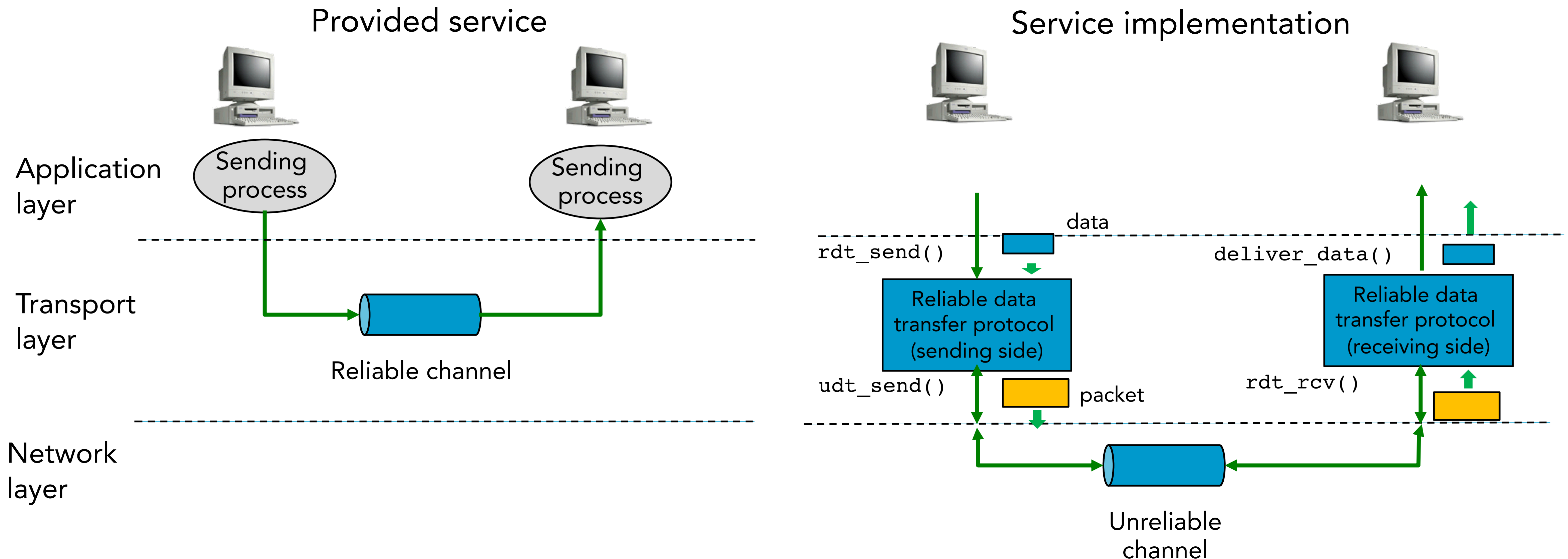
Principles of Reliable Transport

To do ...

- A reliable transport, piece by piece
- ACKs, timers, retransmissions and sequence numbers

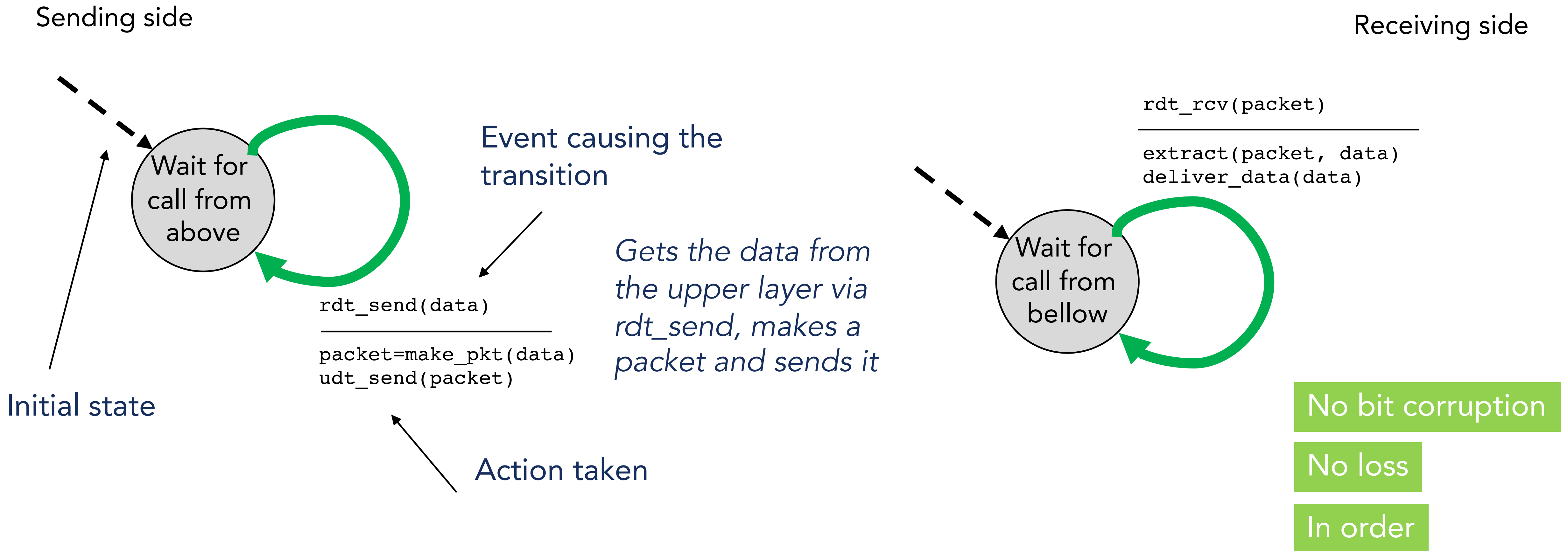
Reliable Data Transfer

- What do you get from a reliable data transfer
  - *Nothing corrupted, nothing lost* and *all in order*
- A framework for discussion



Trivial, RDT Over a Perfectly Reliable Channel

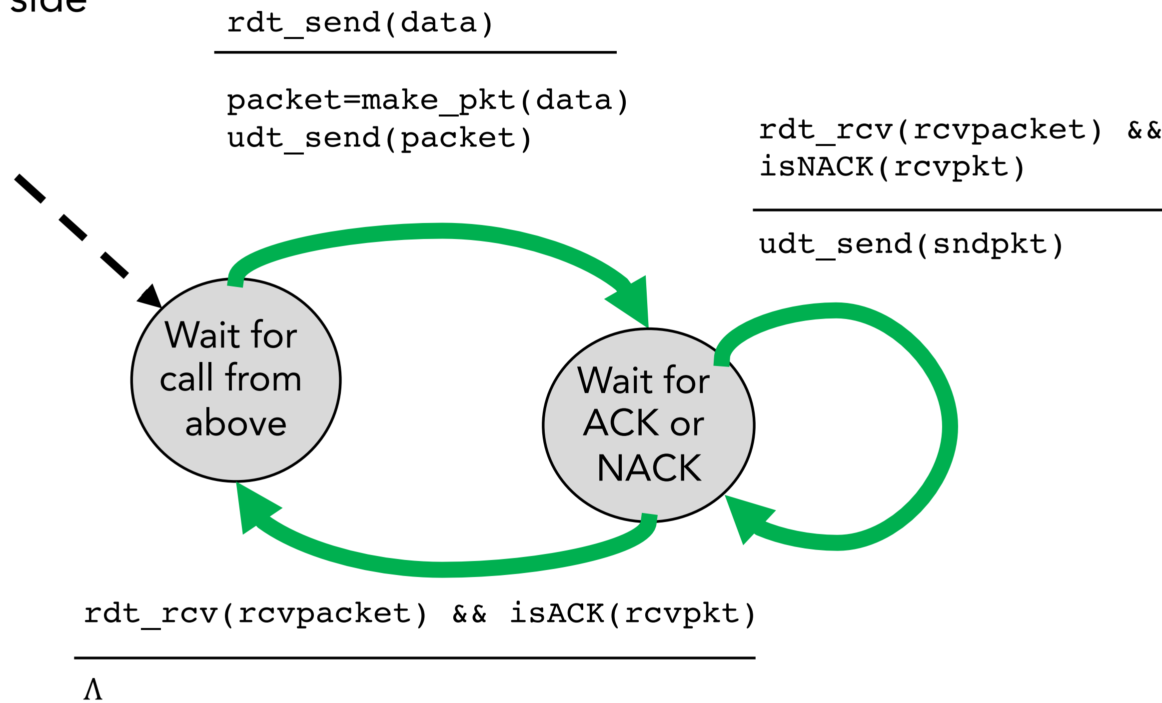
- A finite state machine (FSM) definition

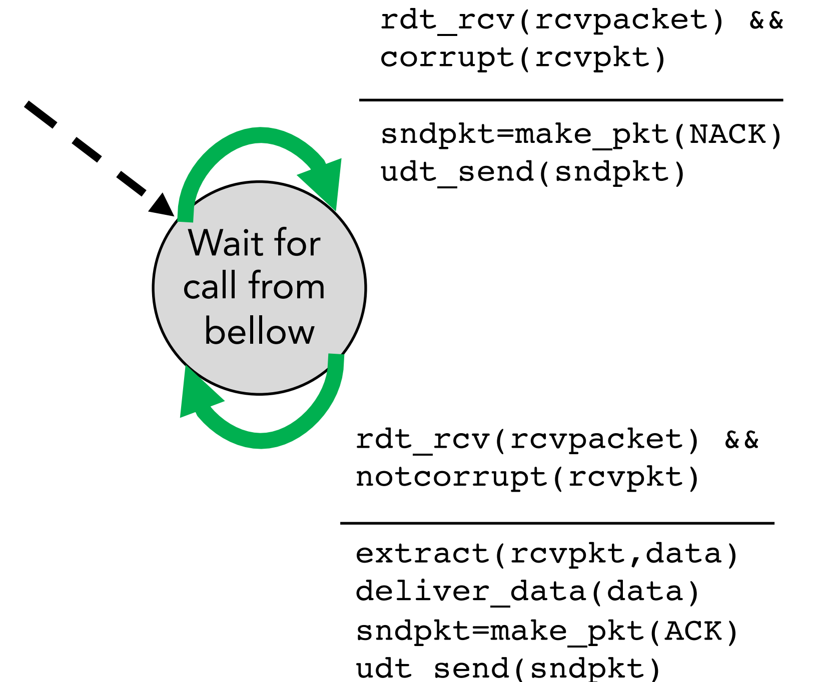


RDT Over a Channel with Bit Errors

- Assume there is no loss for now …
- First, how do people handle a call over a noisy connection?
  - "OK / yeah / …" – *Positive ACK*
  - "Sorry? / Please repeat that / …" – *Negative ACK*
  - Automatic Repeat reQuest (ARQ) protocols
- Three things you need in an ARQ
  - Error detection – requires some extra bits, besides the data itself
  - Receiver feedback – no other way for the sender to know; ACKs and NACKs are examples of this feedback
  - Retransmission

RDT Over a Channel with Bit Errors – rdt2.0





Bit corruption

No loss

In order

When in ‘Wait for ACK or NACK’, sender can’t get any more data from the upper layers - **Stop-and-wait**

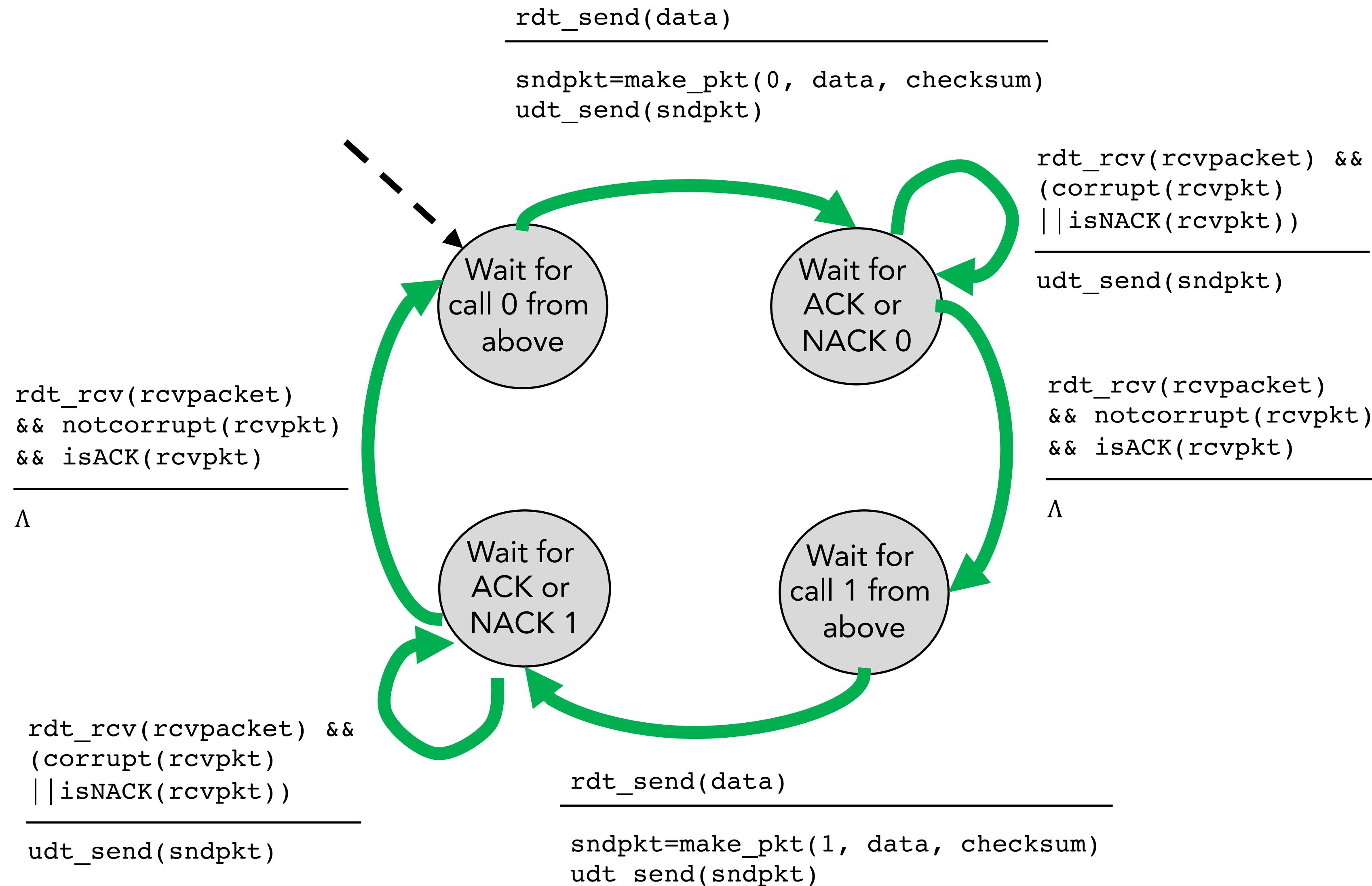
rdt2.0 has a fatal flaw!

- What happens if ACKs/NAKs are corrupted?
  - Sender doesn't know what happened at the receiver!
  - How about retransmit if received a garbled ACK/NACK? Duplicates?
- Handling duplicates
  - Sender retransmits current packet if ACK/NAK corrupted
  - Sender adds sequence number to each packet
  - Receiver discards (doesn't deliver up) duplicate packet

- With a "stop-and-wait" protocol, one bit seq # is enough
  - To tell between a resend and a new packet
  - NACKs/ACKs don't need seq # since there's no loss (so, it must be about the most recently sent packet

rdt2.1

Rdt2.1 sender



Rdt2.1 receiver

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

---

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

rdt_rcv(rcvpacket) && corrupt(rcvpkt)

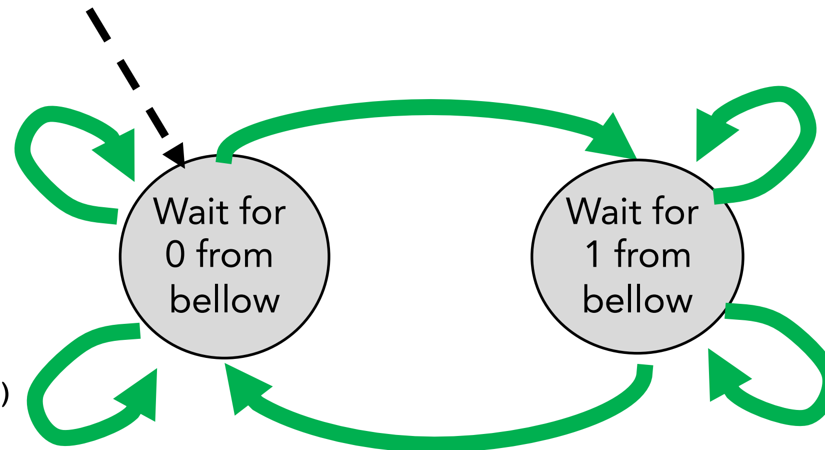
---

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

rdt_rcv(rcvpacket) && corrupt(rcvpkt)

---

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



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

---

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

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

---

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

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

---

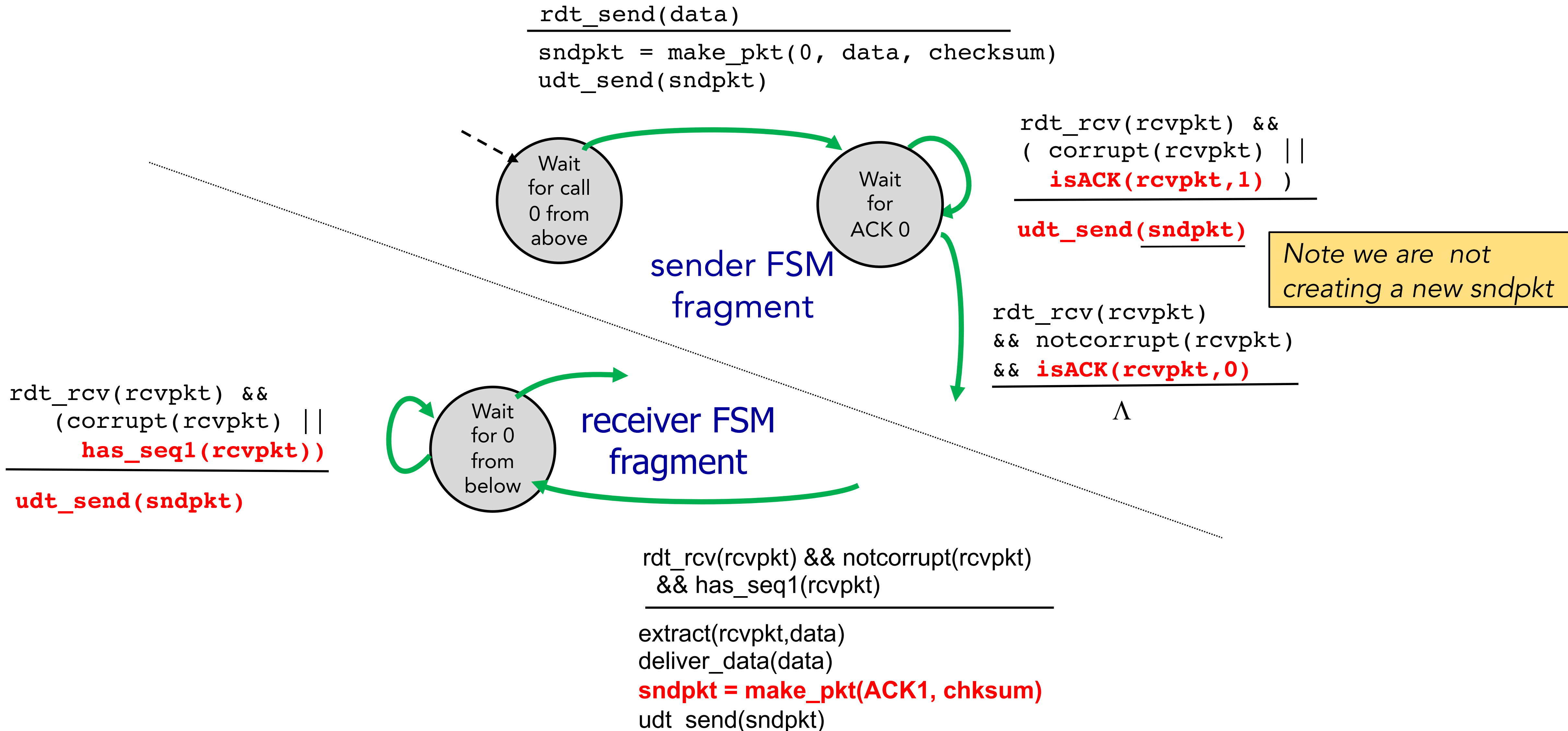
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK,checksum)
udt_send(sndpkt)

NACKs and ACKs

- rdt2.1 uses positive and negative acknowledgments
  - When a corrupted packet arrives, send a NACK
  - When an out of order packet (previous one) arrives, send an ACK
- Replace NACKs with an ACK for last correctly received packet
  - "This is the last thing I understood of what you said" → receiver did not correctly receive the packet following the one being ACKed twice
  - Of course now the ACK needs to include the seq # of the packet ACKed

rdt2.2

rdt2.2: sender, receiver fragments



RDT Over a Lossy Channel with Bit Error

Bit corruption

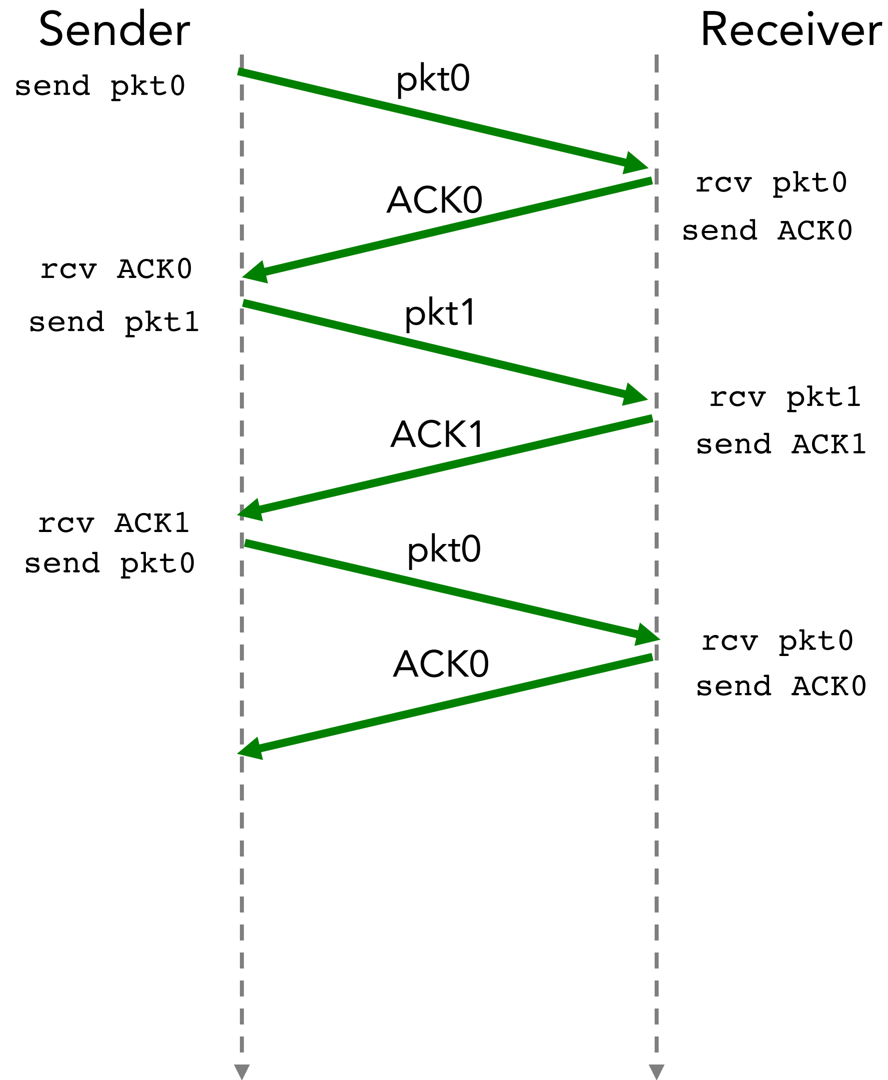
Loss

In order

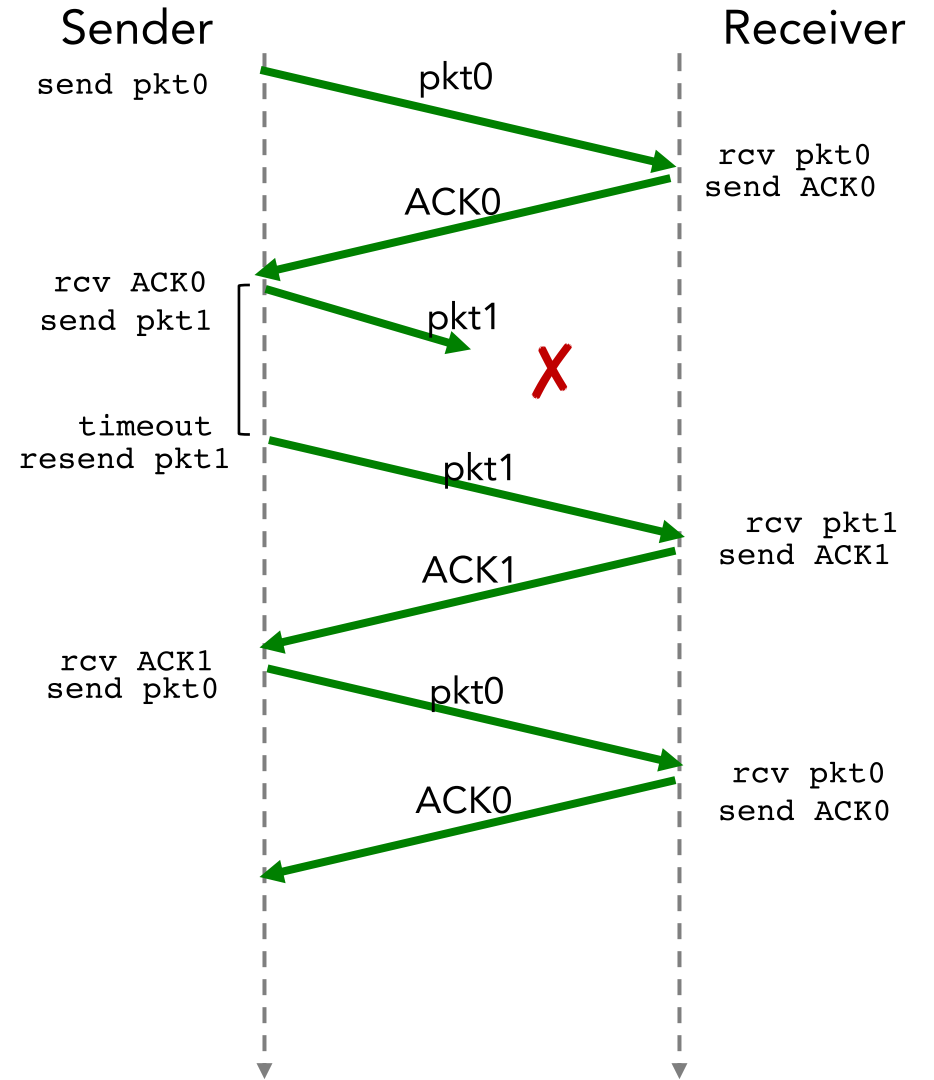
- Two additional concerns to address
  - Detecting packet loss
  - What to do in the event of packet loss
- Who needs to detect loss? We make the sender responsible
- How? Time-out, either the message got lost or the ACK did
  - Retransmit anyway
- But how long? > RTT + processing time … hard to estimate
  - And you want to recover ASAP → Duplicate packets
- Because pckt seq # alternate between 0/1 – **alternating-bit protocol**

rdt3.0

rdt3.0 in Operation

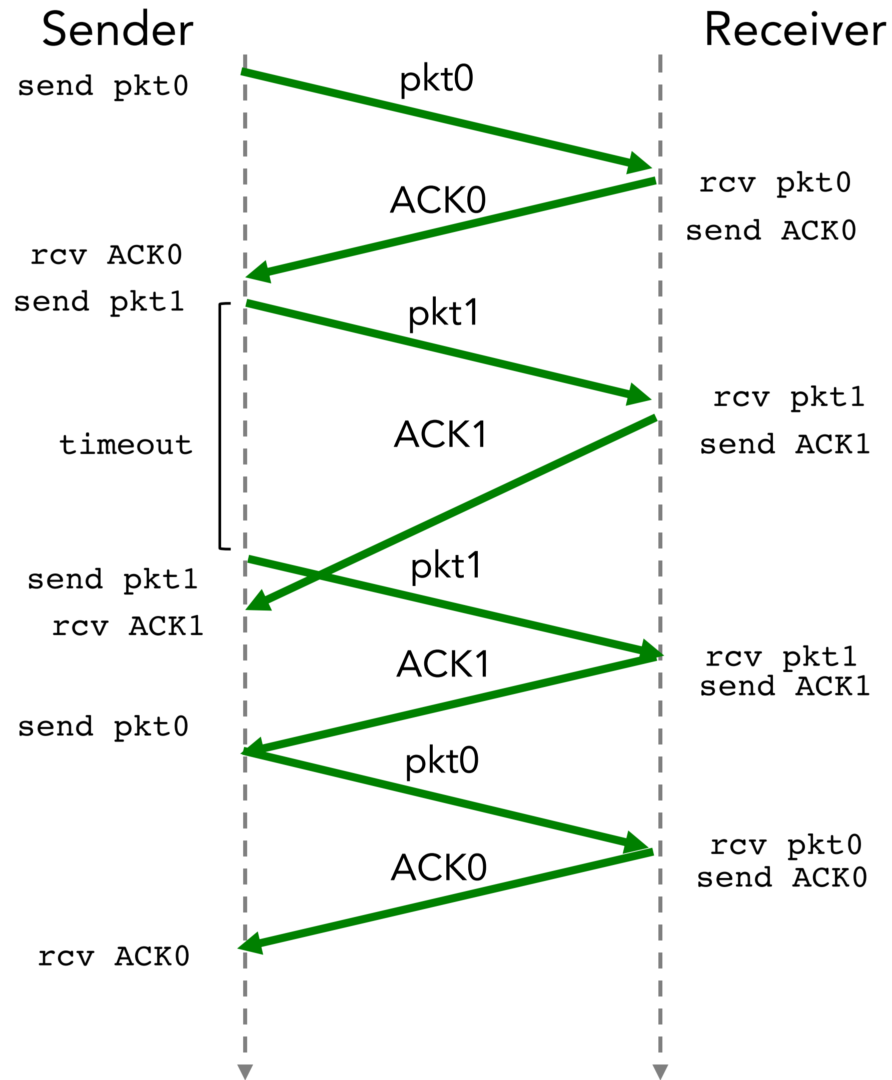


Operation with no loss

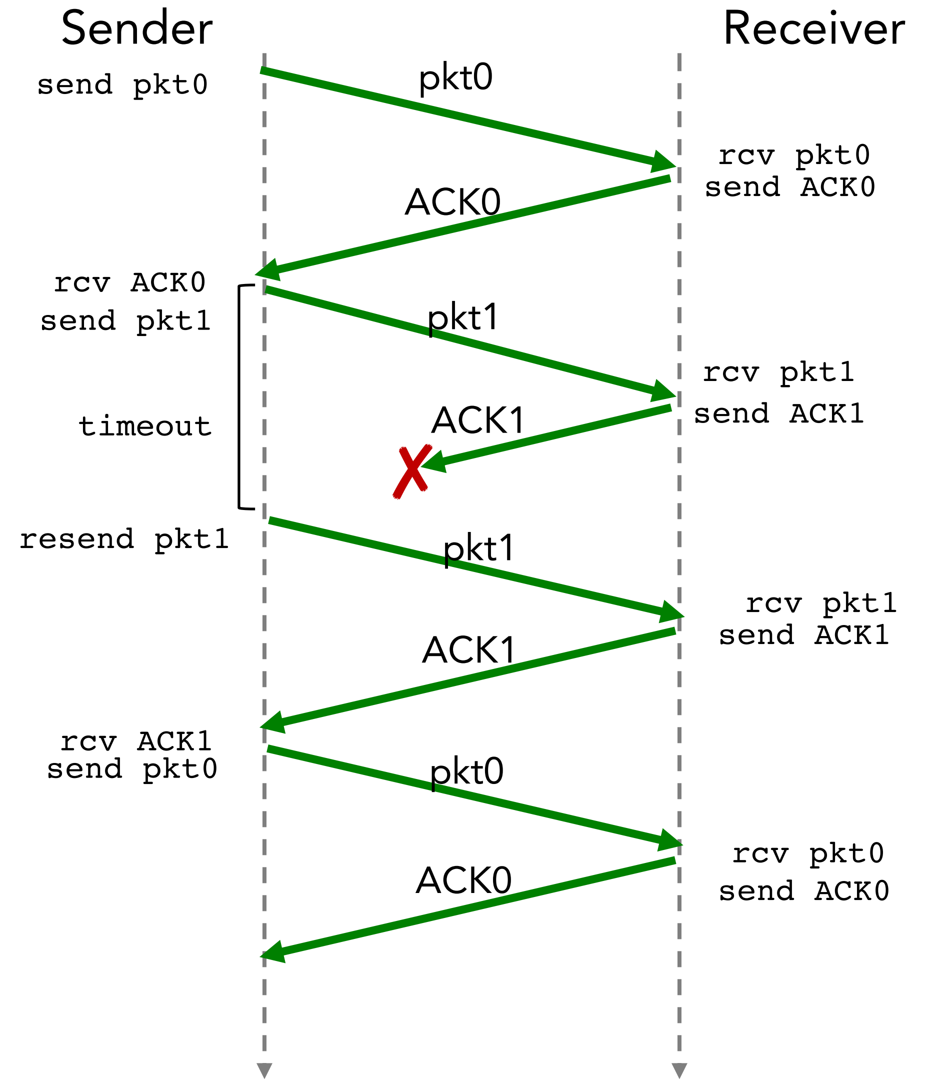


Lost packet

rdt3.0 in Operation



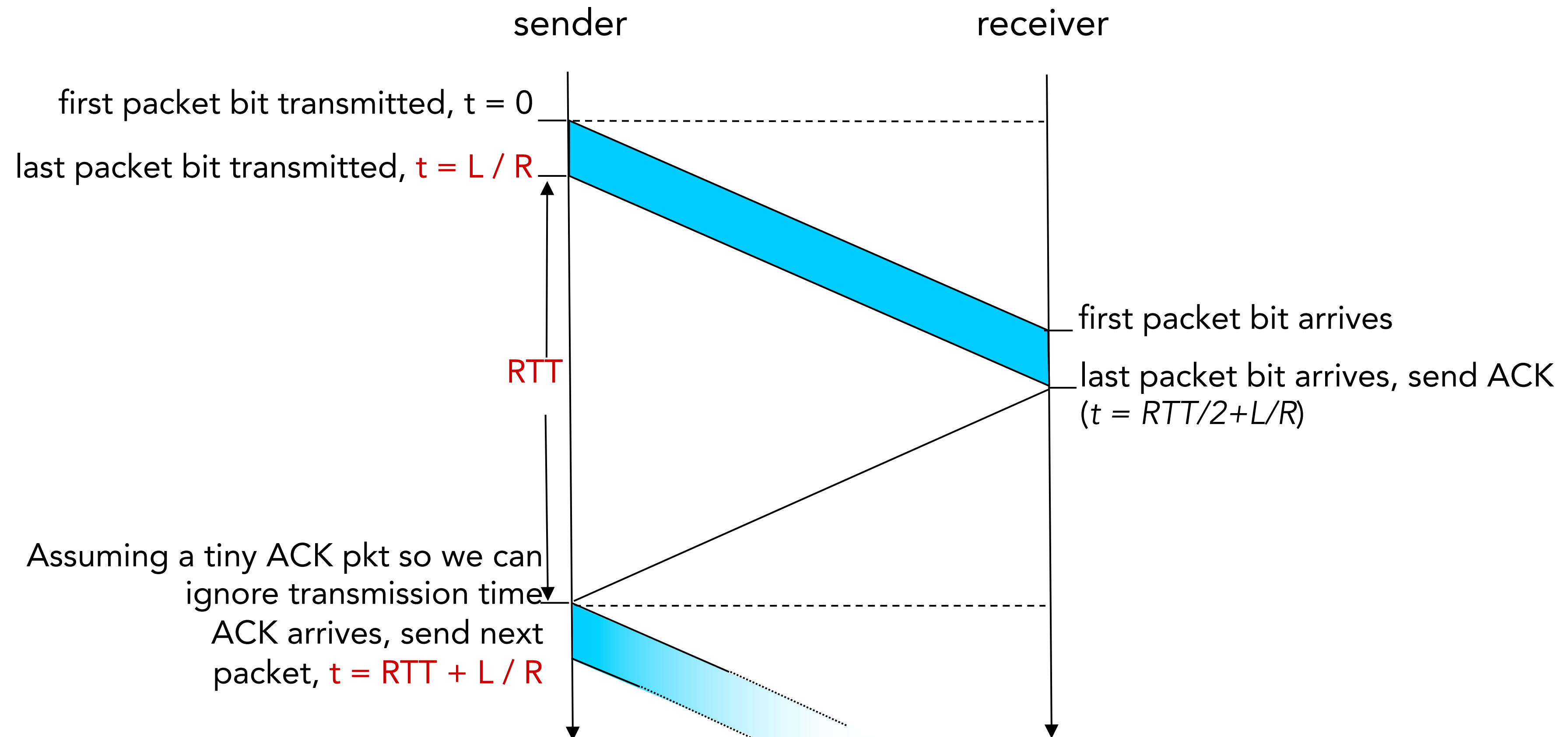
Premature timeout



Lost ACK

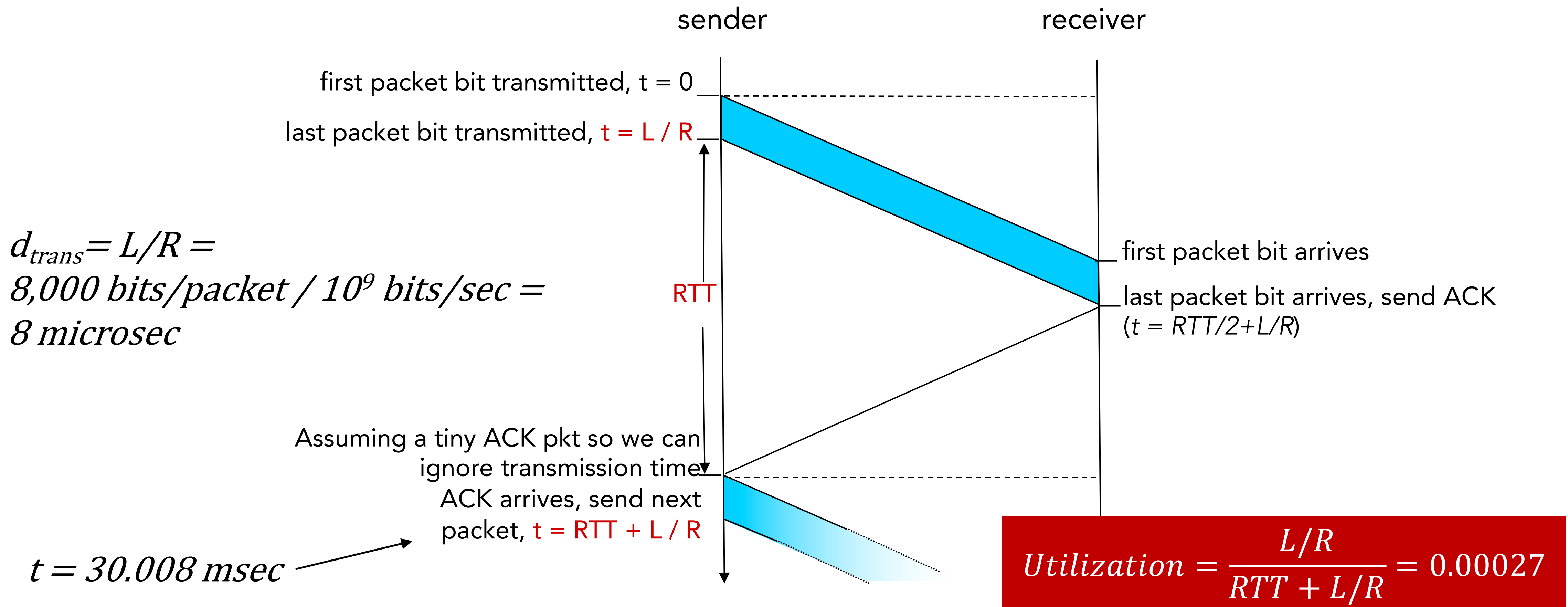
Stop-and-wait

- rdt3.0 – functionally correct but stop-and-wait … poor utilization
  - Utilization – fraction of time the sender is busy sending bits



Stop-and-wait

- Consider two host on two sides of the US (~*30 msec RTT*)
  - Transmission rate *R* (e.g., 1 Gbps) and packet size *L (e.g., 1KB)*

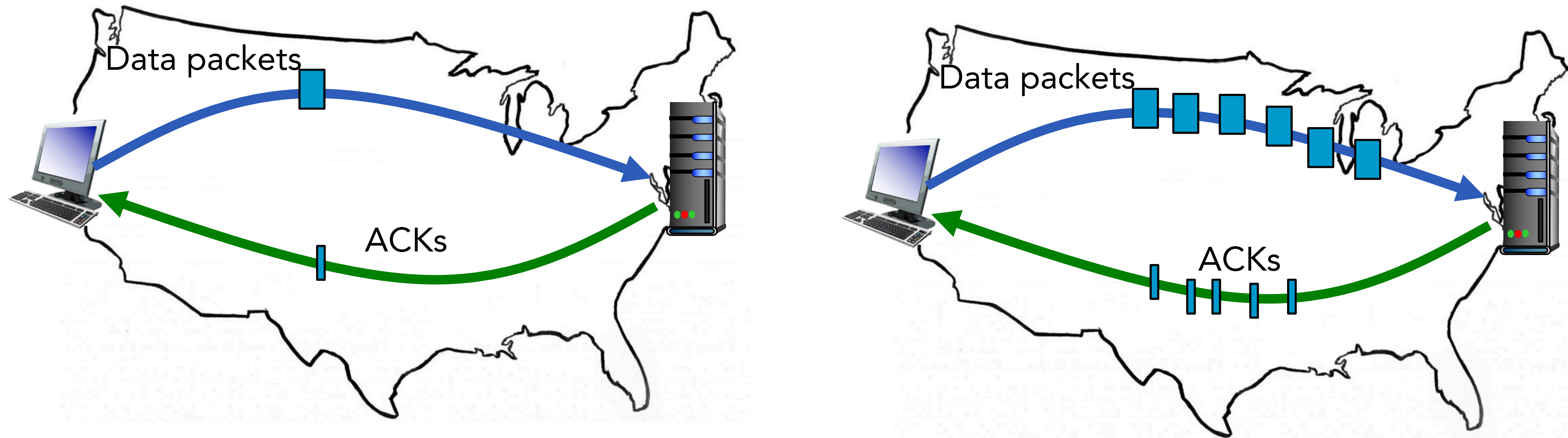


$$Utilization = \frac{L/R}{RTT + L/R} = 0.00027$$

TAKE 5

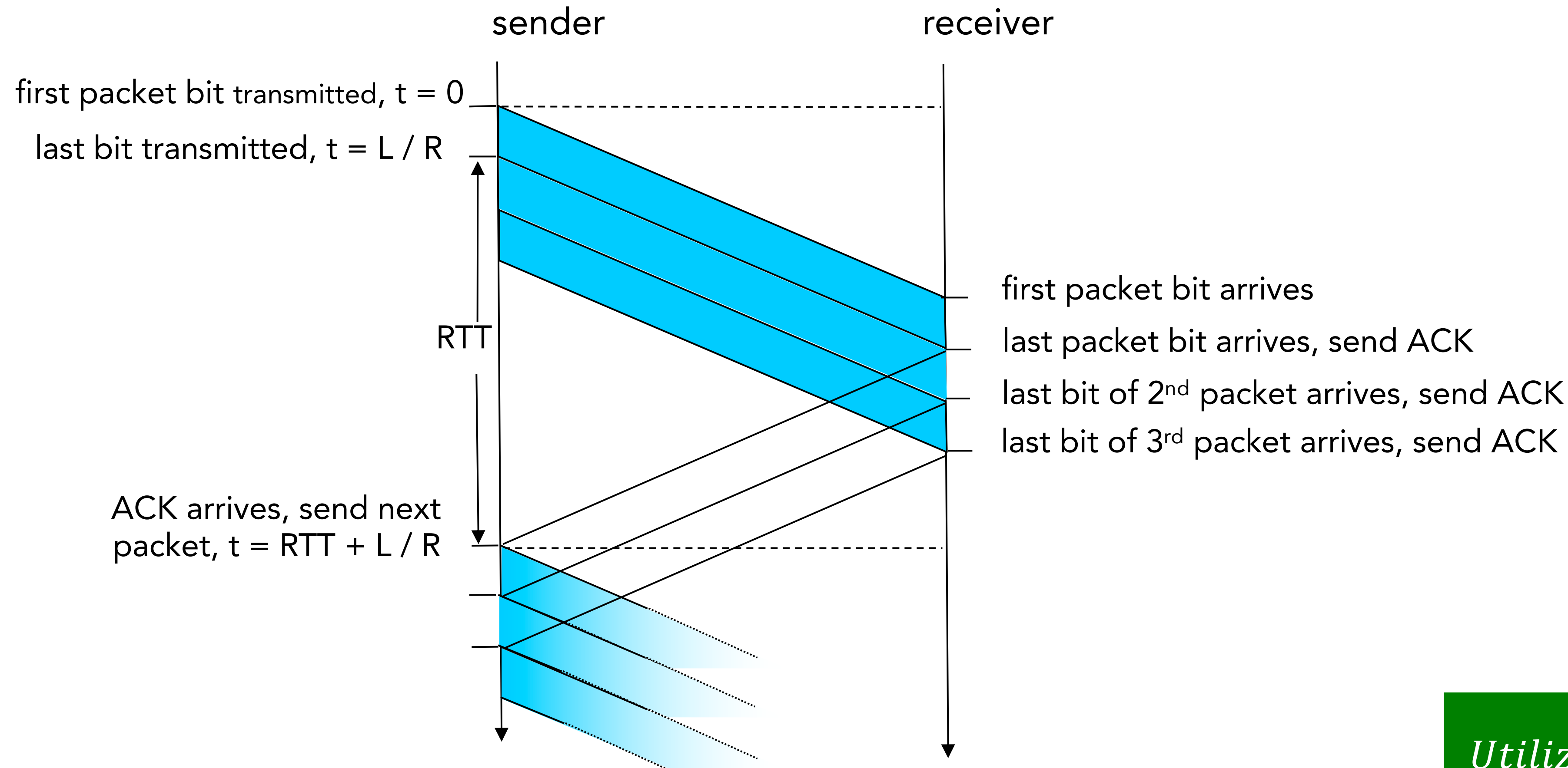
Pipelining hides latency to increase throughput

- Pipelining – Allows multiple "in-flight" pkts, not yet ACKed



- Window size is the max number of in-flight packets
- Finite to limit data buffering needed at each end, and load placed on the network

Pipelining increases link utilization



3-packet pipelining increases utilization by a factor of 3!

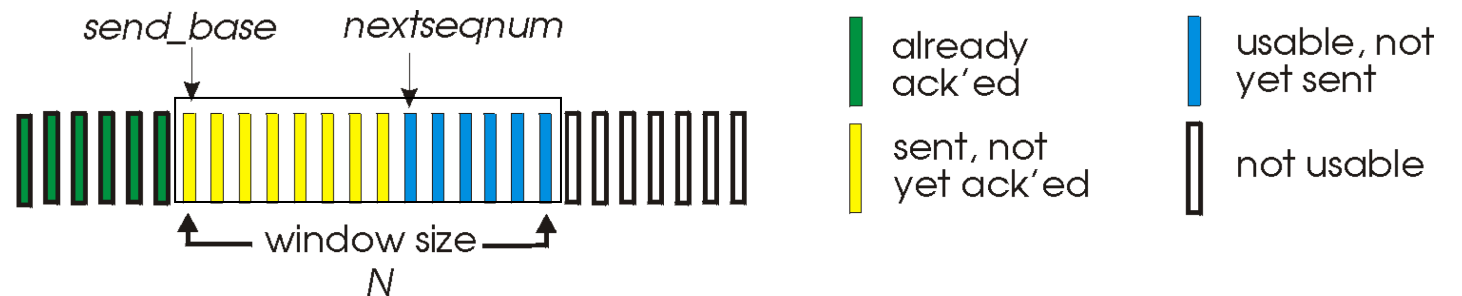
$$Utilization = \frac{3L/R}{RTT + L/R} = 0.00081$$

How to do pipelining

- Packet buffering and acknowledgement become more complex
  - Range of sequence numbers to accommodate for this
    - How to do it depends on how we deal with lost, corrupted or long delayed packets
  - Buffering to hold on packets sent but not yet ACKed
- Need flow/congestion control to prevent overwhelming the receiver/network
- Two basic approaches to pipelined error recovery
  - Go-Back-N
  - Selective repeat

Pipelining option #1: Go Back N

- Window size is *N*, sender can have up to *N* pkts in flight
  - The range of possible pkts transmitted but not yet ack'ed
  - As it runs, the window "slides" forward – sliding-window protocol
- Receiver sends cumulative ACK: "Got everything up to seq. # x"
  - Discard out-of-order pkt, re-send ACK of last in-order seq. #
  - If sender does not get an ACK after some timeout interval, resend all pkts starting from pkt after the last ACK'ed pkt
- If the sender timeout expires several times w/o receiving any ACK, give up on the connection

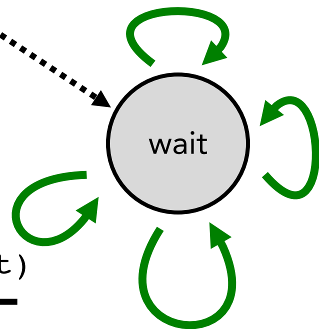


GBN: sender extended FSM

```
rdt_send(data)
______________________________________
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
       start_timer
    nextseqnum++
} else
  refuse_data(data)
```

"Refuse" could be done differently, such as using a synchronization variable that would allow the upper-layer call when the window is not full

```
Λ
______________
base=1
nextseqnum=1
```

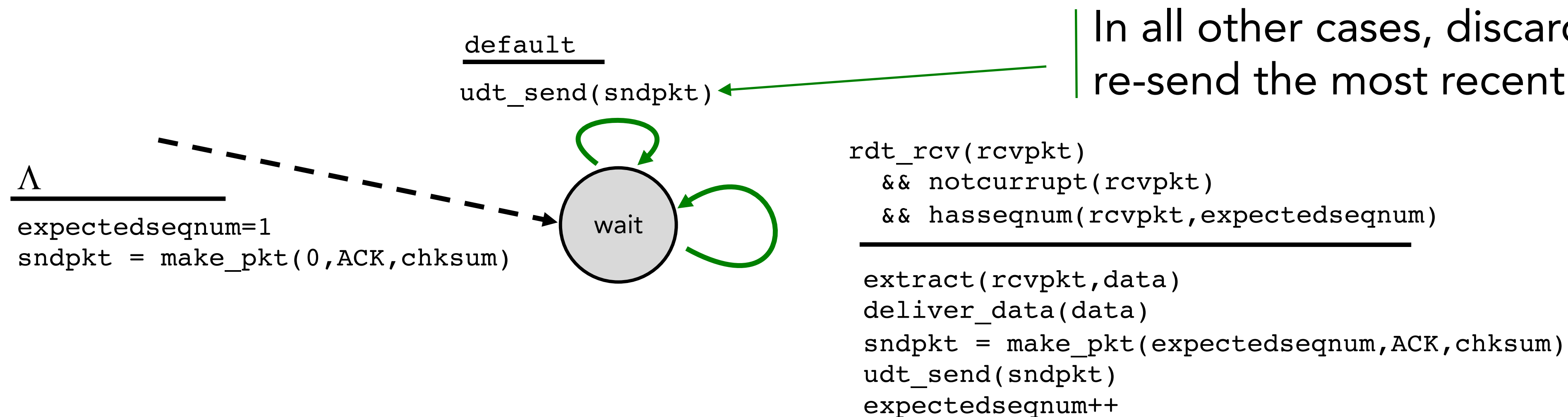



```
timeout
______________________________________
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

```
rdt_rcv(rcvpkt)
   && corrupt(rcvpkt)
______________________
Λ
```

```
rdt_rcv(rcvpkt) &&
   notcorrupt(rcvpkt)
______________________________
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

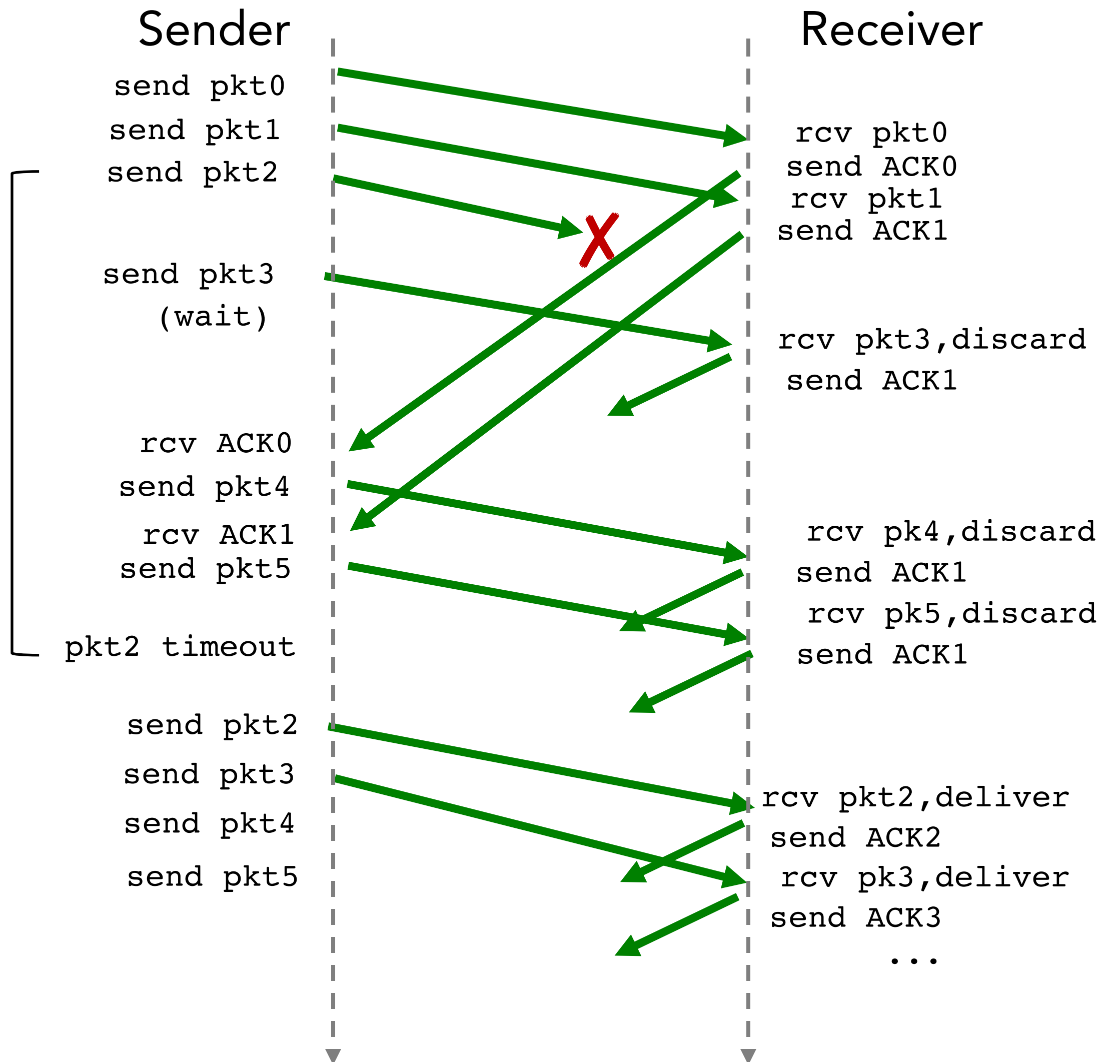
GBN: receiver extended FSM



- ACK-only: always send ACK for correctly-received pkt with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember expected seq#
  - out-of-order pkt:
    - discard (don't buffer): no receiver buffering! (*Why?!*)
    - re-ACK pkt with highest in-order seq #

BTW, how would you implement this?

GBN in action



Window size is 4

Because of window size, sends 0 to 3 and wait

With every ACK (one less packet in the pipe), the sliding window can move forward and sender can send another packet

Out of order packets are discarded

Go Back N pros / cons

- Pros: Easy to implement:
  - Sender just stores # of last ACK and maintains a timer
  - Receiver just stores expected seq number and immediately passes new in-order packets to listening app
- Cons: A **single** lost or **delayed** pkt invalidates all in-flight data
  - When the window size and the bandwidth*delay product are large, there can be a lot of packets in the pipeline
  - Receiver can throw out a lot of good data, just because it's "early"
    - I.e. lacks **receiver buffering**
  - Lose an entire window of data due to one "bad" packet

Pipelining option #2: Selective Repeat

- *Only re-send an pkt whose transmission or ACK was lost*
- Receiver individually ACKs all received pkts
- Out-of-order pkts are stored by receiver and later reassembled
- Sender keeps one timer per each in-flight packet, and will re-send any pkts not ACK'ed before timeout
- Window of size *N* limits the max range of un-ACK'ed pkts
  - Receiver drops received pkts with seq # outside the window
  - This prevents pkts from old connection from getting inserted into new connection's data stream

Selective repeat

sender

data from above:

- if next available seq # in window, send pkt
- else, return up or buffered

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, i.e., consecutively numbered), advance window to next not-yet-received pkt

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

- ACK(n)

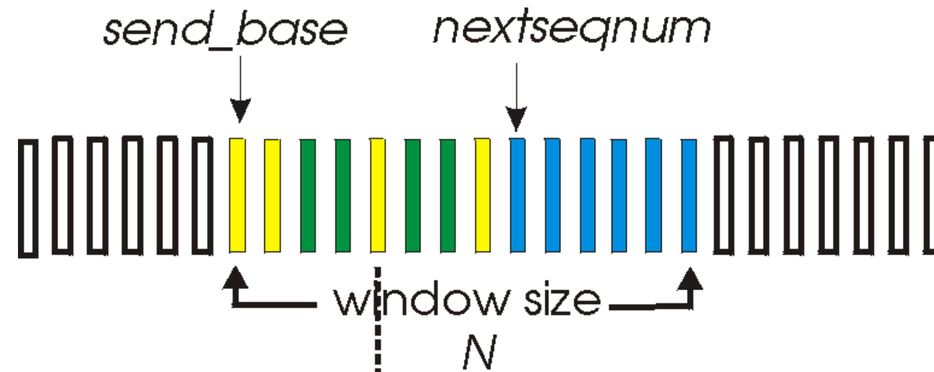
otherwise:

- ignore

Note it re-ack, rather than ignore already received pkts

Selective Repeat windows

Sender view of sequence numbers



already ack'ed

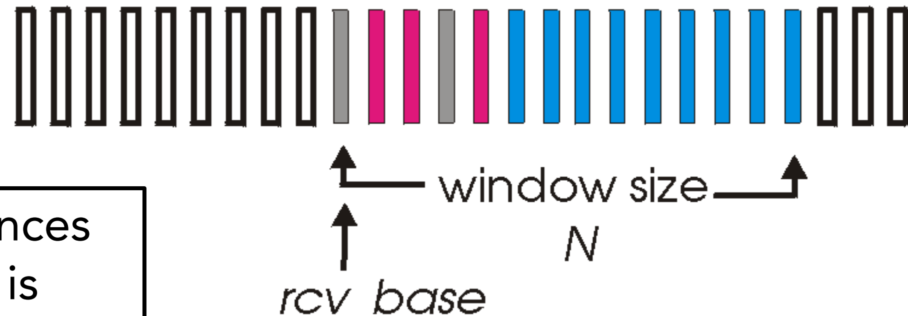
usable, not yet sent

sent, not yet ack'ed

not usable

Sender window advances when lowest packet is ACK'ed.

Receiver view of sequence numbers



out of order (buffered) but already ack'ed

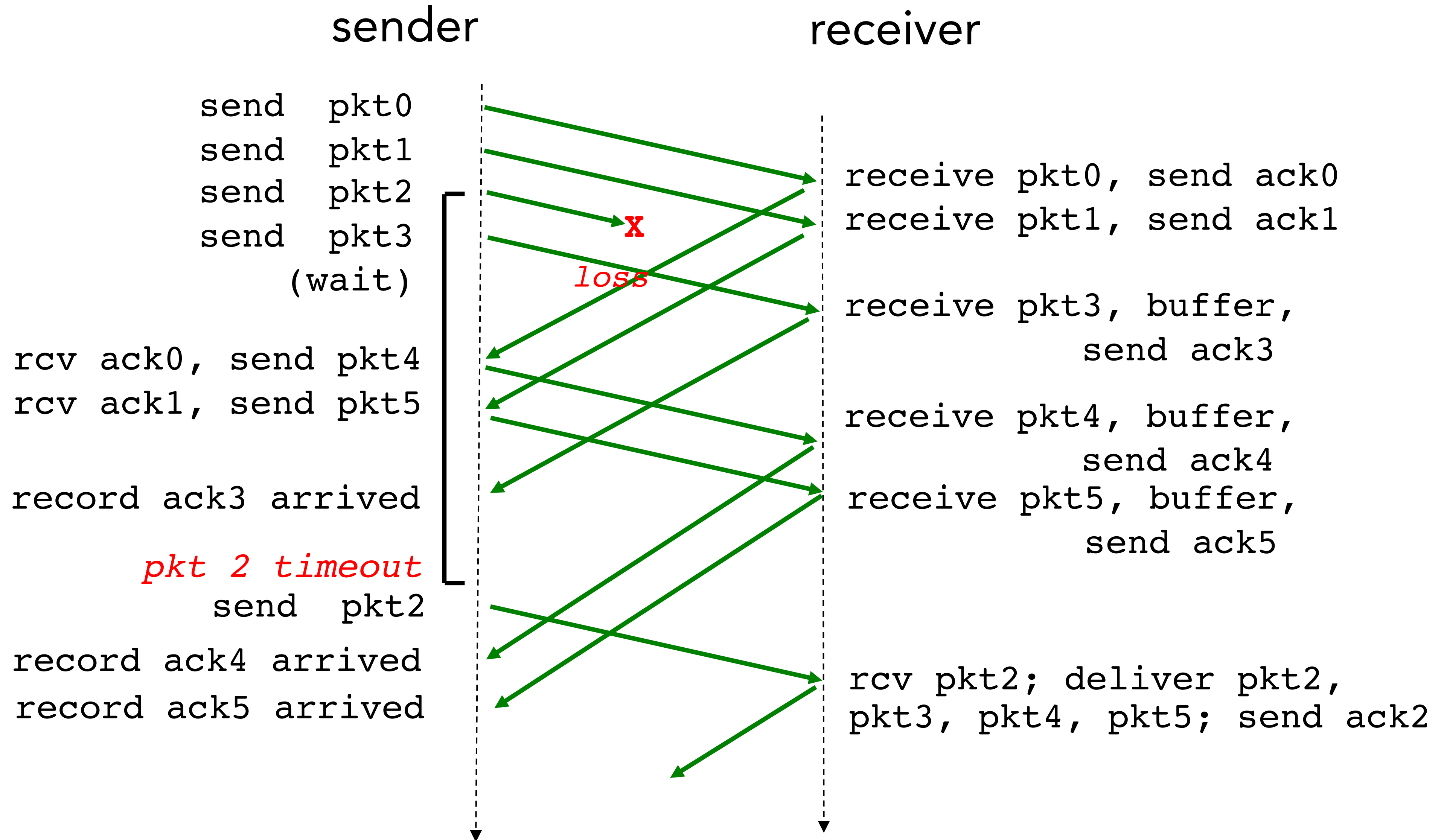
acceptable (within window)

Expected, not yet received

not usable

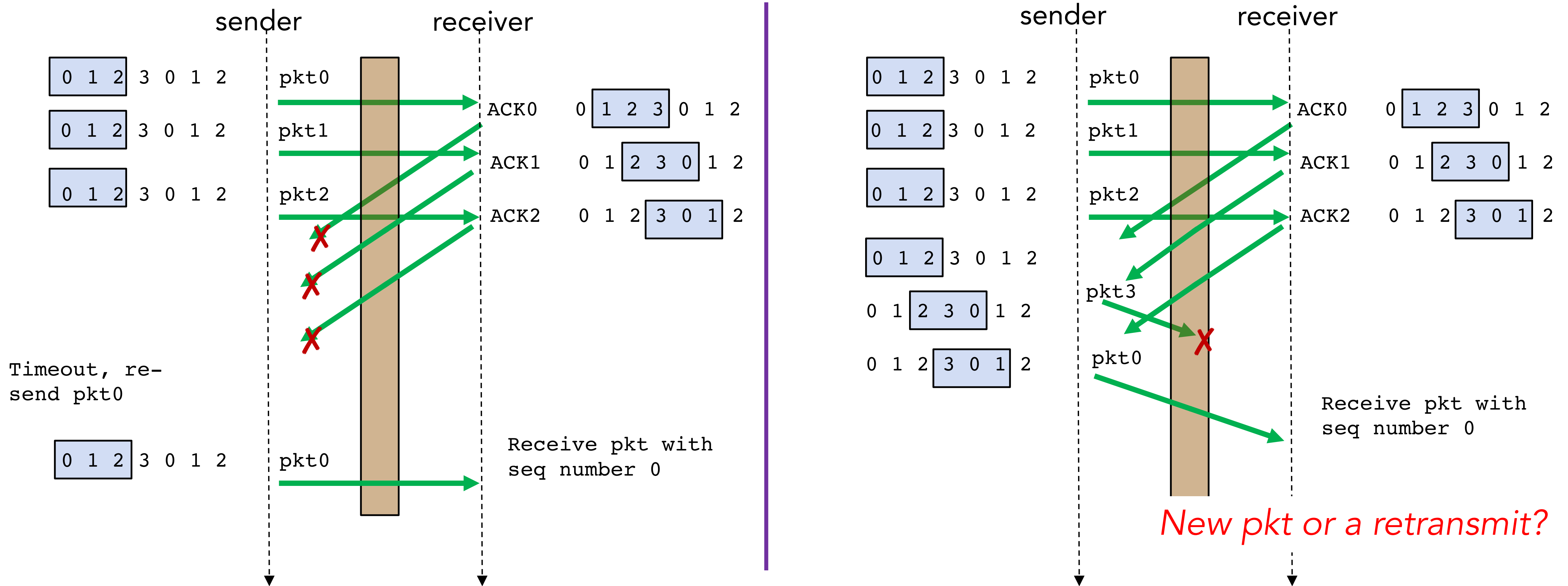
Receiver window advances when lowest packet is received.

Selective repeat in action



Careful with sequence number reuse

- Assumes that 2-bit seq number and a window of size 3



- Solution: window length must be < half the max seq number

Recap

- The underlying principles of reliable transfer → ready for TCP
- A summary of mechanisms and their use

Mechanism	Use
Checksum	Detect bit errors in a transmitted packet
Timer	To timeout/retransmit a packet; since it's a guess, need to handle duplicates
Seq number	To detect gaps in sequences of packets sent and detect duplicates as well
ACK	For the receiver to tell the sender it got it, may be individual or cumulative
NACK	For the receiver to tell the sender that a pkt wasn't received correctly
Window, pipelining	To improve utilization over stop-and-wait mode; the window may be set based on the receiver's ability to receive and buffer msgs, the level of congestion in the network or both