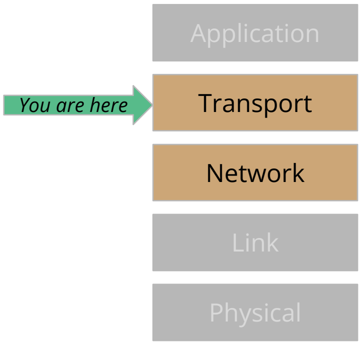
Ch03: Transport Layer

Layered Structure (Recall)



Application

You are here

Transport

Network

Link

Physical

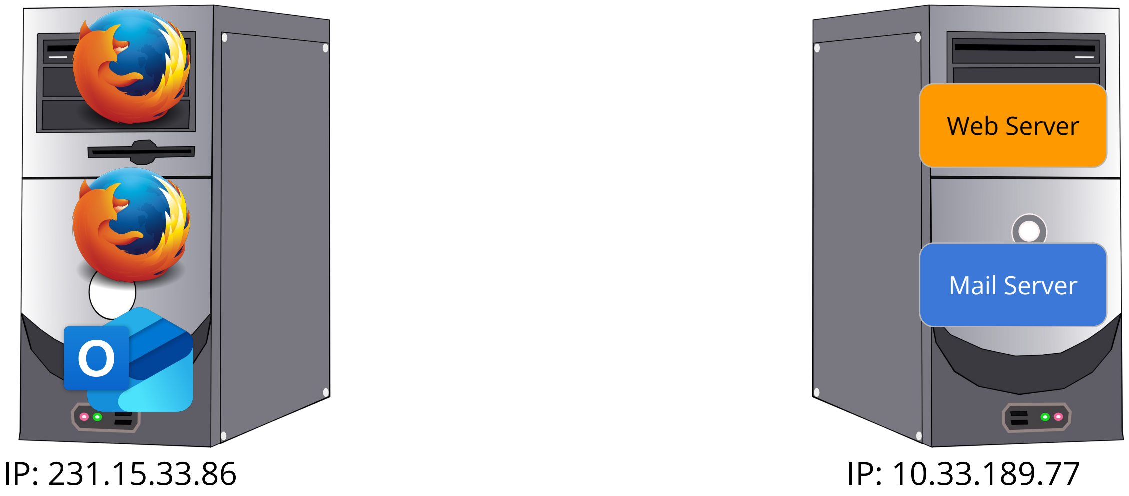
Transport Layer

- data transfer from **process** (*in one host*) to **process** (*in another host*)
- it assumes the existence of a (direct) logical channel between the two processes

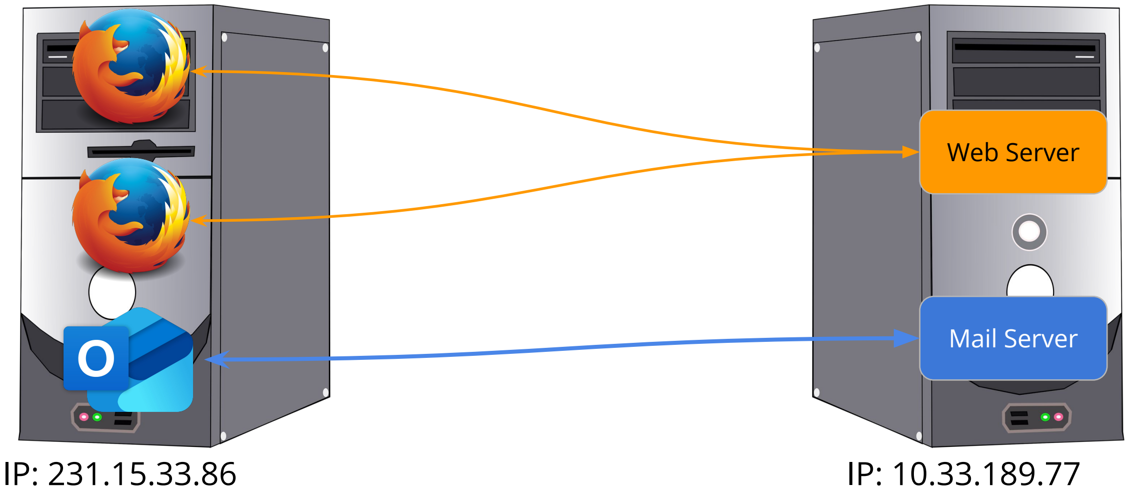
Network Layer

- data transfer from one **host** to another **host**

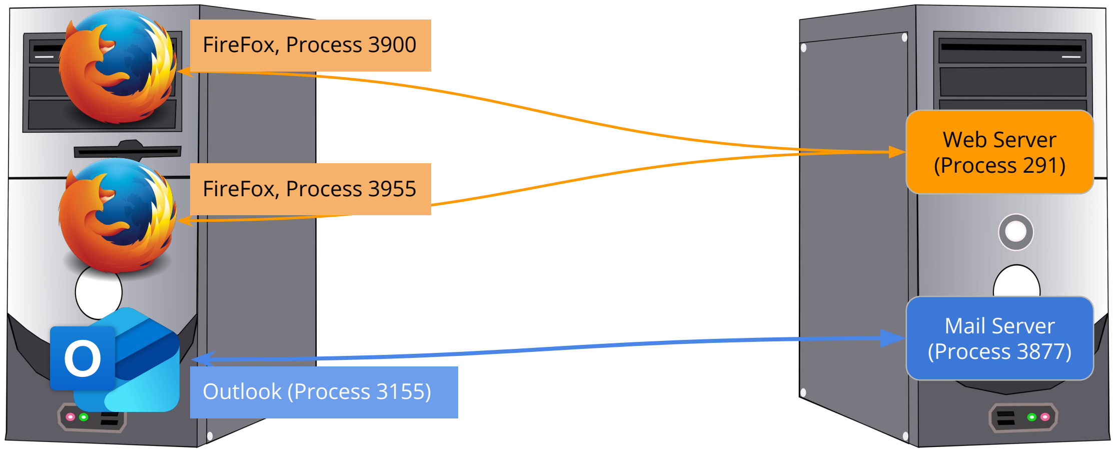
N Processes in One Host



Application Layer Perspective: App to App



Transport Layer Perspective



FireFox, Process 3900

FireFox, Process 3955

Outlook (Process 3155)

Web Server (Process 291)

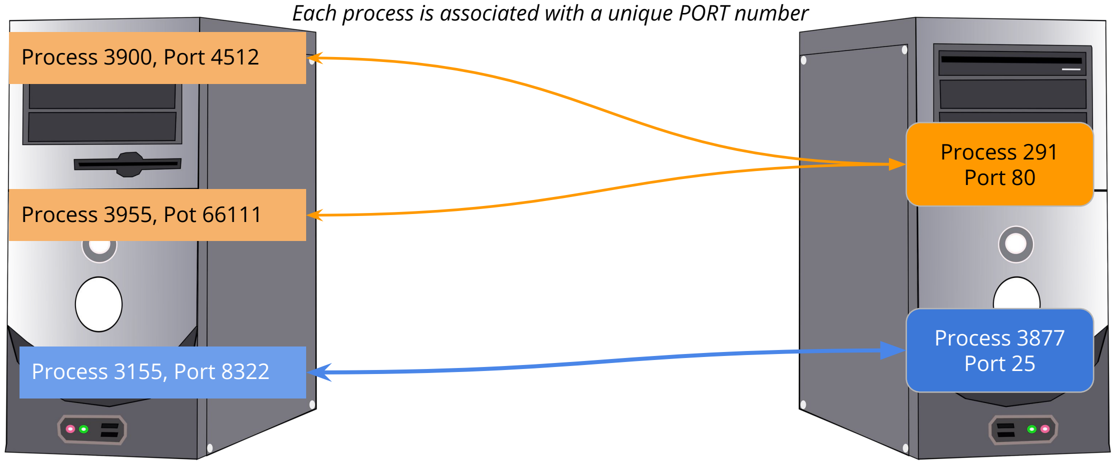
Mail Server (Process 3877)

IP: 231.15.33.86

IP: 10.33.189.77

Transport Layer Perspective: Process to Process

Each process is associated with a unique PORT number



Process 3900, Port 4512

Process 3955, Pot 66111

Process 3155, Port 8322

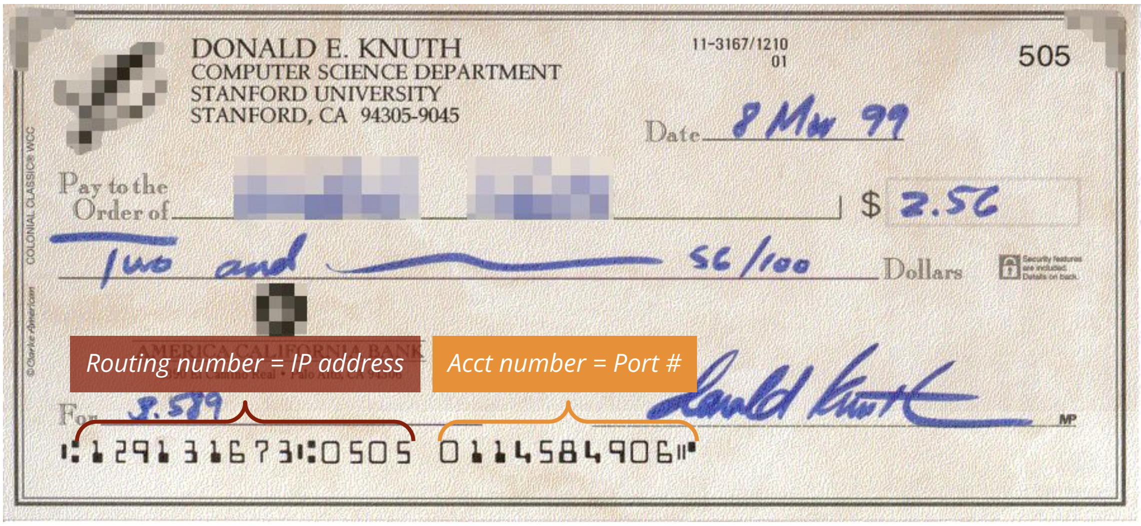
Process 291
Port 80

Process 3877
Port 25

IP: 231.15.33.86

IP: 10.33.189.77

Bank Routing Number & Account Number



DONALD E. KNUTH
COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY
STANFORD, CA 94305-9045

11-3167/1210
01

505

Date 8 Mw 99

Pay to the Order of [redacted] $ 2.56

Two and ~~~~ 56/100 Dollars

AMERICA CALIFORNIA BANK

For 8.589

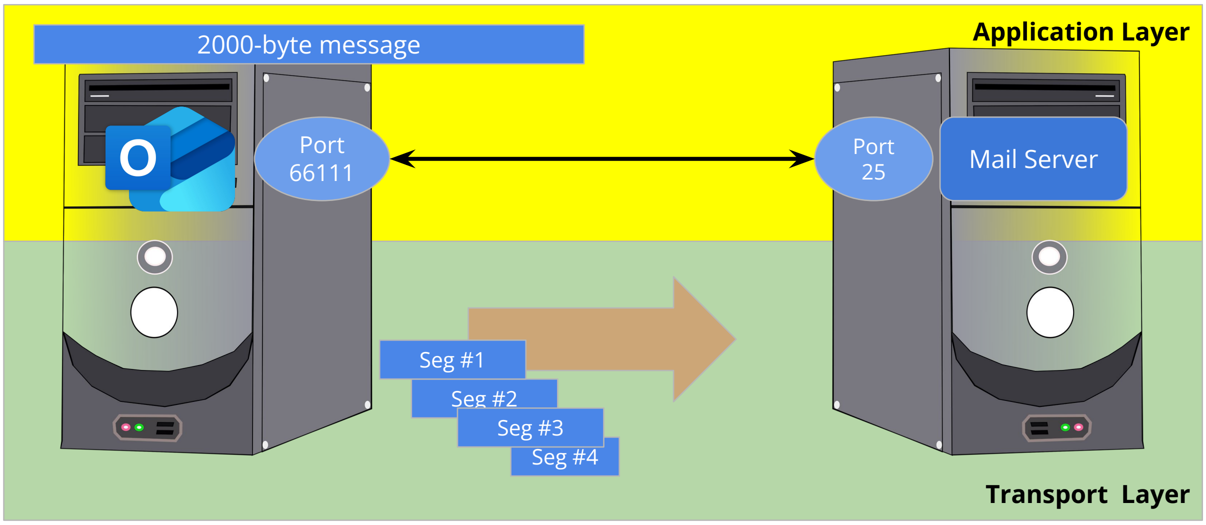
Donald Knuth

⑆129131673⑆0505 0114584906⑈

Routing number = IP address

Acct number = Port #

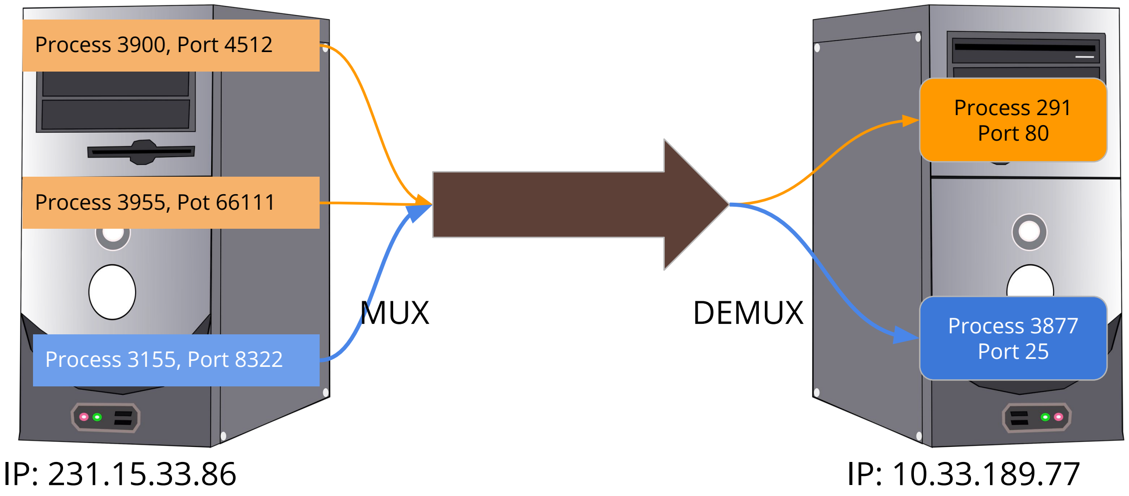
App Data (Messages) vs. Segment



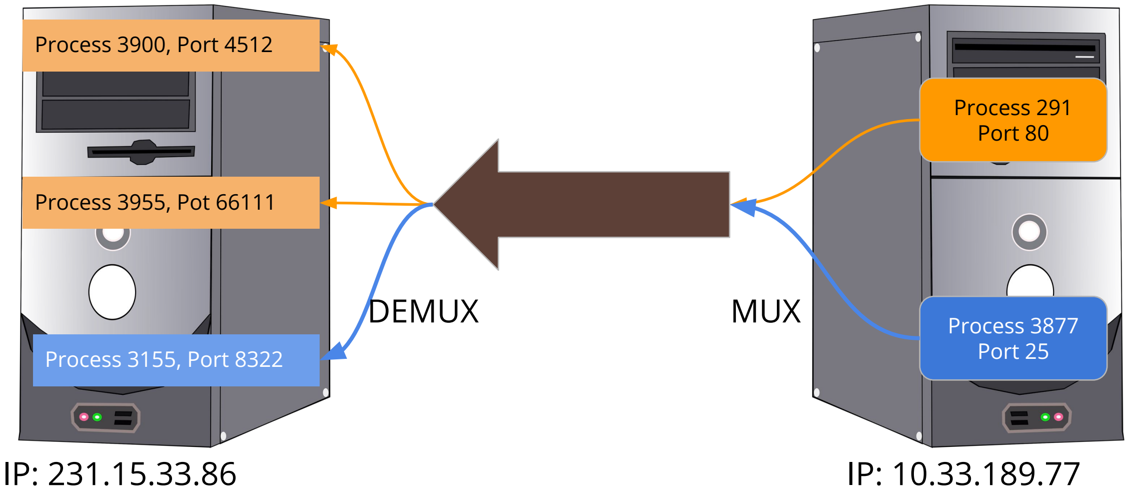
IP: 231.15.33.86

IP: 10.33.189.77

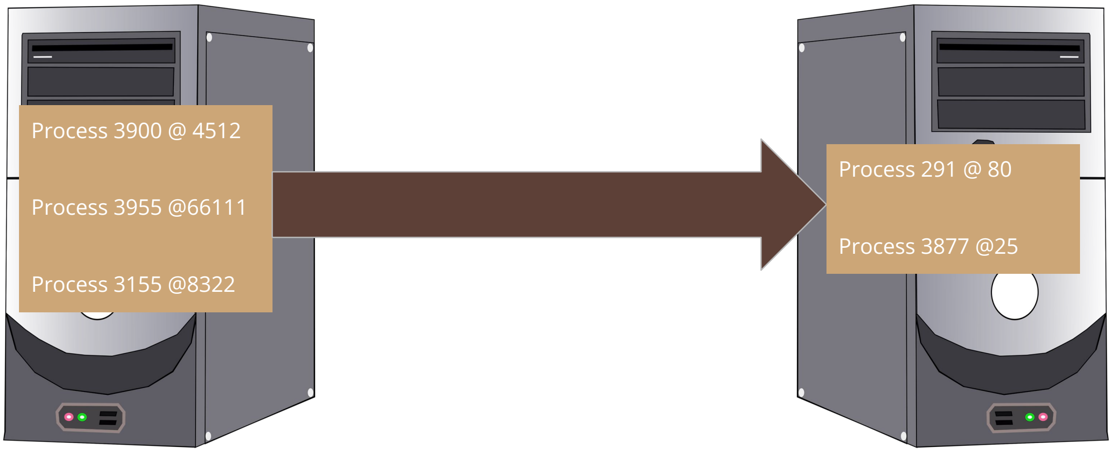
Transport Layer to Network Layer



Transport Layer to Network Layer



Network Layer: Host to Host



Process 3900 @ 4512

Process 3955 @66111

Process 3155 @8322

Process 291 @ 80

Process 3877 @25

IP: 231.15.33.86

IP: 10.33.189.77

Mux/Demux

- On a single host there can be **several processes creating a socket**
- Each socket must be associated with a **unique port** number
  - An attempt to create a socket with a port number currently in use will trigger an error
  - *We can't use the process ID as the port number*, because this will prevent a process from opening multiple sockets simultaneously
- When a data is pushed by the sender socket it will be received by the receiver socket.
  - The sender socket port number is unique among the other sockets on the sender host
  - The receiver socket port number is unique among the other sockets on the receiver host
  - Hence, each packet will always include both the **source** and **destination port numbers**

Mux/DeMux

Multiplexing (on Sender Side)

- On a single host, several processes (hence several sockets) may need to send packets into the network
- The transport layer must tag these packets with *port number of the sending socket* before pushing them into the network

Demultiplexing (on the Recipient Side)

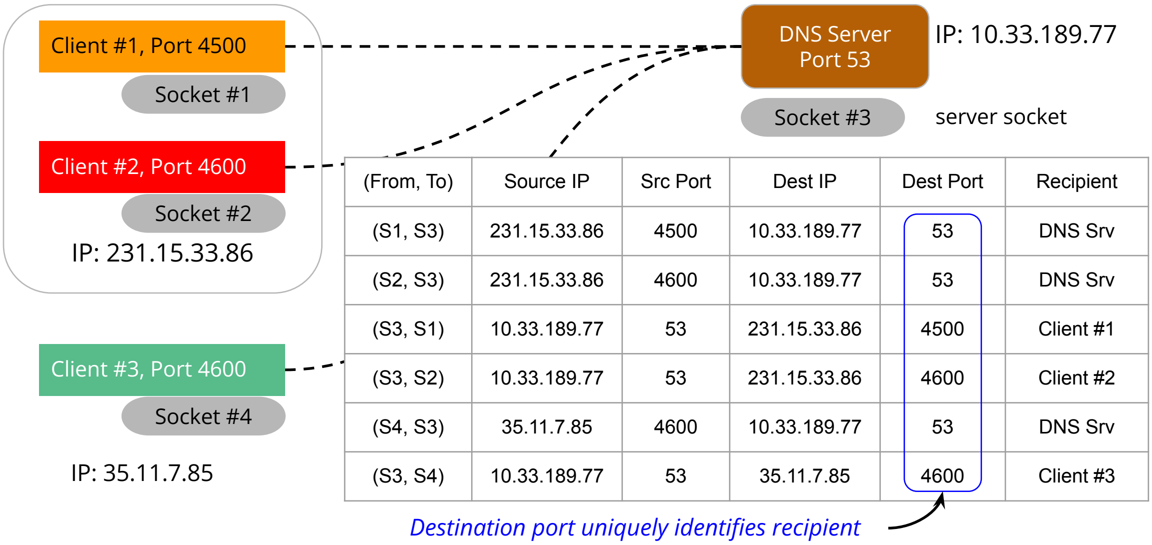
- On single host, several processes (hence several sockets) may be waiting for data (from the network)
- When a packet arrives at the host, the transport layer use the destination port number to forward the packet to the intended recipient socket

Communication with UDP Sockets

```
# Server Side
SERVER_PORT = 53  # DNS
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(“”, SERVER_PORT)
while True:
  data, addr = serverSocket.recvfrom(____)
  # do work here
  serverSocket.sendto(_____)
```

```
# Client Side
SRV_PORT = 53
SRV_ADDR = “xx.yy.zz.ww”
clientSocket = socket(AF_INET, SOCK_DGRAM)
clientSocket.sendTo(___, (SRV_ADDR,PORT))
# Do work here
clientSocket.recvfrom(____)
```

UDP Demux Details



(From, To)	Source IP	Src Port	Dest IP	Dest Port	Recipient
(S1, S3)	231.15.33.86	4500	10.33.189.77	53	DNS Srv
(S2, S3)	231.15.33.86	4600	10.33.189.77	53	DNS Srv
(S3, S1)	10.33.189.77	53	231.15.33.86	4500	Client #1
(S3, S2)	10.33.189.77	53	231.15.33.86	4600	Client #2
(S4, S3)	35.11.7.85	4600	10.33.189.77	53	DNS Srv
(S3, S4)	10.33.189.77	53	35.11.7.85	4600	Client #3

Destination port uniquely identifies recipient

Demultiplexing UDP packets

- Communication via UDP sockets involves *only the two sockets* (one at the sender host and one at the recipient host)
- Dispatching incoming packets to the intended recipient can done by using **only the destination port** number on the recipient host

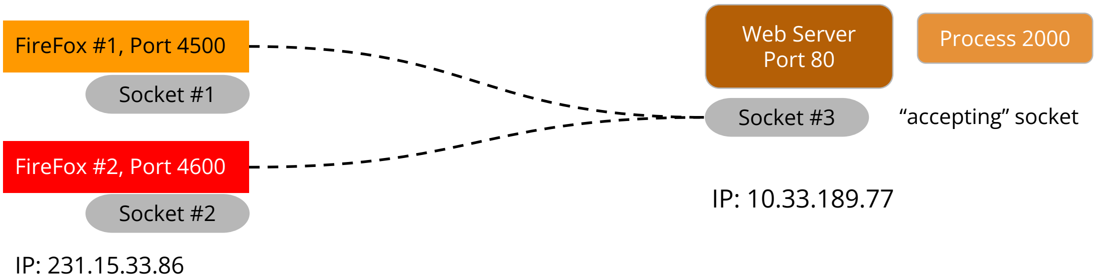
Communication with TCP Sockets

```
# Server Side
SERVER_PORT = 7777
acceptSocket = socket(AF_INET, SOCK_STREAM)
acceptSocket.bind(“”, SERVER_PORT)
acceptSocket.listen(1)
while True:
  connectSocket, addr = acceptSocket.accept()
  # do work here
  connectSocket.close()
```

```
# Client Side #1
SERVER_PORT = 7777
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect(“”, SERVER_PORT)
# Do work here
clientSocket.close()
```

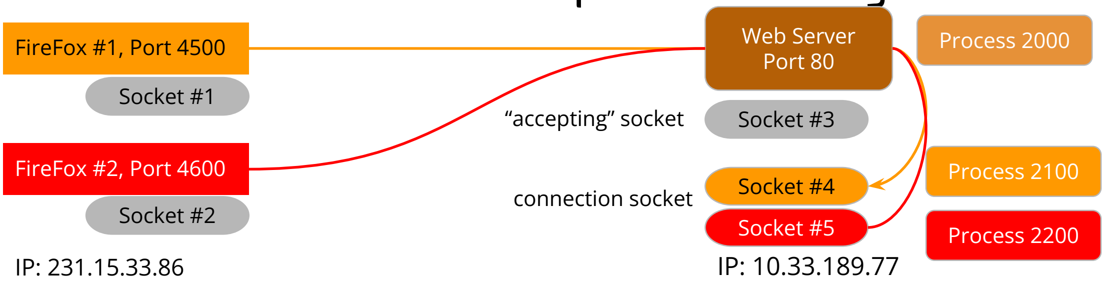
```
# Client Side #2
SERVER_PORT = 7777
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect(“”, SERVER_PORT)
# Do work here
clientSocket.close()
```

TCP Demux Details: Initial Connection



(From, To)	Source IP	Source Port	Dest IP	Dest Port	Recipient
(S1, S3)	231.15.33.86	4500	10.33.189.77	80	PID 2000
(S2, S3)	231.15.33.86	4600	10.33.189.77	80	PID 2000

TCP Demux Details: Subsequent Exchanges



Socket Pair	Source IP	Source Port	Dest IP	Dest Port	Recipient
(S1, S4)	231.15.33.86	4500	10.33.189.77	80	PID 2100
(S2, S5)	231.15.33.86	4600	10.33.189.77	80	PID 2200

Destination port DOES NOT uniquely identifies recipient.

Must also include Source (IP & Port) to identify recipient

Demultiplexing TCP packets

- The server is listening for new connection on the **accepting socket**
- A new client connection creates a **third socket**, created by the server at the time of `accept()` in response to client `connect()`
  - There is always ONE instance of accepting socket
  - But potentially multiple instances of these "third socket"s (one per client connection)
- But the **third socket** is local to the server and the client has no knowledge of its details. The client must continue to use the port number of the **accepting socket** as the destination port number
- Using only the destination port, the server will not able to forward incoming packets to the correct instance of "**third socket**"
  - Hence the 4-tuple (*source IP, source port, dest IP, dest port*) must be used

TCP vs. UDP

	UDP	TCP
Reliable	❌	✅
In-order delivery	❌	✅
Flow Control	❌	✅
Congestion Control	❌	✅
Delay guarantee	❌	❌
Bandwidth guarantee	❌	❌
Require connection setup	No	Yes

UDP Jokes

https://medium.com/pragmatic-programmers/udp-humor-bd20bcdd355e

Joke #1:

I was recently invited to a costume party. I dressed up as a UDP packet, but no one acknowledged me

Joke #2:

The problem with UDP jokes is that I don't get half of them!

Joke #3:

You know the best part of UDP jokes? If the other person doesn't get it, I don't care

Joke #4:

A UDP packet walks into a bar. A walks UDP packet bar a into.

UDP reliability: “correctness” data (*if received*) is verified via checksum

Reliable Data Transfer

Expectations of “Reliability”

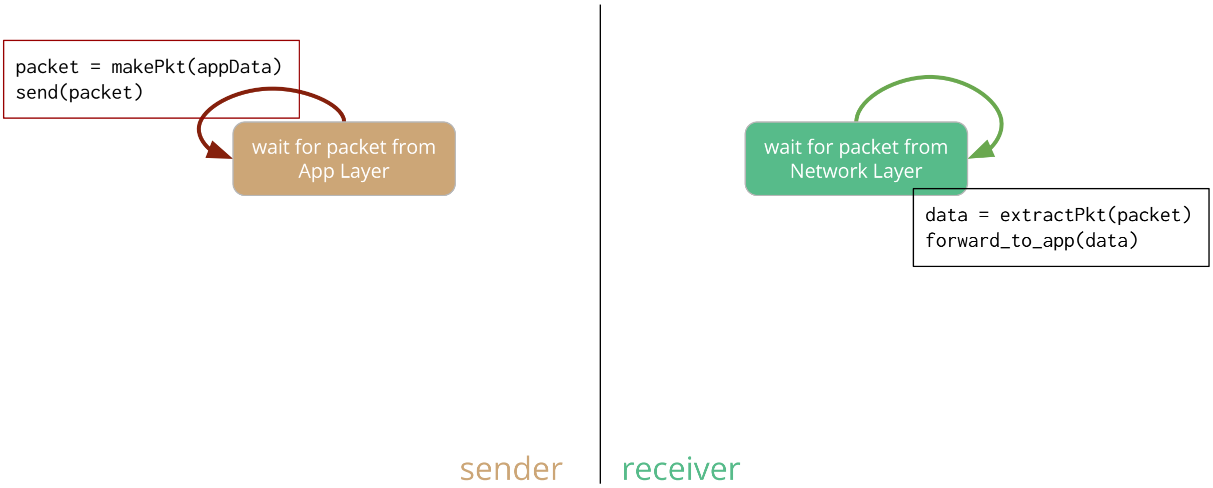
- No packet loss
- No data corruption
- In-order delivery

Textbook: Reliable Data Transfer

My slides:

- Reliable Data Transport
- RDT x.x ⇒ FSM x.x

FSM 1.0 Reliable Data Transport



```
packet = makePkt(appData)
send(packet)
```

wait for packet from App Layer

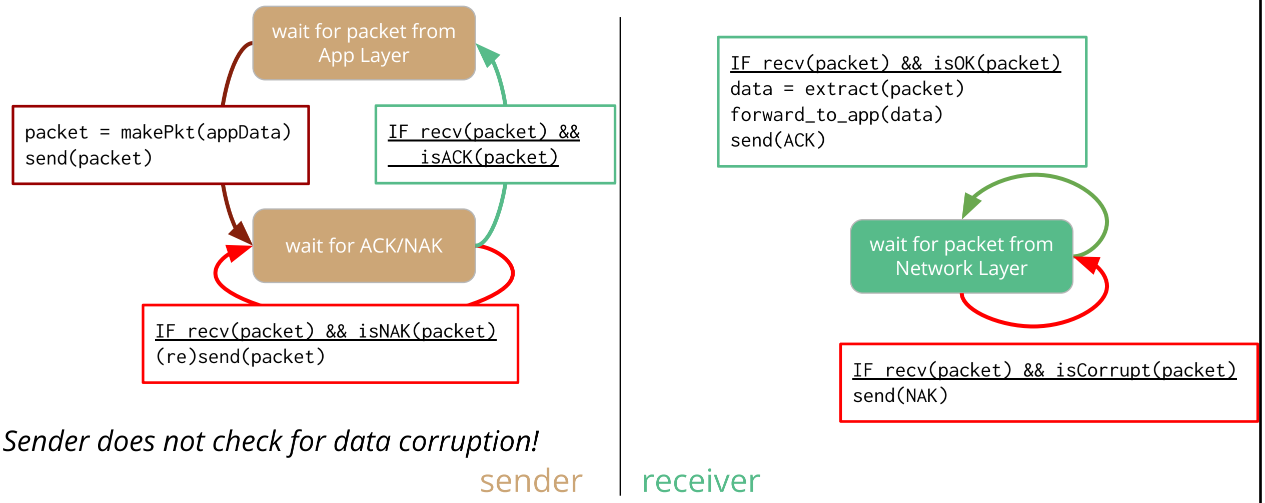
wait for packet from Network Layer

```
data = extractPkt(packet)
forward_to_app(data)
```

sender

receiver

FSM 2.0 Data Transport With Data Errors



wait for packet from App Layer

```
packet = makePkt(appData)
send(packet)
```

```
IF recv(packet) &&
   isACK(packet)
```

wait for ACK/NAK

```
IF recv(packet) && isNAK(packet)
(re)send(packet)
```

```
IF recv(packet) && isOK(packet)
data = extract(packet)
forward_to_app(data)
send(ACK)
```

wait for packet from Network Layer

```
IF recv(packet) && isCorrupt(packet)
send(NAK)
```

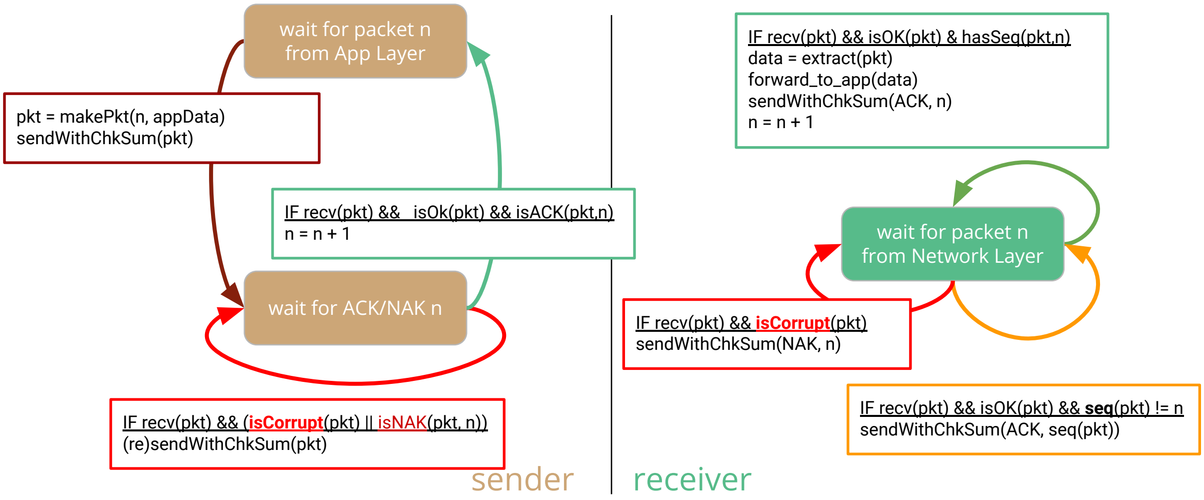
Sender does not check for data corruption!

sender

receiver

FSM 2.0 Issues:
Does not check for ACK/NAK corruption
What to send when ACK/NAK is corrupted?

FSM 2.1 = FSM 2.0 with ACK/NAK errors



wait for packet n from App Layer

pkt = makePkt(n, appData)
sendWithChkSum(pkt)

IF recv(pkt) && isOk(pkt) && isACK(pkt,n)
n = n + 1

wait for ACK/NAK n

IF recv(pkt) && (**isCorrupt**(pkt) || isNAK(pkt, n))
(re)sendWithChkSum(pkt)

sender

IF recv(pkt) && isOK(pkt) & hasSeq(pkt,n)
data = extract(pkt)
forward_to_app(data)
sendWithChkSum(ACK, n)
n = n + 1

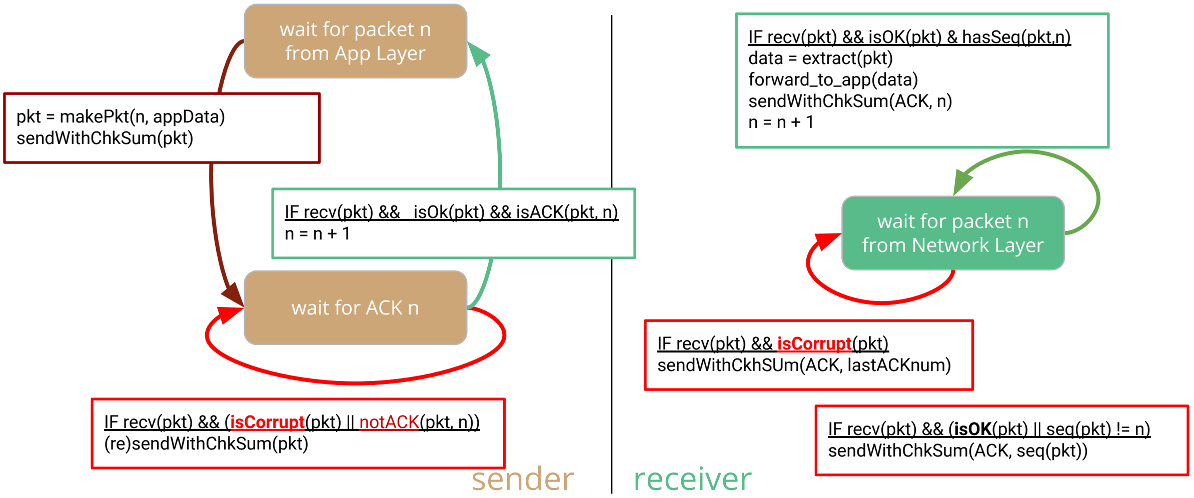
wait for packet n from Network Layer

IF recv(pkt) && **isCorrupt**(pkt)
sendWithChkSum(NAK, n)

IF recv(pkt) && isOK(pkt) && **seq**(pkt) != n
sendWithChkSum(ACK, seq(pkt))

receiver

FSM 2.2 = FSM 2.1 without NAK



wait for packet n from App Layer

pkt = makePkt(n, appData)
sendWithChkSum(pkt)

IF recv(pkt) && isOk(pkt) && isACK(pkt, n)
n = n + 1

wait for ACK n

IF recv(pkt) && (isCorrupt(pkt) || notACK(pkt, n))
(re)sendWithChkSum(pkt)

sender

IF recv(pkt) && isOK(pkt) & hasSeq(pkt,n)
data = extract(pkt)
forward_to_app(data)
sendWithChkSum(ACK, n)
n = n + 1

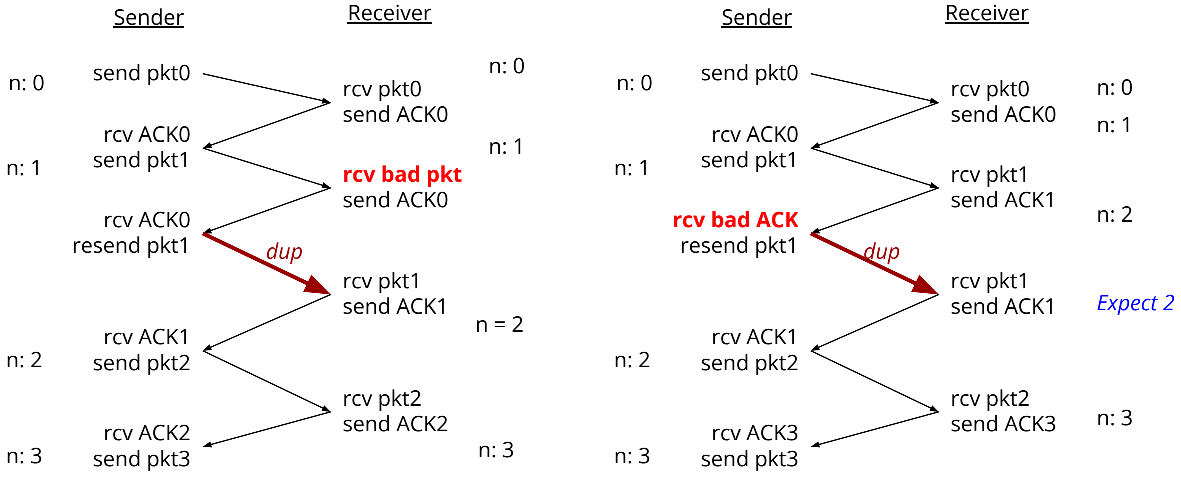
wait for packet n from Network Layer

IF recv(pkt) && isCorrupt(pkt)
sendWithCkhSUm(ACK, lastACKnum)

IF recv(pkt) && (isOK(pkt) || seq(pkt) != n)
sendWithChkSum(ACK, seq(pkt))

receiver

FSM 2.2 in action



Sender

Receiver

n: 0

send pkt0

n: 0

rcv pkt0
send ACK0

n: 1

rcv ACK0
send pkt1

n: 1

rcv bad pkt
send ACK0

rcv ACK0
resend pkt1

dup

rcv pkt1
send ACK1

n = 2

n: 2

rcv ACK1
send pkt2

rcv pkt2
send ACK2

n: 3

rcv ACK2
send pkt3

n: 3

Sender

Receiver

n: 0

send pkt0

rcv pkt0
send ACK0

n: 0

n: 1

n: 1

rcv ACK0
send pkt1

rcv pkt1
send ACK1

n: 2

rcv bad ACK
resend pkt1

dup

rcv pkt1
send ACK1

Expect 2

n: 2

rcv ACK1
send pkt2

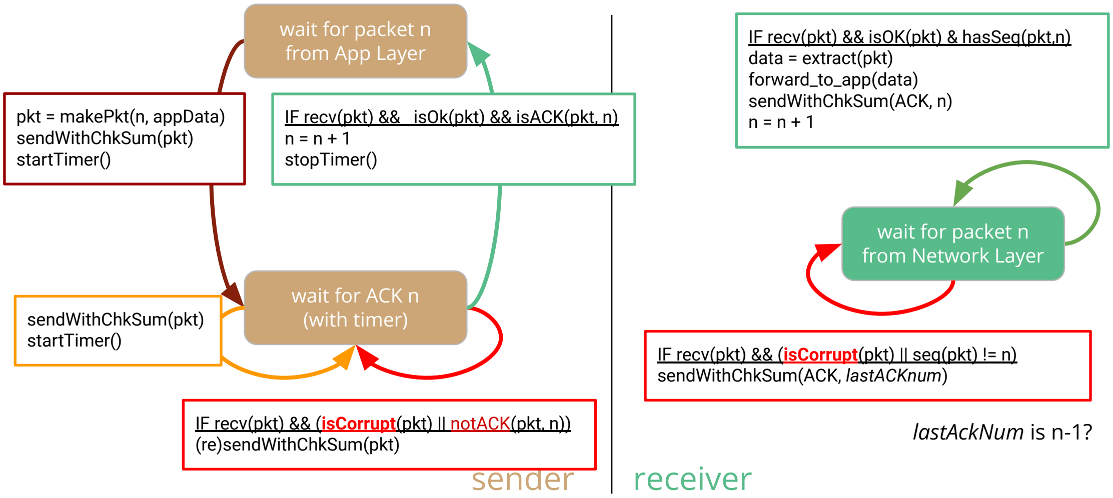
rcv pkt2
send ACK3

n: 3

n: 3

rcv ACK3
send pkt3

FSM 3.0 = FSM 2.2 with Sender TimeOut



wait for packet n from App Layer

pkt = makePkt(n, appData)
sendWithChkSum(pkt)
startTimer()

IF recv(pkt) && isOk(pkt) && isACK(pkt, n)
n = n + 1
stopTimer()

wait for ACK n (with timer)

sendWithChkSum(pkt)
startTimer()

IF recv(pkt) && (**isCorrupt**(pkt) || notACK(pkt, n))
(re)sendWithChkSum(pkt)

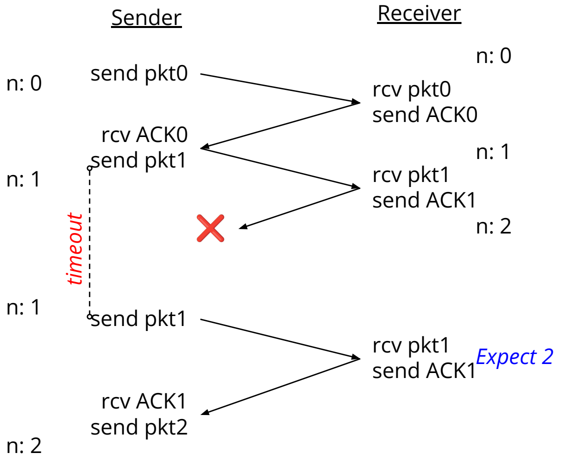
IF recv(pkt) && isOK(pkt) & hasSeq(pkt,n)
data = extract(pkt)
forward_to_app(data)
sendWithChkSum(ACK, n)
n = n + 1

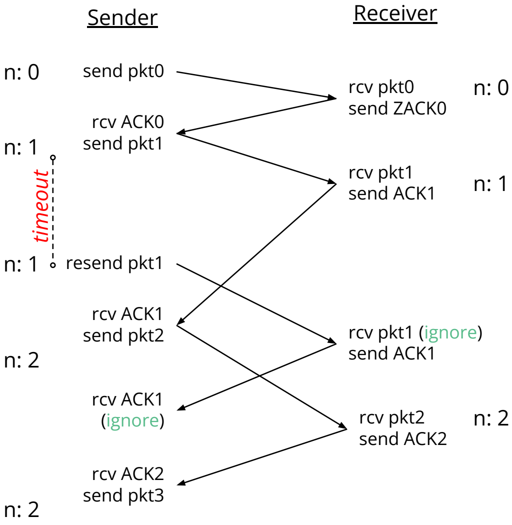
wait for packet n from Network Layer

IF recv(pkt) && (**isCorrupt**(pkt) || seq(pkt) != n)
sendWithChkSum(ACK, *lastACKnum*)

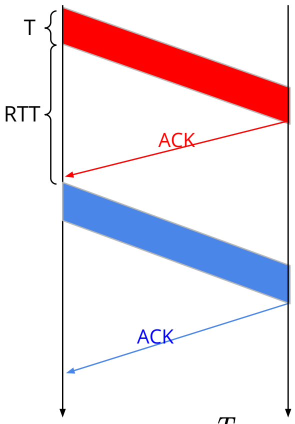
lastAckNum is n-1?

FSM 3.0 in action





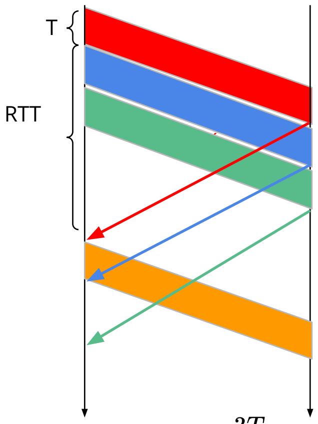
Stop & Wait for ACK



Utilization = $\frac{T}{RTT+T}$

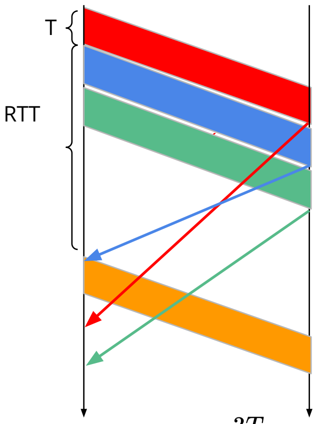
$$\text{Utilization} = \frac{\text{total time to transmit}}{\text{total time until first ACK}}$$

Pipelined Transmission



Utilization = $\frac{3T}{RTT+T}$

$$\text{Utilization} = \frac{\text{total time to transmit}}{\text{total time until first ACK}}$$



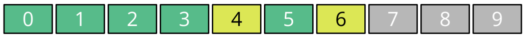
Utilization = $\frac{3T}{RTT+T}$

How long can we increase the pipeline?

Pipelined Packet Types

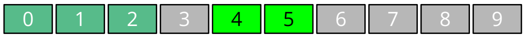
On Sender

- Sent and ACKed
- Sent but non ACKed (in-flight)
- Not (yet) Sent



On Receiver

- Received in-order and ACKed
- Received out-of-order
- Not (yet) Received

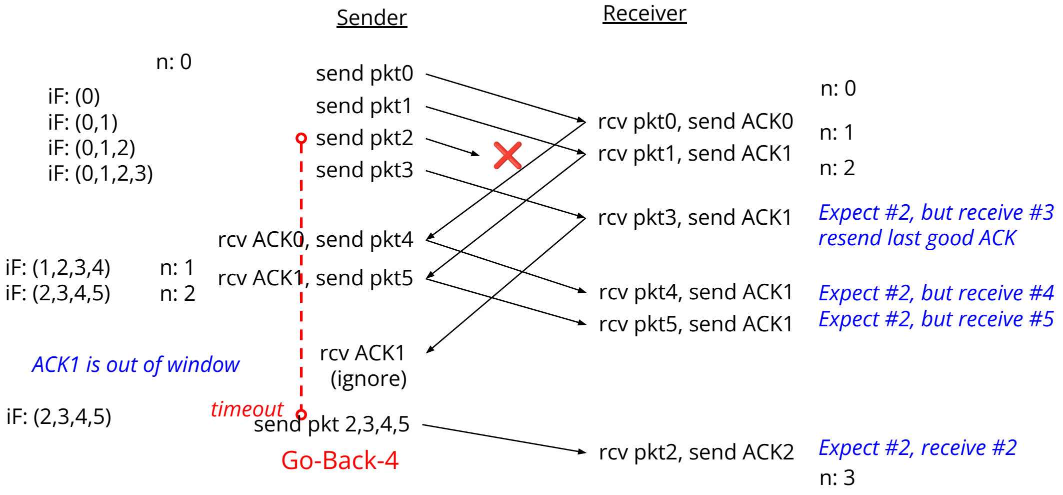


Pipelined Packets: Implementation

	Sender	**Receiver**
Go-Back-N	One timer set for the oldest in-flight packet. OnTimeout: resend all (“Go Back”) N packets	Cumulative ACK
Selective Repeat	Multiple timers: one for each in-flight packet OnTimeout(**k**) resend only packet(**k**). “Selective”	Individual ACK

Go-Back-N = Pipeline + Improved FSM 3.0

FSM 3.0 + PipeLine (in-flight: 4)



Sender

Receiver

n: 0

iF: (0)
iF: (0,1)
iF: (0,1,2)
iF: (0,1,2,3)

send pkt0
send pkt1
send pkt2
send pkt3

n: 0

rcv pkt0, send ACK0

n: 1

rcv pkt1, send ACK1

n: 2

rcv pkt3, send ACK1

Expect #2, but receive #3
resend last good ACK

rcv ACK0, send pkt4

iF: (1,2,3,4) n: 1
iF: (2,3,4,5) n: 2

rcv ACK1, send pkt5

rcv pkt4, send ACK1 *Expect #2, but receive #4*

rcv pkt5, send ACK1 *Expect #2, but receive #5*

ACK1 is out of window

rcv ACK1
(ignore)

iF: (2,3,4,5)

timeout

send pkt 2,3,4,5

Go-Back-4

rcv pkt2, send ACK2 *Expect #2, receive #2*

n: 3

Go-Back-N

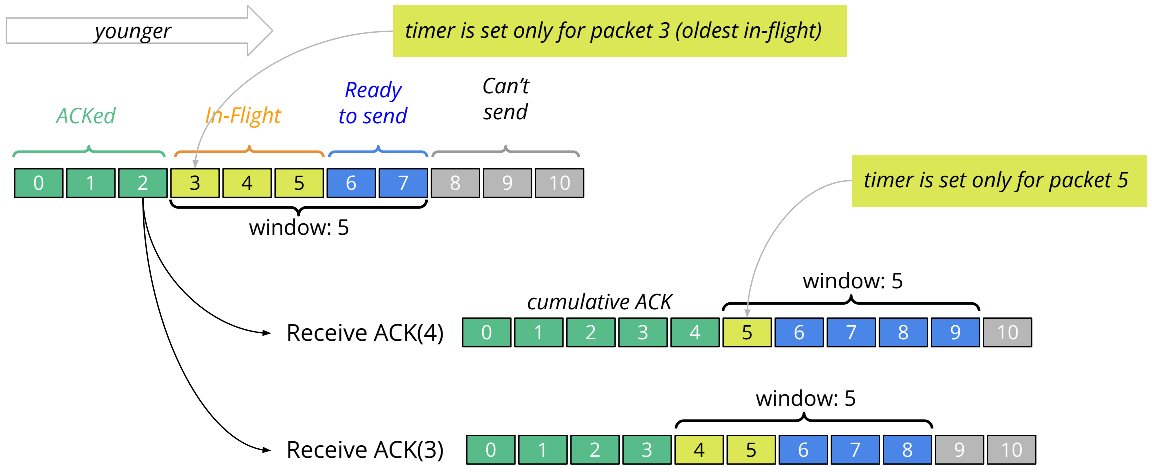
Sender

- A sliding window of size N
- Max N packets allowed to be in the pipeline ("in-flight")
- Cumulative ACK: ACK(N) means all packets k ≤ N have been received.
  - Packet N is the **youngest** ACKed packet
  - The window shifts to position N + 1, i.e. N+1 is now the **oldest** in-flight packet
- Set timer only for the oldest in-flight packet
- On timeout(**p**): resend packet **p** and higher (younger) within the allowed window

Receiver

- Use NO sliding window
- Only ACK in-order packets (**oldest in-flight packet***)*
  - When out-of-order packet arrived, re-ACK with the highest in-order packet (youngest packet ACKed)
- It is sufficient to keep track the youngest ACKed packet
- Young/old is by birth at the sender (not by arrival at the receiver)

Go-Back-5 Sender Window (Size = 5)



younger

timer is set only for packet 3 (oldest in-flight)

ACKed *In-Flight* *Ready to send* *Can't send*

0 1 2 3 4 5 6 7 8 9 10

window: 5

timer is set only for packet 5

window: 5

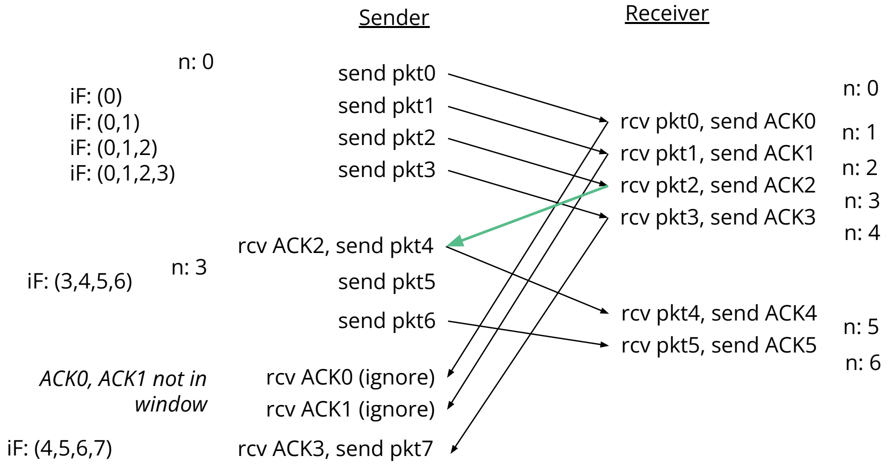
cumulative ACK

Receive ACK(4) 0 1 2 3 4 5 6 7 8 9 10

window: 5

Receive ACK(3) 0 1 2 3 4 5 6 7 8 9 10

Go-Back-N: Cumulative ACK



Sender

Receiver

n: 0

iF: (0)
iF: (0,1)
iF: (0,1,2)
iF: (0,1,2,3)

send pkt0
send pkt1
send pkt2
send pkt3

rcv pkt0, send ACK0
rcv pkt1, send ACK1
rcv pkt2, send ACK2
rcv pkt3, send ACK3

n: 0
n: 1
n: 2
n: 3
n: 4

rcv ACK2, send pkt4

n: 3

iF: (3,4,5,6)

send pkt5

send pkt6

rcv pkt4, send ACK4
rcv pkt5, send ACK5

n: 5
n: 6

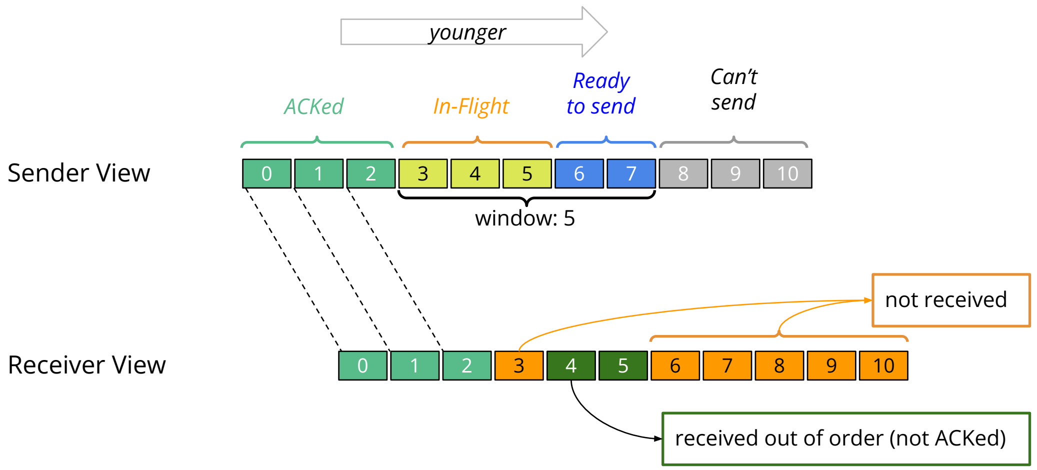
ACK0, ACK1 not in window

rcv ACK0 (ignore)
rcv ACK1 (ignore)

iF: (4,5,6,7)

rcv ACK3, send pkt7

Go-Back-5 Packet Lineup



younger

ACKed

In-Flight

Ready to send

Can’t send

Sender View

0 1 2 3 4 5 6 7 8 9 10

window: 5

not received

Receiver View

0 1 2 3 4 5 6 7 8 9 10

received out of order (not ACKed)

Go-Back-N Animation

Selective Repeat

Selective Repeat

Sender

- A sliding window of size N
- Max N packets allowed to be in the pipeline ("in-flight")
- Set one timer each in-flight packet
  - Timeout can be observed per packet
  - On timeout(**p**): resend only packet **p**
- Slide the window (forward) where there is no gap in the ACKed packets

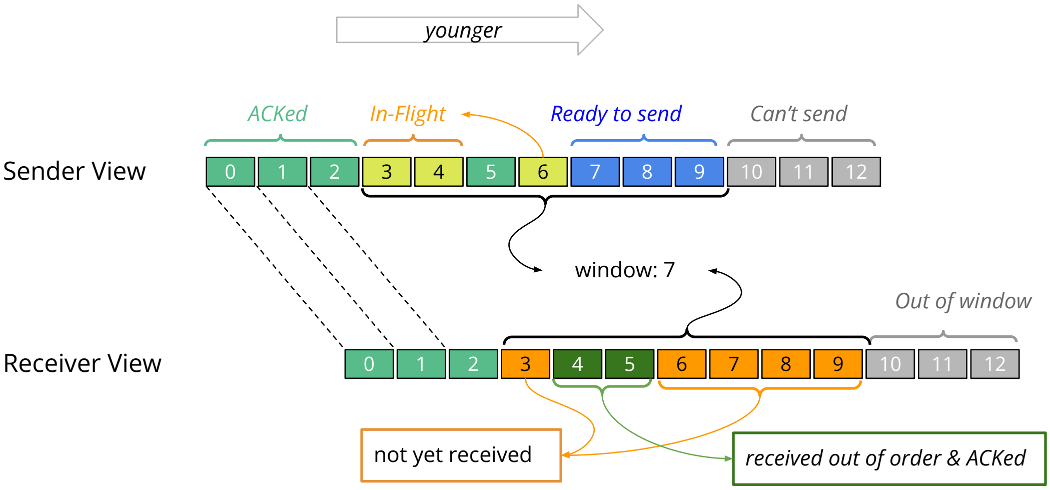
Receiver

- A sliding window of size N
- Max N packets expected to be in-flight

- Individual ACK for both in-order & out-of-order packets

  Slide the window (forward) where there is no gap in the ACKed packets

Selective Repeat Packet Lineup



younger

Sender View

ACKed *In-Flight* *Ready to send* *Can't send*

0 1 2 3 4 5 6 7 8 9 10 11 12

window: 7

Out of window

Receiver View

0 1 2 3 4 5 6 7 8 9 10 11 12

not yet received

received out of order & ACKed

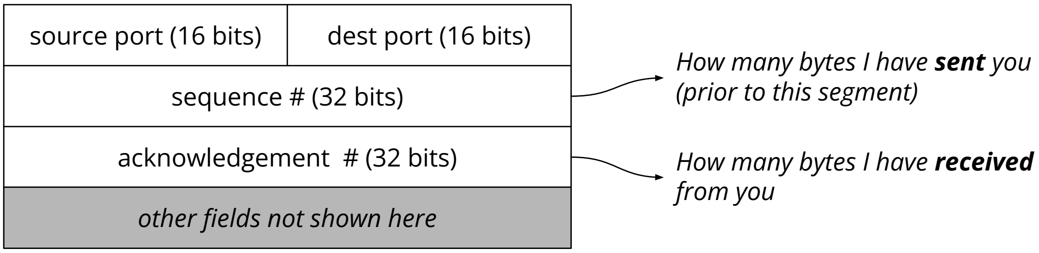
Selective Repeat Animation

TCP (Transmission Control Protocol)

From Go-Back-N/Selective Repeat to TCP

	Go-Back-N	**Selective Repeat**	**TCP**
Sequencing	Packet numbers	Packet numbers	Byte sequence numbers
Acknowledgment	Cumulative	Individual	Byte cumulative
Timer	One timer	Multiple timers	One timer
On Timeout	Resend all N packets	Resend only the packet associated with timeout	Resend only the segment that caused timeout (inferred from **last byte acknowledge**)

TCP Header



source port (16 bits)

dest port (16 bits)

sequence # (32 bits)

acknowledgement # (32 bits)

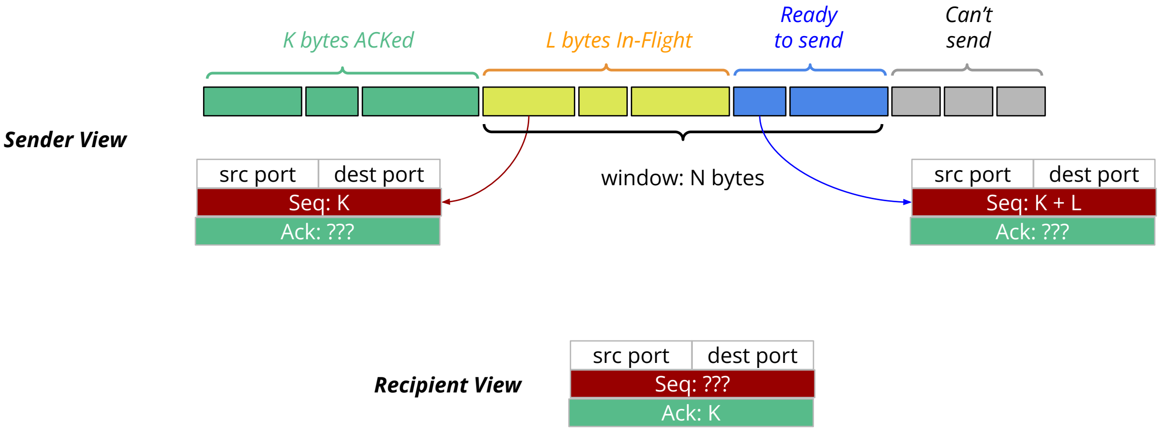
other fields not shown here

*How many bytes I have **sent** you (prior to this segment)*

*How many bytes I have **received** from you*

Each sender/receiver maintains these two variables

TCP Sequence # and Ack



K bytes ACKed

L bytes In-Flight

Ready to send

Can’t send

Sender View

window: N bytes

src port | dest port

Seq: K

Ack: ???

src port | dest port

Seq: K + L

Ack: ???

Recipient View

src port | dest port

Seq: ???

Ack: K

TCP (Sequence & Acknowledgement)

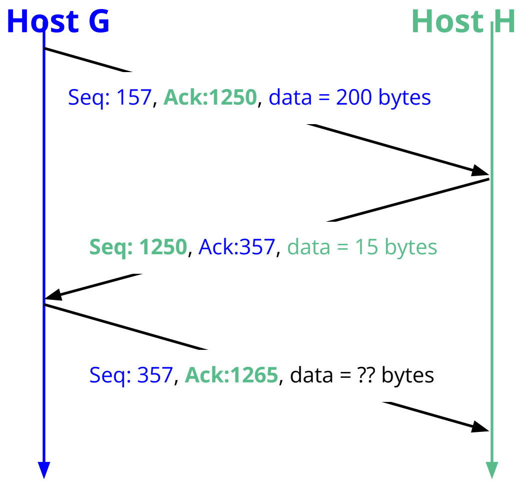
- Bytes in the payload are numbered sequentially from 0
  - During the handshake step both parties exchange a “phantom byte”, so the first byte in the actual application payload is byte #1
- Each TCP segment include both SEQ and ACK numbers
- SEQ # is the sequence number of the FIRST byte sent in the **current payload**
  - **SEQ # also indicates “how many bytes I have sent to you” (prior to this packet)**
- ACK # is the sequence number of the LAST byte received up to and including the **previous payload**
  - **“How many bytes I have received from you”**

Example TCP SEQ & ACK (Ideal Response)

Assume

- G already sent 157 bytes (and ACKed by H)
- H already sent 1250 bytes (and ACKed by G)

- G is about to send 200 bytes and in response H will send 15 bytes



Host G

Host H

Seq: 157, **Ack:1250**, data = 200 bytes

Seq: 1250, Ack:357, data = 15 bytes

Seq: 357, **Ack:1265**, data = ?? bytes

TCP Header

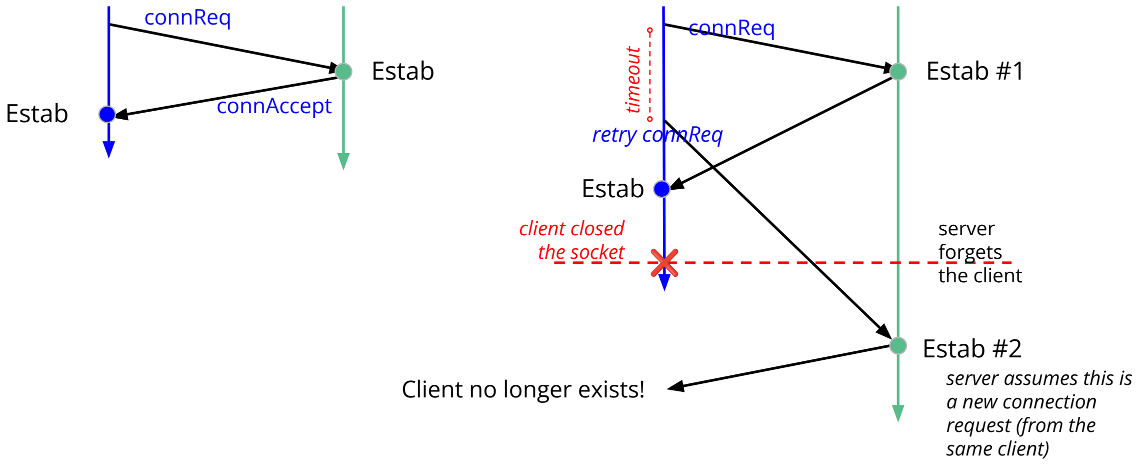
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|E|R|S|F|                               |
| Offset| Reserved  |R|C|O|S|Y|I|            Window             |
|       |           |G|K|L|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

          TCP Header Format (RFC 761, Jan 1980)
```

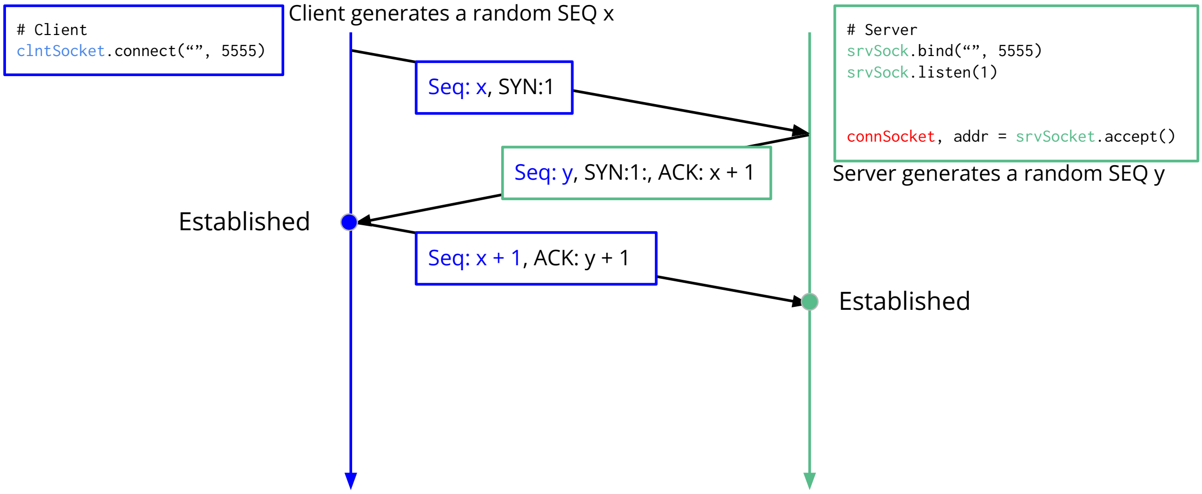
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Rsrvd |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Options                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

          TCP Header Format (RFC 9293, Aug 2022)
```

2-Way Handshake



TCP 3-Way Handshake



```
# Client
clntSocket.connect(“”, 5555)
```

Client generates a random SEQ x

Seq: x, SYN:1

```
# Server
srvSock.bind(“”, 5555)
srvSock.listen(1)


connSocket, addr = srvSocket.accept()
```

Server generates a random SEQ y

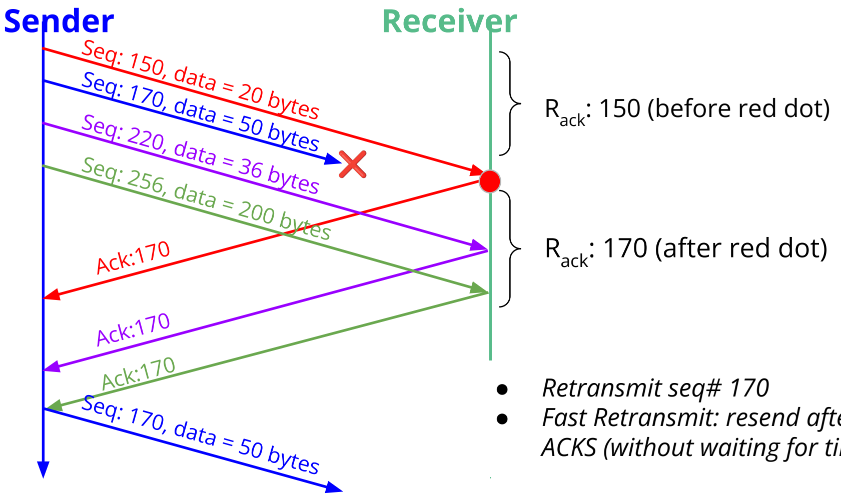
Seq: y, SYN:1:, ACK: x + 1

Established

Seq: x + 1, ACK: y + 1

Established

Packet Loss Induces Duplicate ACKs



Sender

Receiver

Seq: 150, data = 20 bytes

Seq: 170, data = 50 bytes

Seq: 220, data = 36 bytes

Seq: 256, data = 200 bytes

R_{ack}: 150 (before red dot)

R_{ack}: 170 (after red dot)

Ack:170

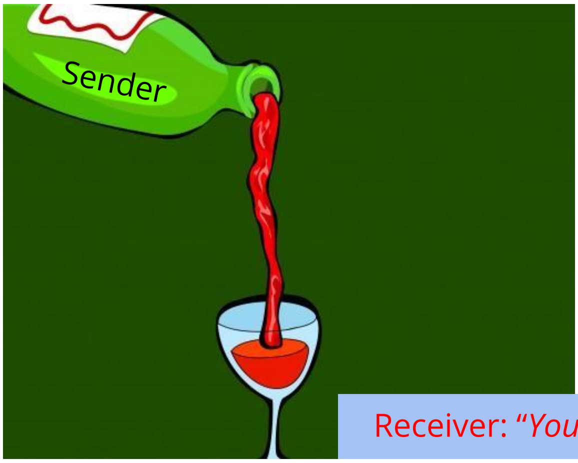
Ack:170

Ack:170

Seq: 170, data = 50 bytes

- *Retransmit seq# 170*
- *Fast Retransmit: resend after 3 duplicate ACKS (without waiting for timer timeout)*

[TCP] Flow Control



Receiver: *“You are giving me too much”*

[TCP] Congestion [Control]

(Grand) River Water Level After Snow Melt

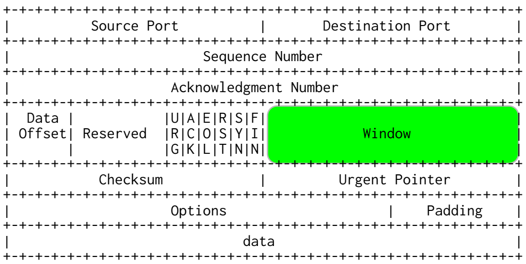
Flow Control vs. Congestion Control

- **Avoid overloading a receiver**
  - The receiver tells the sender how much buffer space is available to receive data
  - TCP: “Receiver Window” (RWND)
- Local issue between a **single sender** and a **single receiver**
  - Easier to resolve
- Issue is detected/prevented by the receiver, and the sender has make necessary adjustments
- Symptom: (larger) packet loss at the receiver

- Avoid overloading the network
- **Too many senders sending too much data too fast**
- Global issue that requires cooperation among participating **hosts** and **routers** in the network
  - Harder to resolve
  - Involve the **Network Layer**
- Issue is detected/prevent by the senders lowering the push/send rate
- Also involves **multiple senders** and **multiple receivers**
- Symptoms:
  - Long delays (long queue time in routers)
  - Packet loss (buffer overflow at routers)

TCP Flow Control

- TCP Header includes the “Receiver Window” field that indicates the size of the receiving buffer on the recipient side
- On receiving this information, the sender should adjust its window size (max bytes allowed in all the in-flight packets)



```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|E|R|S|F|                               |
| Offset| Reserved  |R|C|O|S|Y|I|            Window             |
|       |           |G|K|L|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

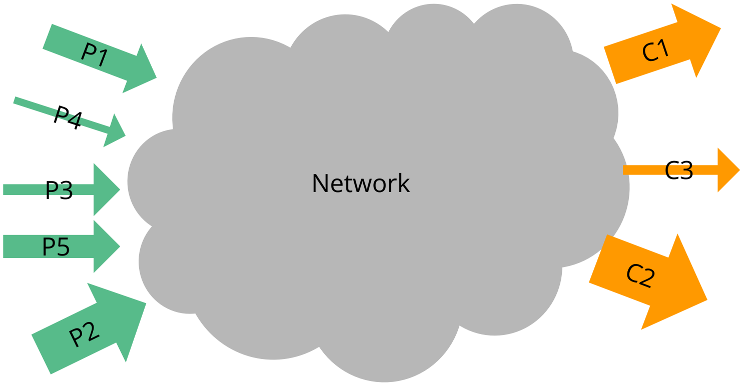
Congestion?

Roads have (limited) carrying capacity (<u>cars/minute</u>), so do network links.

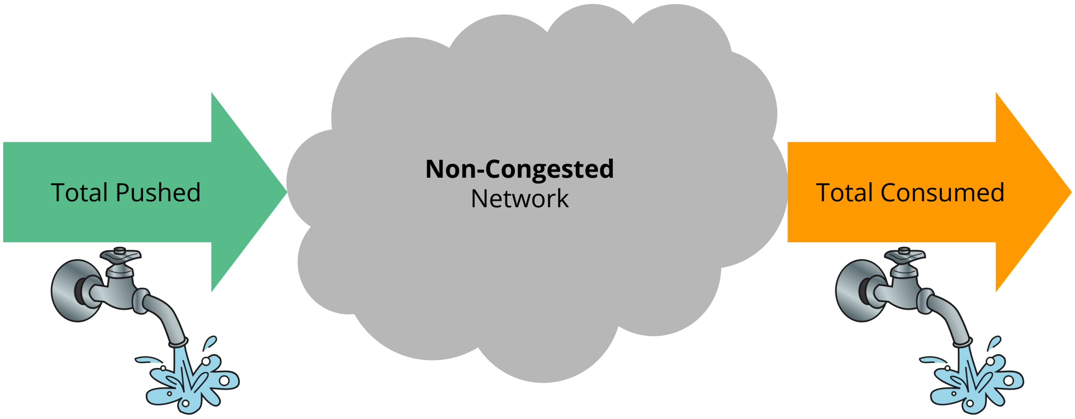
If the number of cars (bits) exceeds this capacity, we experience traffic congestion

How do you detect (& measure) traffic congestion?

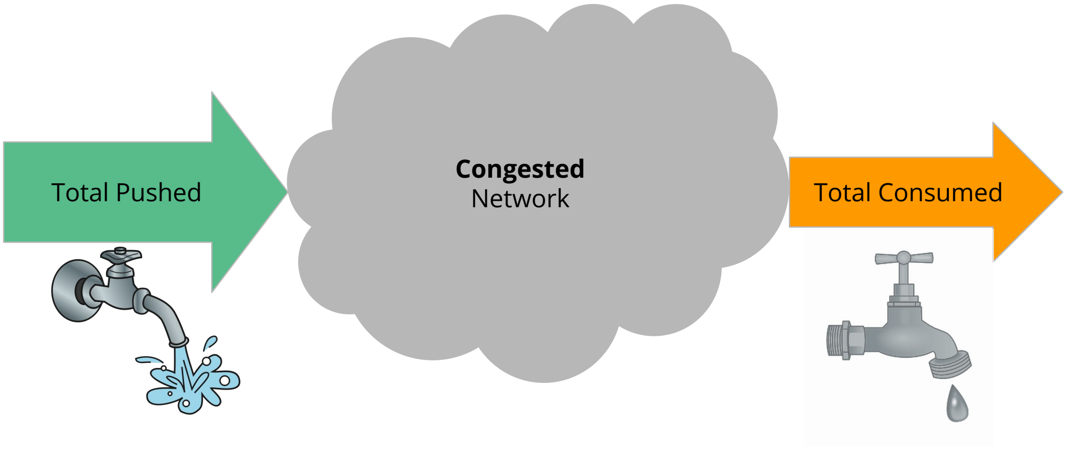
Network: Bits Pushed & Bits Consumed



Network: Bits Pushed & Consumed (Aggregate)



Network: Bits Pushed & Consumed (Aggregate)



Total Pushed

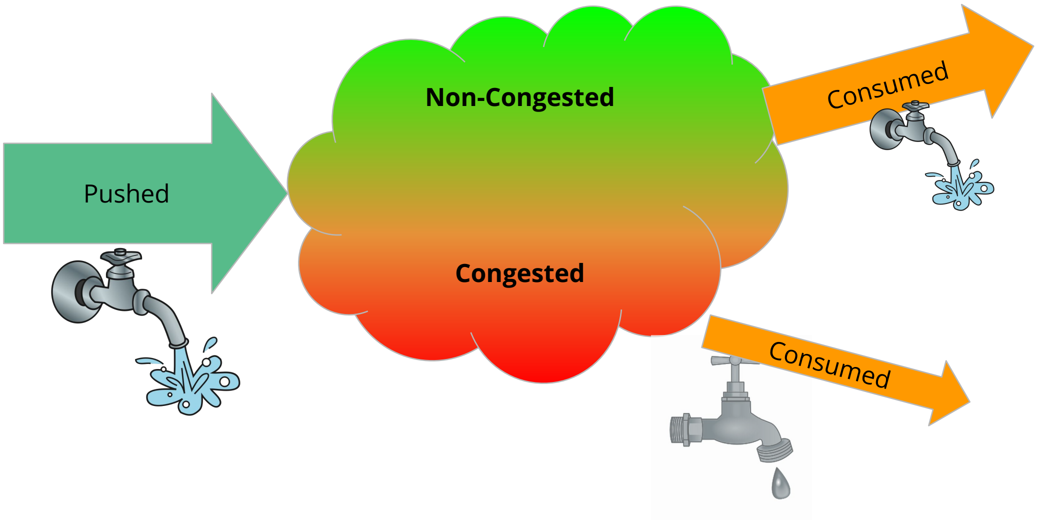
Congested
Network

Total Consumed

How to measure congestion (collectively)

- **Can't measure** congestion by **the amount of data** in the network
  - Must **measure the rate** at which these data are transported
    - 1000 cars on a 3-lane highway
    - 1000 cars on the same highway (but 2 lanes closed)
- Assuming the link carrying capacity is (collectively) R bits/sec:
  - All the senders (collectively) can push bits at the rate at most R bits/sec
  - All the recipients (collectively) can consume bits at the rate at most R bits/sec

Network: Bits Pushed & Consumed



Congestion & Router Buffer Capacity

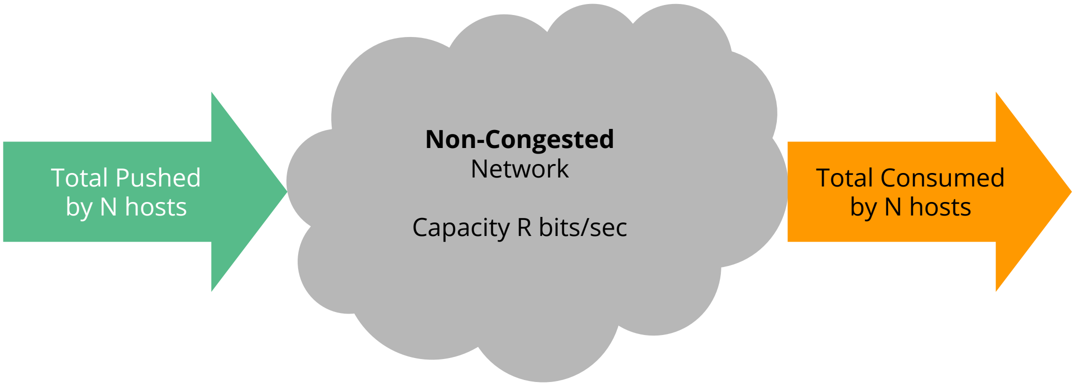
Rate of bits pushed ≪ Network Capacity

	Packet Lost	**Packet Delay**
Infinite Buffer	No	Short
Finite Buffer	No	Short

Rate of bits pushed ≈ Network Capacity

	Packet Lost	**Packet Delay**
Infinite Buffer	No	Long
Finite Buffer	Yes (High)	Long

Network: N hosts Push & N hosts Consume



Total Pushed by N hosts

Non-Congested Network

Capacity R bits/sec

Total Consumed by N hosts

“Fair” push rate per host-pair = R/N

How does a sender measure its own bit push rate?

Network Resource Sharing & Congestion

- Assume we have 2N hosts making up **N** sender-receiver pairs
- The collective carrying capacity of the network is **R** bits/seconds
- If all the hosts are *equally active pushing bits*, each sender-receiver pair can push/consume bits at most **R/N** bits/seconds
- When a sender-receiver pair exchange bits way above **R/N** bits/sec, other sender-receiver pairs will suffer more packet loss, their bit *throughput will be significantly low* (**Congestion Collapse**)

Congestion Control (Non TCP specific)

- Opt #1: End-to-End (*think of it as* "Host-to-Host")
  - Senders do not get warning from the network (routers)
  - The senders themselves must infer congestion by **observing packet loss** (multiple ACKS of the same sequence)
- Opt #2: Network-Assisted
  - senders and/or receivers get direct feedback from the routers. How?
    - Each router knows how busy is the traffic passing through it and who the senders/receivers are
    - **Each router may be able to calculate the desired sending rate**
- In both options, the corrective action is for the senders to dial down bit push rate

Related RFCs

- RFC 793 (Sep 1981): Initial TCP Specification
- RFC 1122 (Oct 1989): Relationship of TCP to other protocols/layers
- RFC 2018 (Oct 1996): TCP Selective ACK
- RFC 5681 (Sep 2009): TCP Congestion Control
- RFC 7323 (Sep 2014): High-Performance TCP

TCP Congestion Control

TCP Congestion Control

	Classic	Delayed-Based (Time-Based)
How to detect congestion?	Observe packet loss	Observe Round-Trip Time
How to reduce congestion?	Sender decreases pipeline size (amount of in-flight bytes)	Sender decreases pipeline size (amount of in-flight bytes)

In a congested network:

	Packet Lost	**Packet Delay**
Infinite Buffer	No	Long
Finite Buffer	Yes (High)	Long

TCP Congestion Control

- Classic
  - Senders **gradually increase** their sending rate until packet loss is observed
  - When packet loss is observed, senders **quickly decrease** sending rate
  - Adjusting “sending rate” = adjust window/pipeline size (max bytes in-flight)
  - Implementation: **continuously observe packet lost (duplicate ACKs)**
- Time-Based
  - Require additional information/assistance from the Network Layer (IP Layer)
  - Time-based (Delay-Based)
  - Implementation: Routers **continuously calculate round-trip time (RTT)**

TCP Congestion Control
Classic: Observe Packet Loss

TCP Classic Congestion Control

- Senders probe the network carrying capacity by
  - Gradually increasing sending rate until it senses packet loss then quickly decreasing sending rate
- During the steady portion of the connection
  - **AIMD**: Additive Increase Multiplicative Decrease = “Add 1, Divide by 2”
    - **Increase pipe line size by 1** each time ACK is received
    - **Halve the pipe line size** each time packet loss is observed (repeated ACK from receiver)
- During initial stage of connection
  - **Double pipe line size** until pipe line size is 50% achievable max rate so far, increase by one thereafter

TCP Classic + Improvement #1: Cubic

- Using AIMD (additive/linear increase) the sending rate ramps up too slowly.
- Improvement: use cubic increase to reach the max-sending-rate faster
  - t_k is the desired future time to reach W_{max}
  - Pipeline size is determined by a cubic function:

$$\text{PipelineSize}(t) = W_{\max} + (t - t_k)^3$$

Desmos Graph

    - Larger increase when current time is further away from t_k
    - Smaller increase when we are approaching t_k

TCP Congestion Control
Time-Based (Delay-Based): Observe RTT

RTT Estimate vs. Actual RTT

Travel time to campus

Day	Actual Travel Time
Mar 7	20 minutes
Mar 8	32 minutes
Mar 9	25 minutes

On the morning of Mar 10, what is your estimate of travel time?

RTT Estimate vs. Actual RTT: Update Daily Estimate

Travel time to campus

Day	Actual Travel Time	Daily Estimate	How Much You're Off
		50 minutes (initial wrong estimate)	unknown
Mar 7	20 minutes	(0.8)(20) + (0.2)(50) = 26	+6 (overestimate)
Mar 8	32 minutes	(0.8)(32) + (0.2)(26) = 30.8	-1.2 (underestimate)
Mar 9	25 minutes	(0.8)(25) + (0.2)(30.8) = 26.16	+1.16 (overestimate)

On the morning of Mar 10, you expect 26.16 minutes of travel time.
But how much off is 26.16 from your actual travel?

RTT Estimate vs. Actual RTT: Update Daily Estimate

Travel time to campus

Day	Actual Travel Time	Daily Travel Estimate (80%, 20%)	How Much You're Off	Daily Off Estimate (75%,25%)
		50 minutes (initial estimate)	unknown	10 minutes
Mar 7	20	(0.8)(20) + (0.2)(50) = 26	+6 (over)	(0.75)(6) + (0.25)(10) = 7
Mar 8	32	(0.8)(32) + (0.2)(26) = 30.8	-1.2 (under)	(0.75)(**1.2**) + (0.25)(7) = 2.65
Mar 9	25	(0.8)(25) + (0.2)(30.8) = 26.16	+1.16 (over)	(0.75)(1.16) + (0.25)(2.65) = 1.53

On the morning of Mar 10, you expect 26.16 minutes of travel time, and expect your estimate will be off by 1.53 minutes

TCP Congestion Control (Time-Based)

- AIMD + Cubic may "probe too far", causing packet loss
- Objective of Time-Based is
  - Avoid inducing/forcing packet loss
- General ideal
  - Periodically compute the current sending rate from the amount of bytes successfully pushed (and ACK'd) and their RTT
  - Lowest RTT (hence highest sending rate) ⇒ Optimal (uncongested) sending rate R_{uc}
- Warning: The textbook calls this "**Delay-based TCP Congestion Ctrl**"

TCP Congestion Control (Time-Based)

Recalculate current sending rate (R) periodically:

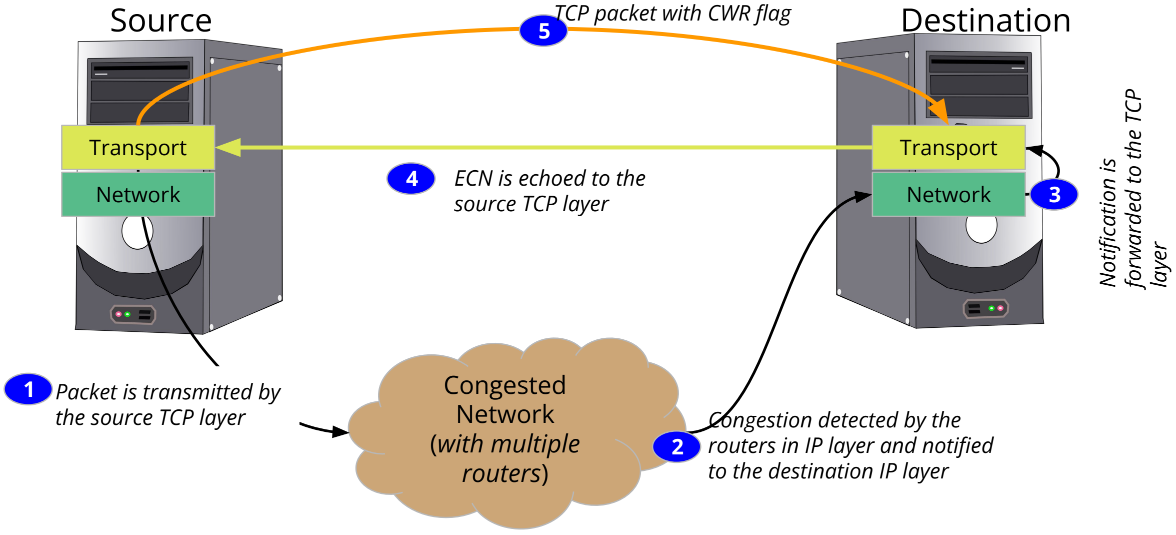
- If the current sending rate R is "**very close to**" the optimal rate R_{uc} ⇒ the network is not congested (yes), **window size can be increased**
- If the current sending rate R is "**far below**" the optimal rate R_{uc} ⇒ the network may be congested, **window size should be decreased**

TCP Congestion Control
ECN: Explicit Congestion Notification
(RFC3168, Sep 2001)

TCP Congestion Control: ECN

- Network-Assisted, i.e. require assistance from the Network Layer (IP Protocol)
- Extra bits in the IP packets to notify congestion to the IP layer destination host
- The TCP layer at the destination host relays the notification to the source host with a ECN Echo (ECE) bit in the TCP packet
- In response, the TCP layer on the source host responds with a CWR (Congestion Windows Reduced) bit in the TCP packet

TCP Explicit Congestion Notif: TCP + IP Layers



TCP packet with CWR flag

ECN is echoed to the source TCP layer

Notification is forwarded to the TCP layer

Packet is transmitted by the source TCP layer

Congestion detected by the routers in IP layer and notified to the destination IP layer