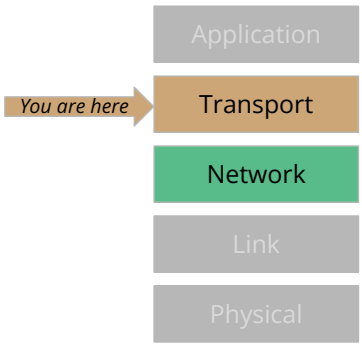
Ch03: Transport Layer

Layered Structure (Recall)



Application Layer:

- Exchange **messages** between **applications**

Transport Layer

- move data (**block of bytes**) 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**

Package Delivery

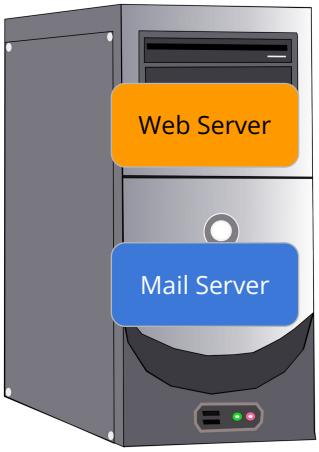


Application Layer:
- 5000 books
- 350 pounds of candy

Transport Layer:
- Tons of boxes

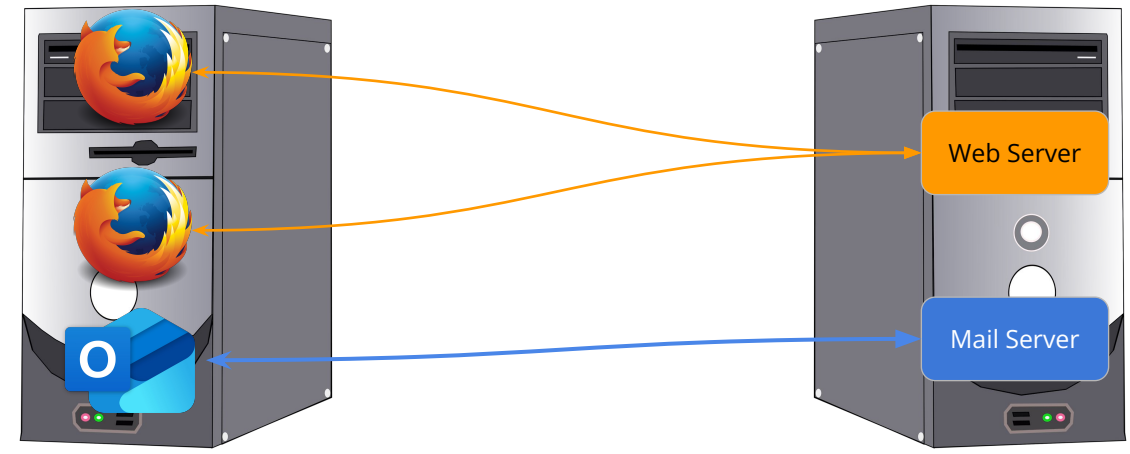
N Processes in One Host

IP: 231.15.33.86

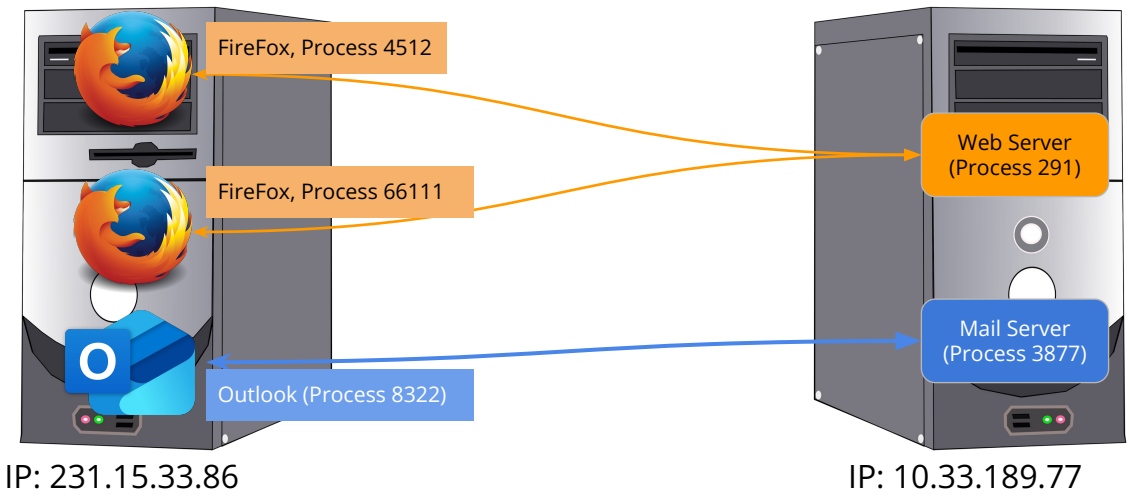


IP: 10.33.189.77

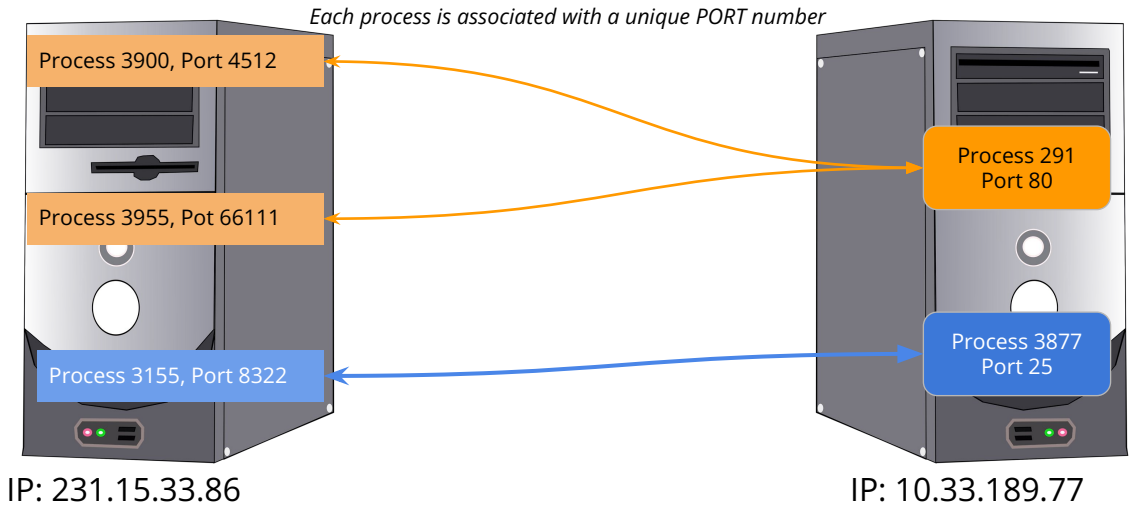
Application Layer Perspective: App to App



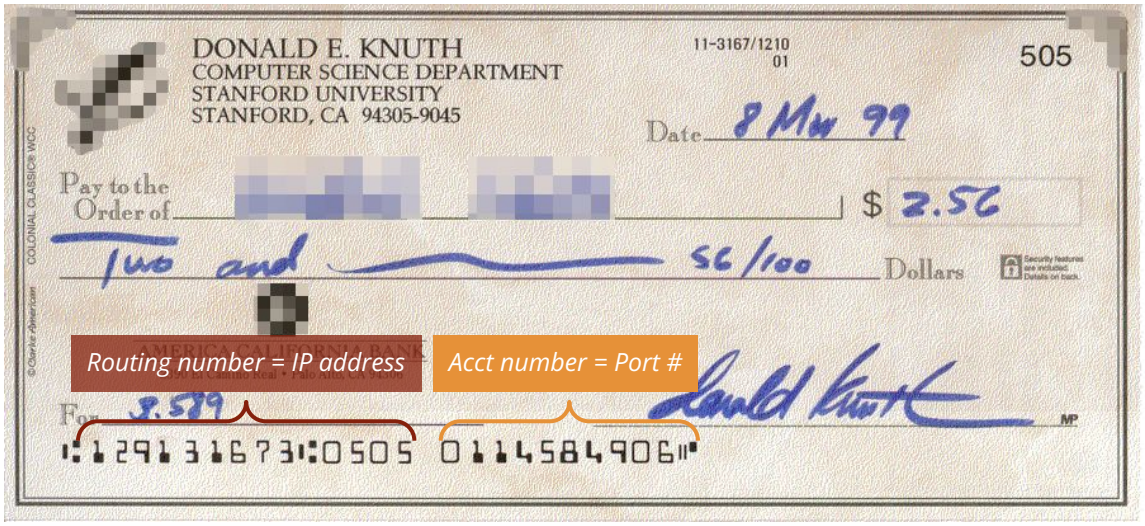
Transport Layer Perspective



Transport Layer Perspective: Process to Process



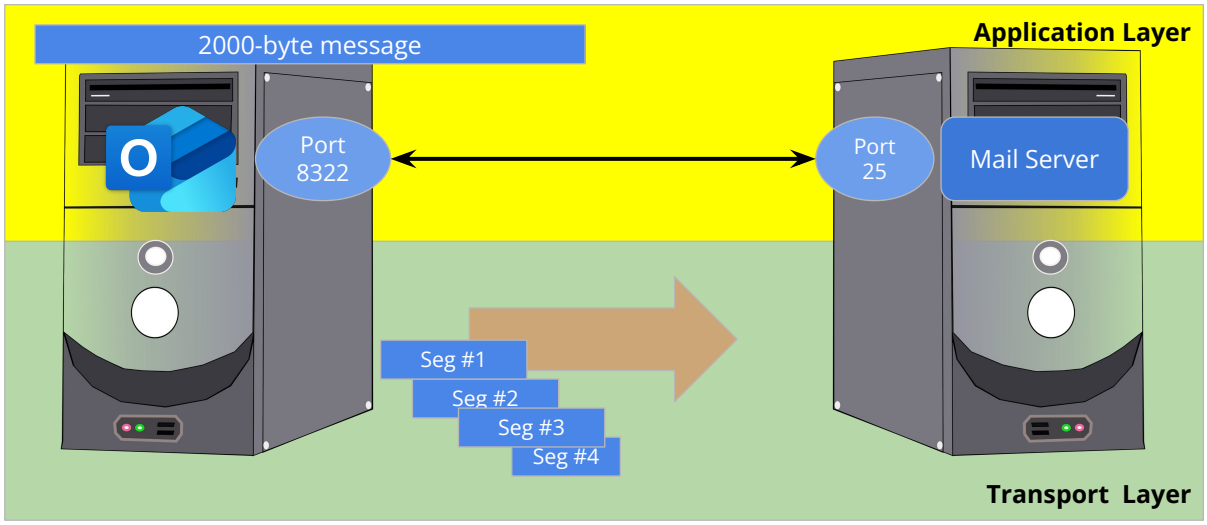
Bank Routing Number & Account Number



Socket Port Identifier

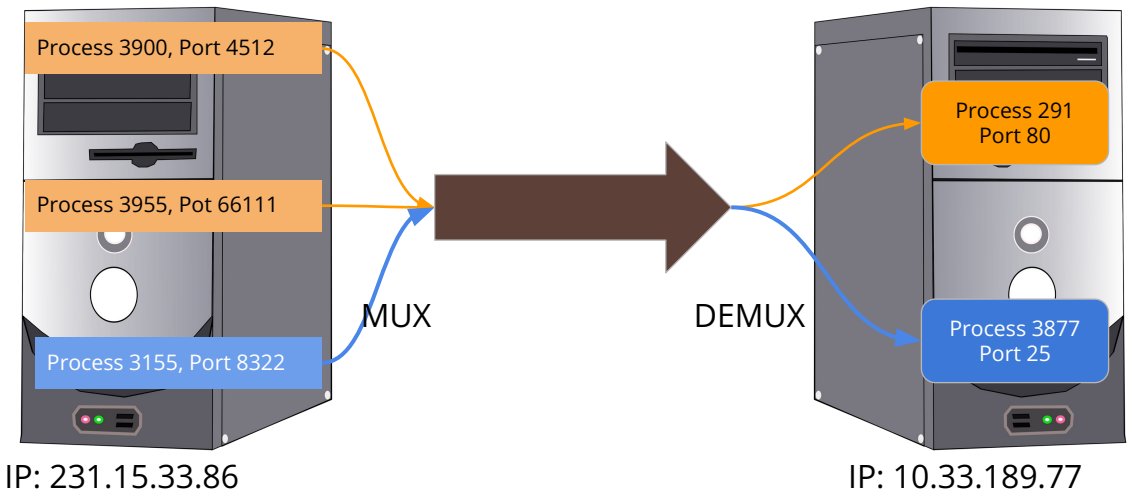
Bank Account Number	**Socket Port Number**
Banks have unique **routing number**	**Hosts** have unique **IP address**
Unique (within a **bank**)	Unique (within a **host**)
One **person** can open multiple **accounts**, each **account** is assigned a unique **number**	One **process** can open multiple **sockets**, each socket is assigned a unique **port number**
A monetary transfer is uniquely identified by a 4-tuple 1. Sender **bank routing number** 2. Sender **account** number 3. Recipient **bank routing number** 4. Recipient **account** number	A data transfer is uniquely identified by a 4-tuple 1. Sender **host IP address** 2. Sender **port** number 3. Recipient **host IP address** 4. Recipient **port** number

App Data (Messages) vs. Segment



(De)Multiplexing
Mux - Demux

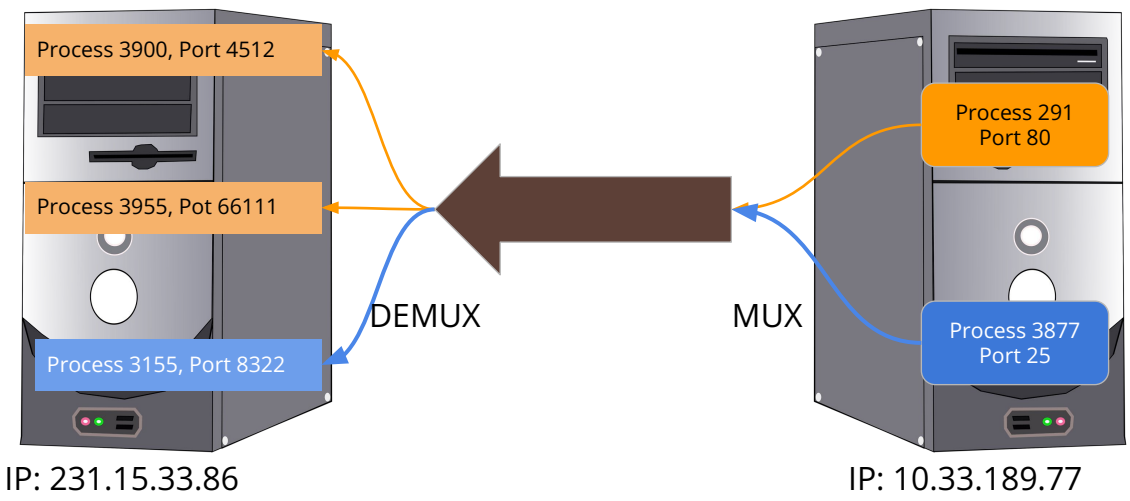
Transport Layer to Network Layer



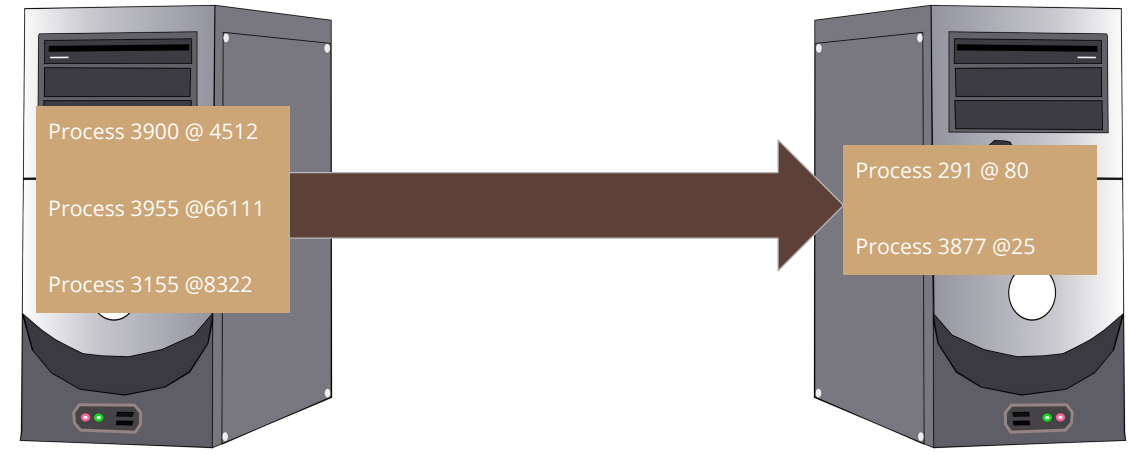
Demux: Unload and Dispatch to Multiple Recipients

*packages to the **same address***
*but different **apartment number***

Transport Layer to Network Layer



Network Layer: Host to Host



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 Using UDP Sockets

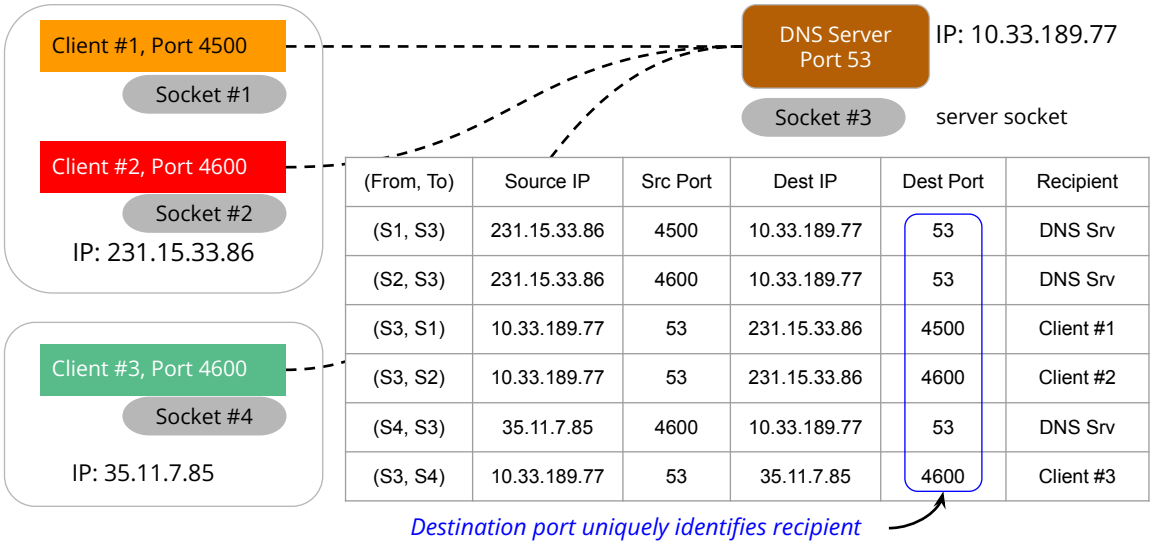
```
# Server Side
SERVER_PORT = 53  # DNS
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind("", SERVER_PORT)
while True:
  data, addr = serverSocket.recvfrom(____)
  # do work here
  serverSocket.sendto(_____)
```

```
# 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(____)
```

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

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

Communication Using TCP Sockets

```
# Client Side #1
SERVER_PORT = 7777
clientSocket = socket(AF_INET, SOCK_STREAM)
# port number of clientSocket is randomly
# assigned by the OS
clientSocket.connect("", SERVER_PORT)
# Do work here
clientSocket.close()
```

```
# Client Side #2
SERVER_PORT = 7777
clientSocket = socket(AF_INET, SOCK_STREAM)
# port number of clientSocket is randomly
# assigned by the OS
clientSocket.connect("", SERVER_PORT)
# Do work here
clientSocket.close()
```

```
# Server Side
SERVER_PORT = 7777
acceptSocket = socket(AF_INET, SOCK_STREAM)
acceptSocket.bind("", SERVER_PORT)
acceptSocket.listen(1)
while True:
  connectSocket, addr = acceptSocket.accept()
  # connectSocket will be assigned
  # a new port number (not 7777)
  #do work here
  connectSocket.close()
```

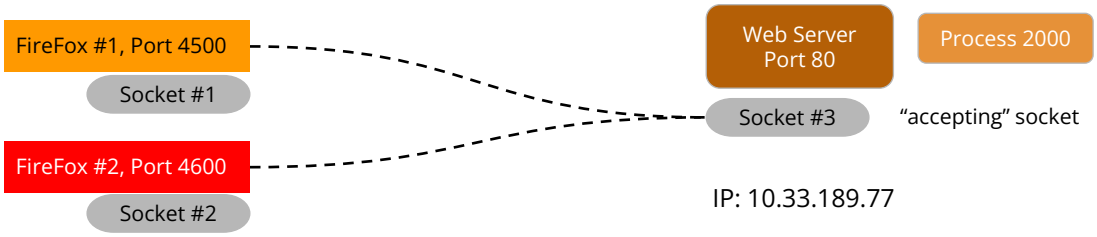
acceptSocket *and* ***connectSocket*** *are assigned two different PORT numbers*

Restaurant Dining Service

Receptionist (**welcomeSocket**)

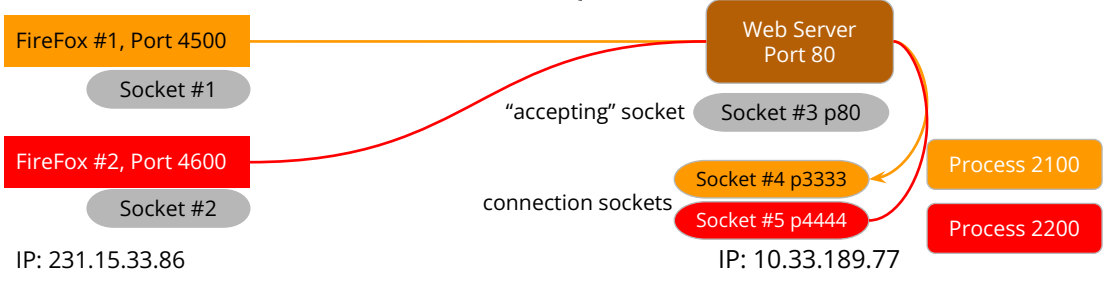
Waitress (**connectSocket**)

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.

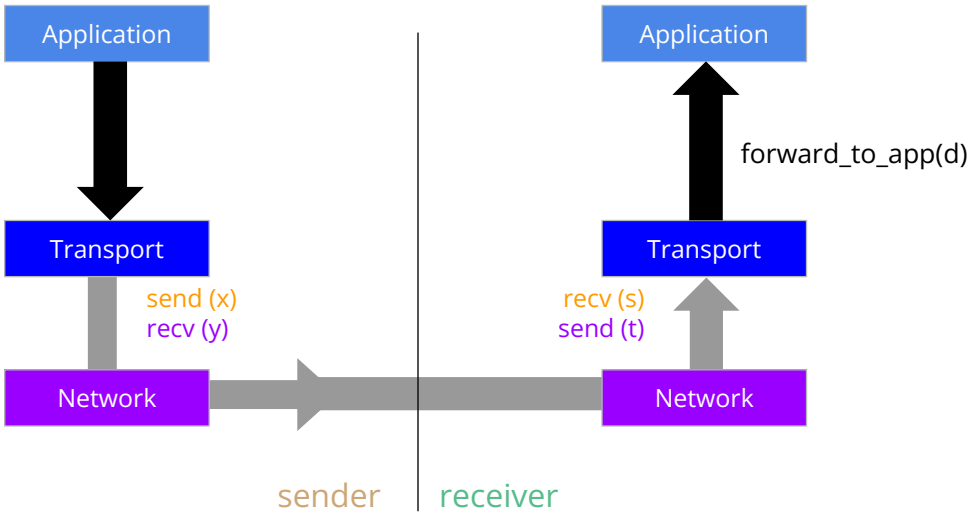
UDP reliability: "correctness" data (*if received*) is verified via checksum

Reliable Data Transfer

Expectations of “Reliability”

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

Layer Interactions: Application ⇔ Transport



Textbook: Reliable Data Transfer

My slides:

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

FSM Transition Between States

```
IF condition(s)
  action1
  action2
  action3
```

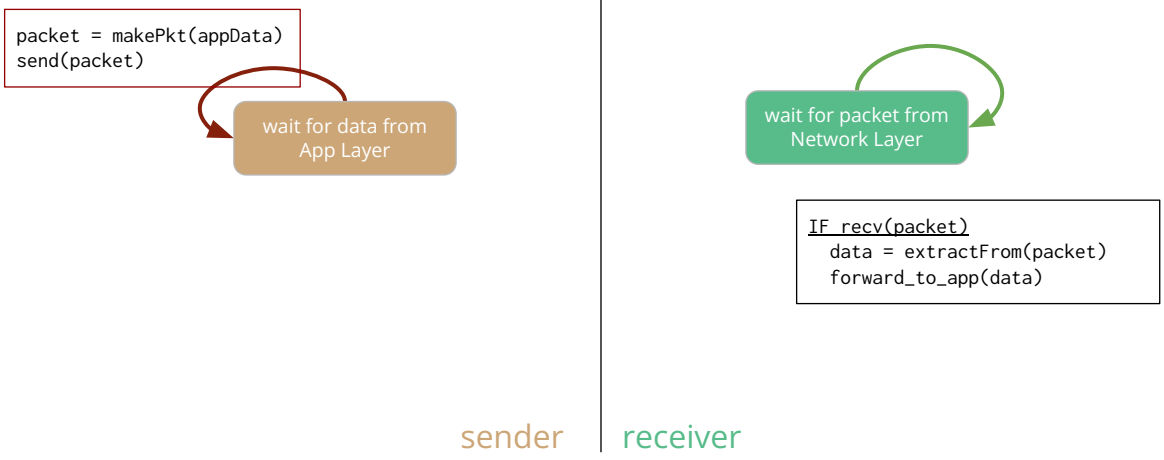
Conditions are underlined
(most) actions are indented

Finite State Machines for Reliable Transport

Version	Data Error	ACK/NAK Error	Improvement(s)
FSM 1.0	No	N/A	
FSM 2.0	Yes	N/A	
FSM 2.1	Yes	Yes	
FSM 2.2	Yes	ACK only	Without NAK
FSM 3.0	Yes / Loss	ACK only	Without NAK, Handle Timeout

FSM 1.0 Reliable Data Transport

(No Error Handling)



Data Errors

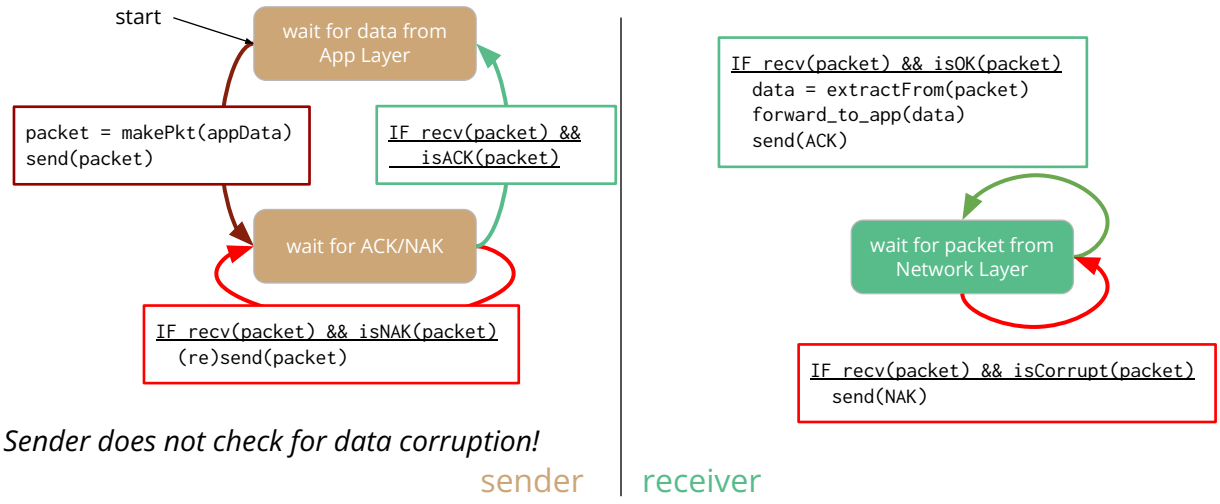
Possible sources of (byte) errors

- During transmission
- During propagation
- During reception

Error detection: byte checksum

Error notification: Negative Acknowledgment (NAK)

FSM 2.0 Data Transport With Data Errors



Sender does not check for data corruption!

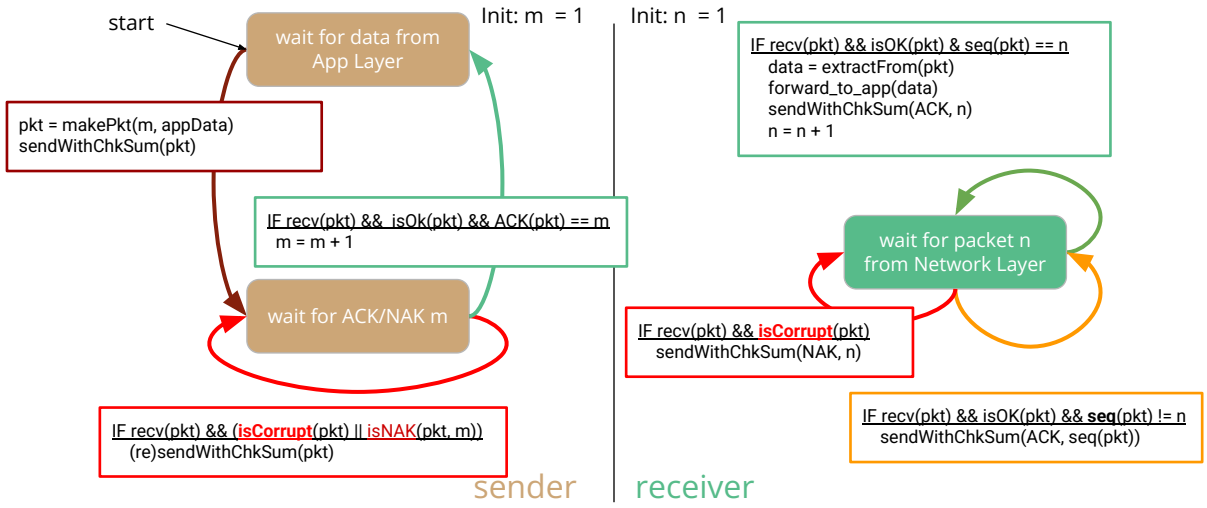
sender | receiver

FSM 2.0:What if ACK/NAK is corrupted?

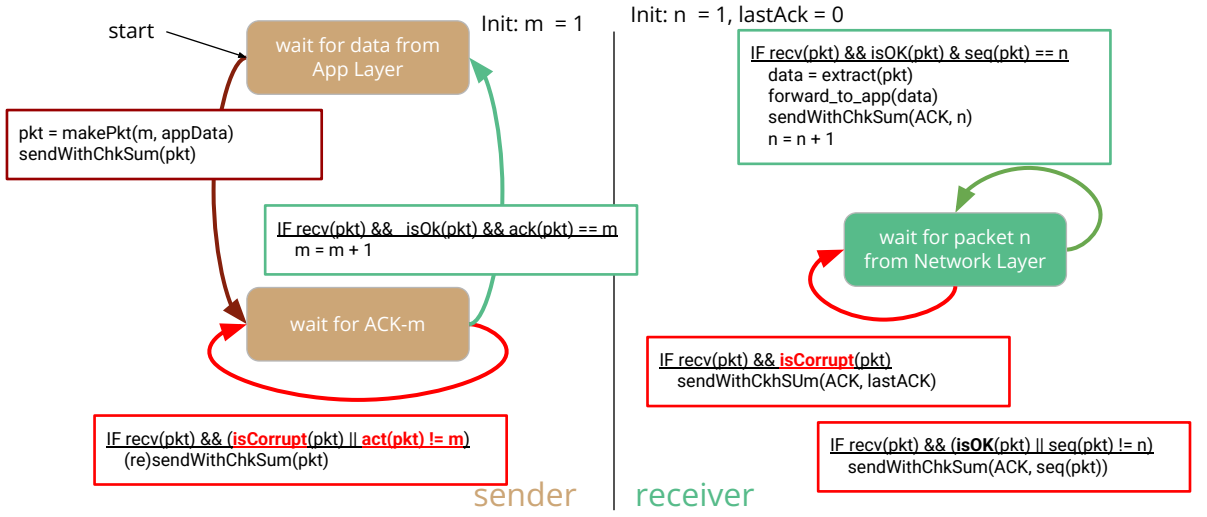
Error	Action	Side Effect #1	Side Effect #2
Data corrupted	Recipient sends NAK		
ACK corrupted	Sender re-sends packet	Recipient may received duplicate packets	Sender may receive duplicate ACKs
NAK corrupted	Sender re-sends packet		

Use sequence numbers for packets and ACK/NAK

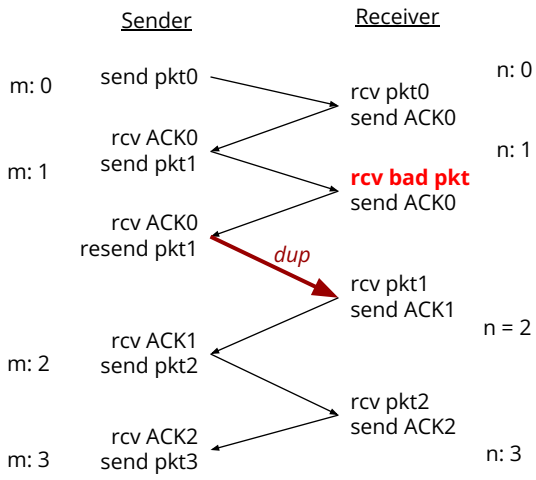
FSM 2.1 = FSM 2.0 with DATA & ACK/NAK errors

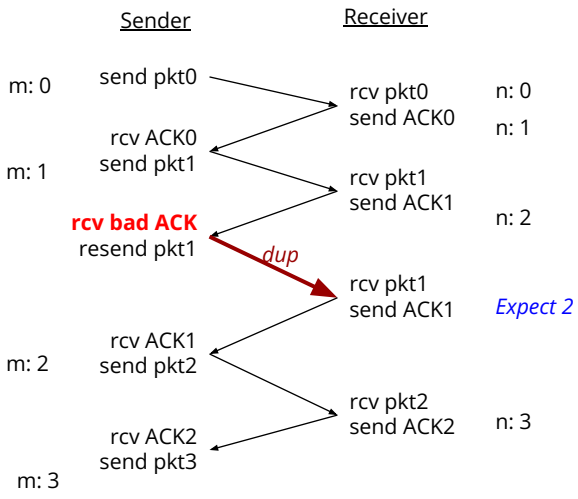


FSM 2.2 = FSM 2.1 without NAK

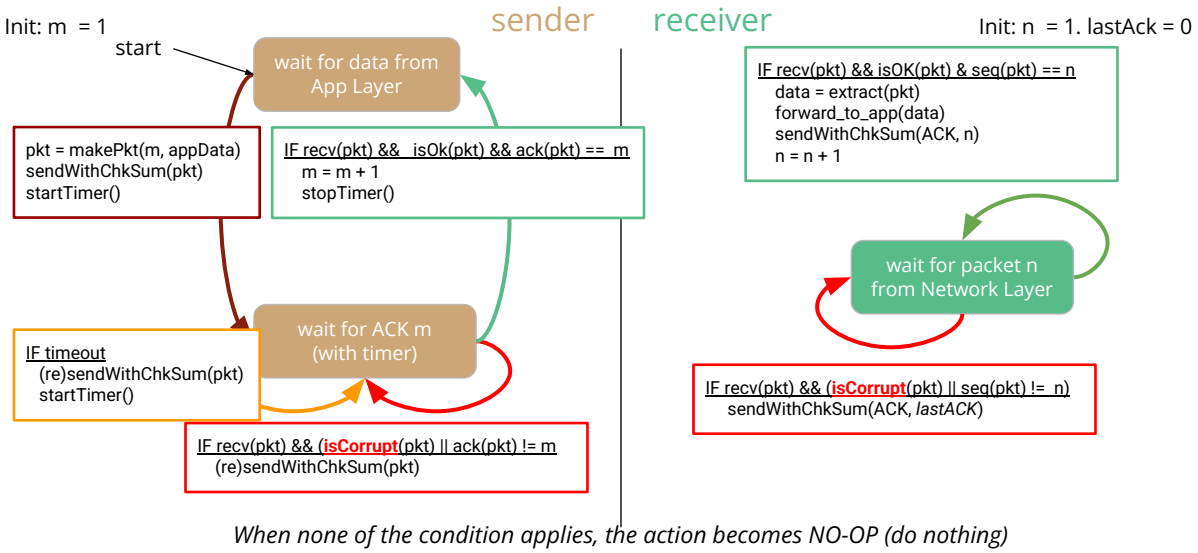


FSM 2.2 in action



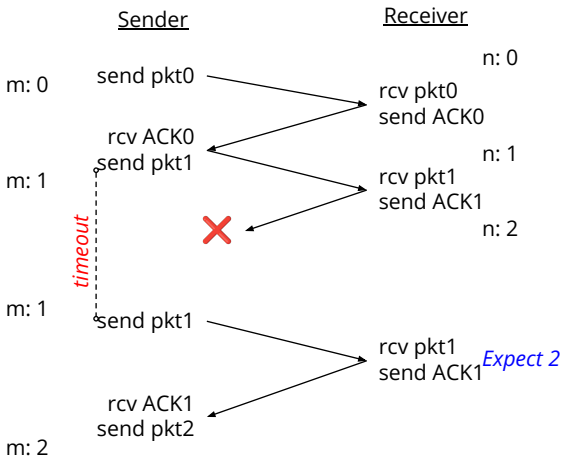


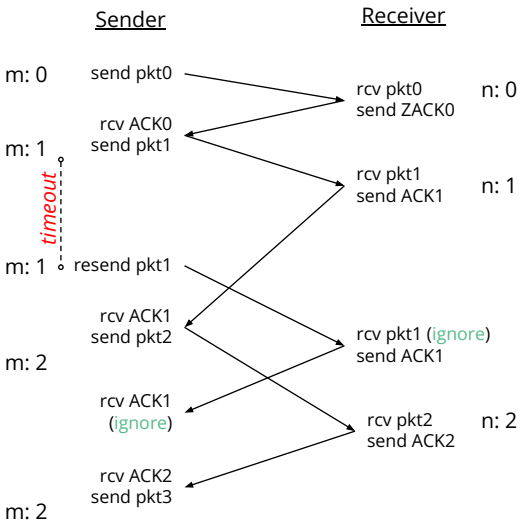
FSM 3.0 = FSM 2.2 with Sender TimeOut



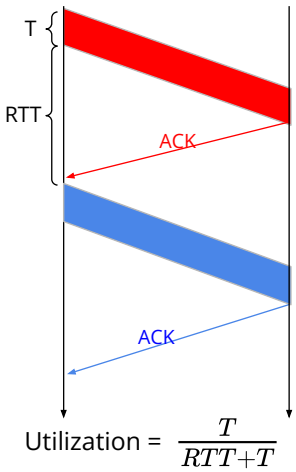
When none of the condition applies, the action becomes NO-OP (do nothing)

FSM 3.0 in action





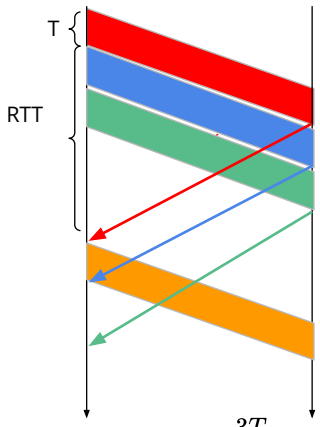
Stop & Wait for ACK



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

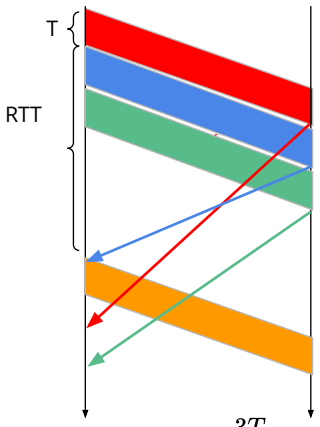
Utilization = $\frac{T}{RTT+T}$

Pipelined Transmission



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

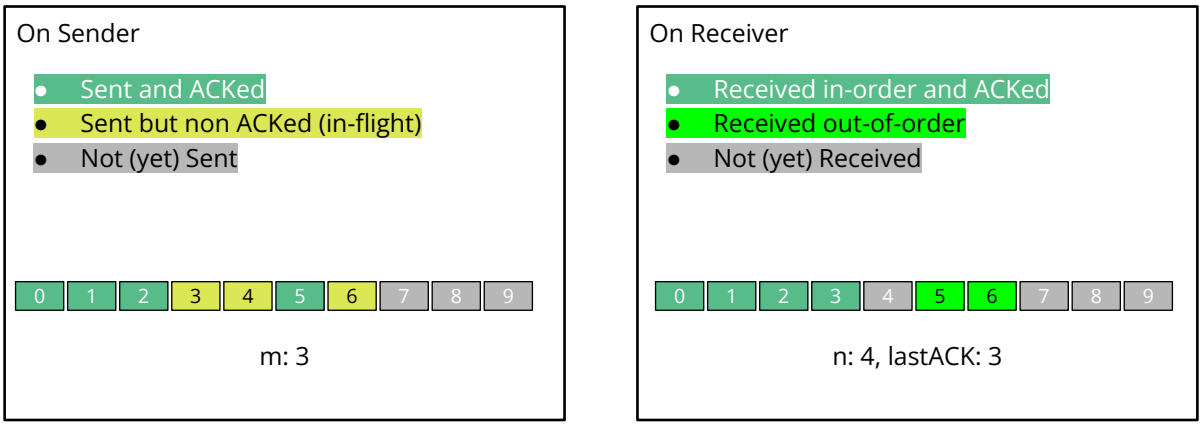
Utilization = $\frac{3T}{RTT+T}$



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

How long can we increase the pipeline?

Pipelined Packet Types



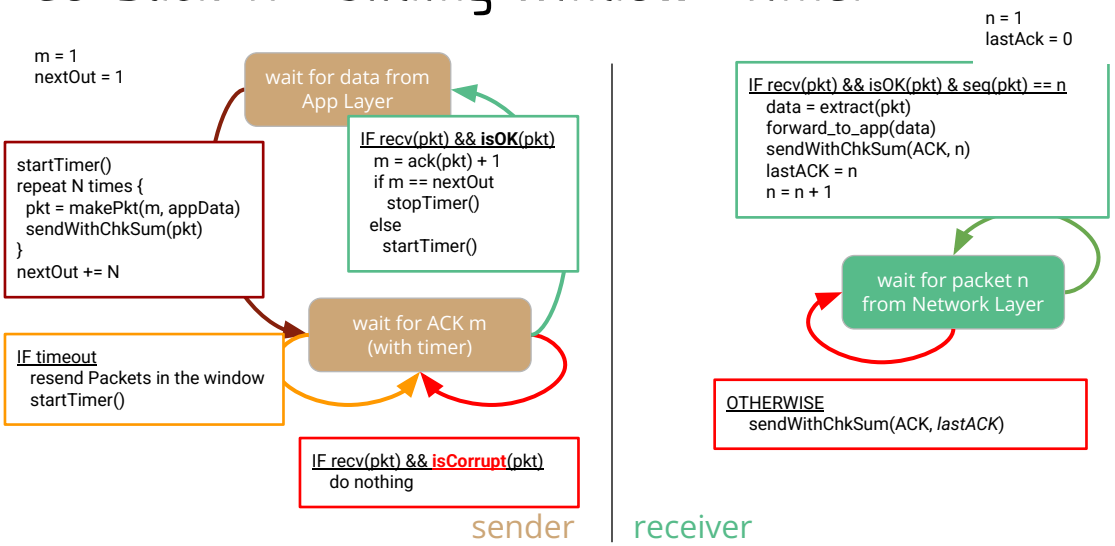
ACK3 is still in transit

Pipelined Packets: Implementation

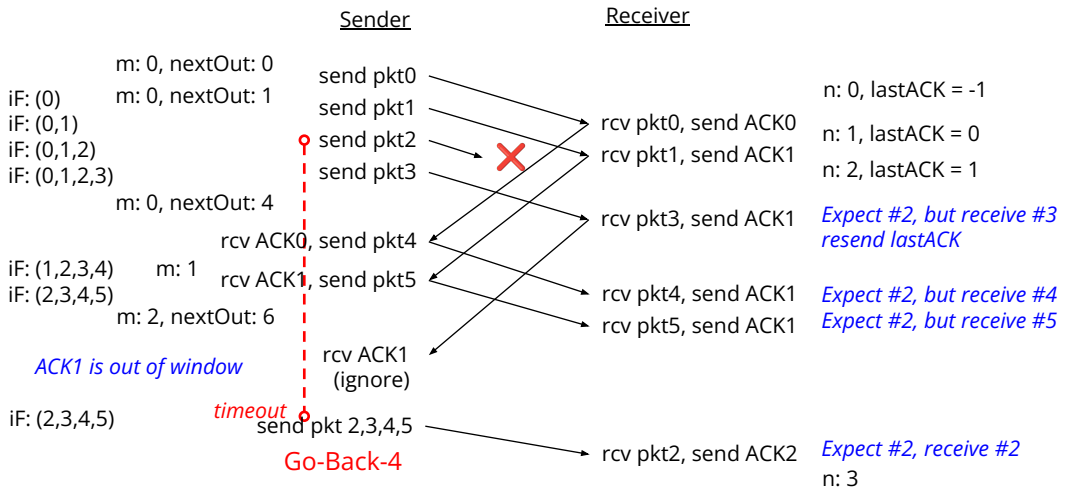
	Sender	Receiver
Go-Back-N	One timer set for the oldest in-flight packet. OnTimeout: resend all ("Go Back") N packets	Cumulative ACK
Selective Repeat	Multiple timers: one for each in-flight packet OnTimeout(**k**) resend only packet(**k**). "Selective"	Individual ACK

Go-Back-N = Pipeline + Improved FSM 3.0

Go-Back-N = Sliding Window + Timer



Go-Back-4 (in-flight: max 4 packets)



m ⇒ Oldest packet in flight

n ⇒ Youngest packet ACKed

Go-Back-N

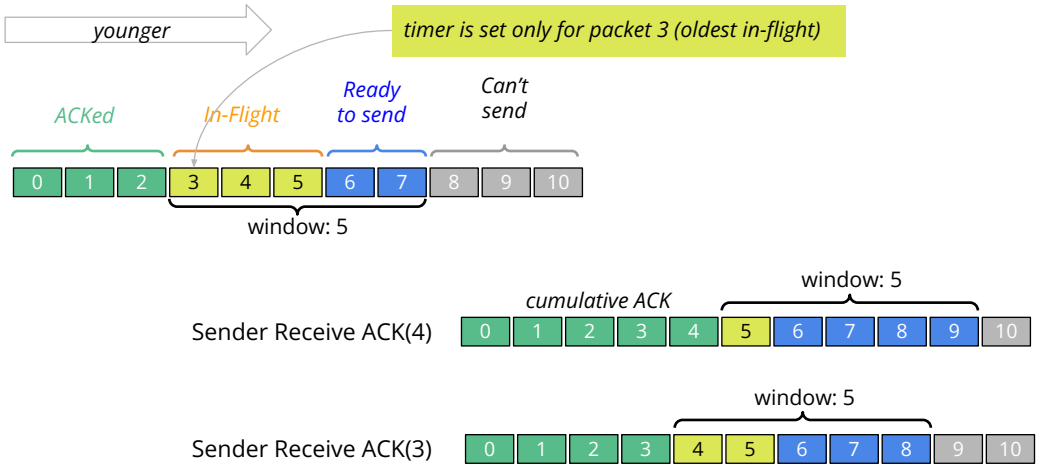
Sender

- 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

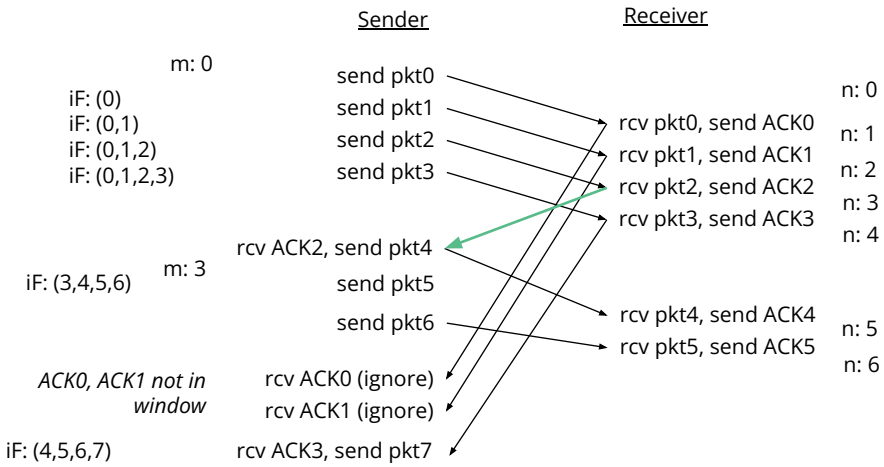
Receiver

- 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)

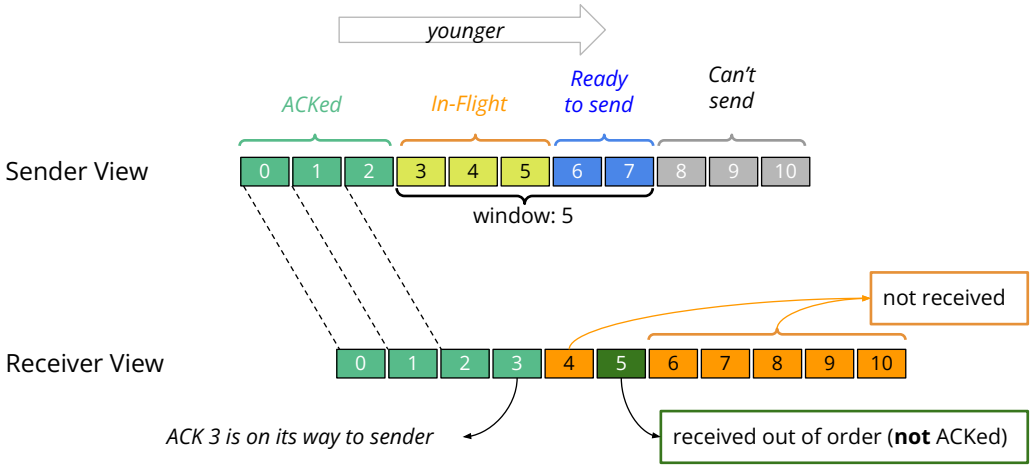
Go-Back-N: Cumulative ACK (Sender View)



Go-Back-N: Cumulative ACK



Go-Back-5 Packet Lineup



Go-Back-N Animation

Selective Repeat

Selective Repeat

Sender

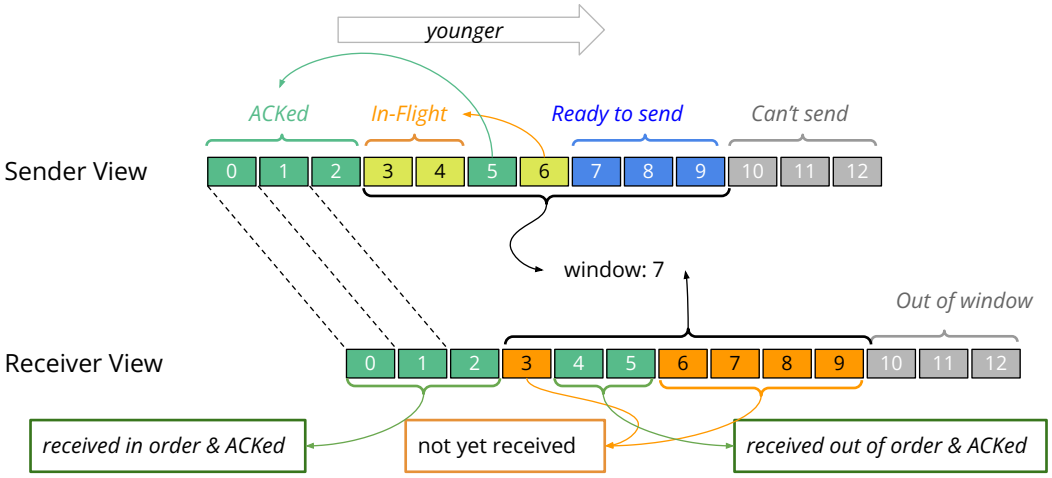
- A sliding window of size N
- Max N packets allowed to be in the pipeline (“in-flight”)
- Set one timer each in-flight packet
  - Timeout can be observed per packet
  - On timeout(**p**): resend only packet **p**
- Slide the window (forward) where there is no gap in the ACKed packets

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

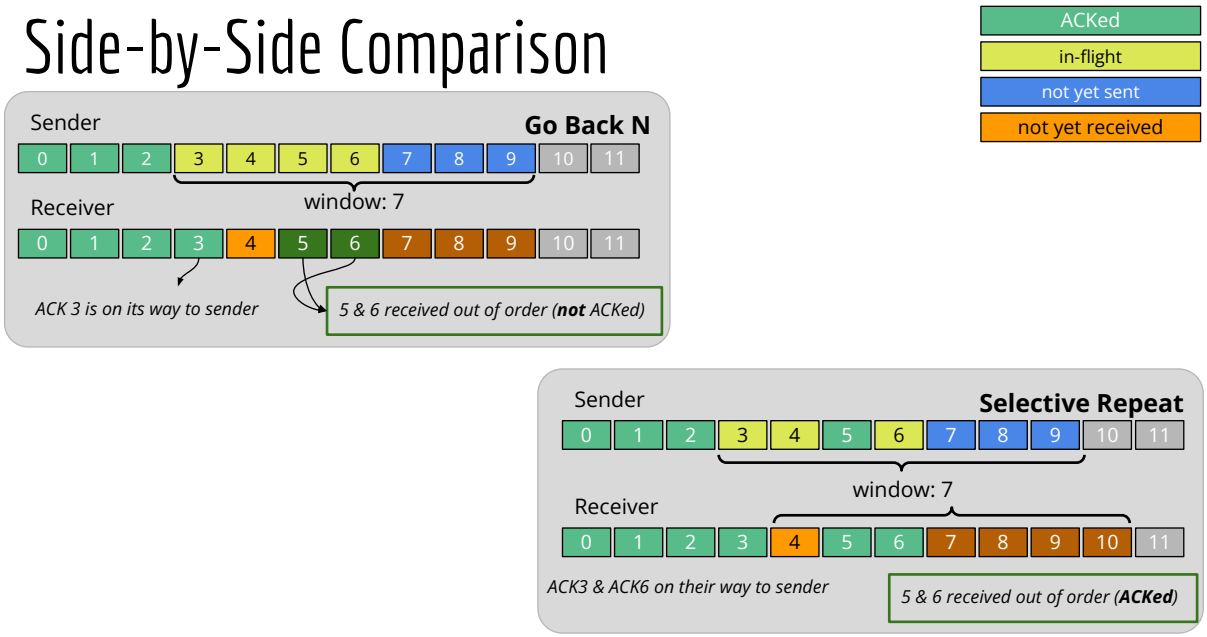
  Slide the window (forward) where there is no gap in the ACKed packets

Selective Repeat Packet Lineup

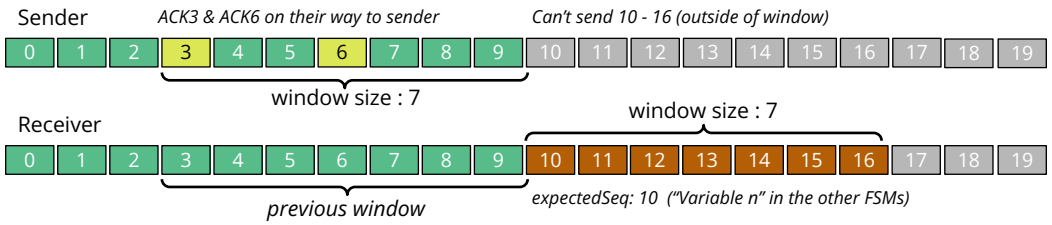


Selective Repeat Animation

Side-by-Side Comparison



Selective Repeat (Non-Overlapping Windows)



Receiver Action:

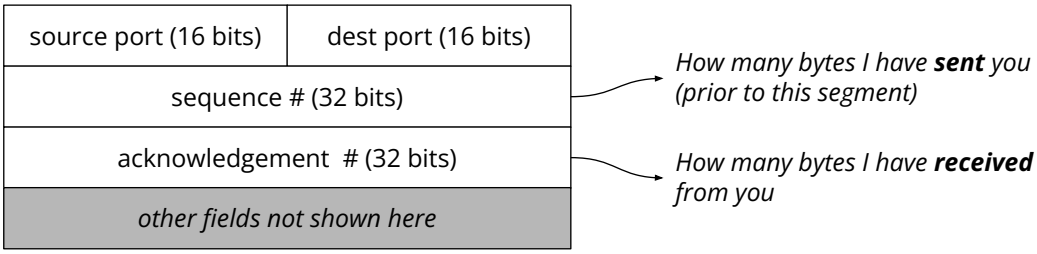
- On receiving packet in the "current window" (10 - 16)
  - Save the packet to buffer & send ACK for the packet sequence number
  - If the packets in the buffer are consecutively numbered (from expectedSeq)
    Forward all the packets to the Application Layer
    Shift the window
- On receiving packet in the "previous window" (3 - 9) or [expSeq - N, expSeq - 1]
  - Send ACK for the packet sequence number

TCP (Transmission Control Protocol)

From Go-Back-N/Selective Repeat to TCP

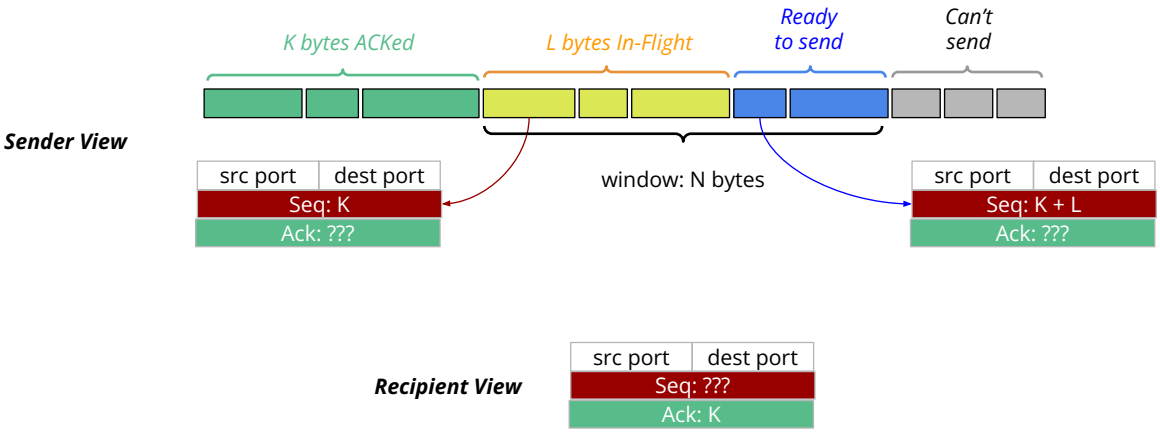
	Go-Back-N	Selective Repeat	TCP
Sequencing	Packet numbers	Packet numbers	Byte sequence numbers
Acknowledgment	Cumulative	Individual	Byte cumulative
Timer	One timer	Multiple timers	One timer
On Timeout	Resend all N packets	Resend only the packet associated with timeout	Resend only the segment that caused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

TCP Header



Each sender/receiver maintains these two variables

TCP Sequence # and Ack



TCP (Sequence & Acknowledgement)

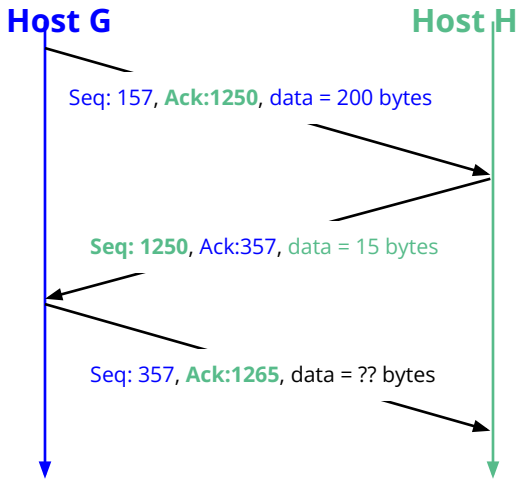
- Bytes in the payload are numbered sequentially from 0
  - During the handshake step both parties exchange a “phantom byte”, so the first byte in the actual application payload is byte #1
- Each TCP segment include both SEQ and ACK numbers
- SEQ # is the sequence number of the FIRST byte sent in the **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)

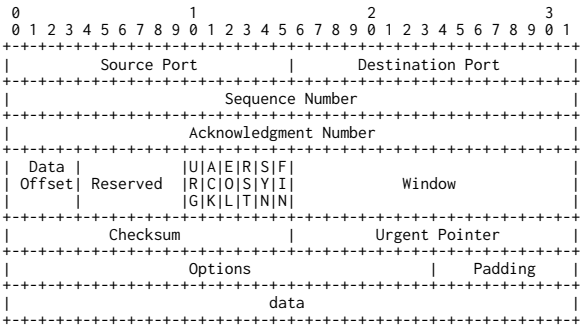
Assume

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

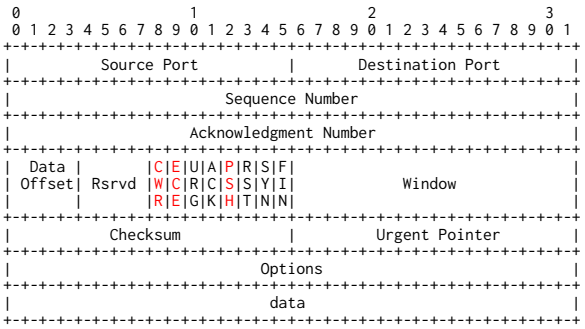
- G is about to send 200 bytes and in response H will send 15 bytes



TCP Header



TCP Header Format (RFC 761, Jan 1980)



TCP Header Format (RFC 9293, Aug 2022)

Checksum

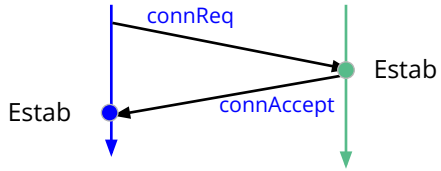
- Add each 2-byte as 16-bit binary number
- The sum must be a 16-bit value
  - If there is a carry, add the carry-bit to the current sum
- Find the 1's complement of the final 16-bit sum

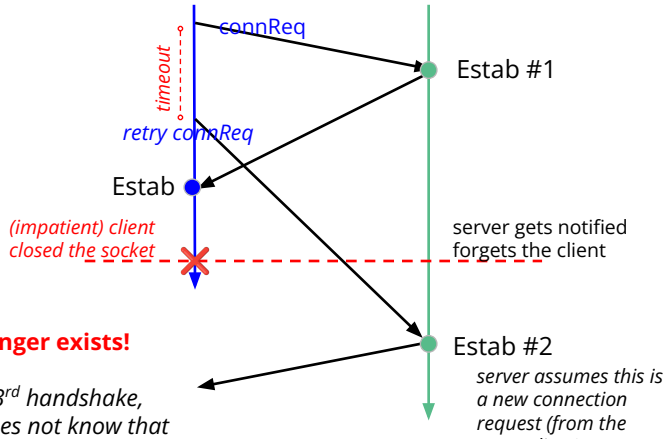
10010011 10000100 01110000 10110100

Example in 8-bit

```
  10010011
  10000100 +
-----------
1 00010111
  00011000      Add the carry bit to the sum
  01110000 +
-----------
  10001000
  10110100 +
-----------
1 00011100
  00011101      Add the carry bit to the sum
```

2-Way Handshake (may end up with half-open sockets)

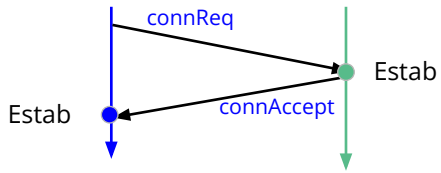


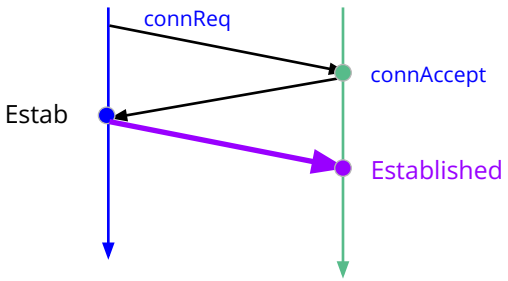


Client no longer exists!

Without the 3rd handshake, the server does not know that the client is already gone

2-Way Handshake vs. 3-Way Handshake

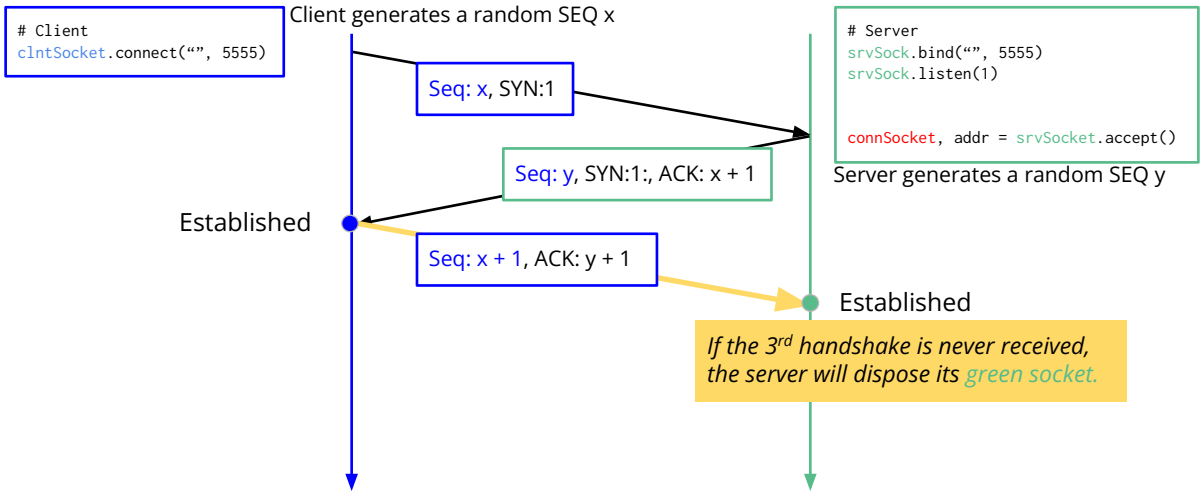




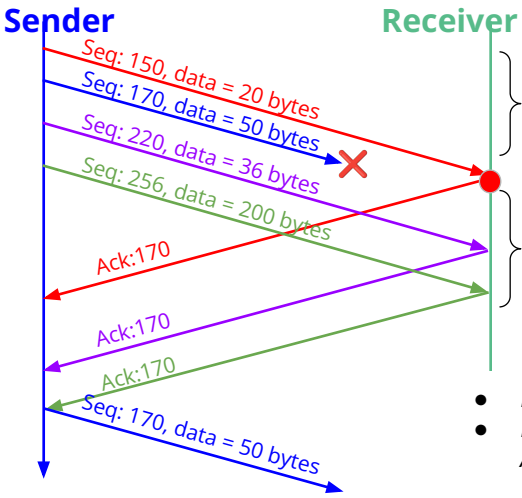
Think the purple "byte" as your 2FA for login.

If you never enter your 6-digit code, the login session will be cancelled

TCP 3-Way Handshake



Packet Loss ⇒ Dup ACKs ⇒ Fast Retransmit

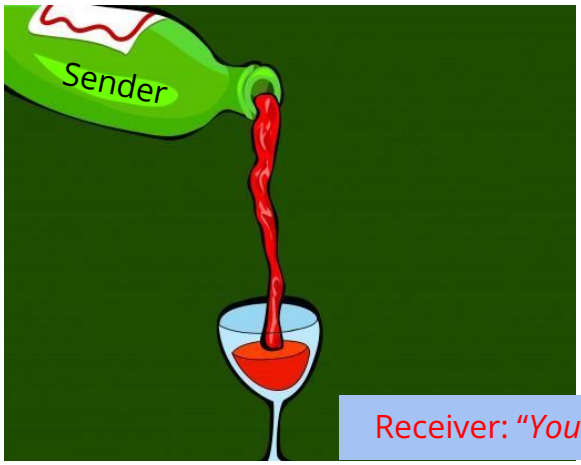


R_{ack}: 150 (before red dot)

R_{ack}: 170 (after red dot)

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

[TCP] Flow Control



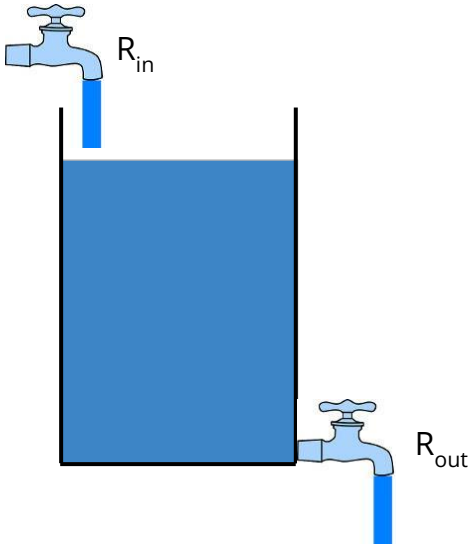
Receiver: "*You are giving me too much*"

[TCP] Congestion [Control]

(Grand) River Water Level After Snow Melt

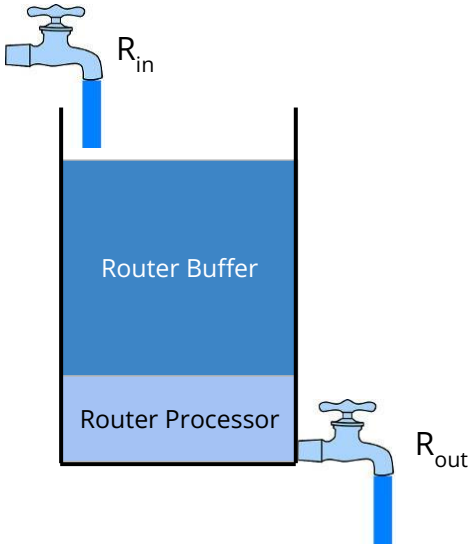
(Liquid) overflow because amount of (liquid) exceeds "container holding capacity"

Overflow if $R_{in} > R_{out}$



Router data loss because amount of incoming data exceeds buffer capacity

Overflow if $R_{in} > Proc_{rate} + R_{out}$



Flow Control vs. Congestion Control

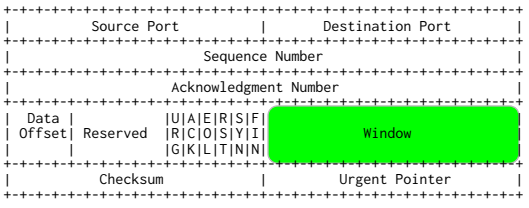
- **Avoid overloading a receiver**
  - The receiver tells the sender how much buffer space is available to receive data
  - TCP: "Receiver Window" (RWND)
- Local issue between a **single sender** and a **single receiver**
  - Easier to resolve
- Issue is detected/prevented by the receiver, and the sender 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)

Solution for both issues: **sender must adjust its sending behavior**

TCP Flow Control

The field in the TCP header for exchanging the **receive(r) window size**



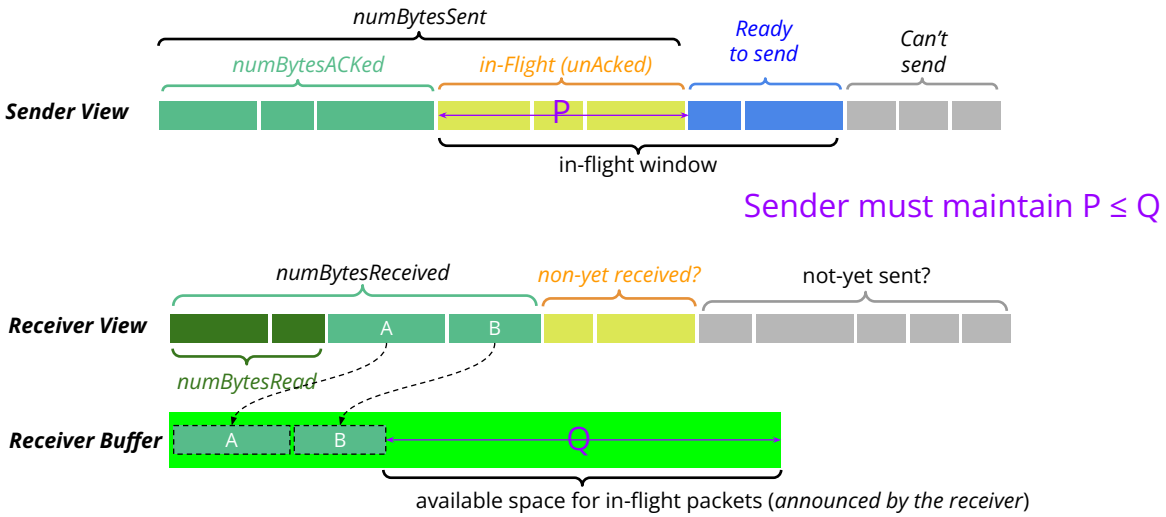
Wireshark Screenshot

Protoc	Lengt	Info
TLSv…	111	Application Data
TCP	54	55581 → 443 [ACK] Seq=1225 Ack=1955 Win=1023 Len=0
TCP	54	59299 → 443 [ACK] Seq=1225 Ack=3026 Win=255 Len=0
TCP	54	50403 → 443 [ACK] Seq=1225 Ack=1939 Win=2047 Len=0
TCP	55	[TCP Keep-Alive] 60408 → 443 [ACK] Seq=1966 Ack=1274 Win=64256 Len=1
TCP	66	[TCP Keep-Alive ACK] 443 → 60408 [ACK] Seq=1274 Ack=1967 Win=32768 Len=
TCP	164	63511 → 8009 [PSH, ACK] Seq=3081 Ack=3081 Win=253 Len=110 [TCP PDU reas
TCP	164	8009 → 63511 [PSH, ACK] Seq=3081 Ack=3191 Win=730 Len=110 [TCP PDU reas
TCP	54	63511 → 8009 [ACK] Seq=3191 Ack=3191 Win=253 Len=0
TLSv…	109	Application Data

TCP Flow Control

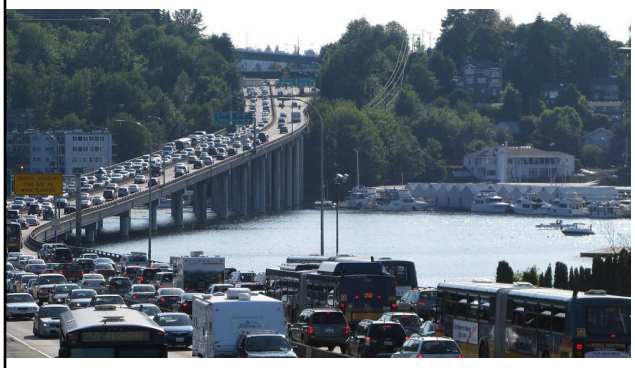
- TCP sender maintains a variable "Receive(r) Window" (`rwnd`) that indicates the *remaining available space* in the buffer <u>on the recipient side</u>
- Receiver variables
  - `numBytesReceived`, `numBytesRead`
  - `numBytesReceived - numBytesRead`: amount of data still in the buffer
  - `rwnd_in_recipient = RecipientBuffSize - (numByteReceived - numByteRead)`
  - Notify sender the current value of `rwnd_in_recipient` **when sending ACK**
- Sender variables
  - `numBytesSent`, `numBytesAcked`
  - `numBytesSent - numBytesAcked =` amount of unACKed data ("in-flight")
  - Must maintain `numByteSend - numByteAcked ≤ rwnd_in_recipient`

TCP Flow Control (in one picture)



(Traffic) Congestion: Overview

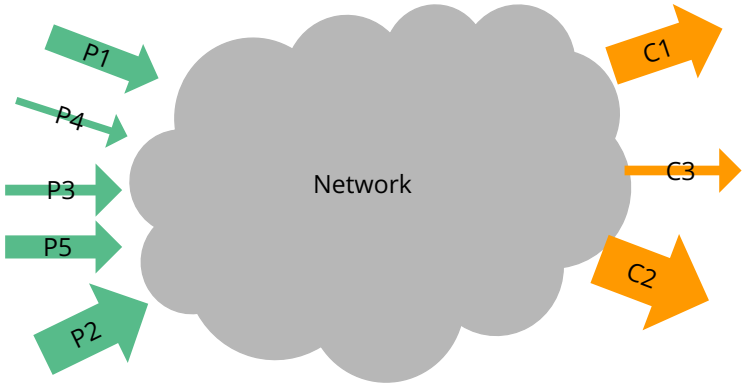
What causes Congestion?



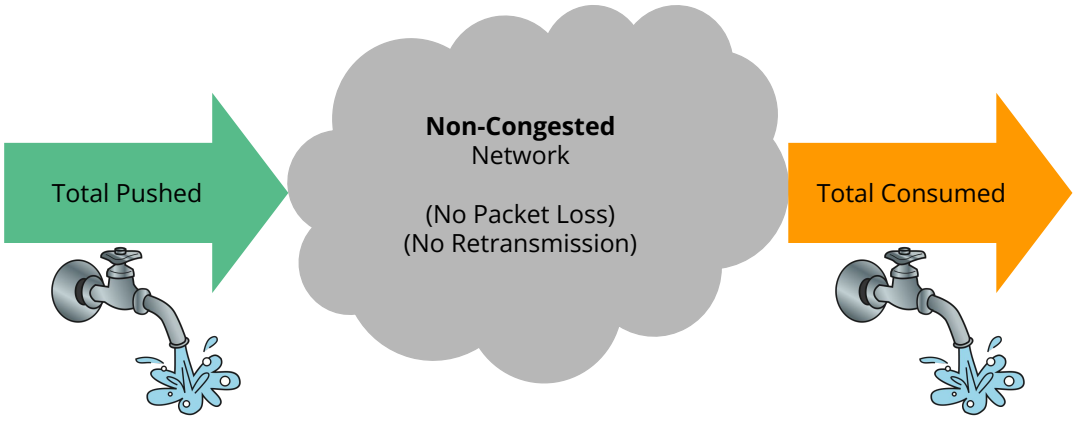
Roads have (limited) carrying capacity (<u>cars/minute</u>), so do network links.

If the number of cars (bits) exceeds this capacity ⇒ traffic congestion

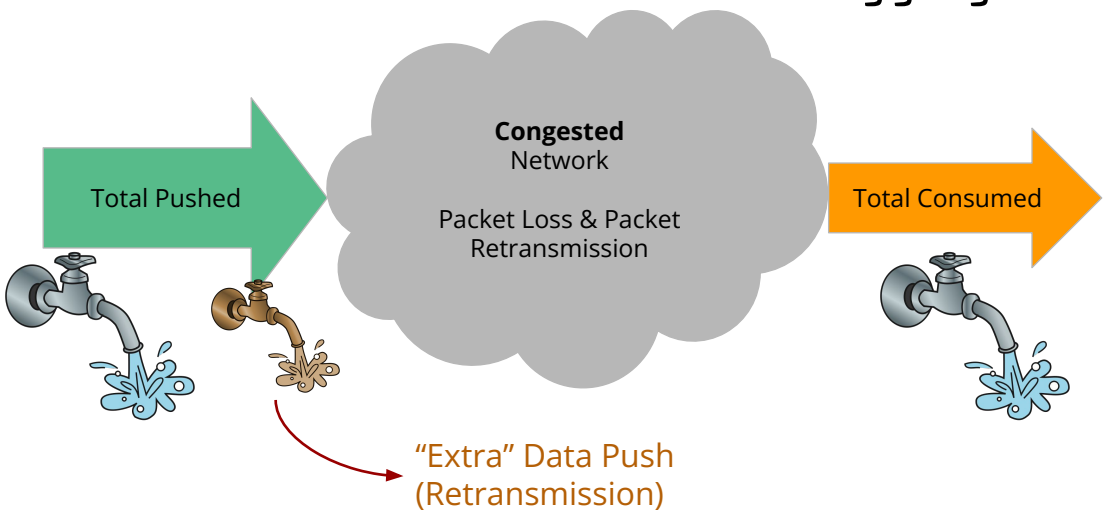
Network: Bits Pushed & Bits Consumed



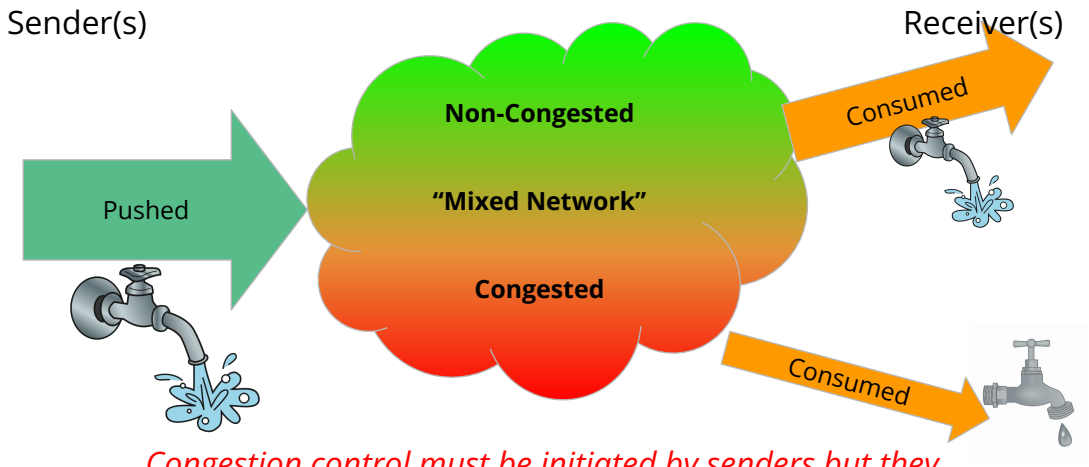
Non-Congested: Bits Pushed & Consumed (Aggregate)



Network: Bits Pushed & Consumed (Aggregate)



Network: Bits Pushed & Consumed



Congestion control must be initiated by senders but they can't measure how much data were consumed!!!

How do you recognize (road) traffic congestion?

- By counting the number of cars (nearby)?
- By knowing the traffic capacity of the road you are on?
- By observing your travel speed?
- By observing your travel time?

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

Congestion & Router Buffer Capacity

Rate of bits pushed ≪ Network Bandwidth

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

Rate of bits pushed ≈ Network Bandwidth

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

Key Takeaway: in a congested network there will be *(high) packet loss* and *(long) packet delay*.
Sender(s) must decrease sending rate.

How does a sender measure, adjust, and limit its own bit push rate?

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 (triple duplicate ACKs or timeout)	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)

TCP Congestion Control
Option #1 Classic: Observe Packet Loss

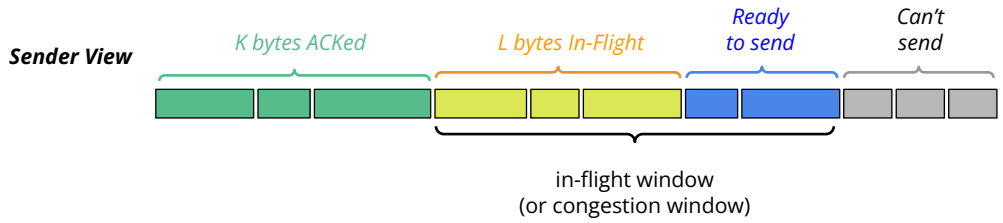
Symptoms: Packet Loss vs. Packet Delay

Issue	Symptom	Timer Timeout	Retransmit
Partial Packet Loss	Duplicate ACKs	No	Fast Retransmit
Severe Packet Loss	No ACK	Yes	Retransmit when timer expires
Packet Delay	Late ACK	Yes	Retransmit when timer expires

In case of any of the above symptom, the sender should decrease its sending rate, i.e. decrease its "**in-flight window size**".

In TCP "in-flight window" is called "**congestion window**" (cwnd)

TCP "In-Flight Window == Congestion Window"



TCP Flow Control & Congestion Control

Variable	Name	Unit	Purpose
rwnd	Receive Window (advertised by receiver)	bytes	Amount of *remaining free space* in the receiver buffer ⇒ *"How much space **you** have*"
cwnd	"Sender" Congestion Window	# of segments	Control how much traffic a sender can push into the network ⇒ "*How much data **I** can push*"

- Both variables are maintained by the sender
- The variable cwnd is a multiplier of (minimum) segment size (MSS), can increase or decrease
- The sender must maintain

$$\texttt{numBytesInFlight} \leq \min\{\texttt{rwnd}, \texttt{cwnd} * \texttt{MSS}\}$$

Adjustment of Sender's cwnd

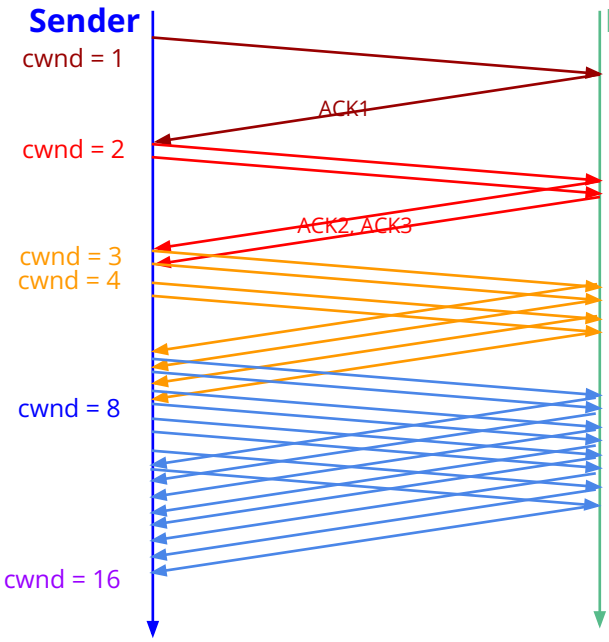
Event	cwnd adjustment	Reason for adjustment
ACK received (Not Dup ACKs)	Increase	The network is delivering well (both ways)
Timeout	Decrease	Packet Loss or ACK propagates very slowly
Dup Triple ACKs	Decrease	Intermittent Packet Loss

Can't increase indefinitely. At some point the sender push rate will be limited by the network bandwidth

Increase  Aggressively Cautiously

Increasing cwnd

- **Aggressively**: increase by one each time the sender receives a ACK of a (new) packet, not an ACK of retransmitted packets
- **Cautiously**: increase by one for each RTT



- cwnd = max in flight packets
- Aggressive increase: each ACK of a (new) packet increases cwnd by 1
- The *cumulative effect* of each **pipeline** is **doubling the cwnd** periodically every RTT

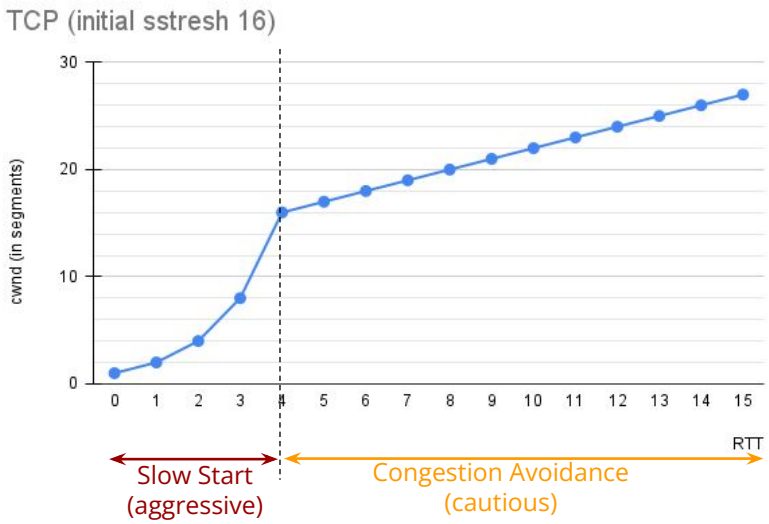
TCP Congestion Control Variables

Variable	Name	Purpose
`rwnd`	Receive Window	Keep track the amount of *remaining free space* in the receiver buffer ⇒ *"How much space **you** have"*
`cwnd`	"Sender" Congestion Window	Control how much traffic a sender can push into the network ⇒ "*How much data **I** can push*"
`sstresh`	Slow Start Threshold	The value at which `cwnd` growth switches from exponential (doubling each RTT) to linear

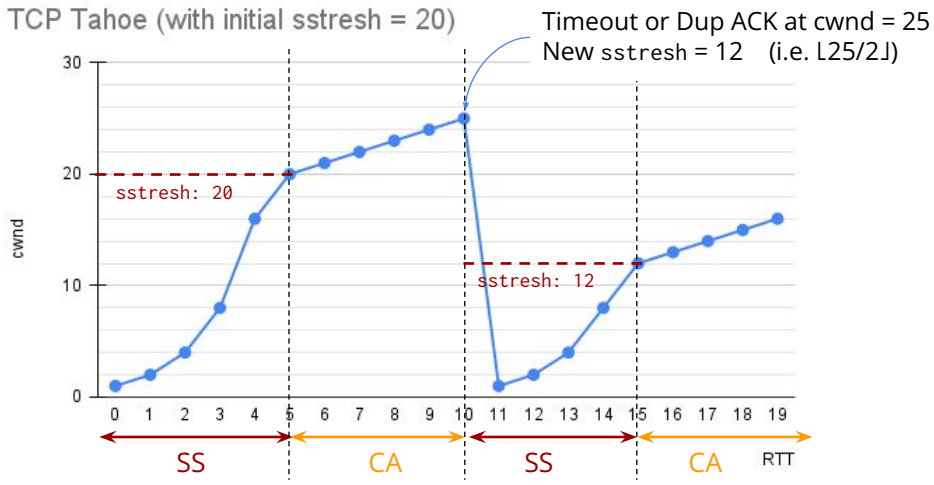
TCP Congestion Control Phases (RFC 5681 Sec 3)

Phase	Required	Description	When to activate
Slow Start	Yes	Double cwnd every RTT	• At the beginning • After timer expiration • After 3 dup ACKs (only in TCP Tahoe)
Congestion Avoidance	Yes	Add 1 to cwnd every RTT	When cwnd ≥ sstresh
Fast Retransmit (in TCP Reno)	No	Retransmit the missed packet(s) sstresh := cwnd / 2 cwnd := sstresh + 3	When sender receives 3 duplicate ACKs
Fast Recovery (in TCP Reno)	No	Add 1 to cwnd every RTT	After Fast Retransmit and until non duplicate ACKs received

TCP Congestion Control: Slow Start & Avoidance



TCP Tahoe

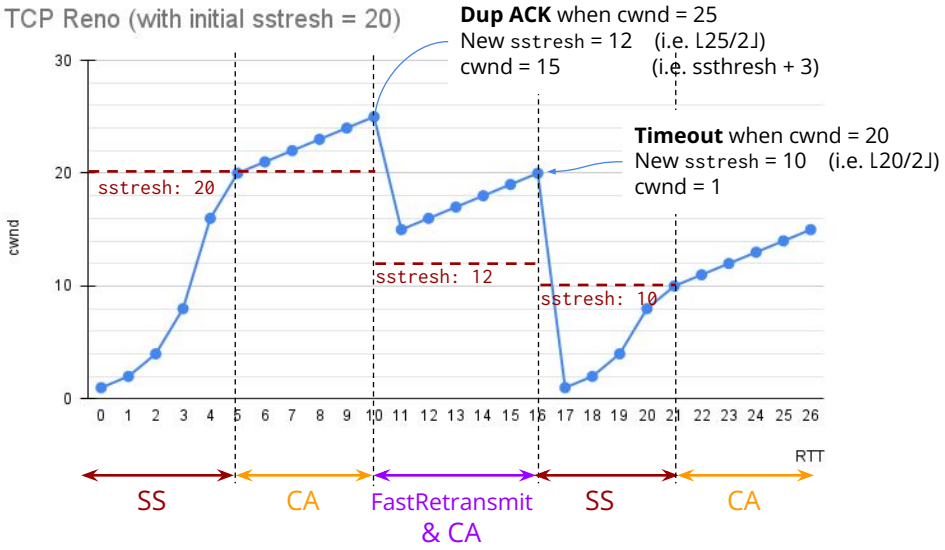


TCP Tahoe vs. TCP Reno

	On Timeout	On Receiving 3 Dup ACKs
Tahoe	sstresh := cwnd / 2 cwnd = 1	sstresh := cwnd / 2 cwnd = 1
Reno	sstresh := cwnd / 2 cwnd = 1	ssthresh := cwnd / 2 cwnd := sstresh + 3

- TCP Tahoe handles both cases of “packet loss” in the same fashion
- On “packet loss” both both TCP Tahoe and Reno reset ssthresh to cwnd / 2
- On Timeout both TCP Tahoe and Reno reset cwnd to 1
- On Triple Dup ACKs, TCP Reno attempts to do “fast(er) recovery” by not dropping cwnd to 1

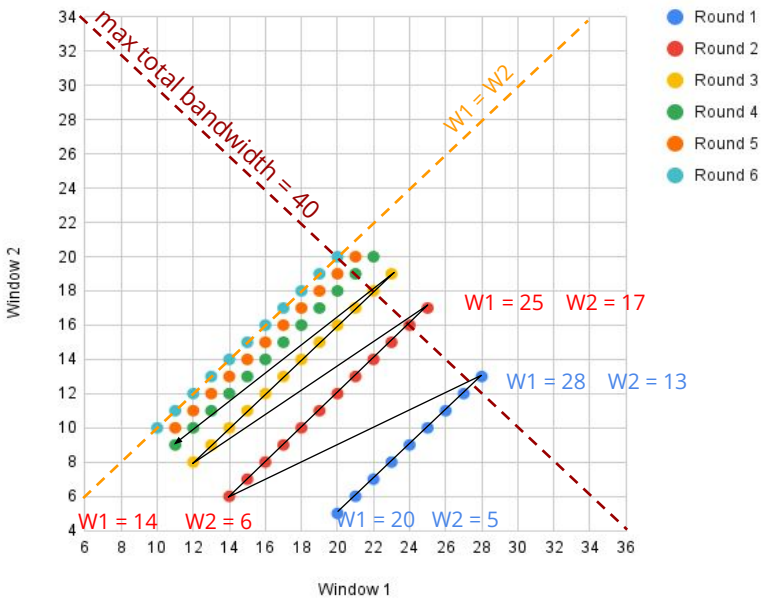
TCP Reno



TCP Fairness

Initial window size
Sender #1: 20
Sender #2: 5

after "zigzagging" both window sizes converge to 10



TCP Congestion Control Animation

Setting Up Timeout Duration?

- Too short: too early (and too frequent)
  - unnecessary retransmissions
  - unnecessary duplicate packets transmitted by the sender (& ignored by receiver)
  - wasting available network bandwidth
- Too long:
  - longer time to recover from packet loss
  - network stays idle for too long ⇒ underutilized network bandwidth
- Ideally, the Time Out Duration should be very close to the average RTT

TCP Congestion Control
Option 2: Time-Based (Delay-Based)

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	How Much You're Off	Daily Estimate
		???	50 minutes (initial wrong estimate)
Mar 7	20 minutes	+30 (overestimate)	(0.8)(20) + (0.2)(50) = 26
Mar 8	32 minutes	-6 (underestimate)	(0.8)(32) + (0.2)(26) = 30.8
Mar 9	25 minutes	+5.8 (overestimate)	(0.8)(25) + (0.2)(30.8) = 26.16

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	30 (over)	(0.75)(30) + (0.25)(10) = 25
Mar 8	32	(0.8)(32) + (0.2)(26) = 30.8	6 (under)	(0.75)(6) + (0.25)(25) = 10.75
Mar 9	25	(0.8)(25) + (0.2)(30.8) = 26.16	5.8 (over)	(0.75)(5.8) + (0.25)(10.75) = 7.04

On the morning of Mar 10, you expect 26.16 minutes of travel time, and expect your estimate will be off by 7.04 minutes

TCP Congestion Control (Time-Based)

- TCP Tahoe & Reno may "probe too far", causing packet loss
- General ideal
  - Available network bandwidth is proportional to $\frac{\text{BytesInFlight}}{RTT}$
  - The minimum RTT (fastest response) can be used to infer the best expected bandwidth
  - Changes in available bandwidth can be detected by change in RTT
    - Longer RTT is an indication of lower available bandwidth
    - Shorter RTT is an indication of higher available bandwidth
  - Changes in RTT can be used to adjust cwnd (maximum bytes in flight)

- Warning: The textbook calls this "**Delay-based TCP Congestion Ctrl**"

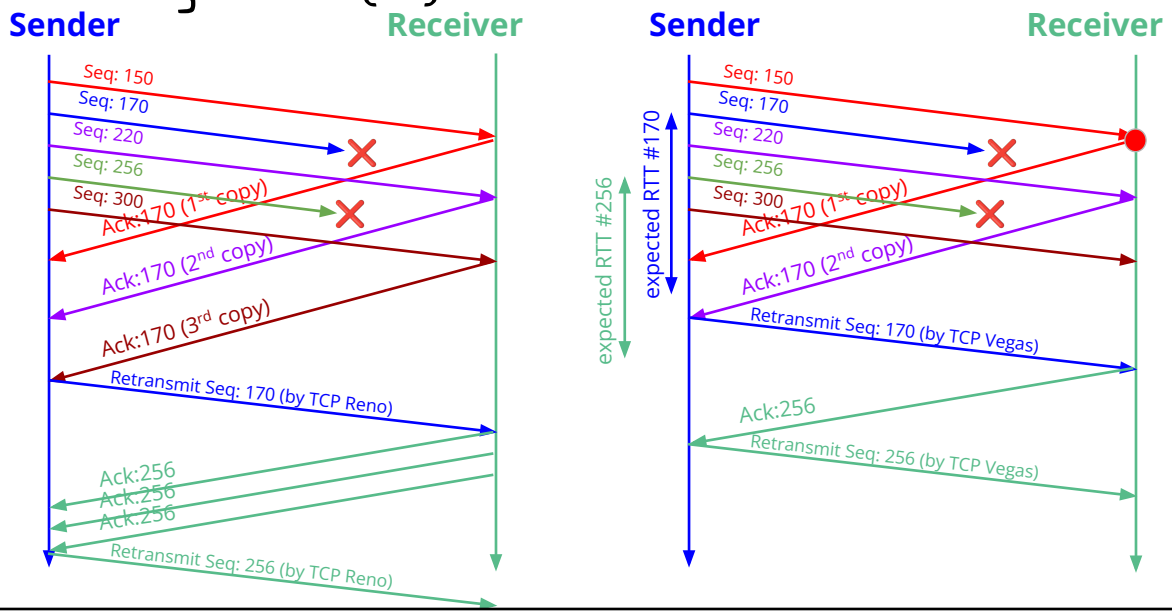
TCP Congestion Control (Time-Based)

- R_{uc} : the RTT in an uncongested network (should be the "min" RTT **so far**)
- Define two thresholds α < β. Goal is to keep: α < inFlightBytes < β
- Compute ExpectedRate (= cwnd/R_{uc}) and ActualRate (= cwnd/actualRTT)
- Compute Diff = ExpectedRate - ActualRate
  - By definition of R_{uc}, Diff cannot not be negative (actualRTT is always ≥ R_{uc})
- Compute InFlightBytes = Diff * BaseRTT
- If InFlightBytes < α, **increase cwnd**
- If InFlightBytes > β, **decrease cwnd**

Retransmission: TCP Reno vs. TCP Vegas

	TCP Reno	TCP Vegas
Fast Retransmit	On receiving 3 duplicate ACKs (1 orig + 2 repeats)	• On receive 2 dup ACKs X (1 orig + 1 repeat) when its actual RTT is longer than its expected RTT • On receive of a new ACK Y (the 1st or 2nd **after retransmission**) when its actual RTT is longer than its expected RTT

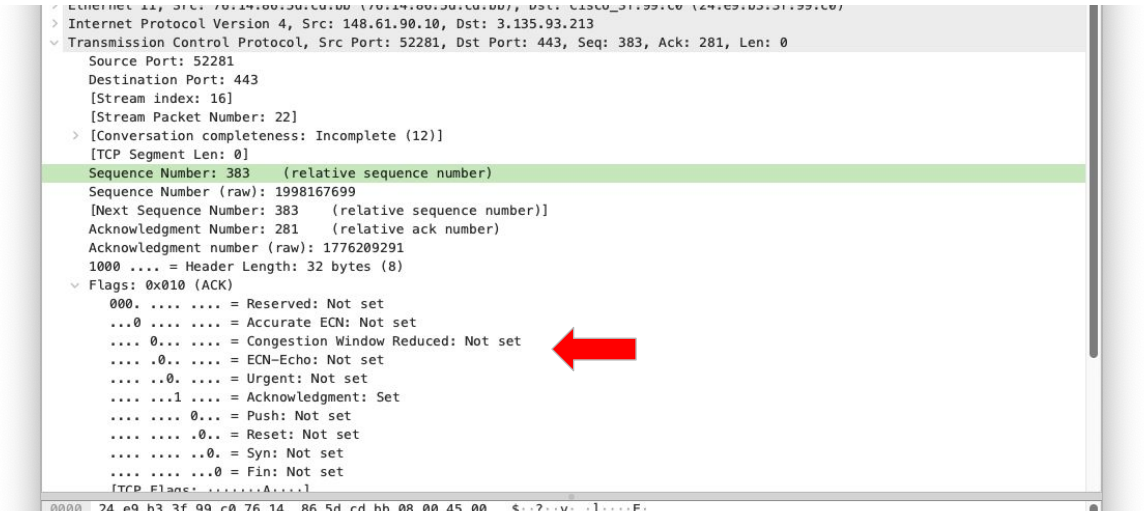
TCP Vegas Fast(er) Retransmit



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

TCP Congestion Control: Explicit Congestion Notification

- Network-Assisted, i.e. require assistance from the Network Layer (IP Protocol)
- Extra bits in the packets to notify congestion to the 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