15-441

Computer Networking

TCP Connection Management, Error Control Mar. 29, 2004

Slides – Randy Bryant, Hui Zhang, Ion Stoica, Dave Eckhardt

(Possible) Transport Protocol Functions

Multiplexing/demultiplexing for multiple applications.

- "Port" abstraction abstracts OS notions of "process"

Connection establishment.

- Logical end-to-end connection
- Connection state to optimize performance

Error control.

- Hide unreliability of the network layer from applications
- Many types of errors: corruption, loss, duplication, reordering.

End-to-end flow control.

- Avoid flooding the receiver

[Congestion control.]

- Avoid flooding the network

Outline

Connection establishment

- Reminder

Error control, Flow control

- Stop & Wait vs. sliding window (conceptual and TCP)
- Ack flavors, windows, timeouts, sequence numbers

Connection teardown

Next Lecture – Wireless/Mobility

Monday – TCP again

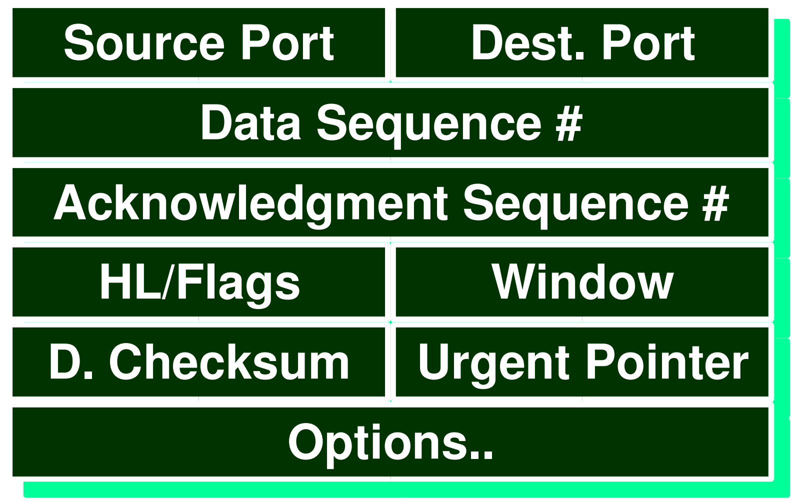
- Congestion control – you will not address in Project 3

Transmission Control Protocol (TCP)

Reliable bi-directional byte stream

Connections established & torn down

- Analogy: setting up & terminating phone call



Multiplexing/ demultiplexing

- Ports at both ends

Error control

- Users see correct, ordered byte sequences

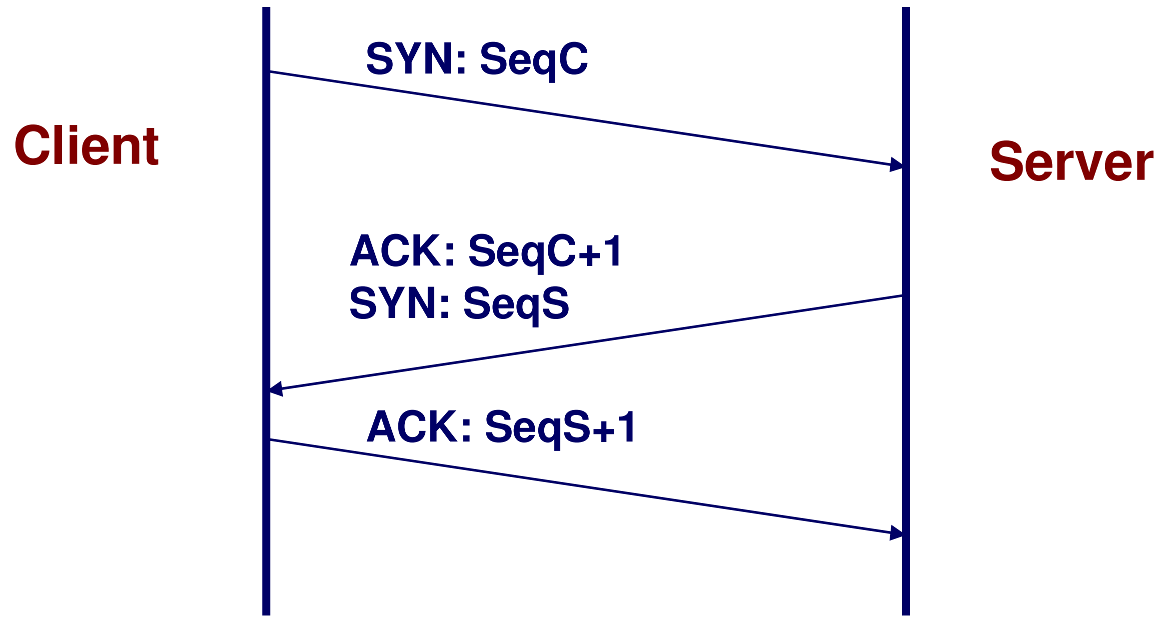
End-end flow control

- Avoid overwhelming machines at each end

Congestion avoidance

- Avoid creating traffic jams within network

Establishing Connection



Three-Way Handshake

- Each side notifies other of starting sequence number it will use for sending
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1
- "Piggy-back" second SYN with first ACK

Error Control – Threats

Network may corrupt frames

- Despite link-level checksum
- Despite switch/router memory ECC
- Example
  - Store packet headers in separate memory from packet bodies
  - Maintain association between header #343 and body #343
    - Most of the time...

Packet-sequencing issues

- Network may duplicate packets (really?)
- Network may re-order packets (why?)
- Network may lose packets (often, actually)

Error Control

Segment corruption problems

- Add end-to-end checksum to TCP segments
- Computed at sender
- Checked at receiver

Packet sequencing problems

- Include sequence number in each segment
  - Byte number of 1st data byte in segment
- Duplicate: ignore
- Reordered: re-reorder or drop
- Lost: retransmit

Error Control

Lost segments detected by sender.

- Receiver won't ACK a lost segment
- Sender can use timeout to detect lack of acknowledgment
- Setting timeout requires estimate of round-trip time

Retransmission requires sender to keep copy of data.

- Local copy is discarded when ACK is received

Error Control Algorithms

Use two basic techniques:

- Acknowledgements (ACKs)
- Timeouts

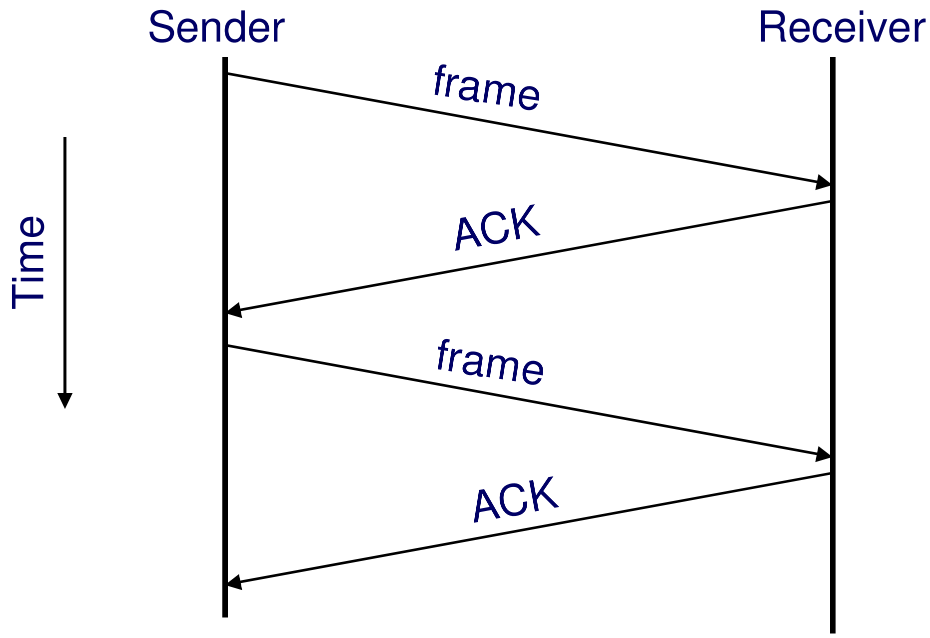
Two examples:

- Stop-and-wait
- Sliding window

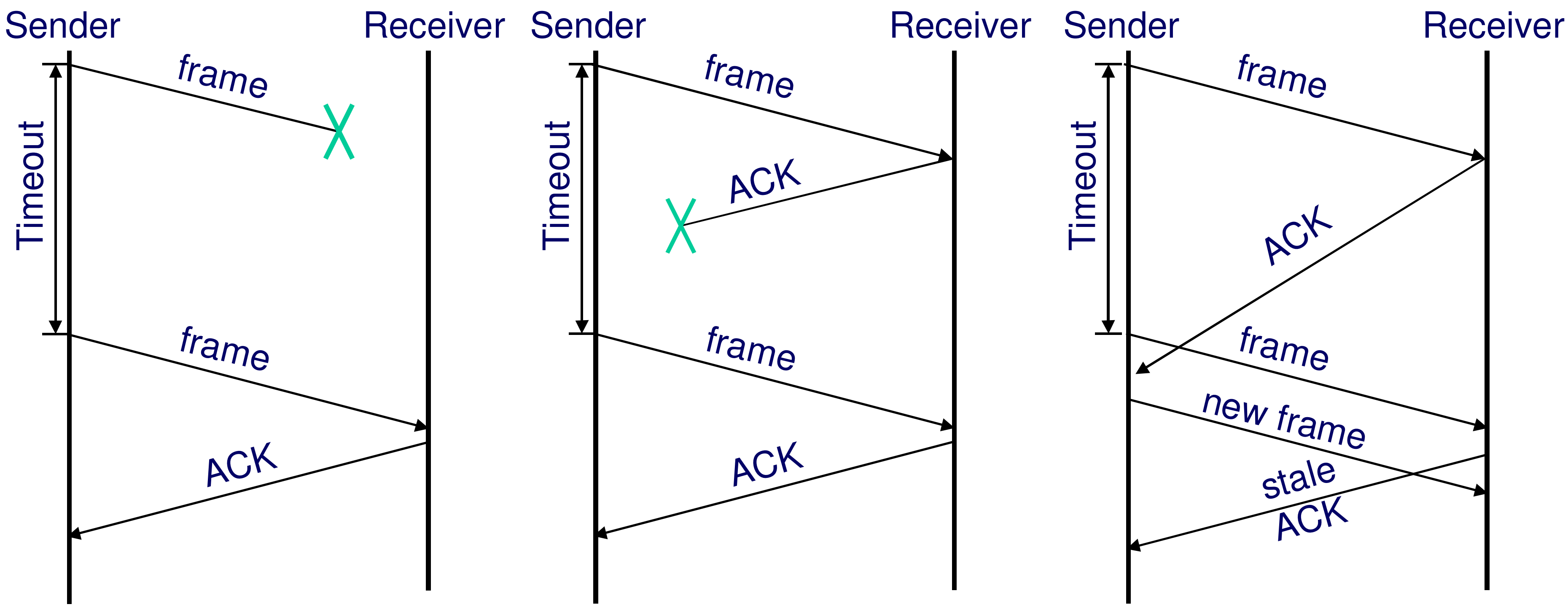
Stop-and-Wait

Receiver: send an acknowledge (ACK) back to the sender upon receiving a packet (frame)

Sender: excepting first packet, send next packet only upon receiving the ACK for the current packet



What Can Go Wrong?



Frame lost - resend it on timeout

ACK lost - resend packet

Receiver must be able to detect this is duplicate, not the next packet.

ACK delayed – resend packet

Sender must be able to detect when an ACK is for an old data packet.

Stop & Wait Sequence Numbers

Need a way to detect stale packets

- Stale data at receiver
- Stale ACK at sender

TFTP stop&wait sequence numbers are conservative

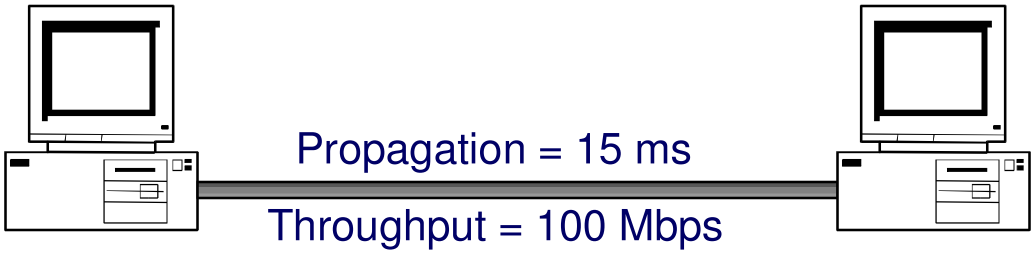
- Each packet, ACK is tagged with file position
- This is overkill
  - Bounding packet lifetime in network allows smaller sequence numbers
  - Special case: point-to-point link, 1-bit sequences numbers

Stop-and-Wait Disadvantage

May lead to inefficient link utilization

Example

- One-way propagation = 15 ms
- Throughput = 100 Mbps
- Packet size = 1000 bytes: transmit = $(8*1000)/10^8 = 0.08$ms
- Neglect queue delay: Latency = approx. 15 ms; RTT = 30 ms

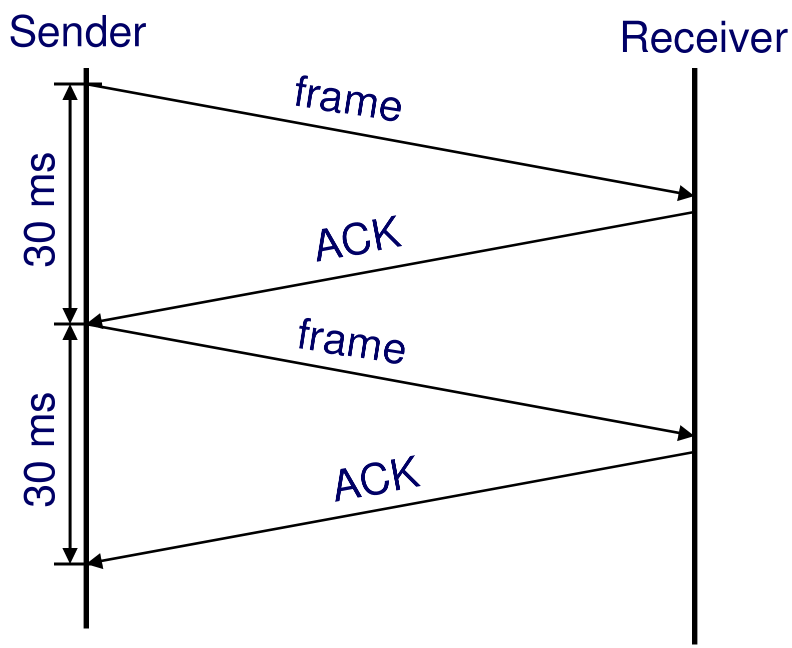


Stop-and-Wait Disadvantage (cont'd)

Send a message every 30 ms

- **Throughput = (8*1000)/0.03 = 0.2666 Mbps**

Thus, the protocol uses less than 0.3% of the link capacity!



Solution

Don't wait for the ACK of the previous packet before sending the next packet!

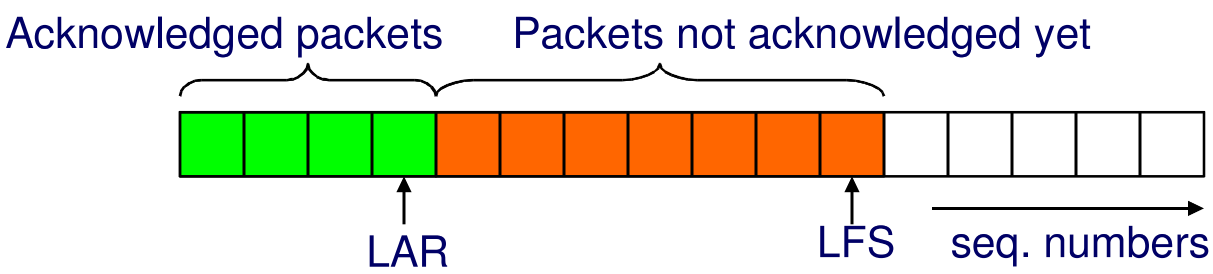
Sliding Window Protocol: Sender

Each packet has a sequence number

- Assume infinite sequence numbers for simplicity

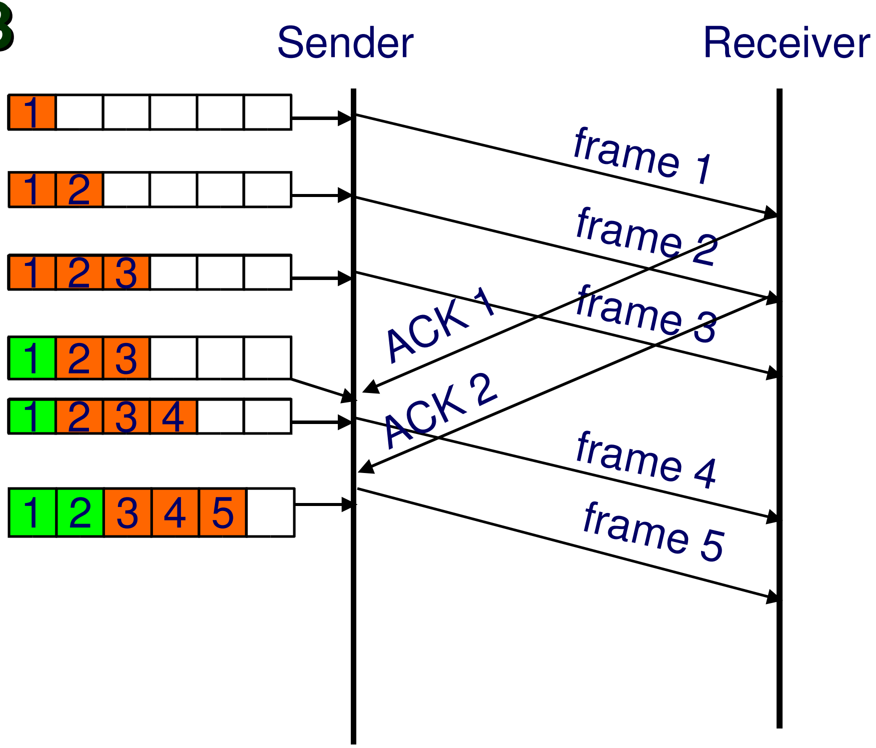
Sender maintains a window of sequence numbers

- SWS (sender window size) – maximum number of packets that can be sent without receiving an ACK
- LAR (last ACK received)
- LFS (last frame sent)



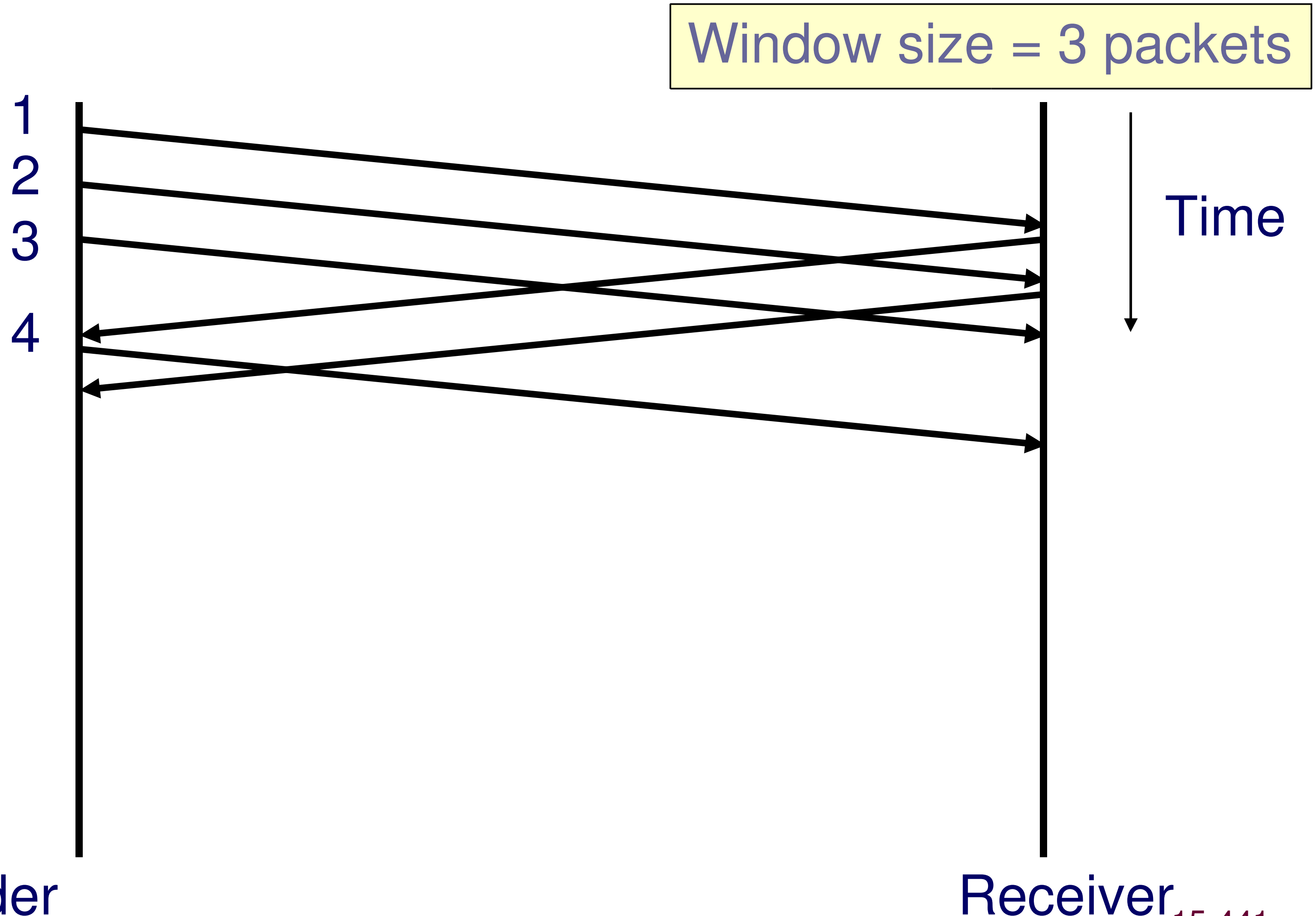
Example

Assume SWS = 3

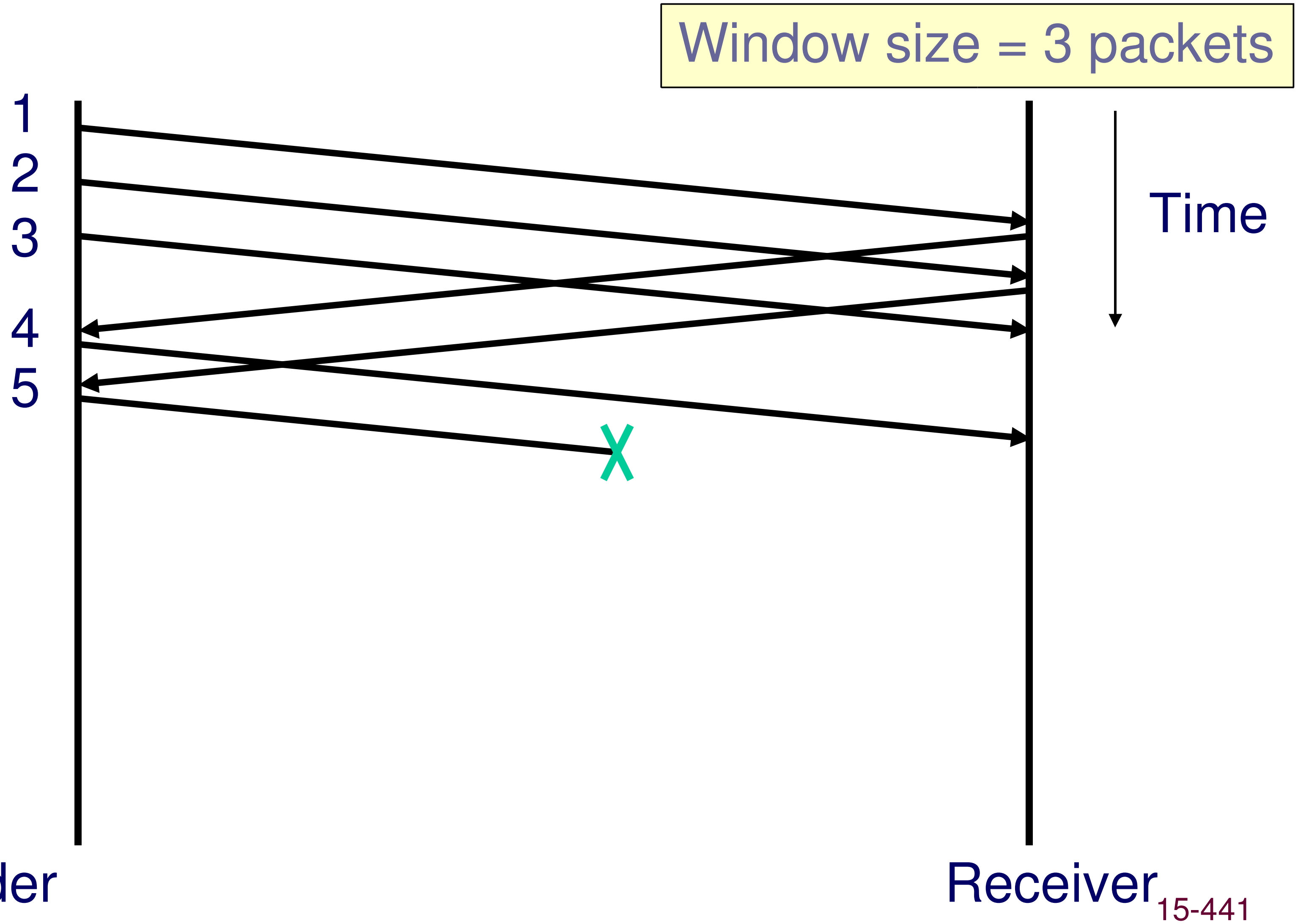


Note: usually ACK contains the sequence number of the first packet in sequence expected by receiver

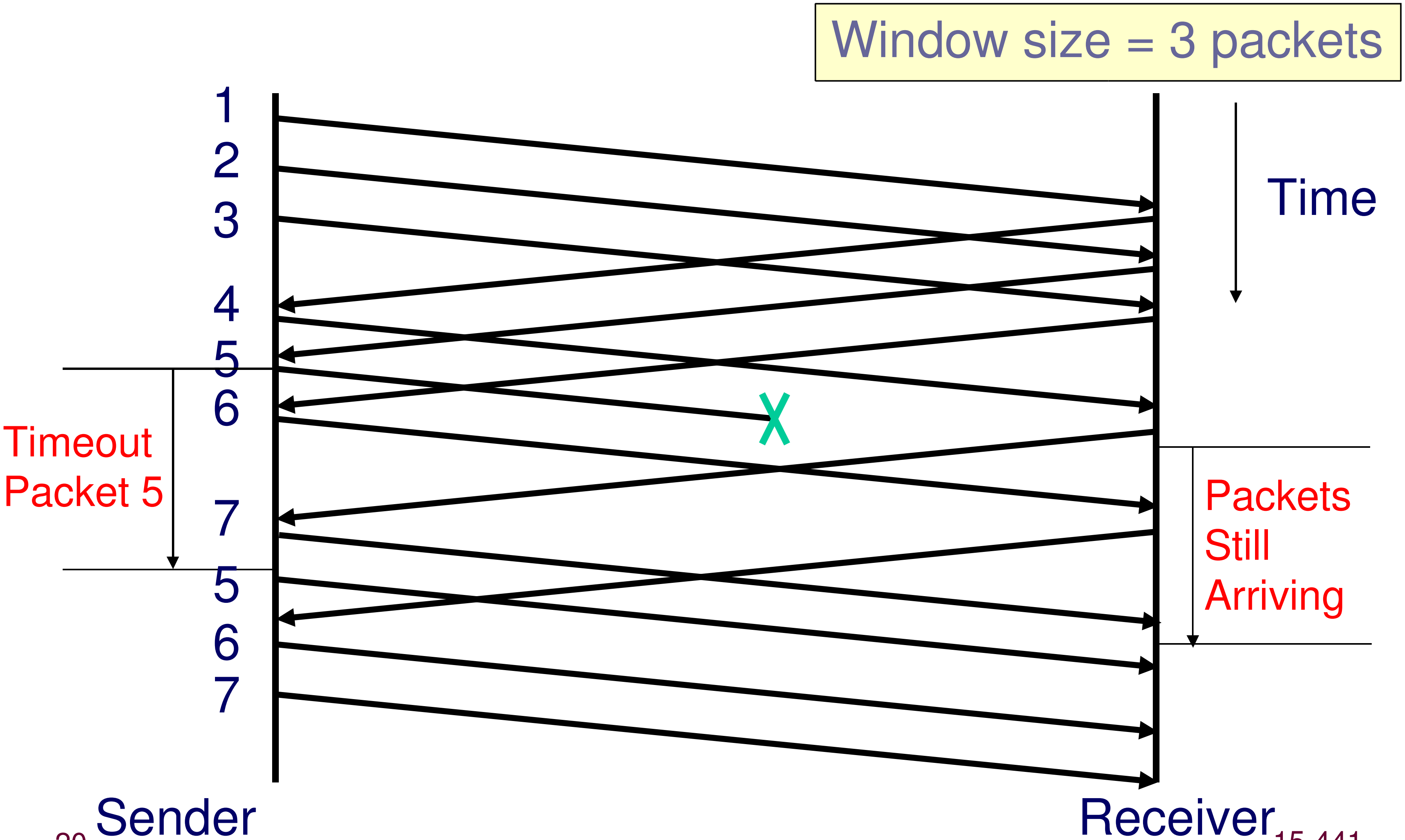
Need for Receiver Window



Need for Receiver Window



Need for Receiver Window



Sliding Window Protocol: Receiver

Receiver maintains a window of sequence numbers

- RWS (receiver window size) – maximum number of out-of-sequence packets that can received
- LFR (last frame received) – last frame received in sequence
- LAF (last acceptable frame)
- LAF – LFR <= RWS

Note that this window is just for sliding-window

- TCP “receiver window” has two purposes
- TCP also has a “congestion window”
  - Secret – does not appear in packet header

Sliding Window Protocol: Receiver

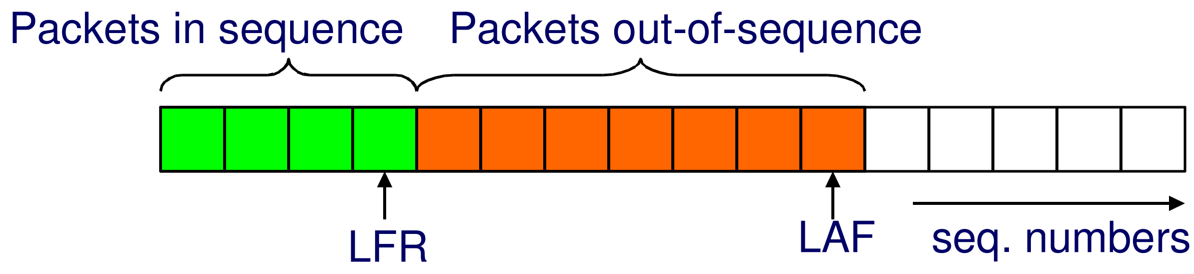
Let seqNum be the sequence number of arriving packet

If (seqNum <= LFR) or (seqNum >= LAF)

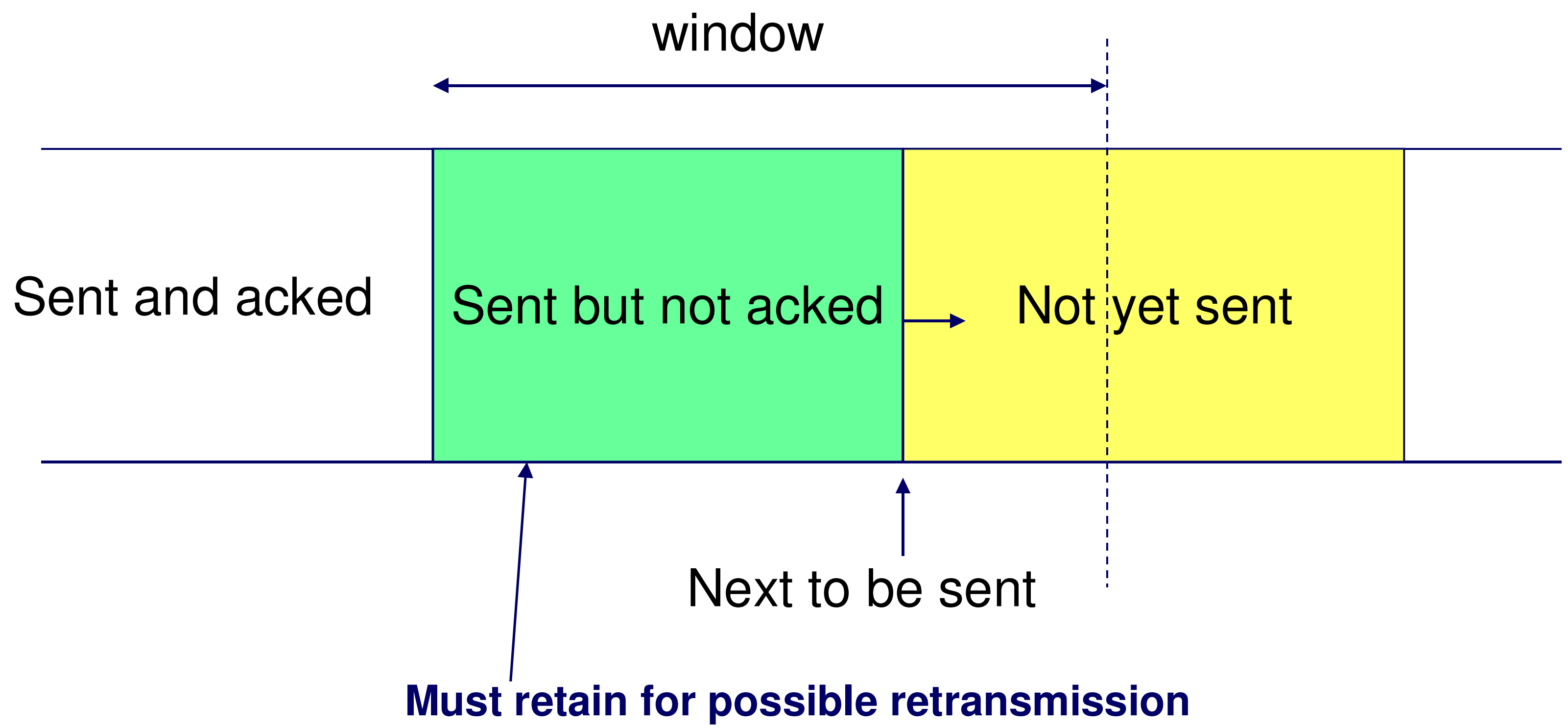
- **Discard packet**

Else

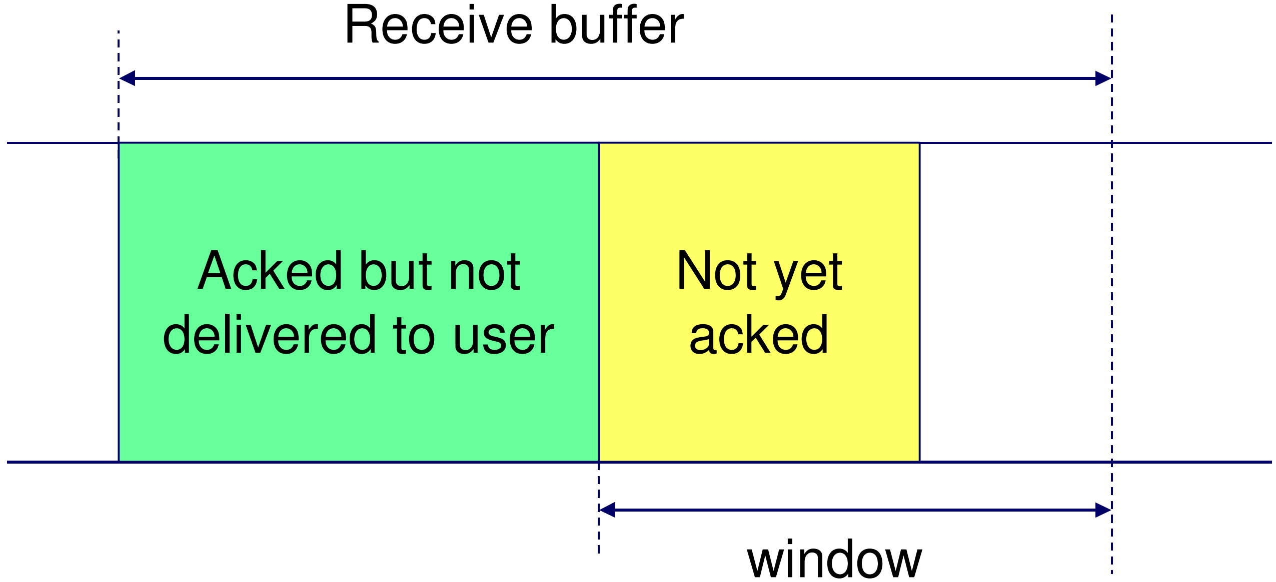
- **Accept packet**
- **ACK largest sequence number seqNumToAck, such that all packets with sequence numbers <= seqNumToAck were received**



Window Flow Control: Send Side

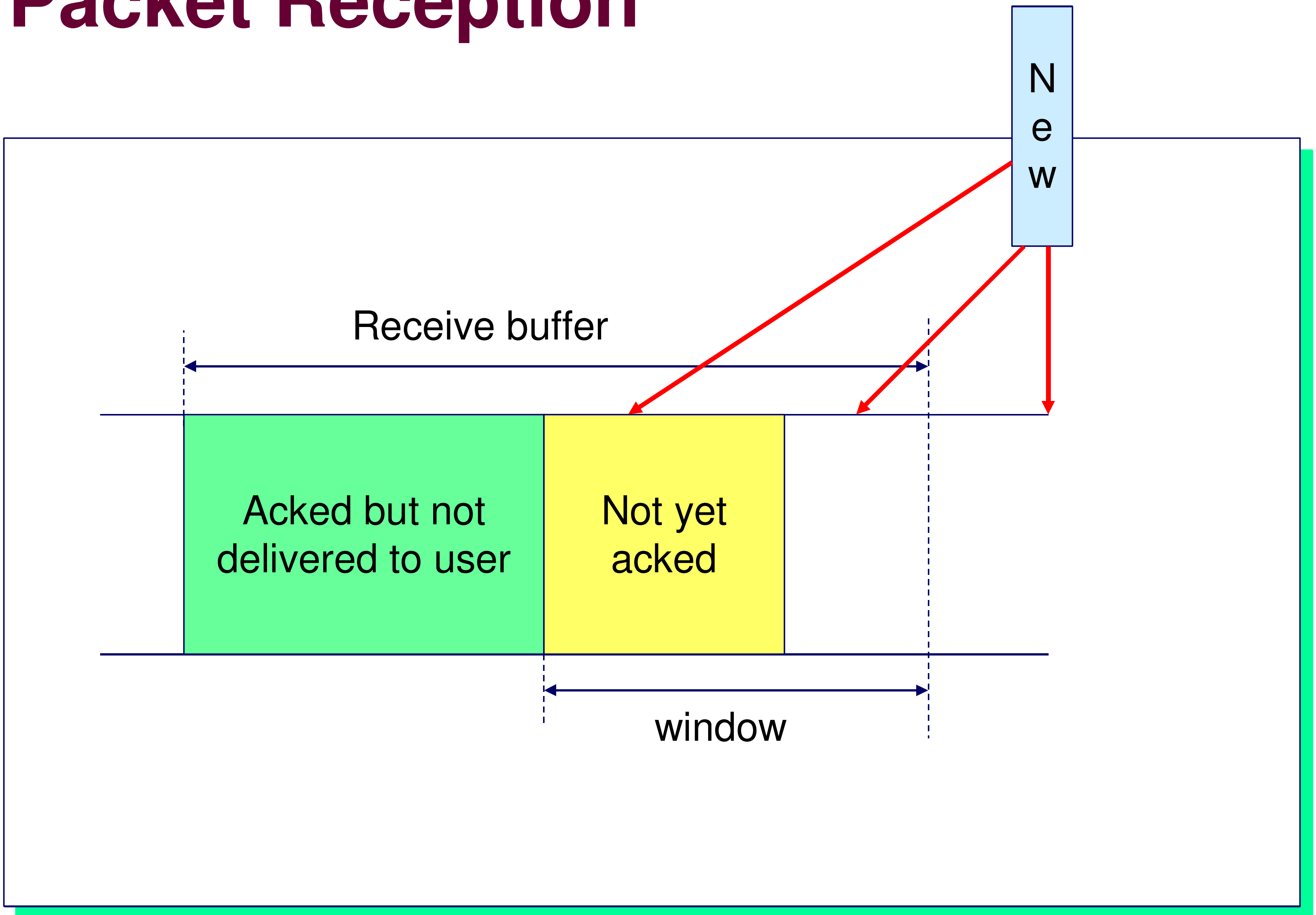


Window Flow Control: Receive Side



Packet Reception

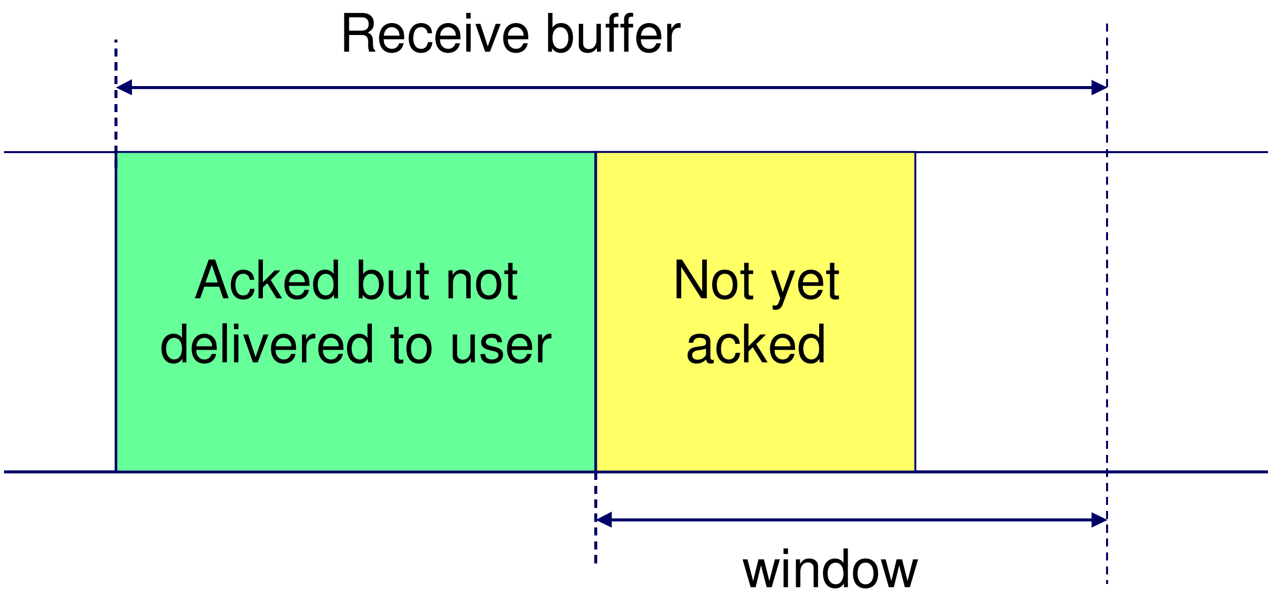
What should receiver do?



Maximum Window Size

Mechanism for receiver to exert flow control

- **Prevent sender from overwhelming receiver's buffering & processing capacity**
- **Max. transmission rate = window size / round trip time**



Choices of Ack

Cumulative ack

- I have received 17..23
- I have [still] received 17..23

Selective ack

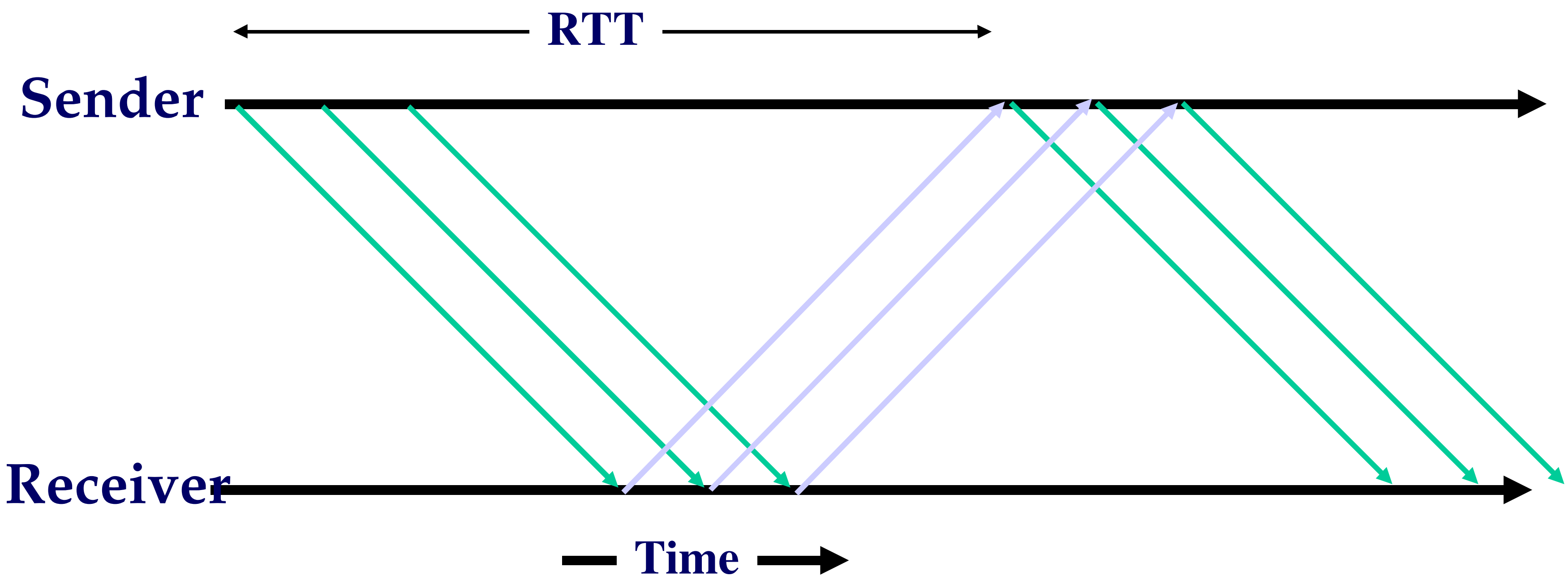
- I received 17-23, 25-27

Negative ack

- I think I'm missing 24...

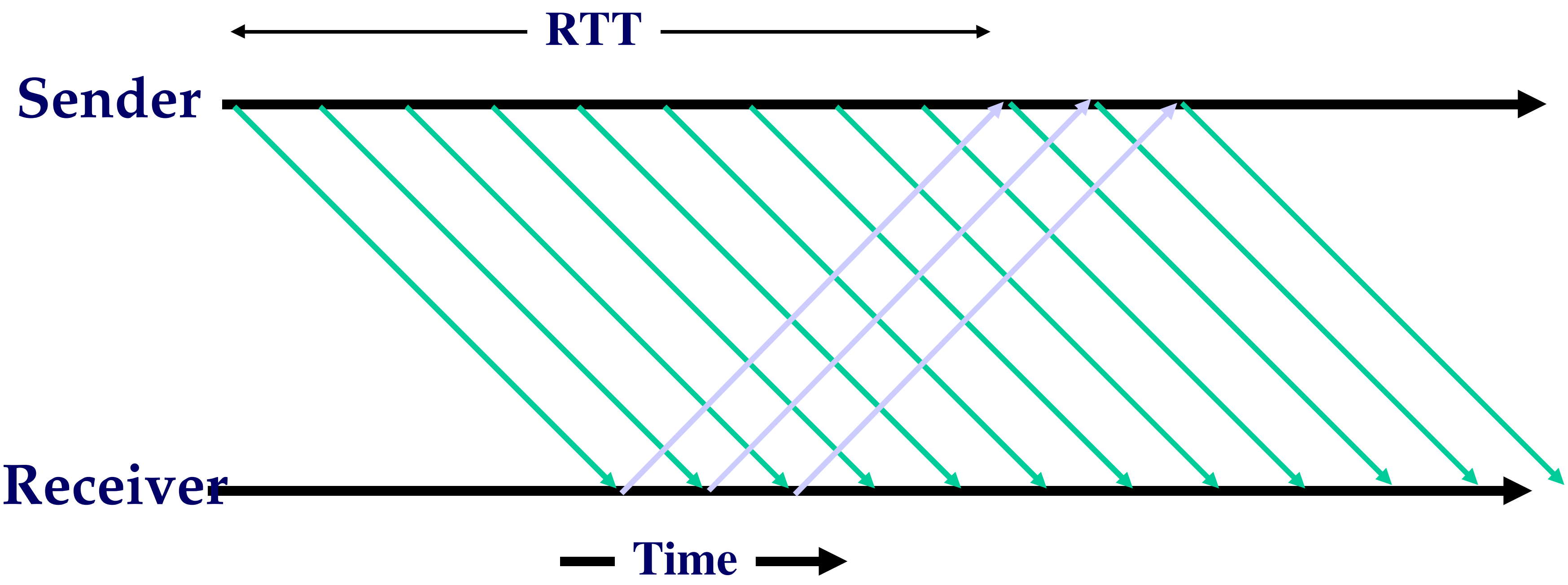
Tradeoffs?

Window Size Too Small



$$\text{Max Throughput} = \frac{\text{Window Size}}{\text{Roundtrip Time}}$$

Adequate Window Size



$$\text{Max Throughput} = \frac{\text{Window Size}}{\text{Roundtrip Time}}$$

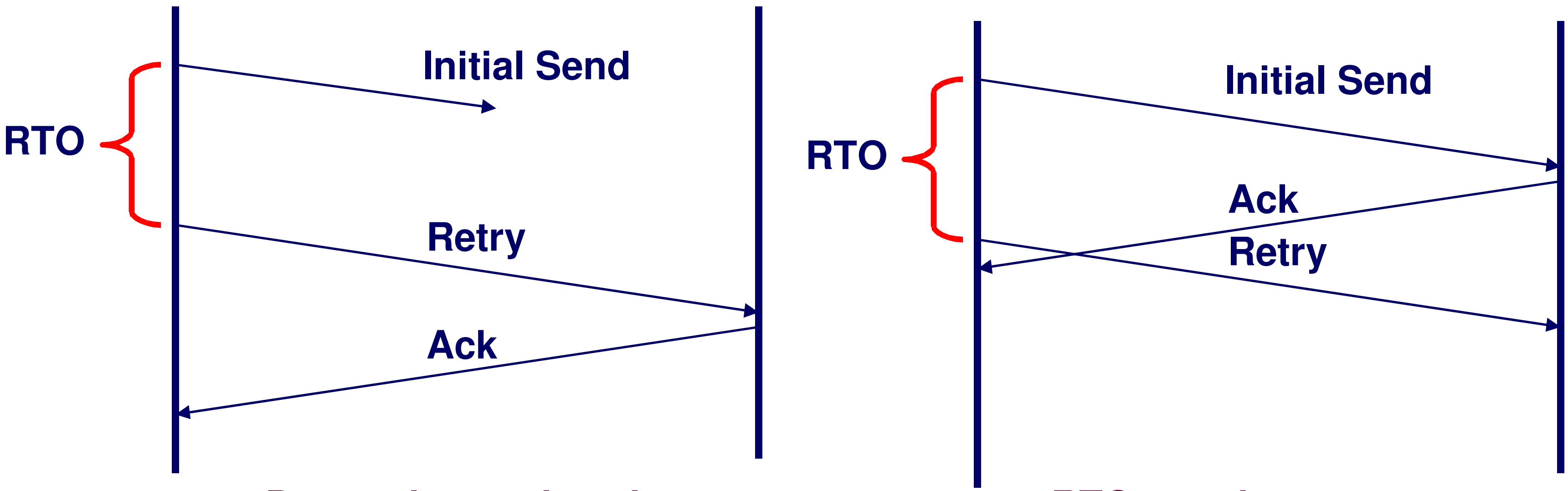
Timeout Value Selection

Long timeout?

Short timeout?

Solution?

Setting Retransmission Timeout (RTO)



Detect dropped packet   RTO too short

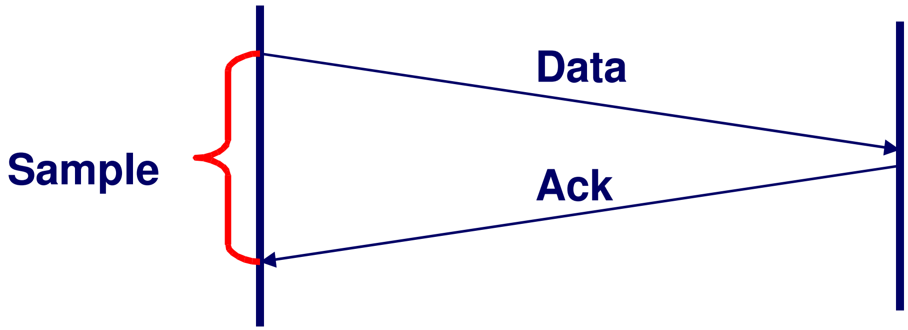
- Time between sending & resending segment

Challenge

- Too long: Add latency to communication when packets dropped
- Too short: Send too many duplicate packets
- General principle: Must be > 1 Round Trip Time (RTT)

Round-trip Time Estimation

Every Data/Ack pair gives new RTT estimate

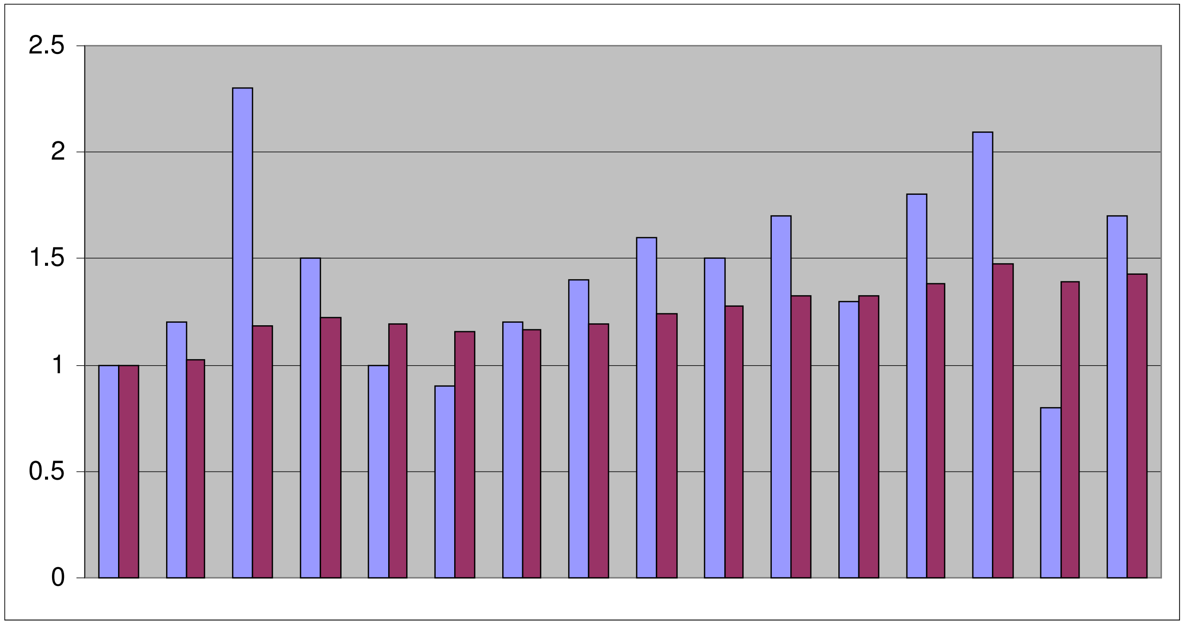


Can Get Lots of Short-Term Fluctuations

Original TCP Round-trip Estimator

Round trip times exponentially averaged:

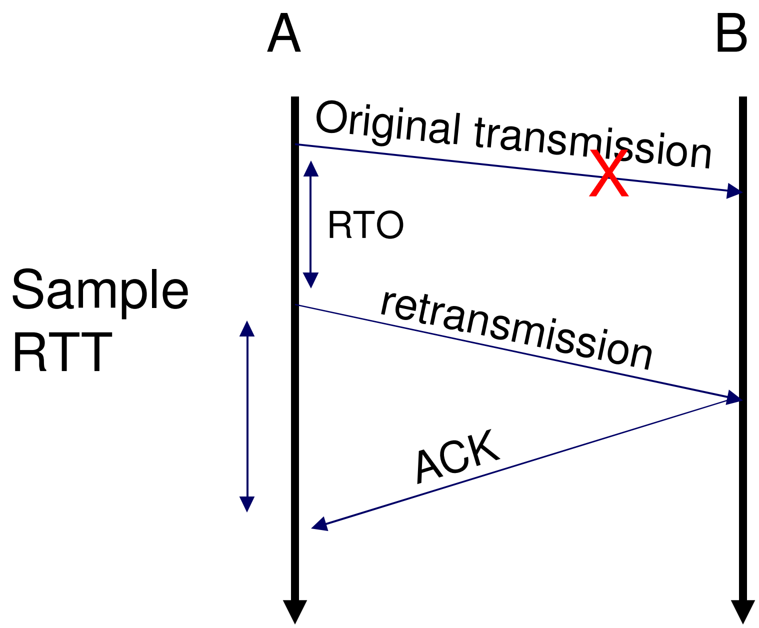
- New RTT = α (old RTT) + (1 - α) (new sample)
- Recommended value for α: 0.8 - 0.9
  - 0.875 for most TCP's

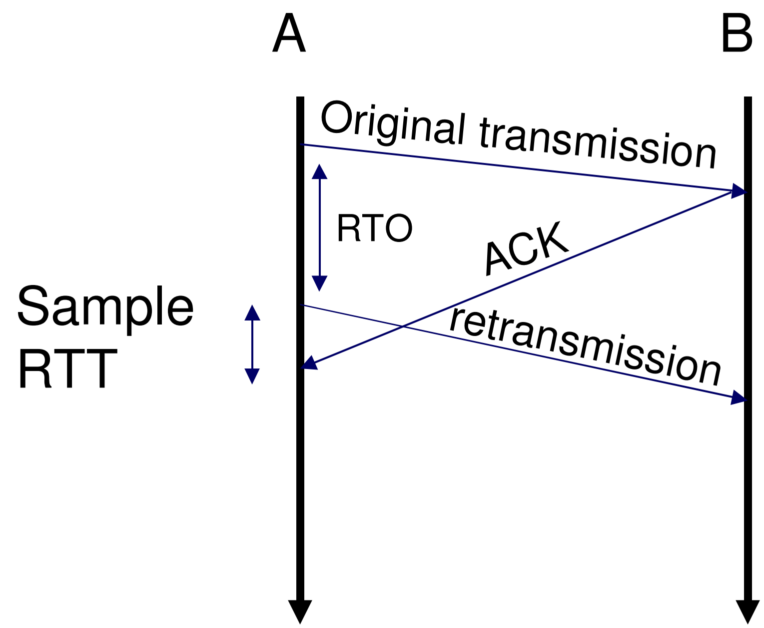


Retransmit timer set to β RTT, where β = 2

- Want to be somewhat conservative about retransmitting

RTT Sample Ambiguity





Karn/Partridge Algorithm

- **Ignore sample for segment that has been retransmitted**
- **Use exponential backoff for retransmissions**
  - **Each time retransmit same segment, double the RTO**
  - **Based on premise that packet loss is caused by major congestion**

Sequence Number Space

Each byte in byte stream is numbered.

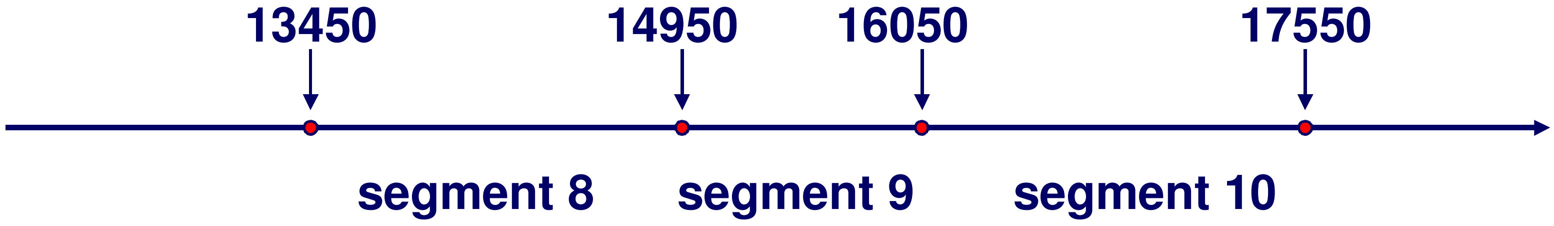
- 32 bit value
- Wraps around
- Initial values selected at start up time

TCP breaks byte stream into packets ("segments")

- Packet size is limited to the Maximum Segment Size

Each segment has a sequence number.

- Indicates where it fits in the byte stream



Finite Length Sequence Number

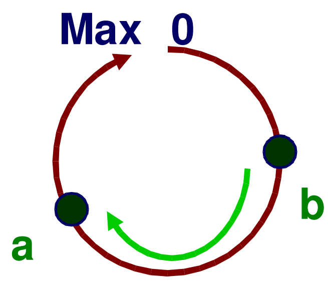
Sequence number can wrap around

- **What is the problem?**
- **What is the solution?**
  - **Hint: not “crash the kernel”**
  - **Not even “crash the connection” or “connection full”**

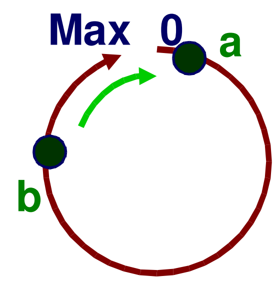
Sequence Numbers

32 Bits, unsigned

Circular Comparison, “b following a”



b < a



b < a, but “b >> a”

Why So Big?

- For sliding window, must have
  - |Sequence Space| > |Sending Window| + |Receiving Window|
    - No problem
- Also must guard against stray packets
  - With IP, packets have maximum lifetime of 120s
  - Sequence number would wrap around in this time at 286MB/s

Error Control Summary

Basic mechanisms

- CRC, checksum
- Timeout
- Acknowledgement
- Sequence numbers
- Window

Many variations and details

TCP Flow Control

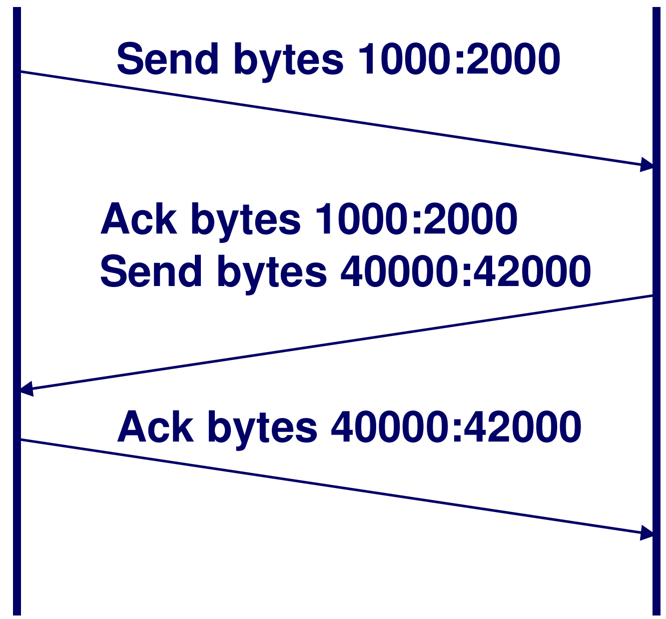
Recall sliding-window as used for *error* control

- For window size *n*, can send up to *n* bytes without receiving an acknowledgement
- When the data are acknowledged then the window slides forward

Achieve *flow* control via dynamically-sized window

- Sender naturally tracks outstanding packets versus max
  - Sending one packet decreases budget by one
- Receiver updates "open window" in every response
  - Packet B $\Rightarrow$ A contains Ack_A and $Window_A$
  - Sender can send bytes up through (Ack_A + $Window_A$)
  - Receiver can increase or decrease window at any time
- Original TCP always sent entire window
  - Congestion control now limits this

Bidirectional Communication



Each Side of Connection can Send *and* Receive

What this Means

- Maintain different sequence numbers for each direction
- Single segment can contain new data for one direction, plus acknowledgement for other
  - But some contain only data & others only acknowledgement

Ongoing Communication

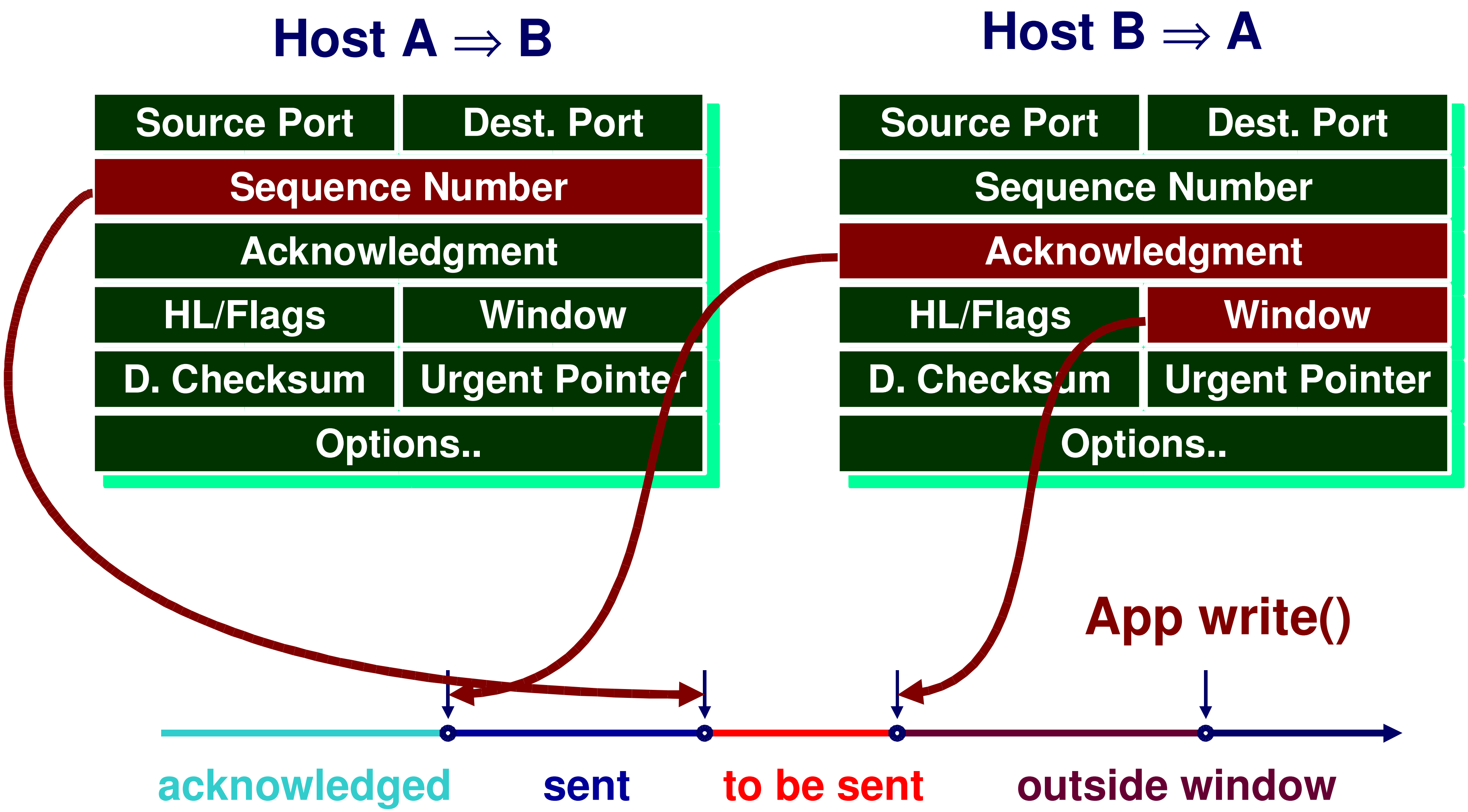
Bidirectional Communication

- Each side acts as sender & receiver
- Every message contains acknowledgement of received sequence
  - Even if no new data have been received
- Every message advertises window size
  - Size of its receiving window
- Every message contains sent sequence number
  - Even if no new data being sent

When Does Sender Actually Send Message?

- When a maximal-sized segment worth of bytes is available
- When application tells it
  - Set PUSH flag for last segment sent
- When timer expires

Window Flow Control: Send Side

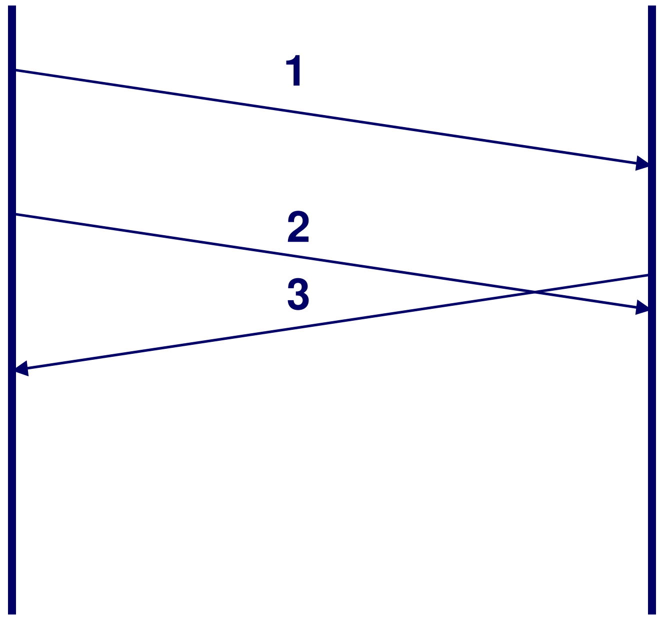


TCP Transmission

```
09:23:33.132509 IP 128.2.222.198.3123 > 192.216.219.96.80: P
  4019802005:4019802801(796) ack 3428951570 win 65535 (DF)

09:23:33.149875 IP 128.2.222.198.3123 > 192.216.219.96.80: .
  4019802801:4019804061(1260) ack 3428951570 win 65535 (DF)

09:23:33.212291 IP 192.216.219.96.80 > 128.2.222.198.3123: . ack
  4019802801 win 7164 (DF)
```



Client sends 796 bytes

Client sends 1260 more bytes

Server acknowledges 796 bytes

Tearing Down Connection

Either Side Can Initiate Tear Down

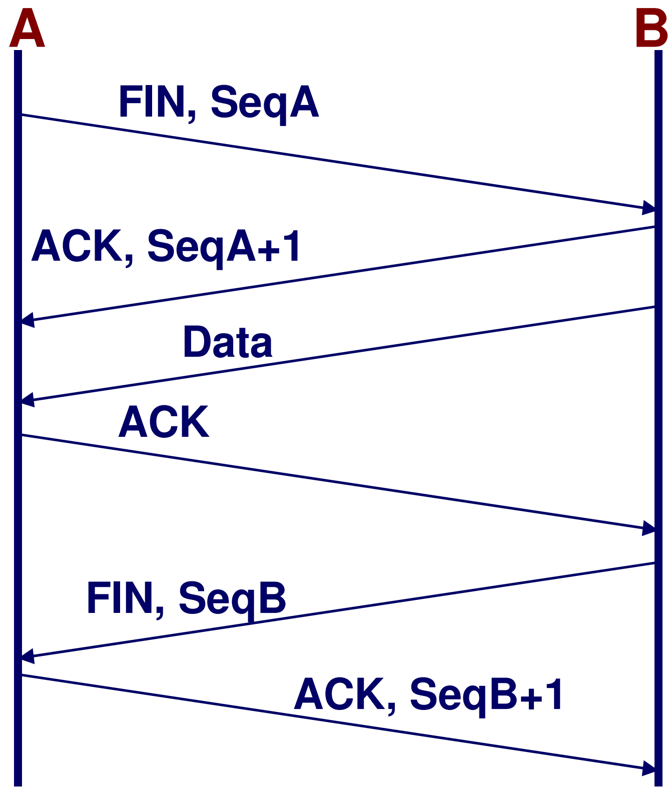
- Send FIN signal
  - "I'm not going to send any more data"

Other Side Can Continue Sending Data

- "Half-open connection"
- I must continue to acknowledge

Acknowledging FIN

- Acknowledge last sequence number + 1



TCP Connection Teardown Example

```
09:54:17.585396 IP 128.2.222.198.4474 > 128.2.210.194.6616: F
  1489294581:1489294581(0) ack 1909787689 win 65434 (DF)

09:54:17.585732 IP 128.2.210.194.6616 > 128.2.222.198.4474: F
  1909787689:1909787689(0) ack 1489294582 win 5840 (DF)

09:54:17.585764 IP 128.2.222.198.4474 > 128.2.210.194.6616: . ack
  1909787690 win 65434 (DF)
```

Session

- Echo client on 128.2.222.198, server on 128.2.210.194

Client FIN

- SeqC: 1489294581

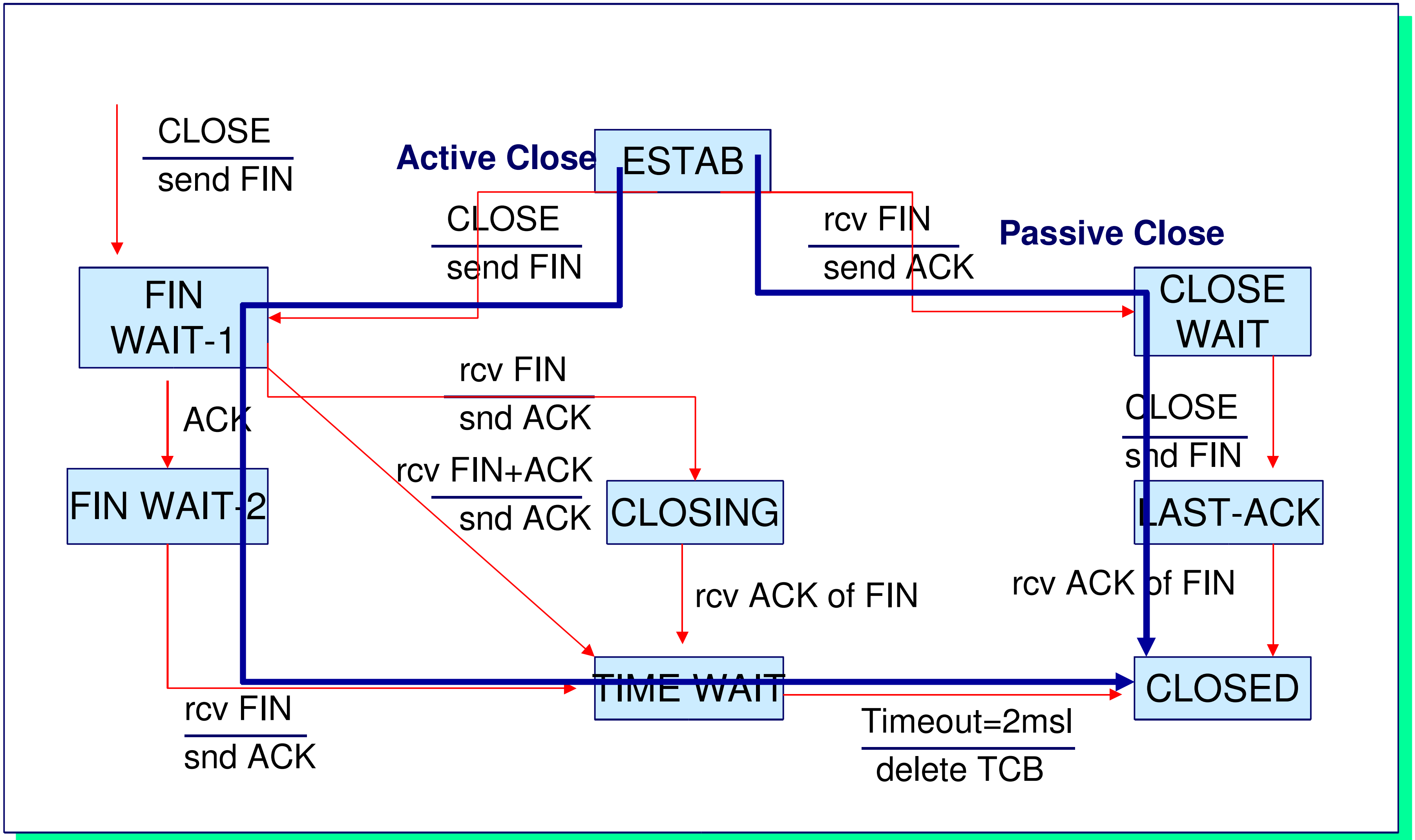
Server ACK + FIN

- Ack: 1489294582 (= SeqC+1)
- SeqS: 1909787689

Client ACK

- Ack: 1909787690 (= SeqS+1)

State Diagram: Connection Tear-down



Key TCP Design Decisions

Connection Oriented

- Explicit setup & teardown of connections

Byte-stream oriented

- vs. message-oriented
- Sometimes awkward for application to infer message boundaries

Sliding Window with Cumulative Acknowledgement

- Single acknowledgement covers range of bytes
- Single missing message may trigger series of retransmissions

No Negative Acknowledgements

- Any problem with transmission must be detected by timeout
- OK for IP to silently drop packets