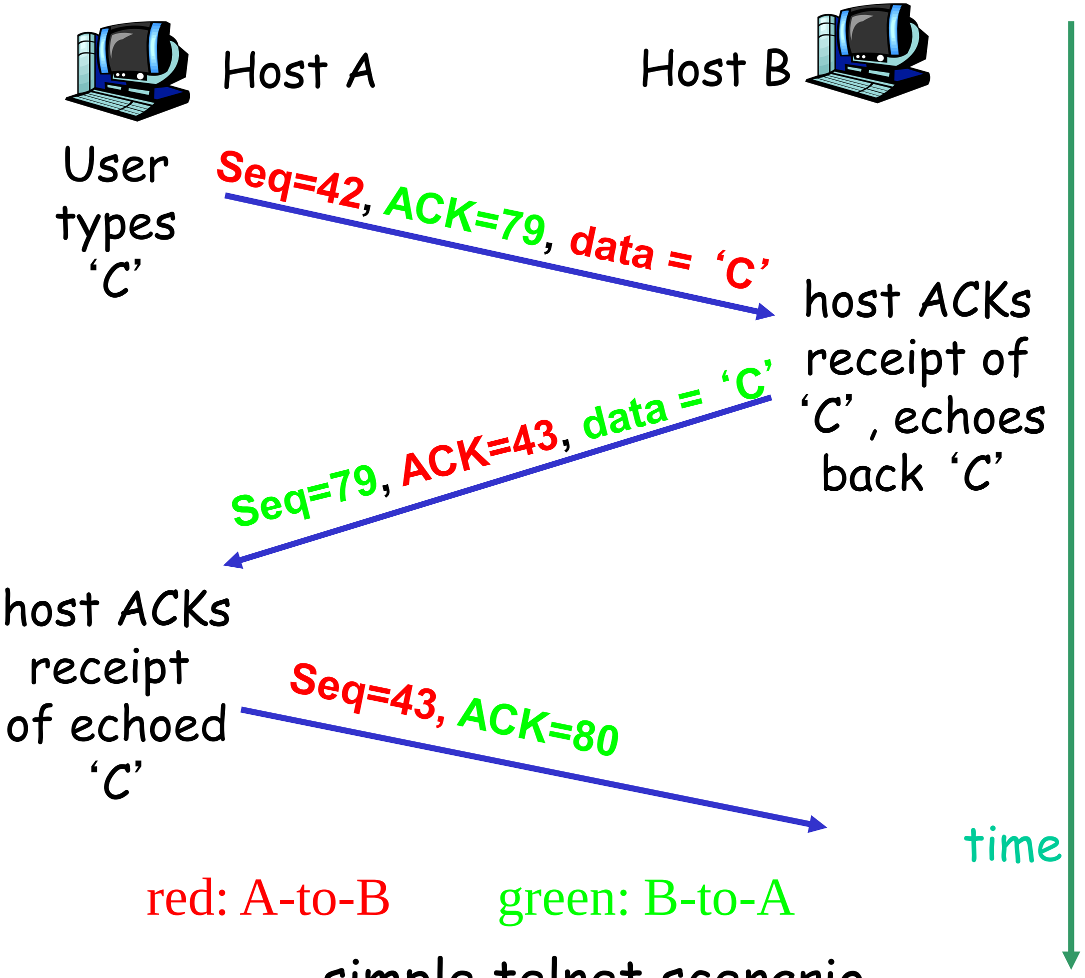
TCP Seq. #'s and ACKs

Seq. #'s:

byte stream "number" of first byte in segment's data

ACKs:

seq # of next byte expected from other side

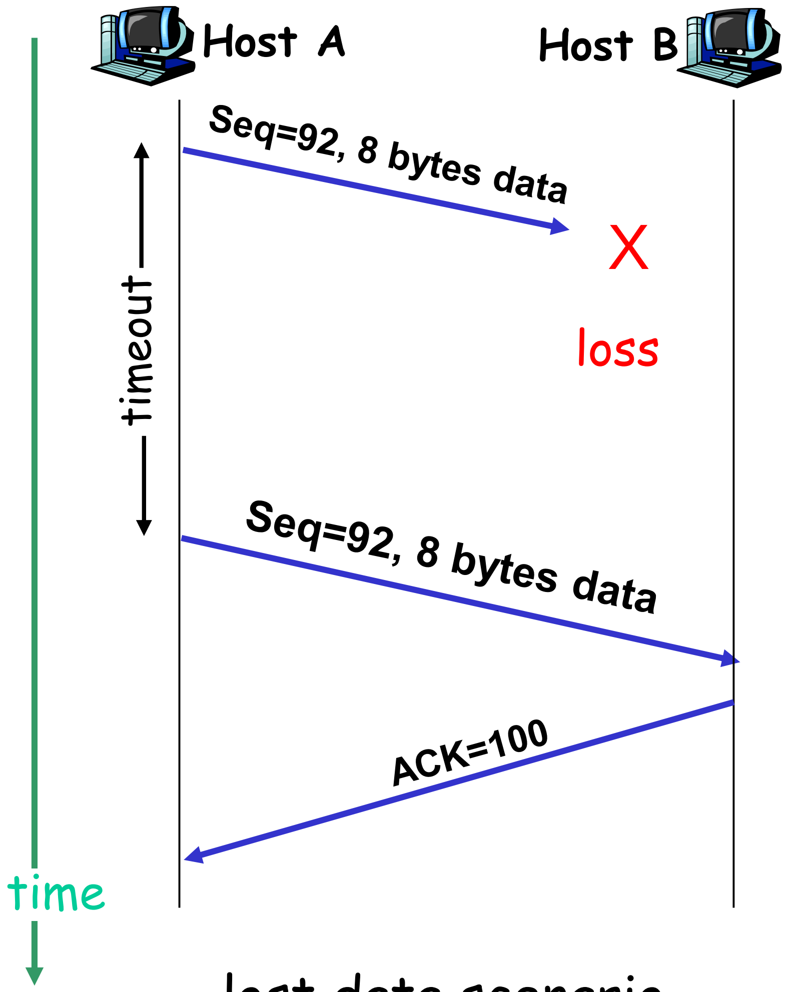


simple telnet scenario

TCP: Error Scenarios

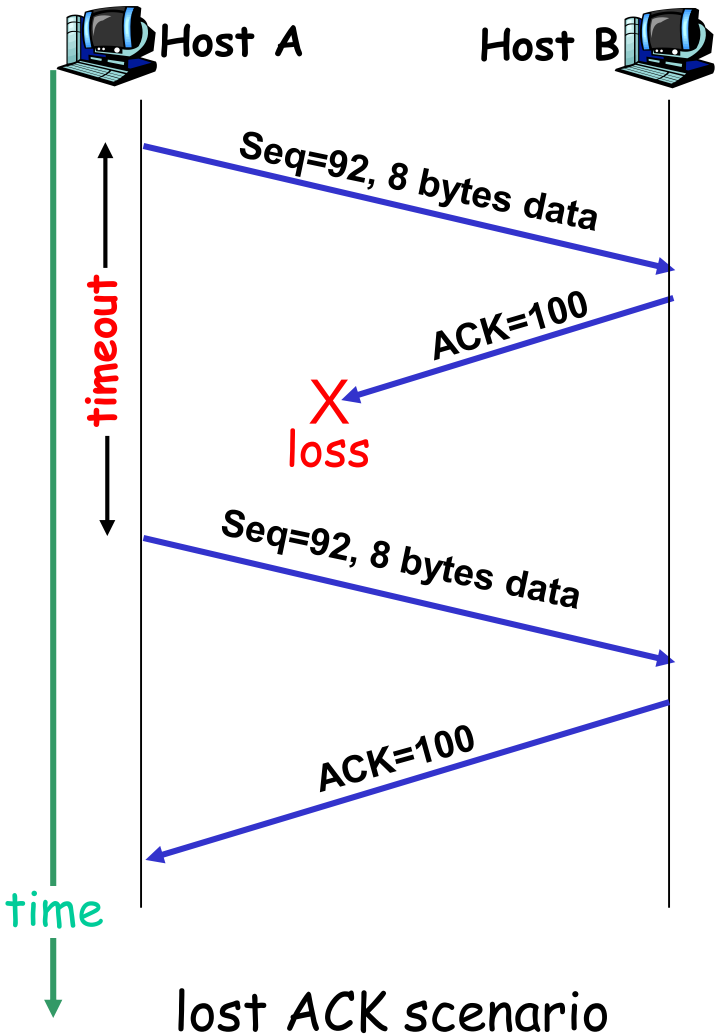
Questions for you:

- How to detect lost packets?
- How to "recover" lost packets?
- Potential consequence of retransmission?
- How to detect duplicate packets?
- "State" maintained at sender & receiver?

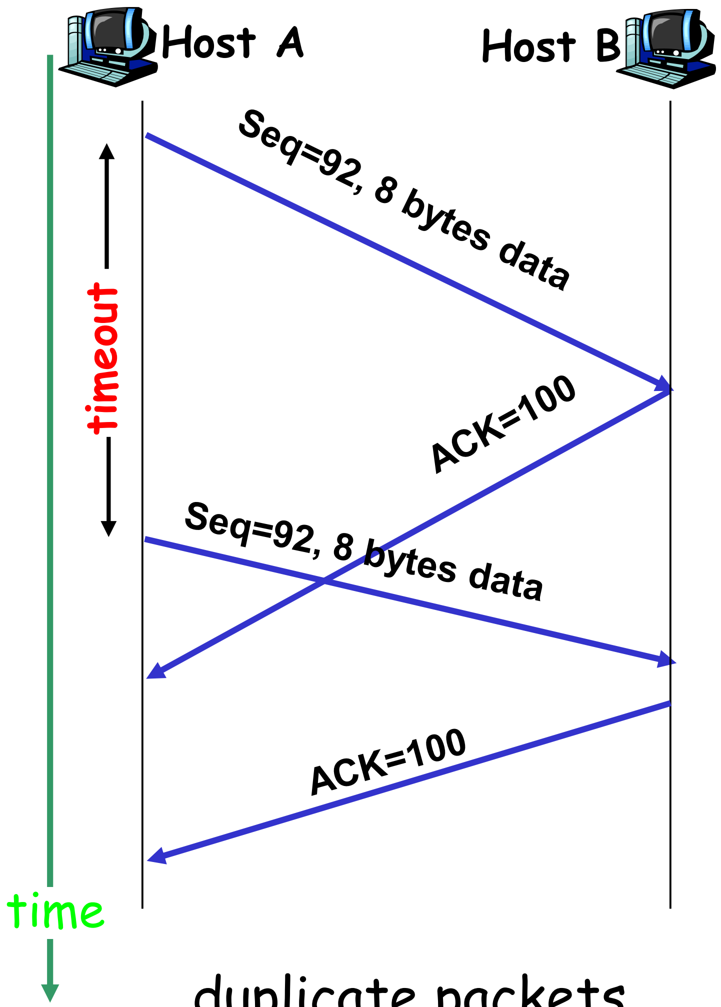


lost data scenario

TCP: Error Scenarios (cont'd)



lost ACK scenario



duplicate packets

A Simple Reliable Data Transfer Protocol

"Stop & Wait" Protocol (aka "Alternate Bit" Protocol)

Sender algorithm:

- **Send Phase**: send data segment (n bytes) w/ seq=x, buffer data segment, set timer
- **Wait Phase**: wait for ack from receiver w/ ack= x+n
  - if received ack w/ ack=x+n, set x:=x+n, and go to sending phase with next data segment
  - if time out, resend data segment w/ seq=x.
  - if received ack w/ ack != x+n, ignore (or resend data segment w/ seq=x)

Receiver algorithm:

Wait-for-Data:

wait for data packet with the (expected) next-seq = x

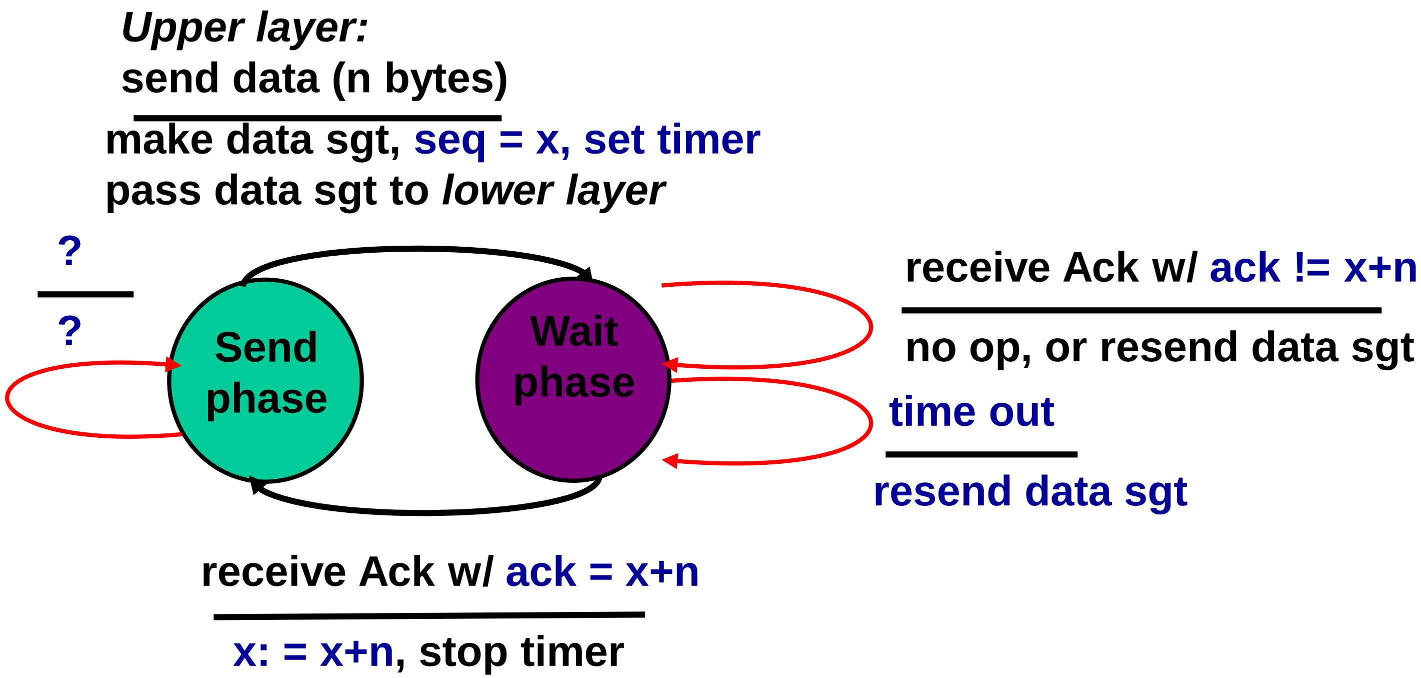
- if received Data packet w/ seq. =x and of size n bytes: send ACK pkt w/ ack = x+n; set next-seq:= x+n; go back to "Wait-for-Data";
- If received Data packet w/ seq != x, (re-)send ACK pkt w/ ack= next-seq; go back to "Wait-for-Data";

Q: what is the "state" information maintained at the sender & receiver, resp.?

SRDTP: Finite State Machine

Sender FSM

Receiver FSM?



info ("state") maintained at sender:
phase it is in (*send*, or *wait*), ack expected, data sgt sent (seq #), timer

TCP Connection Set Up

TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info
- *client:* end host that initiates connection
- *server:* end host contacted by client

Three way handshake:

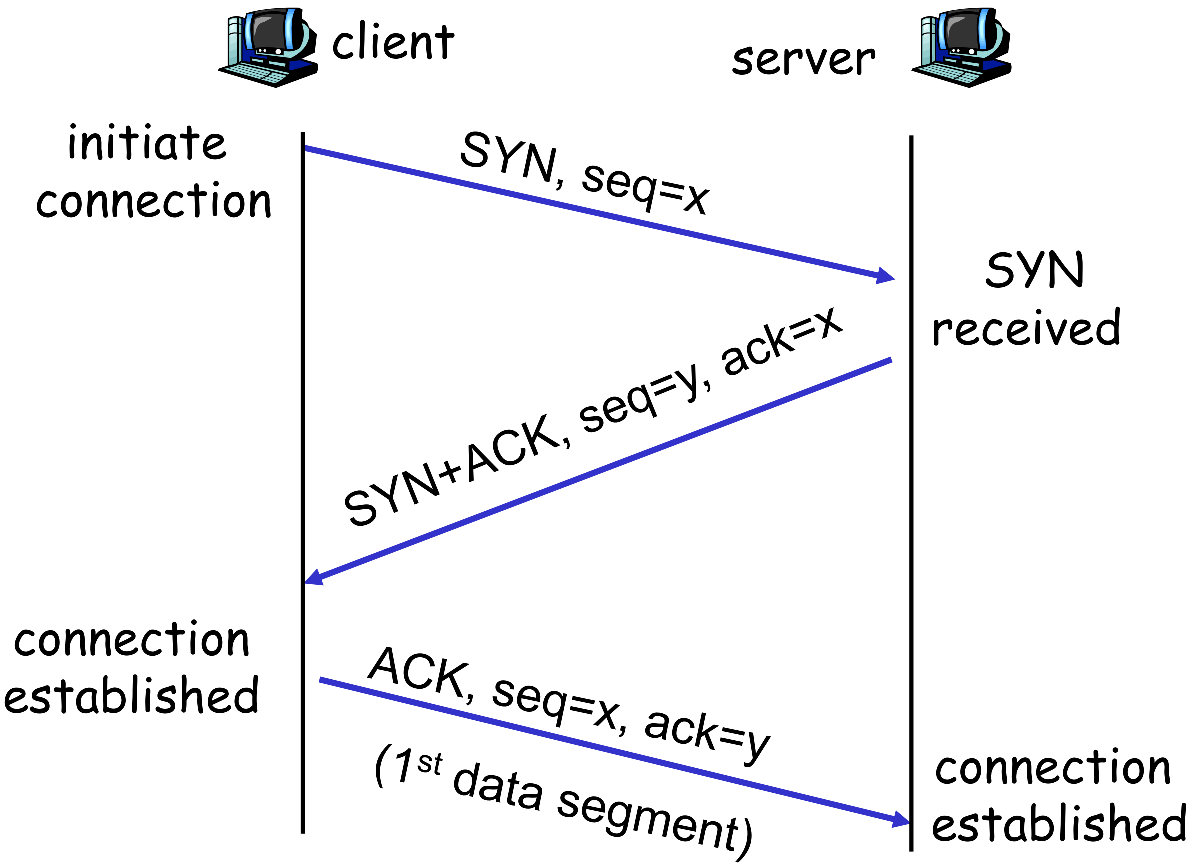
Step 1: client sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server receives SYN, replies with SYN+ACK control segment
- ACKs received SYN
- specifies server → receiver initial seq. #

Step 3: client receives SYN+ACK,

replies with ACK segment (which may contain 1st data segment)

TCP 3-Way Hand-Shake



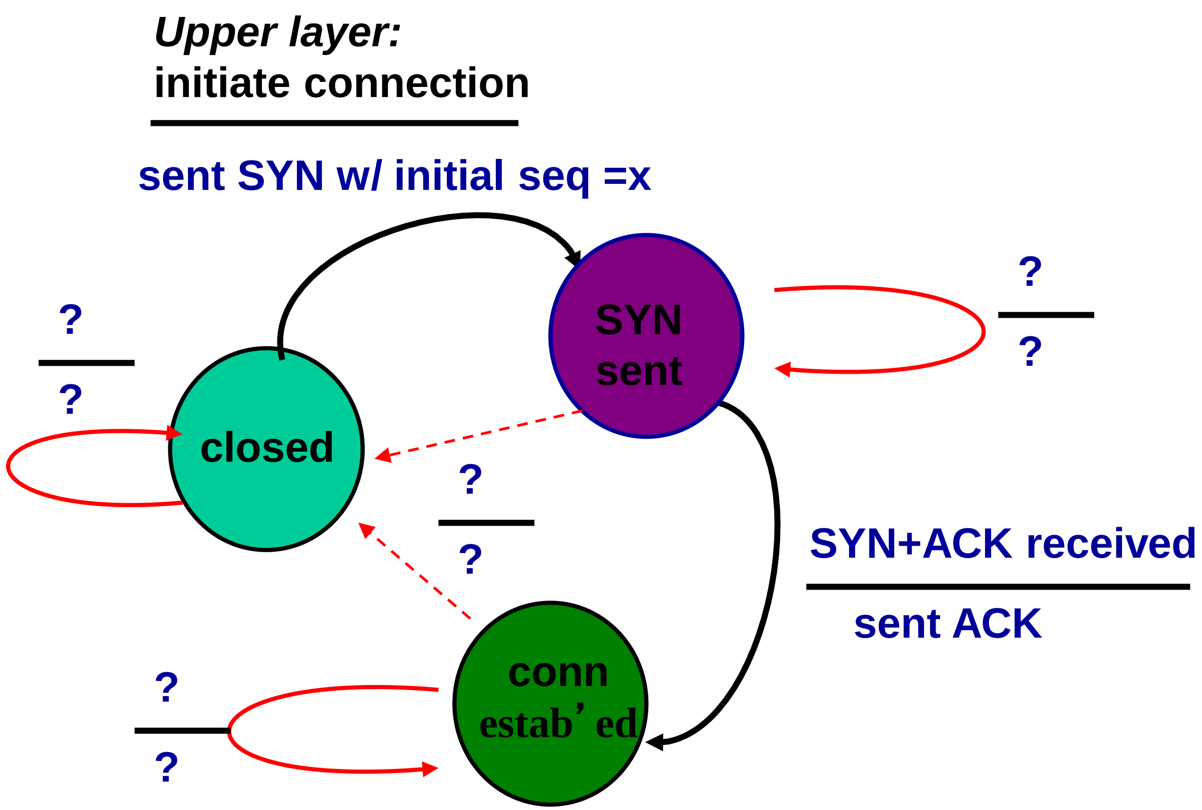
Question:

a. What kind of "state" client and server need to maintain?

b. What initial sequence # should client (and server) use?

3-Way Handshake: Finite State Machine

Client FSM? Server FSM?



info ("state") maintained at client?

Connection Setup Error Scenarios

- Lost (control) packets
  - What happen if SYN lost? client vs. server actions
  - What happen if SYN+ACK lost? client vs. server actions
  - What happen if ACK lost? client vs. server actions
- Duplicate (control) packets
  - What does server do if duplicate SYN received?
  - What does client do if duplicate SYN+ACK received?
  - What does server do if duplicate ACK received?

Connection Setup Error Scenarios (cont'd)

- Importance of (unique) initial seq. no.?
  - When receiving SYN, how does server know it's a new connection request?
  - When receiving SYN+ACK, how does client know it's a legitimate, i.e., a response to its SYN request?
- Dealing with old duplicate (aka "ghost") packets from old connections (or from malicious users)
  - If not careful: "TCP Hijacking"
- How to choose unique initial seq. no.?
  - randomly choose a number (and add to last syn# used)
- Other security concern:
  - "SYN Flood" -- denial-of-service attack

TCP: Closing Connection

Remember TCP duplex connection!

Client wants to close connection:

Step 1: client end system sends TCP FIN control segment to server

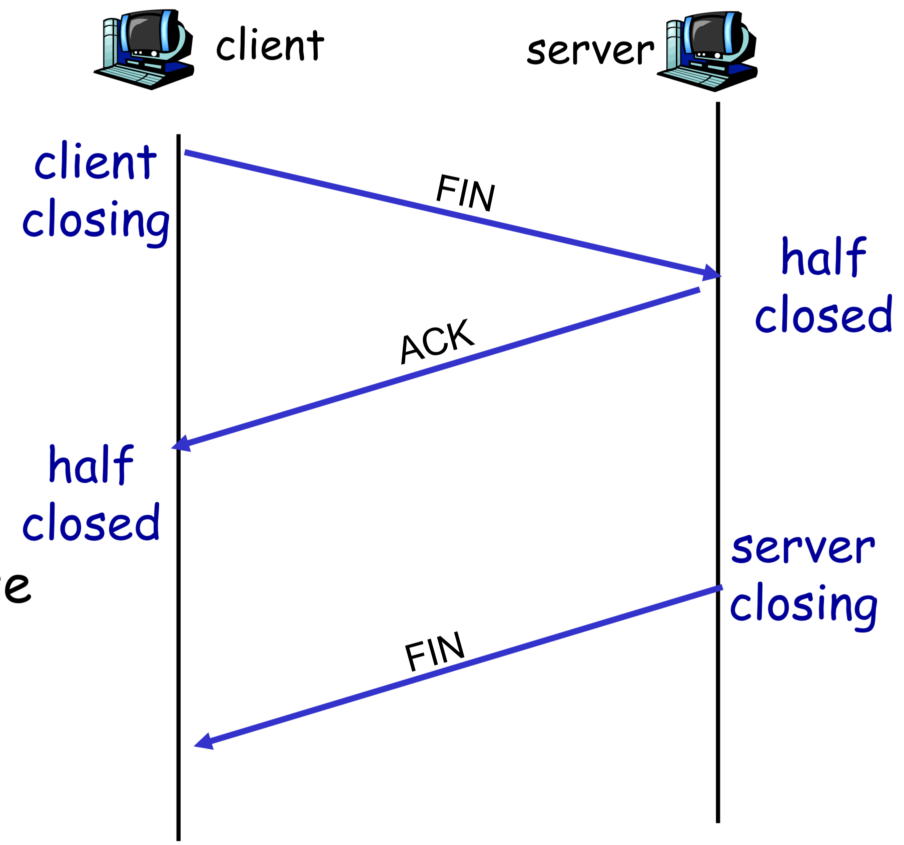
Step 2: server receives FIN, replies with ACK. half closed

Step 3: client receives FIN.

half closed, wait for server to close

Server finishes sending data, also ready to close:

Step 4: server sends FIN.



TCP: Closing Connection (revised)

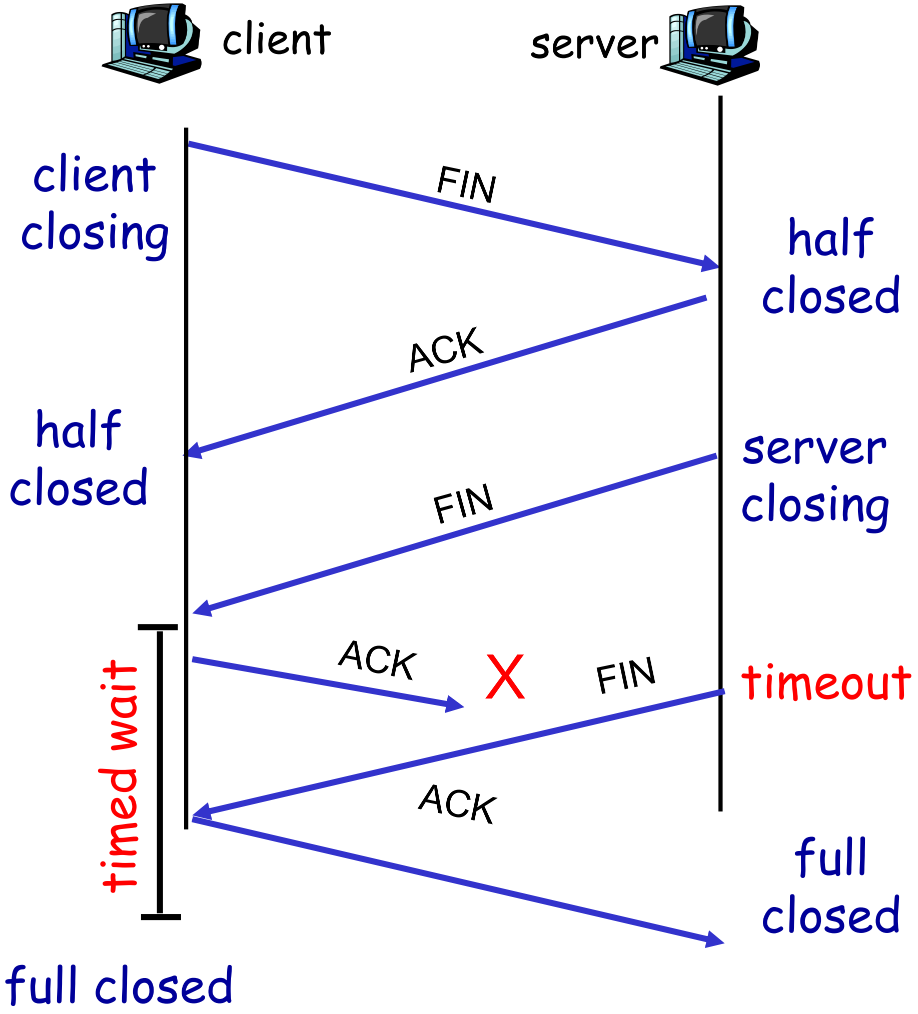
Two Army Problem!

Step 5: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

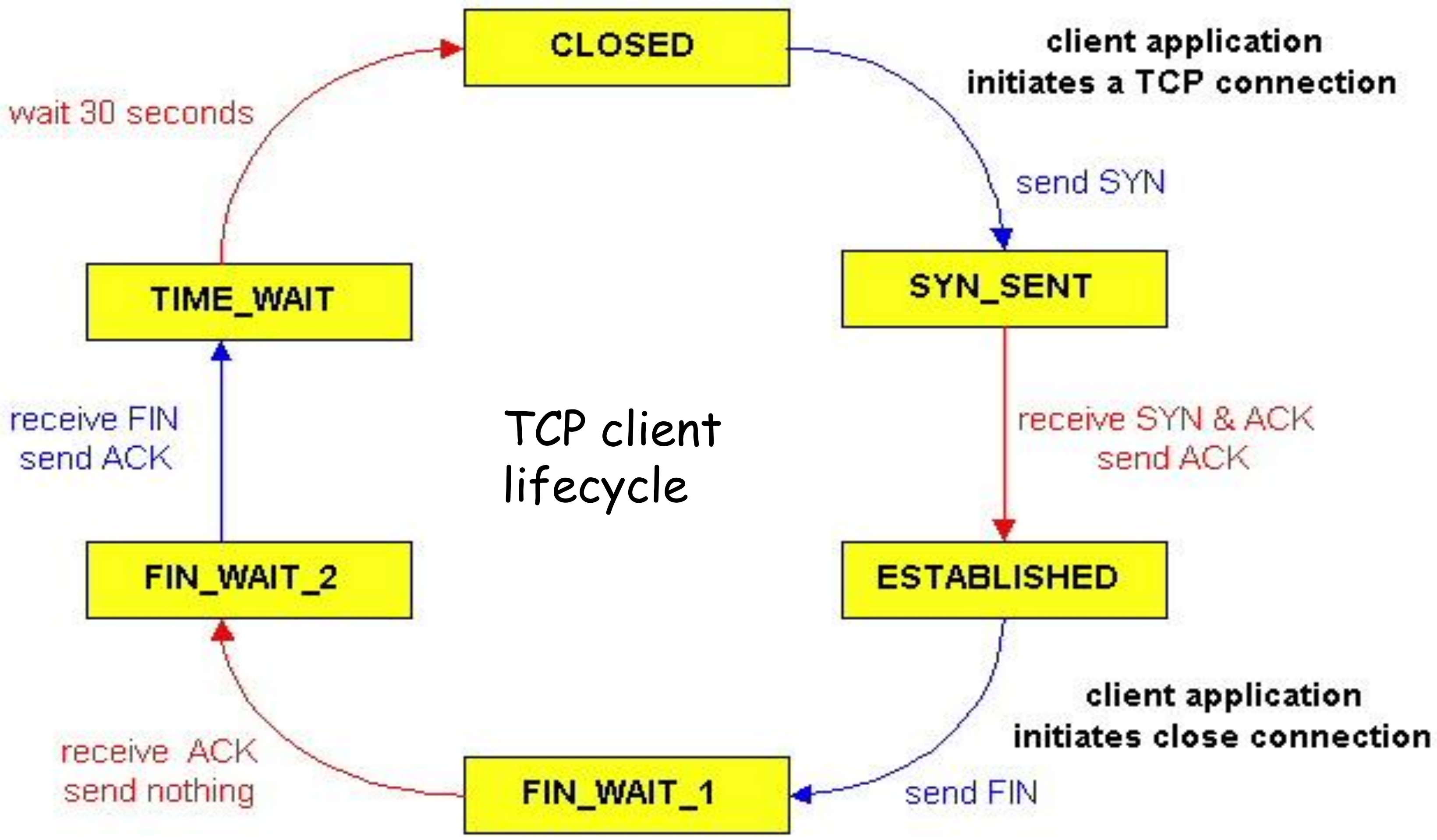
Step 6: server, receives ACK. connection fully closed

Step 7: client, timer expires, connection fully closed



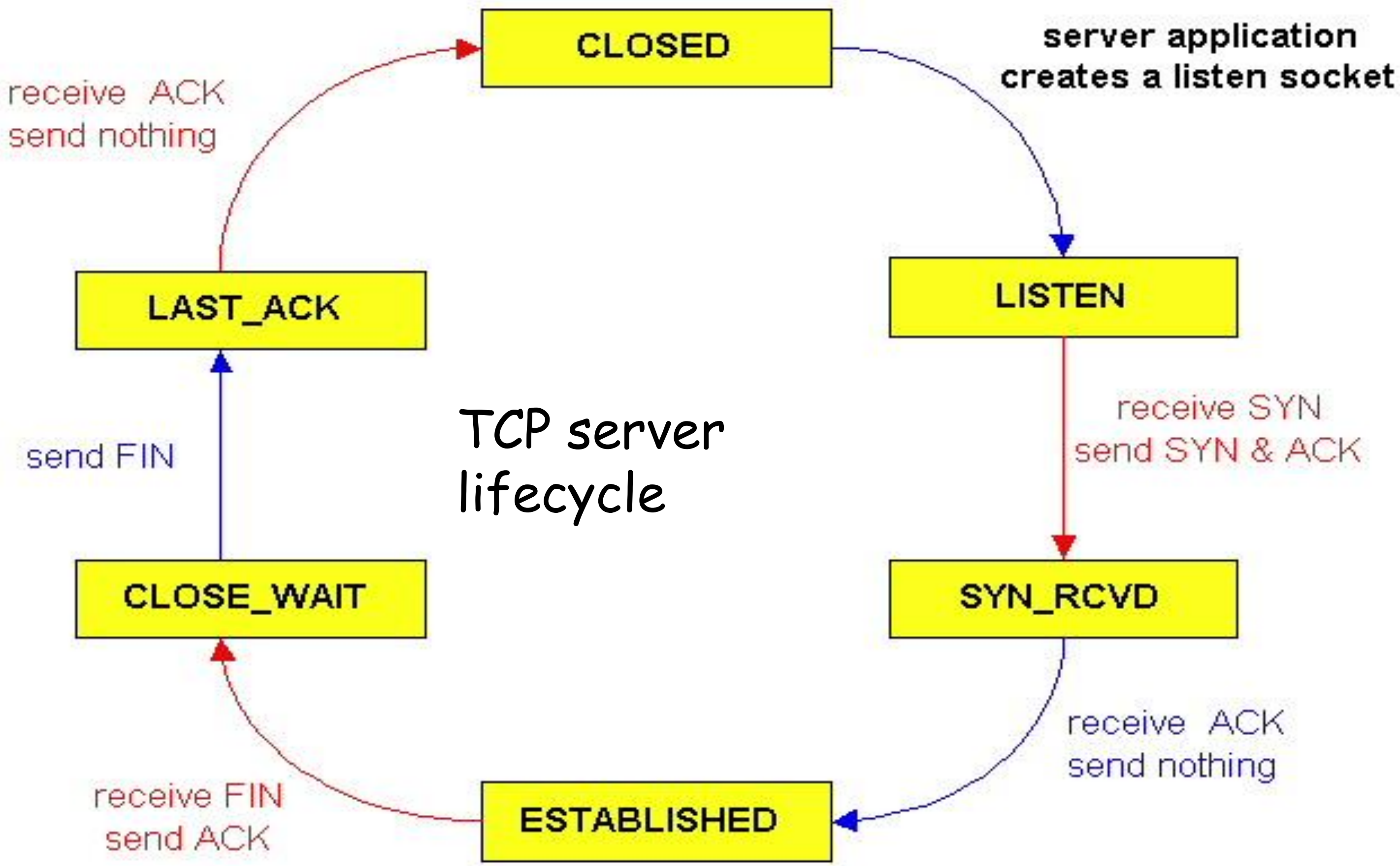
TCP Connection Management FSM

TCP client lifecycle



TCP Connection Management FSM

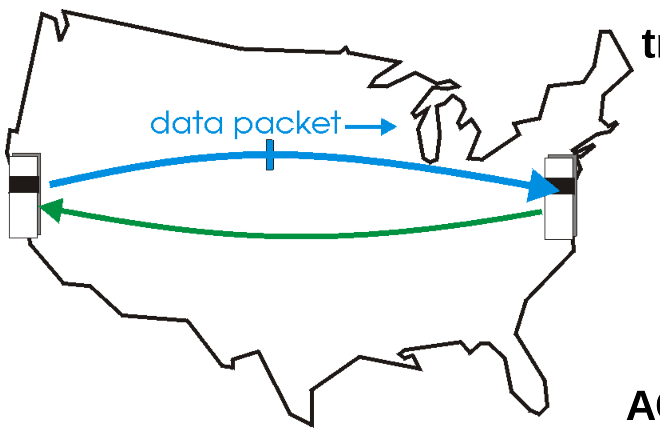
TCP server lifecycle



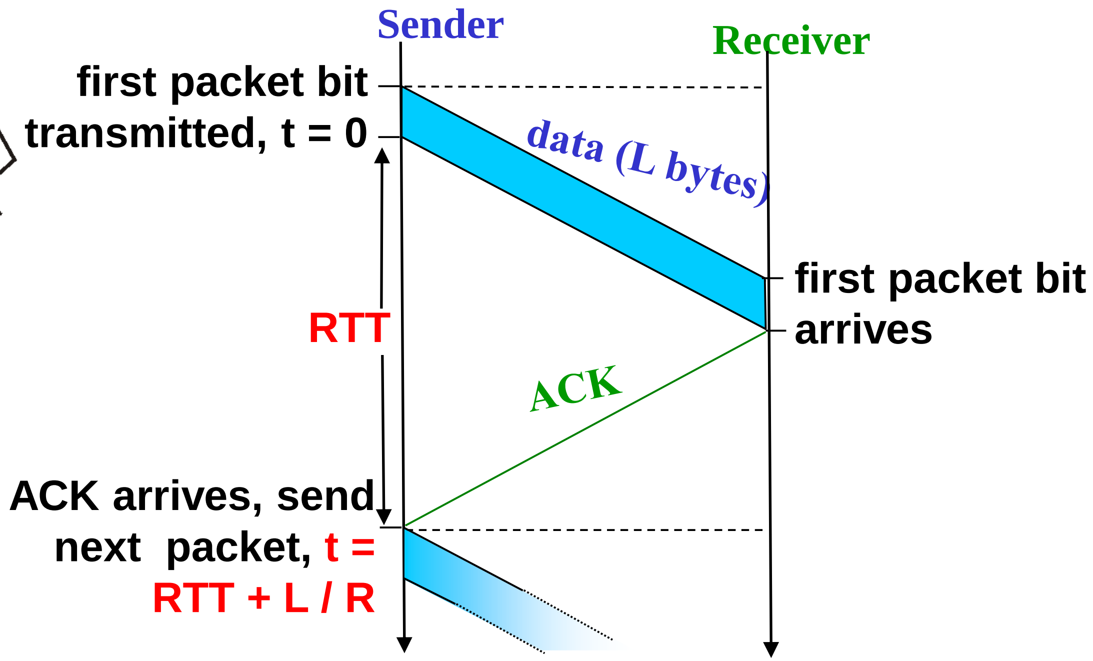
Recall: Simple Reliable Data Transfer Protocol

- "Stop-and-Wait" Protocol
  - also called Alternating Bit Protocol
- Sender:
  - i) send data segment (n bytes) w/ seq =x
    - buffer data segment, set timer, retransmit if time out
  - ii) wait for ACK w/ack = x+n; if received, set x:=x+n, go to i)
    - retransmit if ACK w/ "incorrect" ack no. received
- Receiver:
  - i) expect data segment w/ seq =x; if received, send ACK w/ ack=x+n, set x:=x+n, go to i)
    - if data segment w/ "incorrect" seq no received, discard data segment, and retransmit ACK.

Problem with Stop & Wait Protocol



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



- Can't keep the pipe full
  - Utilization is low
    when bandwidth-delay product (R x RTT)is large!

Stop & Wait: Performance Analysis

Example:

1 Gbps connection, 15 ms end-end prop. delay,

data segment size: 1 KB = 8Kb

$$T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8 \text{ kb}}{10^9 \text{ b/s}}$$

$$= 8 \times 10^{-6} \text{s} = 0.008 \text{ ms}$$

$$\boldsymbol{U}_{\textbf{\textit{sender}}} = \frac{L/R}{RTT + L/R} = \frac{L}{RTT * R + L} = \frac{.008}{30.008} = 0.00027$$

- U_{sender}: utilization, i.e., fraction of time sender busy sending
- 1KB data segment every 30 msec (round trip time)

  --> 0.027% x 1 Gbps = 33kB/sec throughput over 1 Gbps link

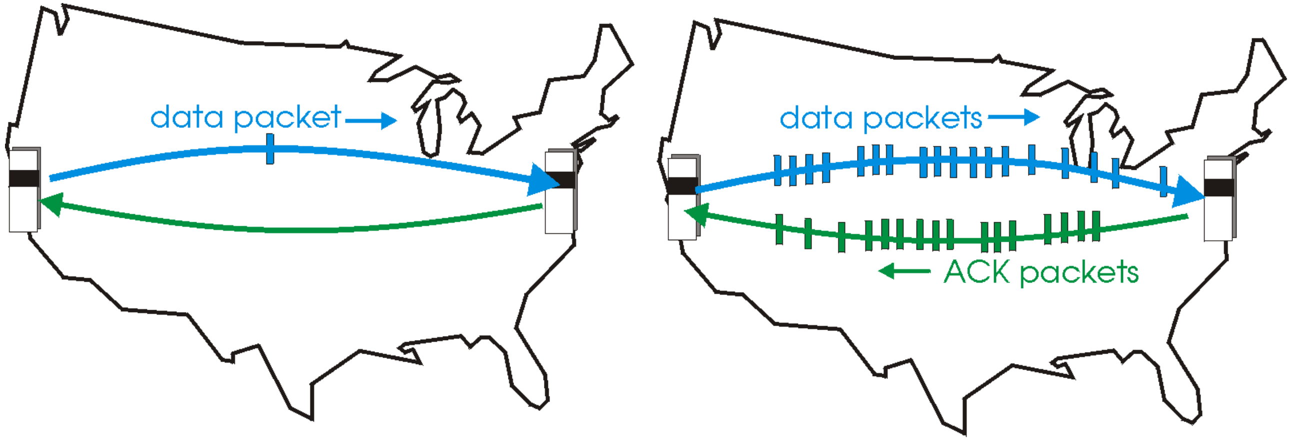
Moral of story:

network protocol ***limits*** use of physical resources!

Pipelined Protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged data segments

- range of sequence numbers must be increased
- buffering at sender and/or receiver

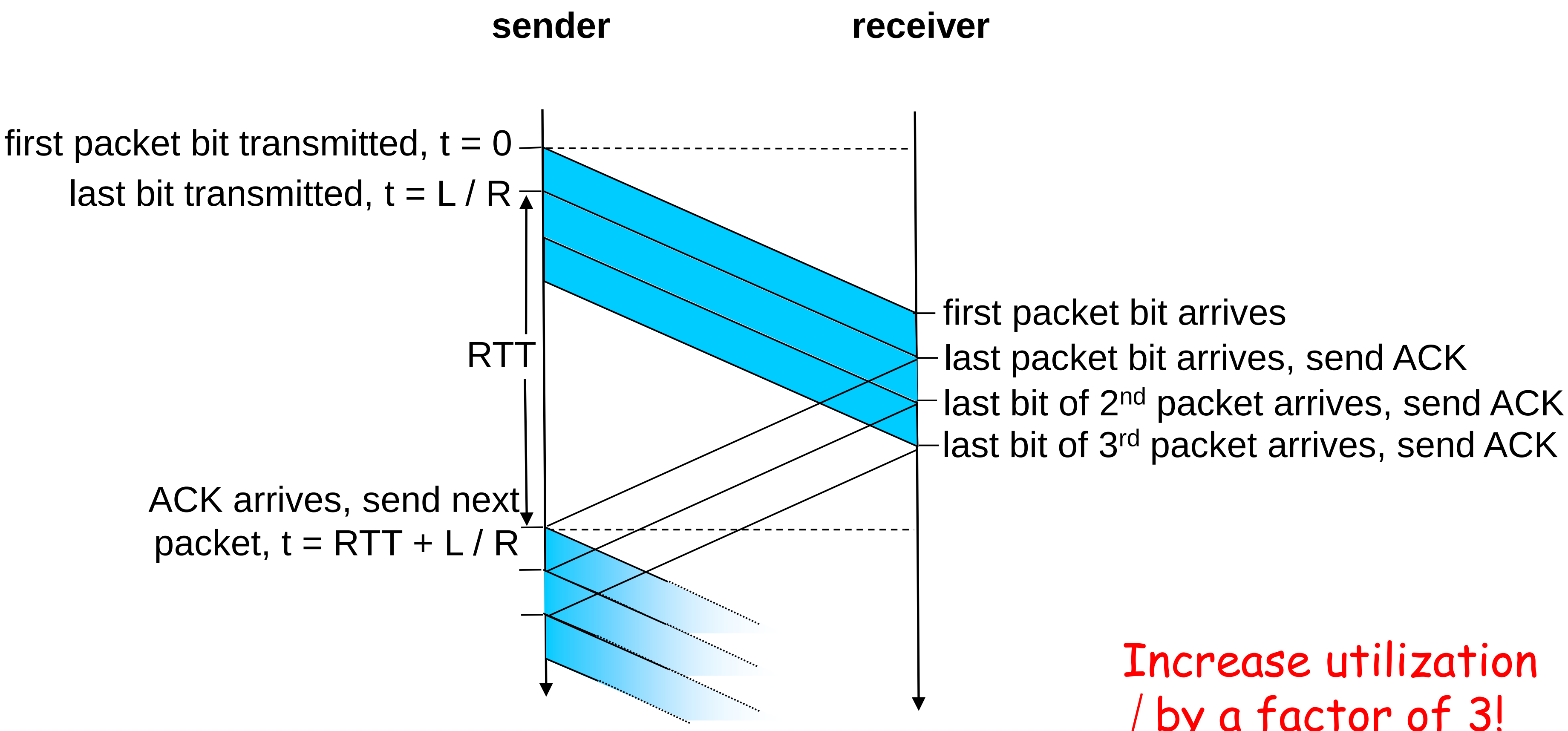


(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols:

  Go-Back-N and Selective Repeat

Pipelining: Increased Utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N: Basic Ideas

Sender:

- Packets transmitted continually (when available) without waiting for ACK, up to N outstanding, unACK'ed packets
- A logically different timer associated with each "in-flight" (i.e., unACK'ed) packet
  - *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

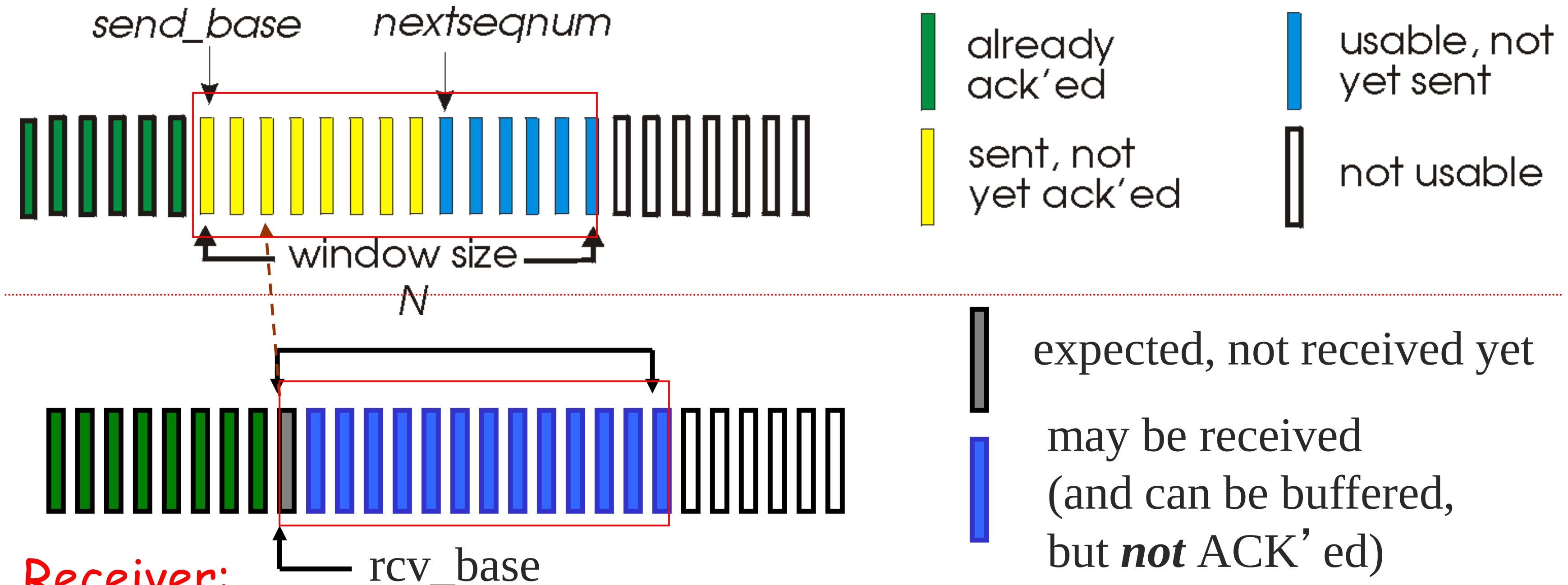
Receiver:

- ACK packet if corrected received and in-order, pass to higher layer, NACK or ignore corrupted or out-of-order packets
- "cumulative" ACK: if multiple packets received corrected and in-order, send only one ACK with ack= next expected seq no.

Go-Back-N: Sliding Windows

Sender:

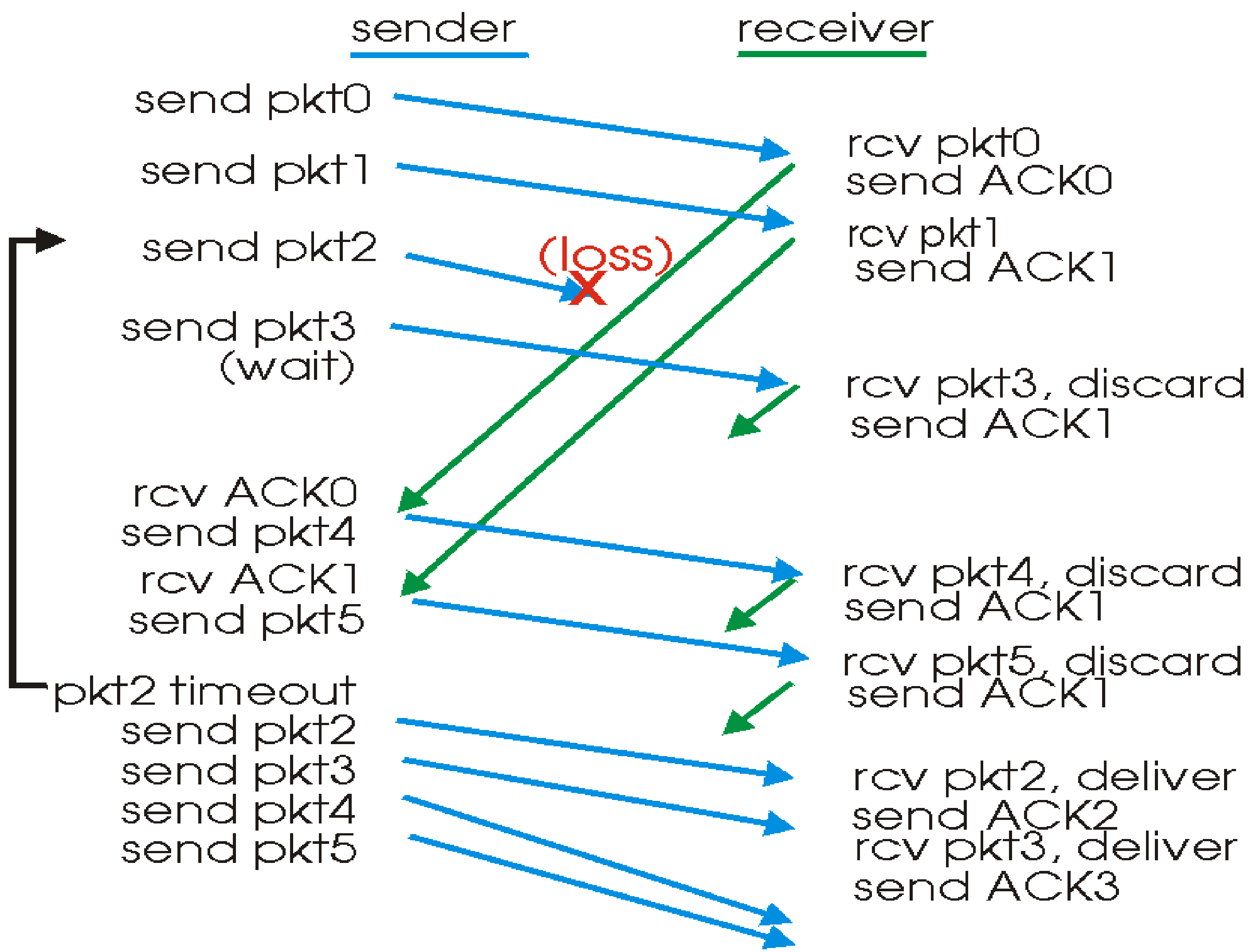
- "window" of up to N, consecutive unack'ed pkts allowed
- send_base: first sent but unACKed pkt, move forward when ACK'ed



Receiver:

- rcv_base: keep track of next expected seq no, move forward when next in-order (i.e., w/ expected seq no) pkt received

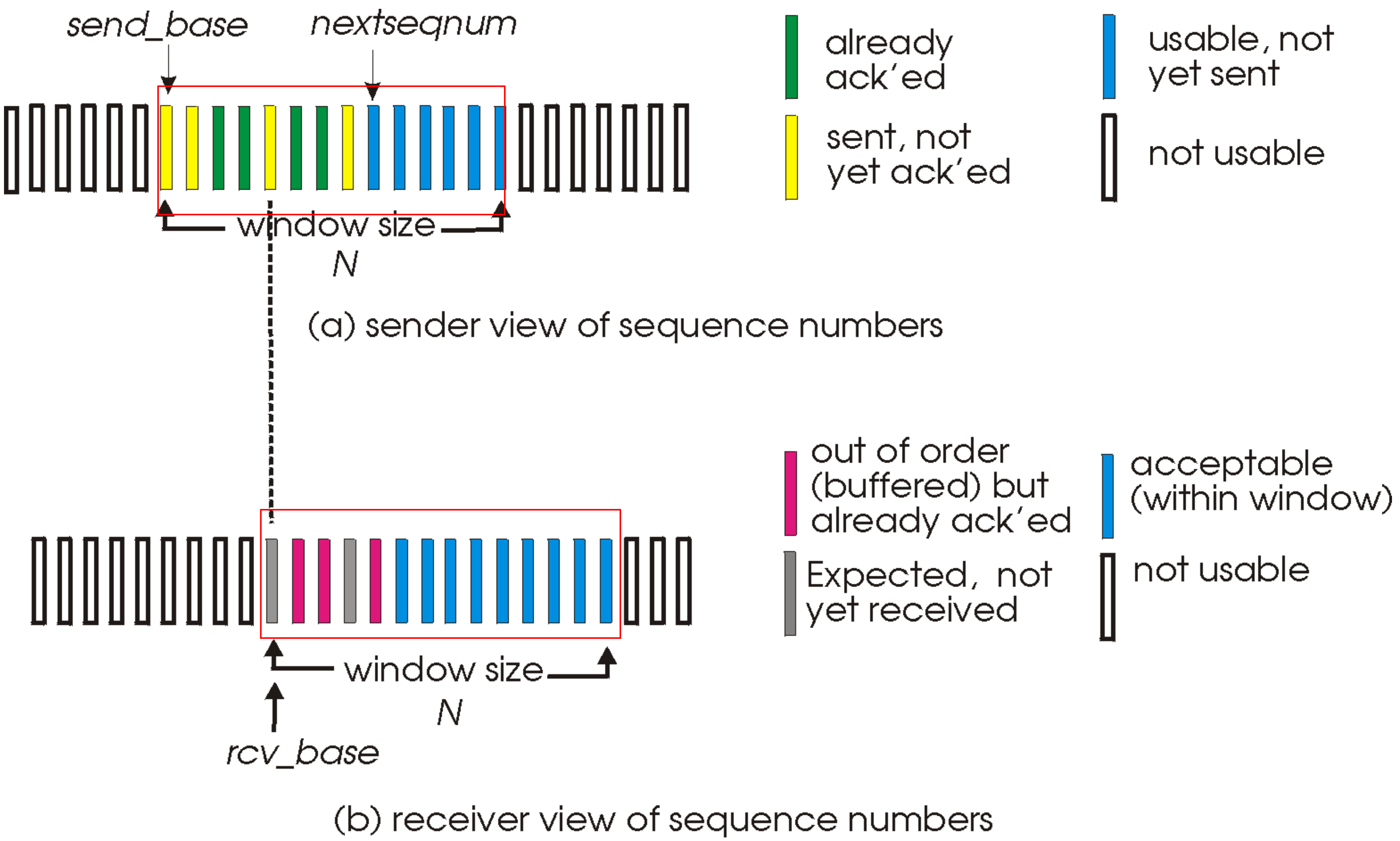
GBN in Action



Selective Repeat

- As in Go-Back-N
  - Packet sent when available up to window limit
- Unlike Go-Back-N
  - Out-of-order (but otherwise correct) is ACKed
  - Receiver: buffer out-of-order pkts, no “cumulative” ACKs
  - Sender: on timeout of packet k, retransmit just pkt k
- Comments
  - Can require more receiver buffering than Go-Back-N
  - More complicated buffer management by both sides
  - Save bandwidth
    - no need to retransmit correctly received packets

Selective Repeat: Sliding Windows



Selective Repeat: Algorithms

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

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

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

receiver

pkt n in [rcvbase, rcvbase+N-1]

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

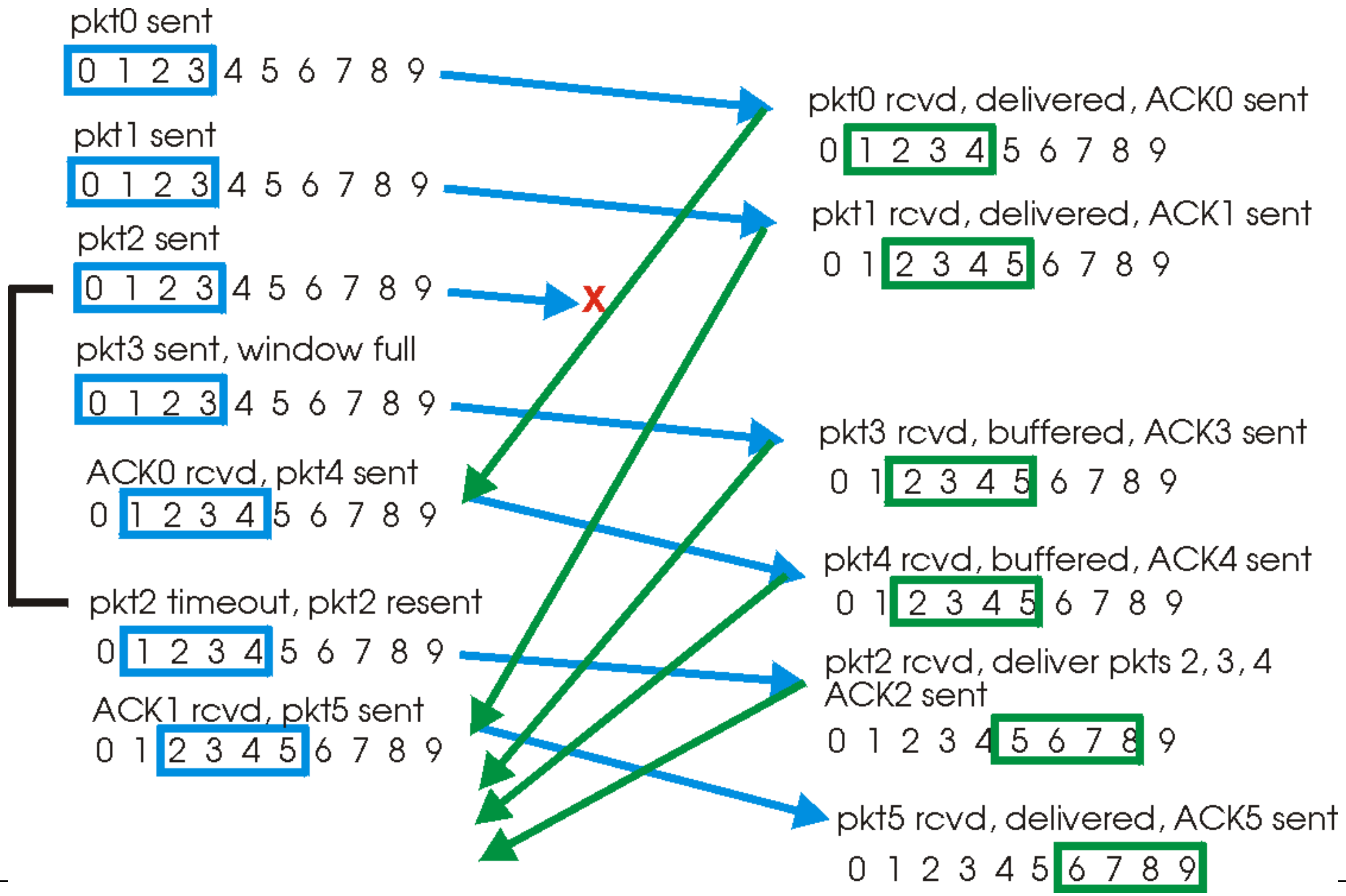
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective Repeat in Action



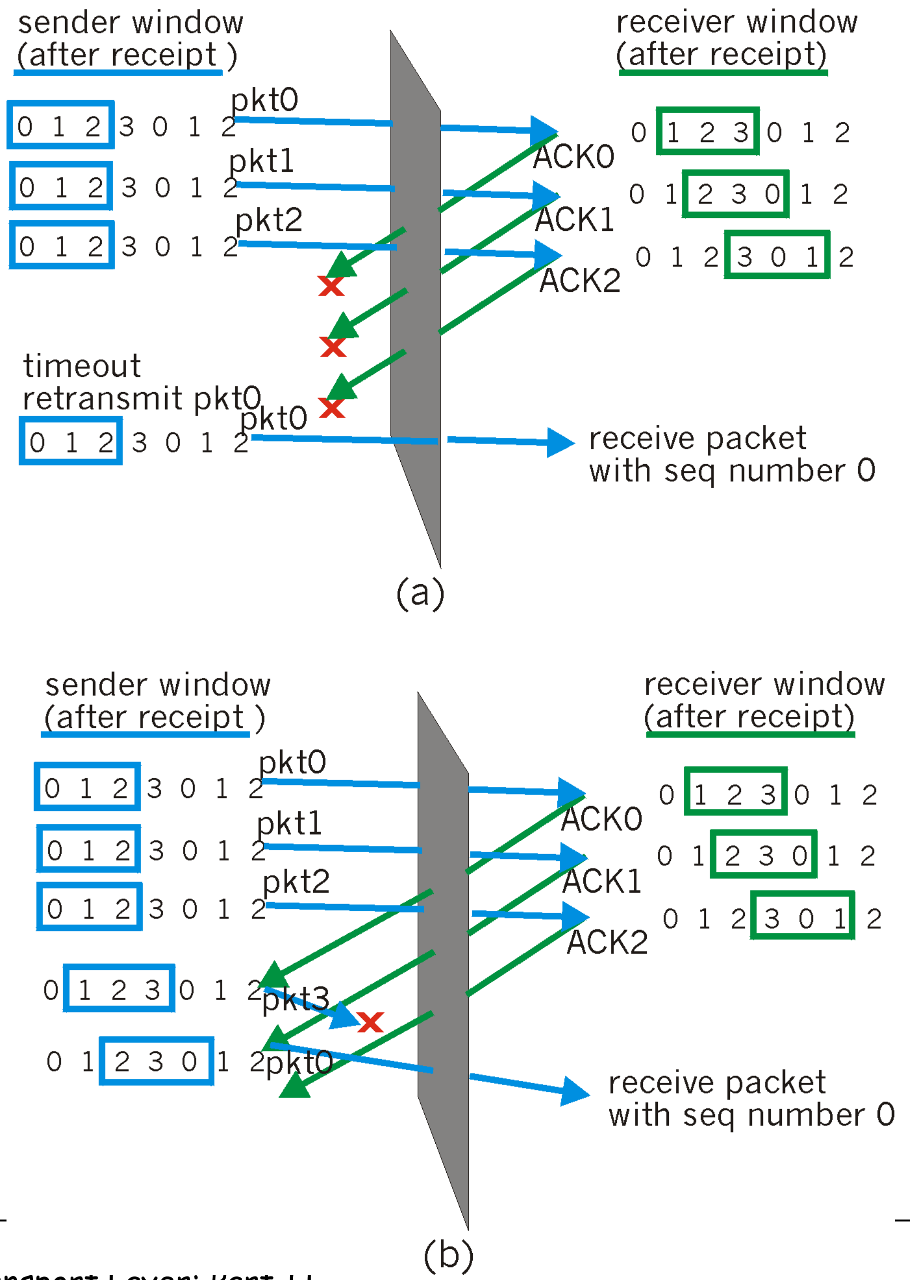
Selective Repeat: Dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?



Seqno Space and Window Size

- How big the sliding window can be?
  - MAXSEQNO: number of available sequence numbers
  - Under Go-Back-N?
    - MAXSEQNO will not work, why?
  - What about Selective-Repeat?

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions, why?
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current `SampleRTT`

TCP Round Trip Time Estimation

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $a = 0.125$

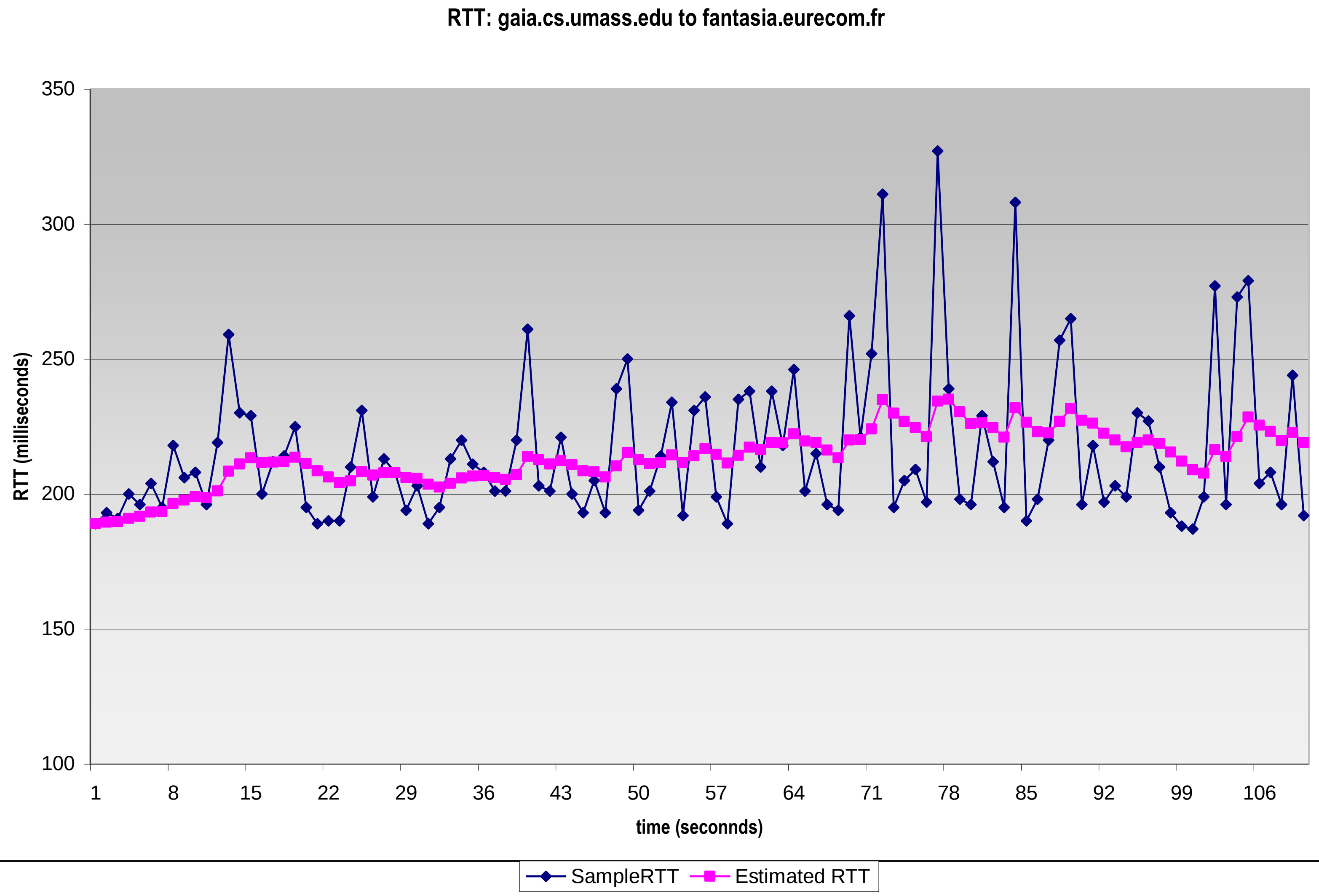
Setting the timeout interval

- `EstimtedRTT` plus "safety margin"
  - large variation in `EstimatedRTT` -> larger safety margin
- "safety margin": accommodate variations in estimatedRTT

```
DevRTT = (1-b)*DevRTT + b*|SampleRTT-EstimatedRTT|
(typically, b= 0.25)
```

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

Example RTT Estimation:



TCP Reliable Data Transfer

- TCP creates reliable data transfer service on top of IP's unreliable service
- Pipelined segments
- Cumulative ACKs
- TCP uses single retransmission timer
  - double TimeoutInterval on timer expiration

- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP Sender Events:

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

ACK received:

- If acknowledges previously unACKed segments, then
  - update what is known to be ACKed
  - start timer if there are outstanding segments

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver Action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	**Delayed ACK.** Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single **cumulative** ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send **duplicate** ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap