Lecture 7

The Transport Layer (2)

Recap: Reliability via "**Stop-and-Wait**"

- Sender key ideas:
  - Sender sends one packet of application data, then **waits** for **acknowledgment** that receiver got it (ACK)
  - Upon receiving an ACK for that packet, sender can move on to send next data packet
    - Which packet an ACK corresponds to is determined based on **sequence number**
  - If the ACK isn't received within a certain **timeout**, sender retransmits

- Receiver key ideas:
  - Upon receiving a data packet, the receiver sends an ACK **for that packet** (i.e. carrying same **sequence number**)
  - If this is a **new** data packet (i.e. next expected sequence number), delivers it "up" to application

Complete Reliable Data Transfer Protocol Using the Stop-and-Wait Approach

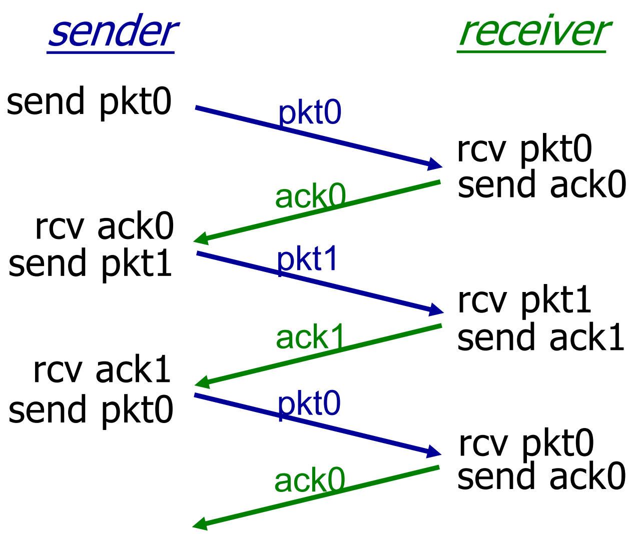
Sender:

- Set seq=0
- Wait for data to send **(1)**
- Create and send packet(seq, data)
- Start timer
- Wait for ACK or Timeout **(2)**
  - If ACK && not corrupt && ACK_seq == seq:
    - Stop timer
    - seq = (seq+1) % 2
    - Wait for more data (1)
  - Else if Timeout:
    - Resend packet
    - Restart timer
    - Wait for ACK (2)

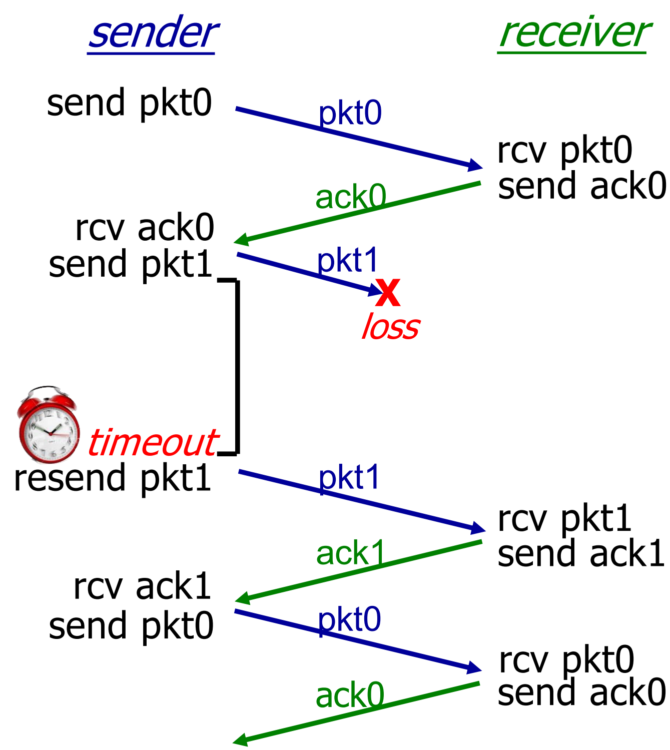
Receiver:

- Set seq=0
- Wait for packet from sender **(1)**
  - If not corrupt:
    - Send ACK(recvd_seq)
    - If recvd_seq == seq:
      - Extract and deliver to application
      - seq = (seq+1) % 2
  - Wait for new packet from sender (1)

Stop-and-Wait: Example Executions

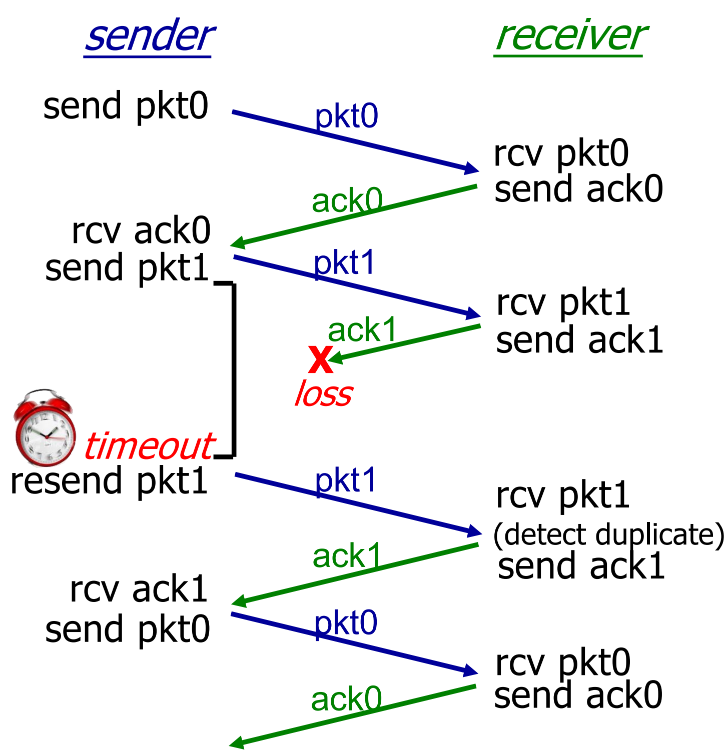


(a) no loss

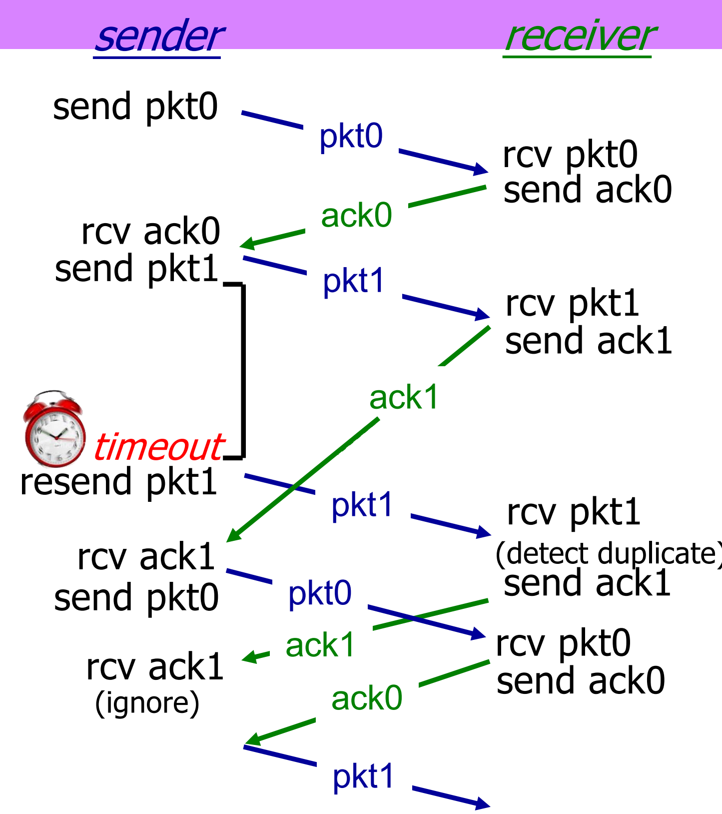


(b) packet loss

Stop-and-Wait: Example Executions



(c) ACK loss



(d) premature timeout/ delayed ACK

Complete Reliable Data Transfer Protocol Using the Stop-and-Wait Approach

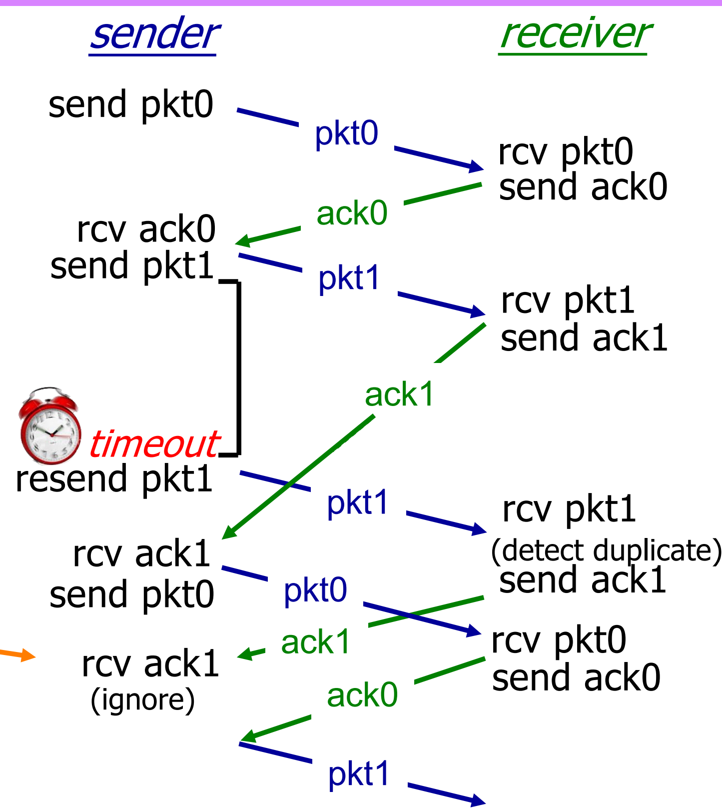
- **Why don't we use duplicate ACKs as evidence of loss?**

- In the preliminary versions of this protocol that we discussed last week, we said the receiver could send a NAK or duplicate ACK after receiving a corrupted message to prompt the sender to retransmit. The final version relies on timeouts only. **Why**?

- i.e. Why not retransmit immediately in the case where ACK seq doesn't match our expected seq?

Stop-and-Wait RDT Operation: Premature Timeouts

What would happen if we didn't ignore this ack?



- Sender retransmits pkt0
- Receiver gets pkt0, assumes ack was lost, retransmits ack0
- Sender gets ack0 after having sent pkt1, so retransmits pkt1
- Receiver gets pkt1 retransmission, retransmits ack1
- …

If the pkt was really lost, retransmitting on duplicate ack lets us respond faster.

But, if not, generates a lot of extra traffic (without additional protocol changes)

And, sender cannot tell which case it's in

Reliable Data Transfer: Mechanism Summary

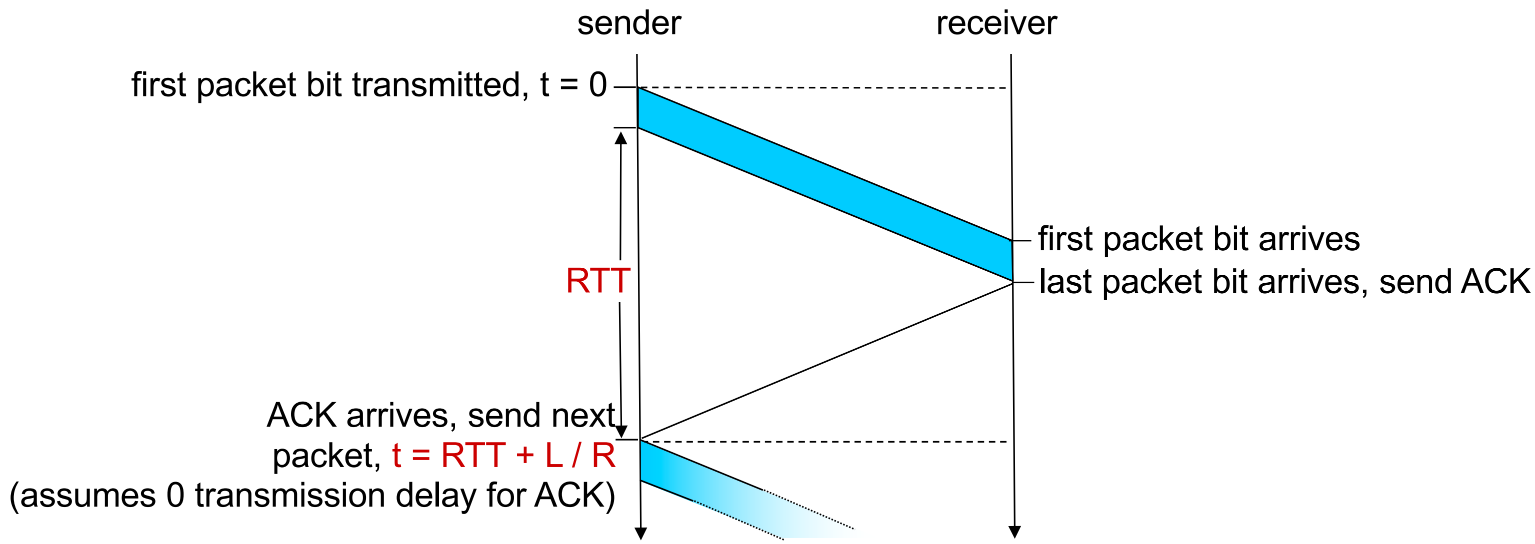
- Checksums: detect errors
- ACKs/NAKs: provide sender with feedback about what has been received
- Retransmissions: sender resends lost/corrupted packets
- Sequence numbers: identify packets, allow de-duplication
- Timeouts: detect (probable) loss, decide when to retransmit

Analyzing Stop-and-Wait

- Our protocol is finally correct…but is it good?

Analyzing Stop-and-Wait

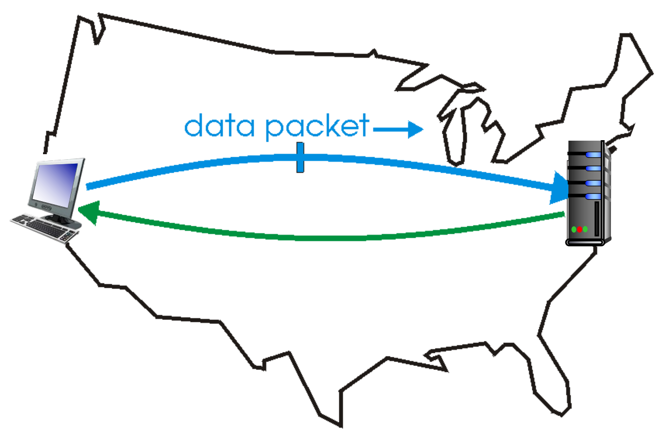
- Our protocol is finally correct…but is it good?



Analyzing Stop-and-Wait

- Our protocol is finally correct…but is it good?

RTT ~60ms
Bandwidth ~1Gbps
Segment size ~1500bytes



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

Transmission delay:
(1500*8)/(10^9) = 12 microsec

Assuming negligible transmission delay for ACK and no queuing/processing, we wait 60.012 ms to send next segment

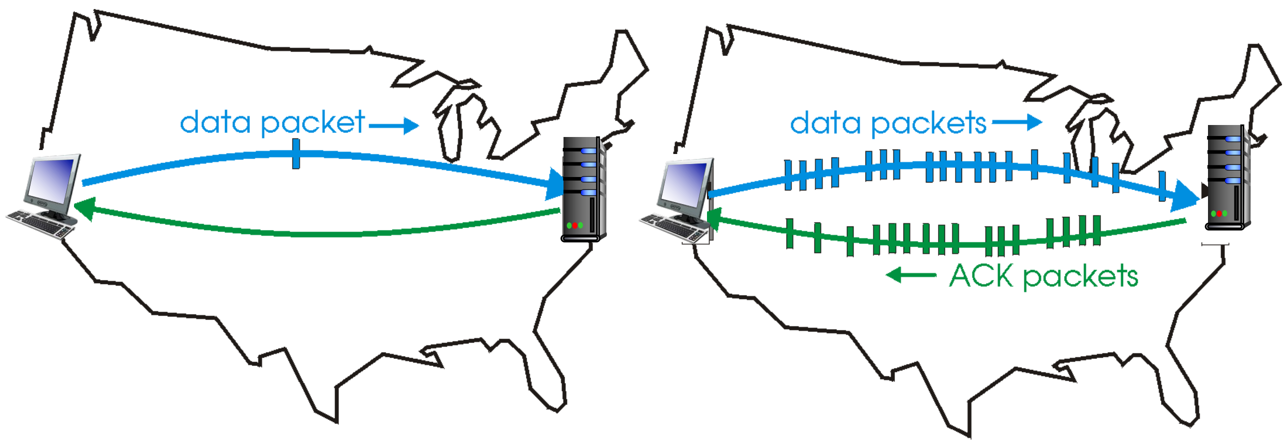
How long would it take to transfer a 1Gb file at this rate?

Improving Performance

- **How can we improve our protocol to get better performance?**
  - Think back to HTTP discussion…what strategies did we use there?

Improving Performance

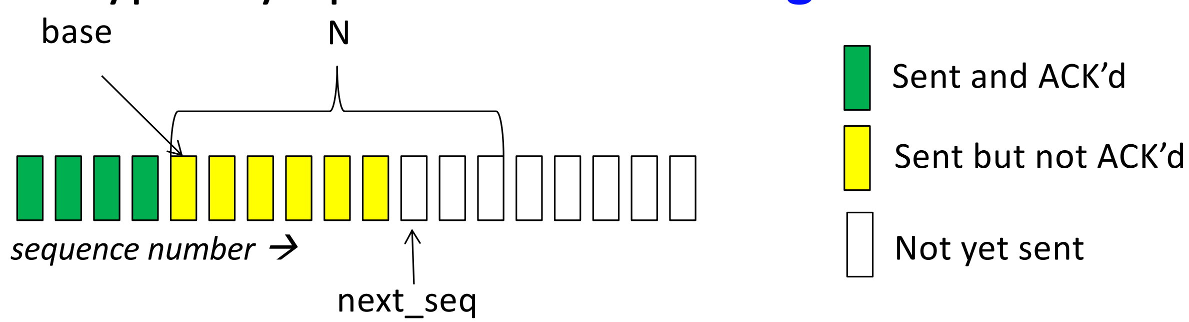
- **How can we improve our protocol to get better performance?**
  - Pipelining – instead of stopping and waiting after **every** packet, we can send **multiple** packets to “fill the pipe” before waiting for acknowledgment



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

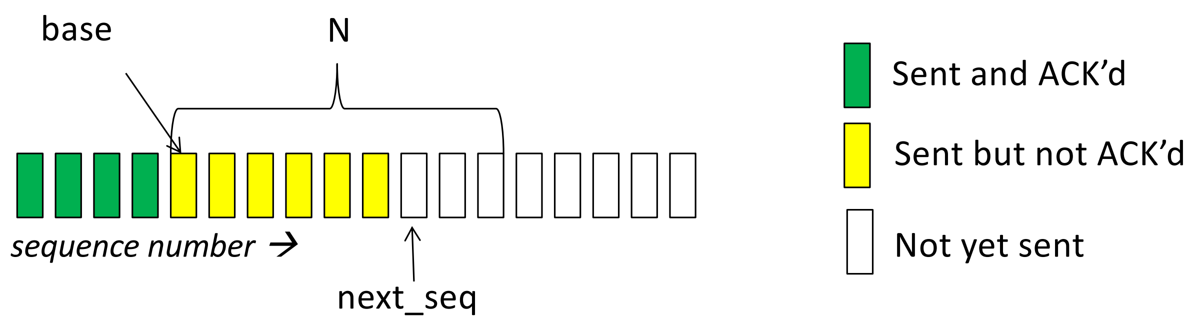
Pipelined Reliable Data Transfer

- Instead of only 1 unacknowledged packet, sender can have **up to N unacknowledged packets** at a time
  - N packets "in-flight"
  - Requires more than 1-bit sequence number
- Sender needs to **buffer** sent but not-yet-acknowledged packets so they can be retransmitted as needed
- Sender buffer typically operates as a "**sliding window**"



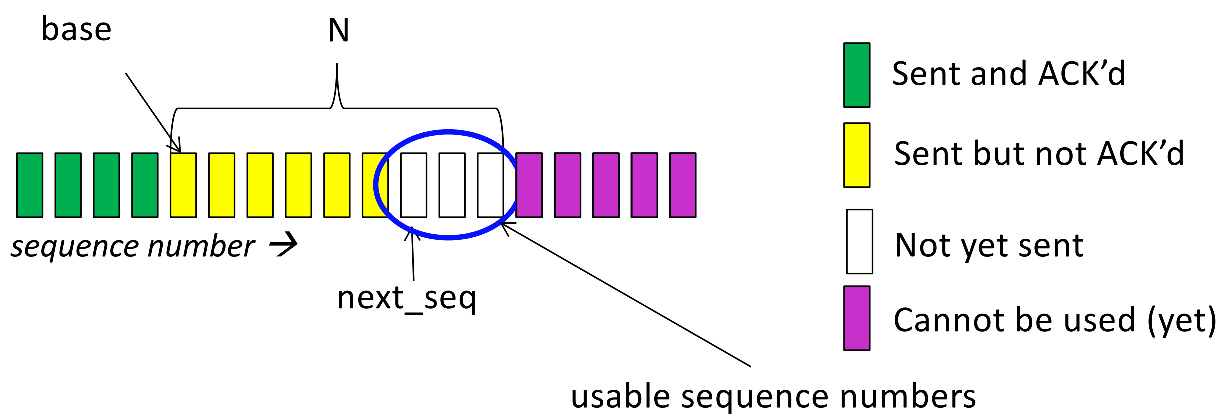
Sliding Window

- Sender buffer typically operates as a "**sliding window**"
  - base is the sequence number of the start of window (last ACK'd packet + 1)
  - next_seq is the sequence number that will be used for the next packet to send



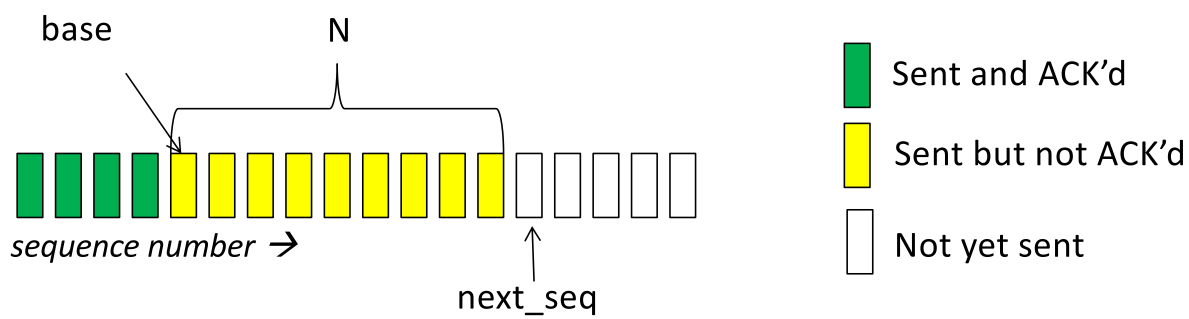
Sliding Window

- Sender buffer typically operates as a "**sliding window**"
  - base is the sequence number of the start of window (last ACK'd packet + 1)
  - next_seq is the sequence number that will be used for the next packet to send



Sliding Window

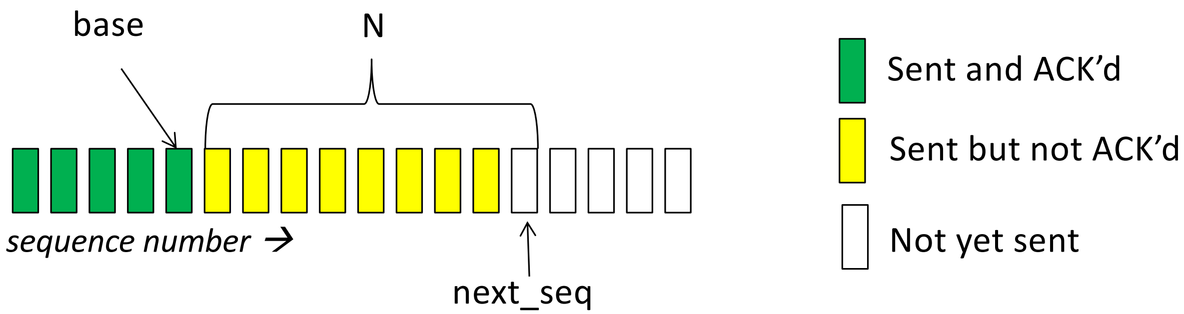
- Sender buffer typically operates as a "**sliding window**"
  - once the window is full (we've used up all the sequence numbers between base and base+N), no new packets can be sent…



Sliding Window

- Sender buffer typically operates as a "**sliding window**"
  - …until a new ACK lets us "slide" the window forward

https://www.etsy.com/listing/1303075498/vintage-post-versalog-slide-rule



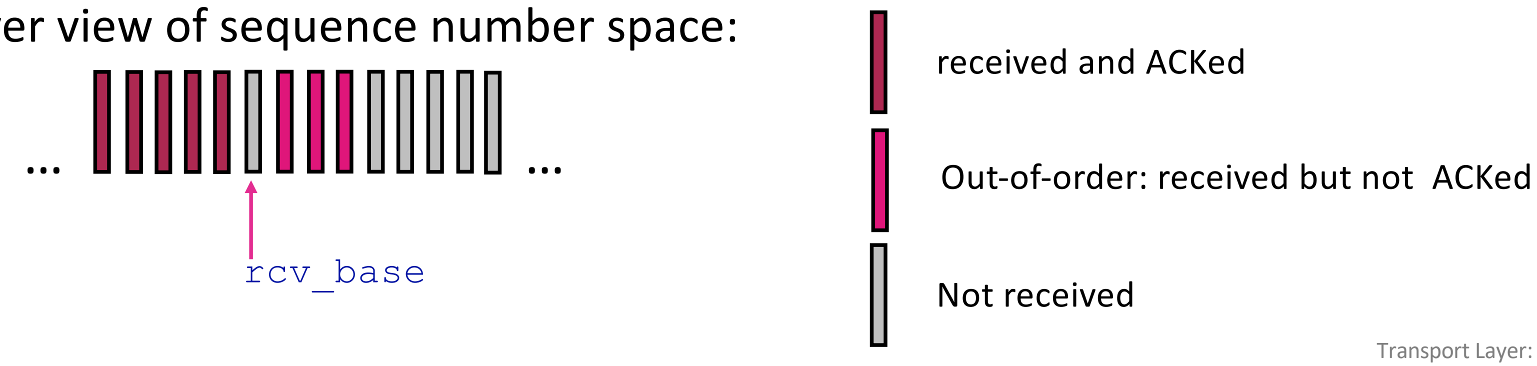
A Pipelined Reliable Data Transfer Protocol: "Go-Back-N"

- Go-Back-N protocol key ideas:
  - Sender can have **up to N unacknowledged packets** "in-flight" at a time
  - Receiver sends **cumulative acknowledgments**
    - Cumulative acknowledgement for sequence number X means I have received everything **up to and including** sequence number X
      - sometimes called ARU for "all received up-to"
    - Acknowledgments let the sender "slide its window forward" and send new packets until there are again N unacknowledged packets
  - When a **timeout** occurs, the sender will **"Go-Back-N"** and resend **ALL not-yet-acknowledged packets** (of which there can be up to N)

Go-Back-N: receiver

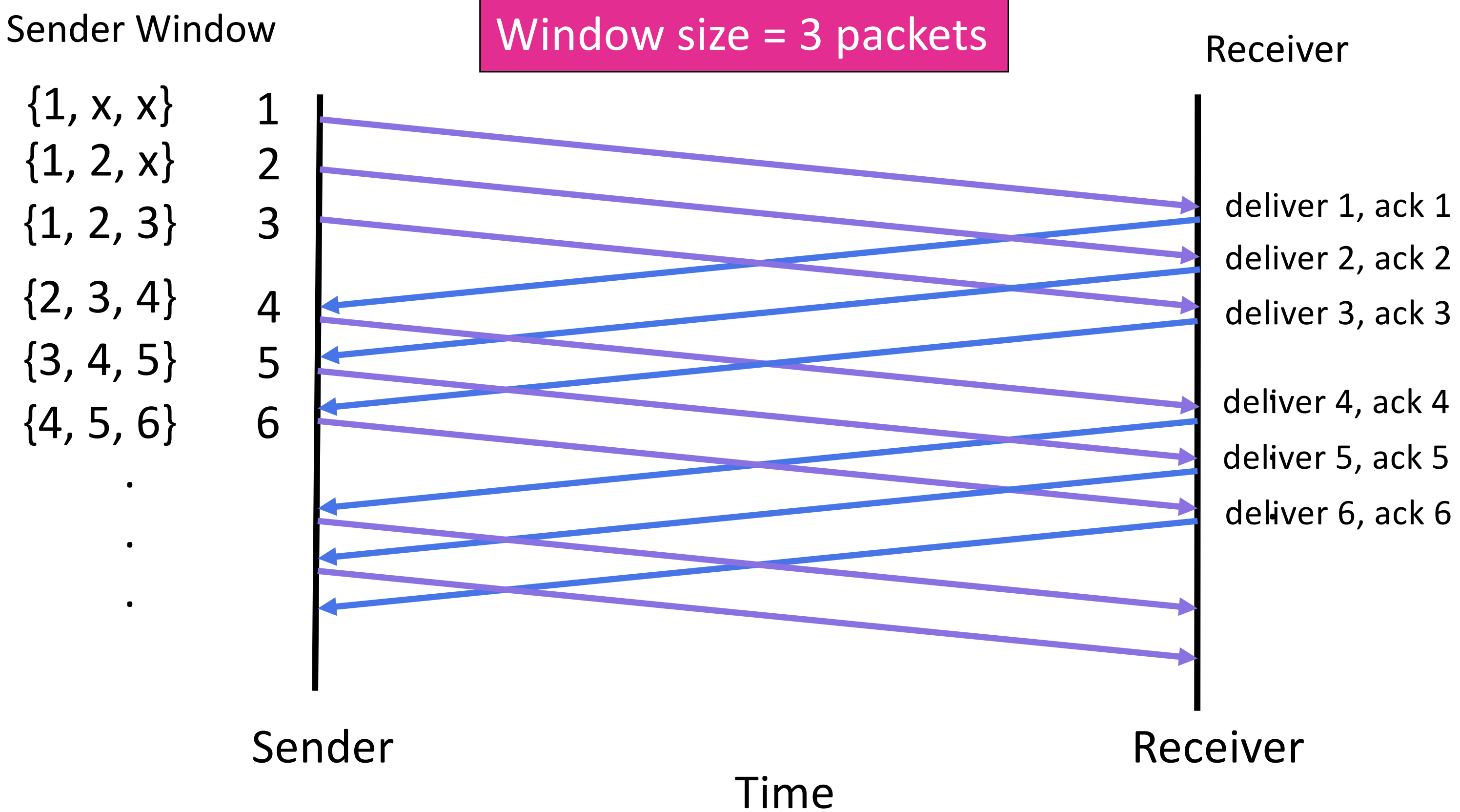
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:

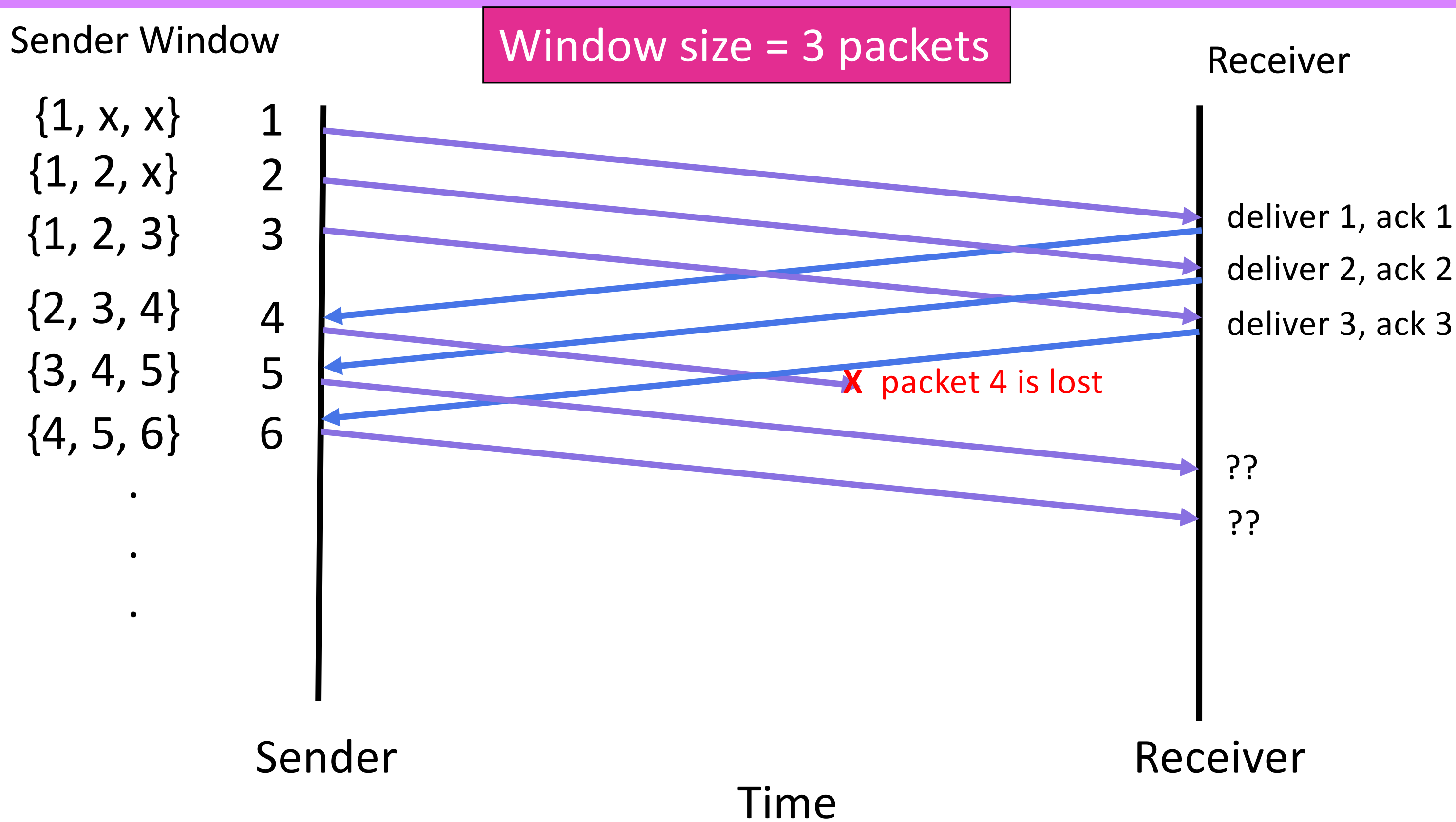


Go-Back-N Normal Case Operation

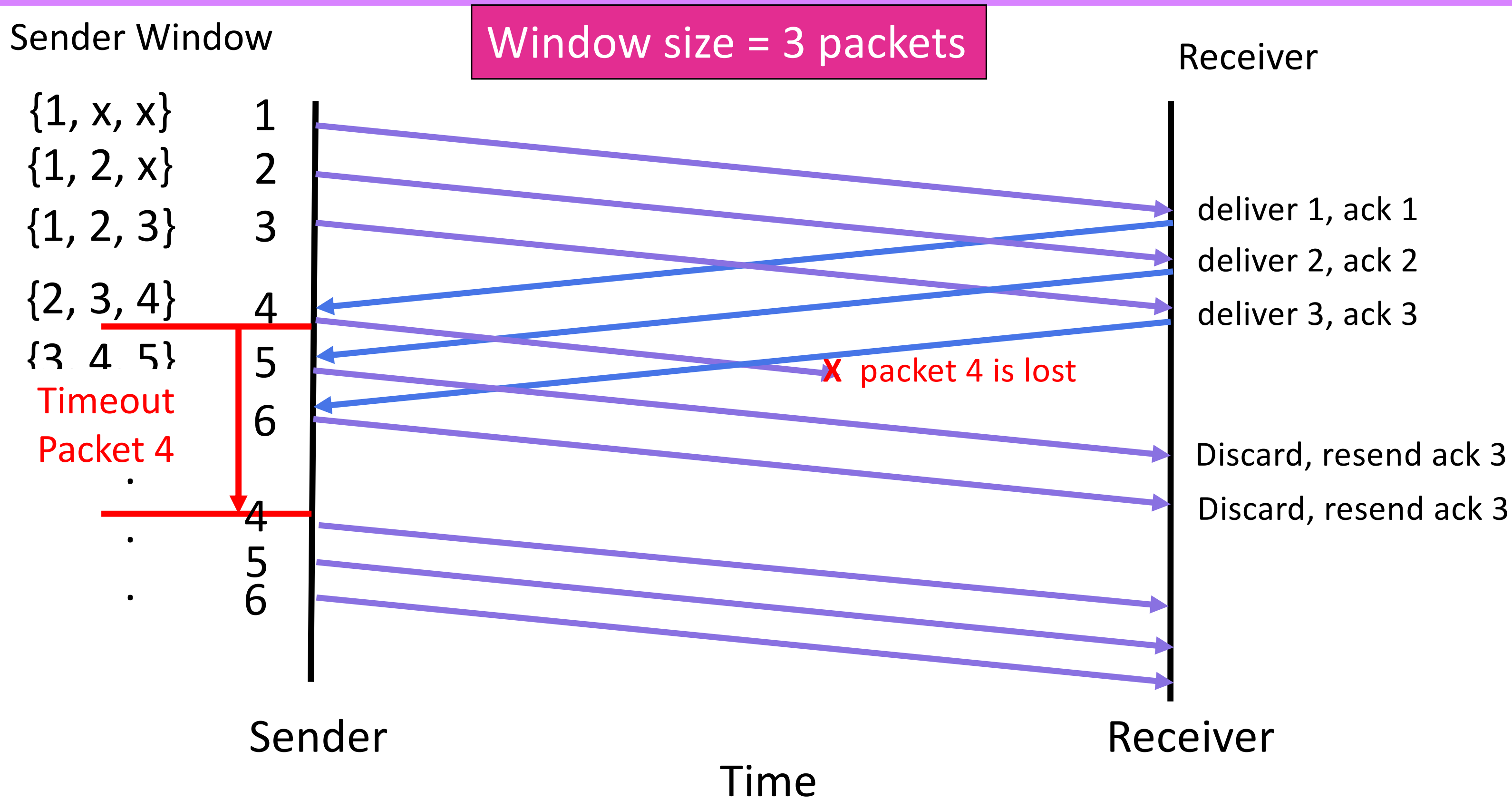
Window size = 3 packets



Go-Back-N Operation with Loss



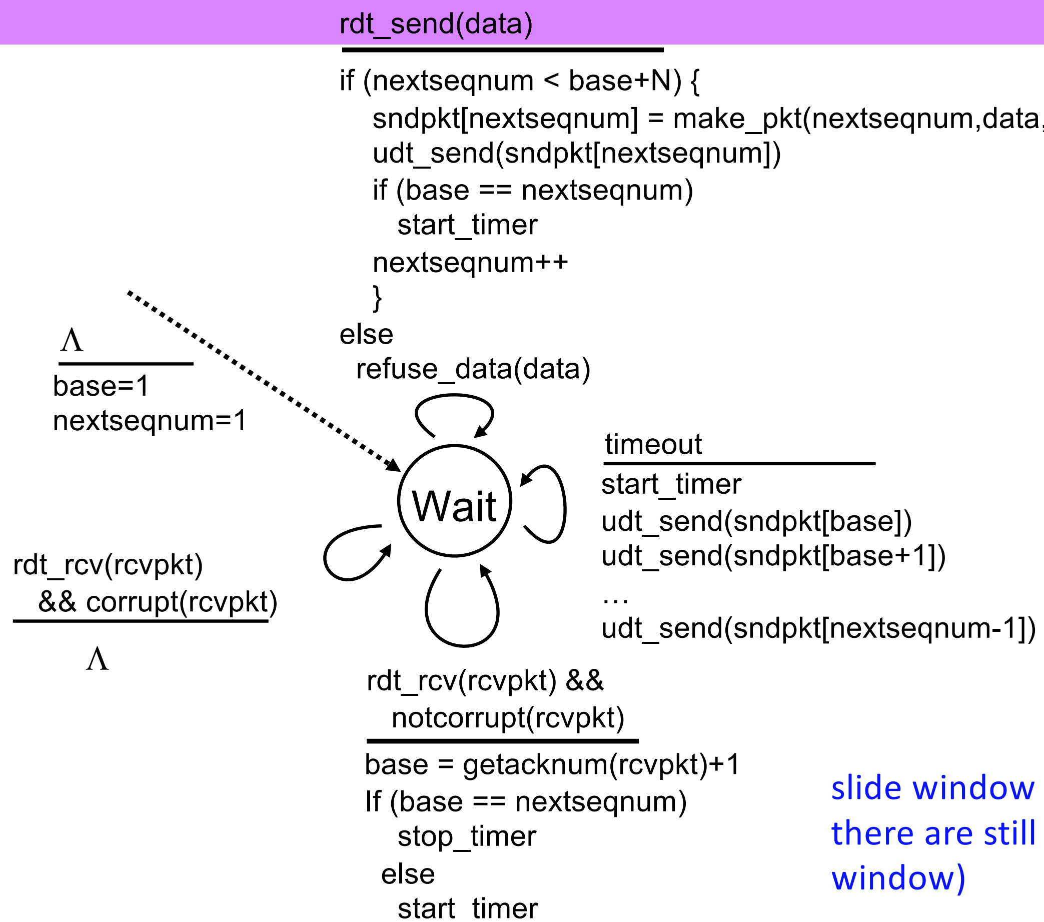
Go-Back-N Operation with Loss



Go-Back-N: Details

- Sender
  - Sends up to N unacknowledged packets
  - Maintains timer for **oldest unacknowledged packet**
    - Set timer when sending first packet in empty window
    - Reset timer when receiving ack that moves up window (or stop if no more un-ack'd packets)
    - (and reset timer after resending window)
  - Resends ALL un-ack'd packets on timeout
- Receiver
  - Sends cumulative ACK upon receiving next expected packet
  - Resends last cumulative ACK upon receiving out-of-order packet
  - Discards packets received out of order (some variants allow these to be buffered – why?)

Go-Back-N: Extended FSM (Sender)

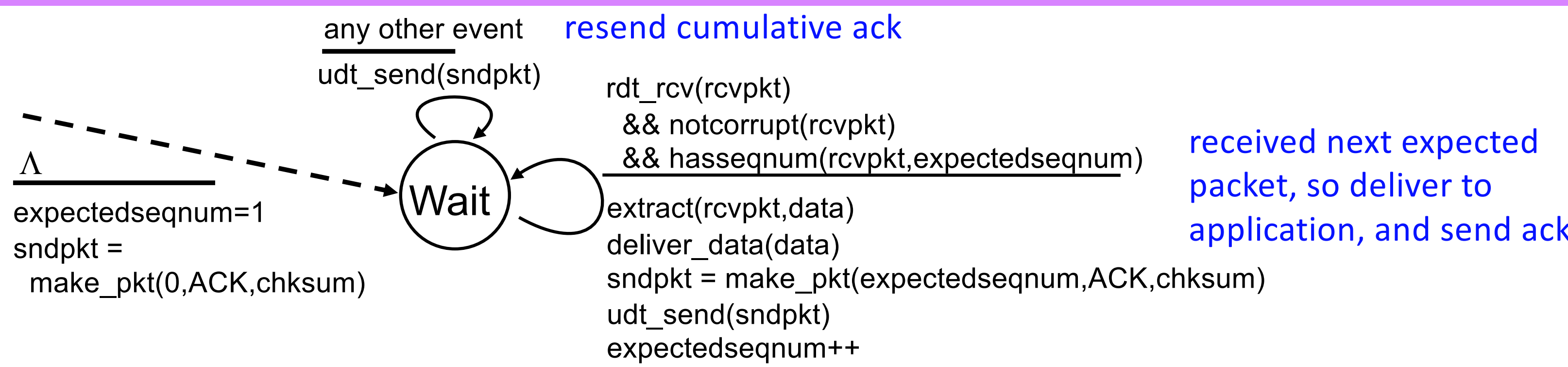


if there is space in window, send new data (starting timer if not already set)

reset timer and resend all packets in window

slide window forward and reset timer (if there are still unacknowledged packets in window)

Go-Back-N: Extended FSM (Receiver)



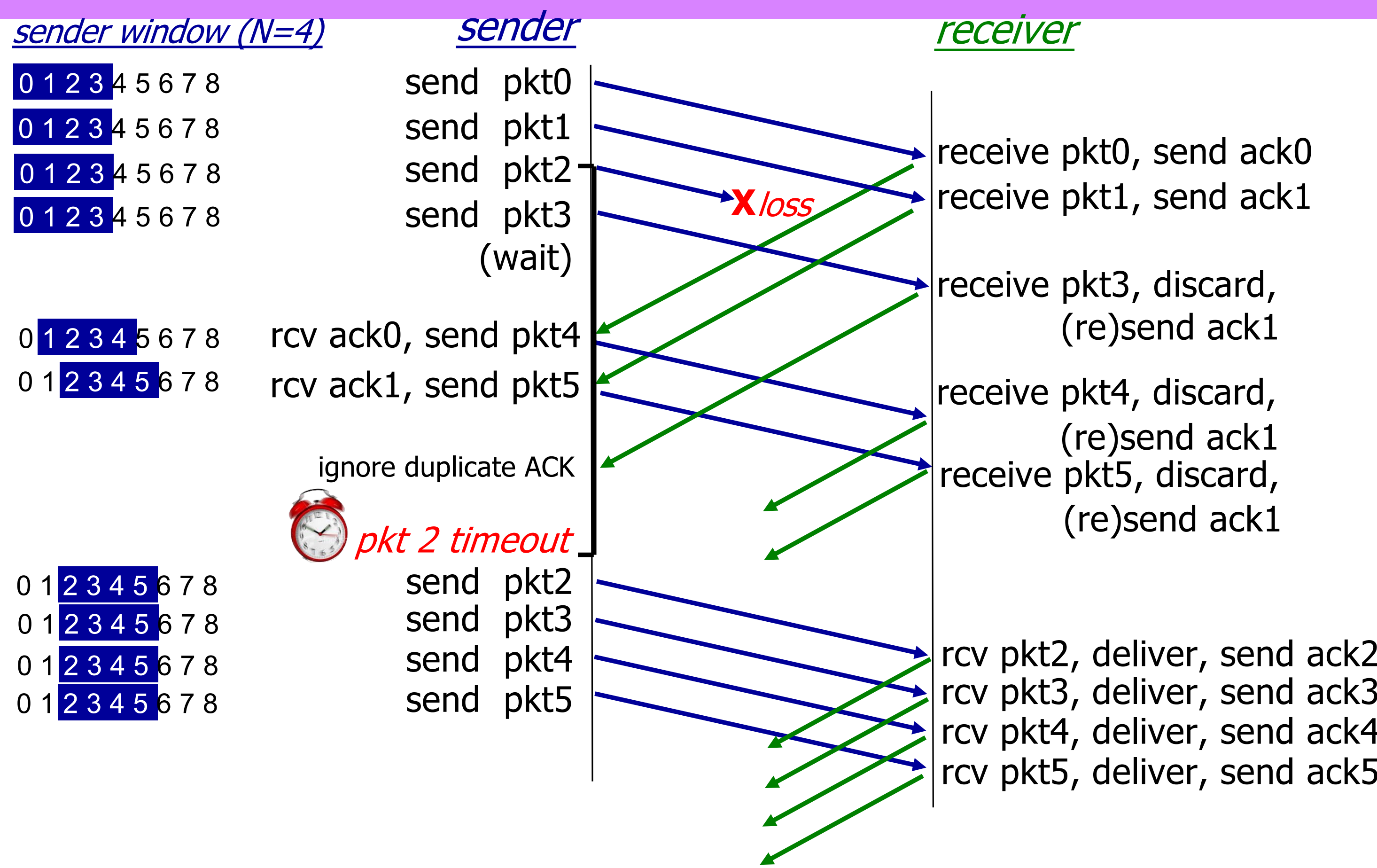
Why limit it to *N* packets?

- Flow Control (later)
- Congestion Control (later)
- Fixed size for sequence numbers
    - How many possible sequence numbers if we have a *k* bit field?

Remarks

- Receiver: Data delivered to application layer one segment at a time
  - If data from segment *n* is sent upward, it means
    - Segment *n* was received correctly, and all previous segments were received correctly and delivered to the application layer
- Receiver discards out of order segments… what a waste?
  - If segment *n* is lost, no point in keeping segment *n* + 1
  - Remember the sender sends all packets in the window!
- Event based programming
  - Call from upper layers, timer interrupts, call from lower layers

Go-Back-N in action

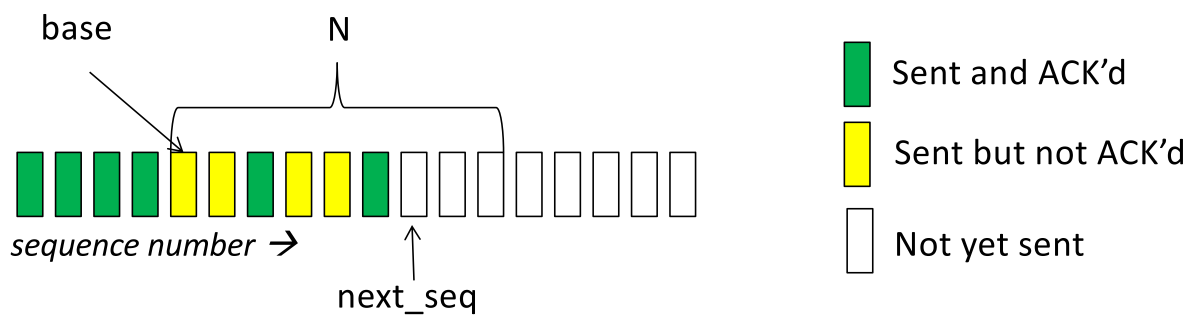


Another Approach: “**Selective Repeat**”

- Sender can send up to N unacknowledged packets
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets
- Receiver **individually** acknowledges each correctly received packet and **buffers** out-of-order packets
- Sender maintains **separate** timer for each un-ACK’d packet to retransmit individually (i.e., repeat selectively)

Another Approach: "**Selective Repeat**"

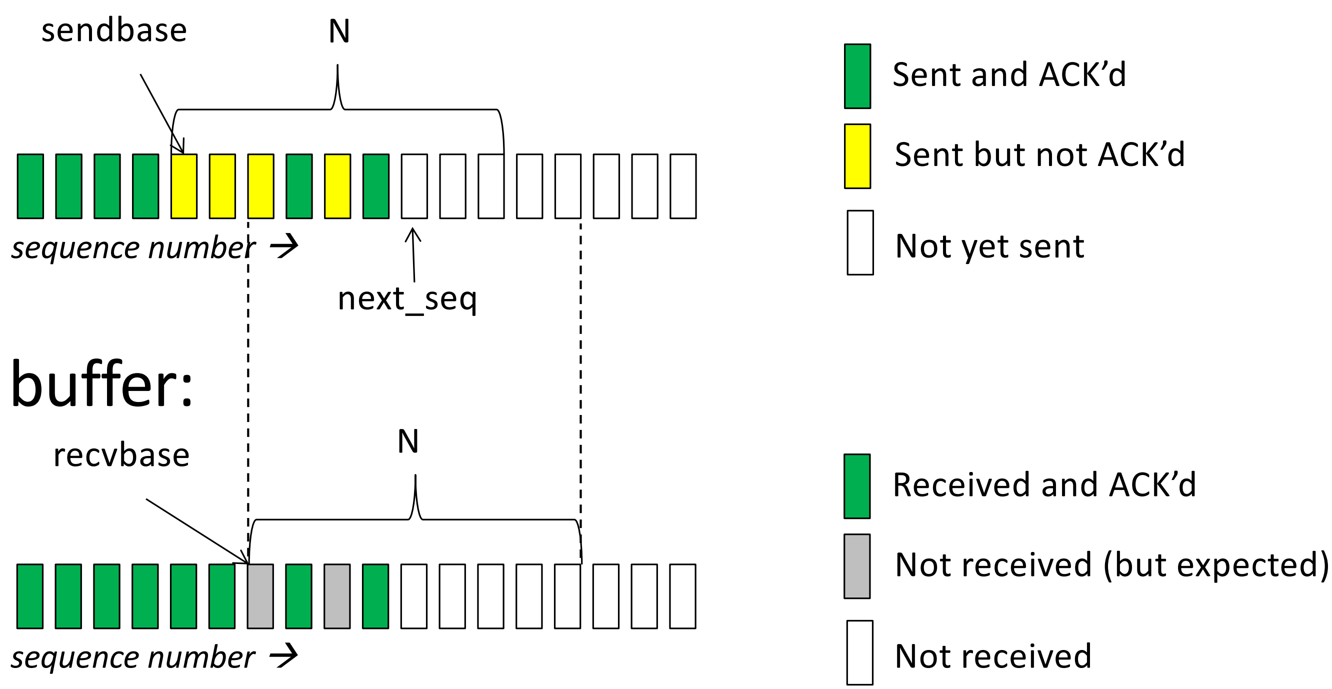
- Sender can send up to N unacknowledged packets
- Receiver **individually** acknowledges each correctly received packet and **buffers** out-of-order packets
- Sender maintains **separate** timer for each un-ACK'd packet to retransmit individually (i.e. repeat selectively)



Another Approach: "**Selective Repeat**"

- Sender buffer:

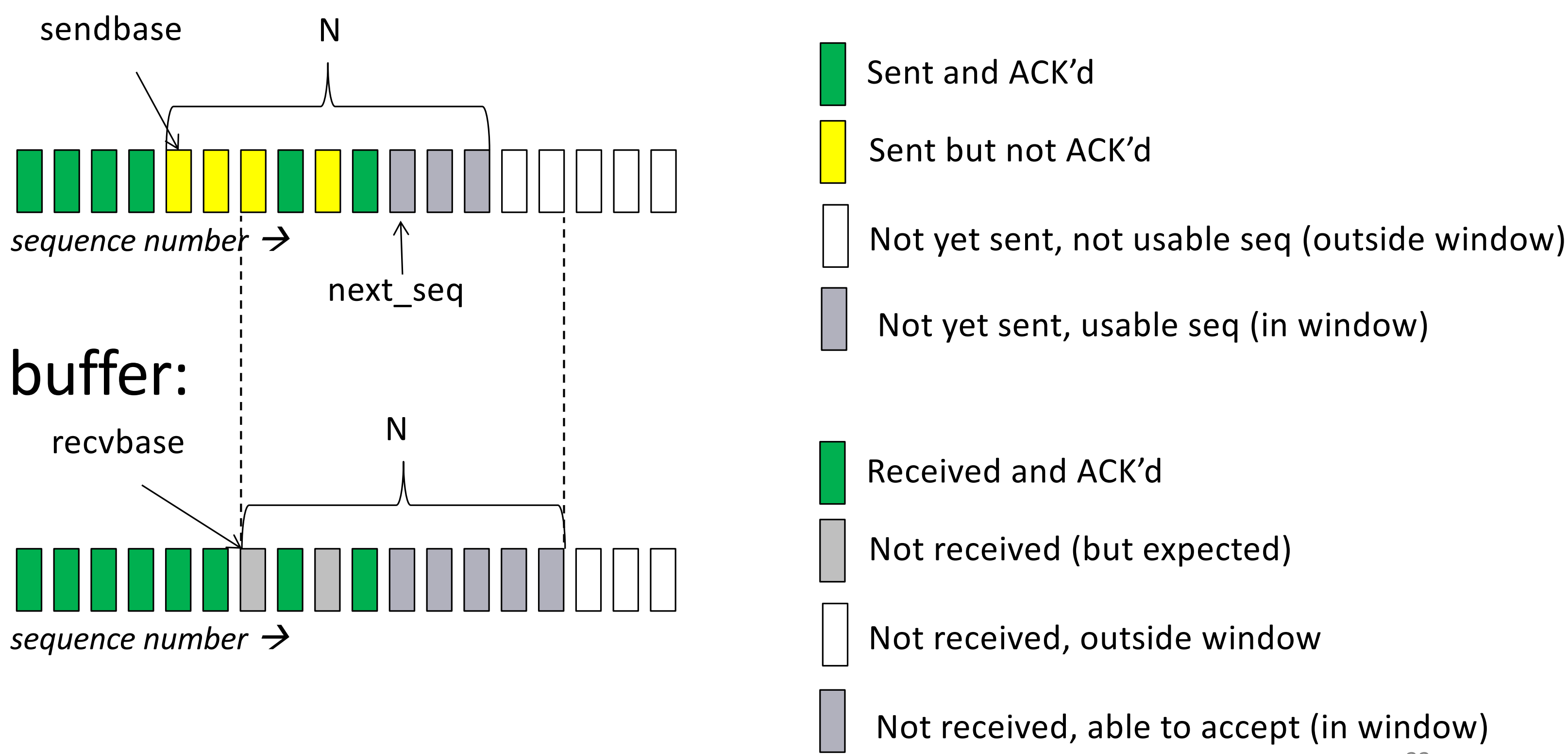
- **Receiver buffer:**



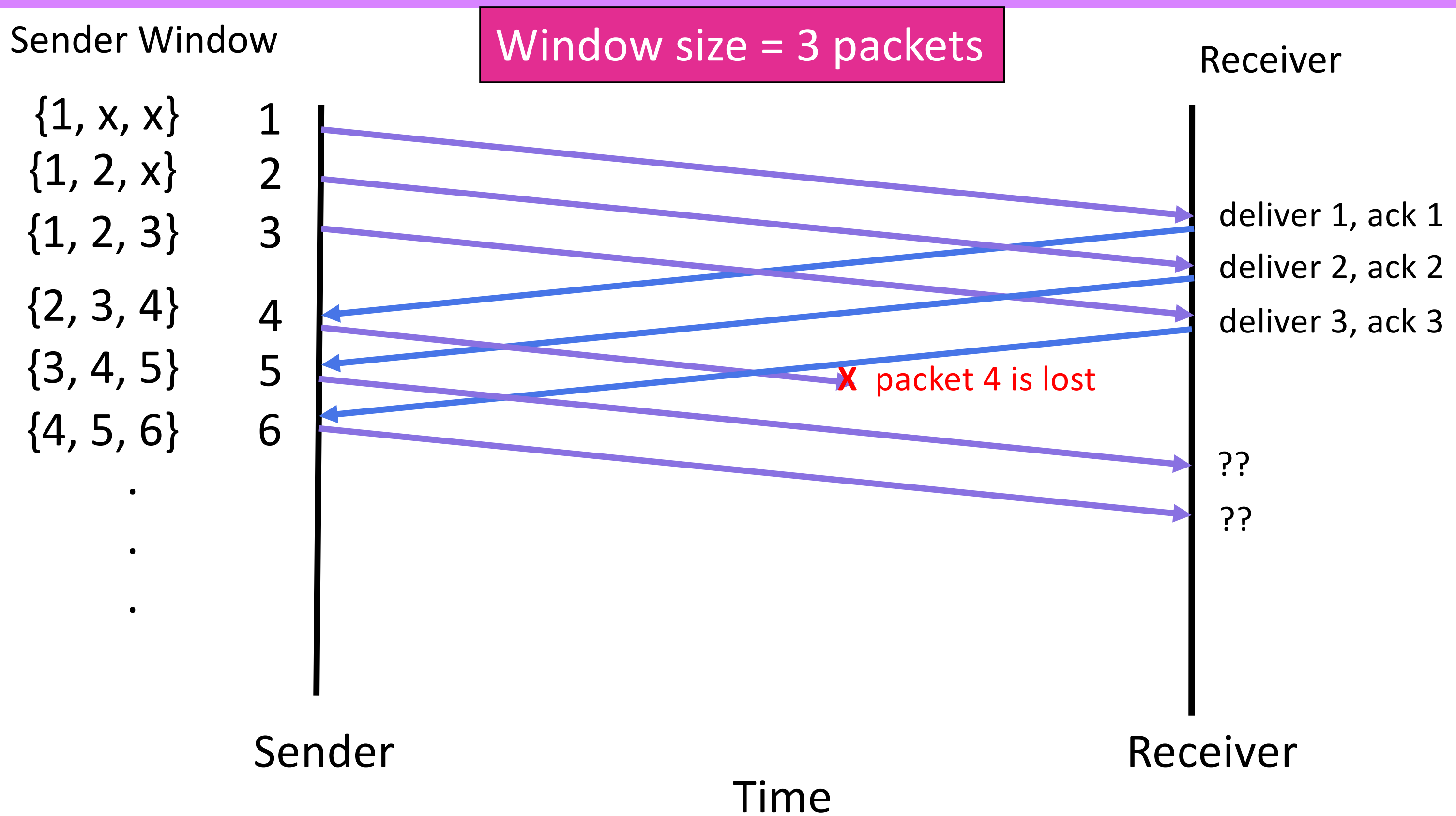
Another Approach: "Selective Repeat"

- Sender buffer:

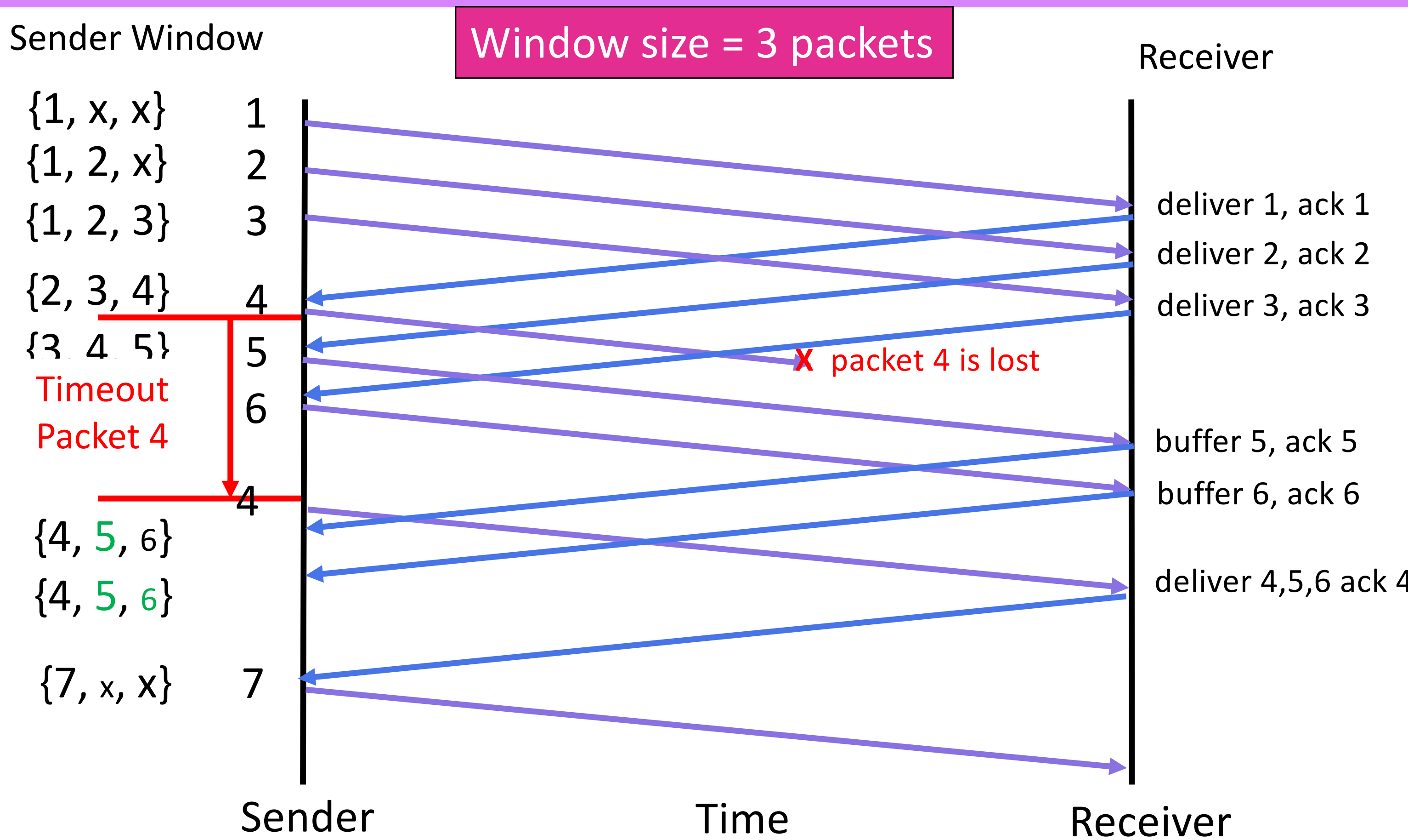
- Receiver buffer:



Selective Repeat Operation with Loss



Selective Repeat Operation with Loss



Selective Repeat Details

sender

data from above:

- if next available seq # in window, send packet & start timer

timeout(*n*):

- resend packet *n*, restart timer

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

- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

packet *n* in [rcvbase, rcvbase+N-1]

- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat Details

sender

data from above:

- if next available seq # in window

tir

-

AC

-

- if n smallest unACKed packet, advance window base to next unACKed seq #

Why?

- Why do we need to acknowledge old packets?
- Why don't we need to acknowledge more than N before our window start?

receiver

packet *n* in [rcvbase, rcvbase+N-1]

- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

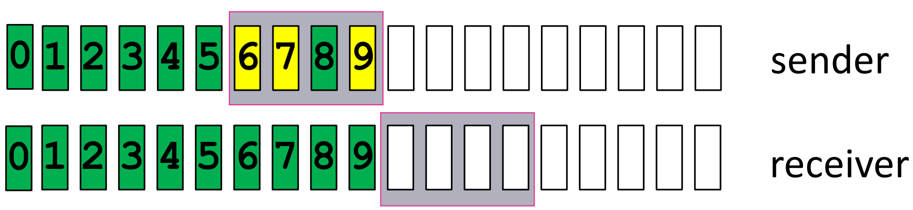
otherwise:

- ignore

Asymmetric Knowledge…

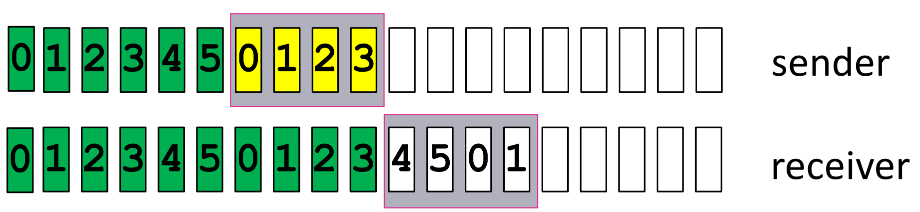
- Why do we need to acknowledge old packets?
  - Same basic reason as in our previous protocols…ACKs can get lost, so sender may not have moved up window yet (and still needs to recv ACK to do so)

- Why don't we need to acknowledge more than N before our window start?
  - Sender window can't be **too** far behind receiver window (because it can't send more than N beyond its own window start)
  - e.g. if I receive seq 9 and window size is 4, sender can't still be waiting for anything before 6



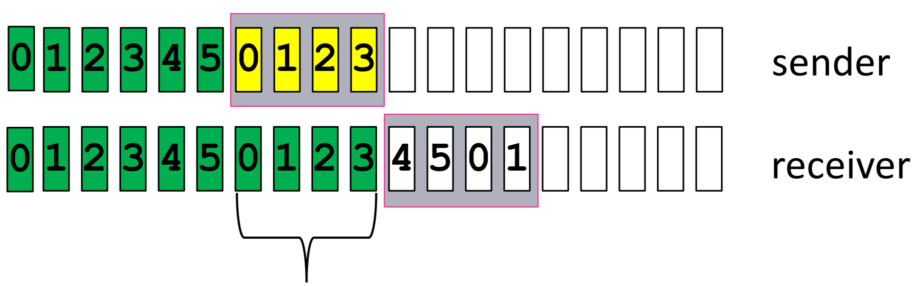
Subtle Points

- In reality, sequence numbers are **finite**
  - e.g. 16-bit sequence number -> 2^16 = 65,536 distinct numbers;
  - 32-bit sequence number -> 2^32 = 4,294,967,296 distinct numbers
- Consider sequence numbers from 0-5, window size = 4:



Subtle Points

- In reality, sequence numbers are **finite**
  - e.g. 16-bit sequence number -> 2^16 = 65,536 distinct numbers;
  - 32-bit sequence number -> 2^32 = 4,294,967,296 distinct numbers
- Consider sequence numbers from 0-5, window size = 4:



receiver has received 0-3, so it moves up its window…

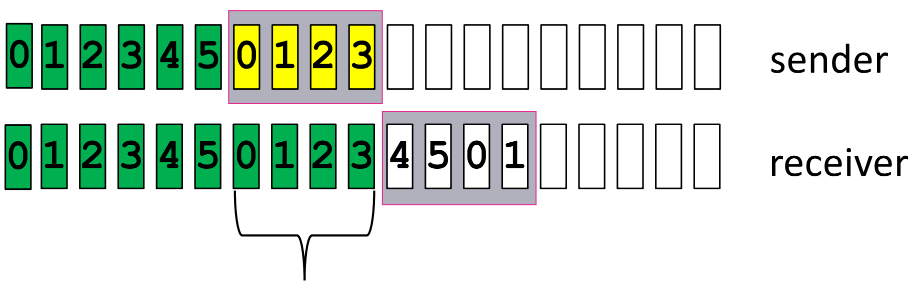
…but the sender doesn't know that yet! (e.g. because acks were lost)

So, sender will retransmit 0, 1, 2, 3

What will the receiver do?

Subtle Points

- In reality, sequence numbers are **finite**
  - e.g. 16-bit sequence number -> 2^16 = 65,536 distinct numbers;
  - 32-bit sequence number -> 2^32 = 4,294,967,296 distinct numbers
- Consider sequence numbers from 0-5, window size = 4:



receiver has received 0-3, so it moves up its window…

…but the sender doesn't know that yet! (e.g. because acks were lost)

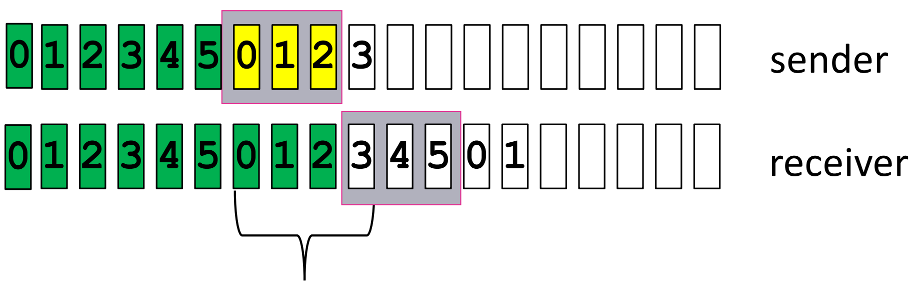
So, sender will retransmit 0, 1, 2, 3

Receiver can't differentiate retransmission from new message reusing its sequence number!

Subtle Points

- In reality, sequence numbers are **finite**
  - e.g. 16-bit sequence number -> 2^16 = 65,536 distinct numbers;
  - 32-bit sequence number -> 2^32 = 4,294,967,296 distinct numbers
- Consider sequence numbers from 0-5, window size = 4:

3



receiver has received 0-3, so it moves up its window…

…but the sender doesn't know that yet! (e.g. because acks were lost)

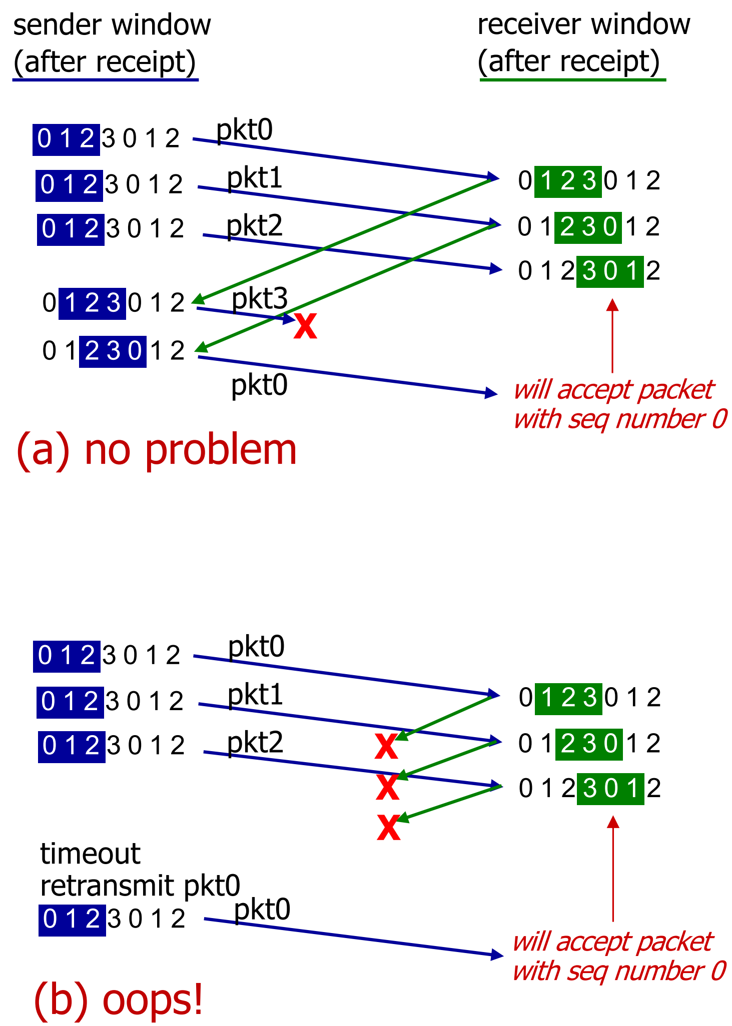
So, sender will retransmit 0, 1, 2, 3

Solution: need to shrink window OR expand sequence range

Selective repeat: a dilemma!

example:

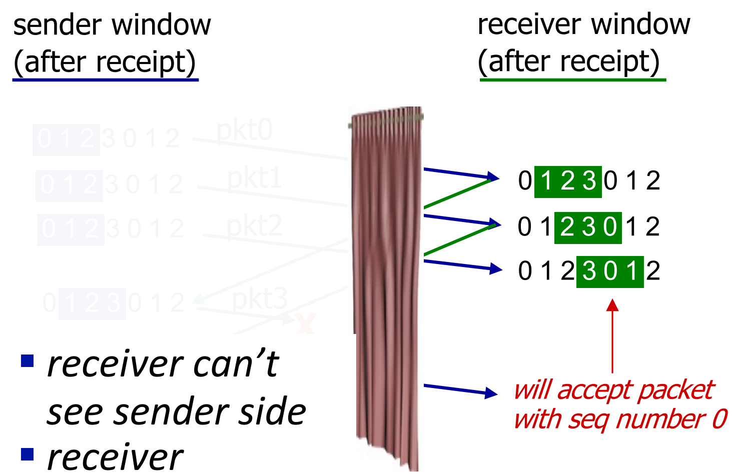
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



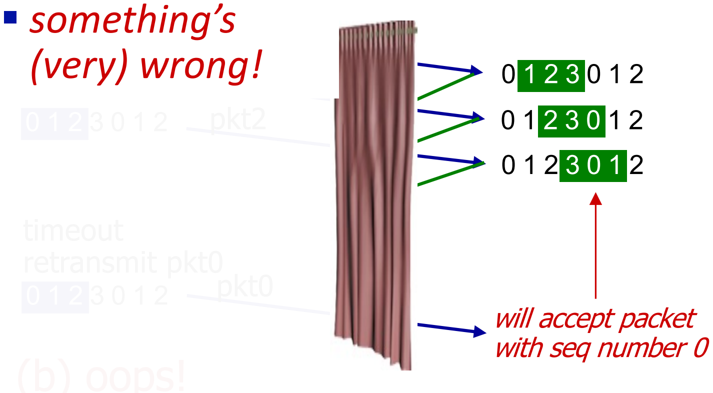
Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



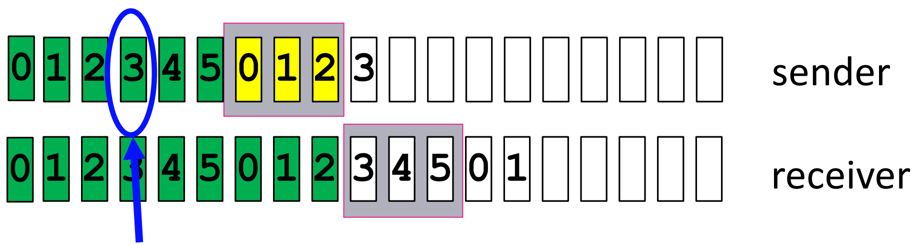
- *receiver can't see sender side*
- *receiver behavior identical in both cases!*
- *something's (very) wrong!*



Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

Subtle Points

- In reality, sequence numbers are **finite**
  - e.g. 16-bit sequence number -> 2^16 = 65,536 distinct numbers;
  - 32-bit sequence number -> 2^32 = 4,294,967,296 distinct numbers
- Consider sequence numbers from 0-5, window size = 3:



What if this message needed to be retransmitted multiple times, and some copy is **delayed** in the network such that it arrives now?

So far, we have assumed messages arrive in the **order** they were sent. **What happens if this is not true?**

Summary

- We can build a reliable transport service on top of an unreliable network
- **Pipelining** is used to improve performance and make reliable data transfer practical
- We need to be careful in reasoning about sender and receiver behavior because they may have different views of the current state (asymmetric knowledge)

Reliable Data Transfer - Recap

- Stop-and-wait
  - Send 1 packet, wait until it is acknowledged to send the next
- Go-Back-N
  - Sender can have up to N unacknowledged packets at any time
  - Receiver discards out of order packets and sends cumulative ACKs
  - Upon timeout for oldest unacknowledged packet, sender resends ALL unacknowledged packets
- Selective Repeat
  - Sender can have up to N unacknowledged packets at any time
  - Receiver buffers out of order packets (that fall within window) and sends selective ACKs
  - Upon timeout for a specific packet, sender resends that packet

TCP Service Abstraction

- **TCP is a connection-oriented protocol that delivers a reliable, in-order, byte stream**
- **Connection-oriented**: two processes coordinate (“handshake”) before beginning to send data to each other
- **Reliable**: TCP resends lost packets
  - Until it gives up and shuts down connection
- **In-order**: TCP only hands consecutive chunks of data to application
- **Byte stream**: TCP assumes there is an incoming stream of data, and attempts to deliver it to application

TCP Mechanisms

- Builds on most of what we've seen so far:
  - Checksums
  - Sequence numbers (byte offsets)
  - Sender and receiver maintain a sliding window
  - Receiver sends cumulative acknowledgements (like **Go-Back-N**)
    - Sender maintains a single retransmission timer
  - Receivers can buffer out-of-sequence packets (like **Selective Repeat**)
    - technically, this is optional…not defined by TCP spec
- And we'll see a few more soon: fast retransmit, timeout estimation algorithms

TCP Segment Structure

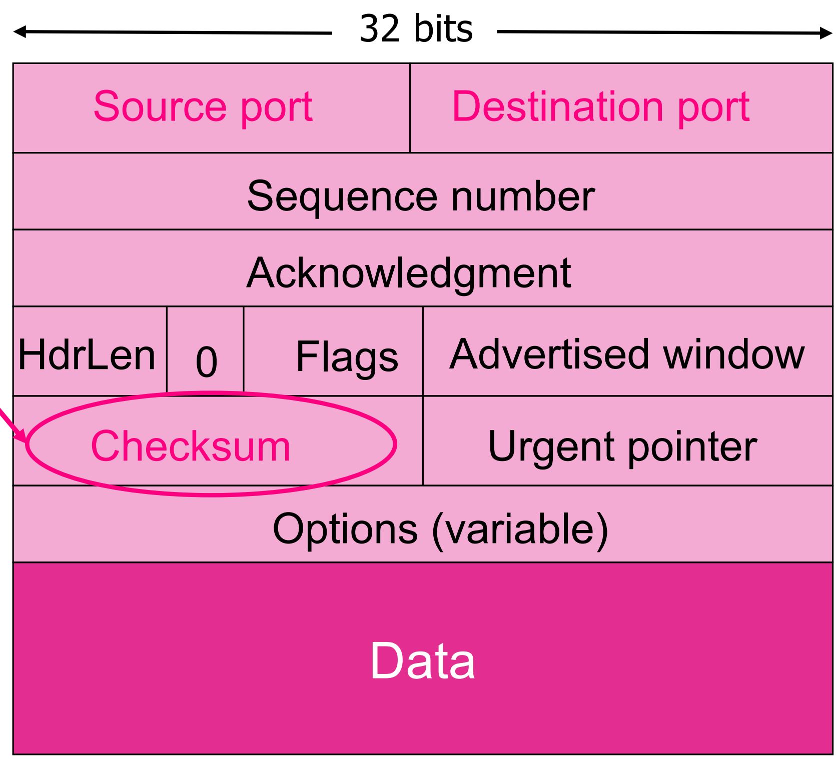
Used for Multiplexing / Demultiplexing (similar to UDP, but remember that TCP demultiplexing actually uses source port/IP)

32 bits

Source port			Destination port
Sequence number			
Acknowledgment			
HdrLen	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			
Data			

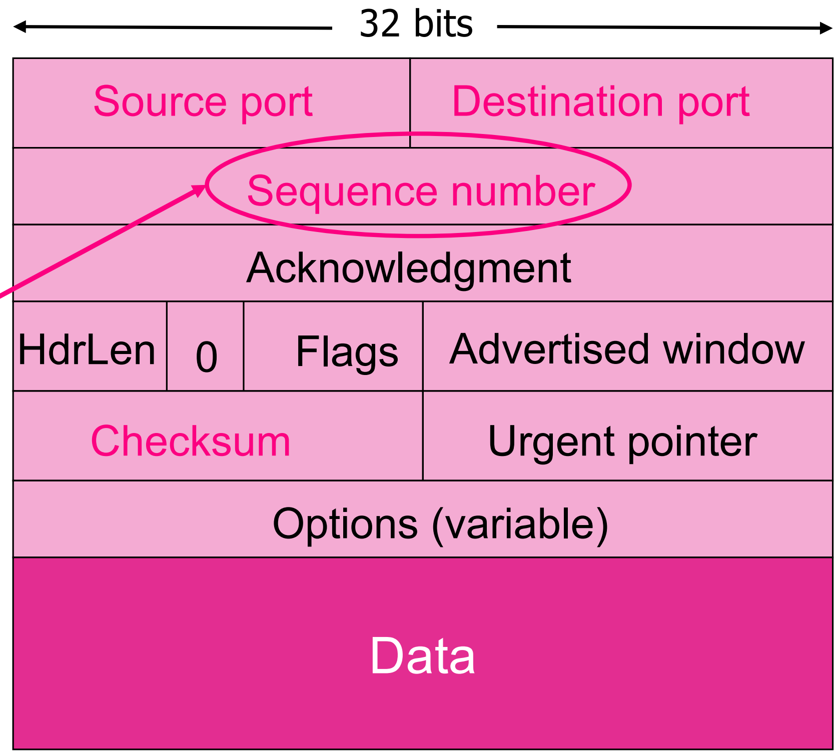
TCP Segment Structure

Used for error detection (same as UDP)



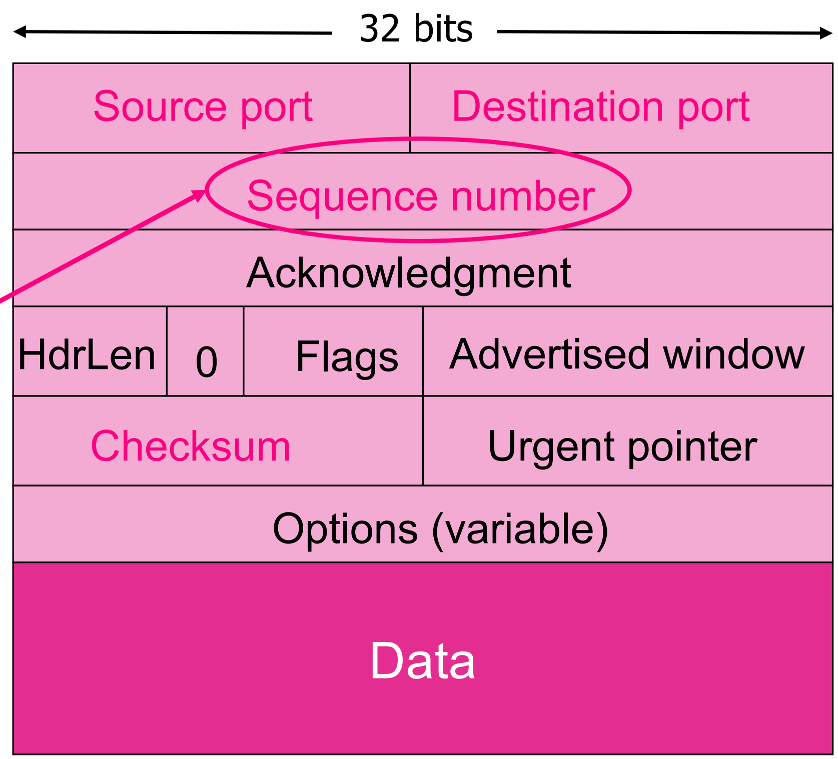
TCP Segment Structure

As discussed last time, used for maintaining **ordered** delivery and differentiating new vs duplicate data



TCP Segment Structure

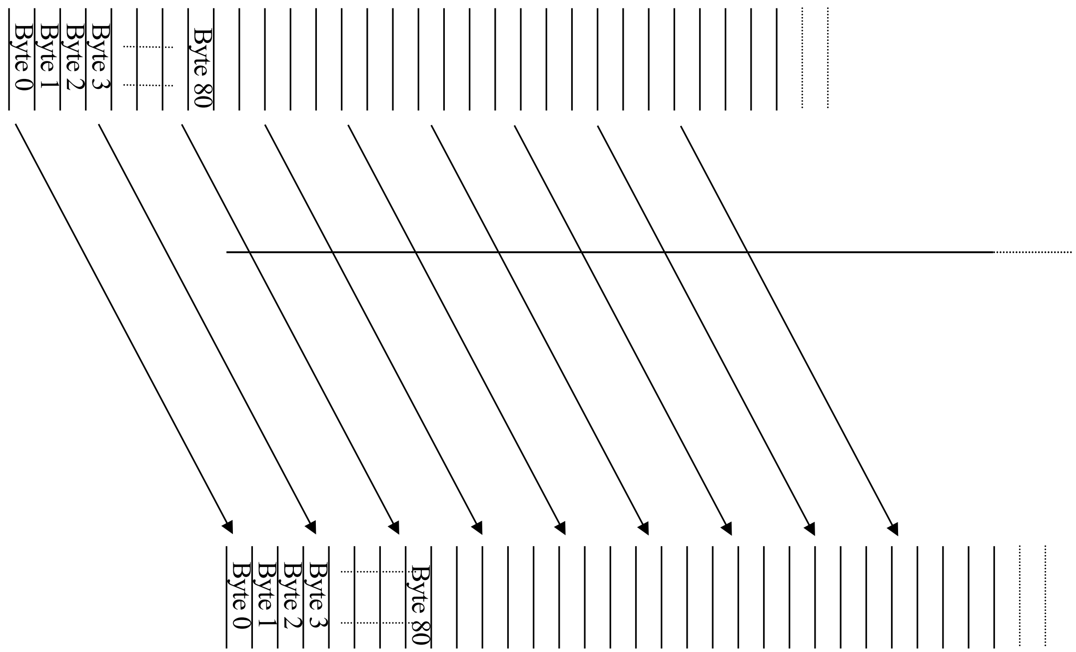
As discussed last time, used for maintaining **ordered** delivery and differentiating new vs duplicate data



But, for TCP, the sequence number is a **byte offset**, **not** a packet id

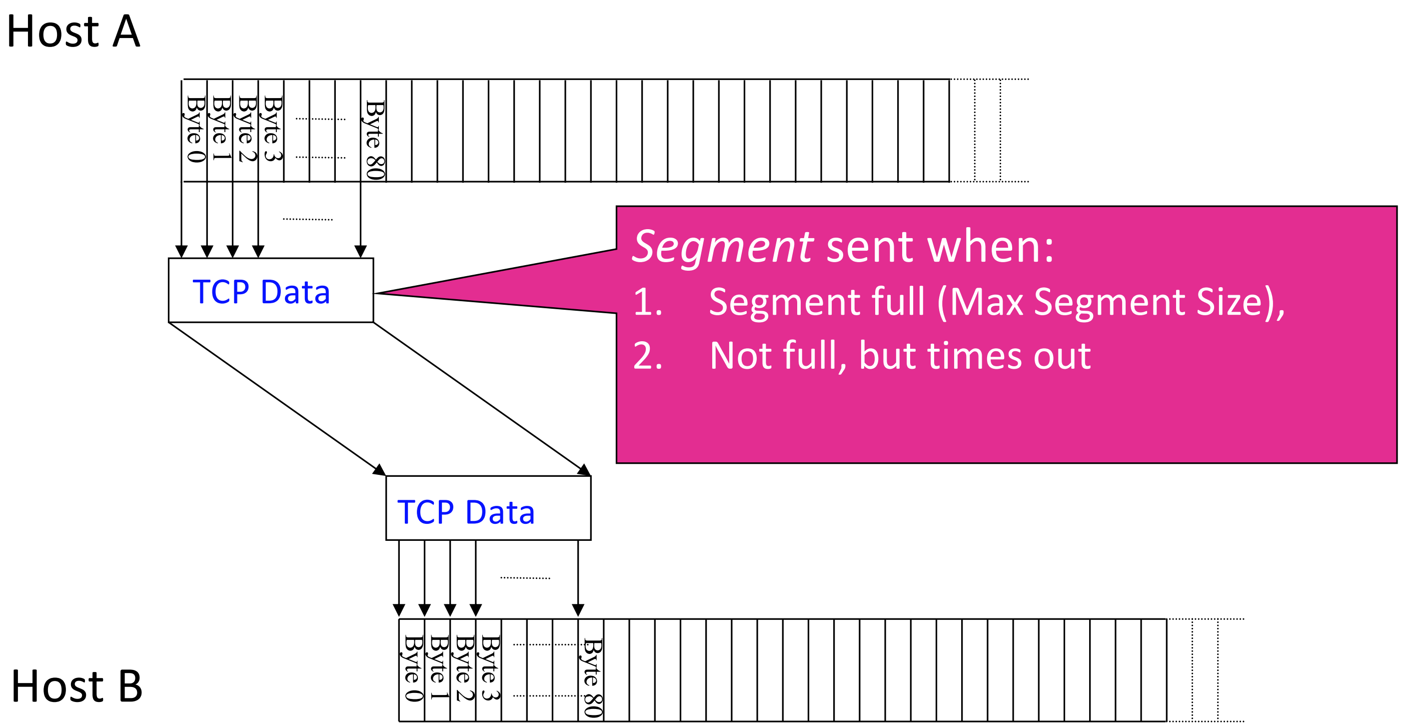
TCP Byte Stream Service…

Application @ Host A



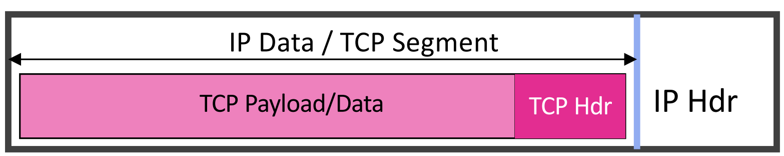
Application @ Host B

…provided using TCP segments

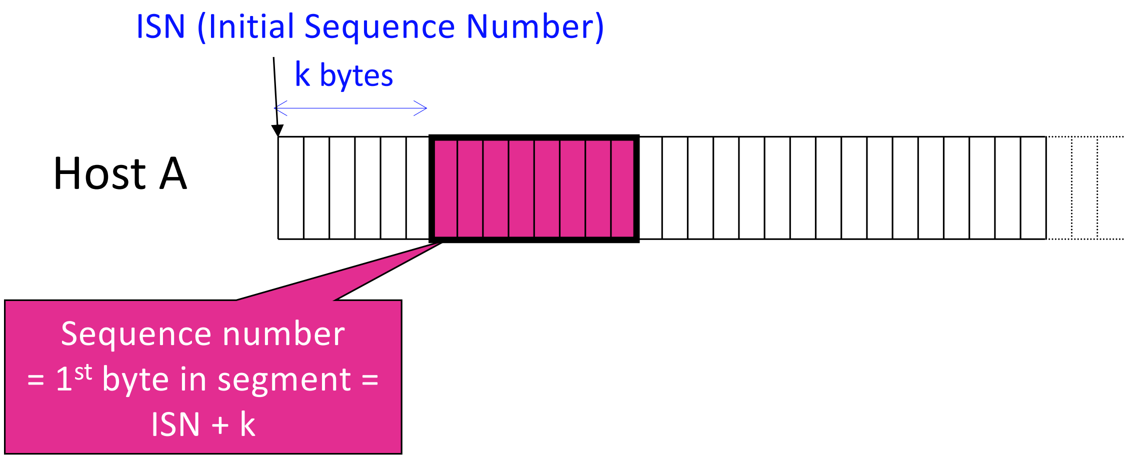


TCP Segments

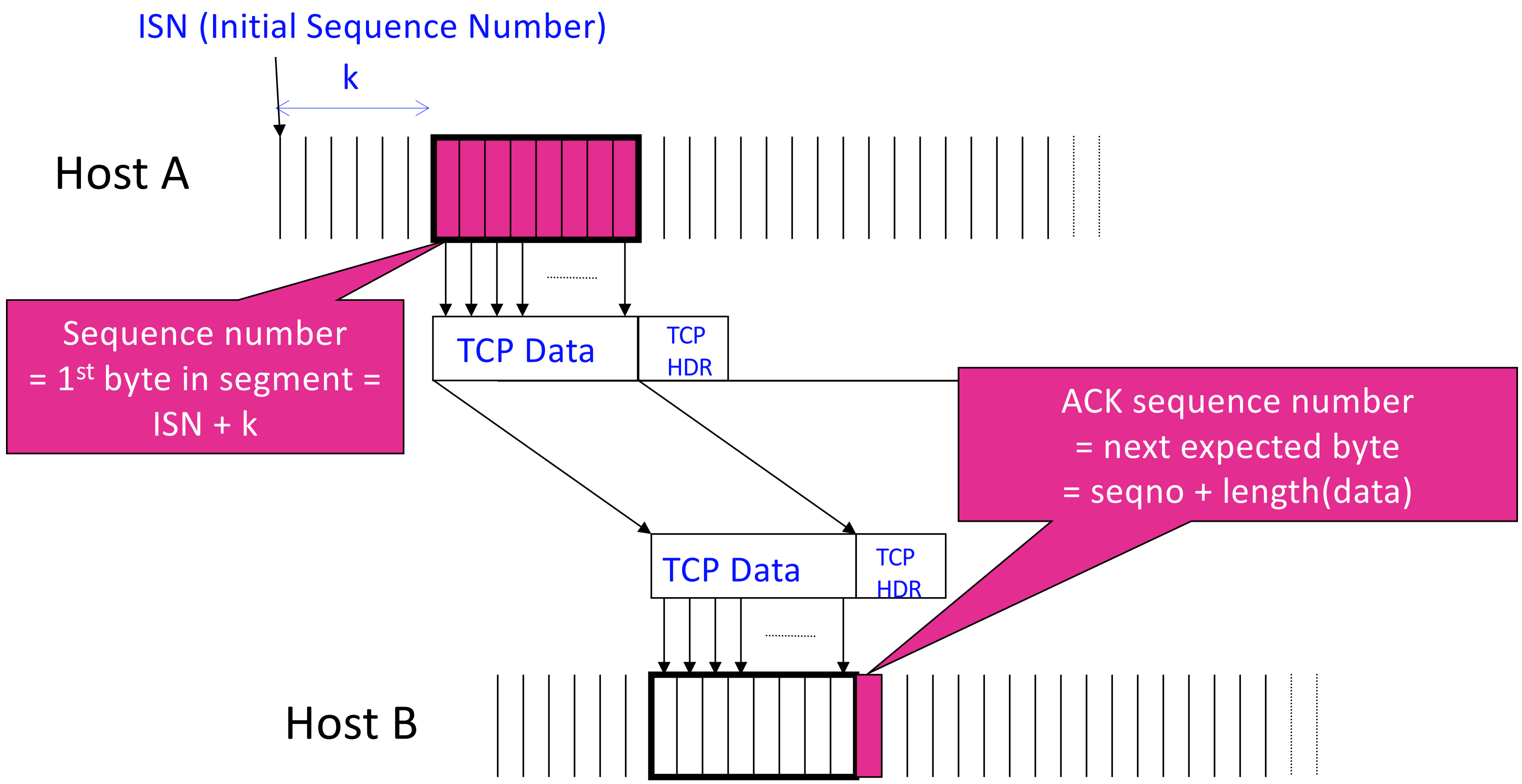
- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- TCP segment
  - IP payload contains segment composed of TCP header and data
  - TCP header ≥ 20 bytes long
- TCP payload/data
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)



TCP Sequence Numbers

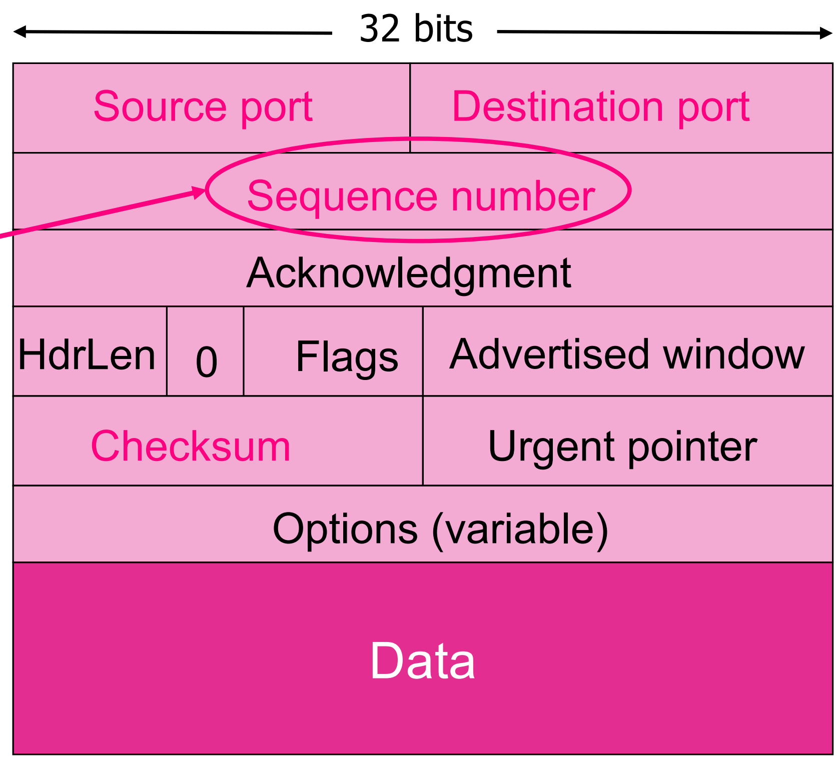


TCP Sequence Numbers



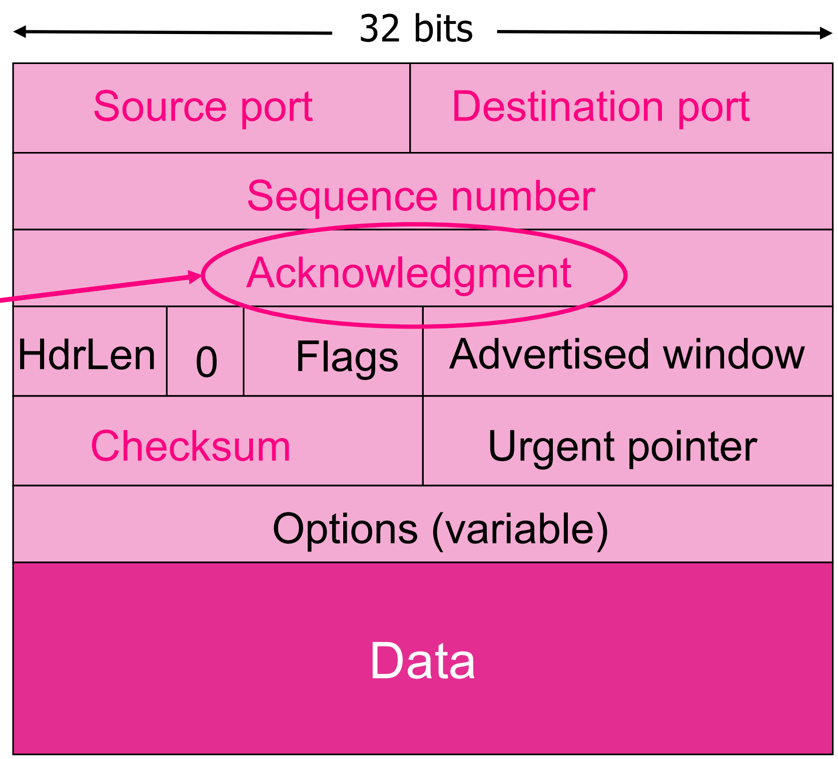
TCP Segment Structure

Starting byte offset of data carried by this segment



TCP Segment Structure

Sequence of **next expected** byte (i.e. sequence number just **after** the last byte received **in order** so far)



TCP ACK Behavior

- Sender sends segment
  - Data starts with sequence number X
  - Segment contains B bytes [X, X+1, X+2, …, X+B-1]
- Upon receipt of segment, receiver sends an ACK
  - If all data prior to X already received:
    - ACK acknowledges X+B (because that is next expected byte)
  - If highest in-order byte received is Y s.t. (Y+1) < X
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

Why?

TCP ACK Behavior: Normal (no loss) Case

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

- Sequence number of next packet is same as last ACK field

TCP ACK Behavior: Loss Case

- **Sender sends packets with 100B and sequence numbers**:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …
- **Assume the fifth packet (seqno 500) is lost, but no others**
- **Stream of ACKs will be**:
  - 200, 300, 400, 500, 500 (seqno:600), 500 (seqno:700), 500 (seqno:800), 500 (seqno:900),…
  - Acknowledgements are **cumulative**

Reliable Data Transfer with TCP

- To recover a lost packet, sender needs to **retransmit**
- What triggers retransmission?
  - Our usual mechanism: **Timeout**
  - Sender maintains a **single timer** for **oldest unacknowledged segment** (like Go-Back-N)
  - On timeout, retransmit **only** that oldest unacknowledged segment (closer to Selective Repeat)

Reliable Data Transfer with TCP: Setting Timeouts

- How long should the timeout be?
  - Too long: slow reaction to loss
  - Too short: waste bandwidth retransmitting packets that were not really lost
- **Goal:** timeout should be *close* to RTT
  - Definitely can't be shorter (many unnecessary retransmissions)
  - But, much longer will make reactions slow

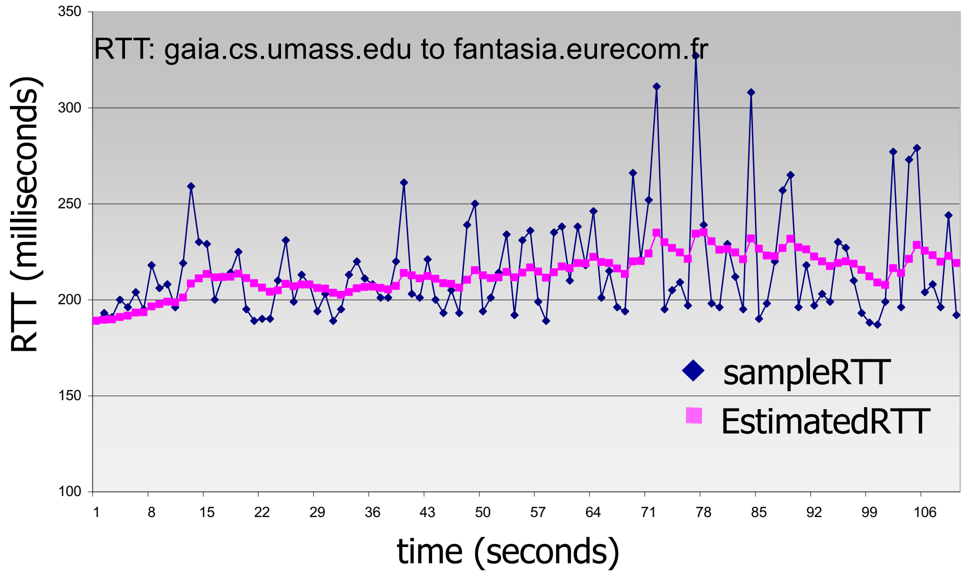
RTT Estimation

- We want timeout close to RTT, but…how do we know what the RTT is?
- We can measure it!
  - `SampleRTT:` measured time from segment transmission until ACK receipt
  - But measurements will vary over time…

RTT Estimation

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

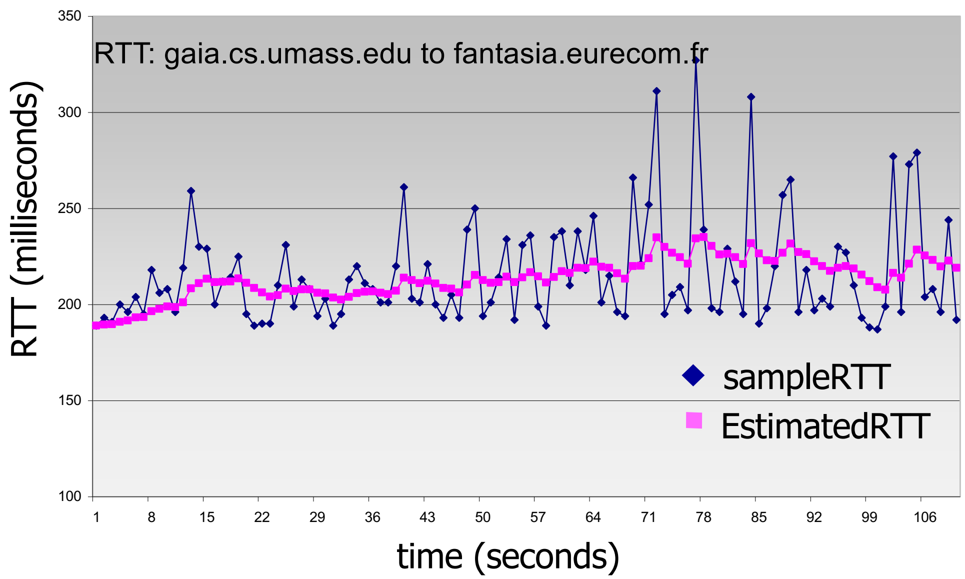
- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



RTT Estimation

`EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT`

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



Example:

- Our current RTT estimate is 100ms
- New SampleRTT is 200ms
- What is updated EstimatedRTT?

.875*100 + .125*200
= 112.5ms

TCP Timeout Setting

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT:`** want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

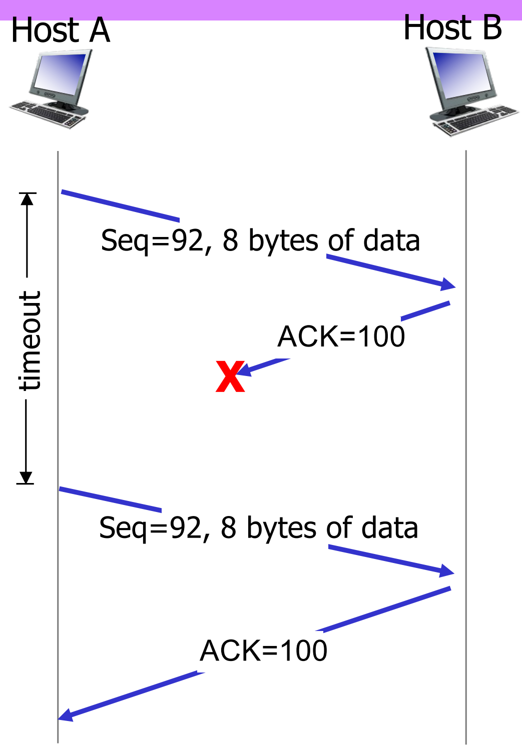
$$\texttt{DevRTT} = (1-\beta)*\texttt{DevRTT} + \beta*|\texttt{SampleRTT-EstimatedRTT}|$$

(typically, β = 0.25)

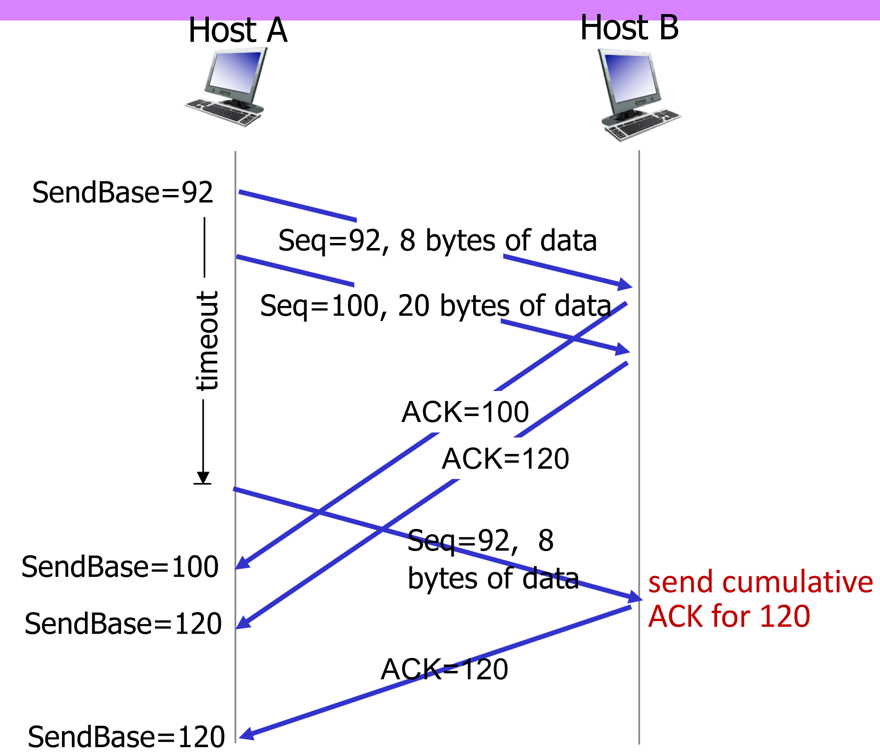
TCP Timeouts: Details

- **Retransmitted** segments are **ignored** for SampleRTT calculations
  - Eliminates ambiguity of whether ACK was sent in response to original or retransmission
- What value should the timeout start at? (before any SampleRTT is measured)
  - Recommendation: 1 second (https://tools.ietf.org/html/rfc6298)
- Timeout is **doubled** when it expires
  - More on congestion control soon…

TCP Retransmission Scenarios

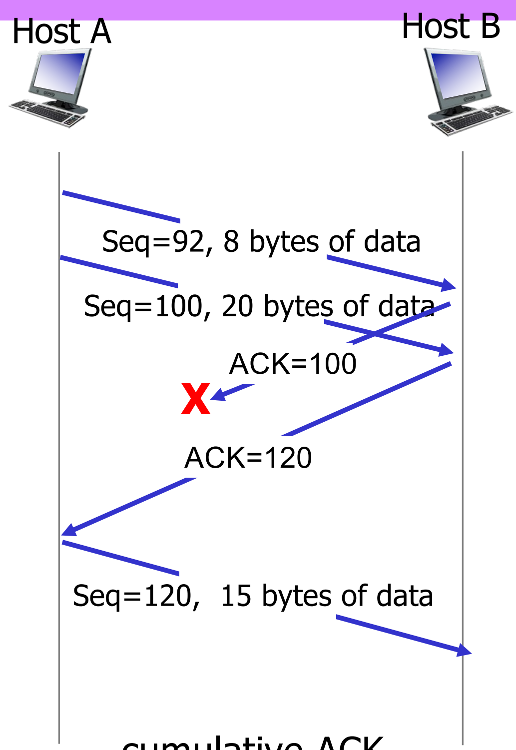


lost ACK scenario



premature timeout

TCP Retransmission Scenarios



cumulative ACK **covers** for earlier lost ACK

Reliable Data Transfer with TCP

- To recover a lost packet, sender needs to **retransmit**
- What triggers retransmission?
  - Our usual mechanism: **Timeout**
    - Sender maintains a **single timer** for **oldest unacknowledged segment** (like Go-Back-N)
    - On timeout, retransmit **only** that oldest unacknowledged segment (closer to Selective Repeat)

Reliable Data Transfer with TCP

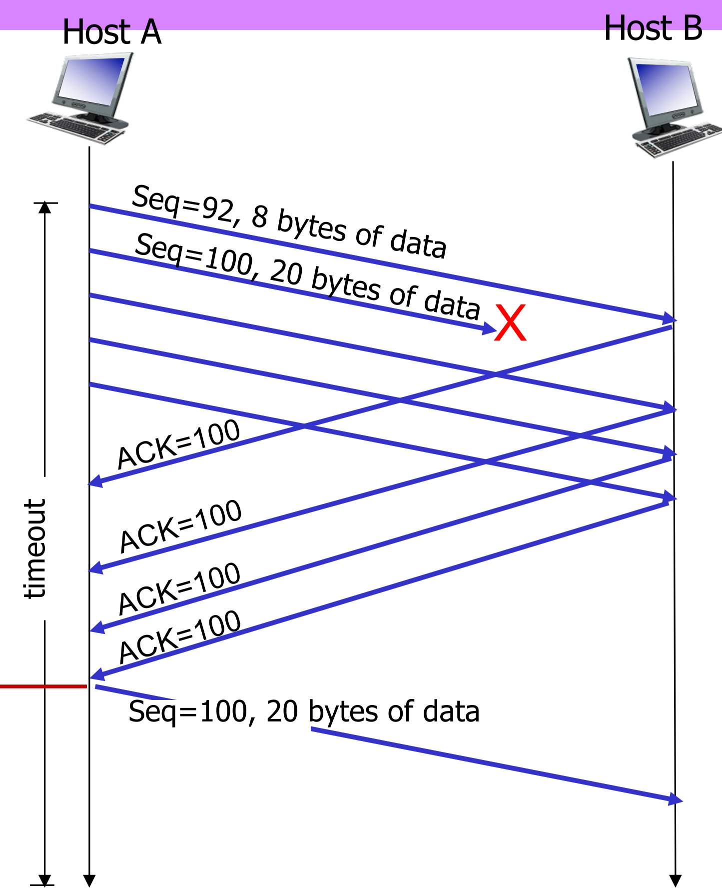
- To recover a lost packet, sender needs to **retransmit**
- What triggers retransmission?
  - Our usual mechanism: **Timeout**
    - Sender maintains a **single timer** for **oldest unacknowledged segment** (like Go-Back-N)
    - On timeout, retransmit **only** that oldest unacknowledged segment (closer to Selective Repeat)
  - An optimization: **Fast Retransmit**

TCP Fast Retransmit

- Recall our loss scenario:
  - Sender sends packets with 100B and seqnos.:
    - 100, 200, 300, 400, 500, 600, 700, 800, 900, …
  - Assume the fifth packet (seqno 500) is lost, but no others
  - Stream of ACKs will be:
    - 200, 300, 400, 500 (seqno:600), 500 (seqno:700), 500 (seqno:800), 500 (seqno:900),…
- **Duplicate ACKs are a sign of isolated loss**
  - 500 clearly hasn't been delivered…but other segments are getting through

TCP Fast Retransmit

- TCP will retransmit a segment upon receiving **3 duplicate ACKs** for that segment
  - without waiting for timeout



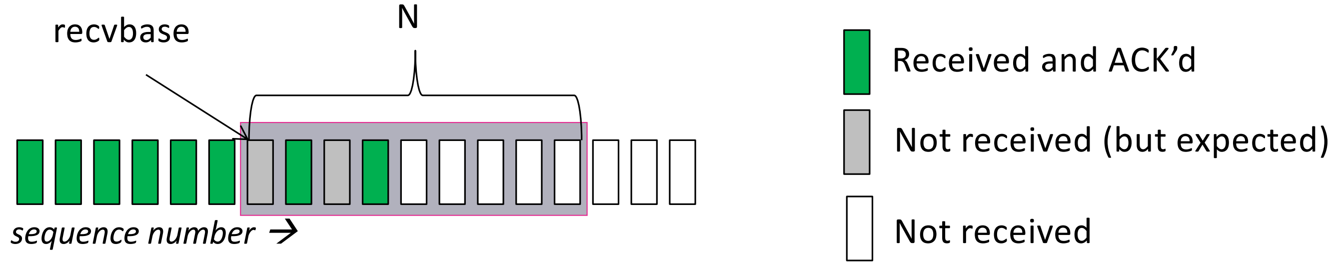
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

TCP Flow Control

- **Flow Control** key idea: sender should not transmit faster than the receiver can process!

Flow Control

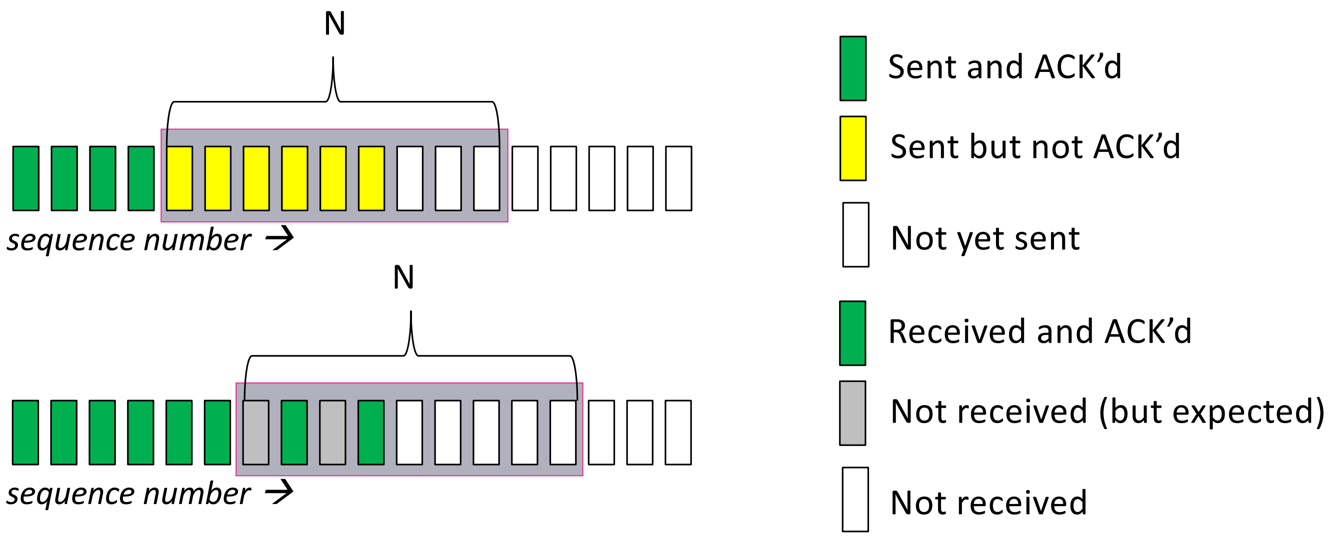
- Recall our discussion from last week on reliable data transfer
  - Receiver maintains a **buffer** that stores received but not-yet-delivered data



  - Buffers have finite storage space, so what would happen if sender sends more data than the receiver can fit in its buffer?

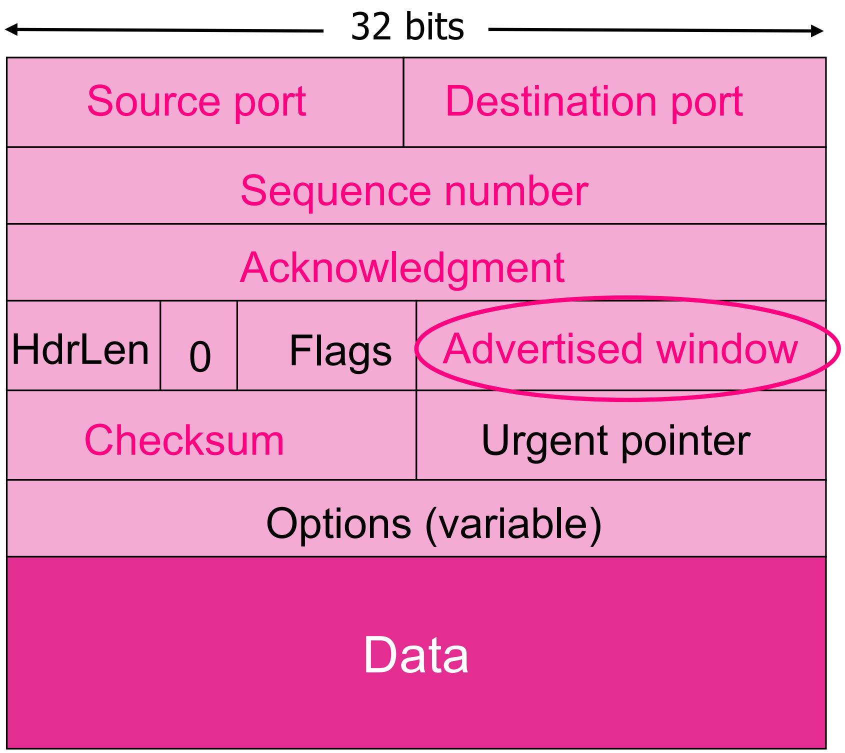
Flow Control

- We saw a **limited form** of flow control in the Go-Back-N and Selective Repeat protocols we looked at:
  - Receiver maintains a **buffer** that **can hold N packets**
  - Sender is allowed to have **up to N** unacknowledged packets out at any time

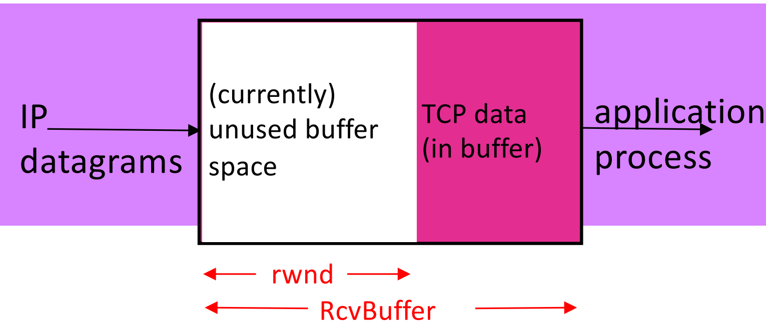


TCP Flow Control

- More flexible, but same basic idea: **sender only sends as much data as it knows receiver can accept**
- Receiver uses an "Advertised Window" (RWND) to tell the sender how many bytes it can accept
  - Receiver indicates value of RWND in ACKs
  - Sender ensures that the total number of bytes in flight <= RWND



TCP Flow Control



- Advertised window limits rate: Sender can send no faster than RWND/RTT bytes/sec
- Receiver only advertises more space when application has consumed old arriving data
- What happens when RWND=0?
  - Sender keeps probing by sending segments with one byte of data
  - Receiver can ACK, but won't increase RWND until application reads some data and buffer space opens up

```
rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]
```

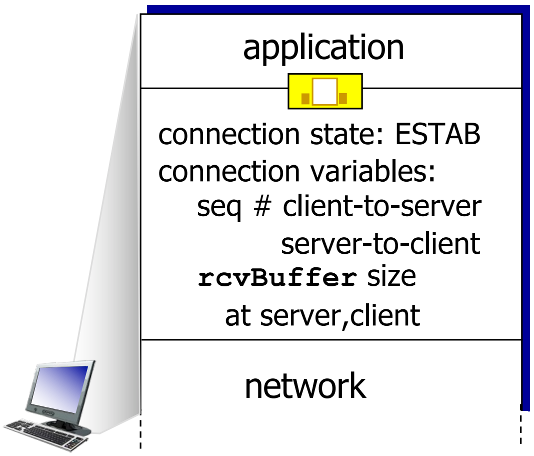
Recall: TCP Service Abstraction

- **TCP is a connection-oriented protocol that delivers a reliable, in-order, byte stream**

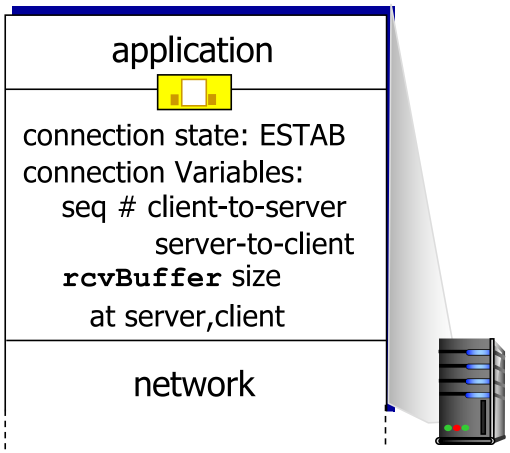
- **Connection-oriented**: two processes coordinate ("handshake") before beginning to send data to each other
- **Reliable**: TCP resends lost packets
  - Until it gives up and shuts down connection
- **In-order**: TCP only hands consecutive chunks of data to application
- **Byte stream**: TCP assumes there is an incoming stream of data, and attempts to deliver it to application

TCP Connection Management

- Before exchanging data, sender/receiver "handshake":
  - agree to establish connection (each knowing the other willing to establish connection)
  - agree on connection parameters (e.g., starting seq #s, initial buffer sizes)




```
clientSocket.connect("hostname,
  "port number");
```



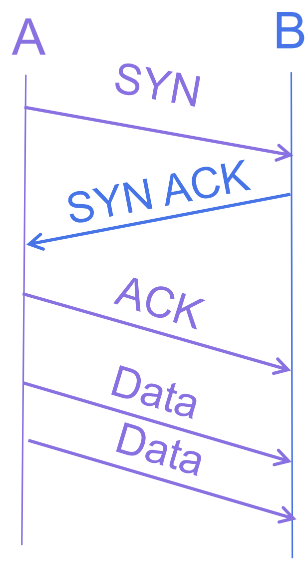

```
connectionSocket =
  welcomeSocket.accept();
```

Connection Parameters: Initial Sequence Number

- Sequence number for the **very first byte**
- **Why not just use ISN = 0?**
  - Practical issue
    - IP addresses and port #s uniquely identify a connection
    - Eventually, though, these port #s do get used again; small chance an old packet is still in flight
    - Also, others might try to spoof your connection
  - Why does using ISN help?
- Hosts exchange ISNs when establishing connection

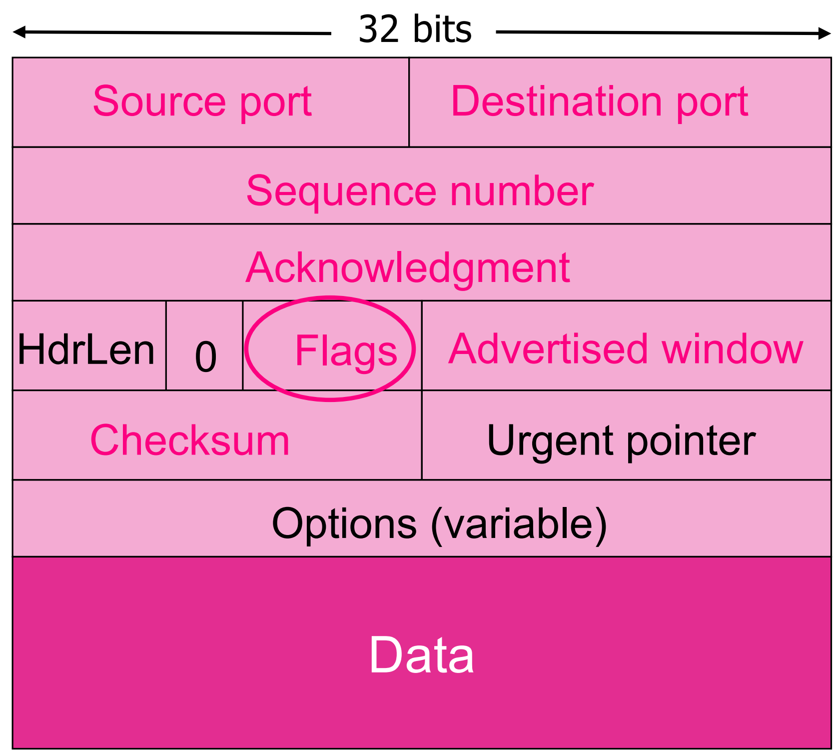
Establishing a TCP Connection

- Three-way handshake to establish connection
  - Host A sends a SYN (open; "synchronize sequence numbers") to host B
  - Host B returns a SYN acknowledgment (SYN ACK)
  - Host A sends an ACK to acknowledge the SYN ACK



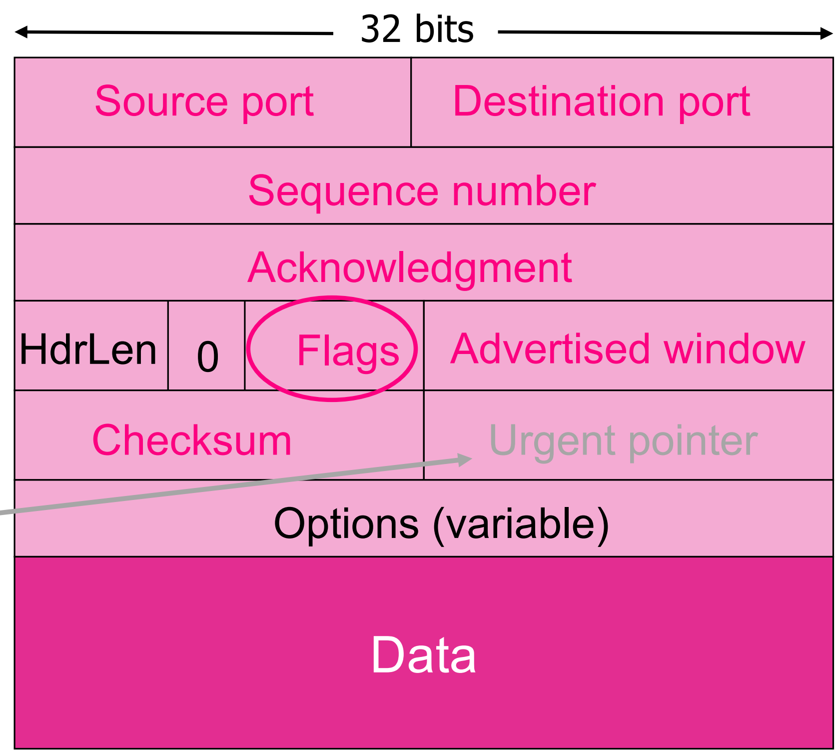
Establishing a TCP Connection

Flags:
SYN
ACK
FIN
RST
PSH
URG
CWR
ECE



Establishing a TCP Connection

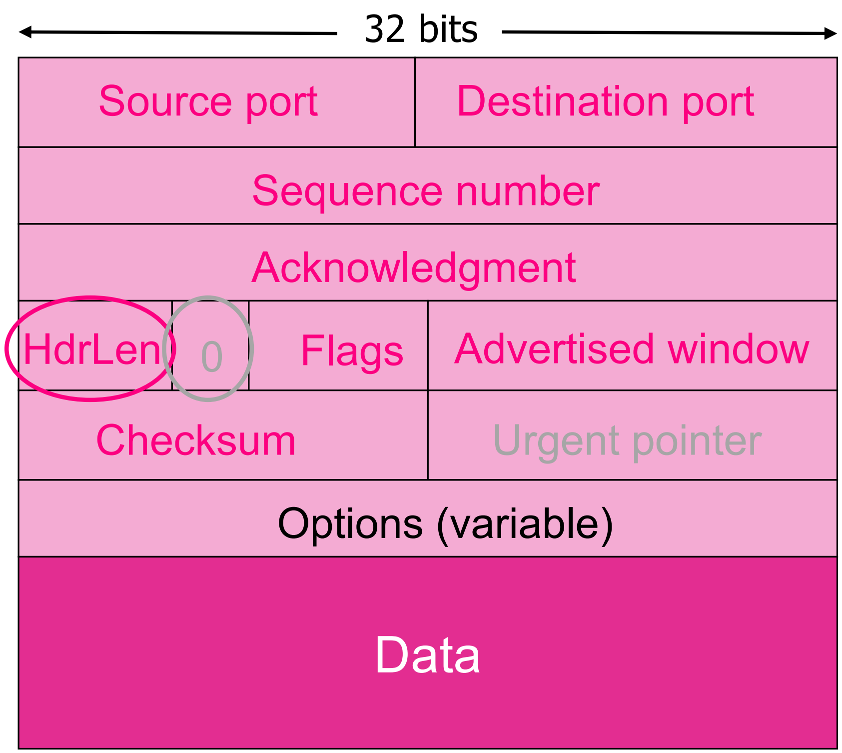
Flags:
SYN – connection setup
ACK – contains valid ack
FIN – connection teardown
RST – connection teardown
PSH – pass data up immediately
URG – data marked urgent
CWR – congestion control
ECE – congestion control



Wrapping up TCP Segment Structure

Length of header in 32-bit words

Unused



Step 1: Host A's SYN

A tells B to open
a connection

A's port		B's port	
A's Initial Sequence Number			
N/A			
5	0	SYN	Advertised window
Checksum		Urgent pointer	

Step 2: Host B's SYN-ACK

B tells A it accepts and is ready to accept next packet

B's port		A's port	
B's Initial Sequence Number			
ACK=A's ISN+1			
5	0	SYN|ACK	Advertised window
Checksum		Urgent pointer	

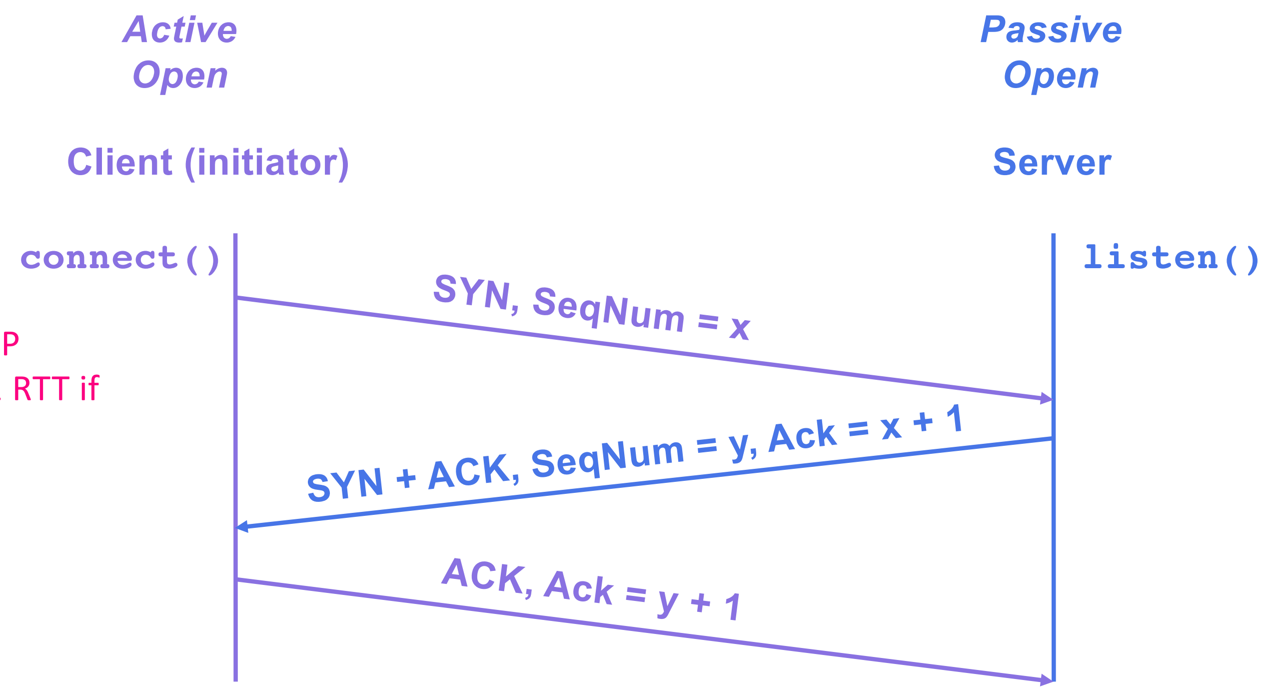
B's Initial Sequence Number = A's Initial Sequence Number?

Step 3: Host A’s ACK of B’s SYN-ACK

A tells B to open a connection

A’s port		B’s port	
A’s Initial Sequence Number			
ACK=B’s ISN+1			
5	0	ACK	Advertised window
Checksum		Urgent pointer	

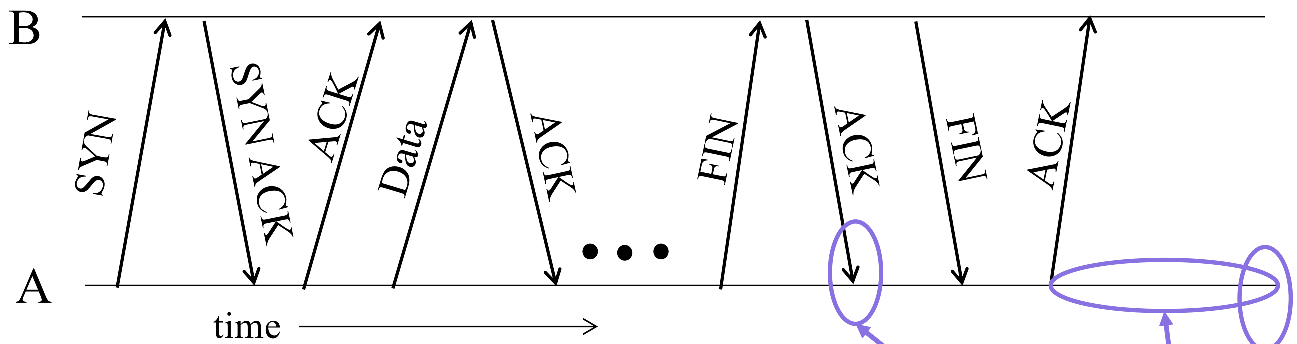
TCP 3-way Handshake



Why did we say TCP connection takes 1 RTT if there are really 3 delays?

Last ACK can be "piggybacked" on first data packet

Closing a TCP Connection: Normal Termination Example



- Finish (FIN) to close and receive remaining bytes
  - FIN occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A's side of the connection, but not B's
  - Until B likewise sends a FIN
  - Which A then acks

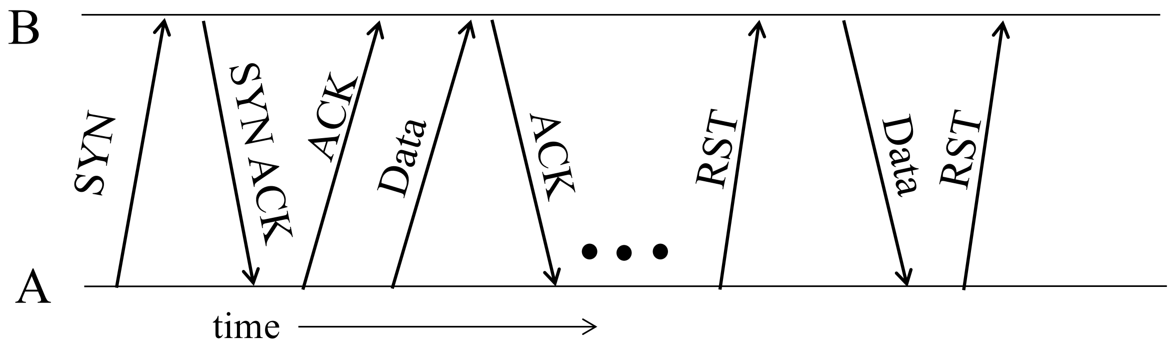
Connection now **half-closed**

Connection now **closed**

TIME_WAIT:

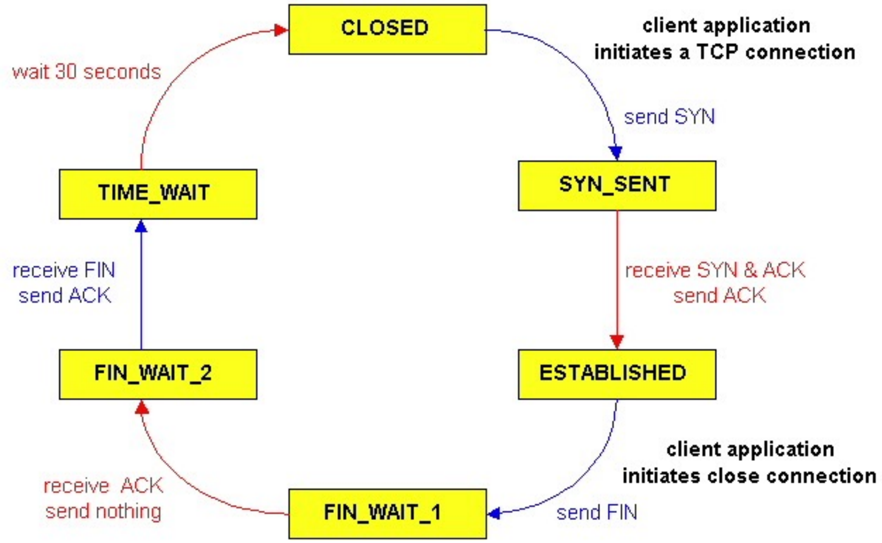
B will retransmit FIN if ACK is lost

Closing a TCP Connection: Abrupt Termination



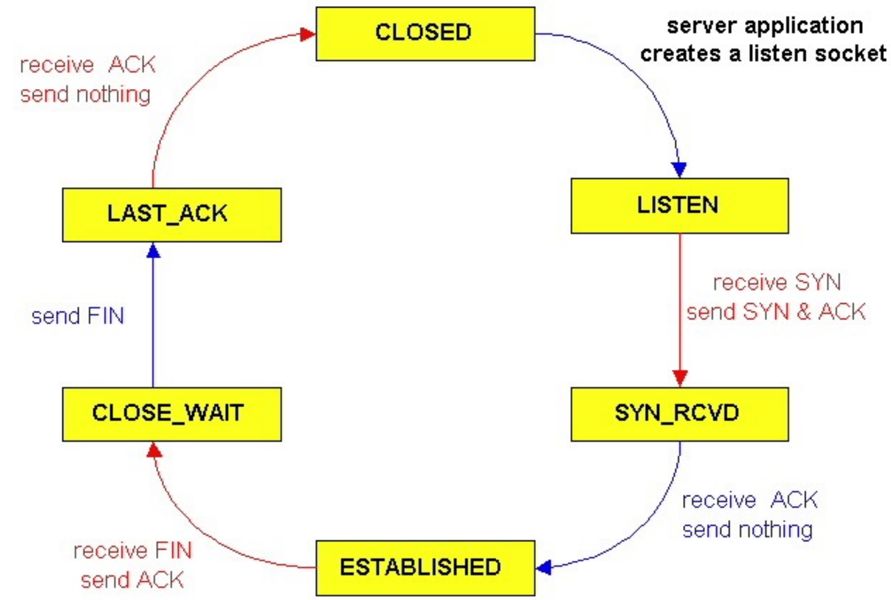
- A sends a RESET (RST) to B
  - E.g., because application process on A crashed
- That's it
  - B does not ack the RST
  - Thus, RST is not delivered reliably, and any data in flight is lost
  - But: if B sends anything more, will elicit another RST

TCP State Diagrams (Normal Case)



TCP client lifecycle

TCP server lifecycle



Summary

- TCP: connection-oriented protocol delivering reliable, in-order byte stream
  - Reliability builds on mechanisms we've seen: checksums, sequence numbers (adapted to byte stream abstraction), cumulative acknowledgments, sliding window
    - New mechanisms: dynamic timeout estimation, fast retransmit
  - Flow control prevents buffer overruns
  - Handshaking procedures used to set up and cleanly teardown connections