Network Transport Layer: Transport Reliability: Sliding Windows; Connection Management

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

11/6/2018

Admin.: PS4

- Part 1
  - Discussion checkpoint: Nov. 11; code checkpoint Nov. 13
- Part 2
  - Discussion checkpoint: Nov. 16; all due Nov. 27

proj-sol:

```
 129   400  3045 FishThread.java
 388  1457 12873 Node.java
  51   167  1145 PingRequest.java
  83   250  2106 SimpleTCPSockSpace.java
 181   605  5248 TCPManager.java
 889  3088 26381 TCPSock.java
  60   149  1316 TCPSockID.java
 123   382  3866 TransferClient.java
 147   500  5059 TransferServer.java

2051  6998 61039 total
```

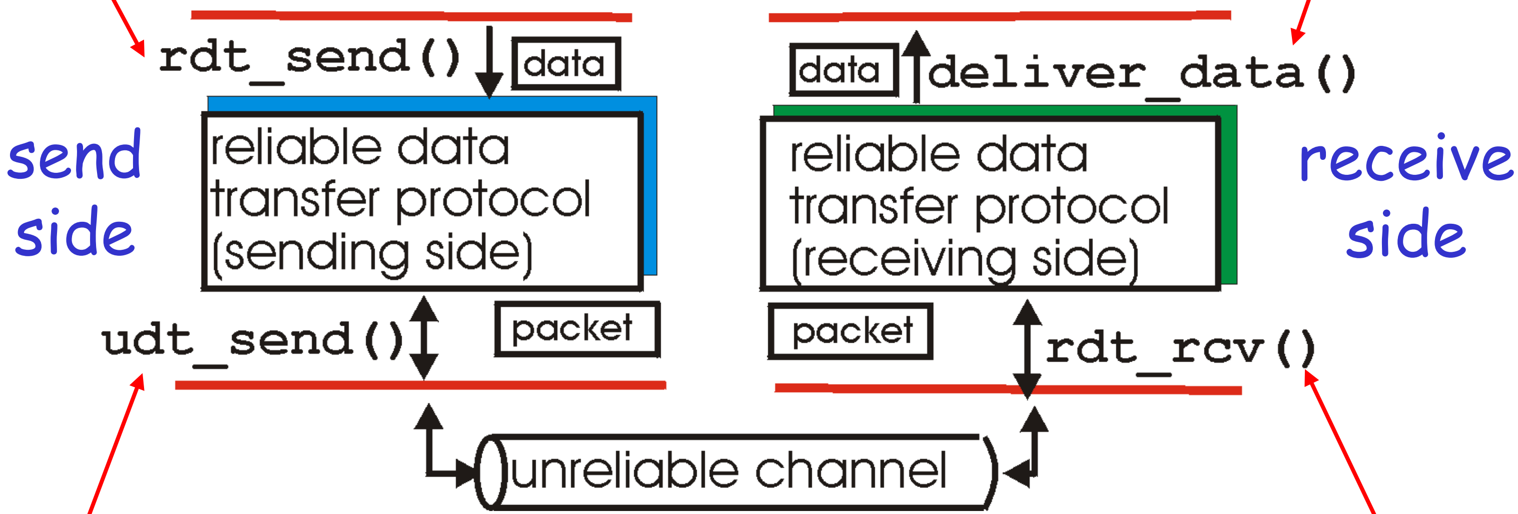
proj:

```
 129   400  3045 FishThread.java
 341  1301 11313 Node.java
  51   167  1145 PingRequest.java

  50   128   909 TCPManager.java
 132   460  3146 TCPSock.java

 123   382  3866 TransferClient.java
 147   500  5059 TransferServer.java

 973  3338 28483 total
```

Recap: Reliable Data Transfer Context

rdt_send(): called from above, (e.g., by app.)

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



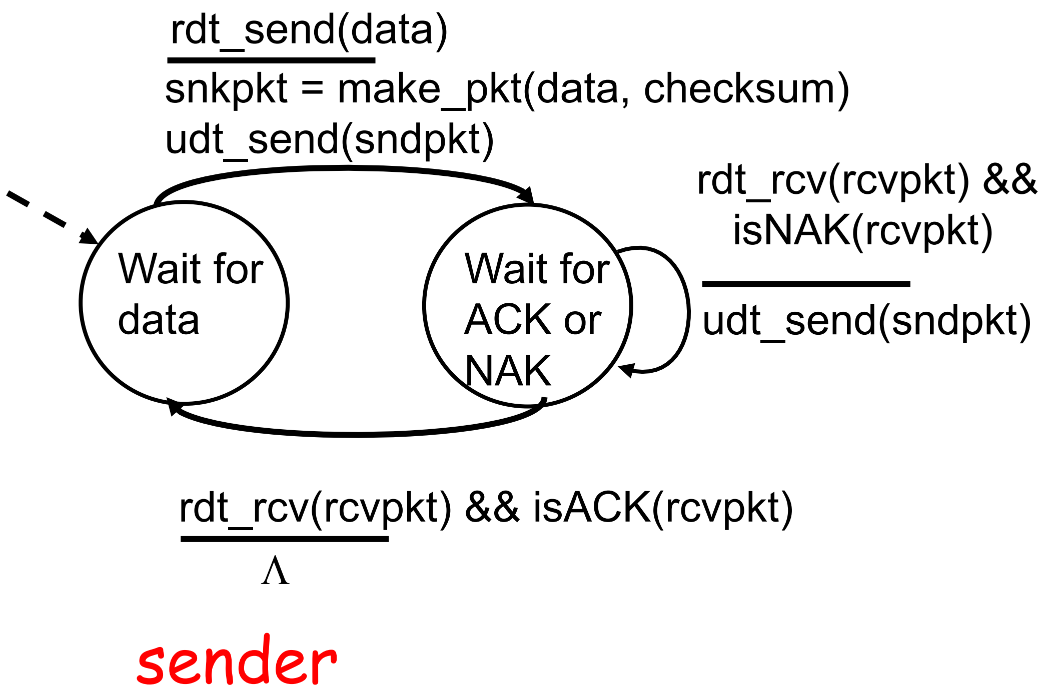
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

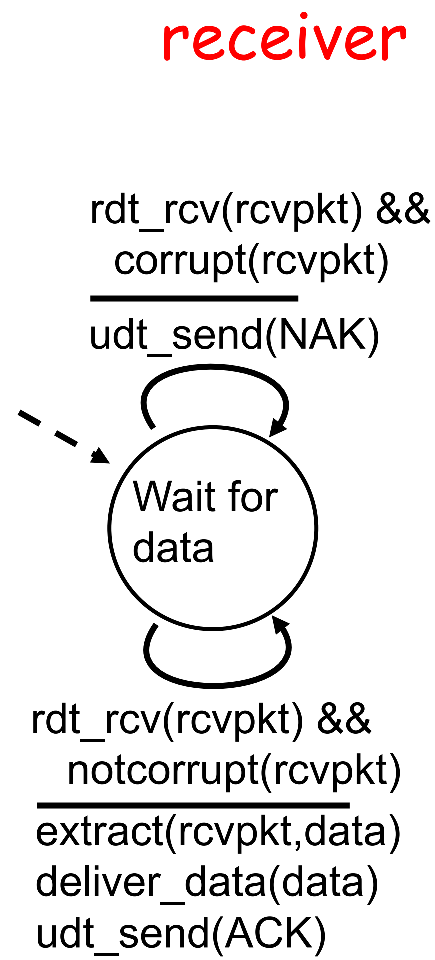
rdt_rcv(): called from below; when packet arrives on rcv-side of channel

Recap: Potential Channel Errors

- Factors to pay attention when designing rdt
  - Types of channel errors: Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt).
    - bit errors
    - loss (drop) of packets
    - reordering or duplication
  - Not only protocol but also analysis techniques

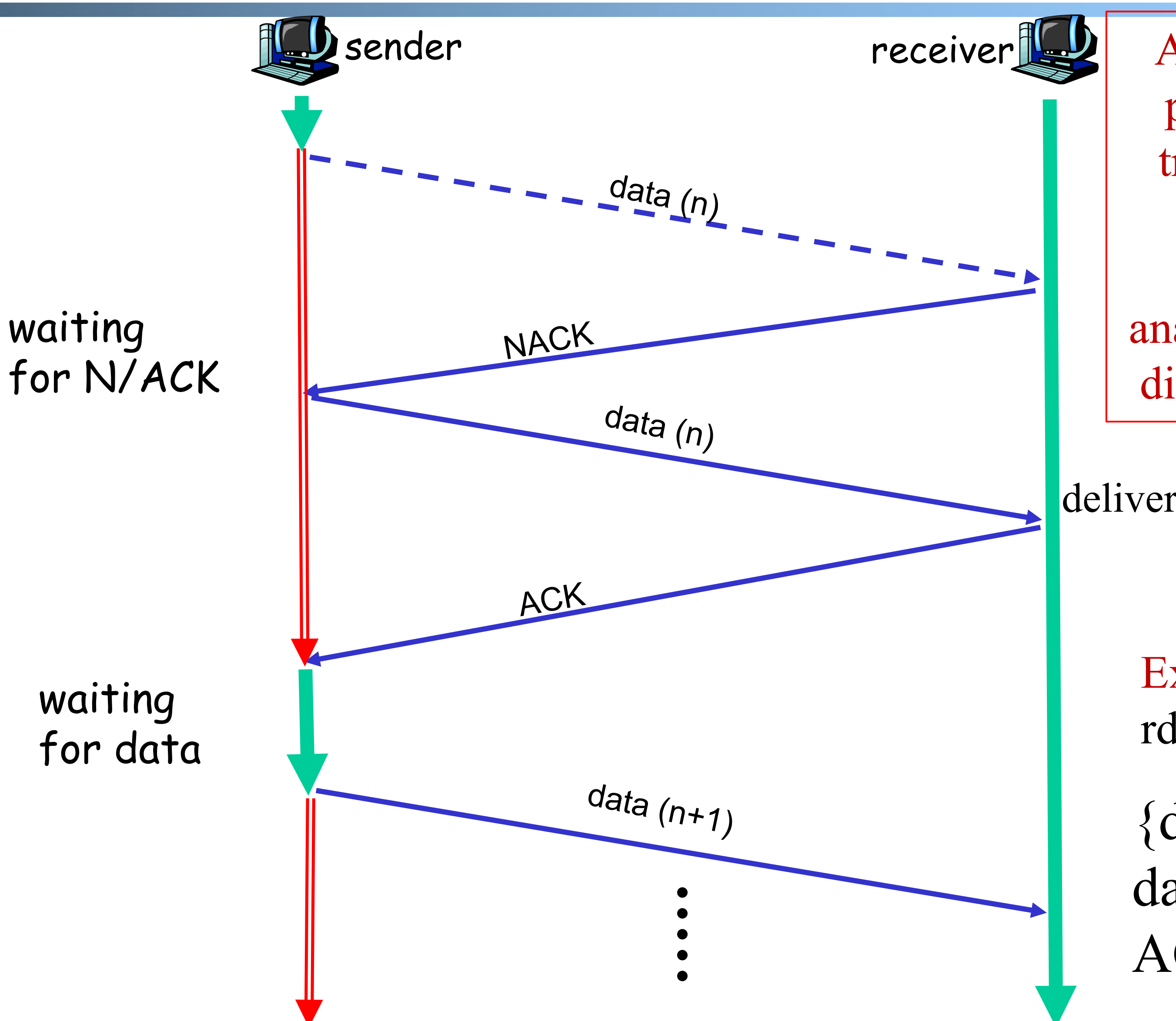
Recap: rdt2.0: Reliability allowing only Data Msg Corruption





Recap: Rdt2.0 Analysis

data^: <S data> <R data’>
data: <S data> <R data>

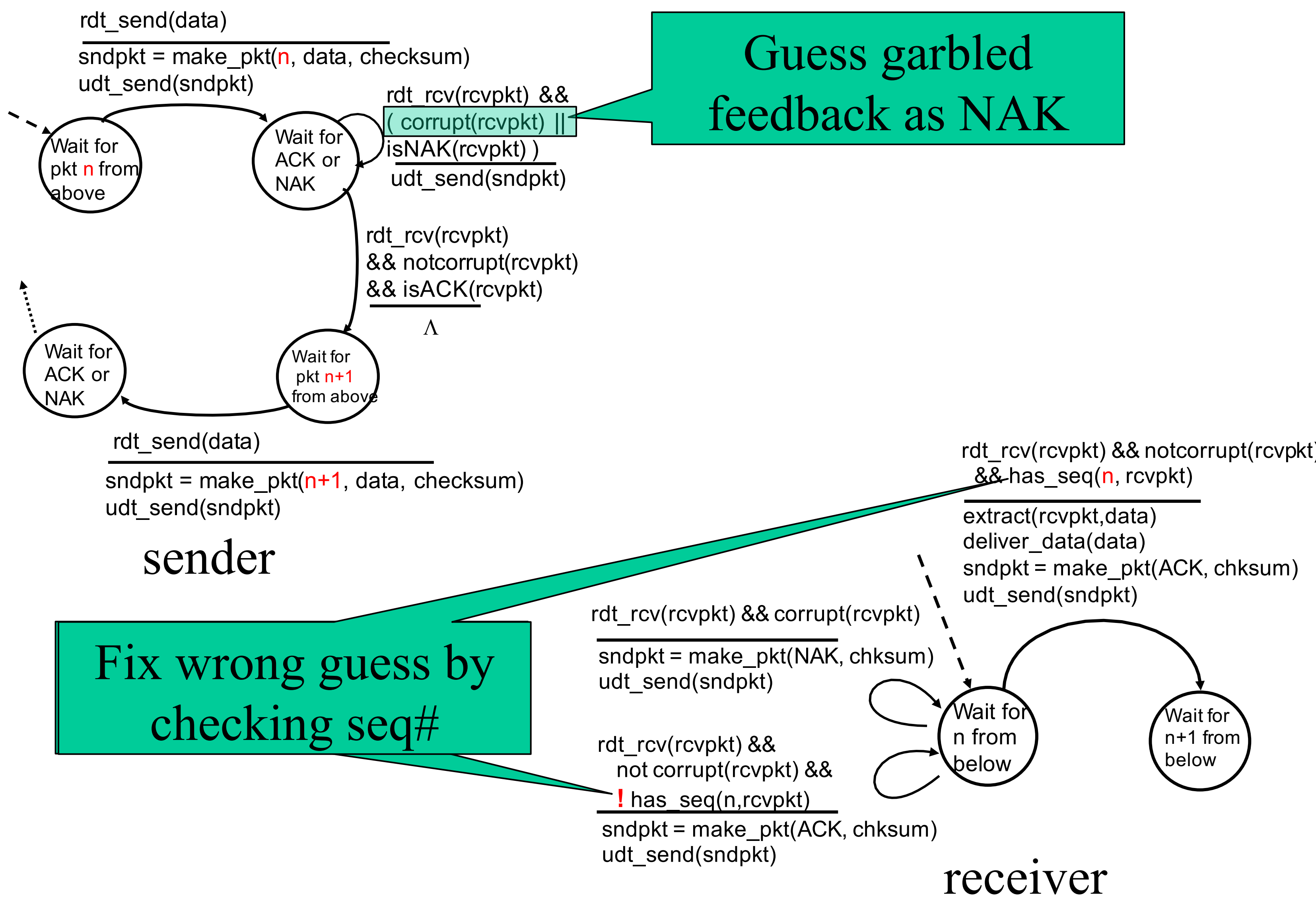


Analyzing set of all possible execution traces is a common technique to understand and analyze many types of distributed protocols.

Execution traces of rdt2.0:

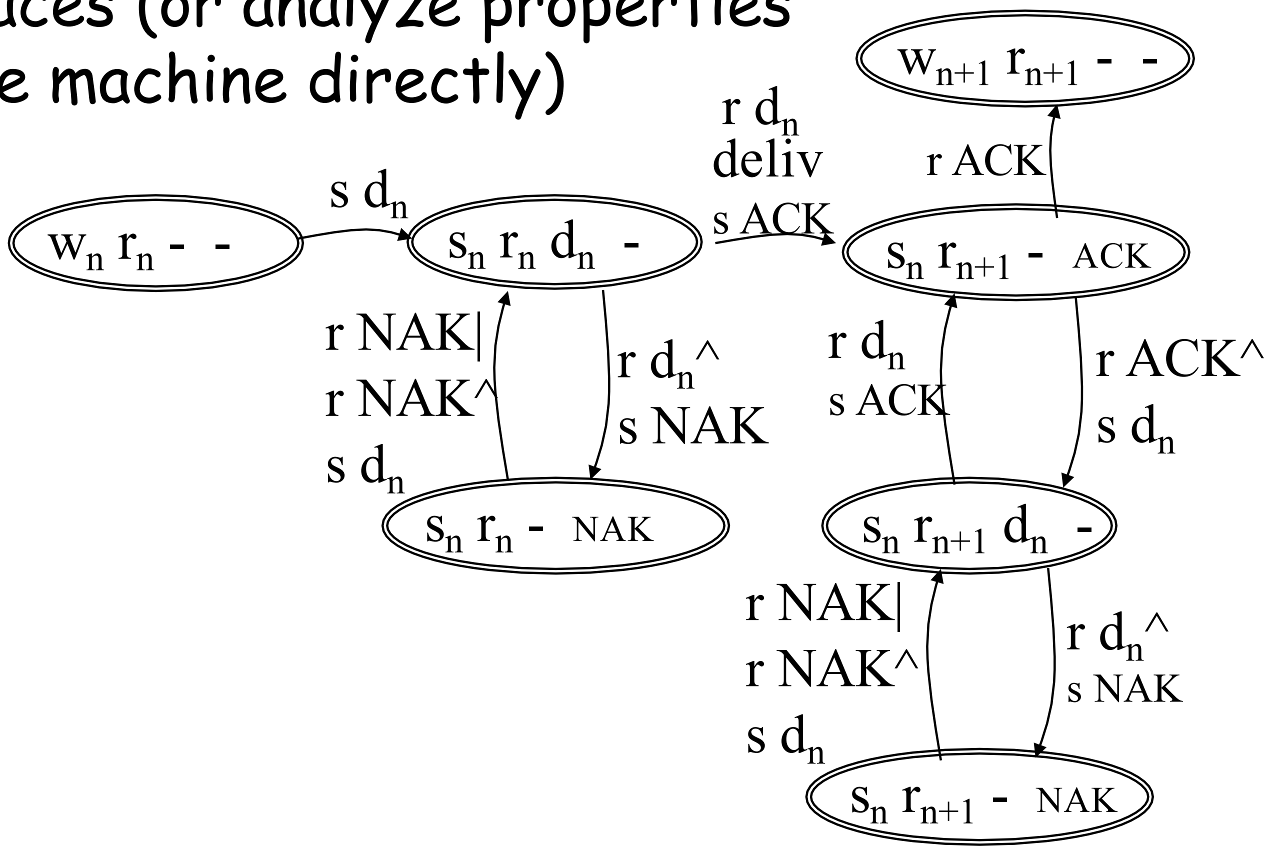
{data^ NACK}*
data deliver
ACK

Recap: rdt2.1b: Reliability allowing Data/Control Msg Corruption



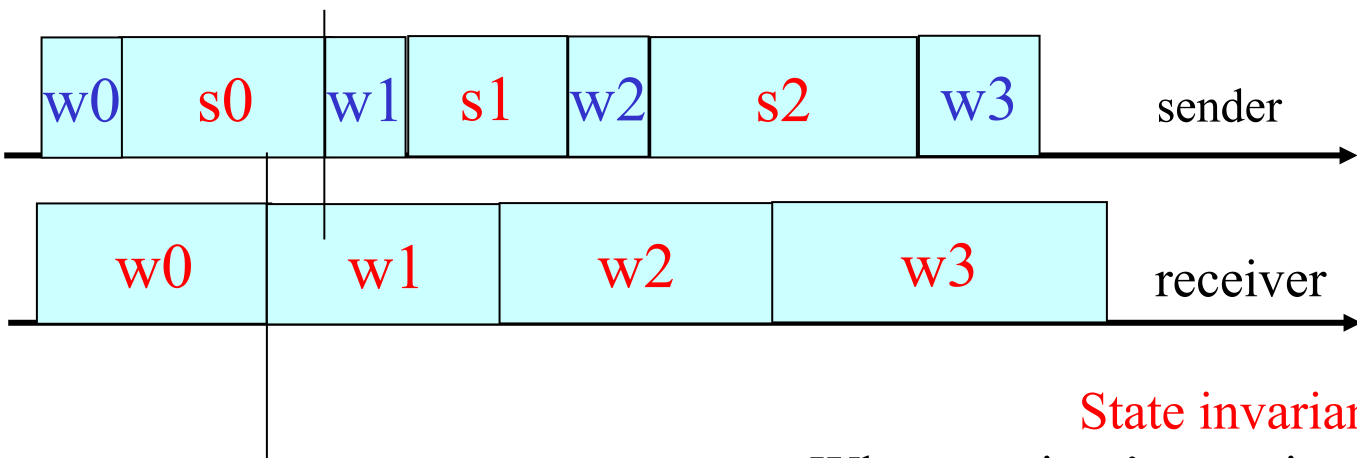
Recap: Protocol Analysis using (Generic) Execution Traces Technique

- A systematic approach to enumerating execution traces is to compute joint sender/receiver/channels state machine, and then convert the state machine to traces (or analyze properties on the machine directly)



Recap: Protocol Analysis using State Invariants

W1: wait for data with seq. 1
S1: sending data with seq. 1

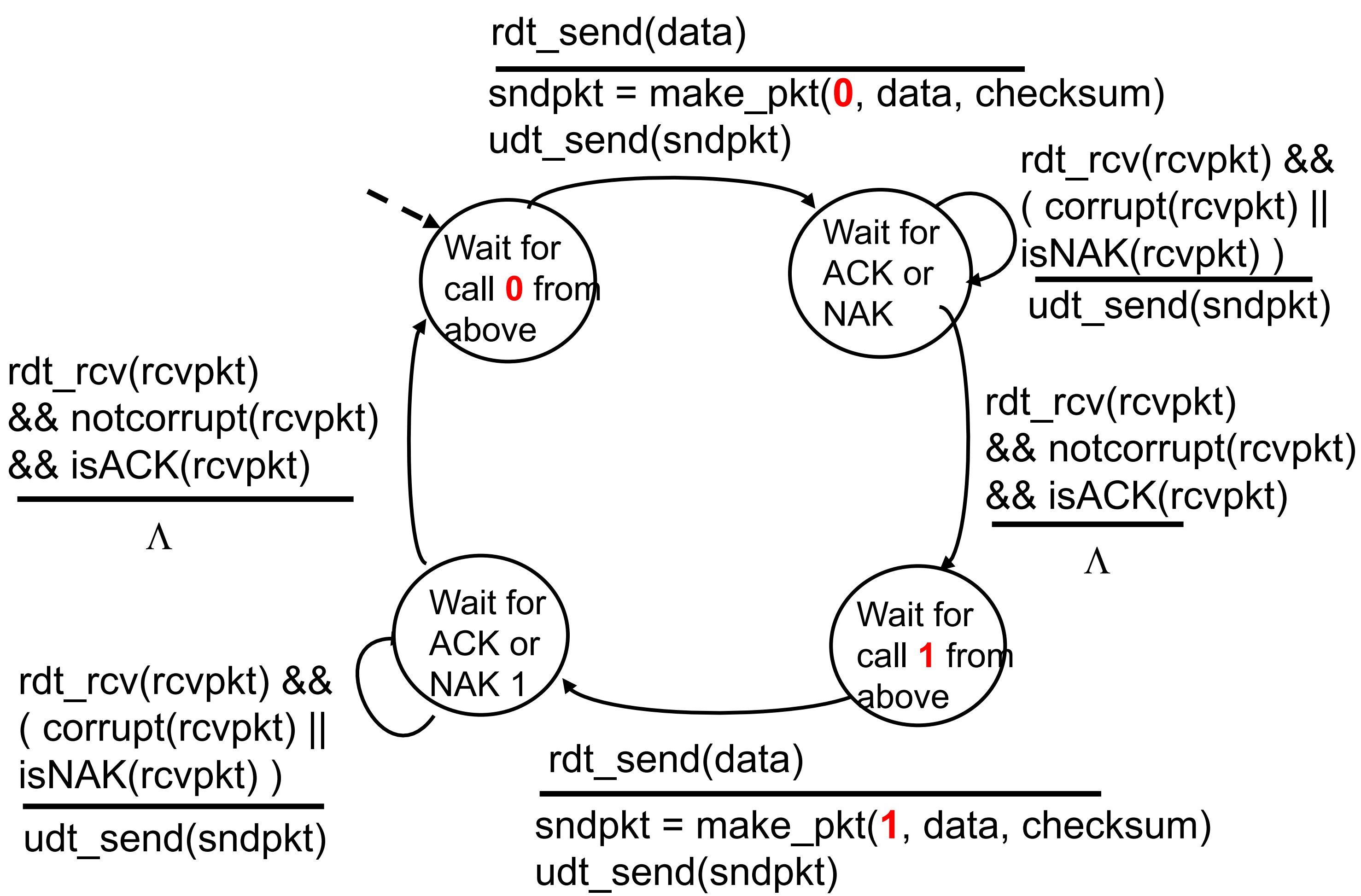


State invariant:

- When receiver's state is waiting for seq #n, sender's state can be sending either seq#n-1or seq#n, and only either #n or #n-1 packets can arrive

Implication: One bit (the last bit) is enough to distinguish the two states

rdt2.1c: Sender, Handles Garbled ACK/NAKs: Using 1 bit (Alternating-Bit Protocol)



rdt2.1c: Receiver, Handles Garbled ACK/NAKs: Using 1 bit

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

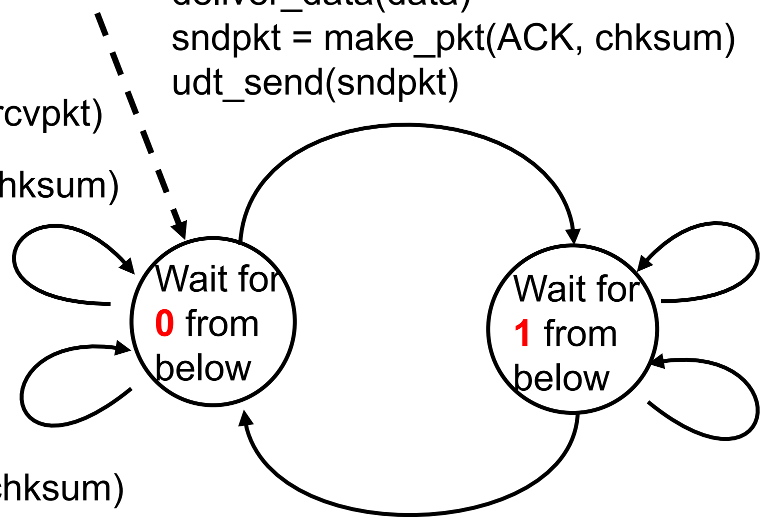
---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)



rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1c: Summary

Sender:

- state must "remember" whether "current" pkt has 0 or 1 seq. #

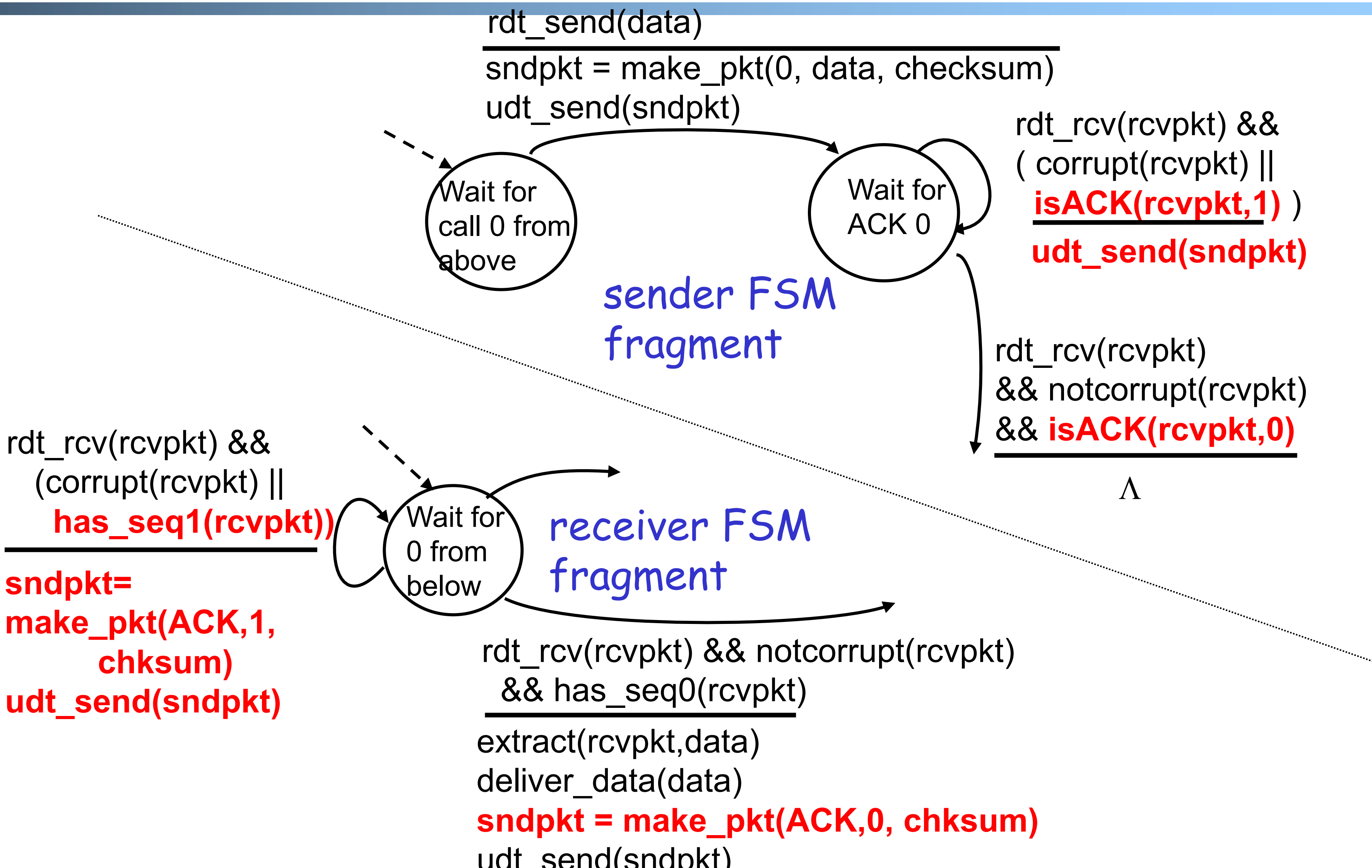
Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #

rdt2.2: a NAK-free protocol

- Same functionality as rdt2.1c, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: Sender, Receiver Fragments



Outline

- Admin and review
- Reliable data transfer
  - perfect channel
  - channel with bit errors
  - channel with bit errors and losses

rdt3.0: Channels with Errors *and* Loss

New assumption: underlying channel can also lose packets (data or ACKs)

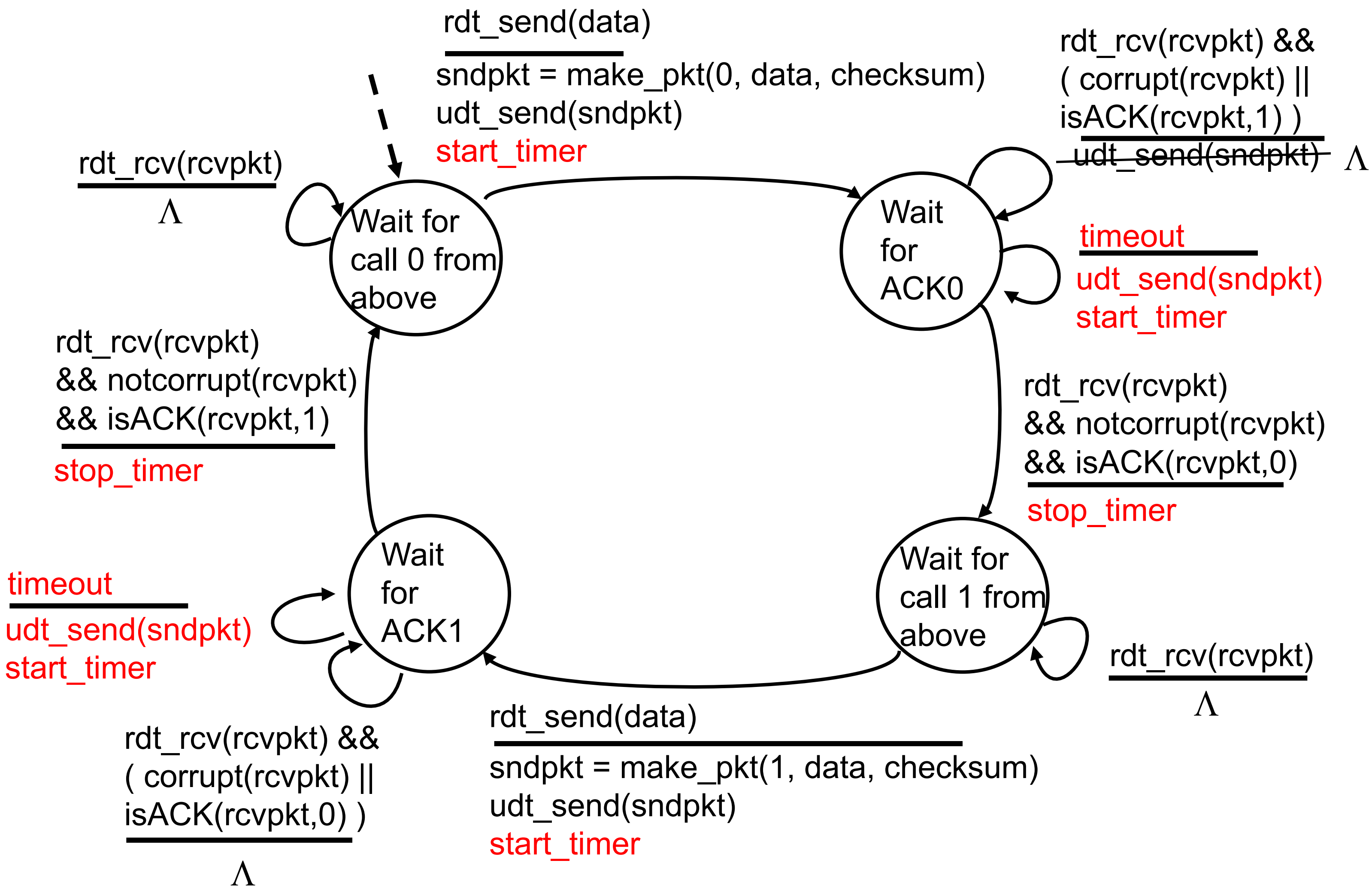
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: What can rdt2.2 go wrong under losses?

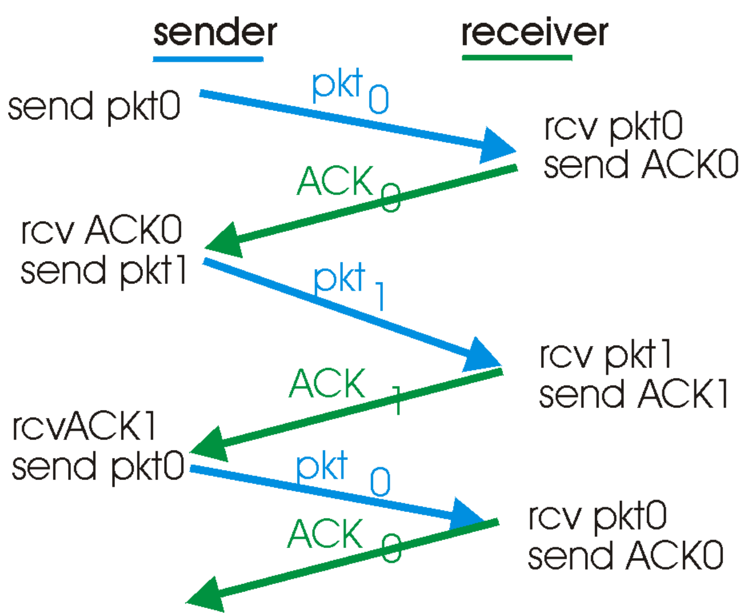
Approach: sender waits "reasonable" amount of time for ACK

- requires countdown timer
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed

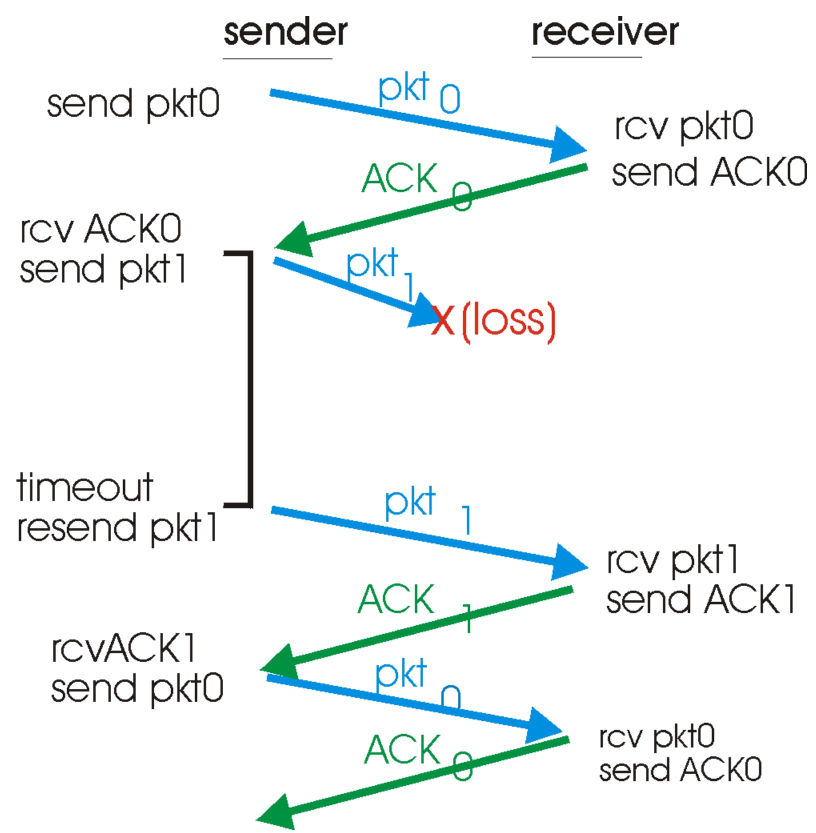
rdt3.0 Sender



rdt3.0 in Action

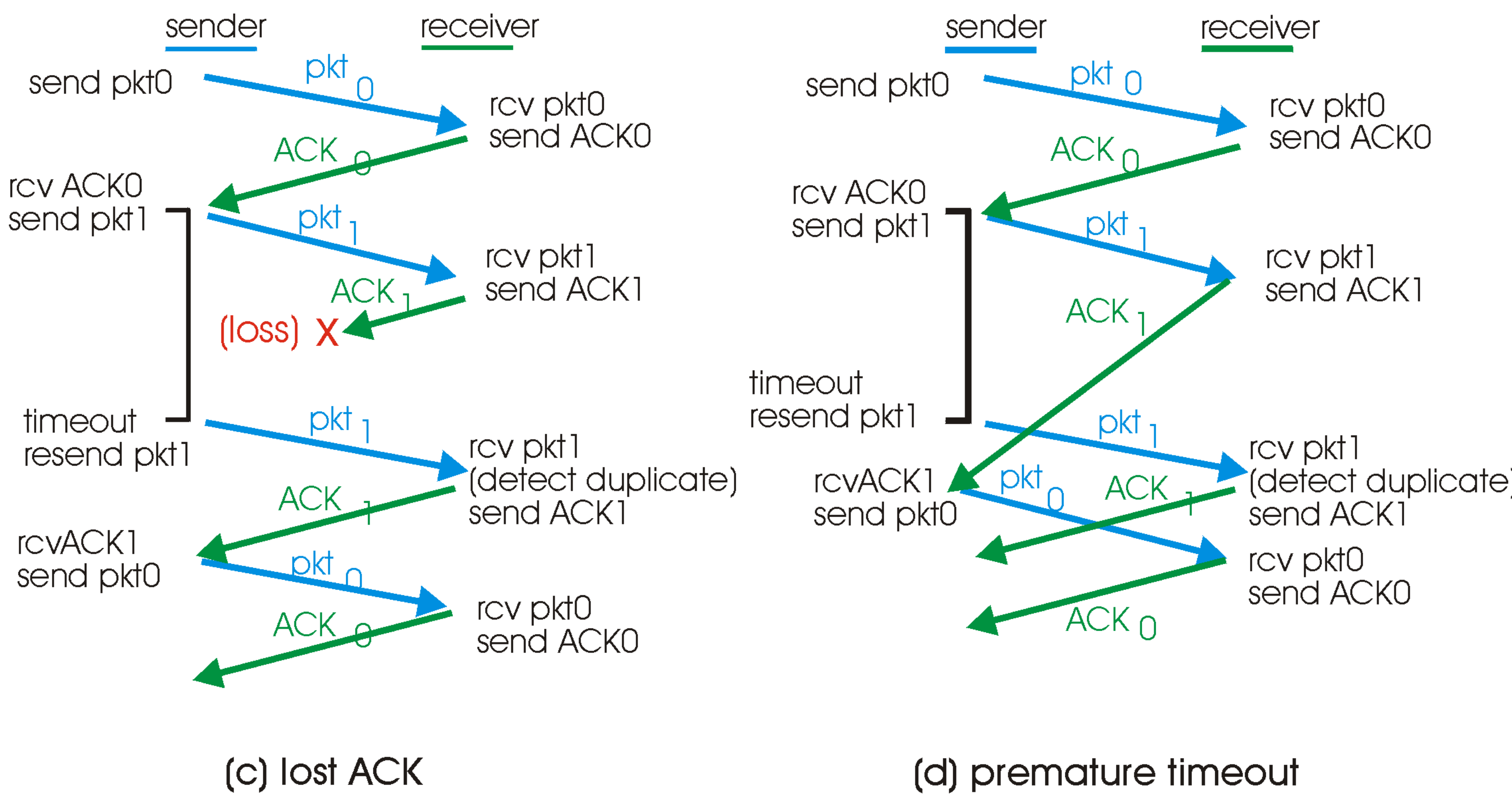


(a) operation with no loss



(b) lost packet

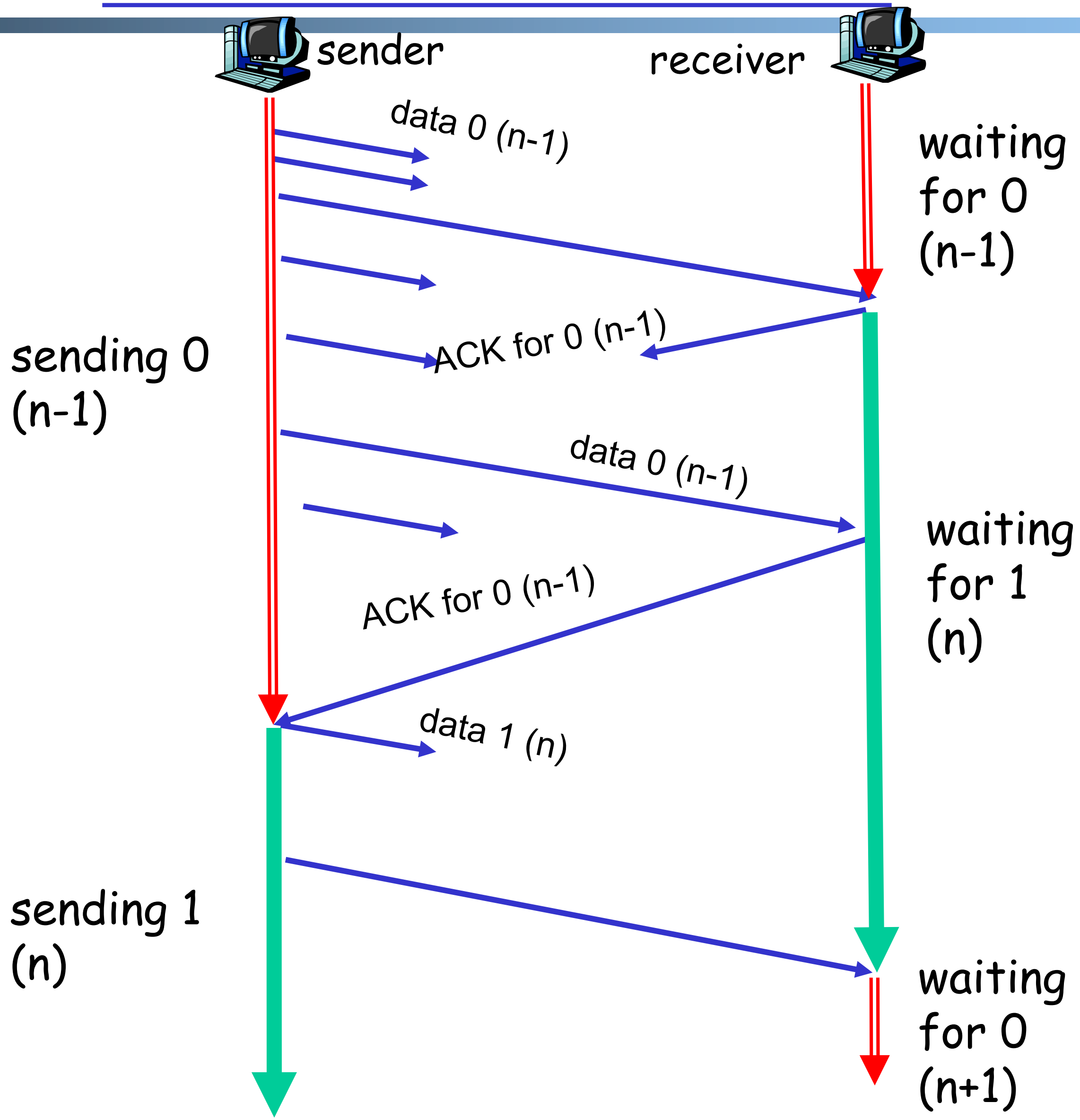
rdt3.0 in Action



Question to think about: How to determine a good timeout value?

Home exercises: (1) What are execution traces of rdt3.0? What are some state invariants of rdt3.0? (2) What are some state invariants?

rdt3.0: Protocol Analysis using State Invariants

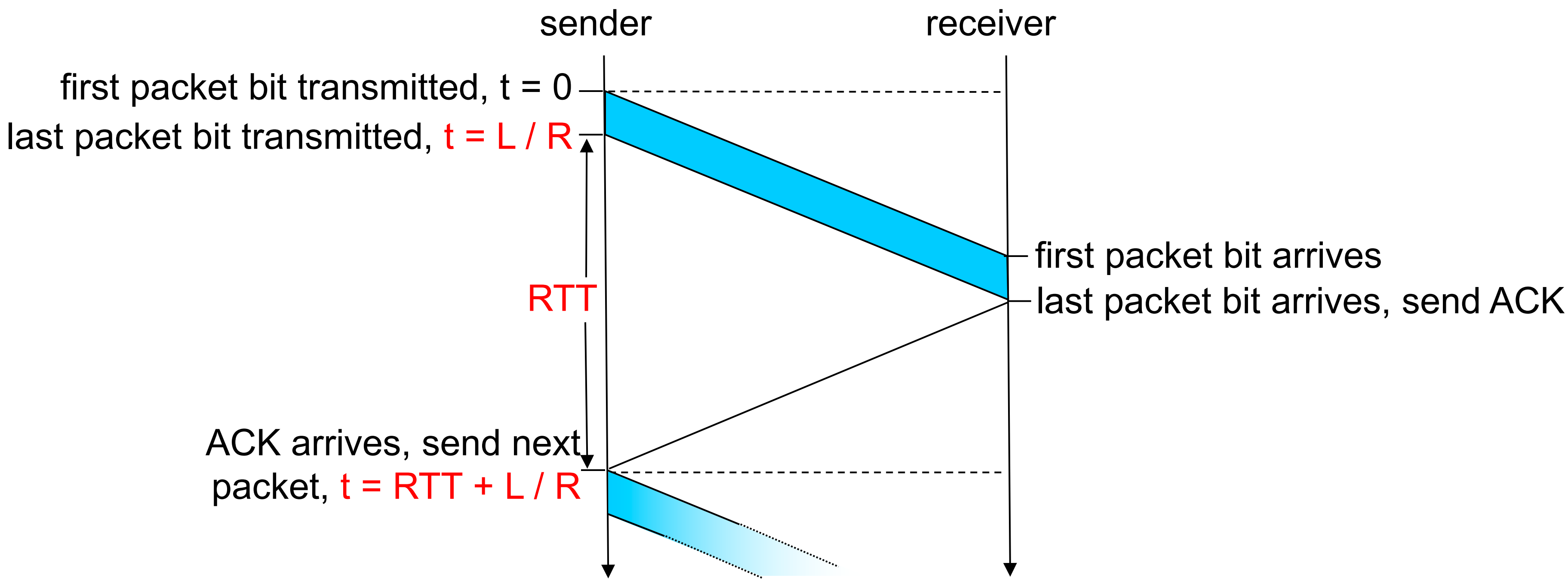


State consistency:

When receiver's state is waiting n, the state of the sender is either sending for n-1 or sending for n

When sender's state is sending for n, receiver's state is waiting for n or n + 1

rdt3.0: Stop-and-Wait Performance



What is U_{sender}: utilization – fraction of time link busy sending?

Assume: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

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

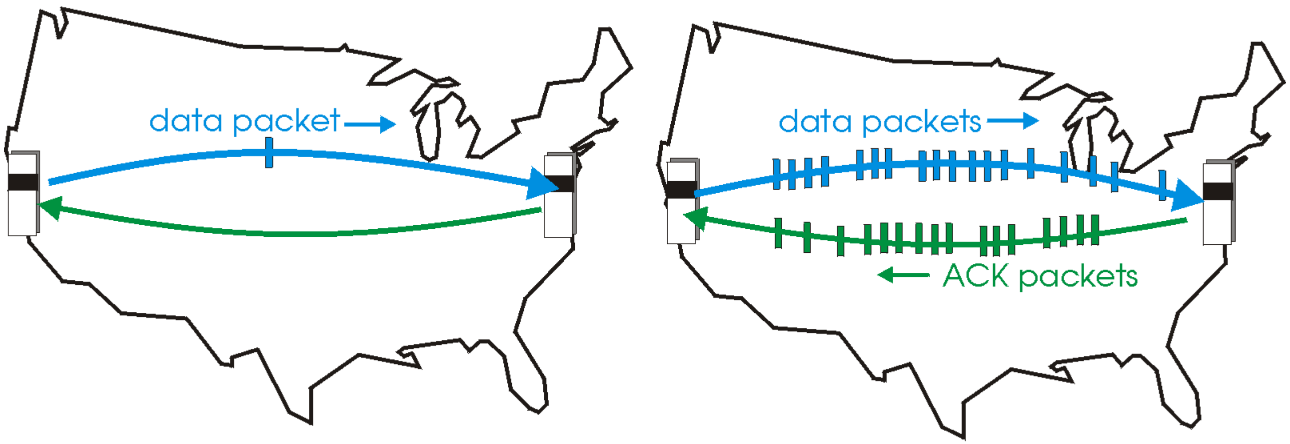
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources !

Sliding Window Protocols: Pipelining

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

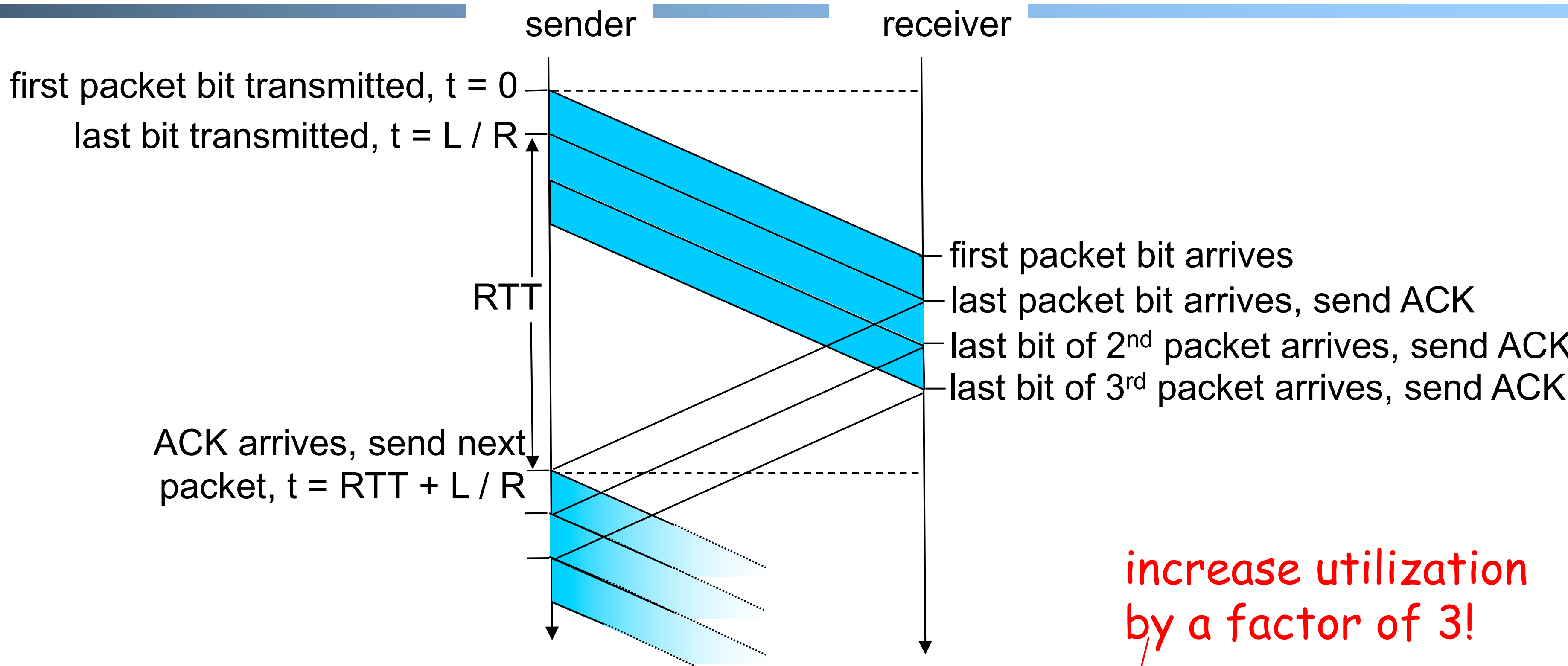
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

Pipelining: Increased Utilization



increase utilization by a factor of 3!

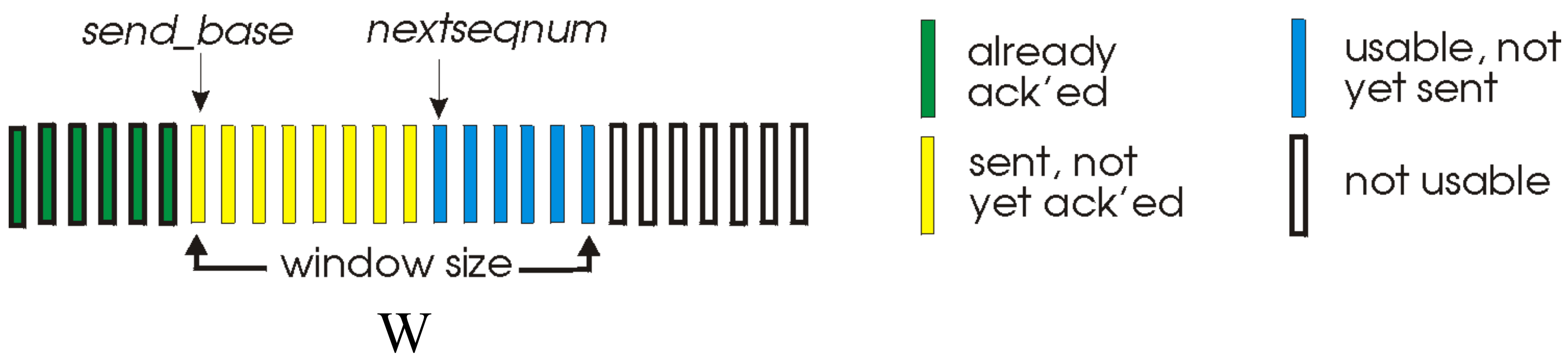
$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Question: a rule-of-thumb window size?

Realizing Sliding Window: Go-Back-n

Sender:

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



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - note: ACK(n) could mean two things: I have received upto and include n, or I am waiting for n
- timer for the packet at base
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

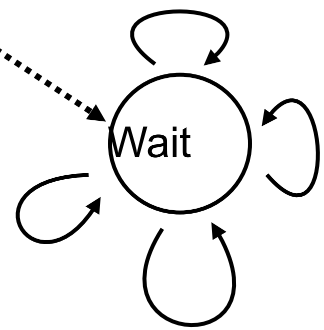
GBN: Sender FSM

rdt_send(data)

```
if (nextseqnum < base+W) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum) start_timer
    nextseqnum++
 } else
    block sender
```

Λ

base=1
nextseqnum=1



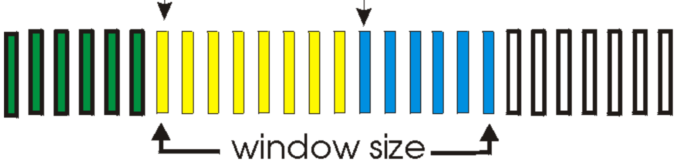
timeout

start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

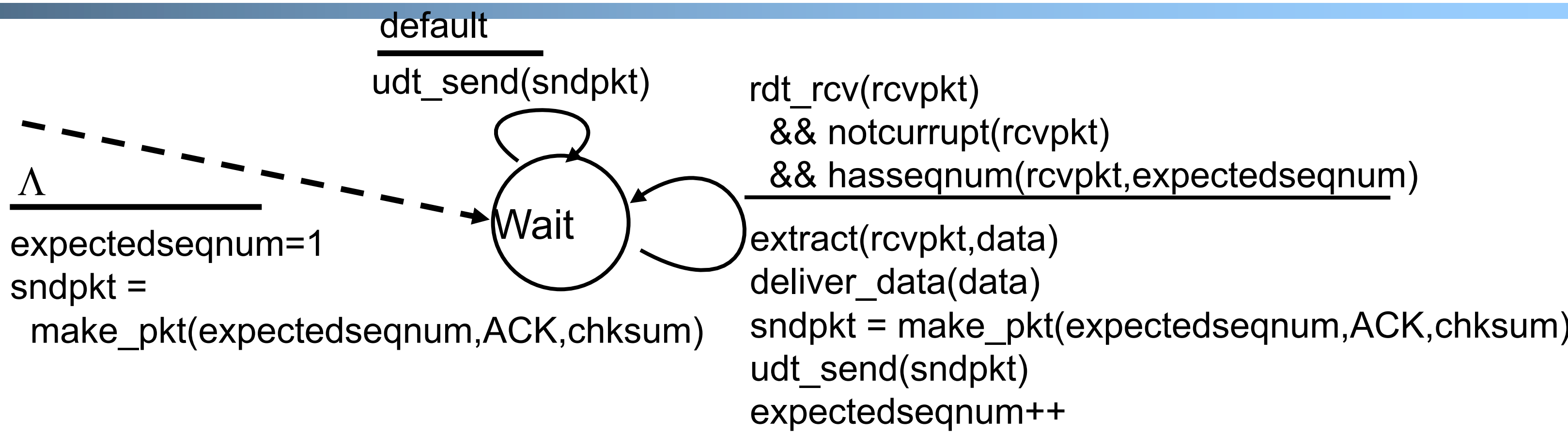
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

```
if (new packets ACKed) {
   advance base;
   if (more packets waiting)
       send more packets
}
if (base == nextseqnum)
  stop_timer
else
  start_timer for the packet at new base
```



GBN: Receiver FSM

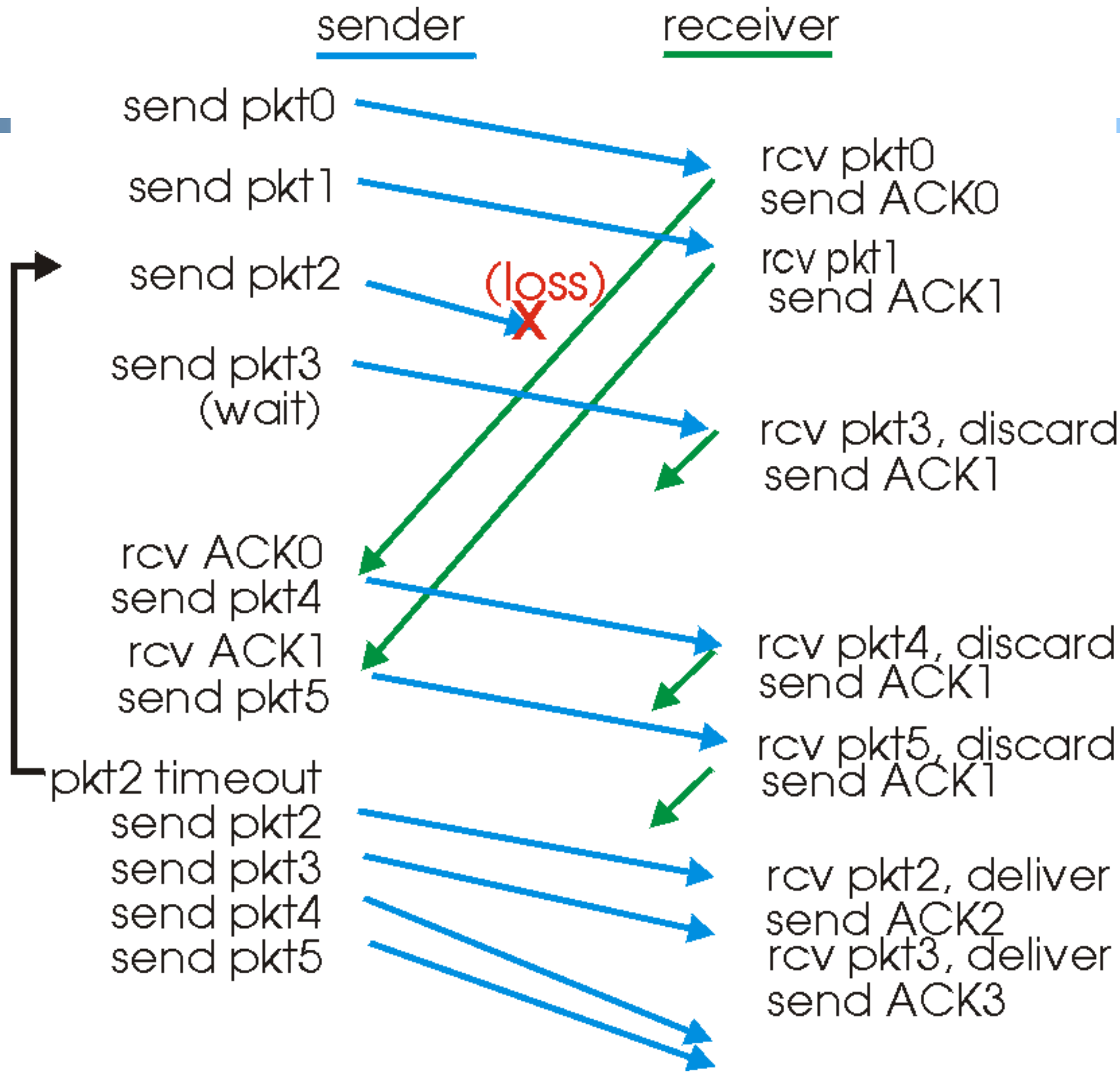


Only state: **`expectedseqnum`**

- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - re-ACK pkt with highest in-order seq #
  - may generate duplicate ACKs

GBN in Action

window size = 4



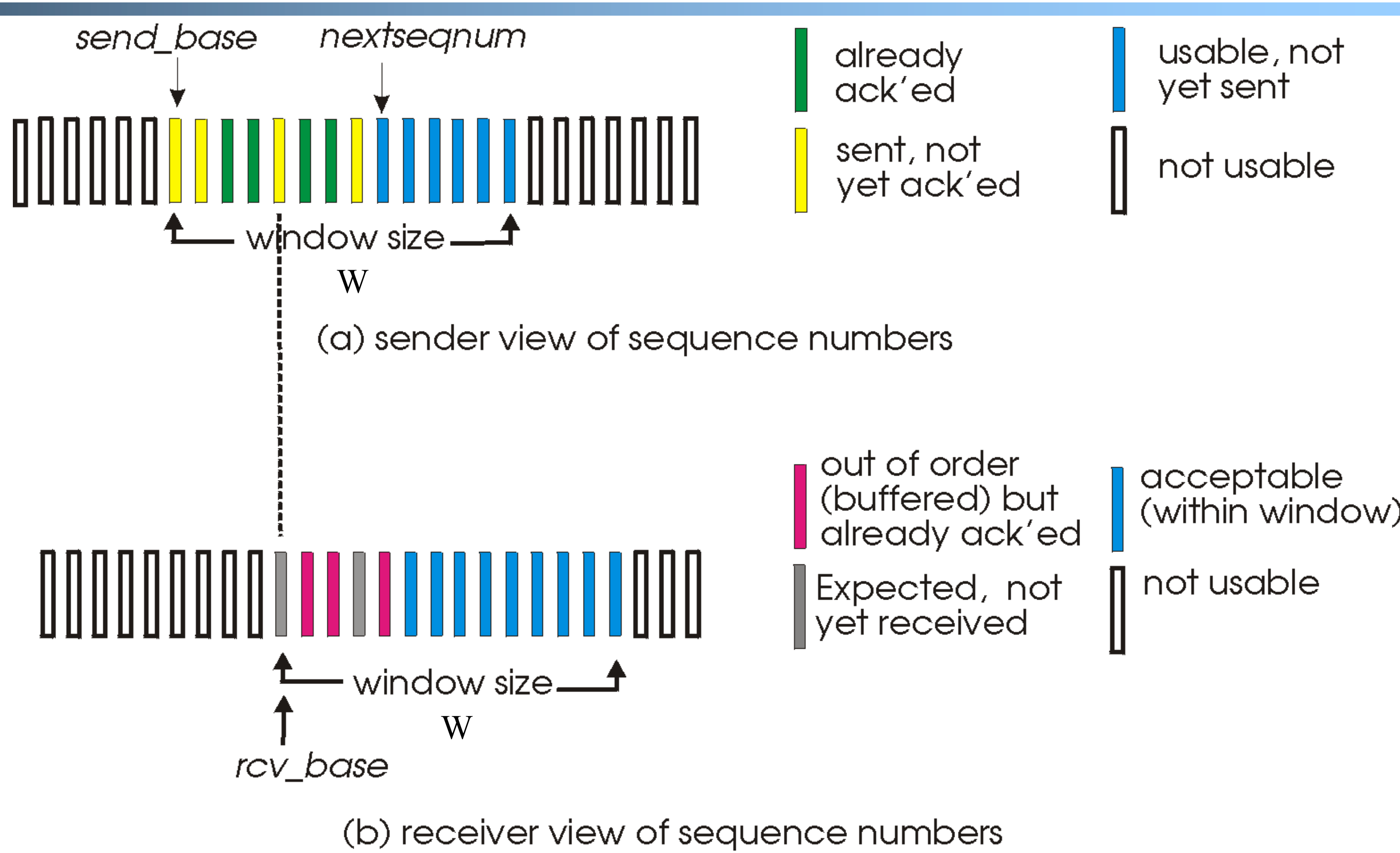
Analysis: Efficiency of Go-Back-n

- Assume window size W
- Assume each packet is lost with probability p
- On average, how many packets do we send for each data packet received?

Selective Repeat

- Sender window
  - Window size W: W consecutive unACKed seq #'s
- Receiver *individually* acknowledges correctly received pkts
  - buffers out-of-order pkts, for eventual in-order delivery to upper layer
  - ACK(n) means received packet with seq# n only
  - question: buffer size at receiver?
- Sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt

Selective Repeat: Sender, Receiver Windows



Selective Repeat

sender

data from above :

- unACKed packets is less than window size W, send; otherwise block app.

timeout(n):

- resend pkt n, restart timer

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

- mark pkt n as received
- update sendbase to the first packet unACKed

receiver

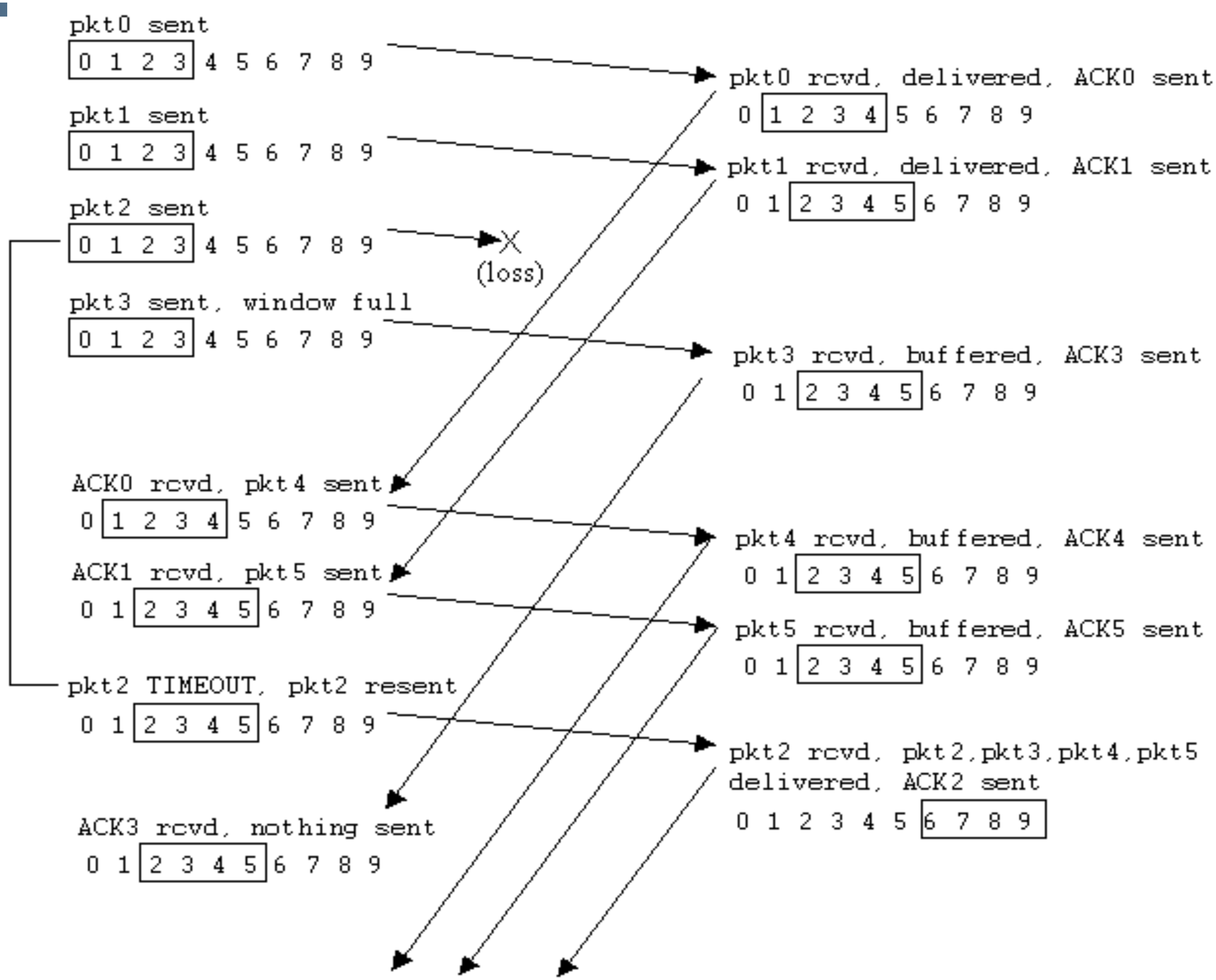
pkt n in [rcvbase, rcvbase+W-1]

- send ACK(n)
- if (out-of-order)
  mark and buffer pkt n
  else /*in-order*/
  deliver any in-order packets

otherwise:

- ignore

Selective Repeat in Action



Discussion: Efficiency of Selective Repeat

- Assume window size W

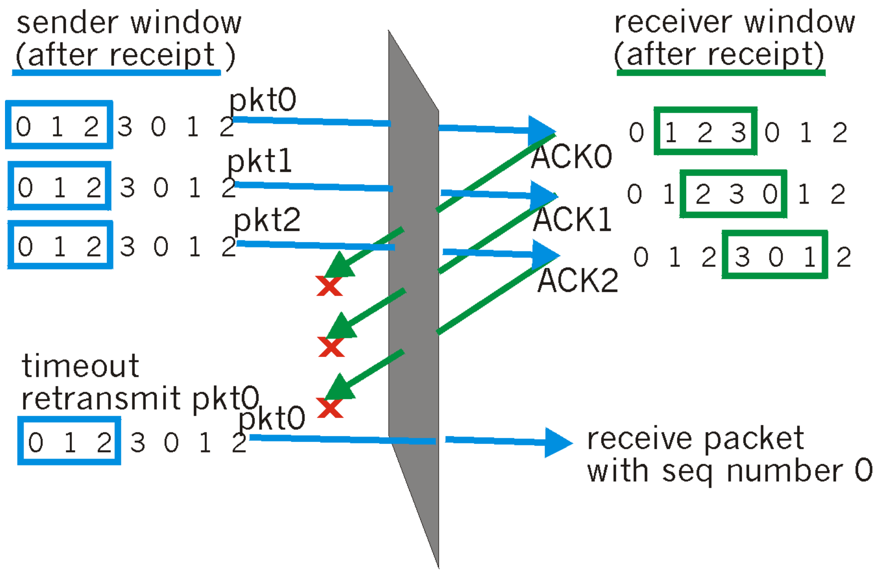
- Assume each packet is lost with probability p

- On average, how many packets do we send for each data packet received?

Selective Repeat: Seq# Size and Window Size

Example:

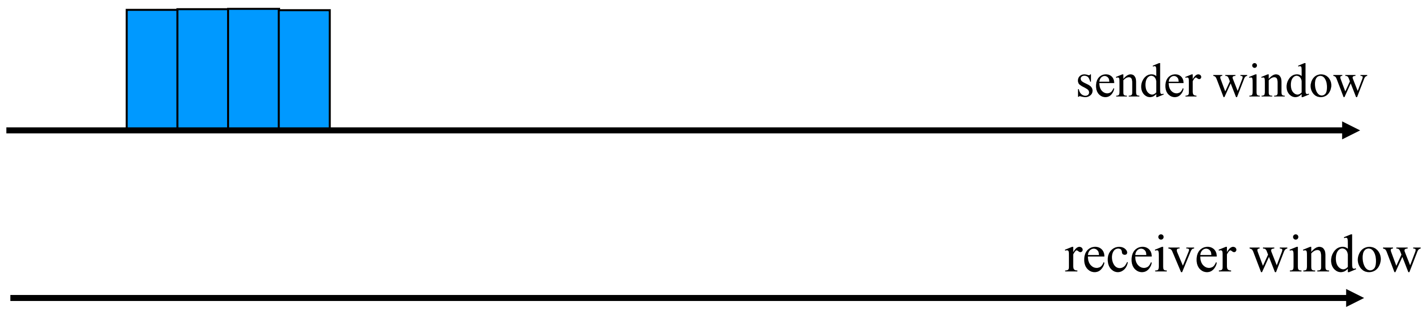
- seq #'s (2 bits): 0, 1, 2, 3
- window size=3



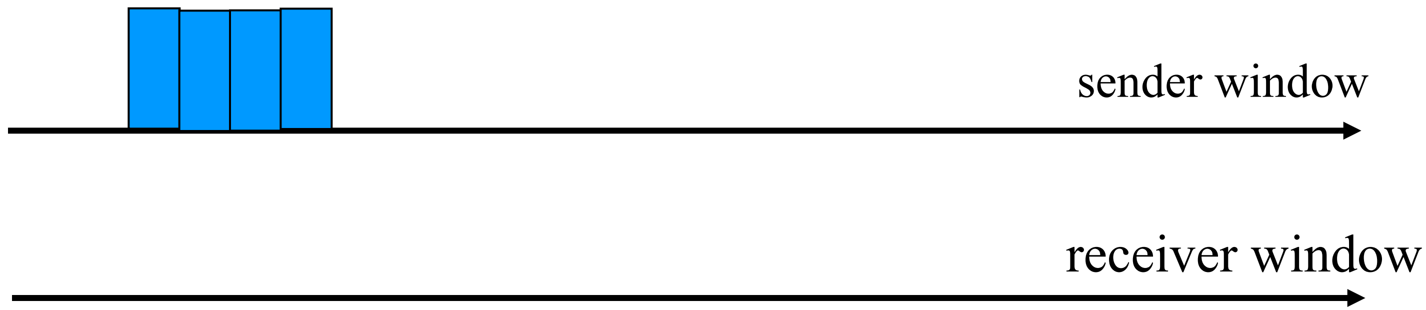
- Error: incorrectly passes duplicate data as new.

State Invariant: Window Location

- Go-back-n (GBN)



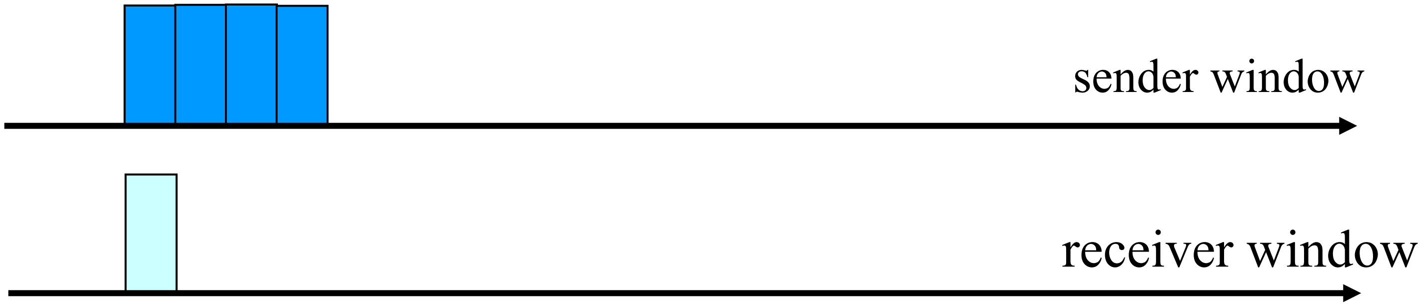
- Selective repeat (SR)



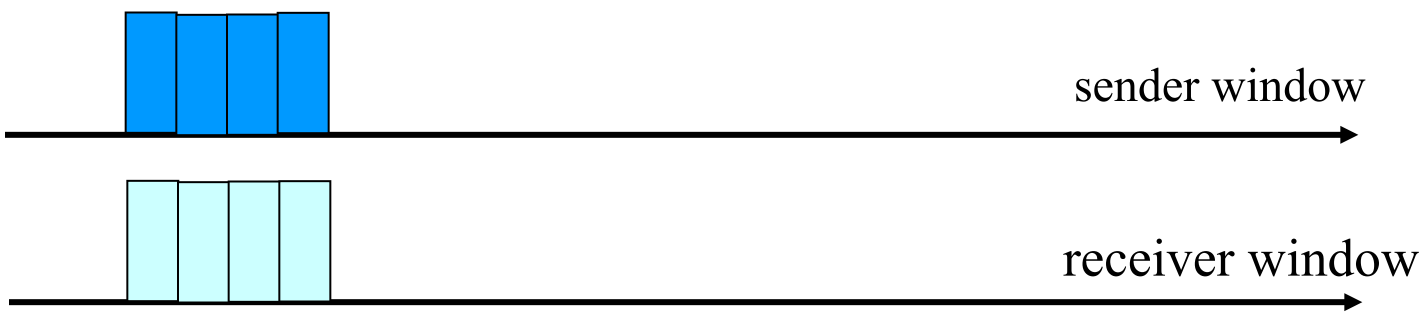
Window Location

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

- Go-back-n (GBN)



- Selective repeat (SR)

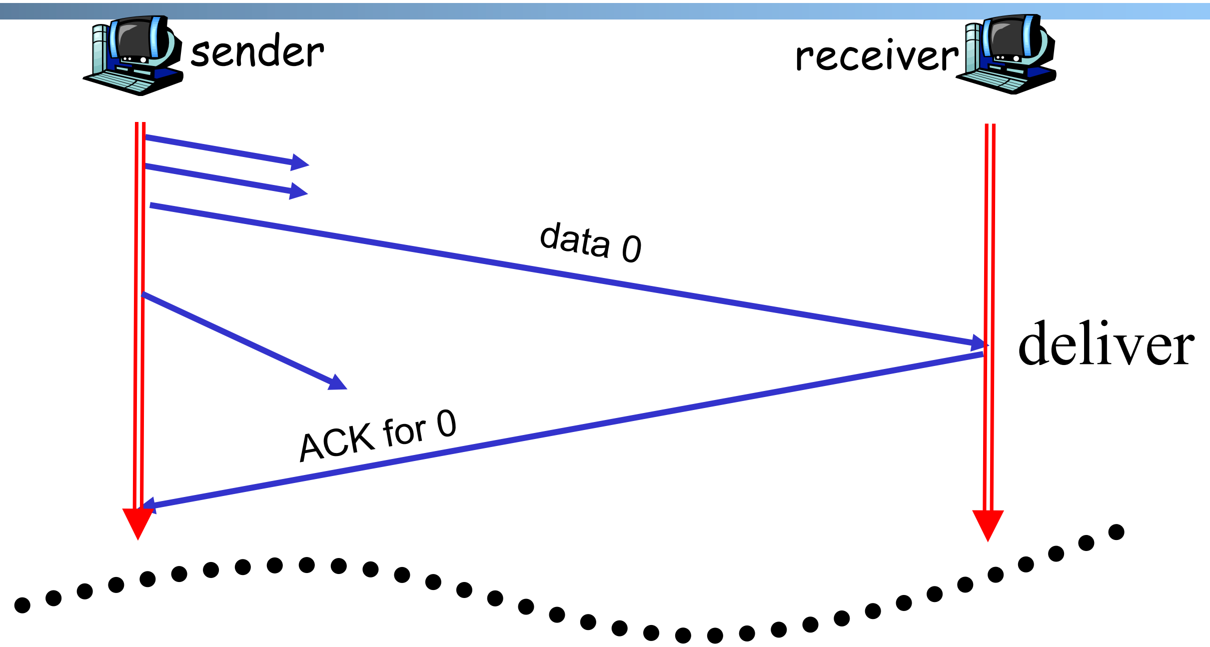


Sliding Window Protocols: Go-back-n and Selective Repeat

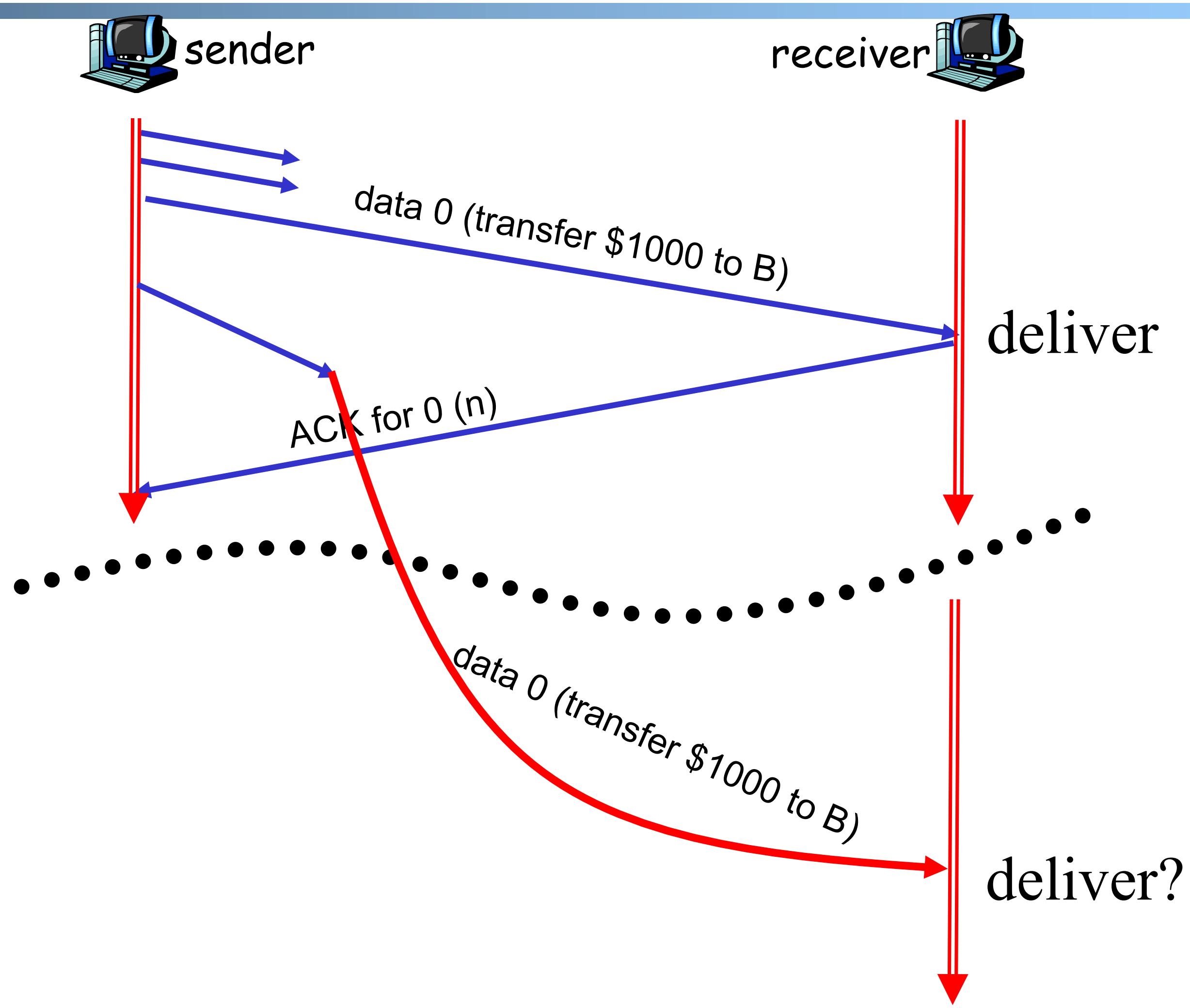
	Go-back-n	Selective Repeat
data bandwidth: sender to receiver (avg. number of times a pkt is transmitted)	Less efficient $\frac{1-p+pw}{1-p}$	More efficient $\frac{1}{1-p}$
ACK bandwidth (receiver to sender)	More efficient	Less efficient
Relationship between M (the number of seq#) and W (window size)	M > W	M ≥ 2W
Buffer size at receiver	1	W
Complexity	Simpler	More complex

p: the loss rate of a packet; M: number of seq# (e.g., 3 bit M = 8); W: window size

Question: What is Initial Seq# and When to Accept First Packet?



Question: What is Initial Seq# and When to Accept First Packet?

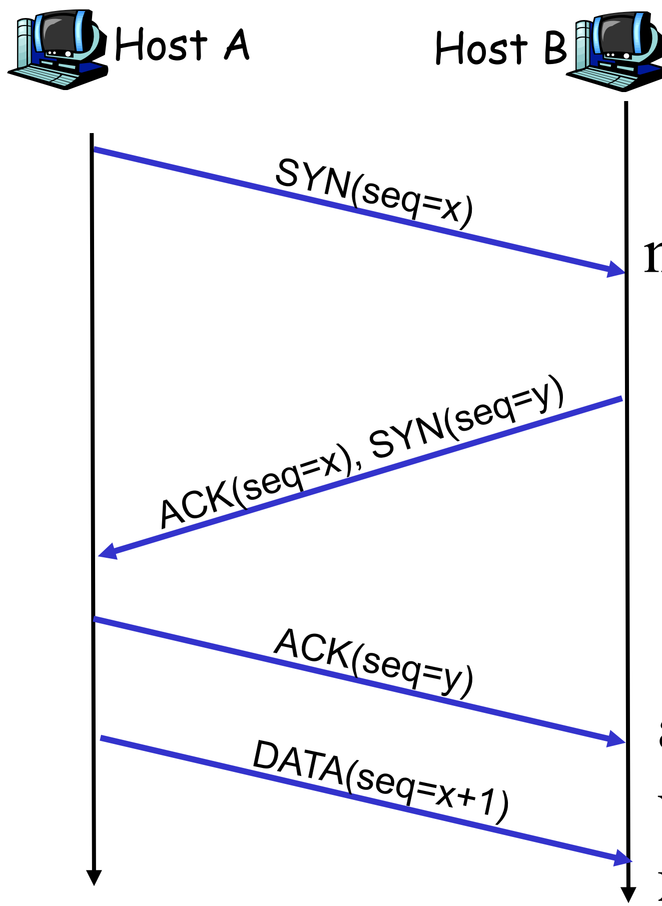


Discussion: Condition for receiver to deliver first data?

Outline

- Admin and recap
- Reliable data transfer
  - perfect channel
  - channel with bit errors
  - channel with bit errors and losses
  - sliding window: reliability with throughput
  - connection management

Three Way Handshake (TWH) [Tomlinson 1975]



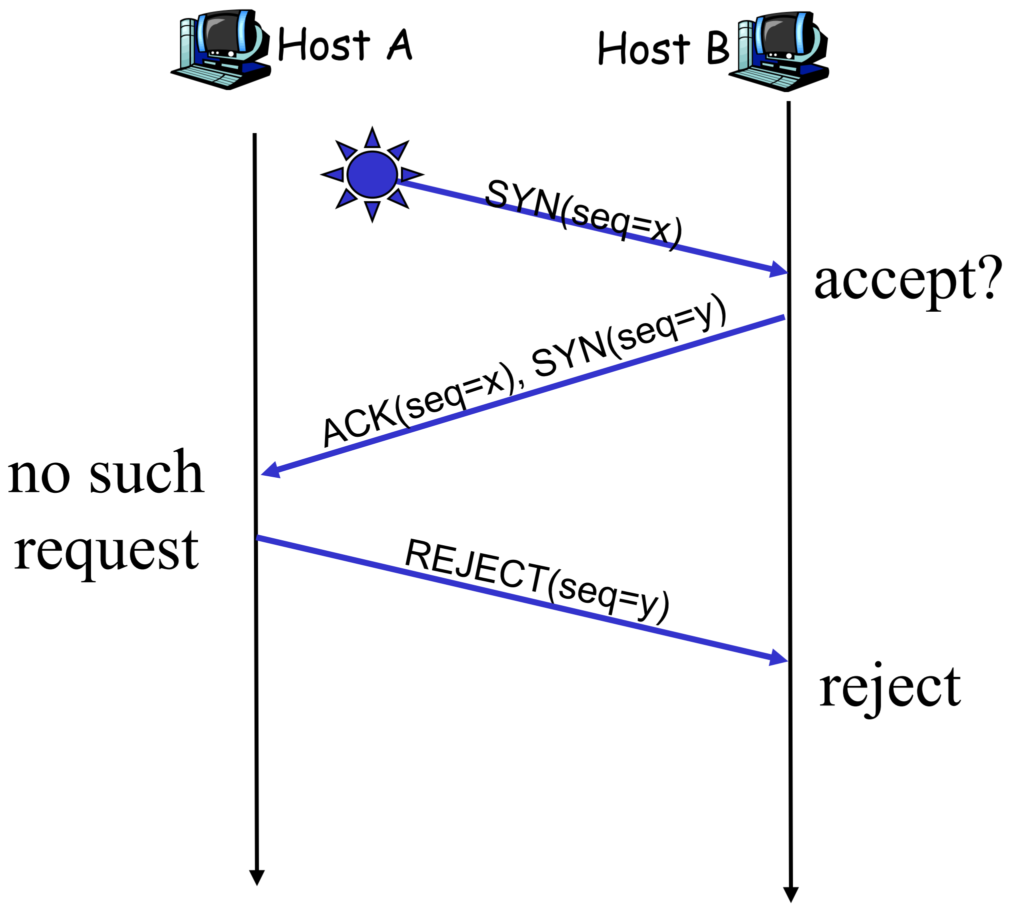
notify initial seq#. Accept?

think of y as a challenge for freshness

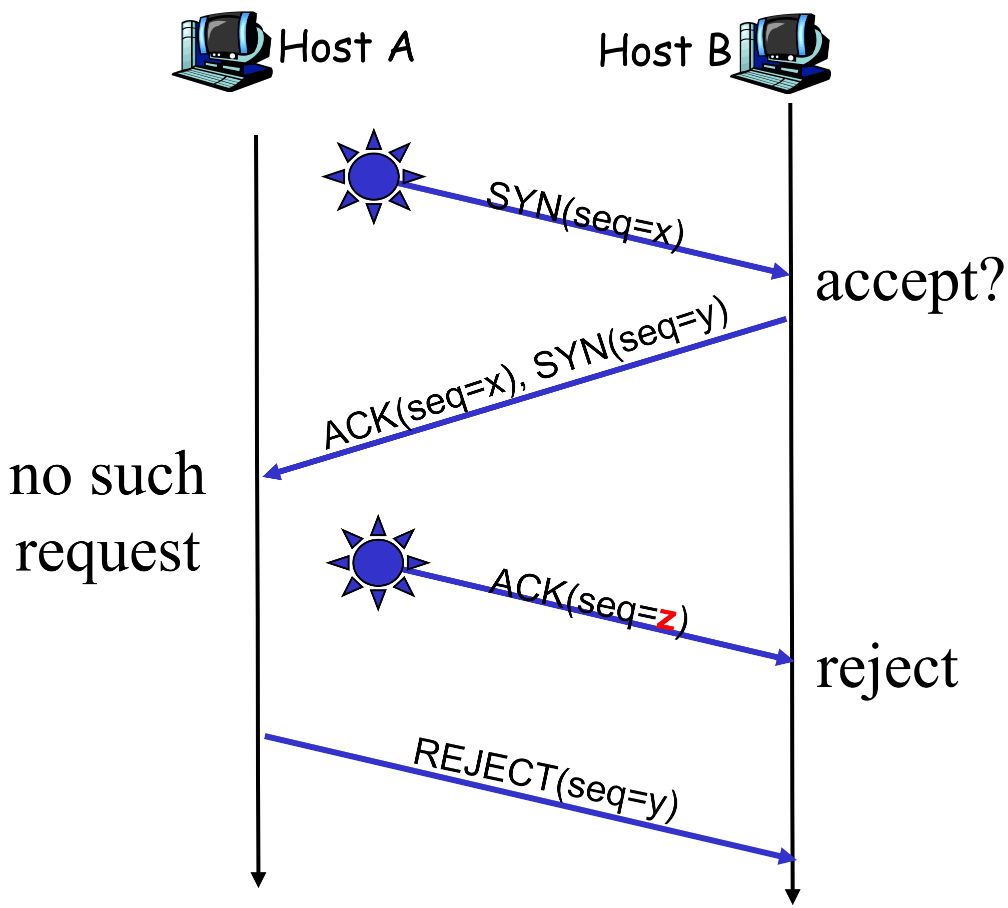
accept data only after verified y is bounced back
x is the init. seq

SYN: indicates connection setup

Scenarios with Duplicate Request/SYN Attack



Scenarios with Duplicate Request/SYN Attack

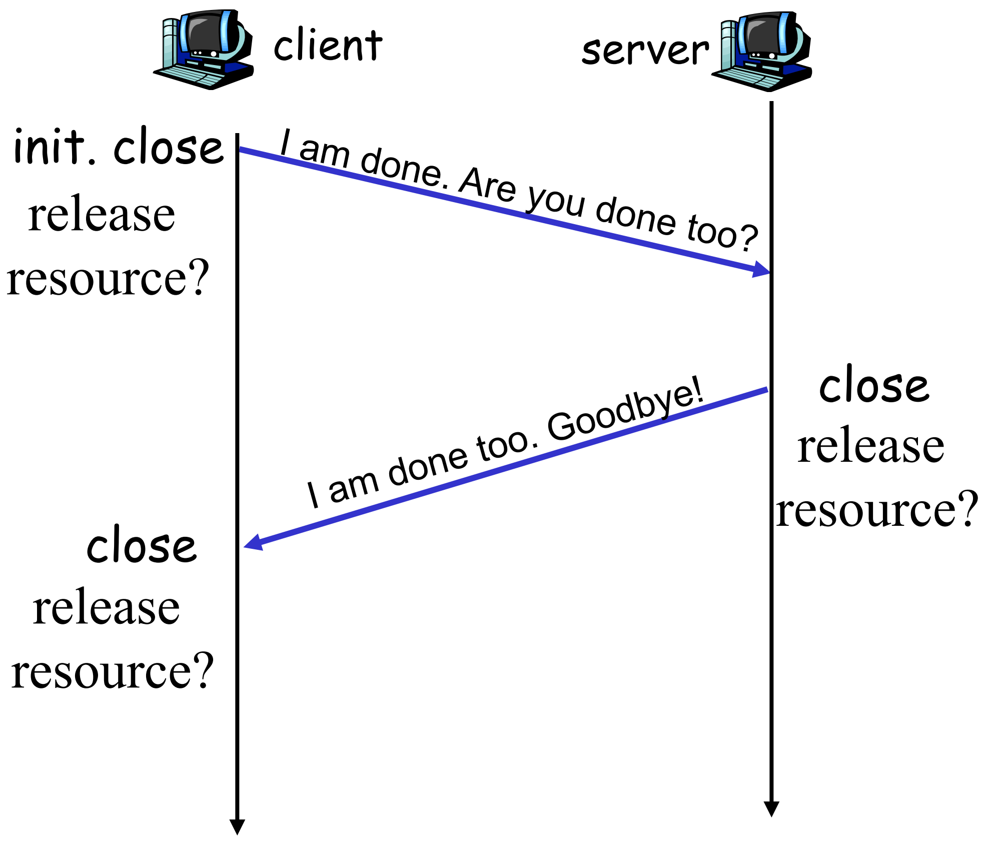


Make "Challenge y" Robust

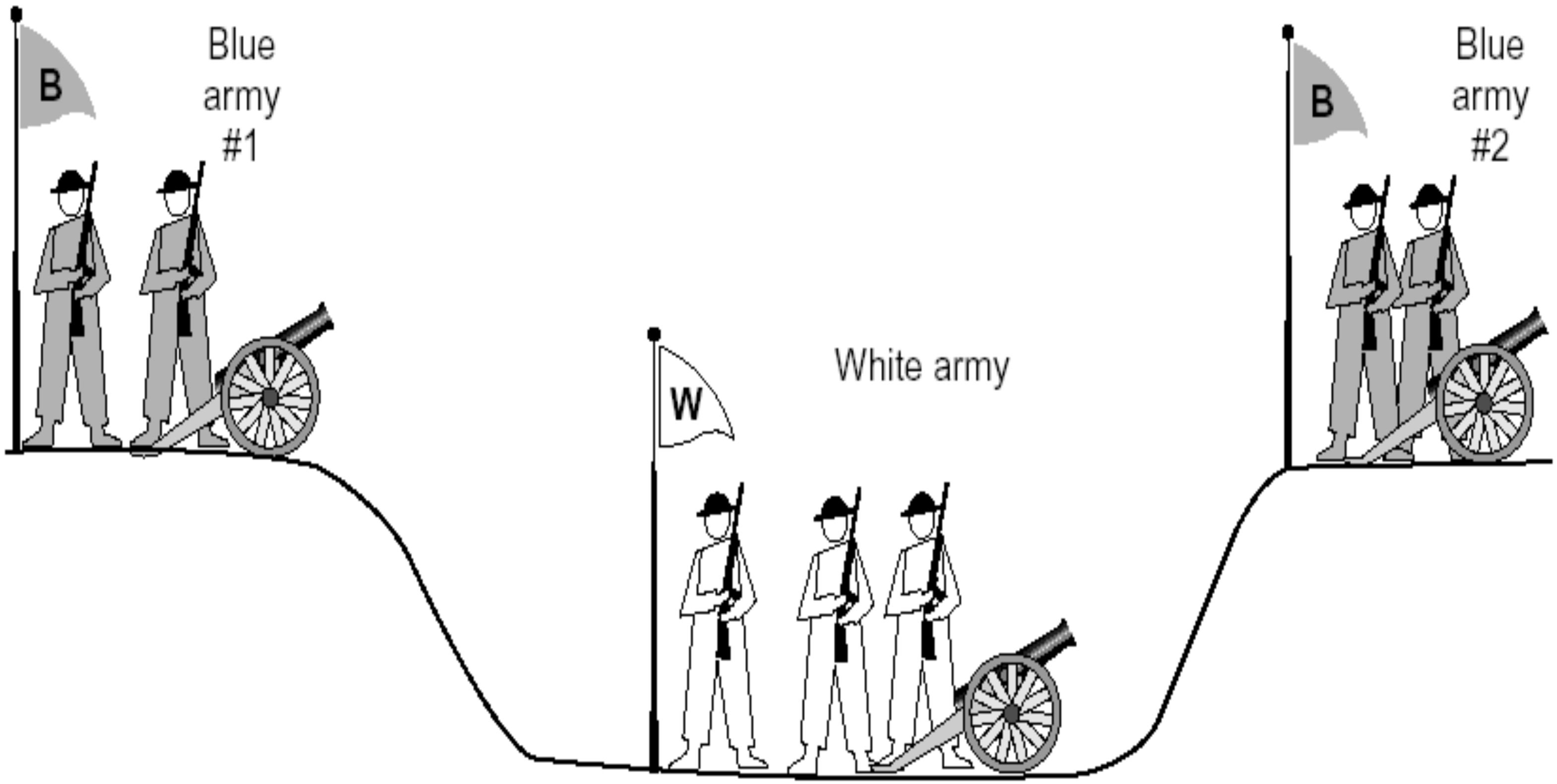
- To avoid that "SYNC ACK y" comes from reordering and duplication
  - for each connection (sender-receiver pair), ensuring that two identically numbered packets are never outstanding at the same time
    - network bounds the life time of each packet
    - a sender will not reuse a seq# before it is sure that all packets with the seq# are purged from the network
    - seq. number space should be large enough to not limit transmission rate

Connection Close

- Why connection close?
  - so that each side can release resource and remove state about the connection (do not want dangling socket)



General Case: The Two-Army Problem



The gray (blue) armies need to agree on whether or not they will attack the white army. They achieve agreement by sending messengers to the other side. If they both agree, attack; otherwise, no. Note that a messenger can be captured!

Discussion: Potential approaches to close state?

Four Way Teardown