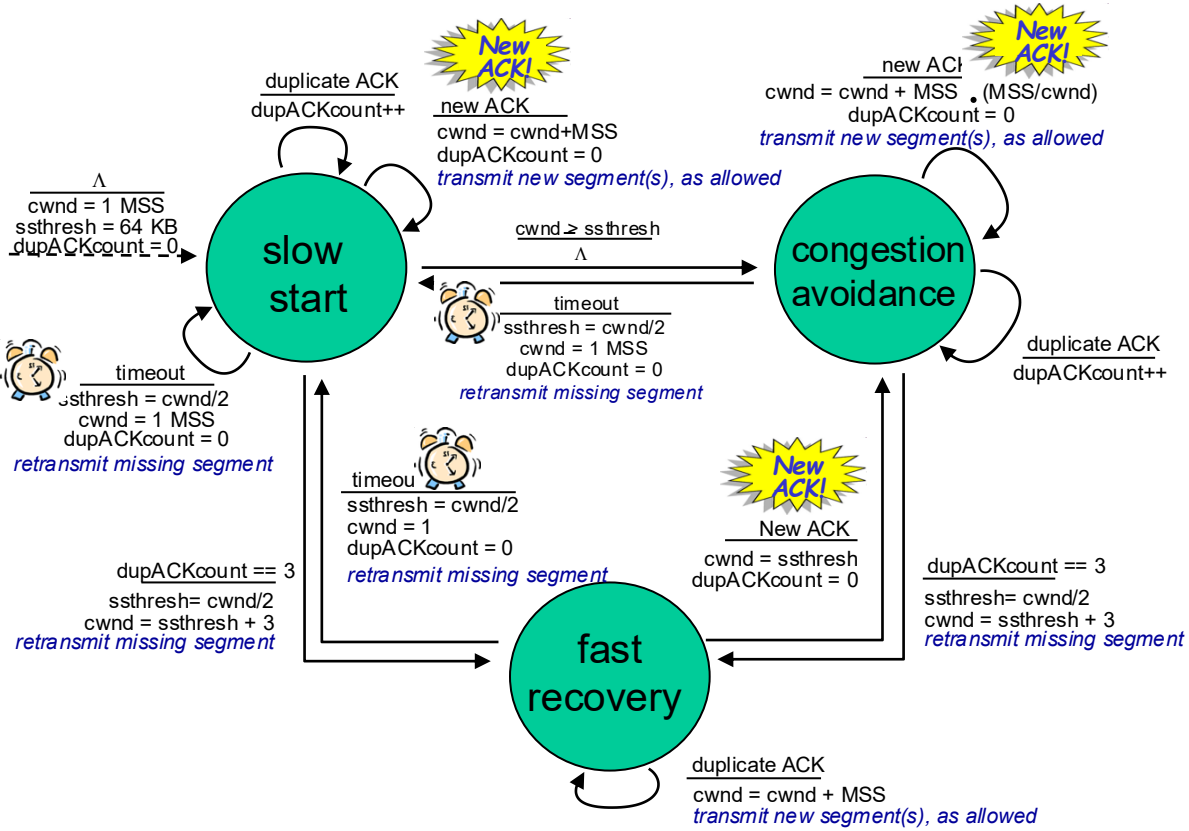
Implementation:

- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event



* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

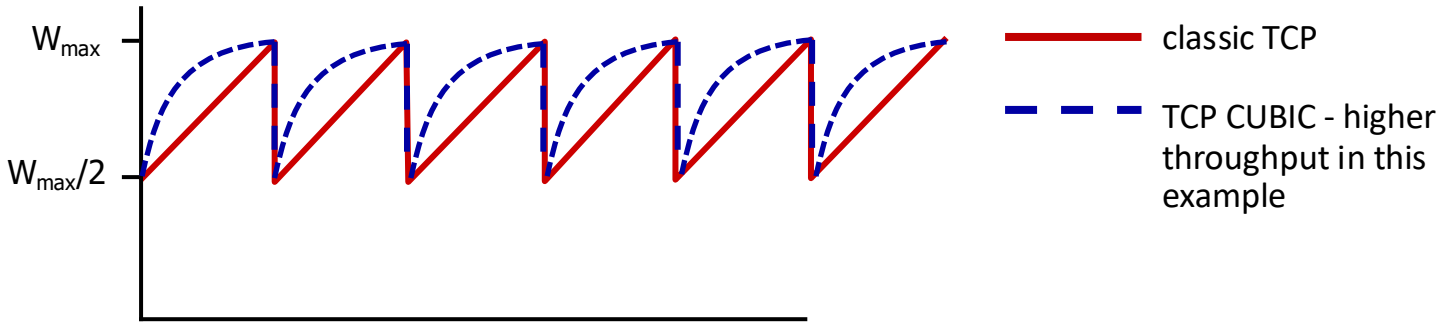
Summary: TCP congestion control



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP CUBIC

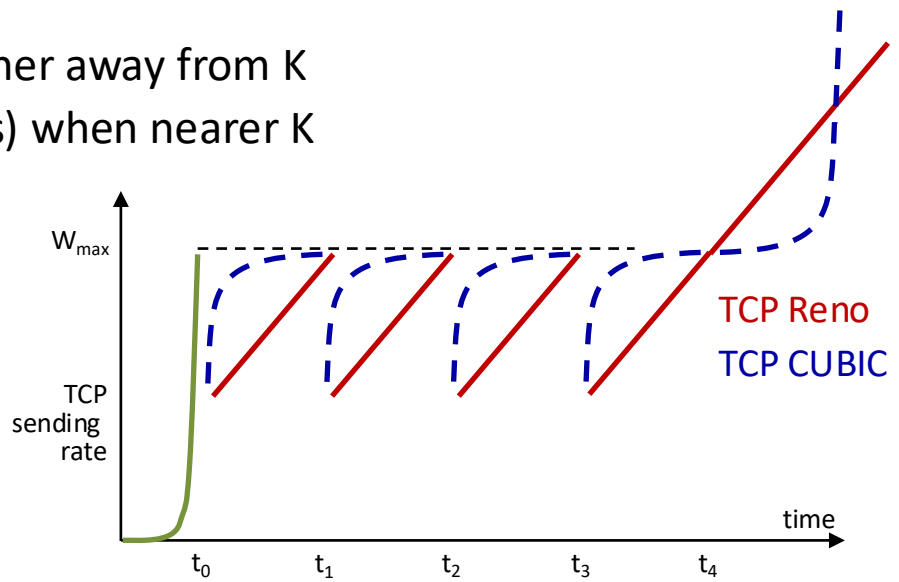
- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to W_{max} *faster*, but then approach W_{max} more *slowly*



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

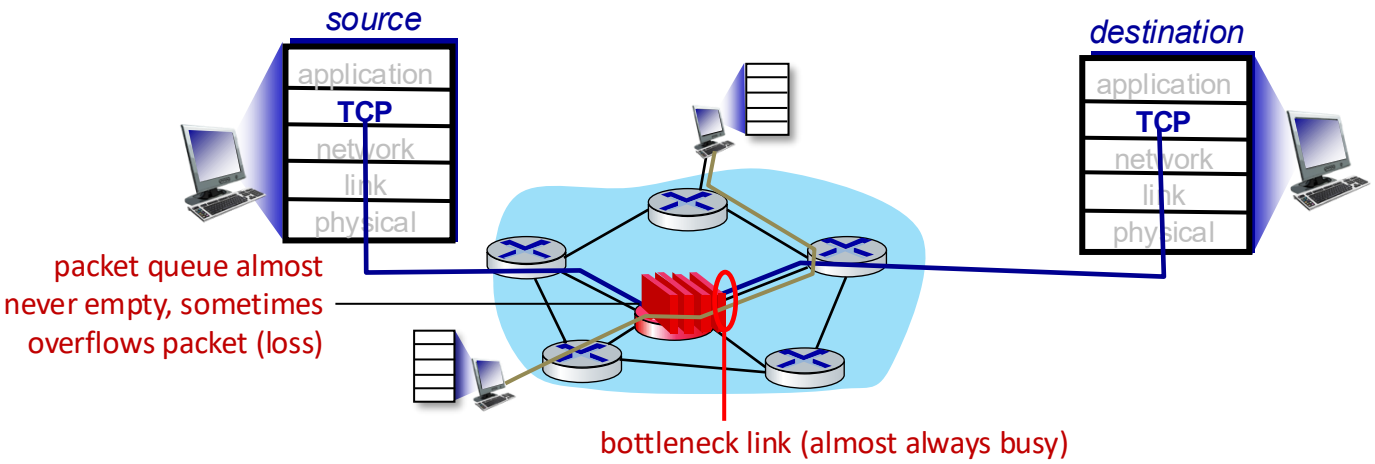
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



TCP and the congested "bottleneck link"

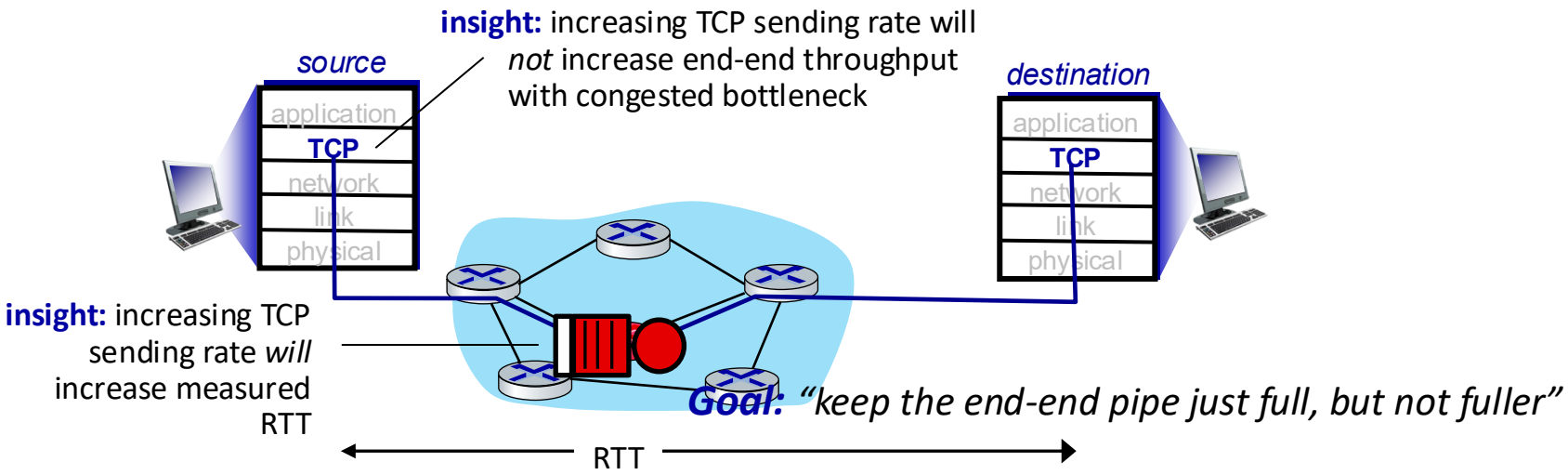
- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP and the congested "bottleneck link"

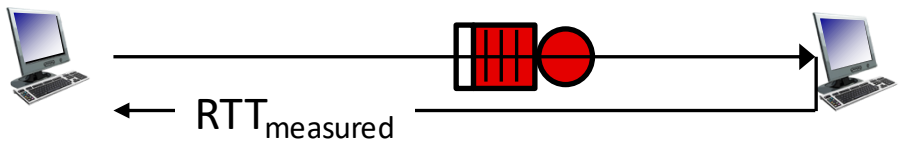
- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Delay-based TCP congestion control

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{\text{RTT}_{\text{measured}}}$$

if measured throughput "very close" to uncongested throughput
    increase `cwnd` linearly /* since path not congested */
else if measured throughput "far below" uncongested throughput
    decrease `cwnd` linearly /* since path is congested */

Delay-based TCP congestion control

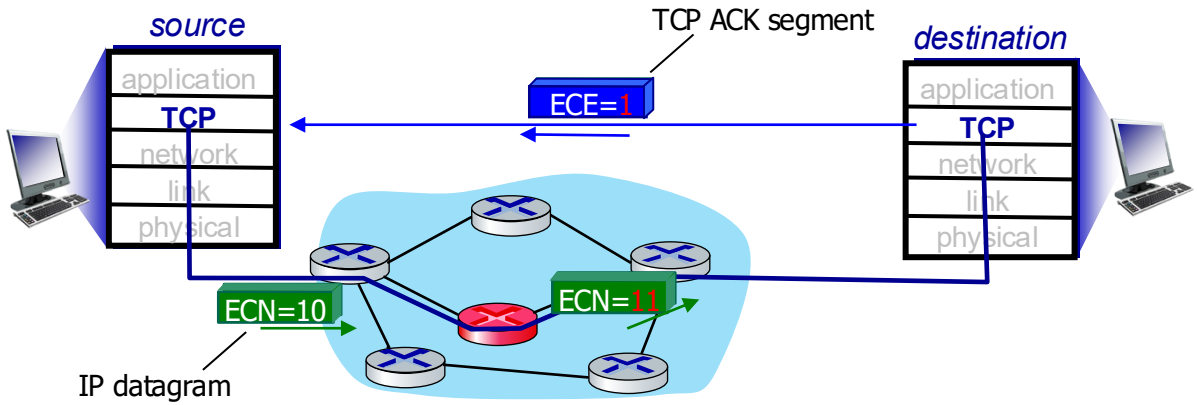
- congestion control without inducing/forcing loss
- maximizing throughput (“keeping the pipe just full… ”) while keeping delay low (“…but not fuller”)
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google’s (internal) backbone network

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

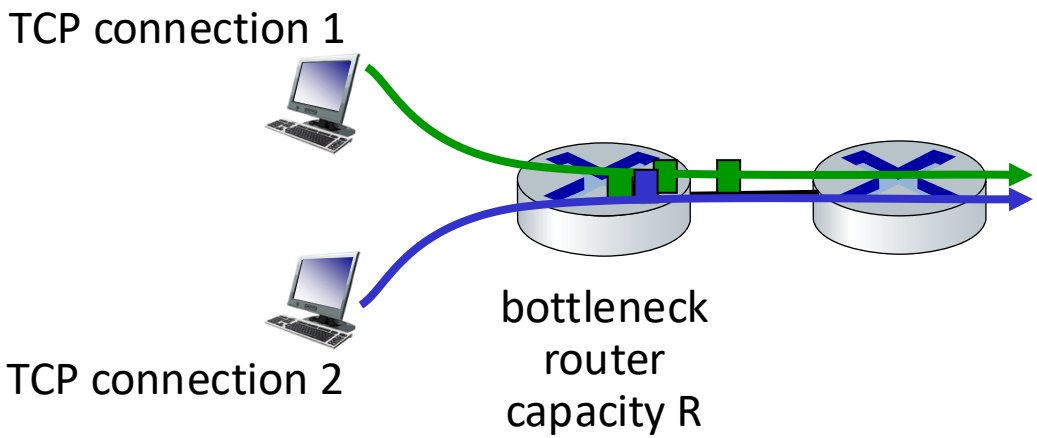
- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

TCP fairness

Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*

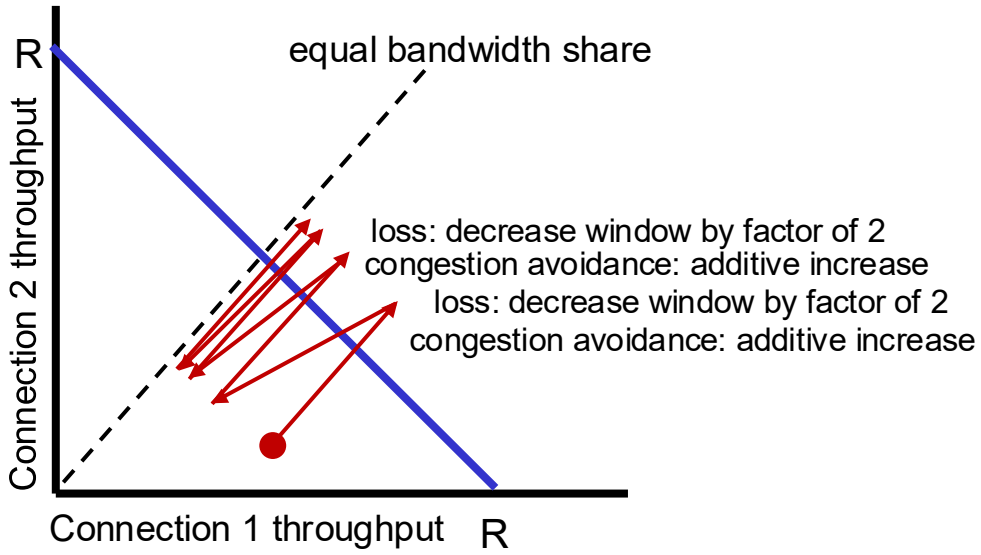


Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Fairness: must all network apps be “fair”?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no “Internet police” policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Evolving transport-layer functionality

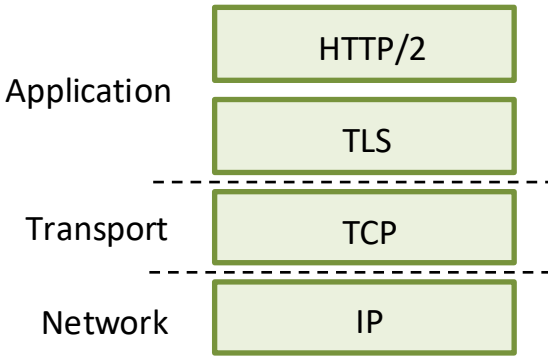
- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets "in flight"; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, "background" TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

- Application-layer protocol, on top of UDP
  - Increase performance of HTTP
  - Deployed on many Google servers, apps (Chrome, mobile YouTube app)



HTTP/2 over TCP

QUIC: Quick UDP Internet Connections

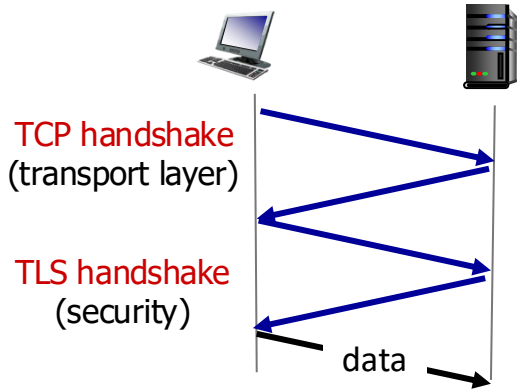
QUIC adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

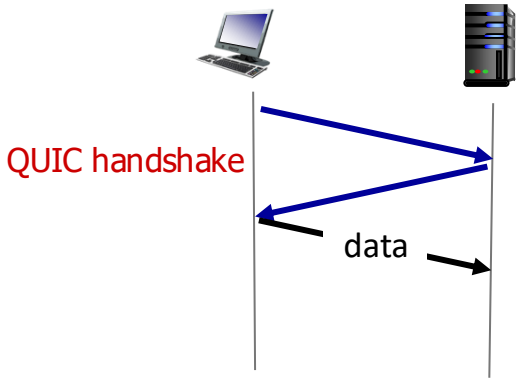
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

QUIC: Connection establishment



TCP (reliability, congestion control state) + TLS (authentication, crypto state)

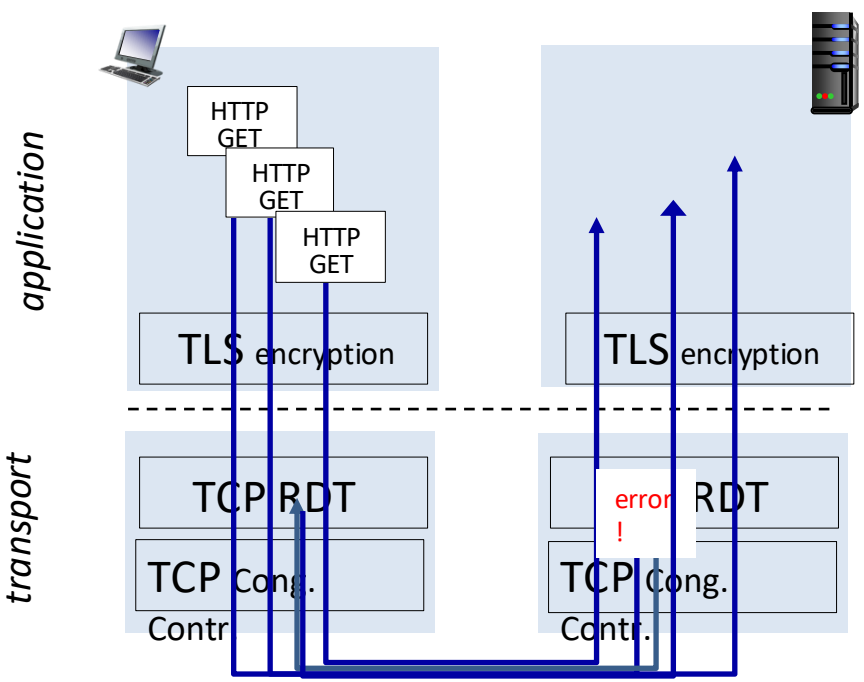
- 2 serial handshakes



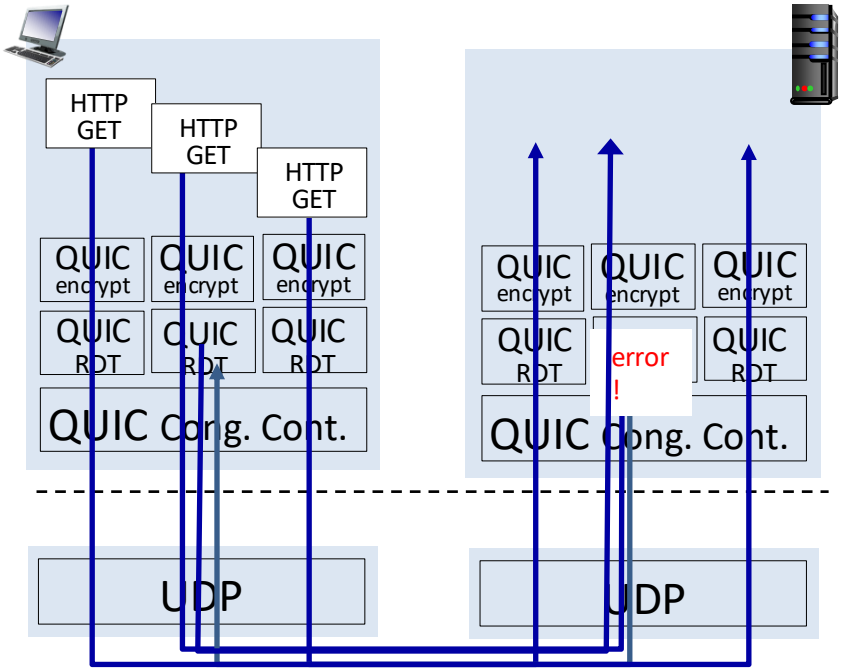
QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 2



(b) HTTP/2 with QUIC: no HOL blocking

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Summary

- Principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

- Leaving the network "edge" (application, transport layers)
- Into the network "core"
- Two network-layer days:
  - data plane
  - control plane

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach