Module M6

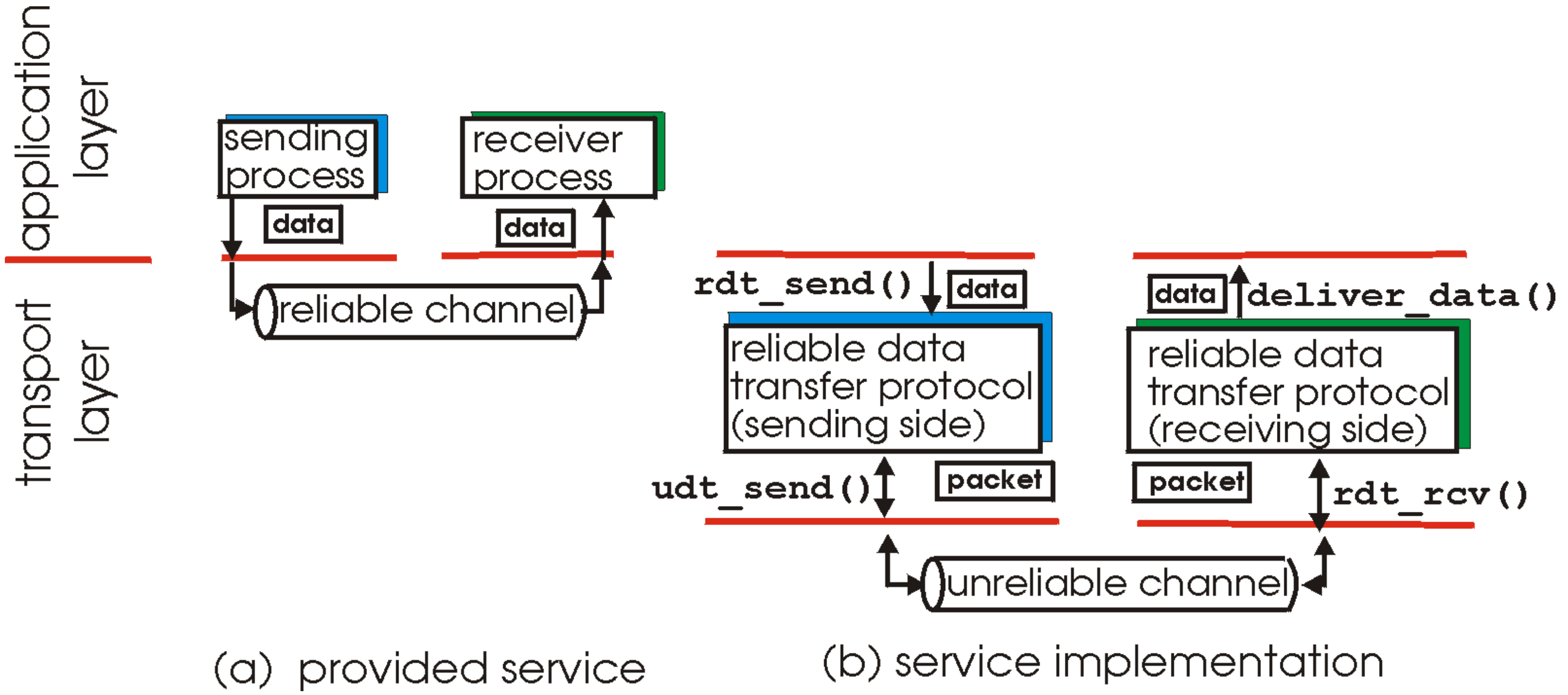
CPSC 317

November 2, 2022

RELIABLE DATA TRANSFER

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

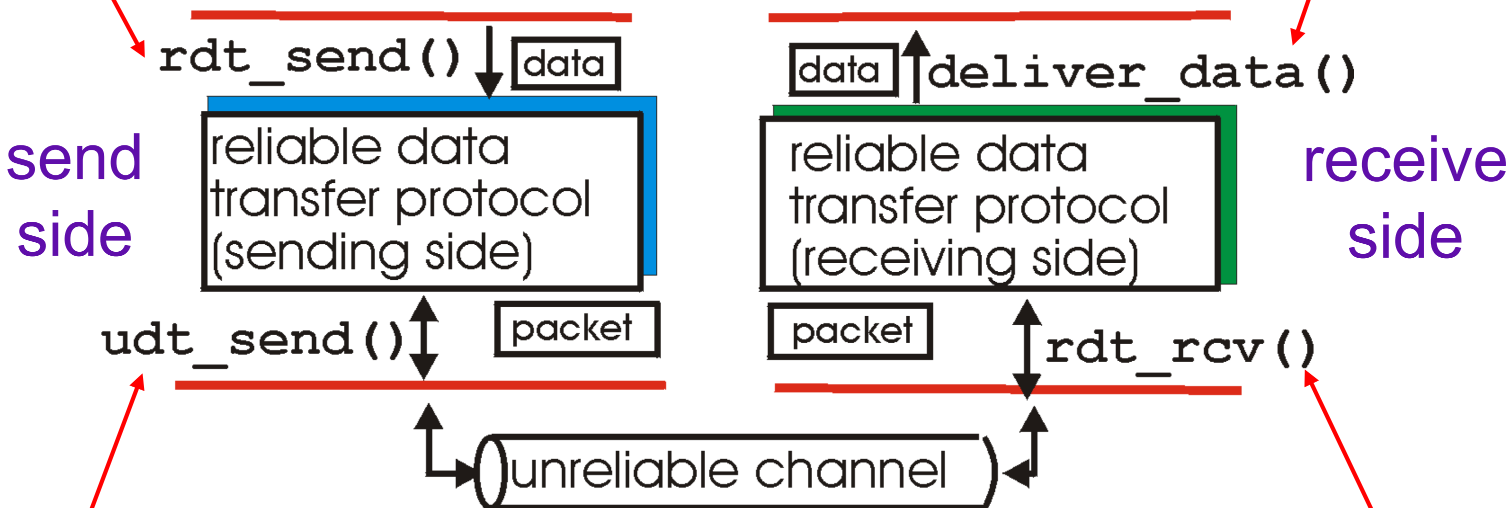


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

The plan

- ❑ Reliable channel
- ❑ Channel that can corrupt messages
- ❑ Channel that can corrupt and lose messages
- ❑ What if we can re-order messages???

Can't do this one

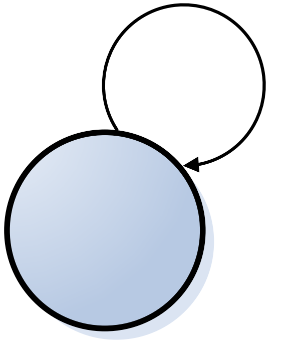
Programming State Machines

- What does the software look like?

```
Switch( event ):
   event:
         action()
   event:
         action()
End_switch
```

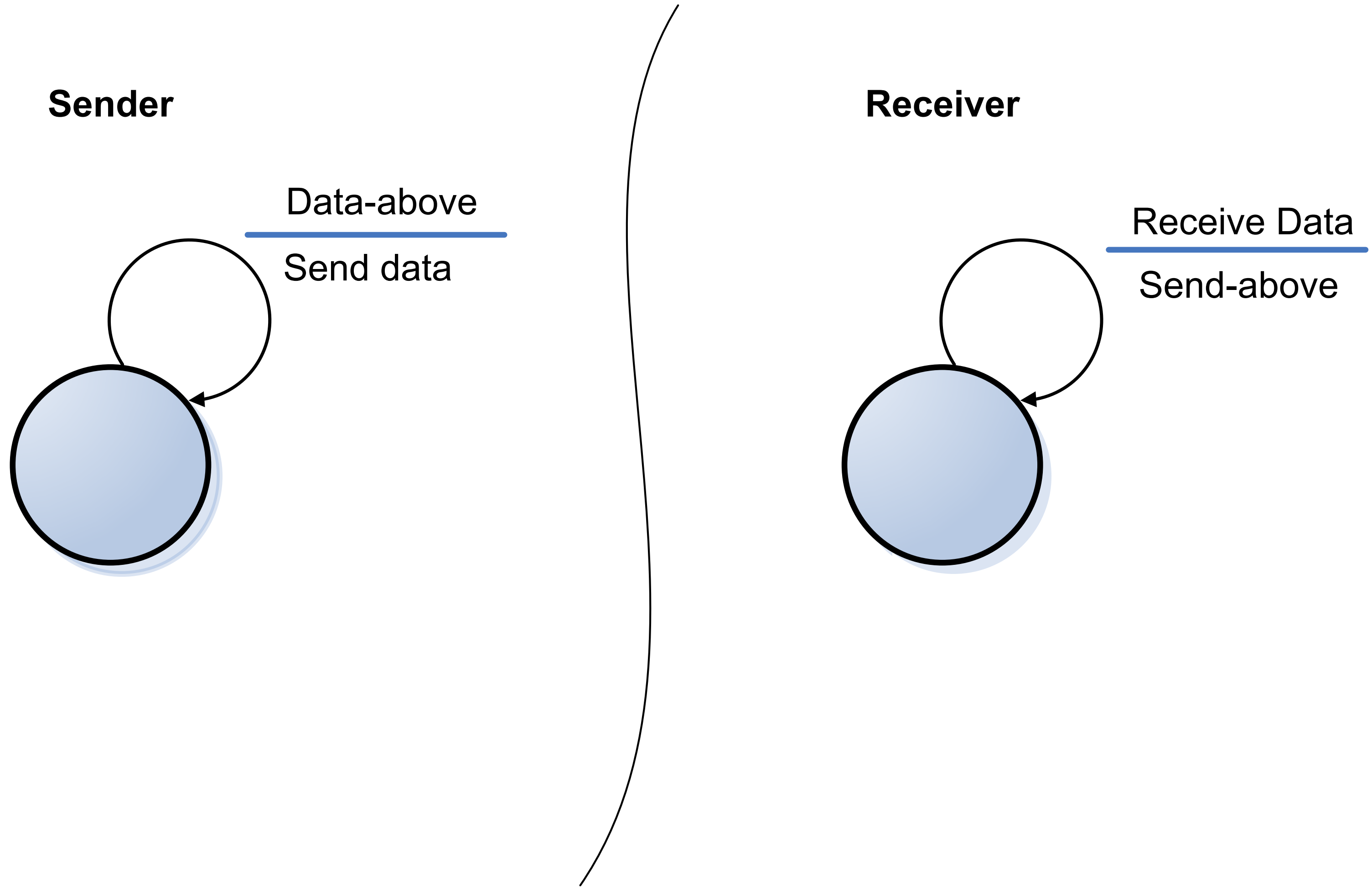
State Machines: Events and Actions

- Events:



- Actions:

Reliable Channel Communicating State Machines



Unreliable -- Bit Errors

- Messages contents may be garbled.
- What do we do?

Scenario (trace)

Solution rdt 2.0

SENDER:

- Events
  - App message ready
  - NAK recv'ed
  - ACK recv'ed

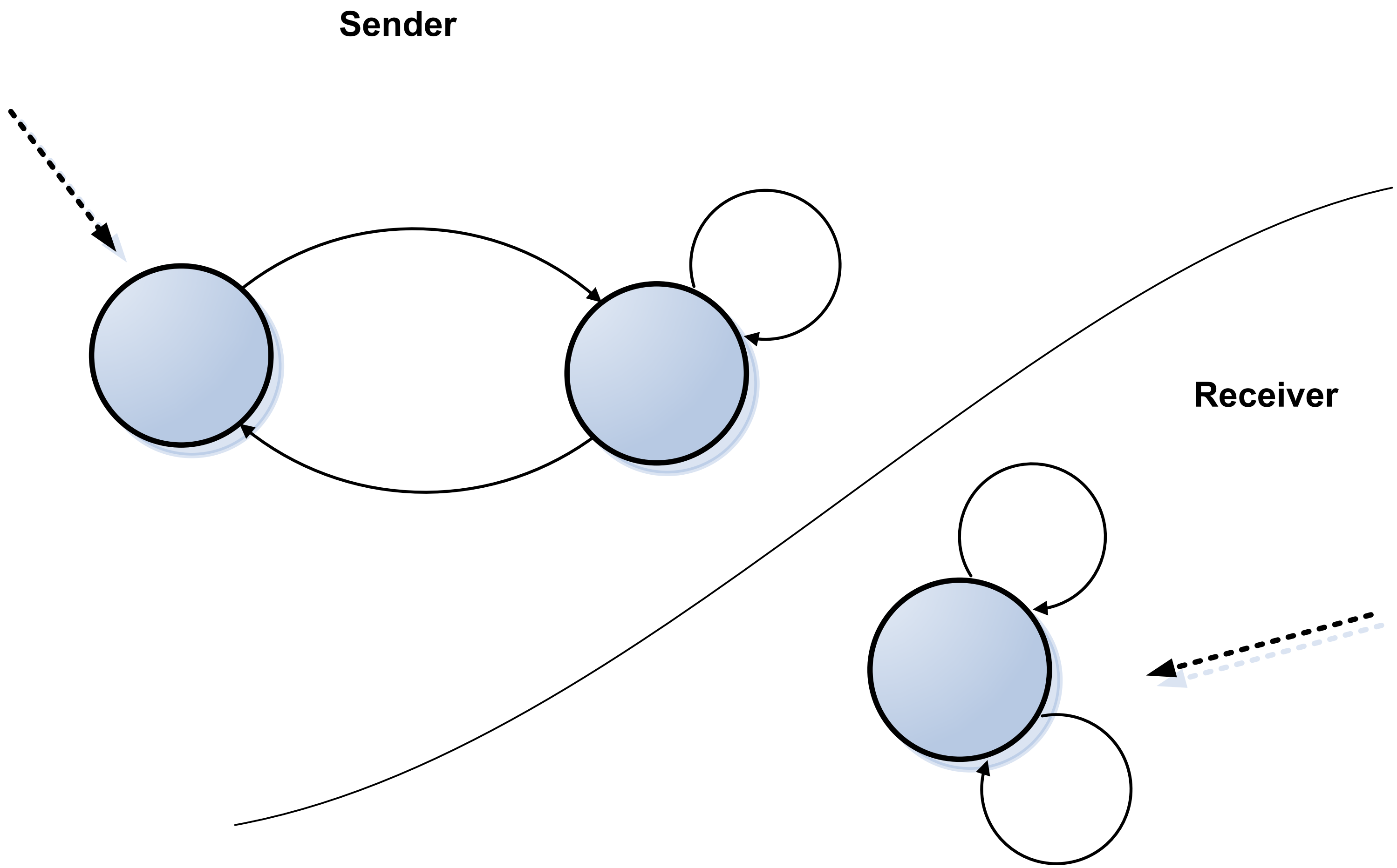
- Actions
  - Recv from app
  - Send to link

RECEIVER:

- Events
  - Link packet ready
  - Corrupt packet

- Actions
  - Send message to app
  - Discard, send NAK
  - Send ACK

rdt 2.0 -- State Diagrams

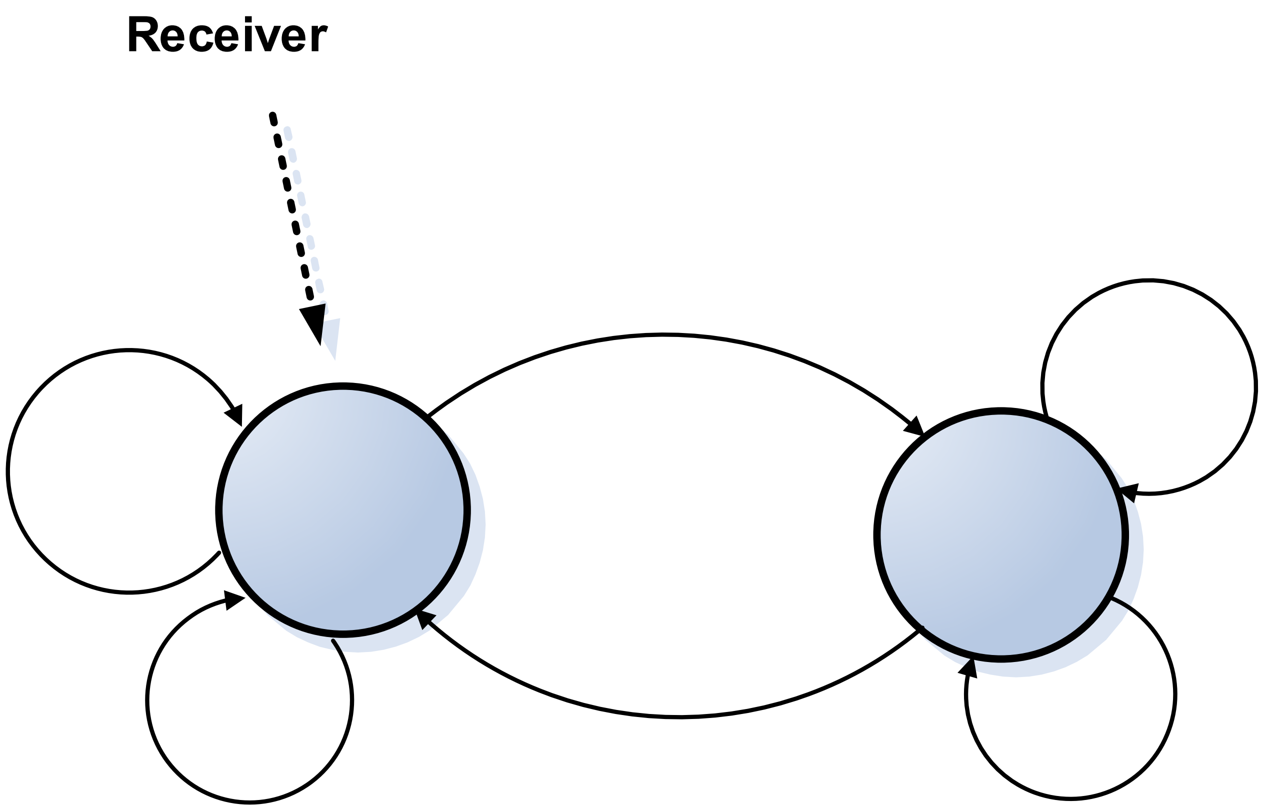


Scenario (corrupt ptk)

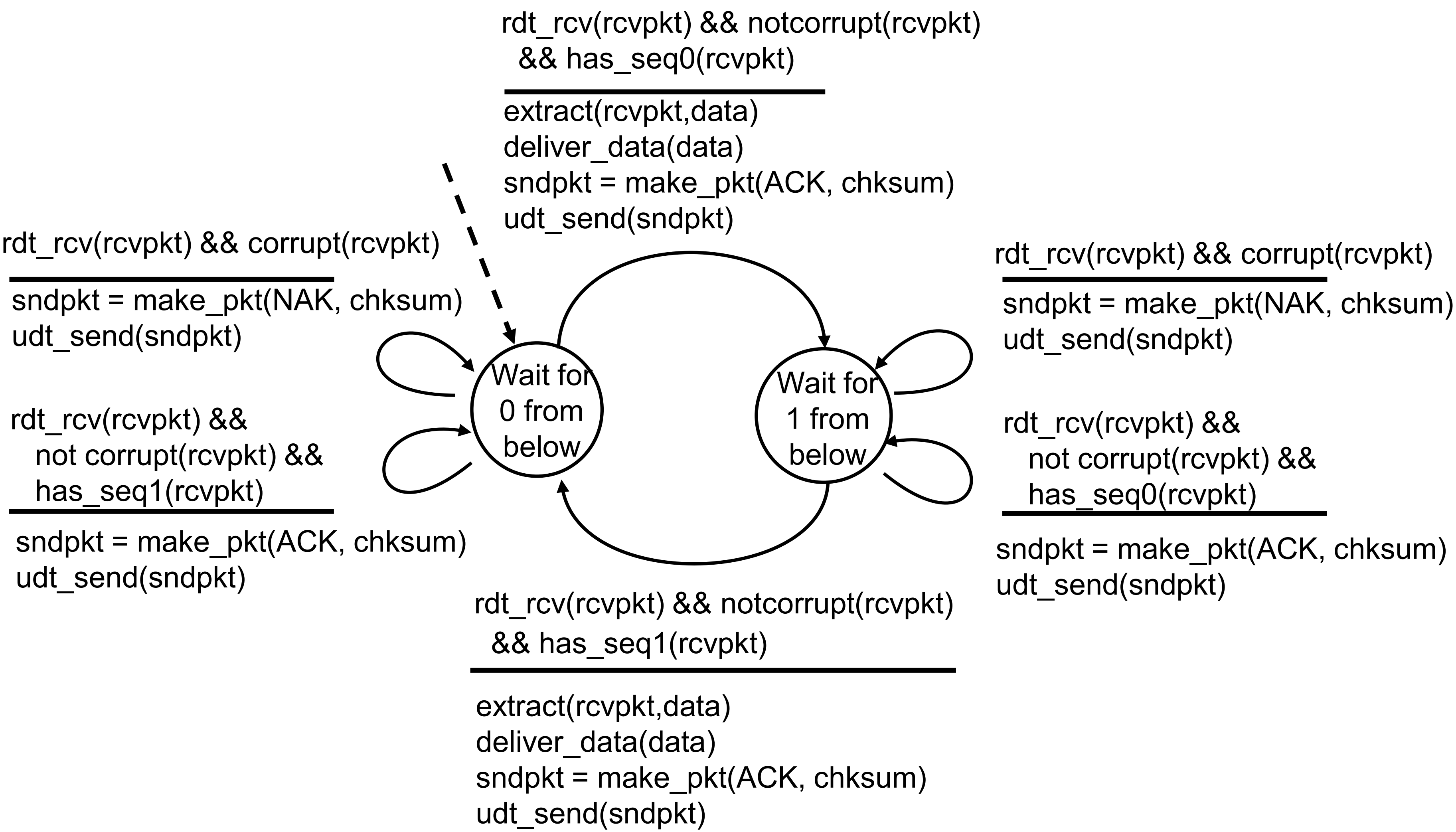
Scenario (corrupt ack/nack)

FIXING ACK/NACK PROBLEM

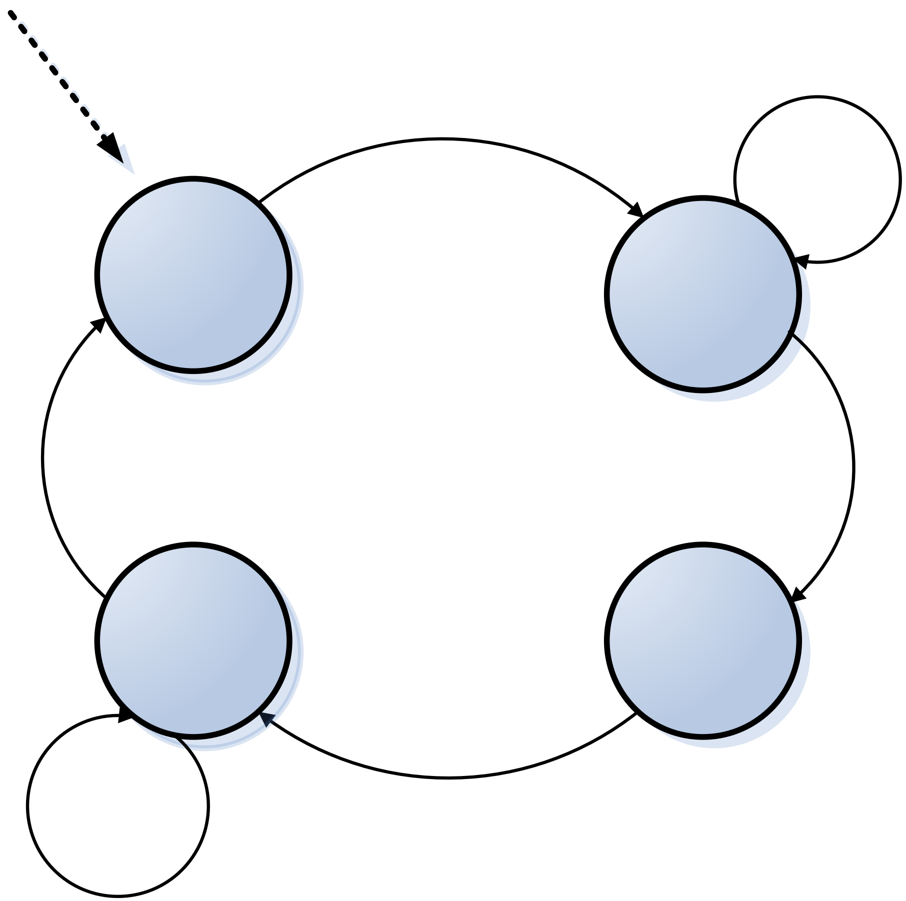
Receiver rdt2.1



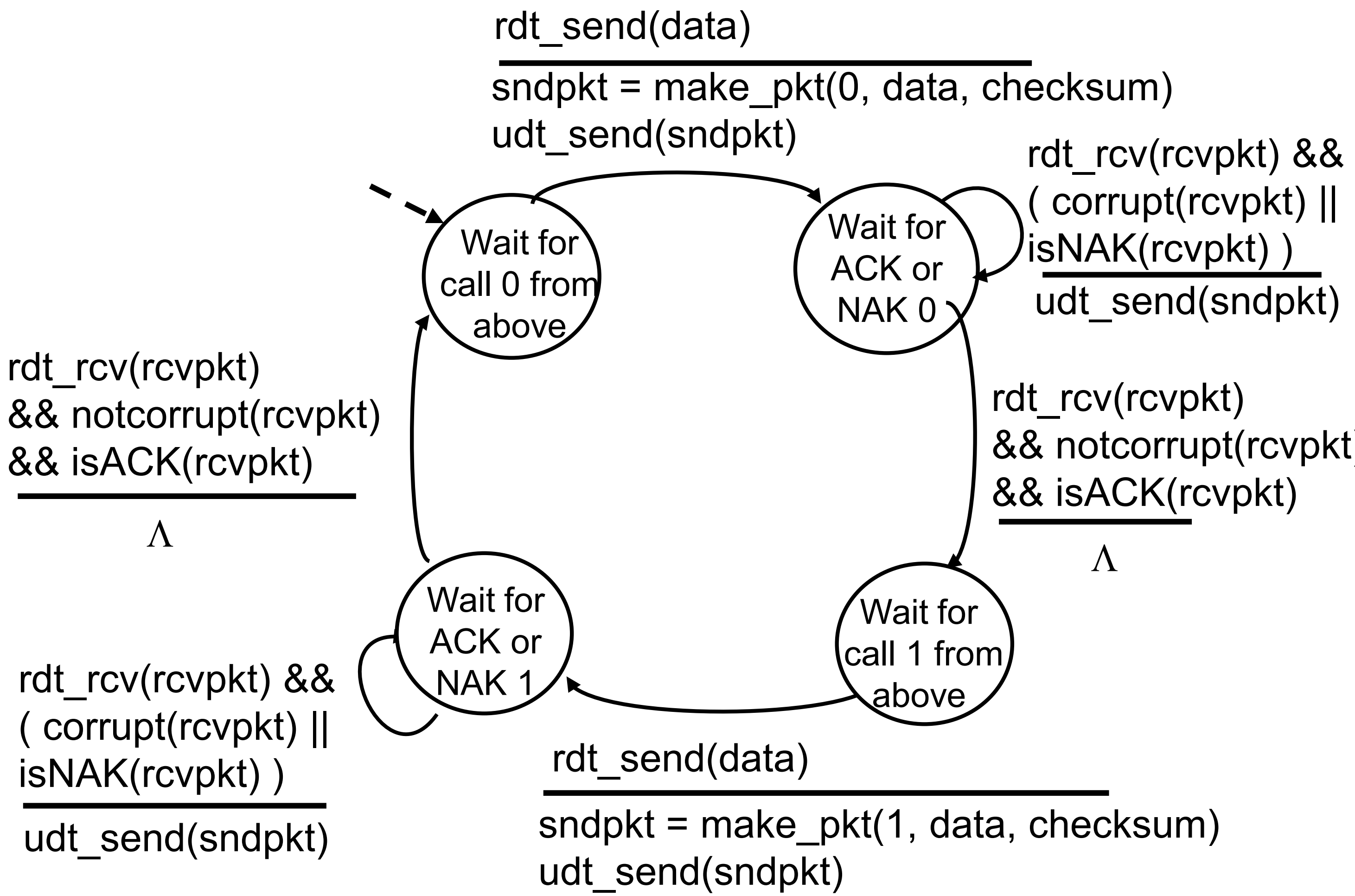
rdt2.1: receiver, handles garbled ACK/NAKs



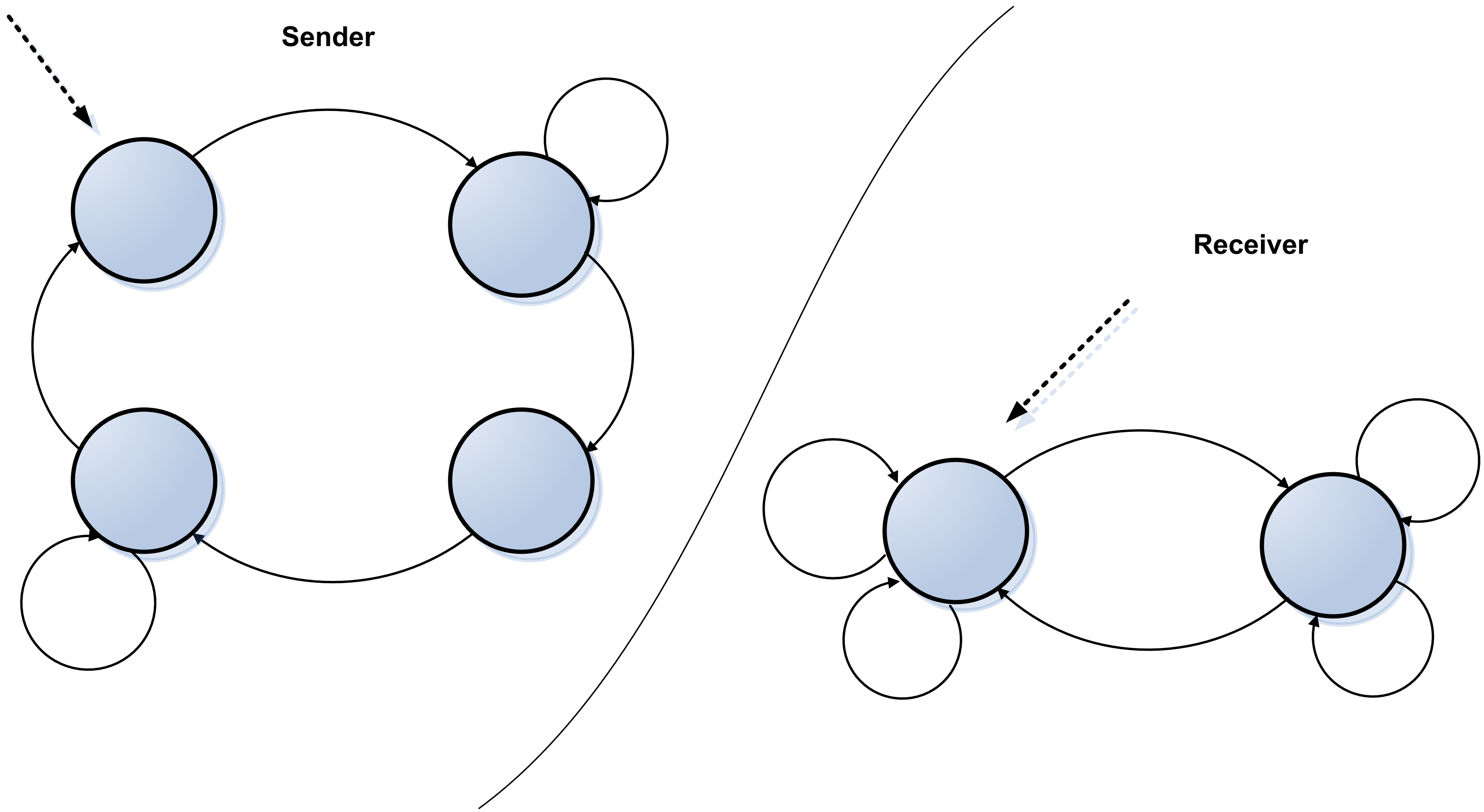
Sender rdt2.1



rdt2.1: sender, handles garbled ACK/NAKs

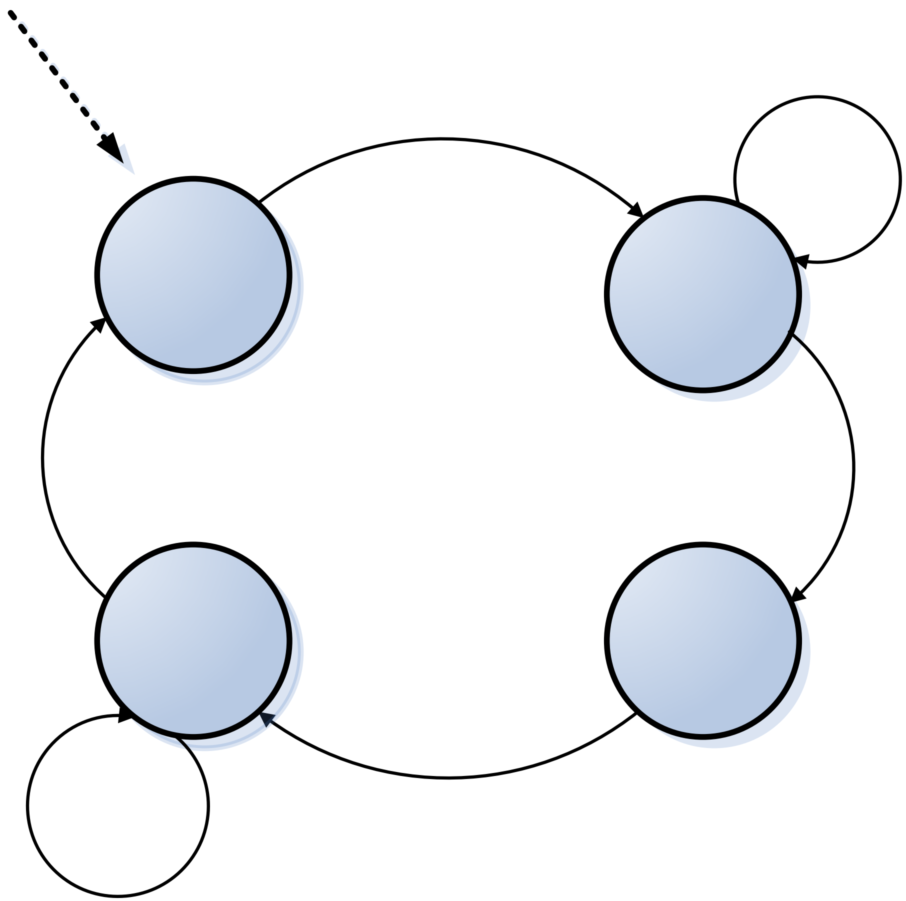


Solution rdt2.1

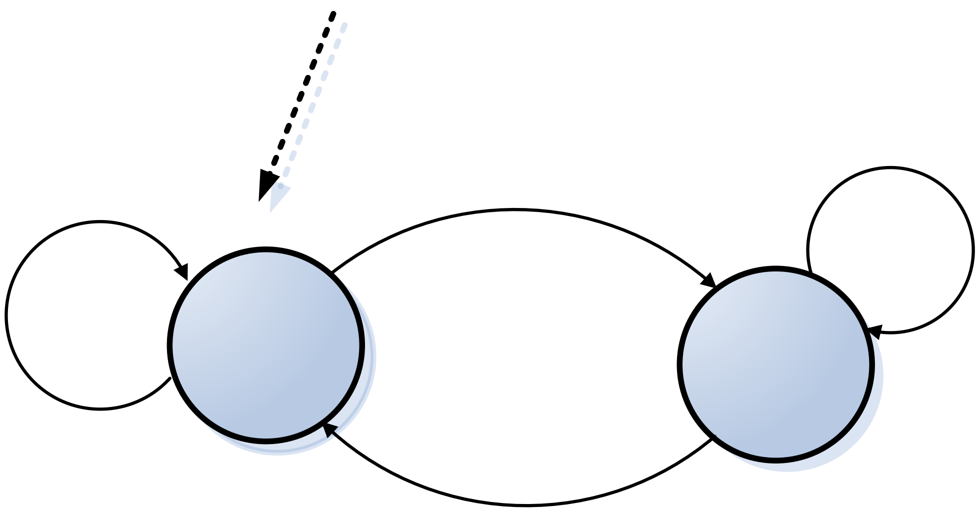


Scenario (corrupt nackless)

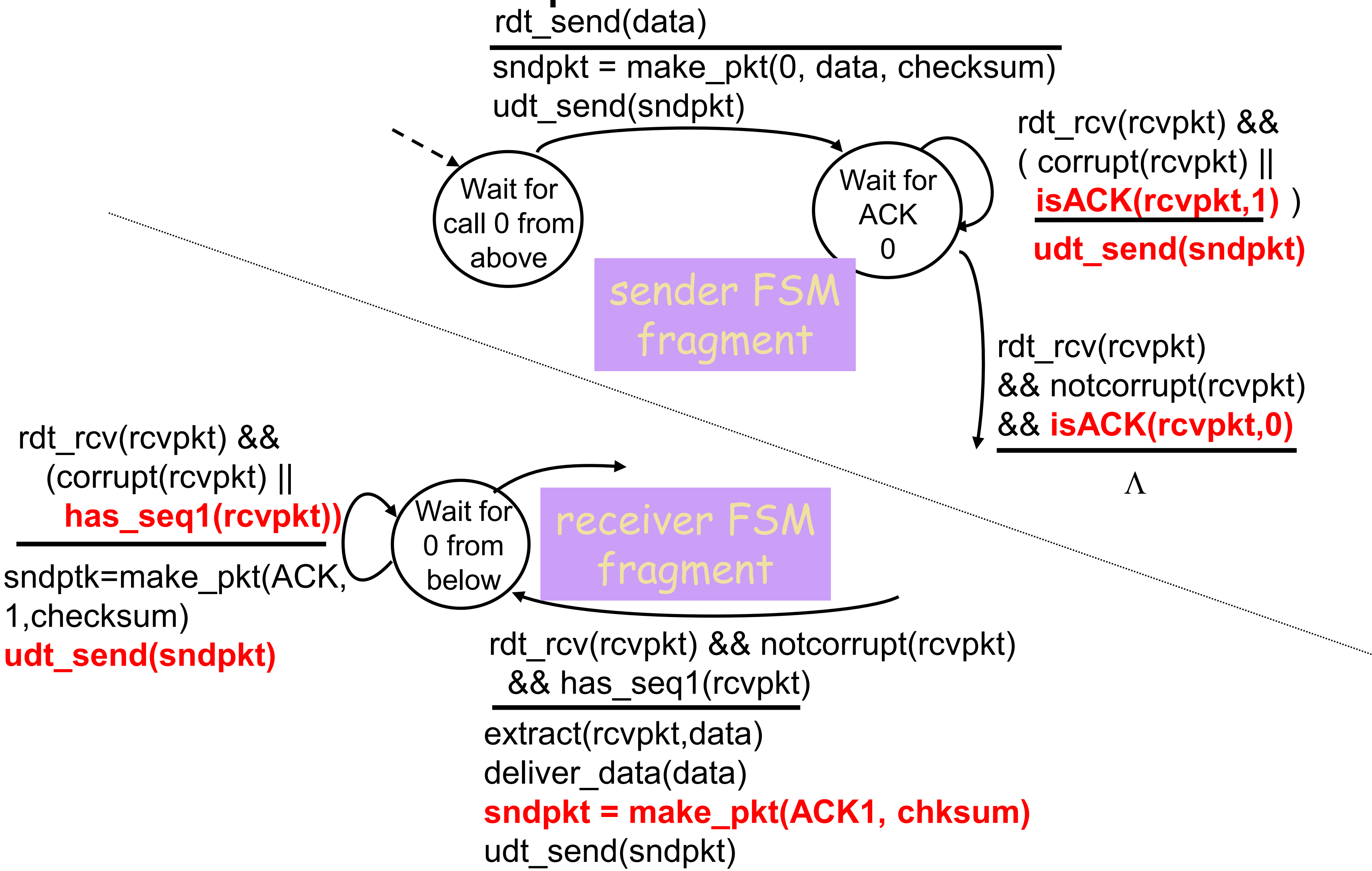
Nakless Sender rdt2.2



Nakless Receiver rdt2.2



rdt2.2: sender, receiver fragments: sequence numbers



LOSS

The plan

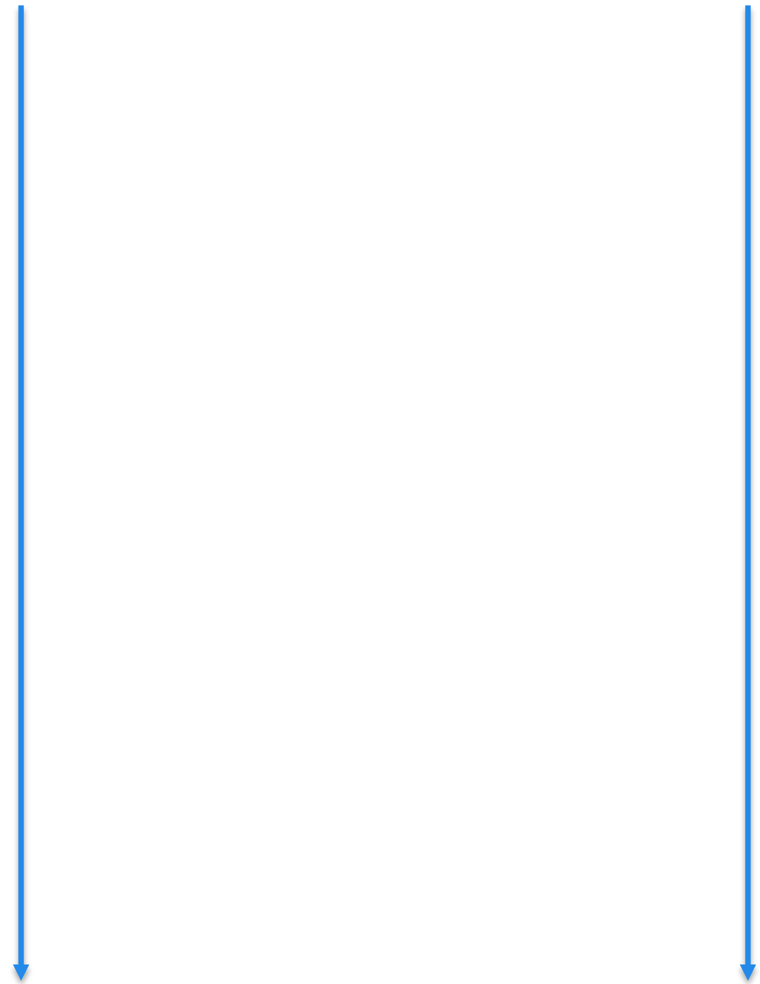
- Reliable channel
- Channel that can corrupt messages
- Channel that can corrupt and lose messages
- What if we can re-order messages???

Need

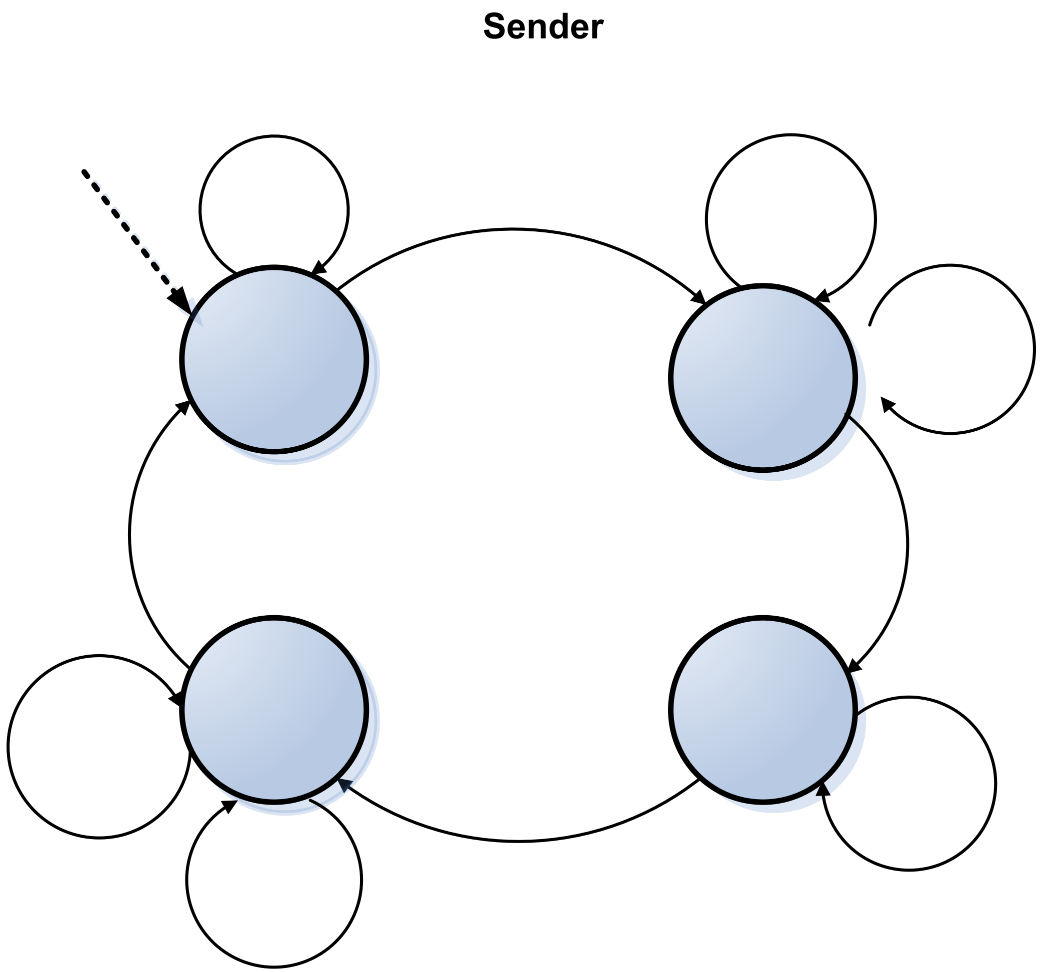
What to do about loss?

How to detect it?

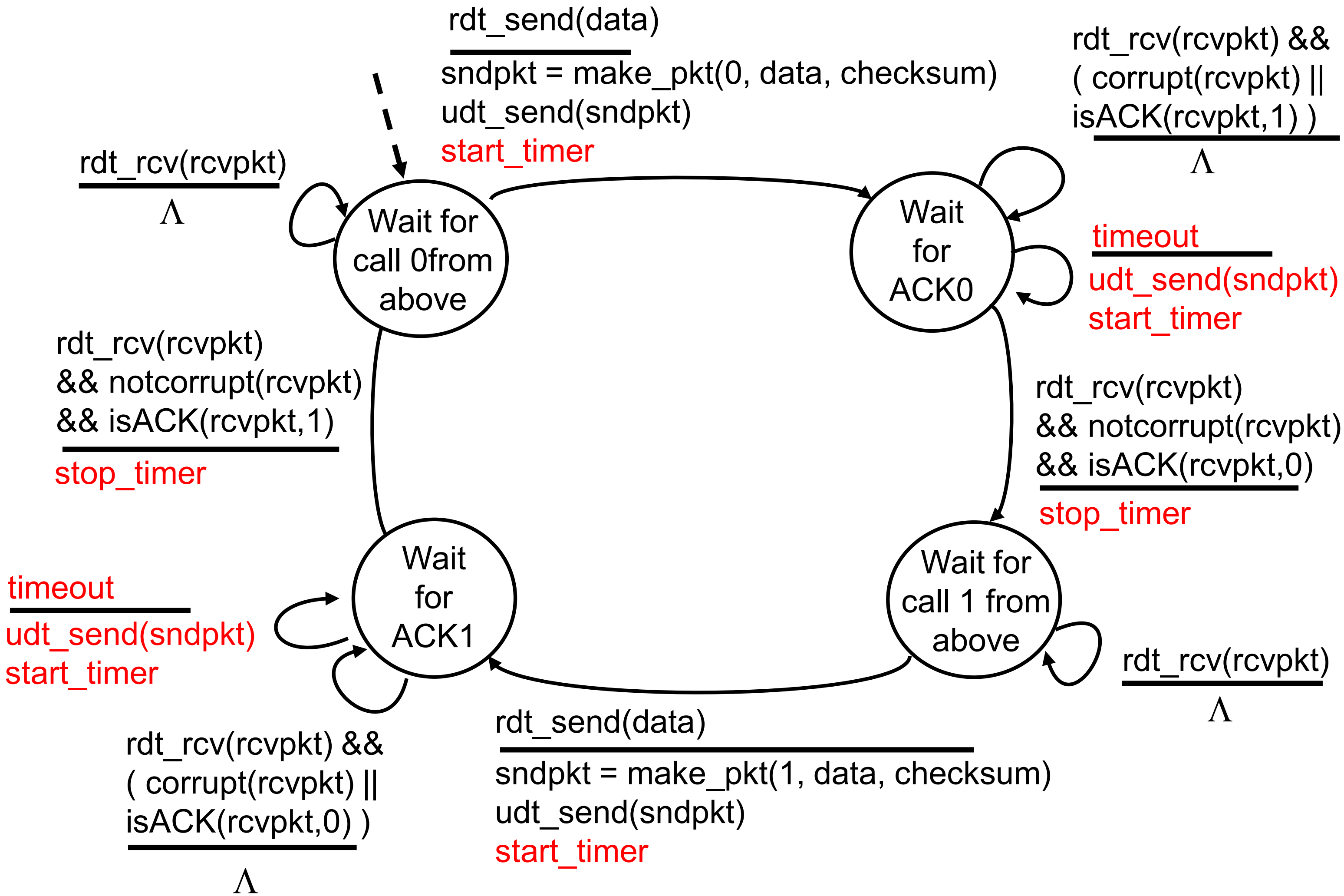
Scenario (loss?)



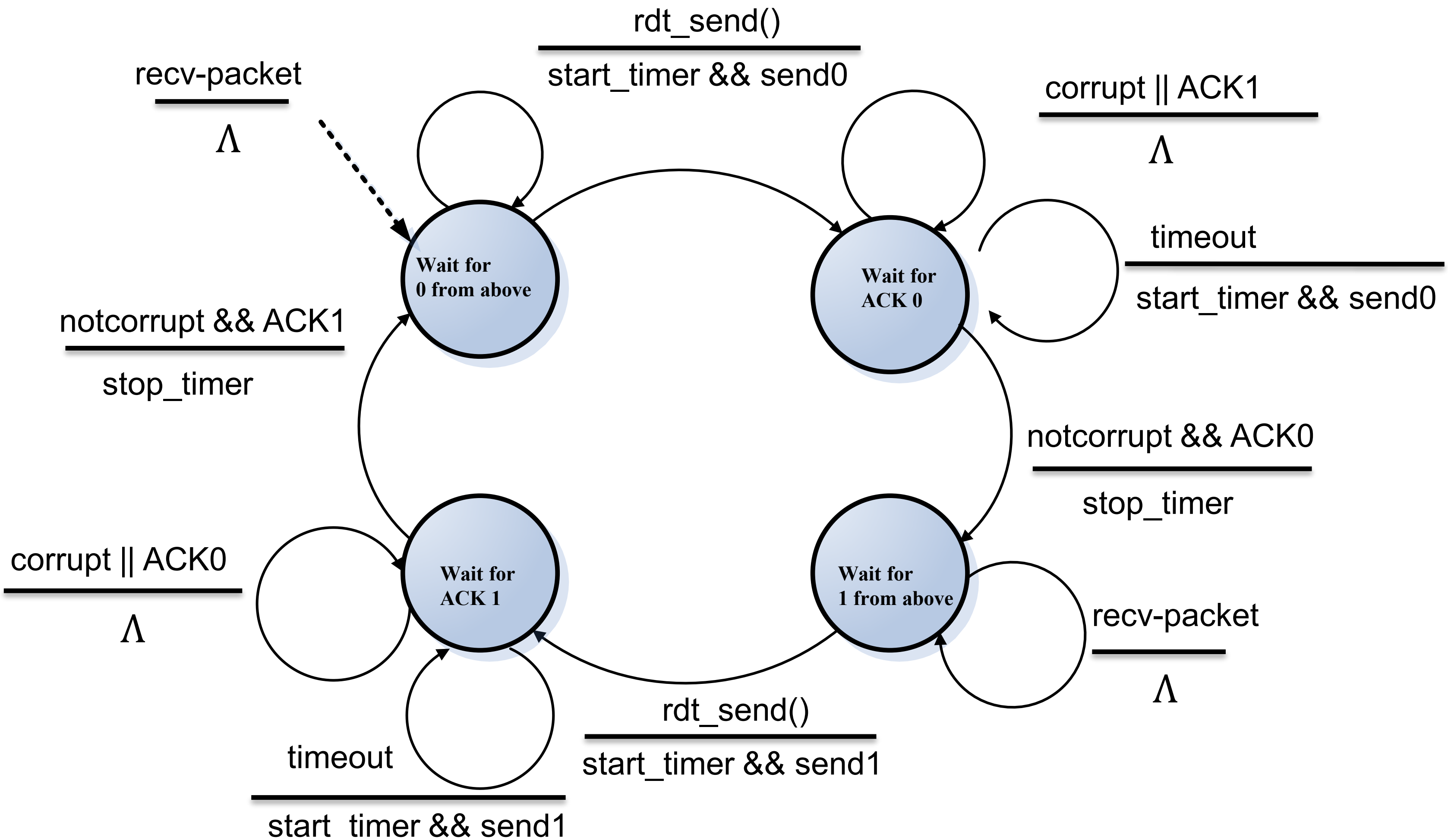
Sender rdt3.0



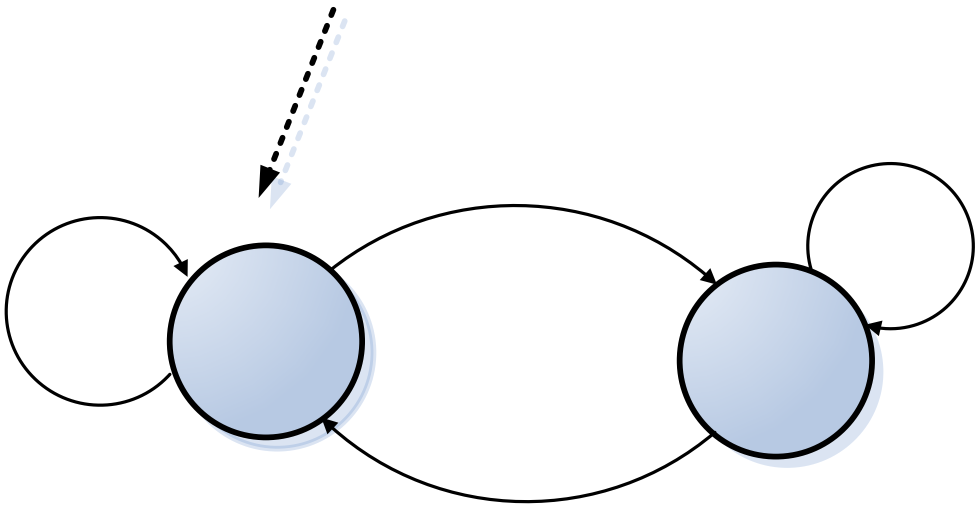
rdt3.0 sender



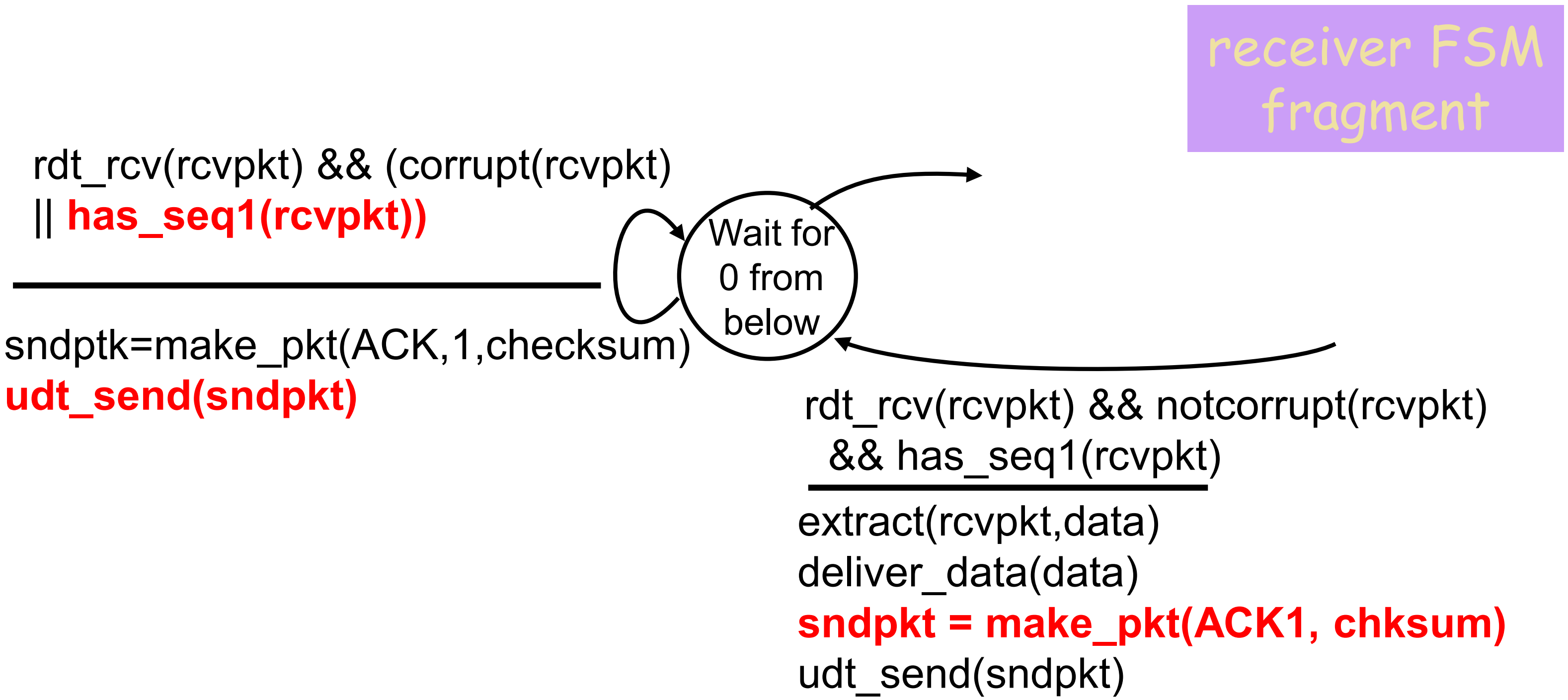
Simplified Sender rdt3.0



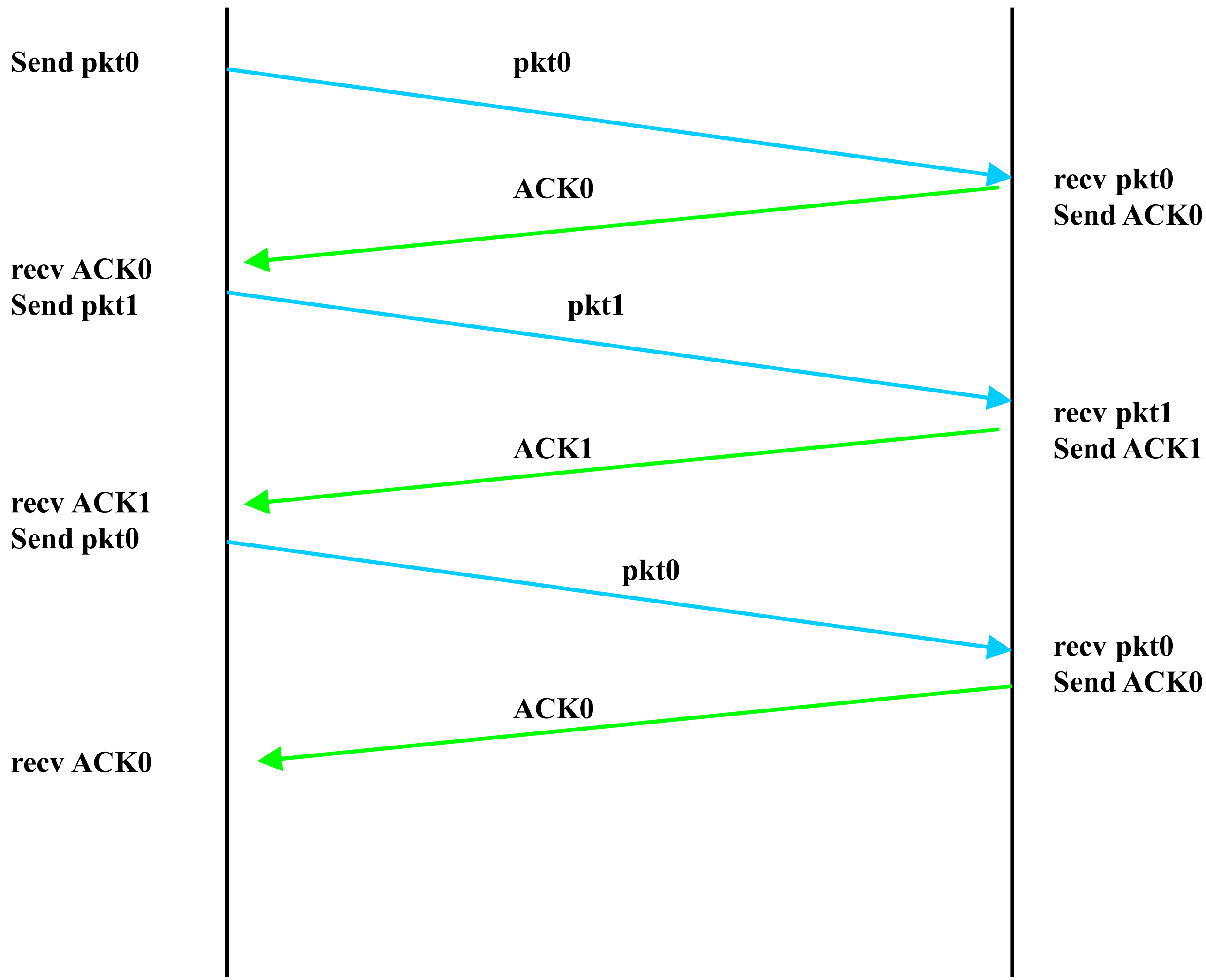
Receiver rdt3.0



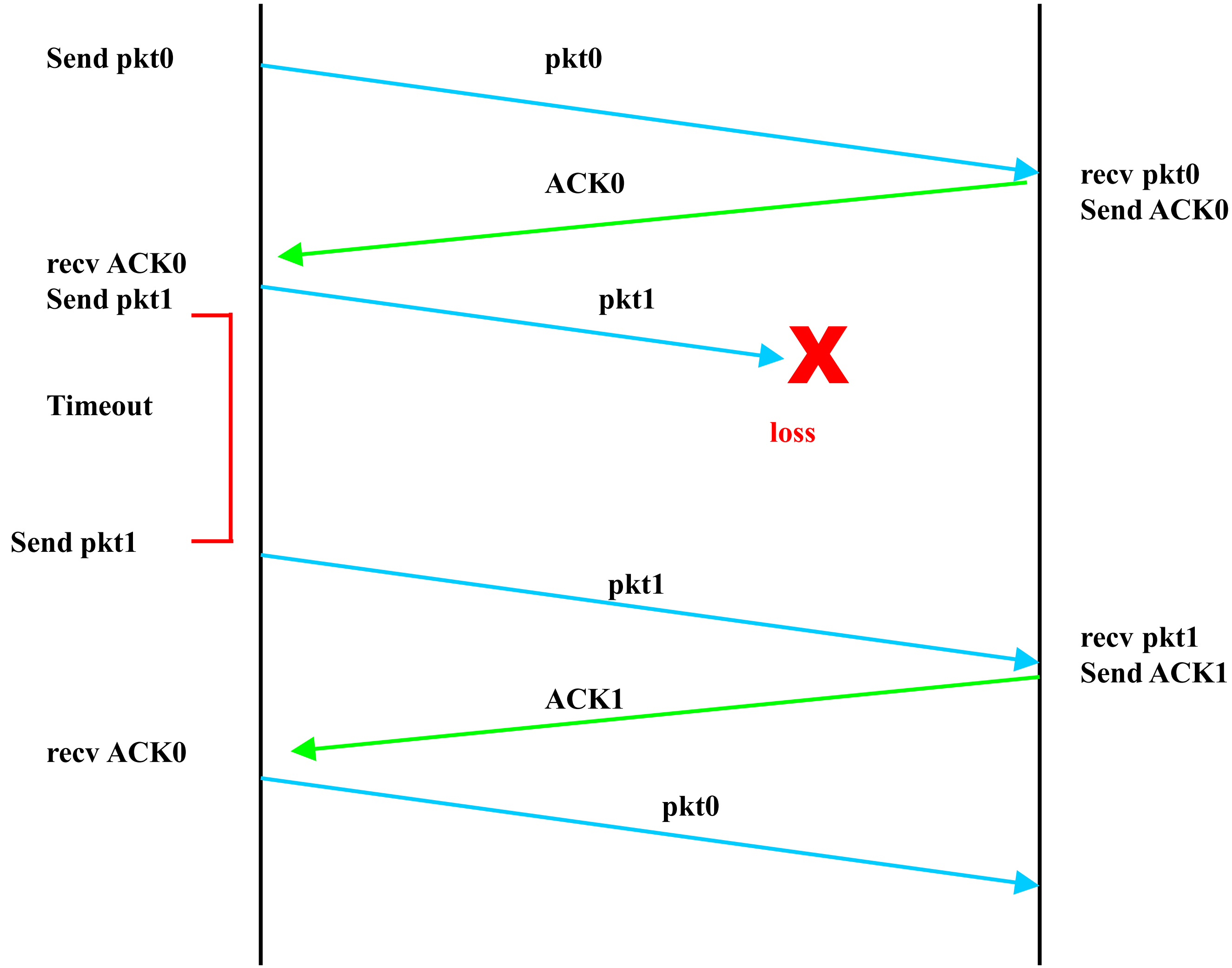
rdt3.0: receiver fragments



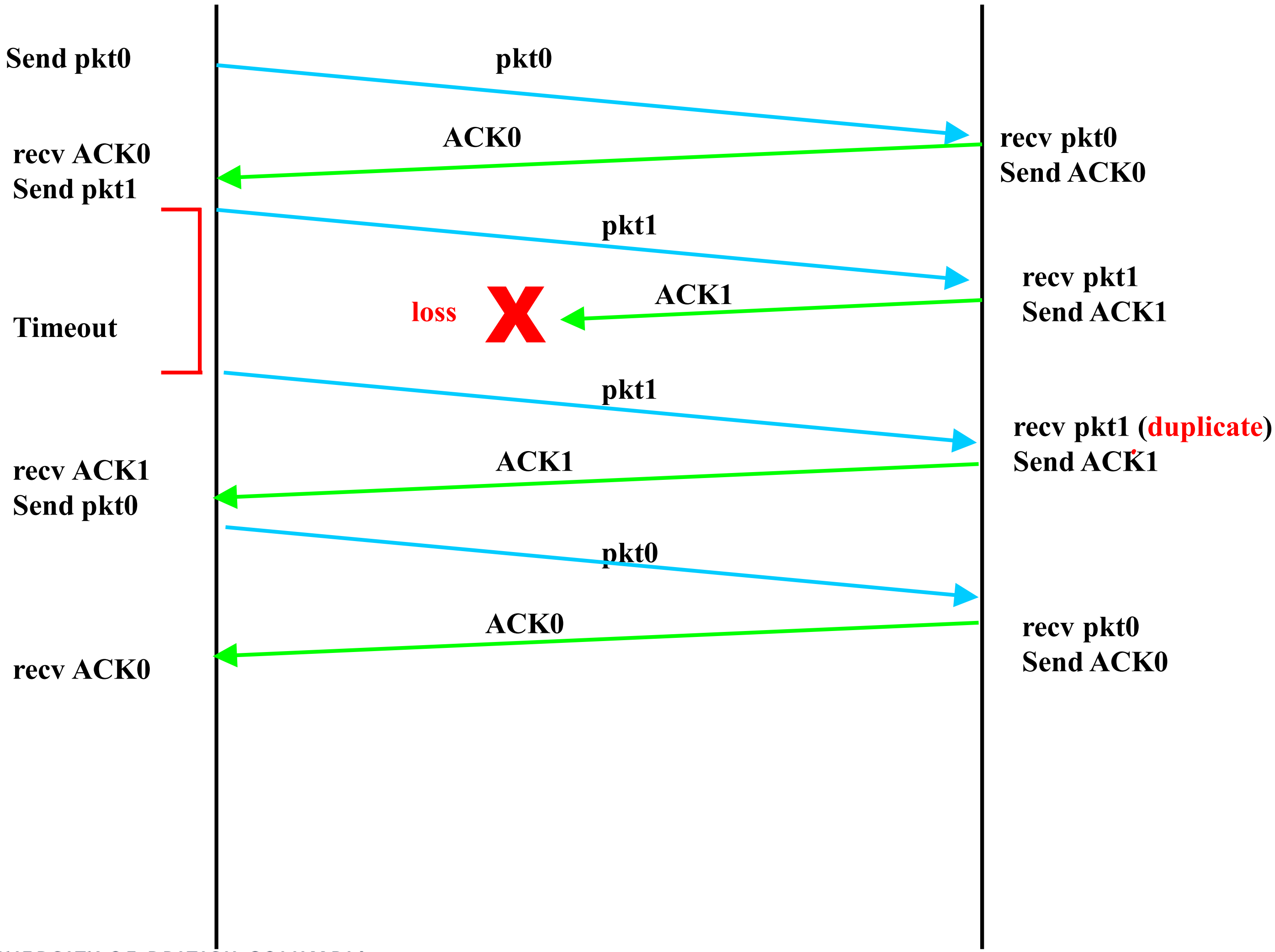
Normal Operation, no loss



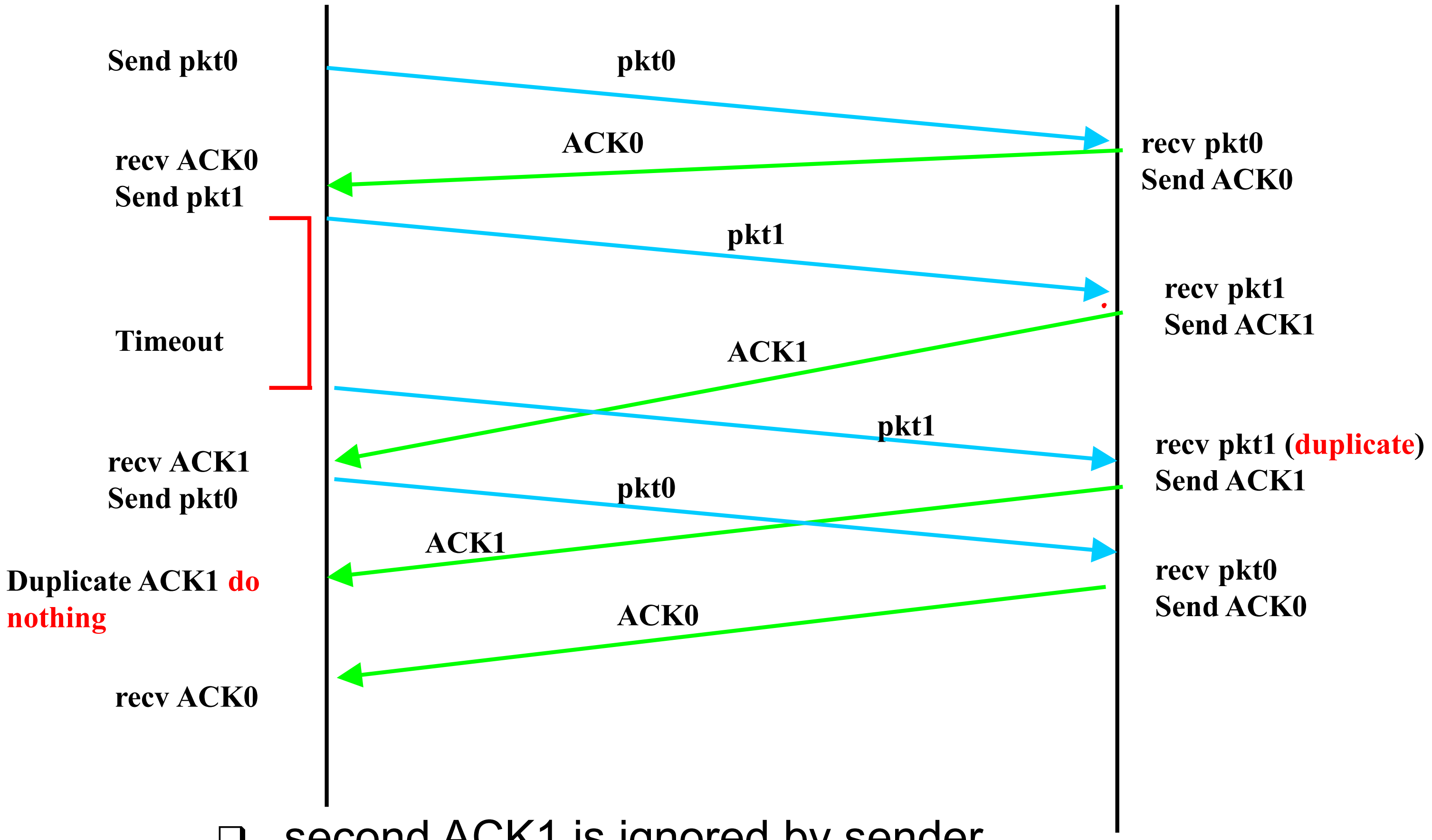
Lost Packet


Send pkt0
pkt0
recv pkt0
Send ACK0
ACK0
recv ACK0
Send pkt1
pkt1
loss
Timeout
Send pkt1
pkt1
recv pkt1
Send ACK1
ACK1
recv ACK0
pkt0

Duplicate Packet at Receiver



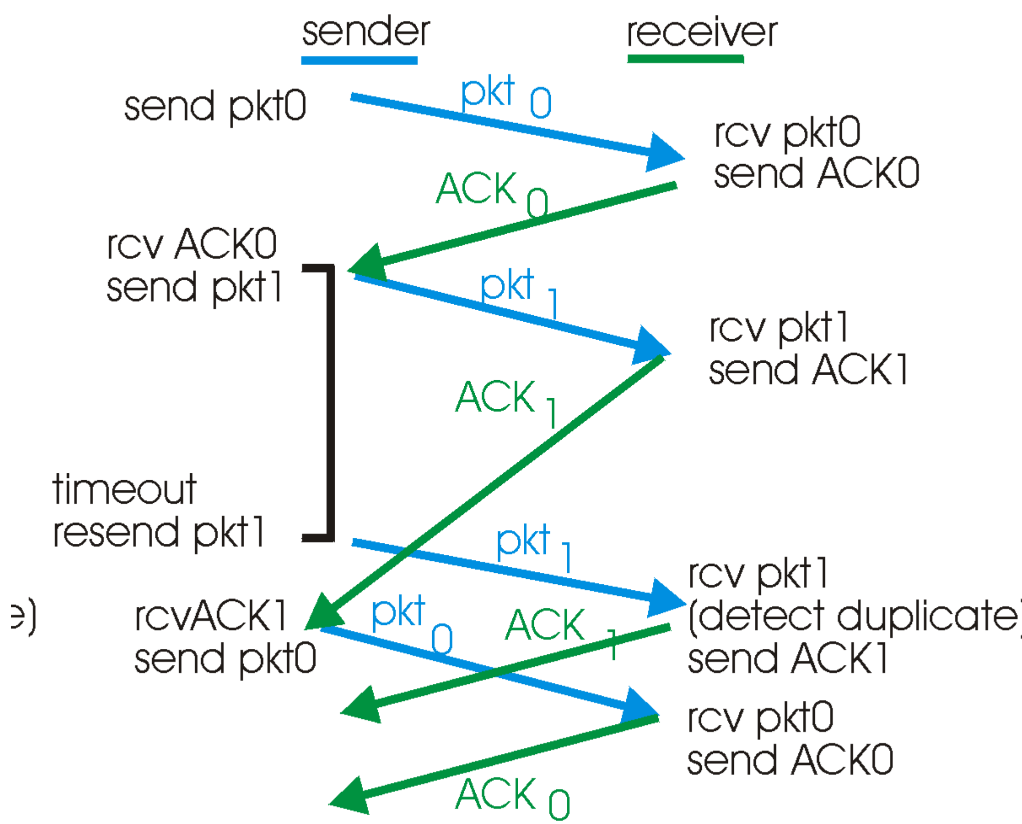
Duplicate Acknowlegement(s)



- second ACK1 is ignored by sender
- sender has sent pkt0; now expecting ACK0, ignores all else.

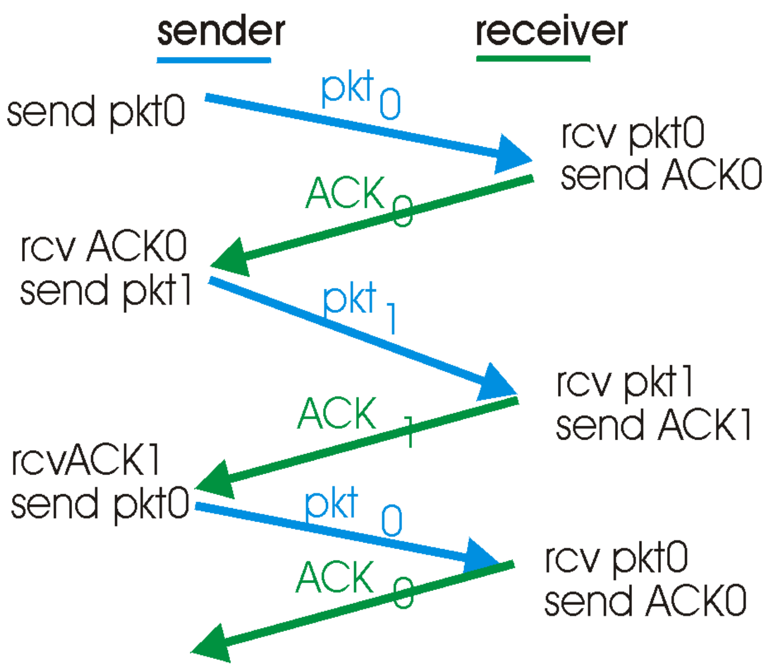
Clarify Time-out situation

- Second ACK1 is ignored.
- Sender has sent pkt0 so is now expecting a ACK0, ignores everything else.

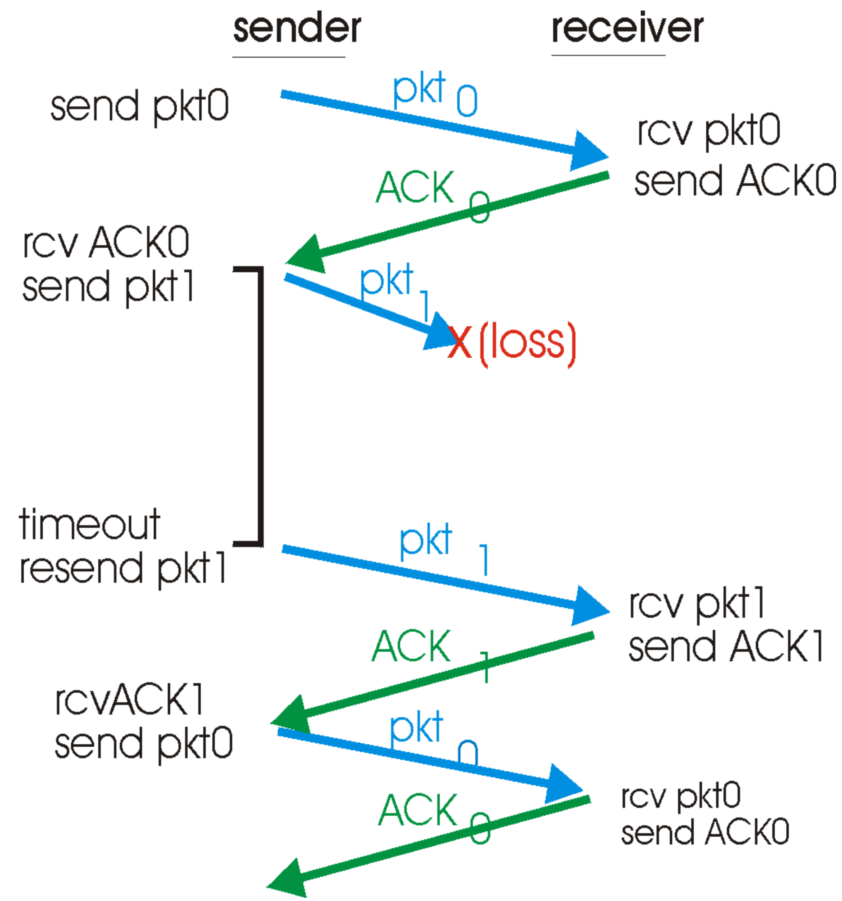


(d) premature timeout

rdt3.0 in action



(a) operation with no loss



(b) lost packet

Reliable Data Transfer Summary

- Acknowledgements (Negative ACKS)
- Re-transmissions
- Checksum (for detecting corrupt packets)
- Sequence Numbers
- Timer (needed when there is loss)

- No solution for OUT-OF-ORDER

PERFORMANCE STOP&WAIT

Performance of rdt3.0

- rdt3.0 works, but performance is TERRIBLE
- example: 1 Gbps link, 15 millisecond propagation delay, 8000 bit packet:

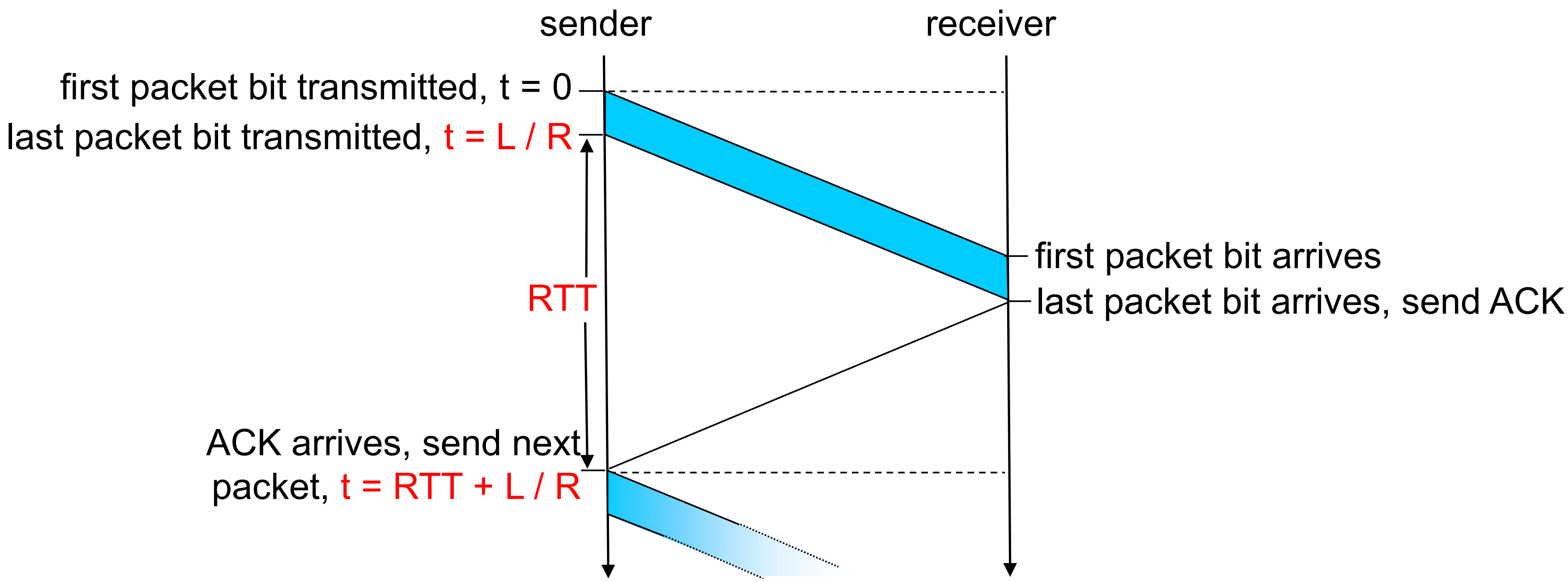
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\ \text{bps}} = 8\ \text{microseconds}$$

- U_{sender}: utilization – fraction of time sender busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00026$$

- 1 pkt every 30 msec -> 0.26 Mbps throughput over 1 Gbps link
- network protocol limits use of physical resources!

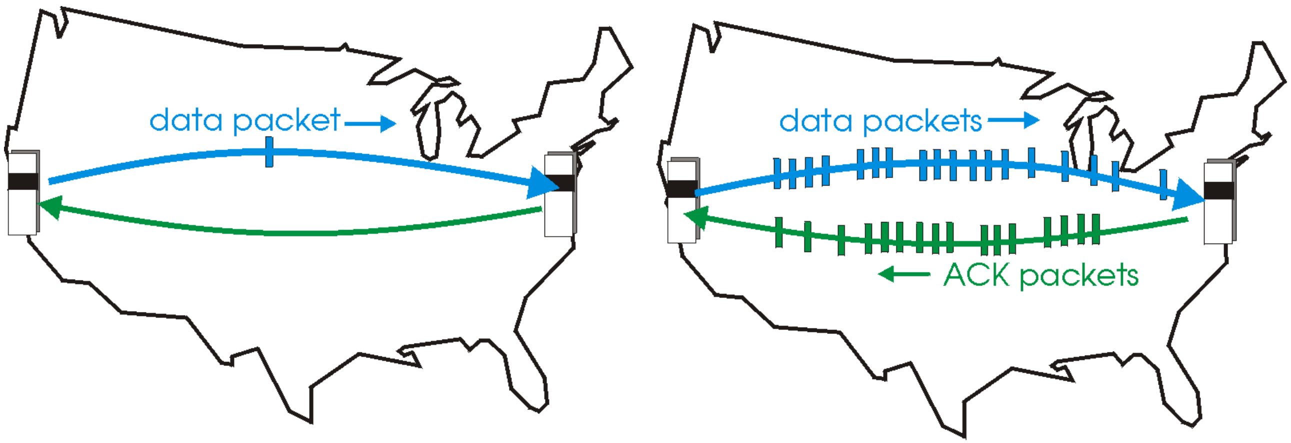
rdt3.0: stop-and-wait operation



Pipelined protocols

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

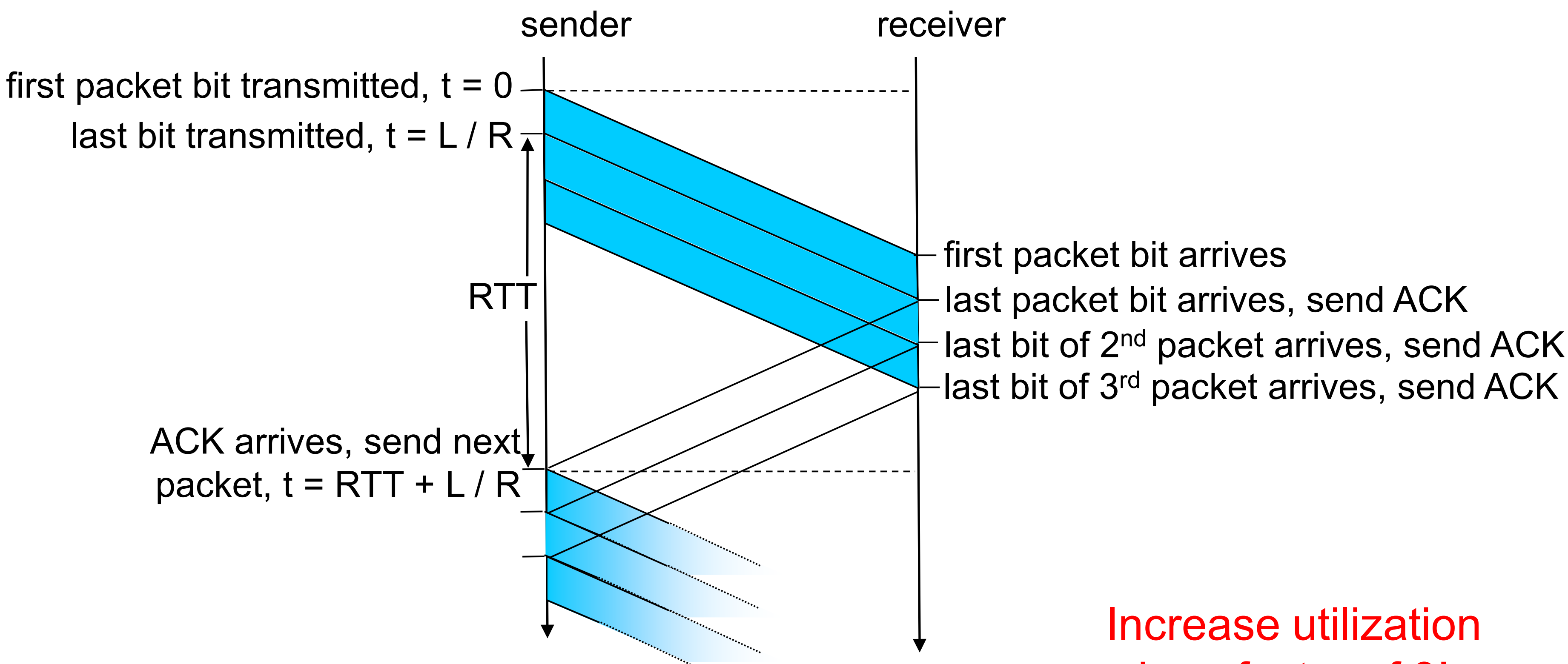
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

- Two generic forms of pipelined protocols: *various TCP ones, go-Back-N, selective repeat*

Pipelining: increased utilization



Increase utilization by a factor of 3!

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

SLIDING WINDOW

SLIDING WINDOW

https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html

https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html

http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

SLIDING WINDOW

- Developed ARQ method called
  - Alternating Bit Protocol or
  - Stop and Wait

- Link utilization (throughput) is low and solution was pipelining (more packets in flight)

ARQ
(automatic repeat request)

- ABP (S&W) (Alternating Bit Protocol, Stop and Wait)
- SWP (Sliding window protocols)
  - GBN, go-back-N
  - SR, selective repeat

Sliding Window in Action

Terminology

Sender side:

- **SWS:** send window size
- **LAR**: last ACK received
- **LFS:** last frame sent

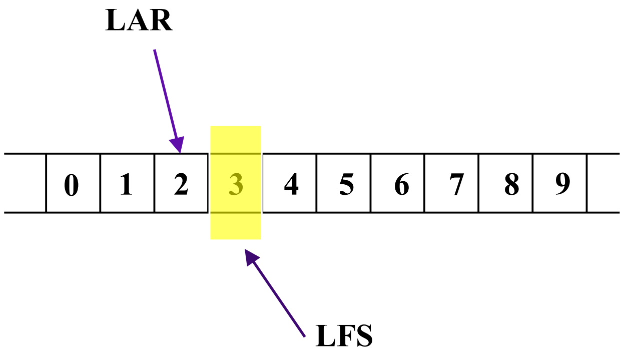
Receiver side:

- **LFR:** last frame received
- **LAF**: largest acceptable frame

	0	1	2	3	4	5	6	7	8	

	0	1	2	3	4	5	6	7	8	

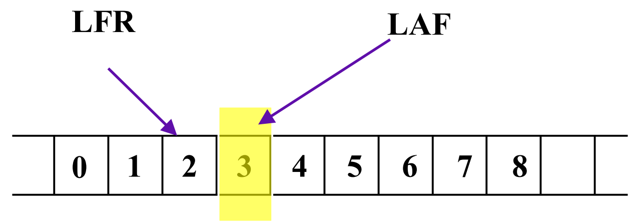
Stop and Wait



Sender side:

LAR: last ACK received

LFS: last frame sent

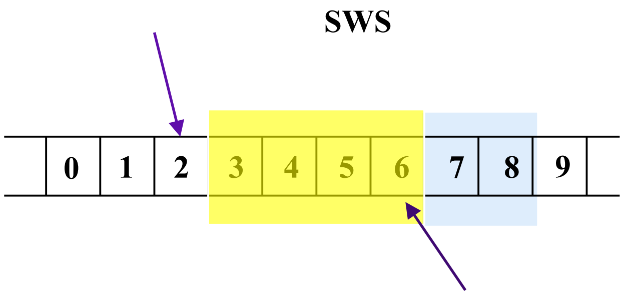


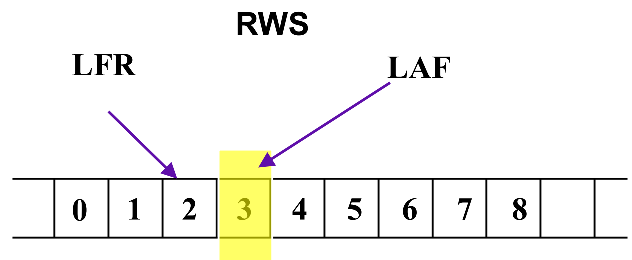
Receiver side:

LFR: last frame received

LAF: largest acceptable frame

Sliding Window





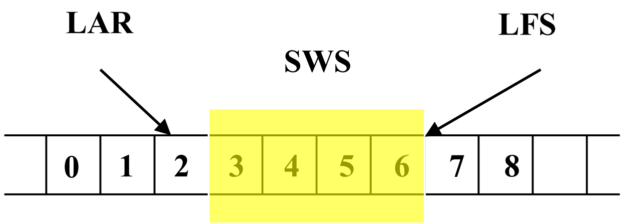
Sender side:

SWS: send window size
LAR: last ACK received
LFS: last frame sent

Receiver side:

RWS: receive window size
LFR: last frame received
LAF: largest acceptable frame

Sender



Sender side:

- **SWS:** send window size
- **LAR**: last ACK received
- **LFS:** last frame sent

Sender:

if more data to send (LFS-LAR < SWS)

then send data, LFS++

if recv'ed ACK for LAR+1

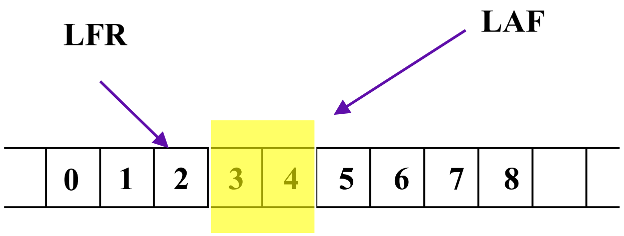
then LAR++

if timer expires

then send [3 or 3-4-5-6]

Two strategies:
(a) Go-Back-N
(b) Selective Repeat

Receiver



Receiver side:

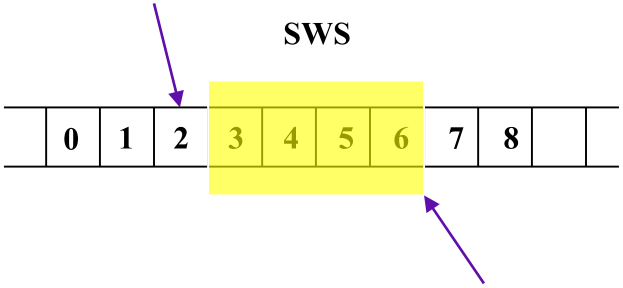
LFR: last frame received

LAF: largest acceptable frame

Receiver:
  if recv'ed K > LAF
    then discard
    else
      if K == LFR+1 then
        store
        LFR++, LAF++ (slide window)
      else store [or discard]
  ACK, largest in-order received frame

Two strategies:
(a) Go-Back-N
(b) Selective Repeat

Sliding Window

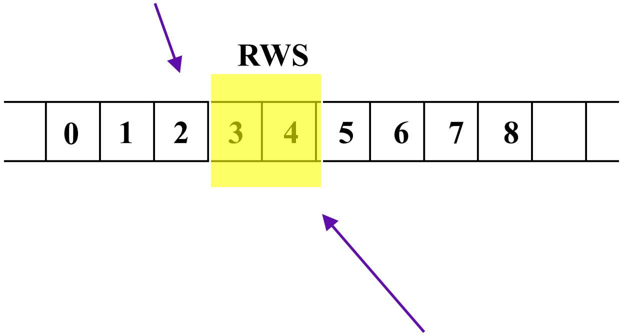


Sender:

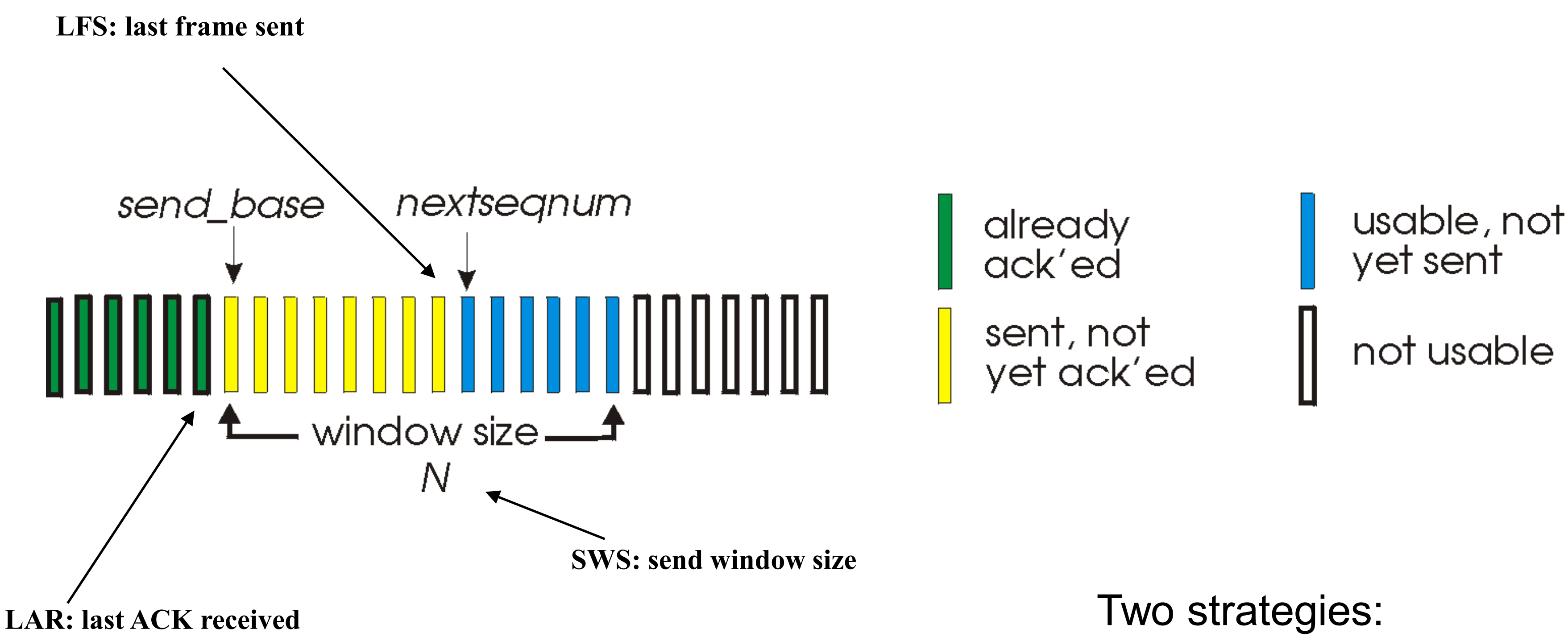
if more data to send (LFS-LAR < SWS)
then send data, LFS++
if recv'ed ACK for LAR+1
then LAR++
if timer expires
then send LAR+1

Receiver:

if recv'ed K > LAF
then discard
else
if K == LFR+1 then
store
LFR++, LAF++ (slide window)
else
discard
ACK, largest in-order received frame



Book's Terminology

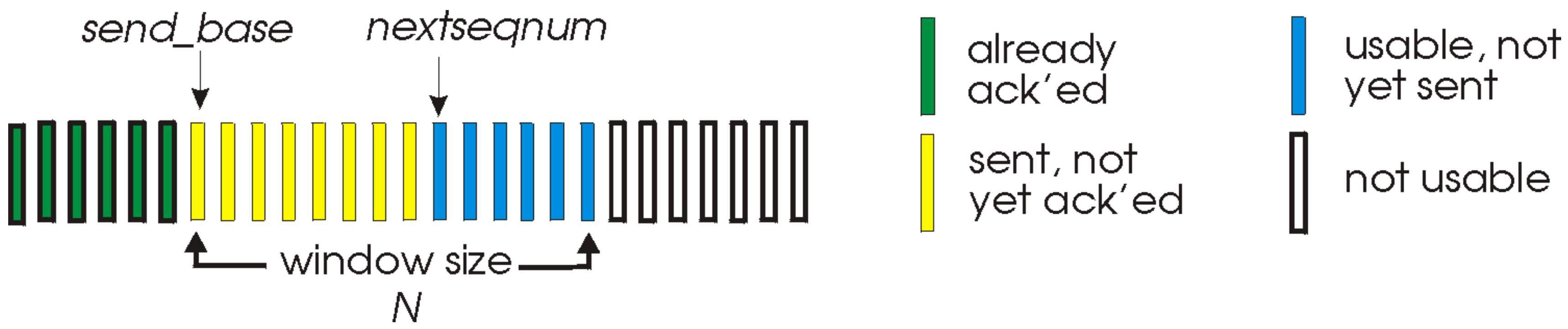


Two strategies:
(a) Go-Back-N
(b) Selective Repeat

Go-Back-N

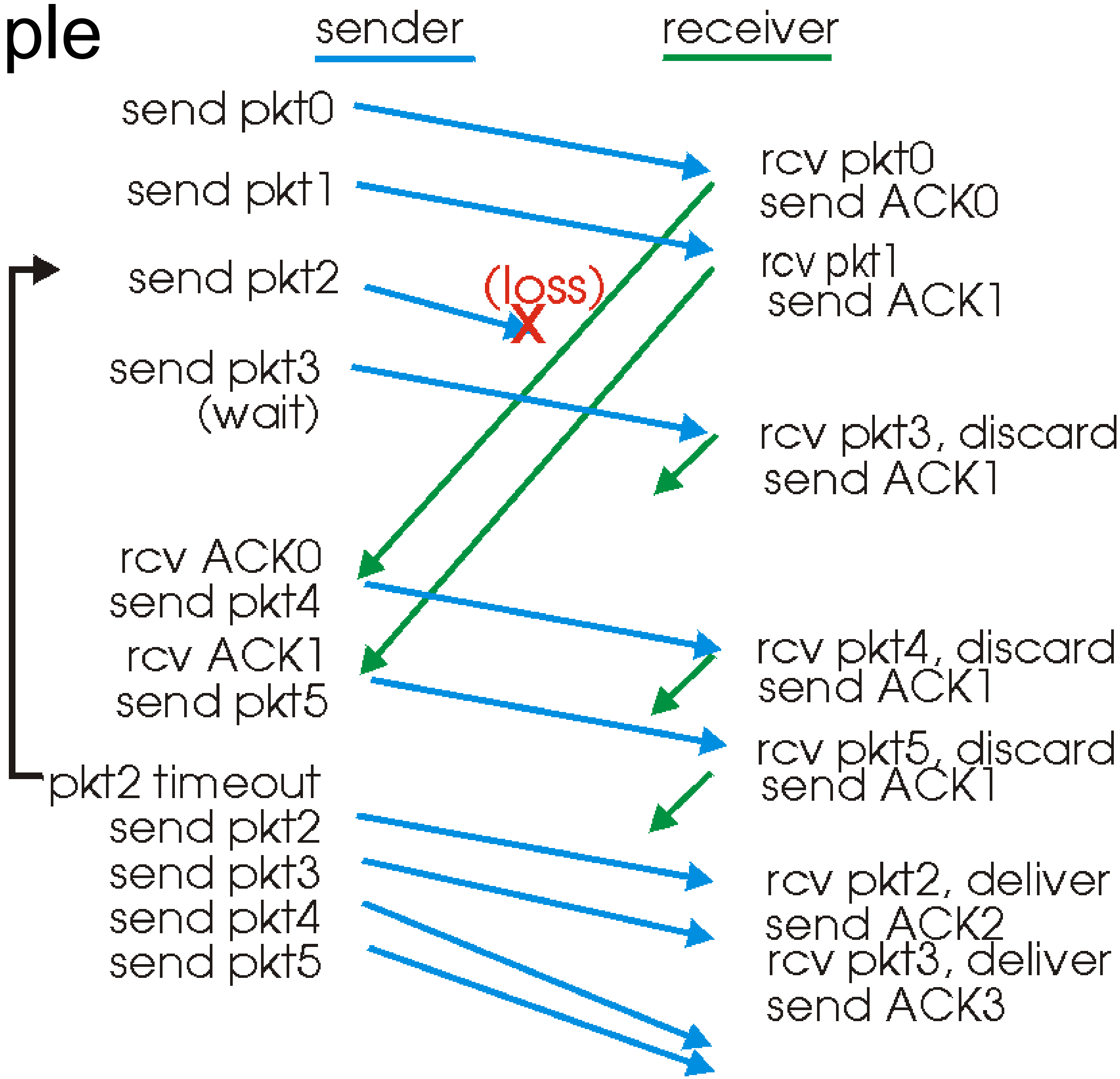
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer only for smallest sequence number sent but not ack'ed
- *timeout(n):* re-transmit pkt n and all higher seq # pkts in window

GBN Example



sender
receiver
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
(loss)
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv ACK0
send pkt4
rcv ACK1
send pkt5
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

Selective Repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective repeat

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [send-window]:

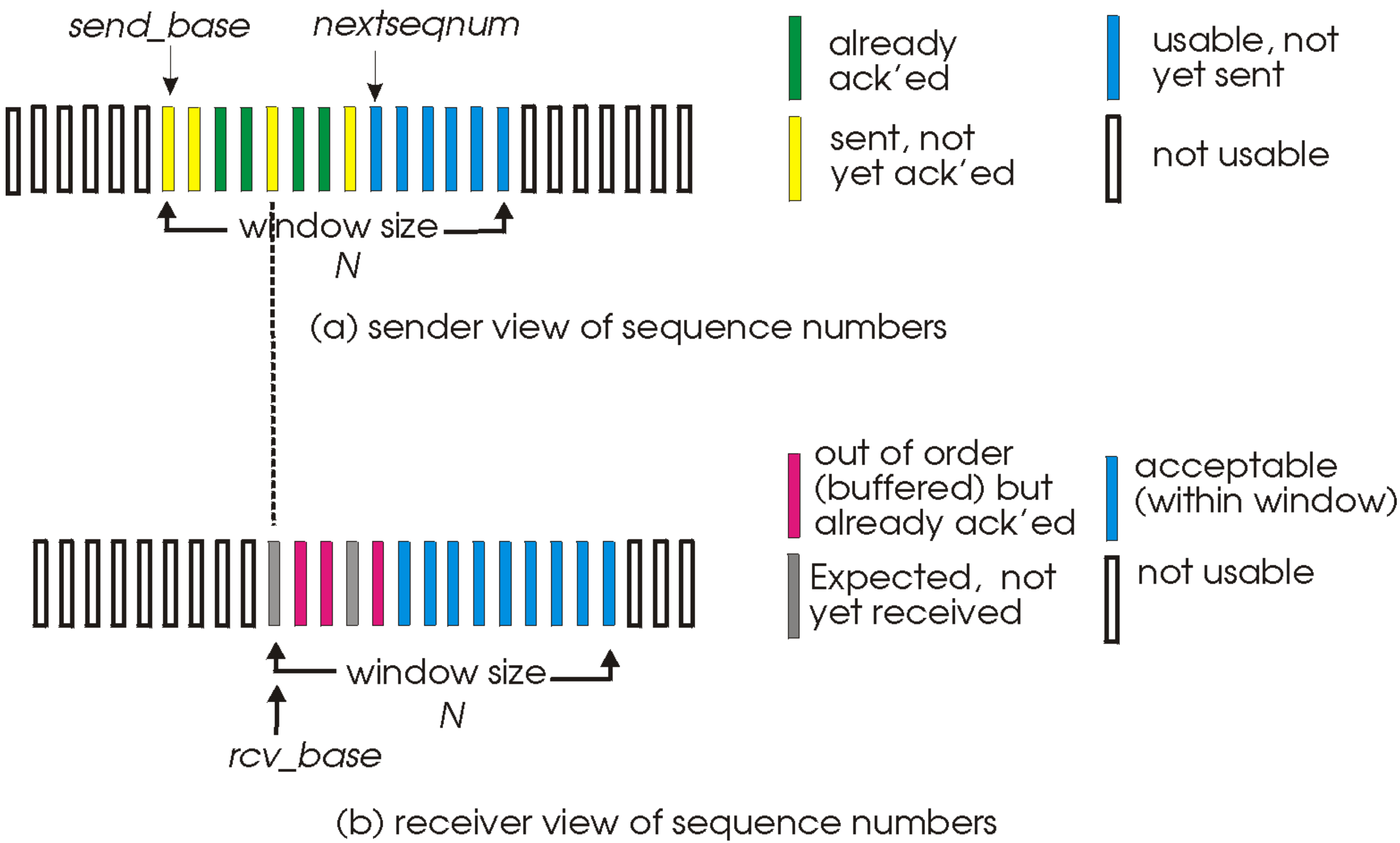
- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

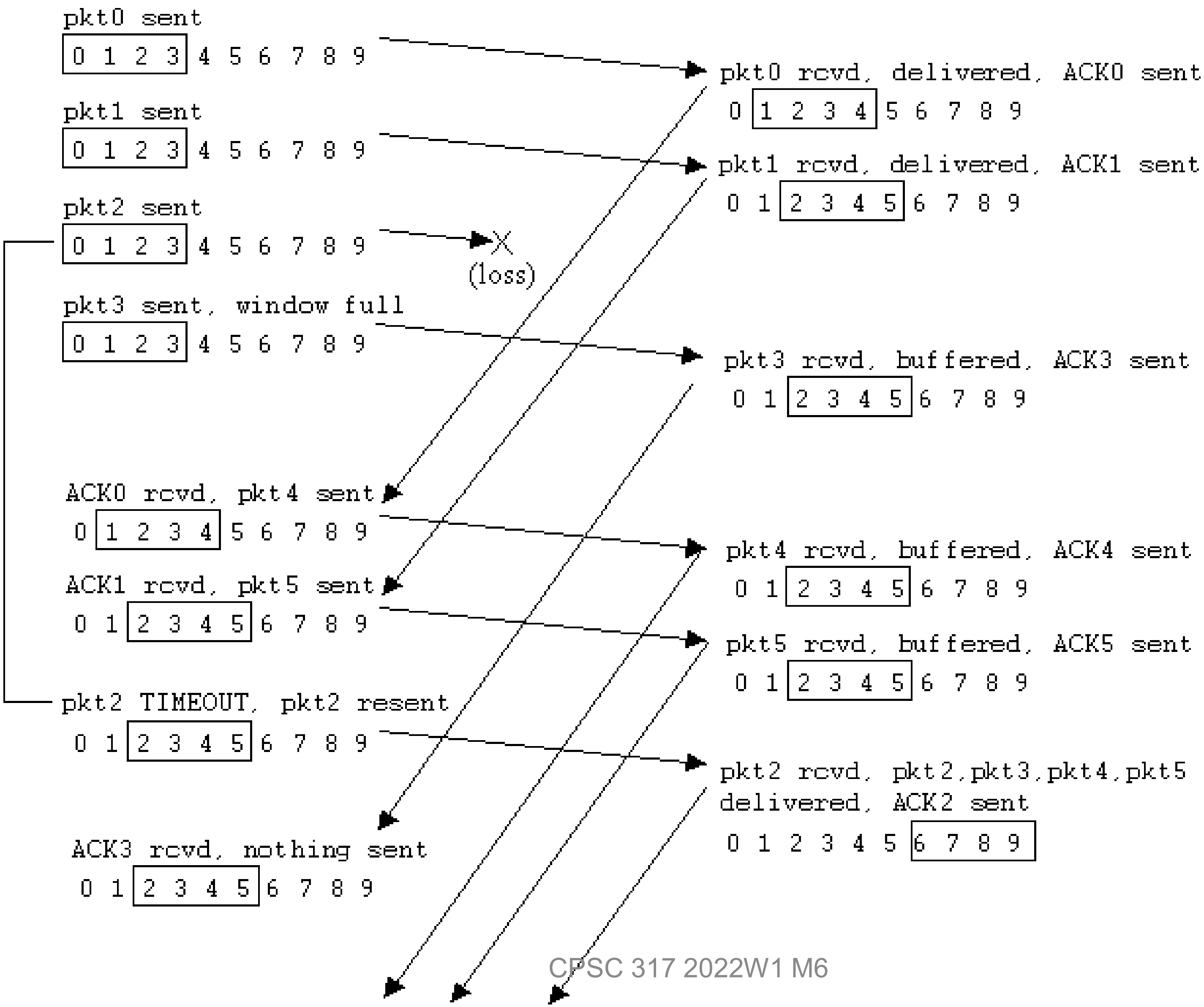
pkt n in [recv-window]

- 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: sender, receiver windows



Selective repeat in action



SEQUENCE NUMBERS

Sequence Number Range

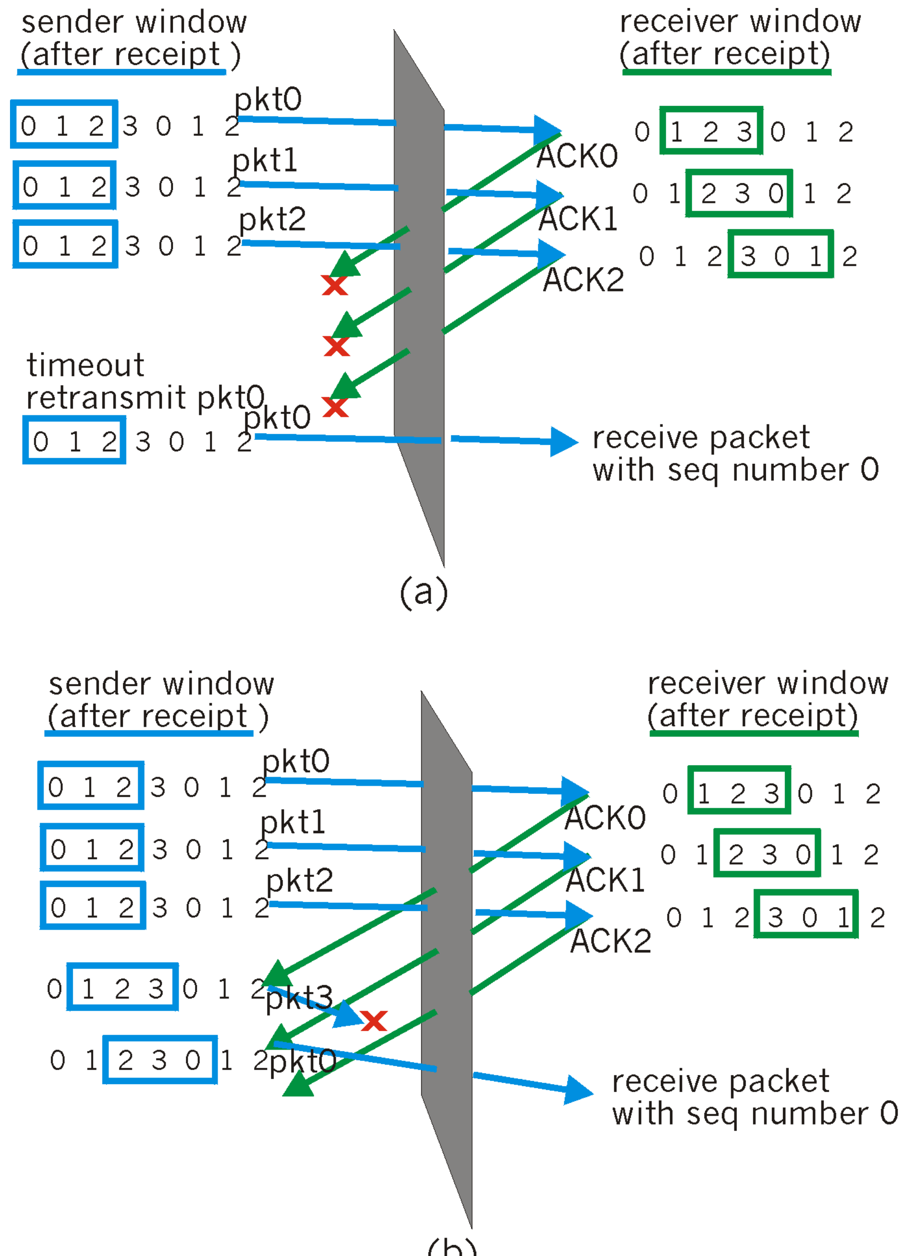
- Must fit into K bits
- Finite
- Is there a limit on the ranges that work?
  - SWS = N, RWS = 1
  - SWS = N, RWS = N

- Does it make sense for RWS>SWS?

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)



Sequence Number Range

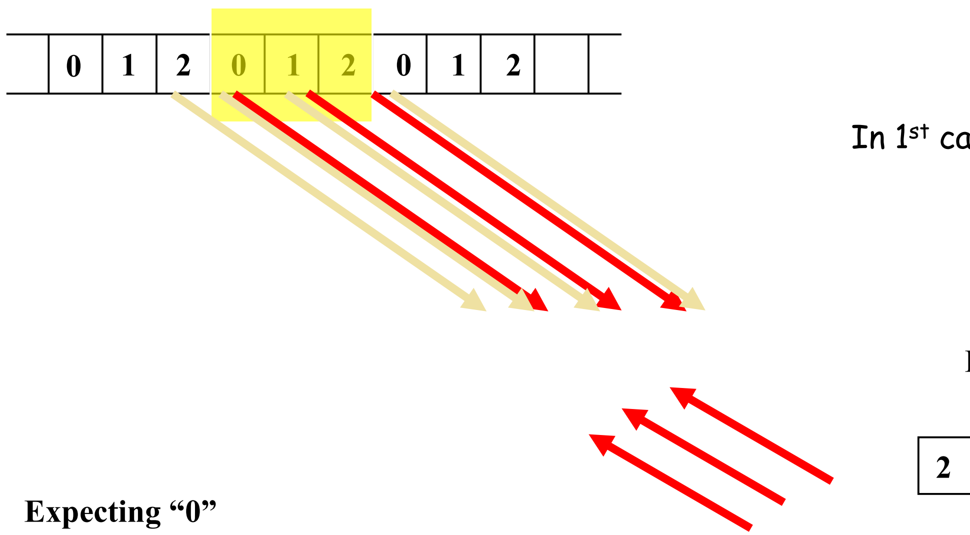
- Must fit into K bits
- Finite
- What is the relationship between RWS, SWS and the number of sequence numbers?

# of sequence numbers >= SWS + RWS

GBN Sequence Space Example

3 sequence numbers
SWS = 3
RWS = 1

SWS



In 1st case, receiver gets the original zero, fine.

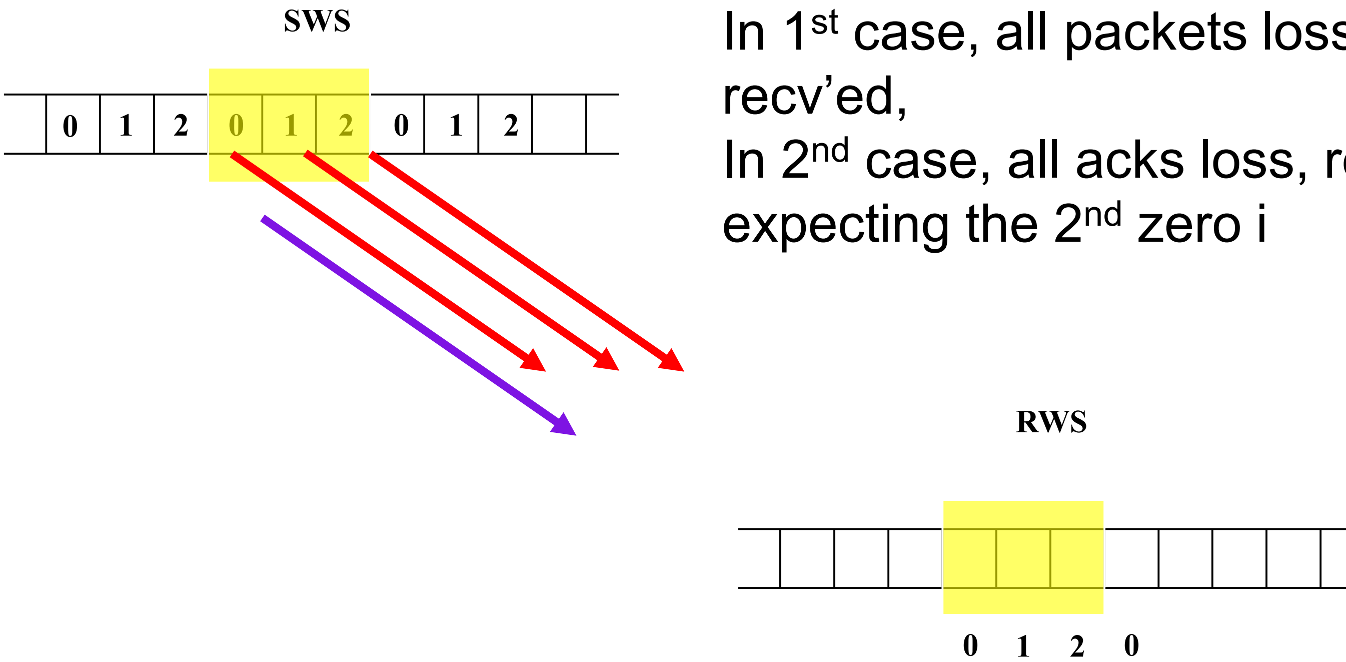
RWS

Expecting “0”

In 2nd case, receiver gets 0, 1, and 2, and ACKS them all, slides and is now expecting the next zero, WRONG

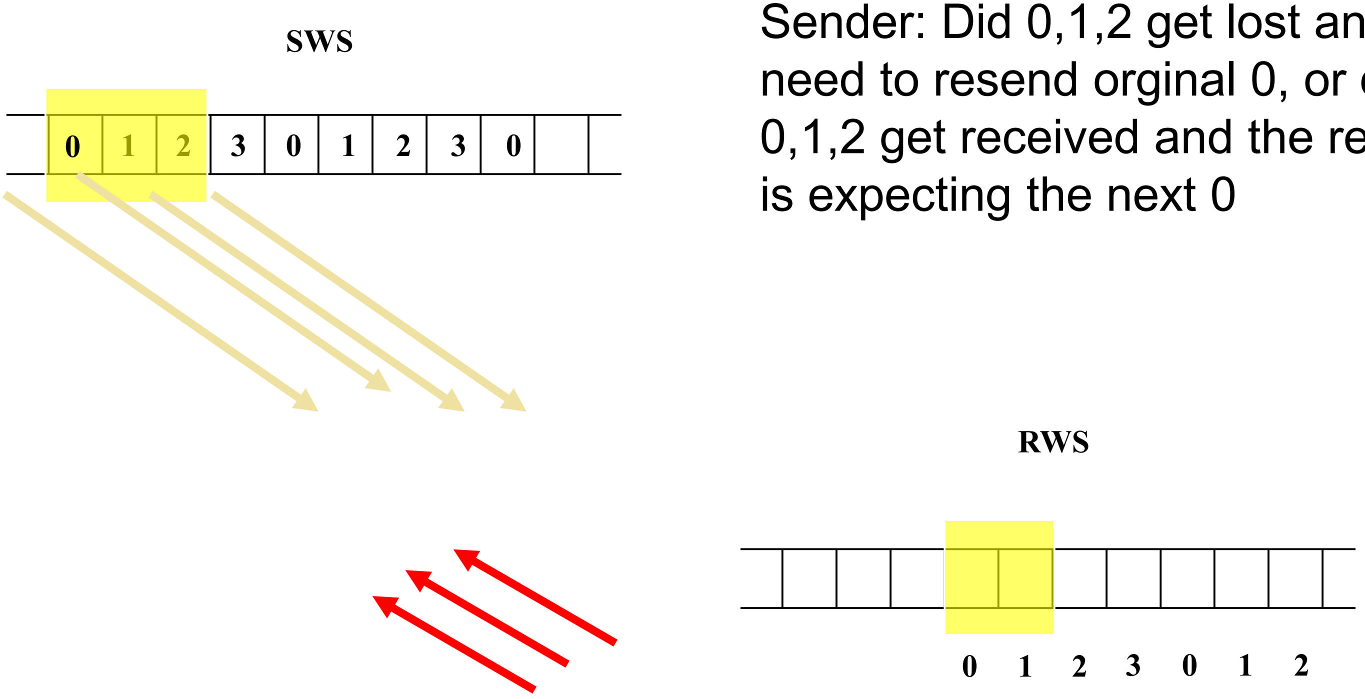
Sequence space must be at least SWS+1

SR Sequence Space Example



In 1st case, all packets loss, 1st zero is recv'ed,
In 2nd case, all acks loss, receiver is expecting the 2nd zero i

SR Example



Sender: Did 0,1,2 get lost and I need to resend orginal 0, or did 0,1,2 get received and the receiver is expecting the next 0

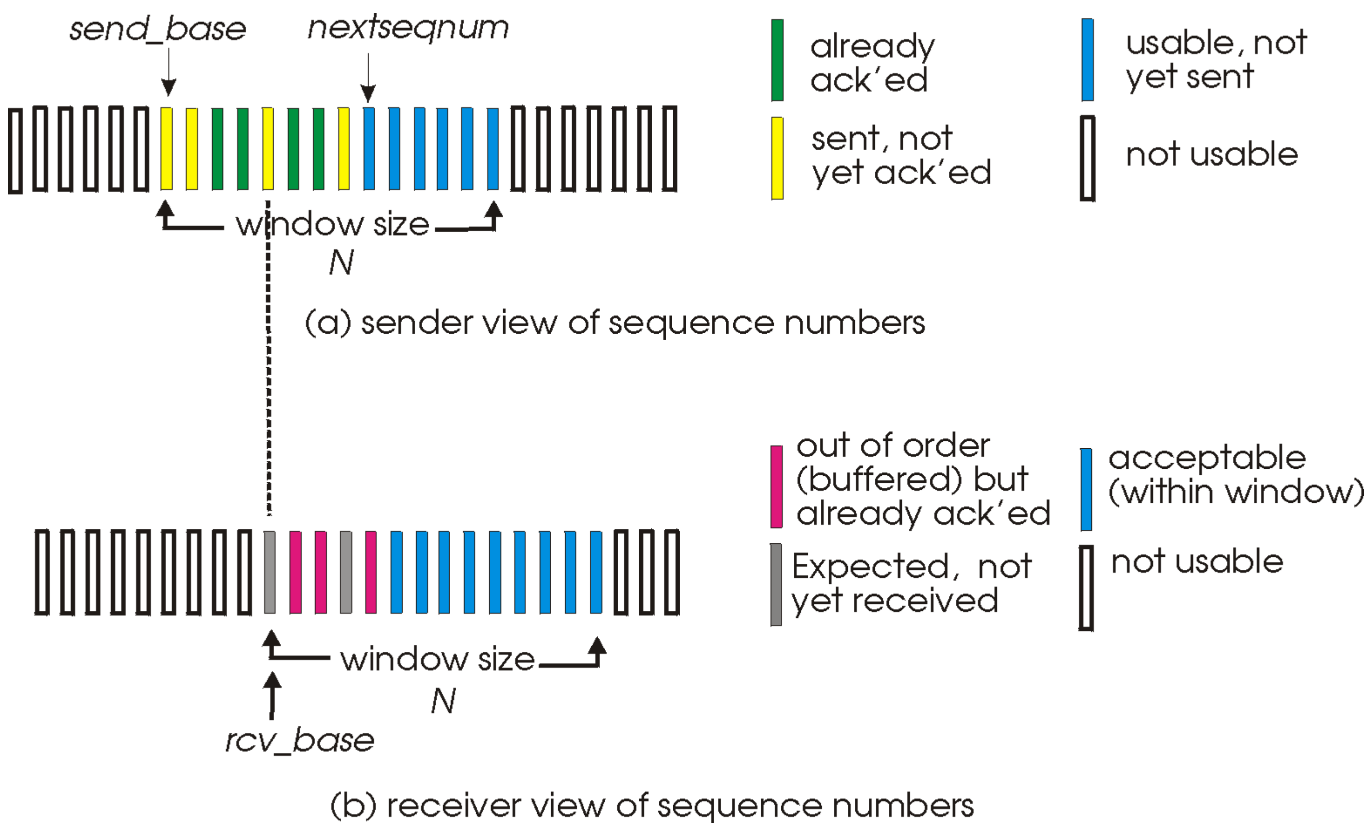
Sequence space SWS + RWS

SEQUENCE NUMBERS (sliding window)

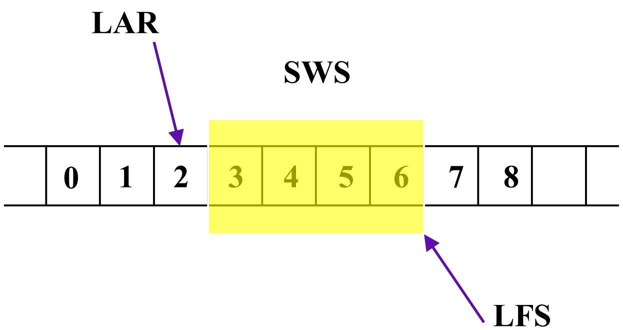
What do we ACK?

- The packet we just received (Kurose and Ross rdt3.0, and earlier ones)
- Sliding window (Kurose and Ross)
  - ACK the packet we just received
  - Cumulative ACK, ACK the largest in-order received packet
- Same as above but ACKing next expected packet rather than the one received. (TCP)

Selective repeat (vs GBN, vs CUMULATIVE)



Sliding Window (TCP like)



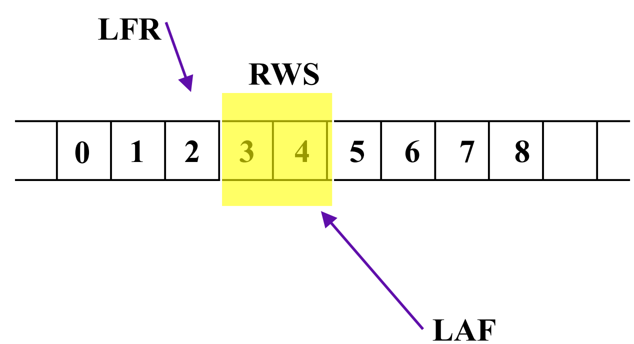
Sender:

- if more data to send (LFS-LAR < SWS)
  - then send data, LFS++
- if recv'ed ACK for LAR+1
  - then LAR++
- if timer expires
  - then send LAR+1

LAR: last ACK received
LFS: last frame sent

Receiver:

- if recv'ed K > LAF
  - then discard
  - else
    - if K == LFR+1
    - then
      - store
      - LFR++, LAF++ (slide window)
- ACK, LFR (after it was incremented)



LFR: last frame received
LAF: largest acceptable frame

Sequence Numbers Cases

TRUE # of sequence numbers >= SWS + RWS

- RWS > SWS? FALSE: RWS >= SWS
- SWS > # of sequence numbers FALSE: duplication
- SWS=RWS=1 STOP AND WAIT
- SWS=N, RWS=1 GBN
- SWS=RWS=N Selective Repeat

Summary

RDT:

- Added retransmit, checksum, sequence numbers, acknowledgments, and timers.
- Pipelined, needed to introduce sliding windows and more sequence number space.
- Still cannot handle out of order packets (i.e. packet A leaves before packet B, but packet B arrives before A, or to say that an earlier packet in transit can arrive later than a packet sent after the earlier packet)

STILL A PROBLEM

Strategies for Sliding Window

Go-back-N

Selective Repeat

Slight variations of the above, cumulative ACK or next packet instead of last one.

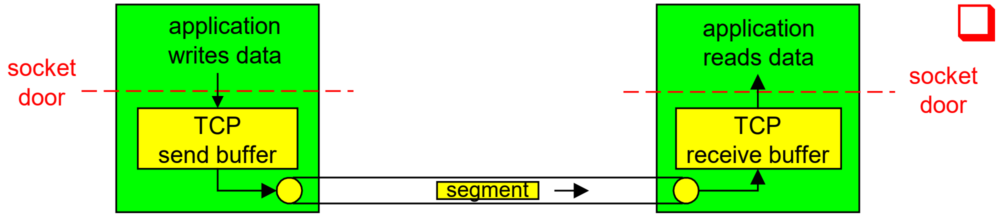
Sliding window makes it possible to improve throughput and a mechanism for flow control.

TCP

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries”
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*

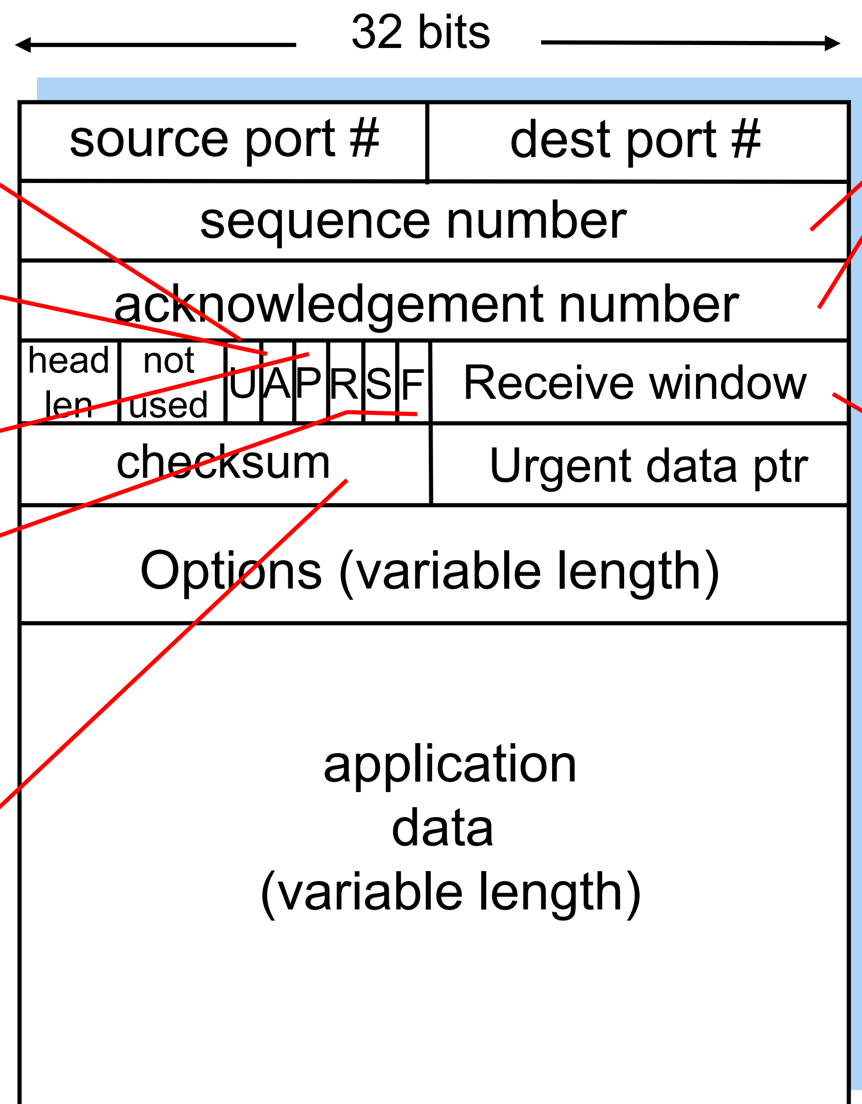


- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init’s sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP

- What's in the header?
- Sliding window
- RTT estimation
- More on sliding window
- Flow control
- Connection management
- Congestion

TCP segment structure



URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection estab (setup, teardown commands)

Internet checksum (as in UDP)

counting by bytes of data (not segments!)

bytes rcvr willing to accept

RST: reset, drop link
SYN: send flag
FIN: shut down link gracefully

TCP

- What's in the header?
- Sliding window
- RTT estimation
- More on sliding window
- Flow control
- Connection management
- Congestion

TCP Sliding Window

- Cumulative acknowledgements
- Store out of order frames that are within the size of the receive window
- ACK next expected byte
- Sequence number is of the first byte in segment
- Variations of TCP: TCP-vegas, TCP-reno, TCP-sack

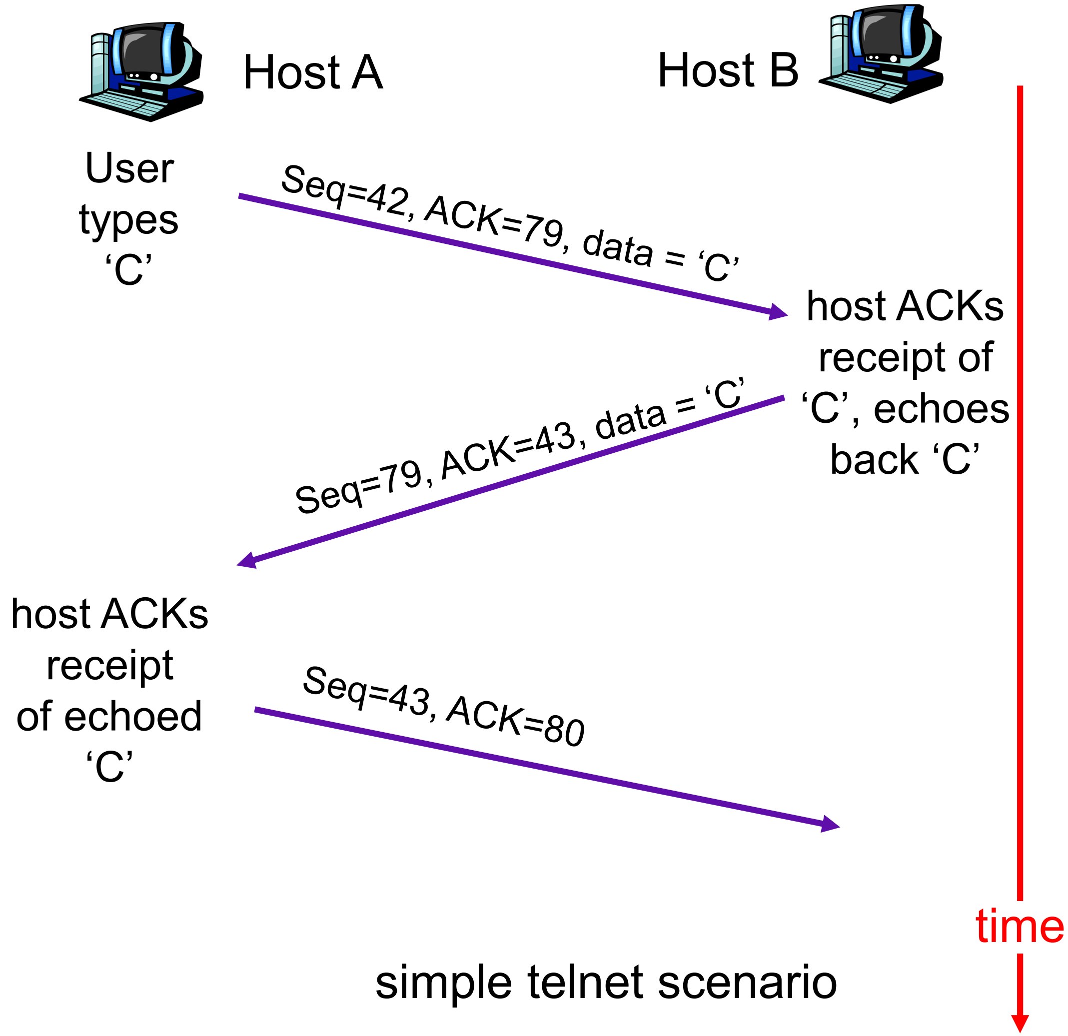
TCP seq. #'s and ACKs

Seq. #'s:

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

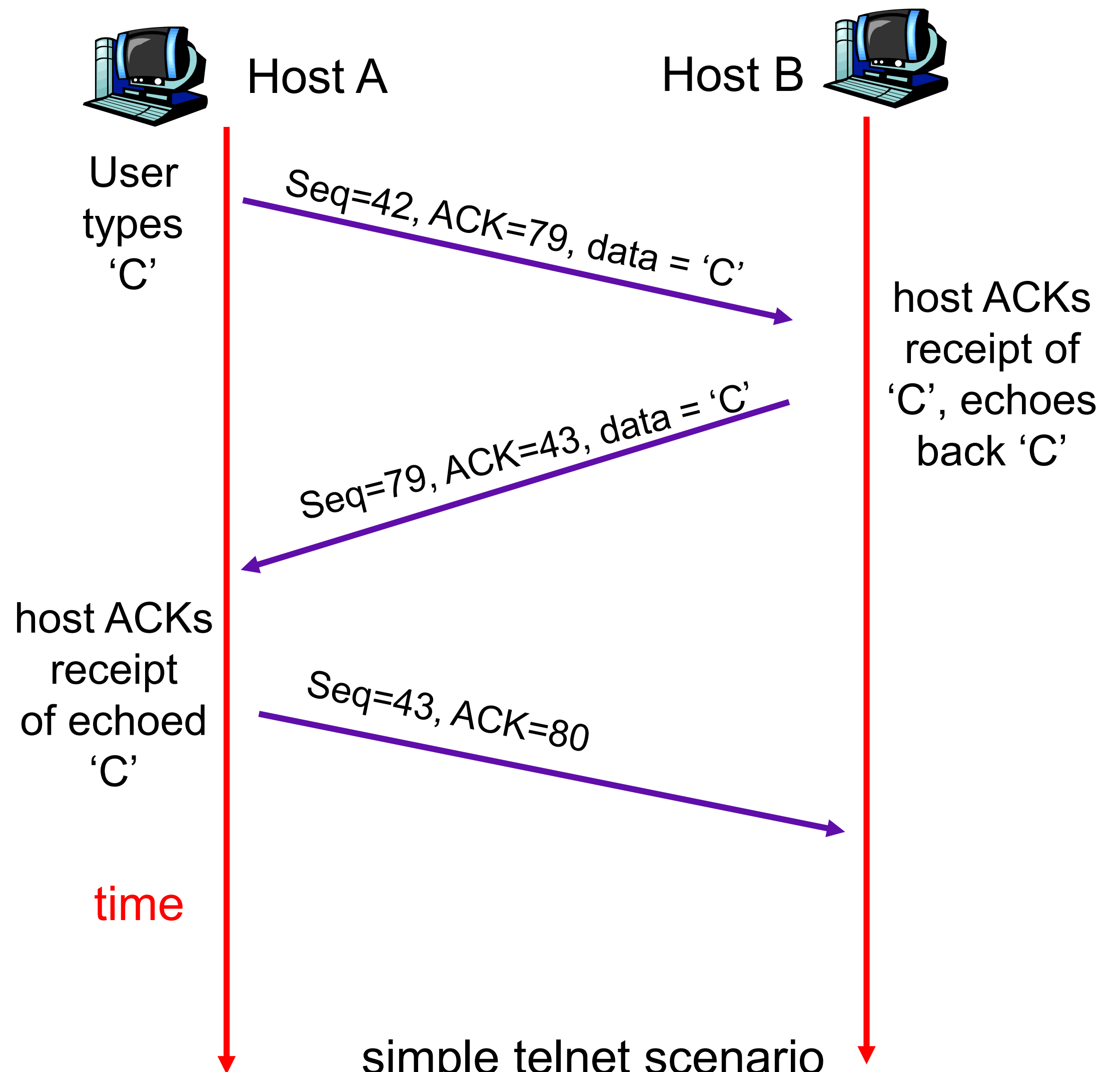
ACKs:

- seq # of next byte expected from other side
- cumulative ACK



simple telnet scenario

TCP seq. #'s and ACKs

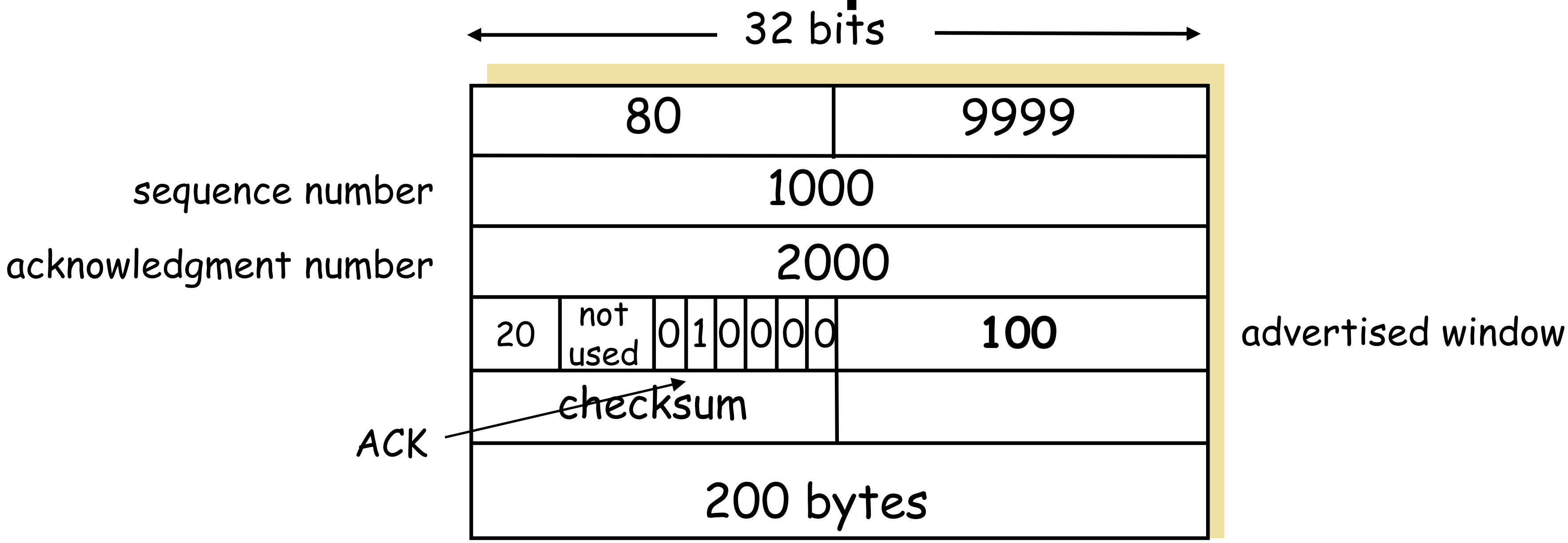


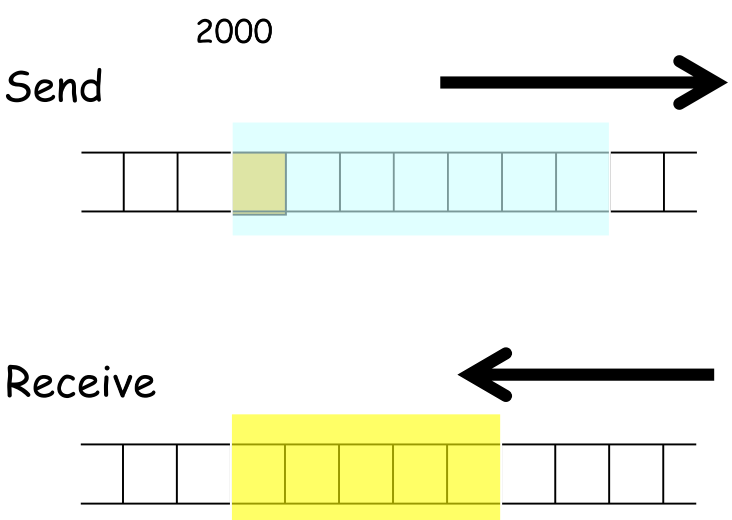
simple telnet scenario

TCP

- What's in the header?
- Sliding window
- RTT estimation
- More on sliding window
- Flow control
- Connection management
- Congestion

Example

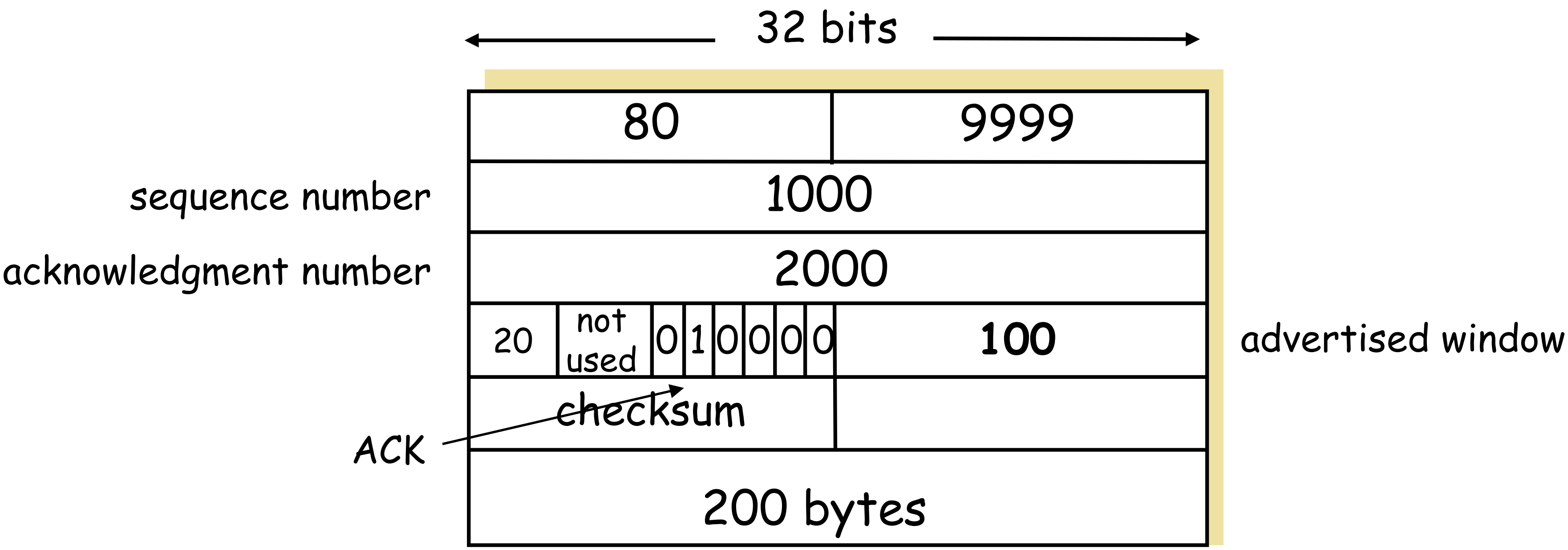


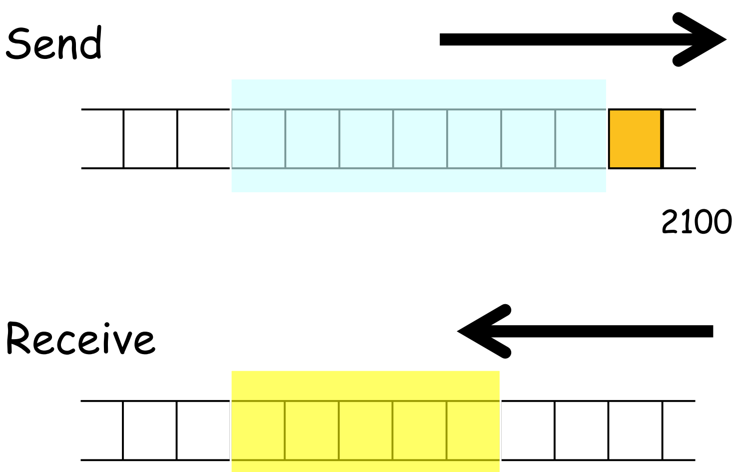


What is the minimum that A's SendBase value can be when A is recv'ing this segment?

Assume the segment IS a duplicate ACK

Example

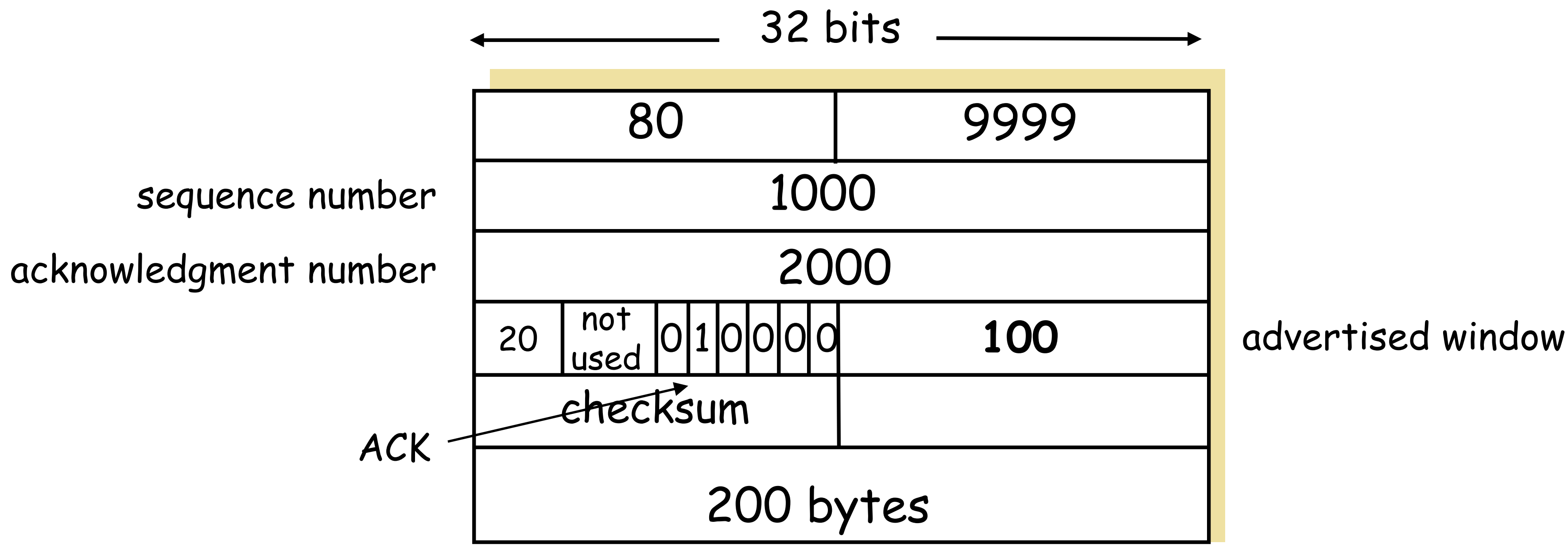


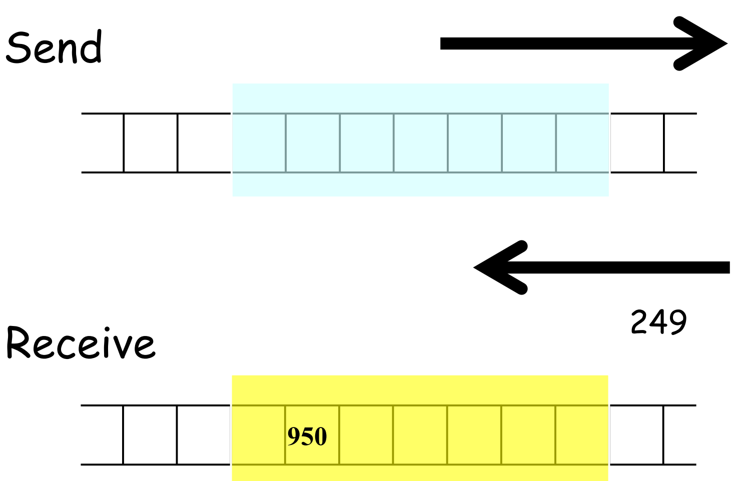


What is the maximum that A's NextSeqNum value can be when A is recv'ing this segment.

Assume the segment is NOT a duplicate ACK

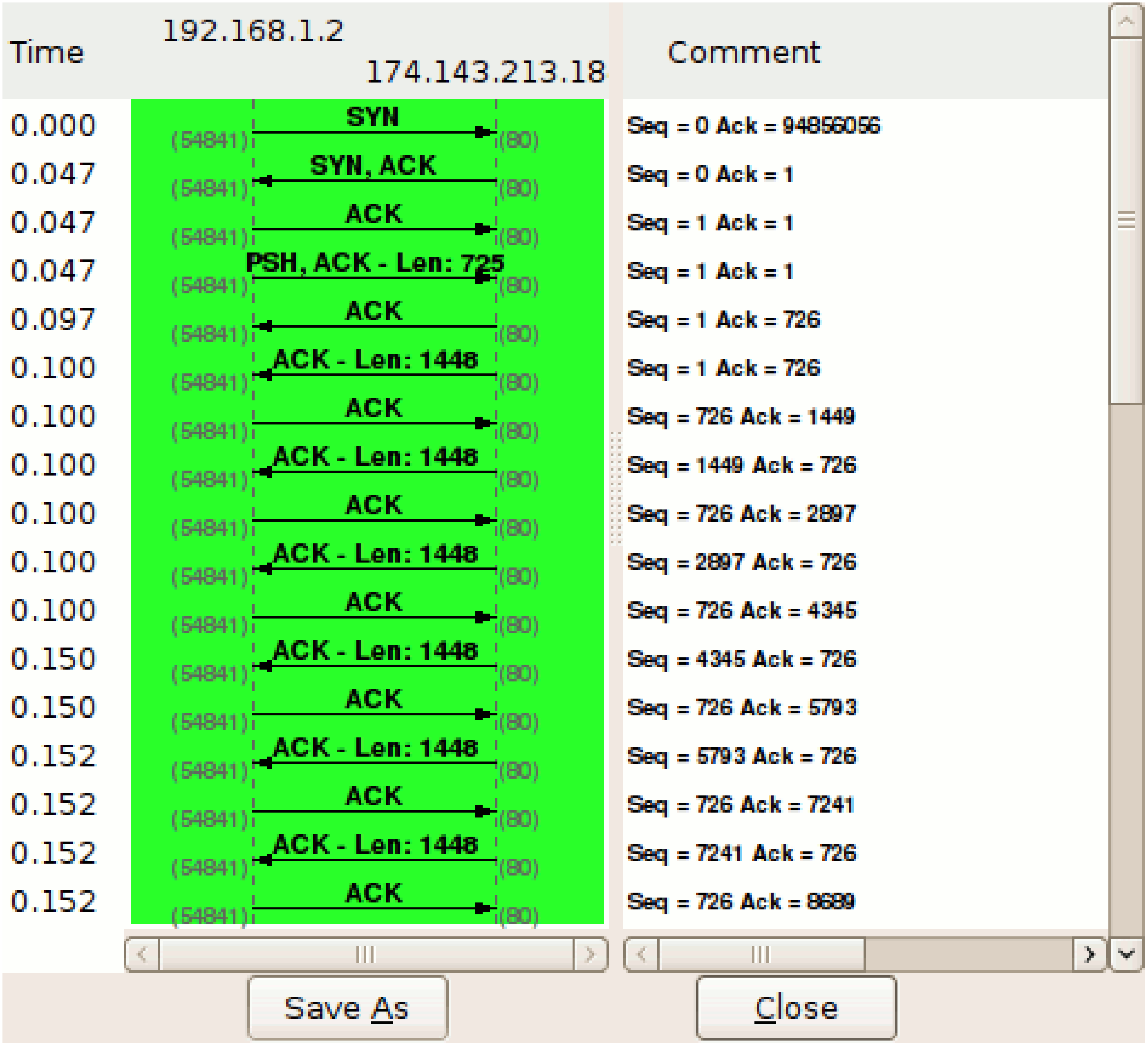
Example



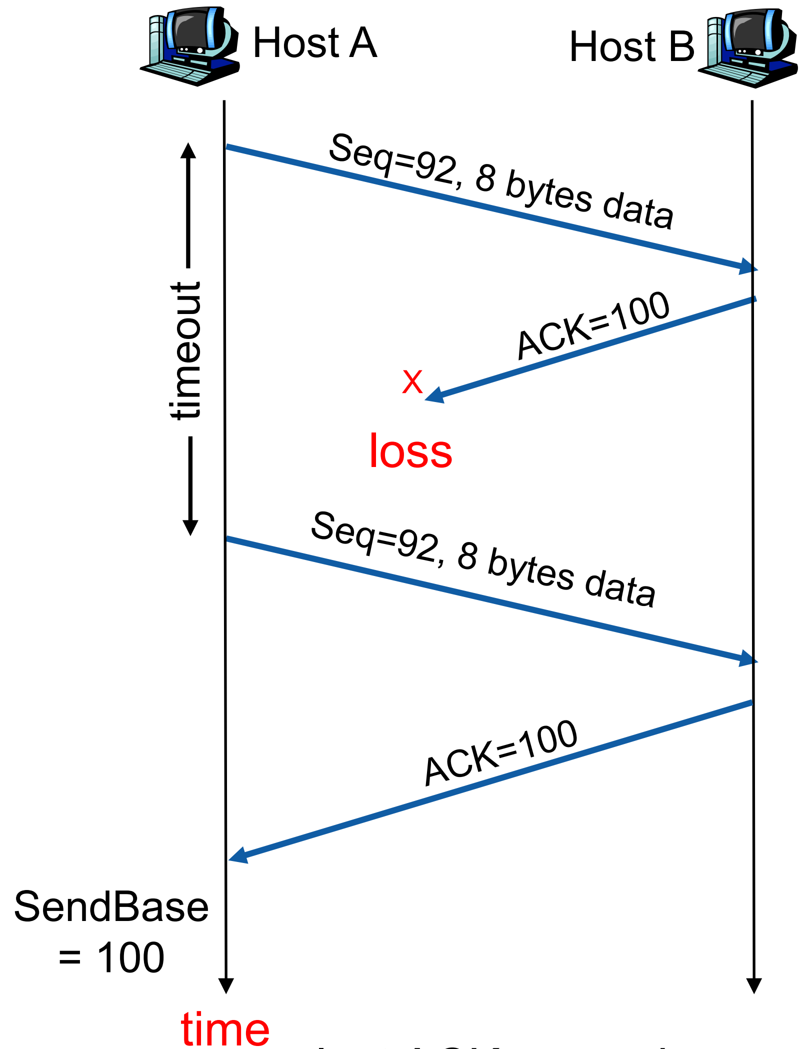


Suppose that B's data byte 950 is in A's recv'ing window.

What is the minimum amount of space in bytes, that A's receiving window must possess <u>beyond</u> **byte 950.**

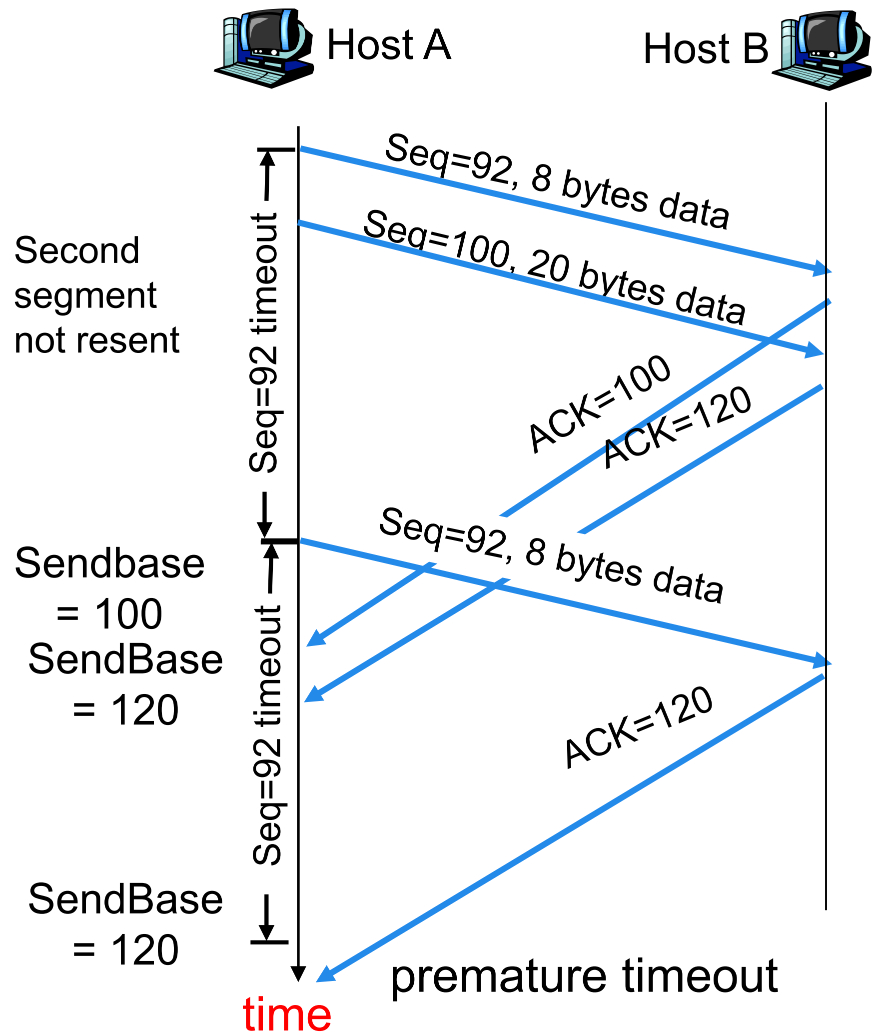

Time
192.168.1.2
174.143.213.18
Comment
0.000 SYN (54841) (80) Seq = 0 Ack = 94856056
0.047 SYN, ACK (54841) (80) Seq = 0 Ack = 1
0.047 ACK (54841) (80) Seq = 1 Ack = 1
0.047 PSH, ACK - Len: 725 (54841) (80) Seq = 1 Ack = 1
0.097 ACK (54841) (80) Seq = 1 Ack = 726
0.100 ACK - Len: 1448 (54841) (80) Seq = 1 Ack = 726
0.100 ACK (54841) (80) Seq = 726 Ack = 1449
0.100 ACK - Len: 1448 (54841) (80) Seq = 1449 Ack = 726
0.100 ACK (54841) (80) Seq = 726 Ack = 2897
0.100 ACK - Len: 1448 (54841) (80) Seq = 2897 Ack = 726
0.100 ACK (54841) (80) Seq = 726 Ack = 4345
0.150 ACK - Len: 1448 (54841) (80) Seq = 4345 Ack = 726
0.150 ACK (54841) (80) Seq = 726 Ack = 5793
0.152 ACK - Len: 1448 (54841) (80) Seq = 5793 Ack = 726
0.152 ACK (54841) (80) Seq = 726 Ack = 7241
0.152 ACK - Len: 1448 (54841) (80) Seq = 7241 Ack = 726
0.152 ACK (54841) (80) Seq = 726 Ack = 8689
Save As
Close

TCP: retransmission scenarios

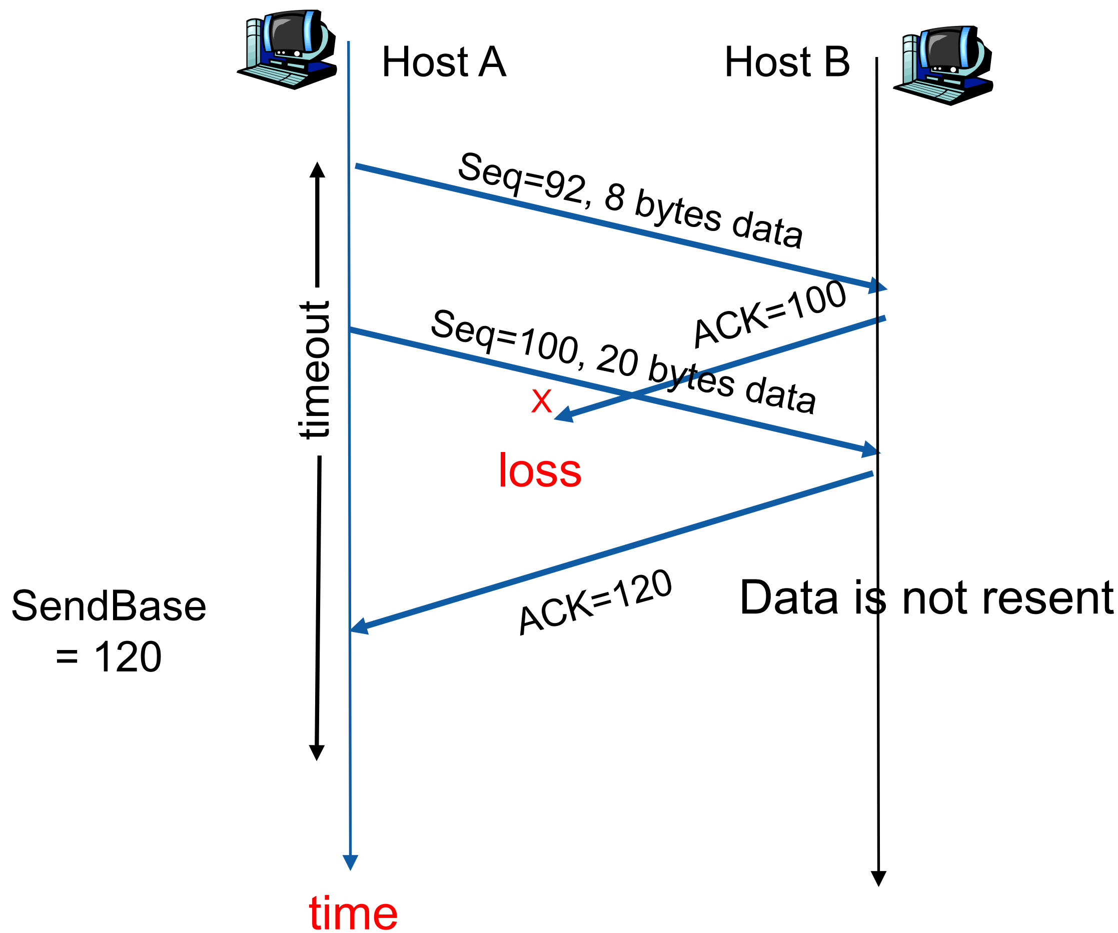


lost ACK scenario

TCP: retransmission scenarios



TCP retransmission scenarios (more)



Cumulative ACK scenario

TCP

- What's in the header?
- Sliding window
- RTT estimation
- More on sliding window
- Flow control
- Connection management
- Congestion

Time-out Problem?

- Rate depends on the window sizes, loss rate and round-trip time in acknowledging data.
- BUT
  - Congestion in the Internet
  - Conditions at the end-stations
  - Properties of the network
  - Size and timing of data segments
- Through-put rate is going to vary dynamically

Time-out and Retransmission

- Purpose: sets timer for each segment, retransmit earliest unacknowledged segment when timer goes off. (one timer, adds segment to retransmission queue)
- Problem: In the Internet we don’t a priori know the RTT of a segment. It is going to vary depending on the traffic.

TIMER MANAGEMENT

Setting the Time-out value?

- too short: premature timeout
  - unnecessary retransmissions
  - add to network congestion
  - Retransmission (hurts everyone)

- too long: slow reaction to segment loss
  - sluggish performance
  - slow
  - delayed transmission (hurts you, helps everyone)

How to estimate RTT?

- Static time-out? No!
- Adaptive time-out Yes!
  - Estimating time-out is difficult because
    - Peer TCP entity may accumulate acknowledgements and not acknowledge immediately
    - For retransmitted segments, can't tell whether acknowledgement is response to original transmission or retransmission
    - Network conditions may change suddenly
  - Better over-estimate than under-estimate!

RTT variance (Comer)

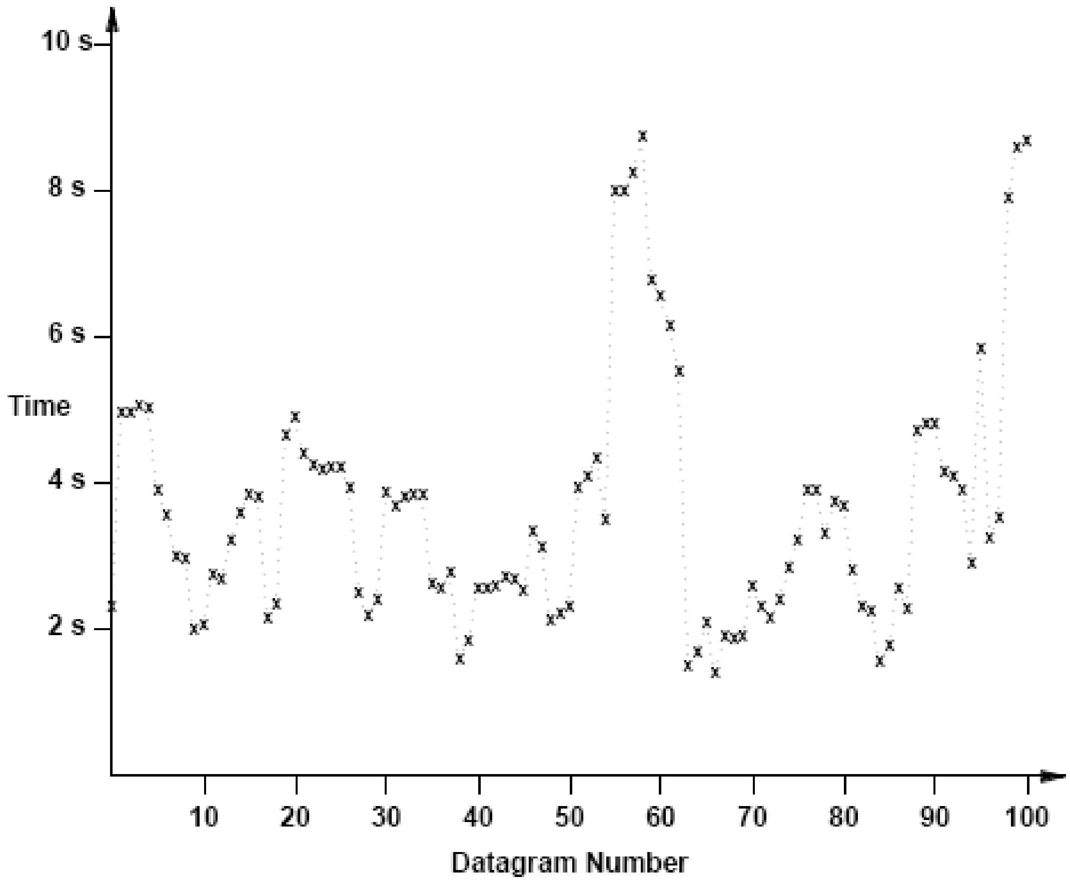
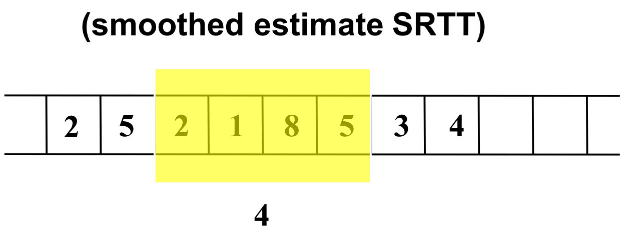


Figure 13.10 A plot of Internet round trip times as measured for 100 successive IP datagrams. Although the Internet now operates with much lower delay, the delays still vary over time.

How to estimate RTT?

- `SampleRTT`:

  - Moving average



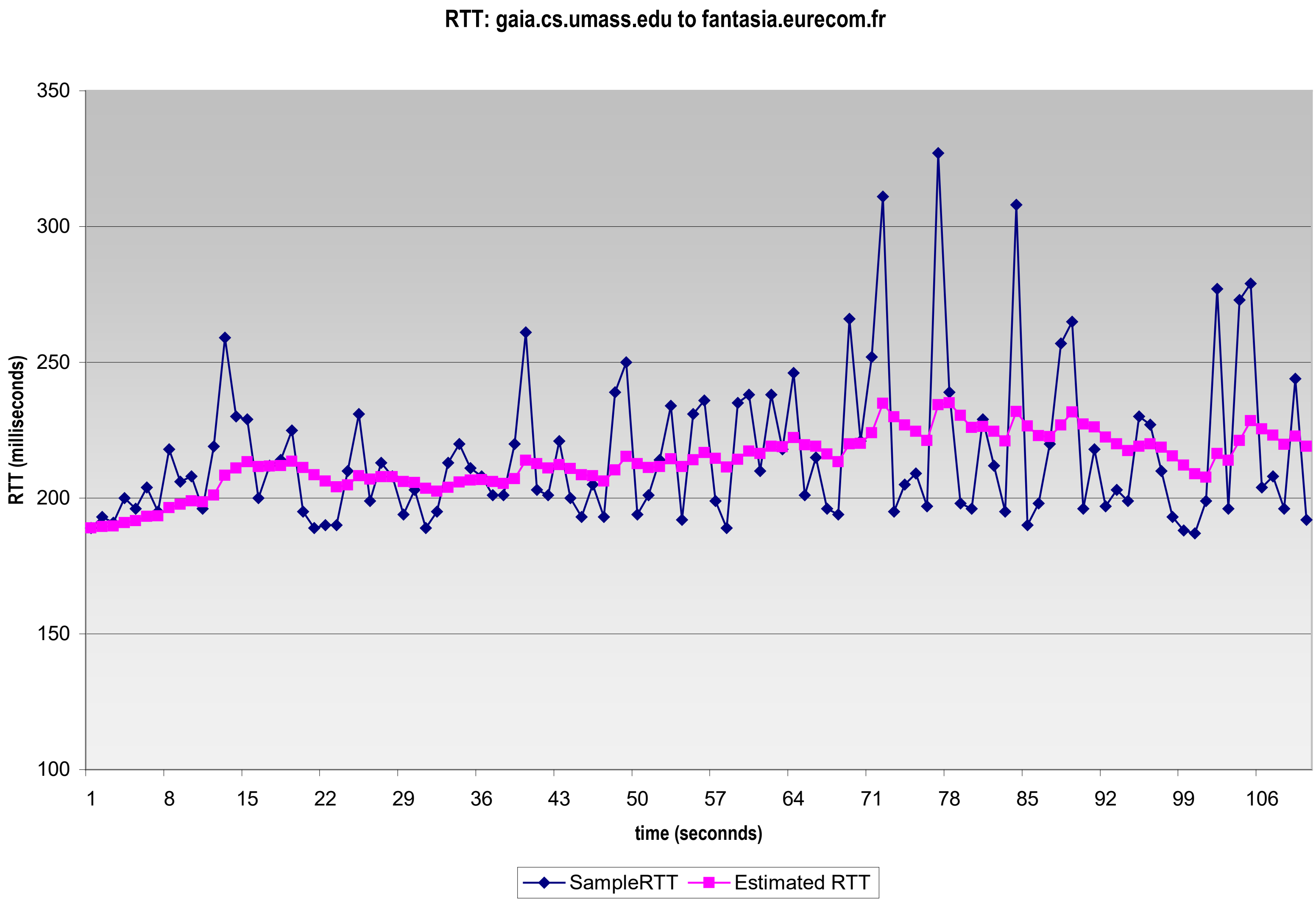
  - Exponential weighted moving average

Initially TCP used:

$$\textbf{EstimatedRTT} = (1-\alpha)*\textbf{EstimatedRTT} + \alpha*\textbf{SampleRTT}$$

- Exponential weighted moving average. (A dial we can adjust to change the sensitivity of RTT-estimate to history)
- typical value: α **=** 0.125
- Time-out is a constant times EstimatedRTT
  - **Time-out = β × EstimatedRTT**
  - Recommended setting for β was 2

Example RTT estimation:



A better Estimate

- Research (Jacobson) showed that this estimate did not respond quickly in high variance situations.

- 1989 TCP specification required estimates of both average and variance

High Variance (Comer)

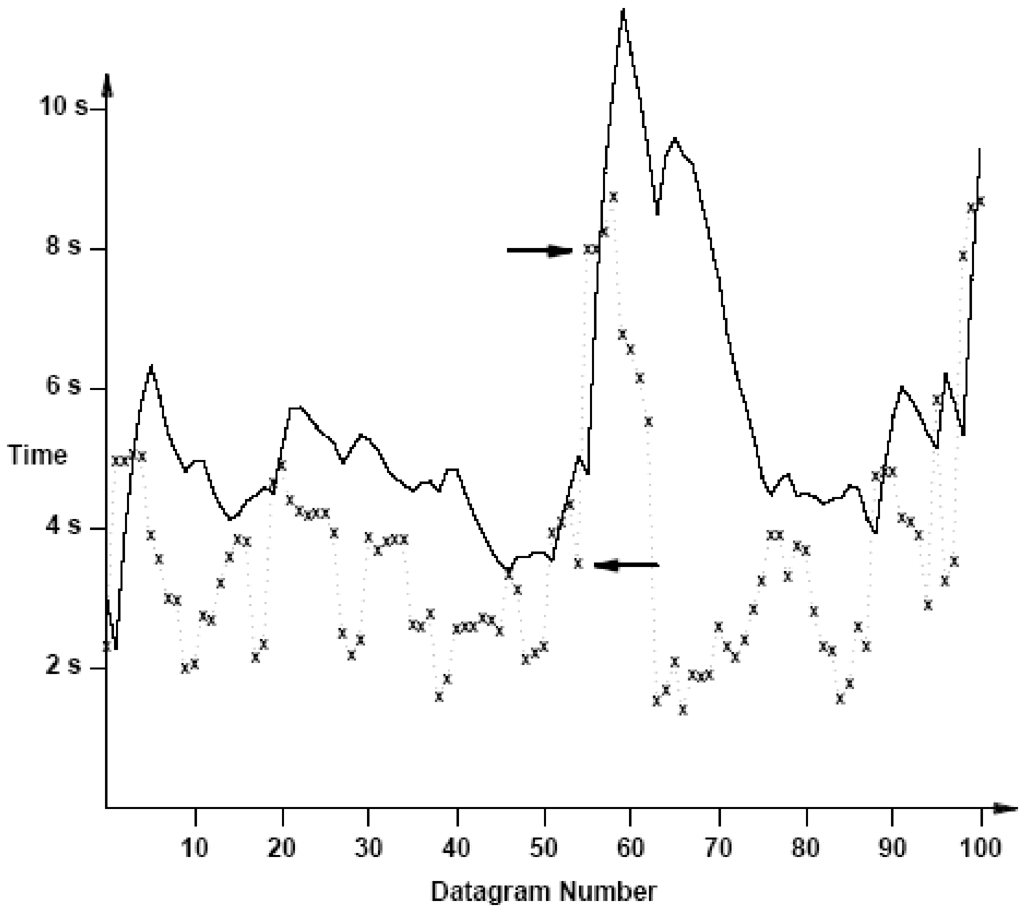


Figure 13.12 The TCP retransmission timer for the data from Figure 13.10. Arrows mark two successive datagrams where the delay doubles.

Summary

- Need both average and variance and being selective
- Want to avoid time-outs
- Existing techniques use selective sampling using estimates of the average variance of the RTT time
- Internet and TCP makes it difficult to predict

TCP

- What's in the header?
- Sliding window
- RTT estimation
- More on sliding window
- Flow control
- Connection management
- Congestion -- omit

TCP Connection Management

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

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- *client:* connection initiator

```
Socket clientSocket = new
 Socket("hostname","port

 number");
```

- *server:* contacted by client

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

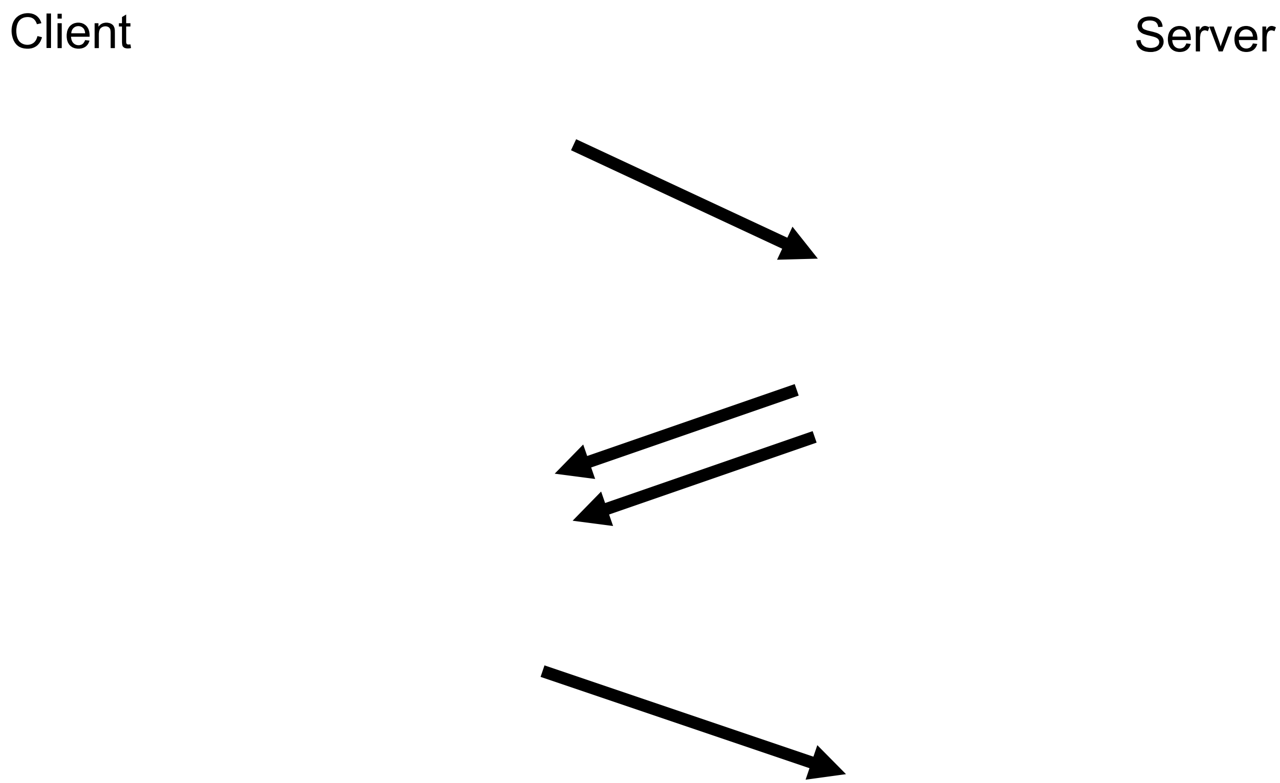
Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

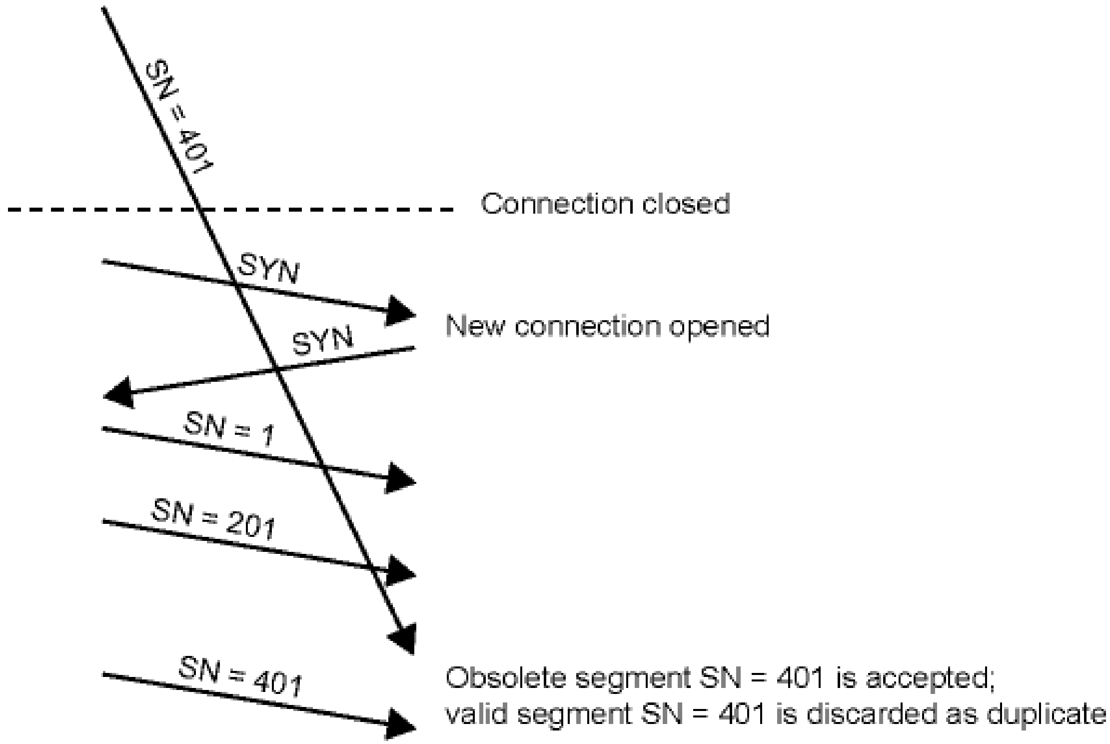
Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

Initial Sequence Number (ISN)



Single ISN problems

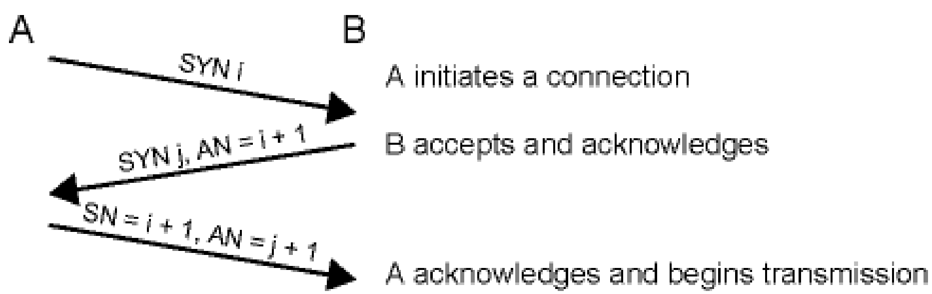


- Add a unique session ID to each stream
- But what if machine re-boots, clocks?
  - What if the machine re-boots?

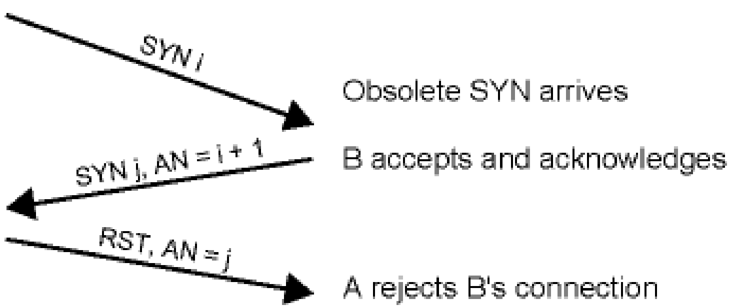
Some Possible Scenarios

ISN initial sequence number

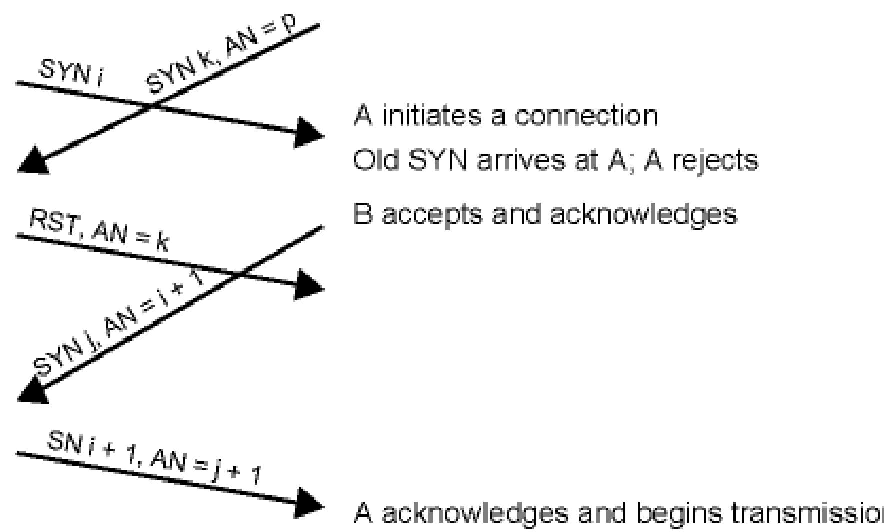
Assumption: MSL (maximum segment lifetime --- two minutes)



(a) Normal operation



(b) Delayed SYN



(c) Delayed SYN, ACK

Closing a connection

- Objective
  - Close without abruptly dropping the connection!

Graceful Close

- Send FIN i and receive FINACK i
- Receive FIN j and send FINACK j
- Wait twice maximum expected segment lifetime

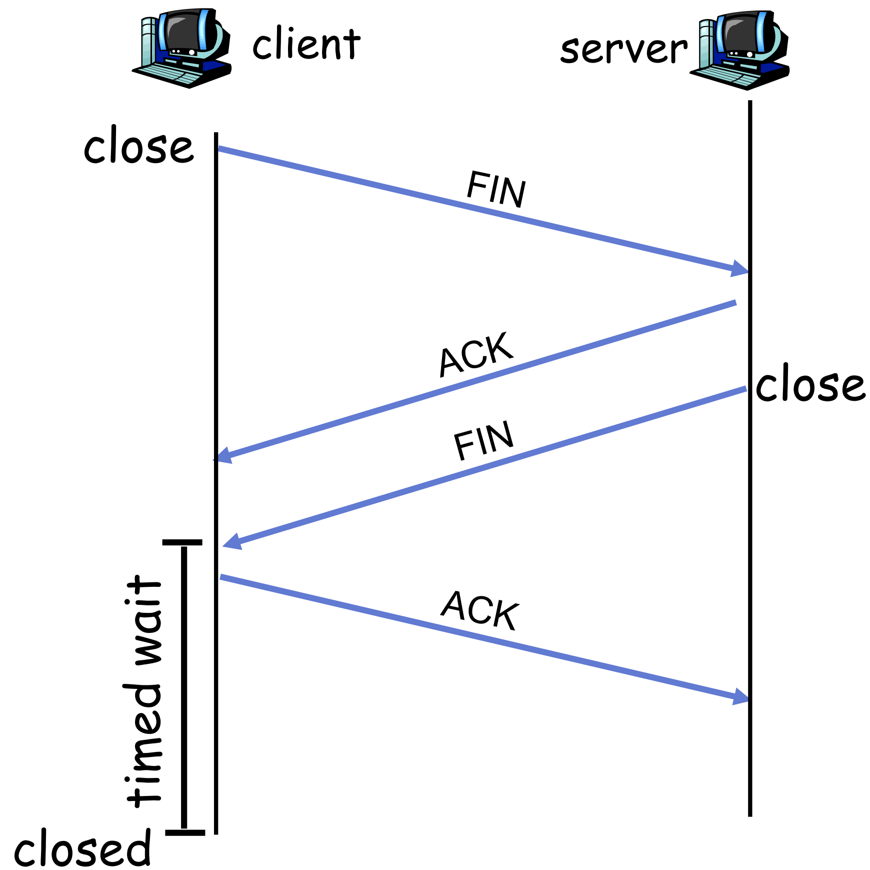
TCP Connection Management (cont.)

<u>Closing a connection:</u>

client closes socket:
`clientSocket.close();`

<u>Step 1:</u> client end system sends TCP FIN control segment to server

<u>Step 2:</u> server receives FIN, replies with ACK. Closes connection, sends FIN.



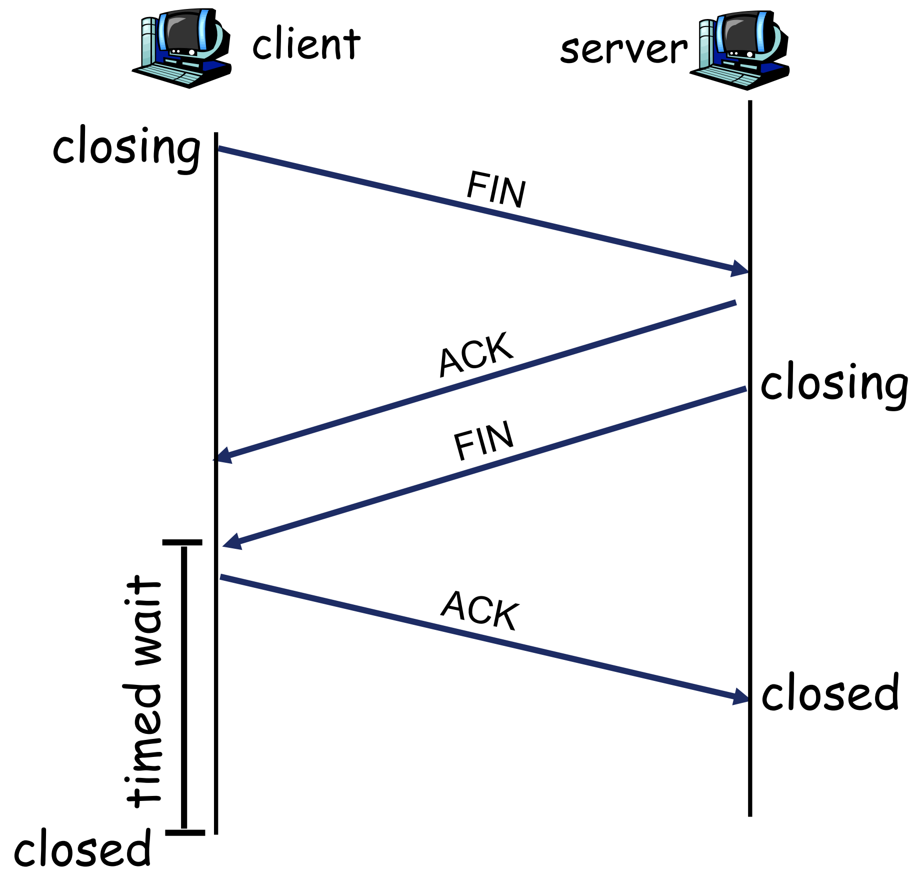
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

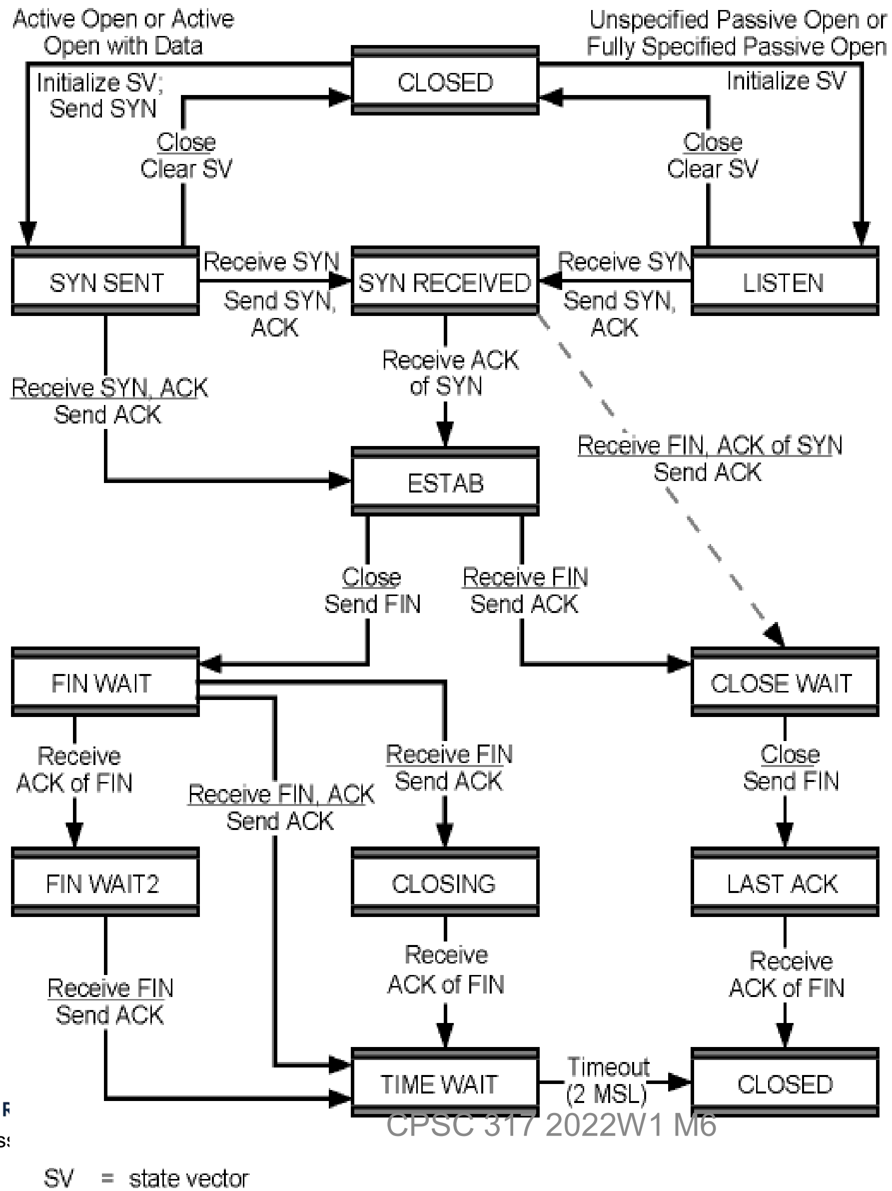
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP State Machine



Steven’s TCP state machine