Reliable Transport and TCP

Kevin Scrivnor

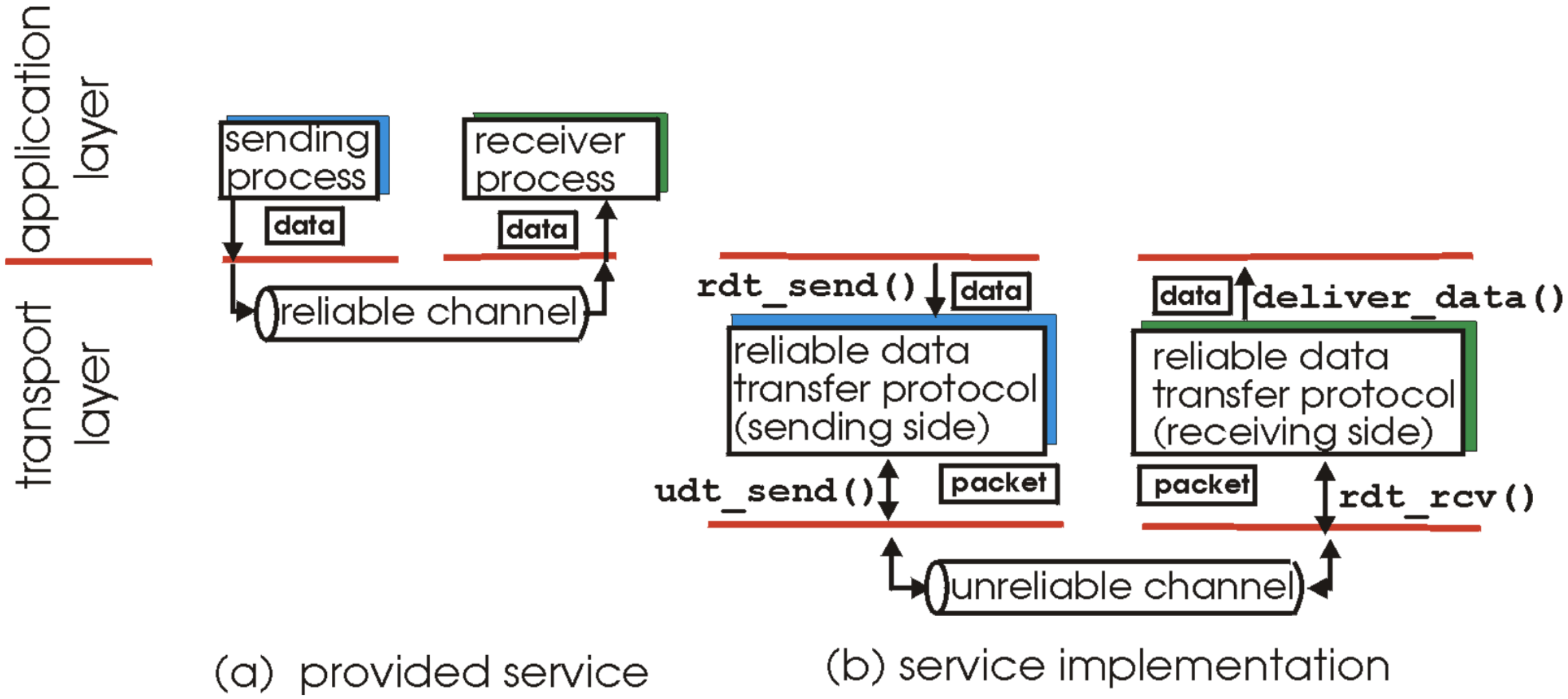
COMP 429

Fall 2023

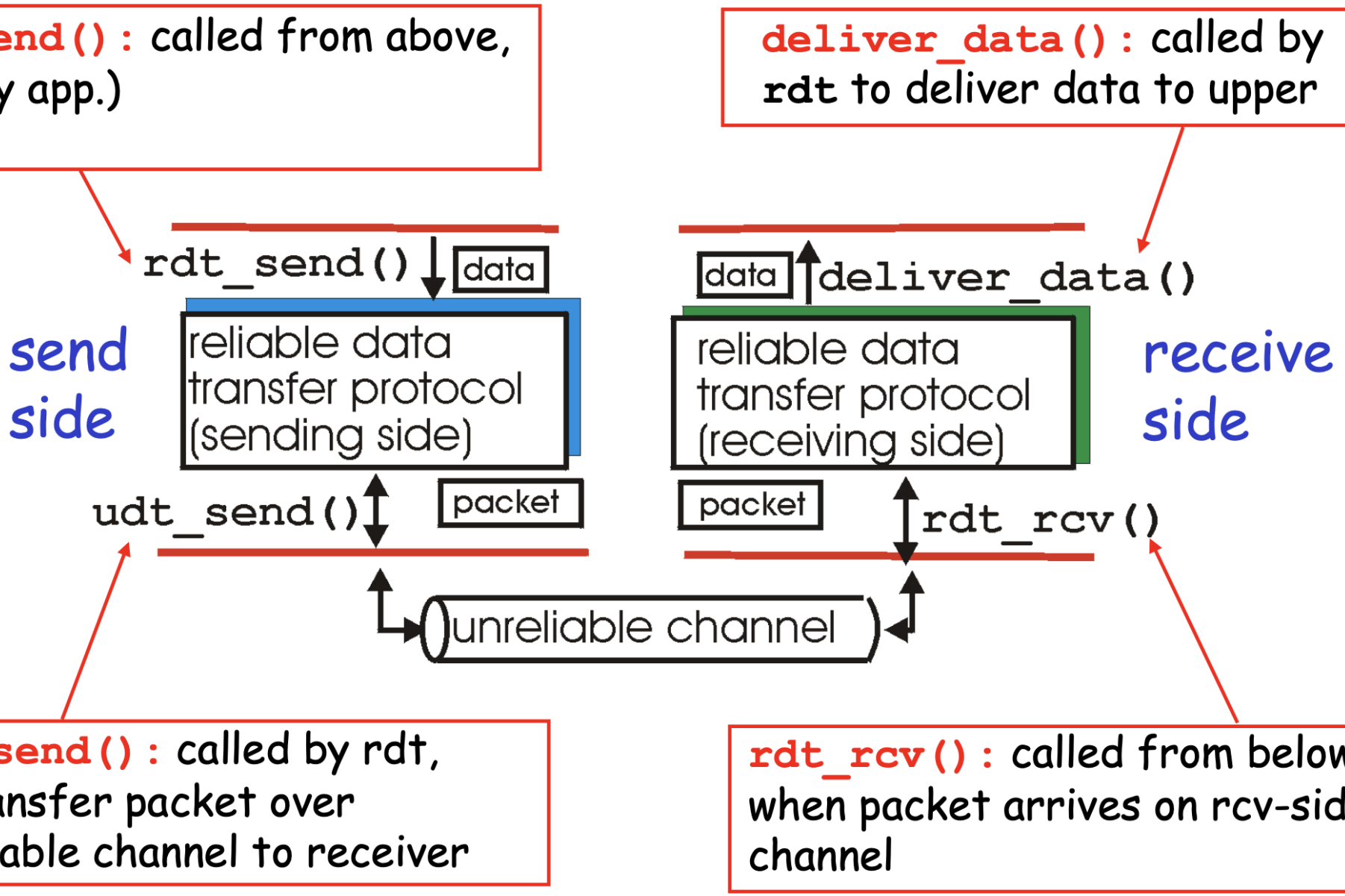
RDT

Reliable Transfer: Abstraction

- Reliably transmit data over an unreliable channel



Reliable Transfer: Context



Reliable Data Transfer: Goals and Assumptions

- Incrementally develop sender side and receiver side of rdt

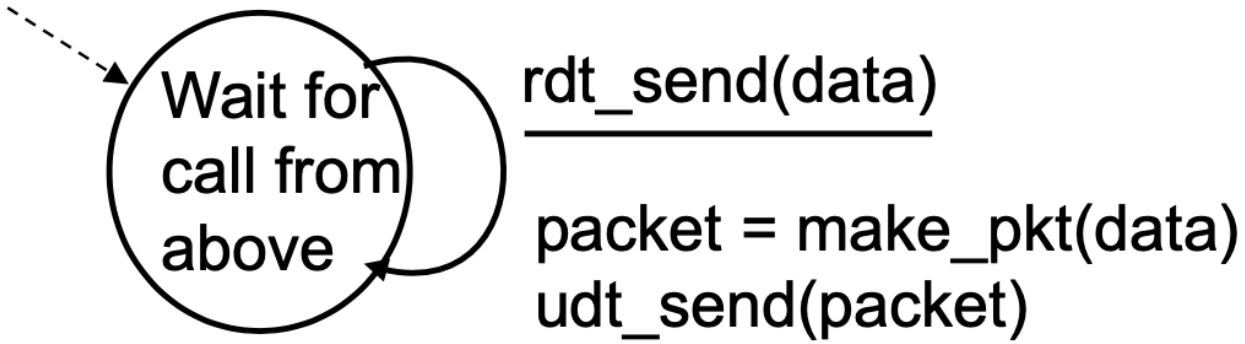
- Consider only unidirectional data transfers
  - But control info will flow in both directions

- Use a finite-state machine (FSM) to specify sender/receiver

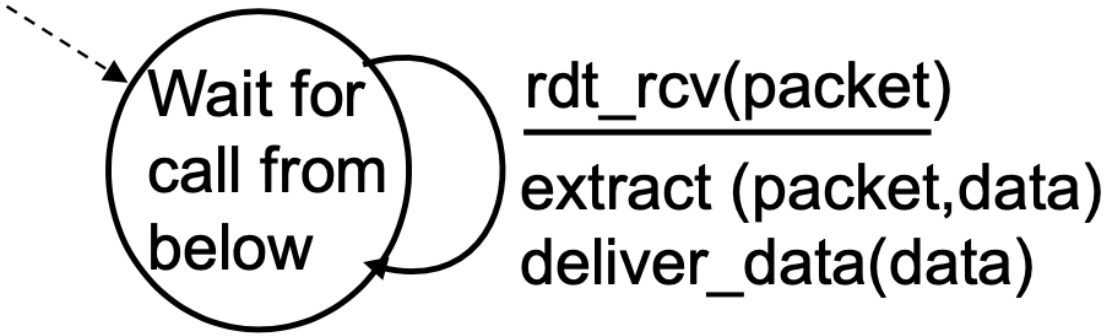
rdt 1.0: reliable transfer over reliable channel

- FSM for sender and FSM for receiver
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel

- Why is there no need for any checks?



sender



receiver

Potential Channel Errors

- Bit errors
- Loss (drop) of packets
- Reordering or duplication

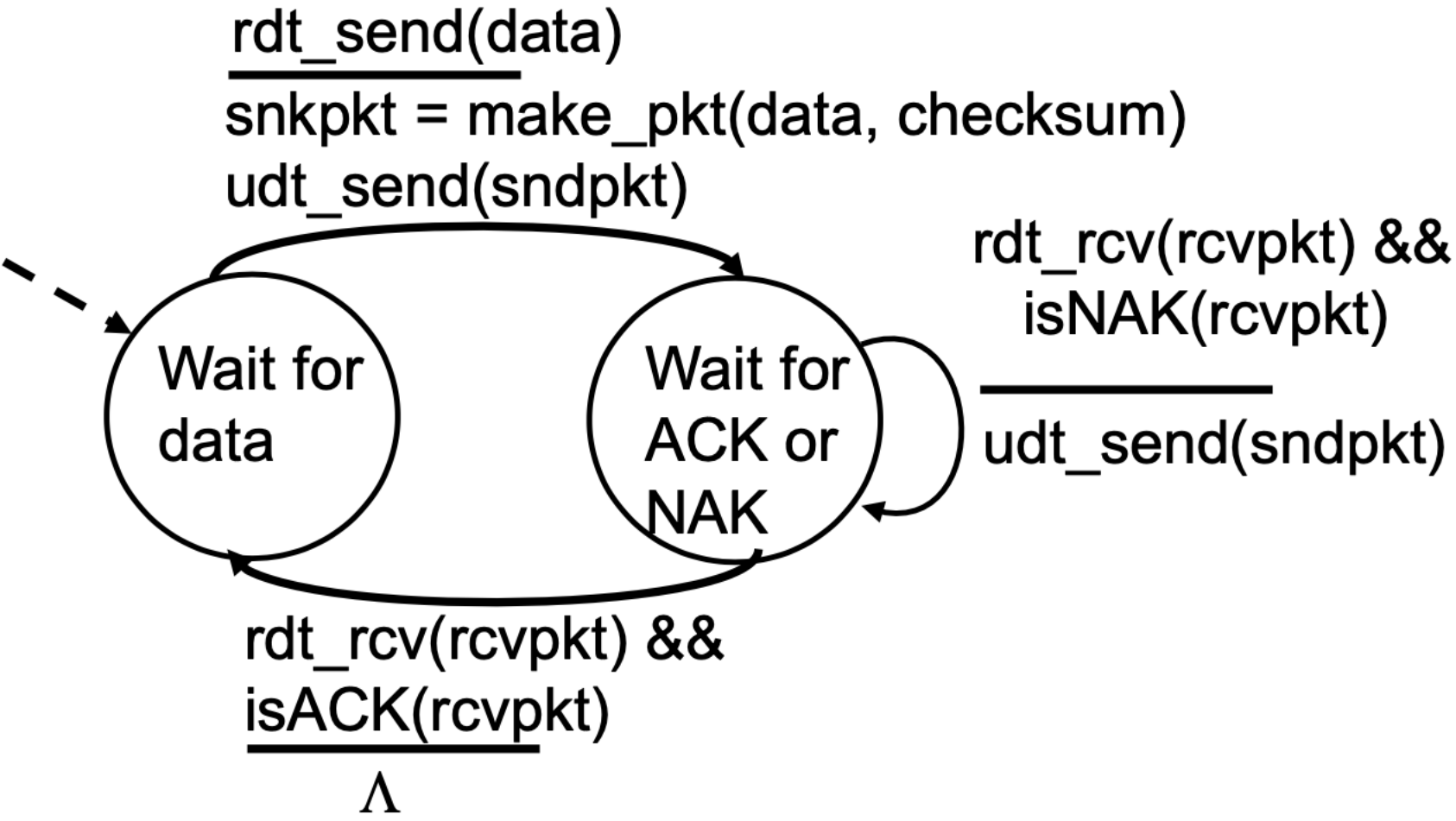
- Each characteristic of the unreliable channel below the transport layer will add to the complexity of the rdt protocol.
- We'll deal with them one by one, building rdt.

rdt 2.0: Channel with Bit Errors

- Assume: channel may flip bits in transit
- Example: “Hi” turns into “Ho” in transit.

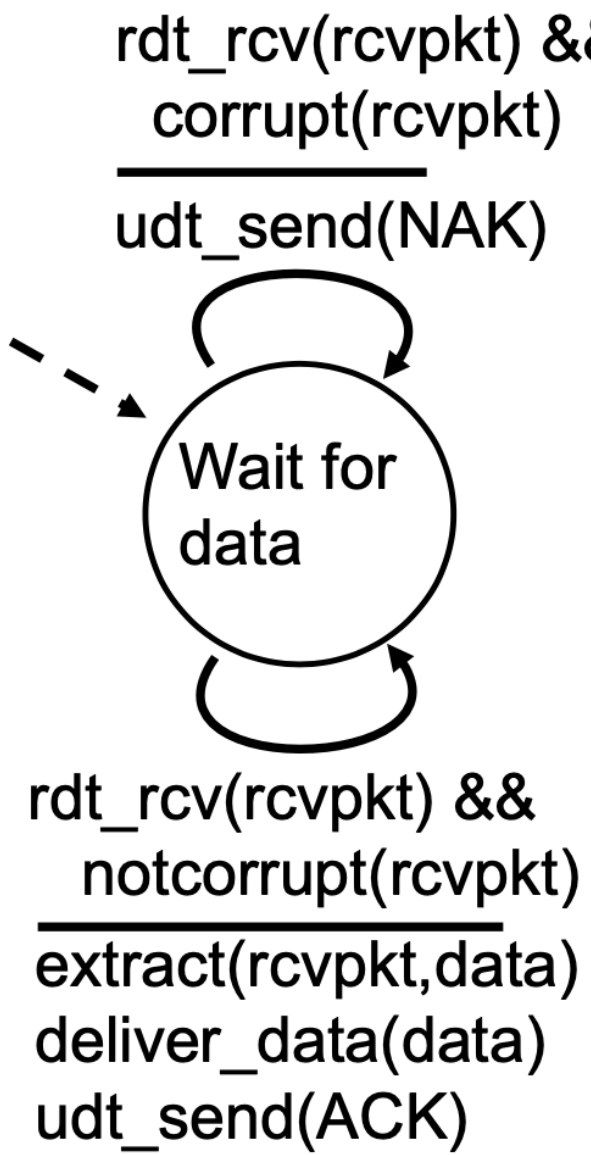
- New mechanisms required (complexity)
  - Checksum for error detection
  - Feedback messages (ACK, NAK)
    - **ACK**nowledgment messages: OK
    - **N**egative **A**c**K**nowledgments: pkt had errors
  - Sender retransmission
    - Resend packet that had errors

rdt 2.0: FSM Spec



sender

receiver



rdt 2.0: Issues

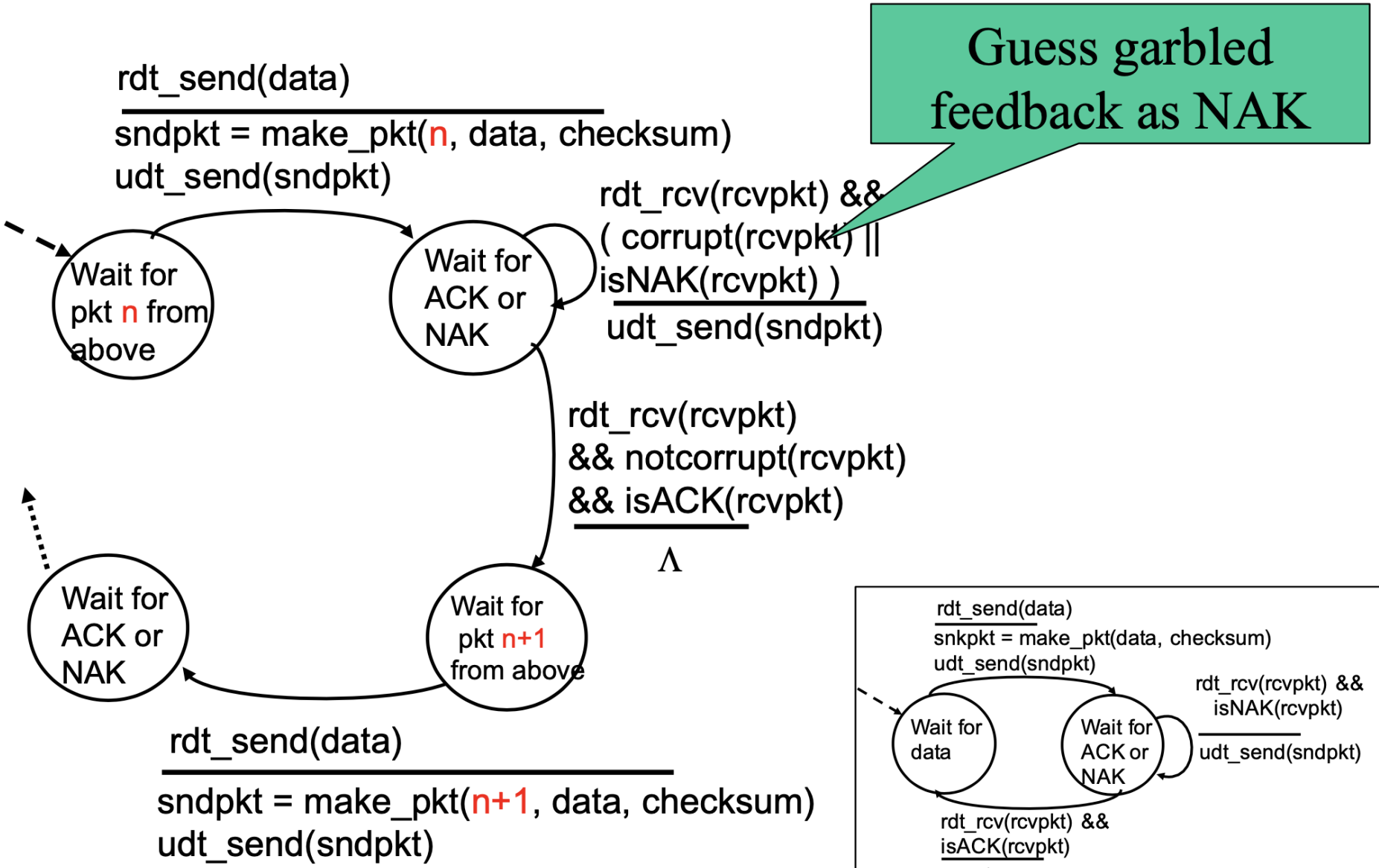
- What if ACK/NAK is corrupted?

- Always harder to handle errors in control messages!

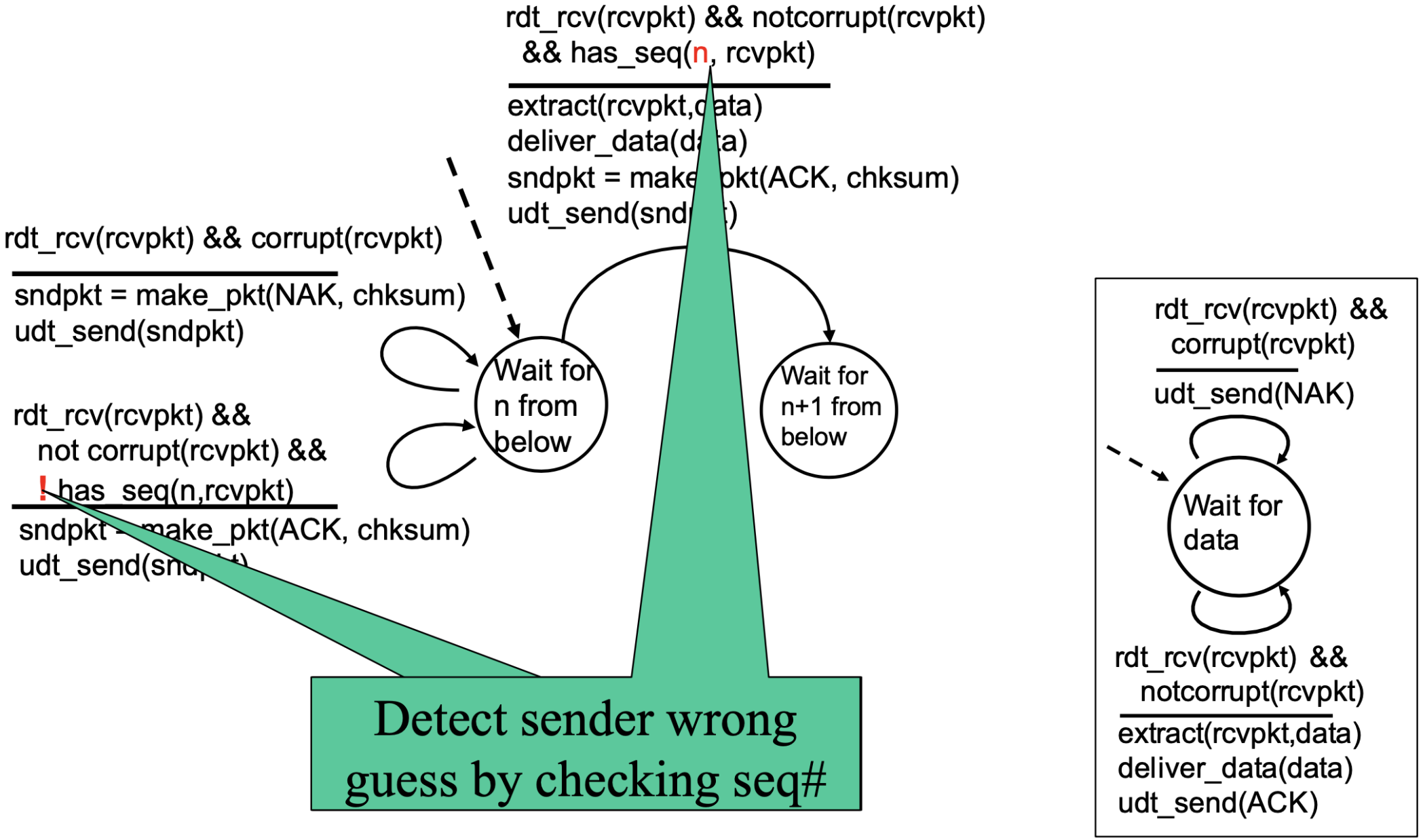
- Complexity Added for corrupted NAK
  - Assume NAK on corrupted packet
  - Sender adds sequence number to each packet
  - Receiver discards duplicate packets
    - Fix effect of wrong guess

- Stop-and-Wait: Sender sends one packet, waits for ACK, sends next…

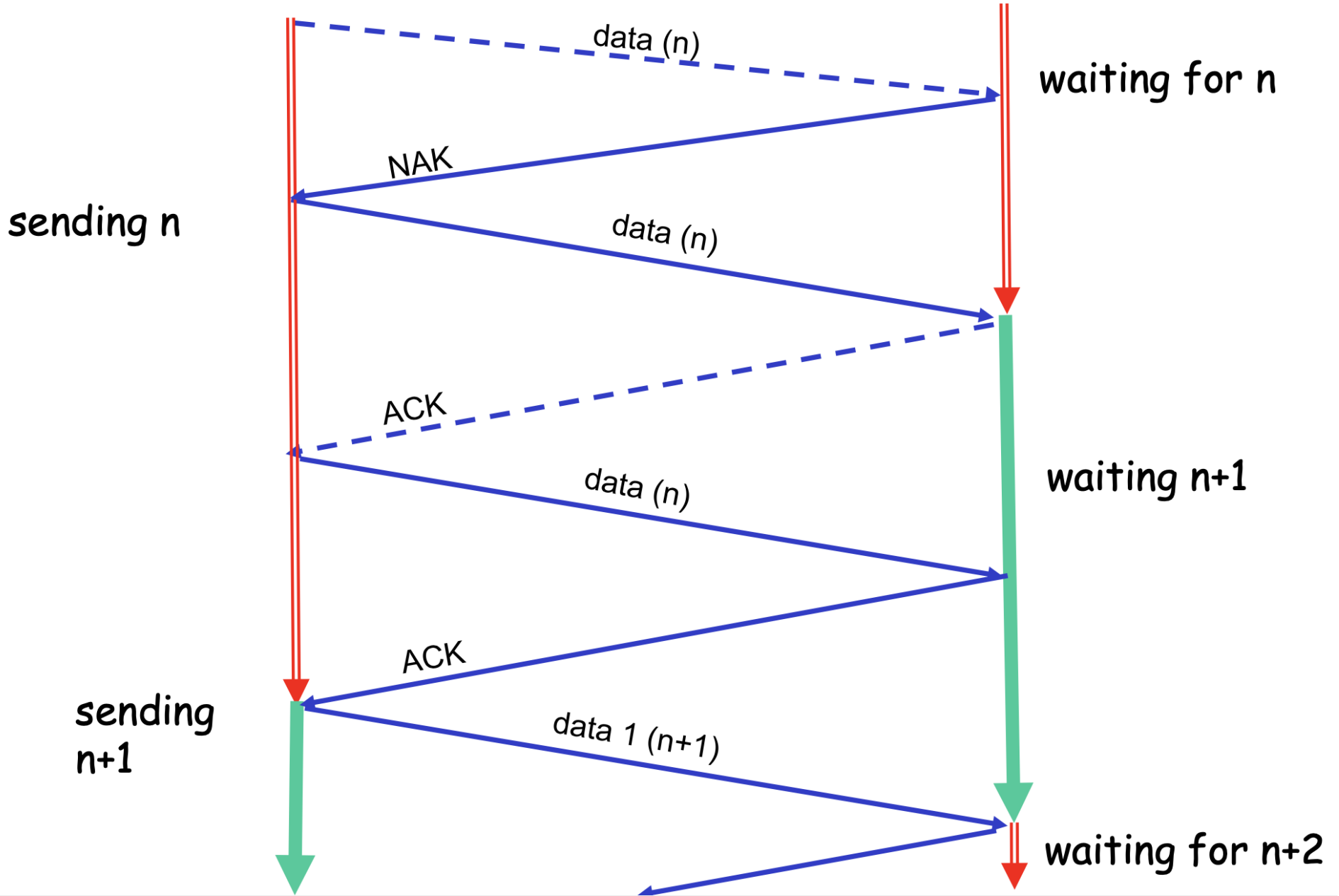
rtd 2.1: Sender Handles Corrupted ACK/NAK



rdt 2.1: Receiver Handles Corrupted ACK/NAK



rdt 2.1: Analysis



rdt 2.1: Summary

- Sender
  - Sequence # added to packet
  - Must check if ACK/NAK is corrupted

- Receiver
  - Must check if packet is a duplicate
    - By checking if the packet has the correct sequence number

Recap: Issues to Deal With

- Every potential error adds complexity to protocol
  - Bit errors
  - Loss or drop of packets
  - Reordering or duplication

- What if the ACK never comes? Then you have to deal with duplicates.
- What if the ACK is corrupted? Duplicates.
- What if the message was corrupted? NAK sent

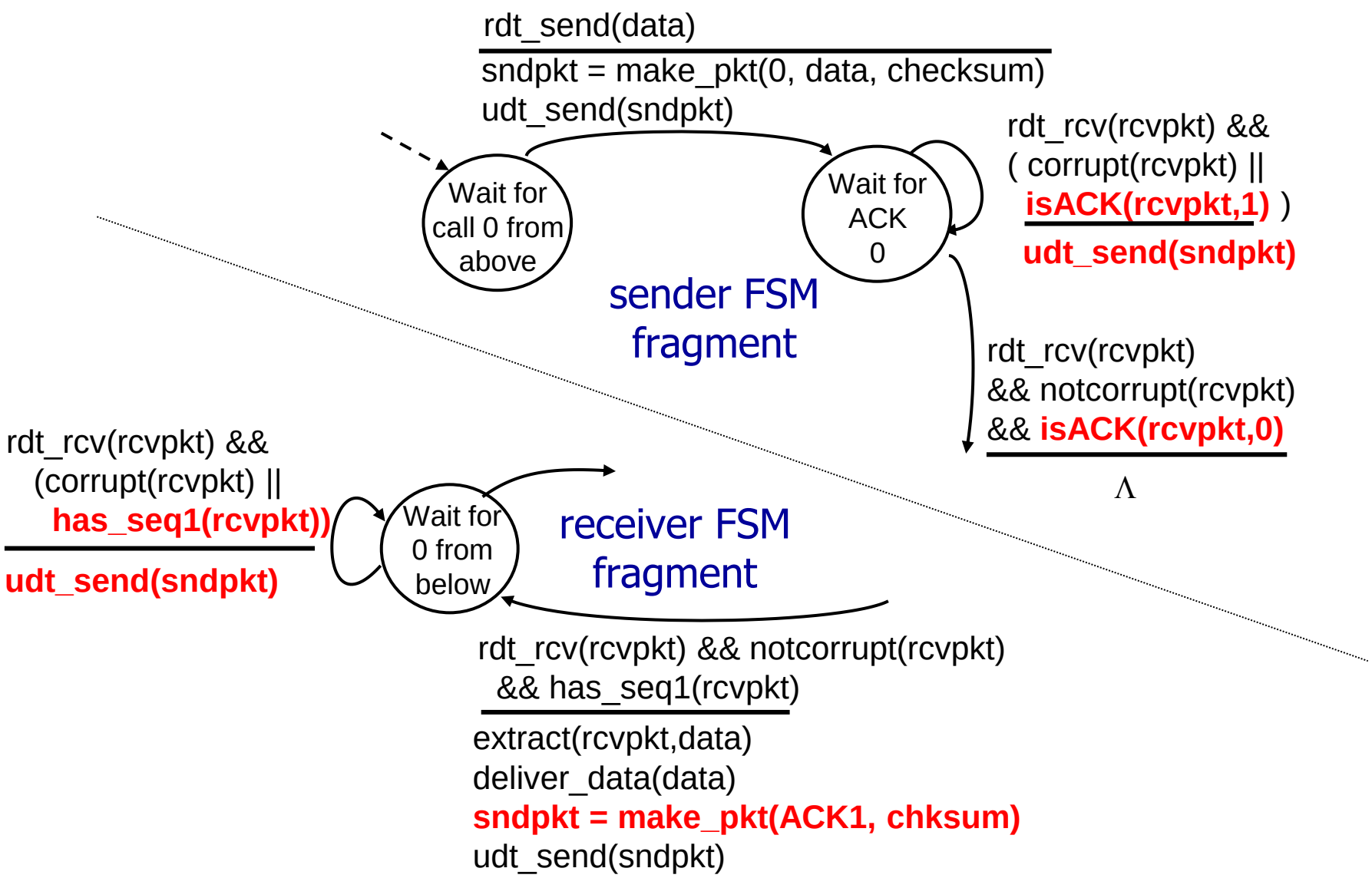
rdt2.2: NAK Free Protocol

- Same functionality as previous iteration

- Instead of sending a NAK, receiver sends ACK for *last successful* sequence number.
  - If waiting on seq 1, but it arrives corrupted, send ACK for seq 0
  - Meaning, seq 0 was the last successful seq number recv'd

- Therefore, duplicate ACK at sender results in same action as a NAK.
- Now we can have one less control message (Just ACKs)

rdt2.2: sender, receiver fragments



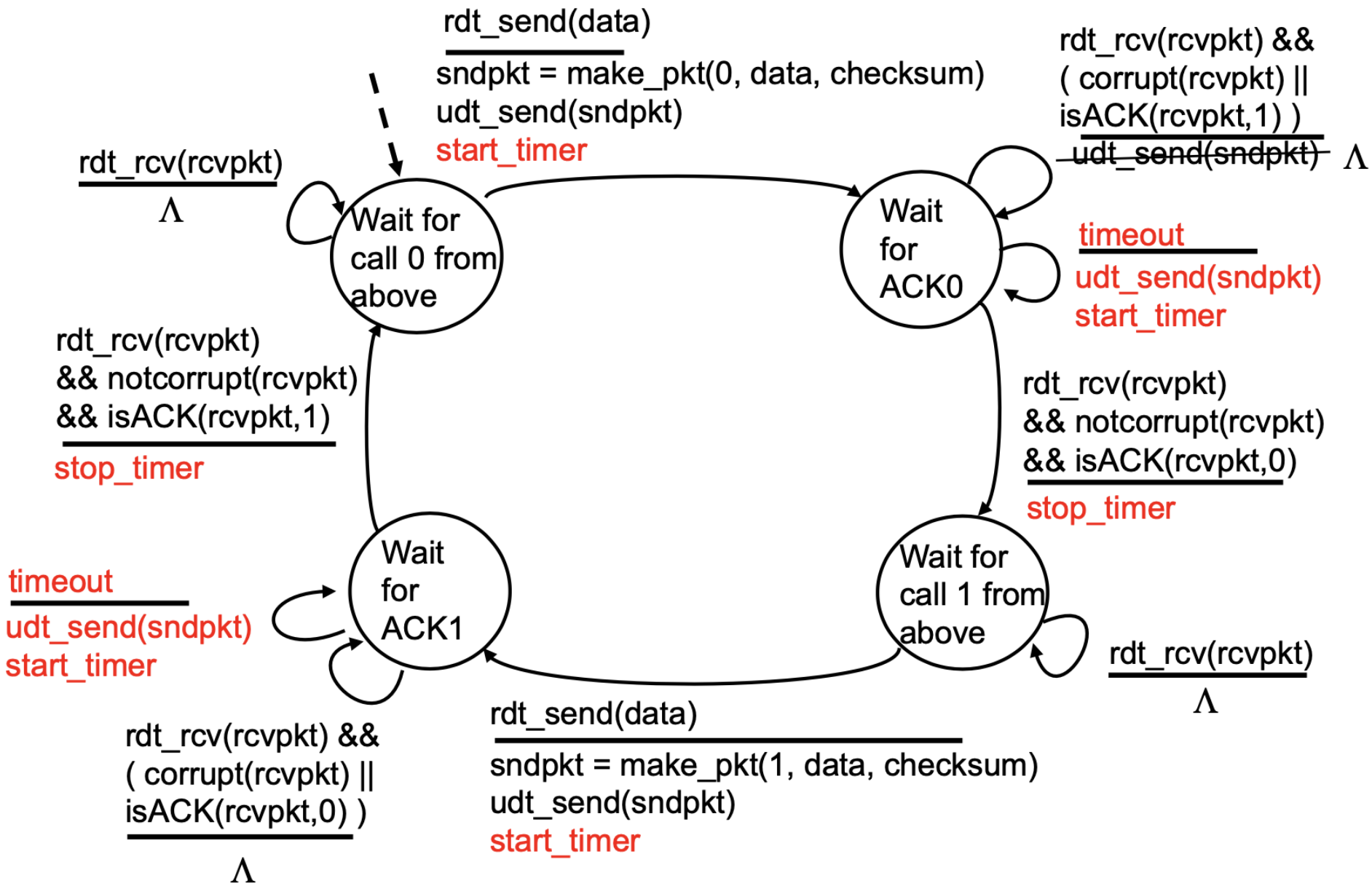
rdt3.0: Channel Errors and Loss

- New Assumption
  - Underlying (unreliable) channel can also lose packets (data/ACKs)
  - What happens if the protocol never receives a packet?

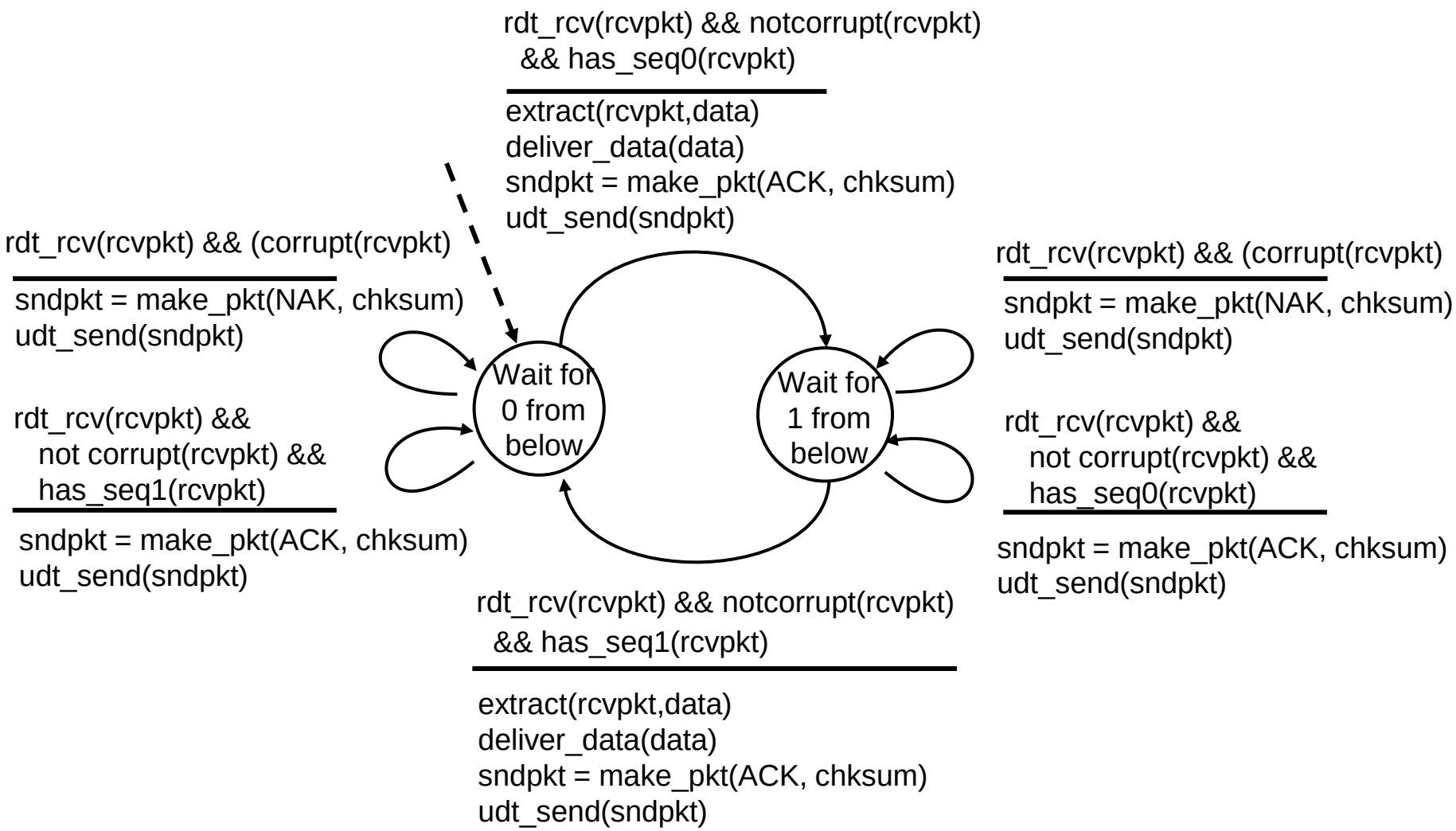
- Idea
  - Sender waits a reasonable amount of time for ACK

- New Requirements
  - Requires a countdown timer
  - Retransmits if no ACK is received
  - If ACK is delayed
    - Retransmission will be a duplicate (already handled)
    - Receiver must specify seq number of packet being ACK'd

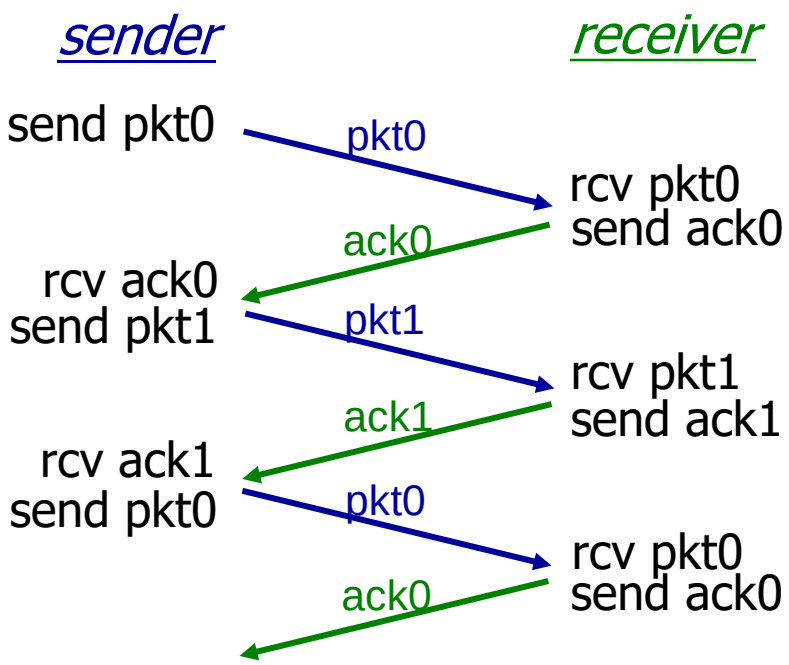
rdt3.0 Sender



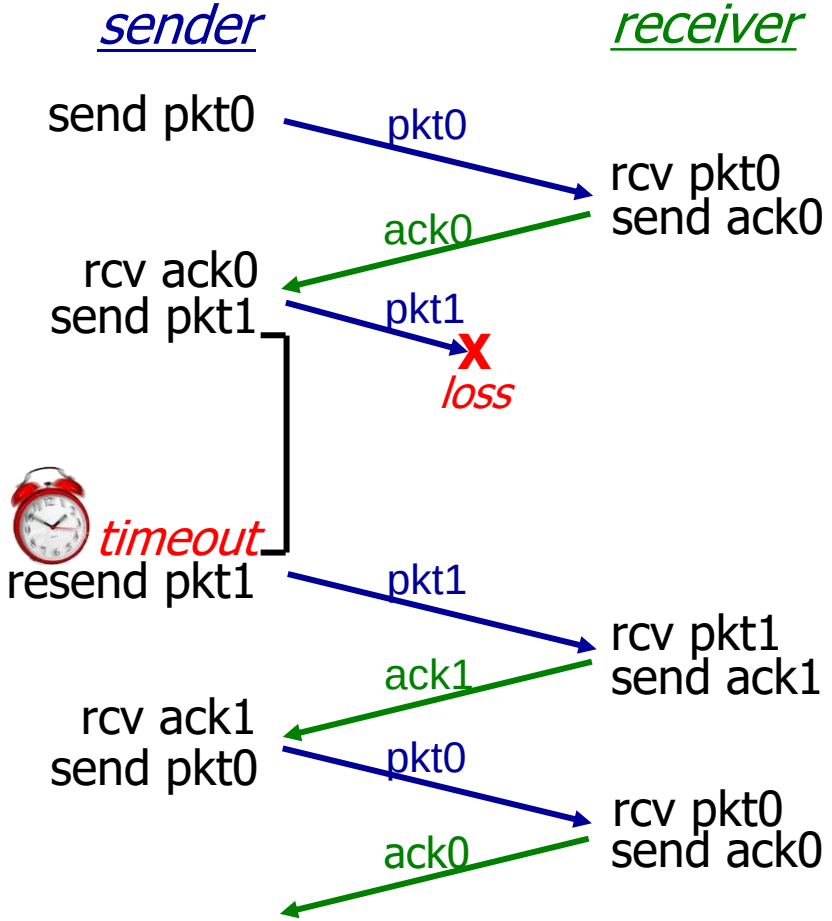
rdt3.0 Receiver (same as 2.2)



rdt3.0 in action

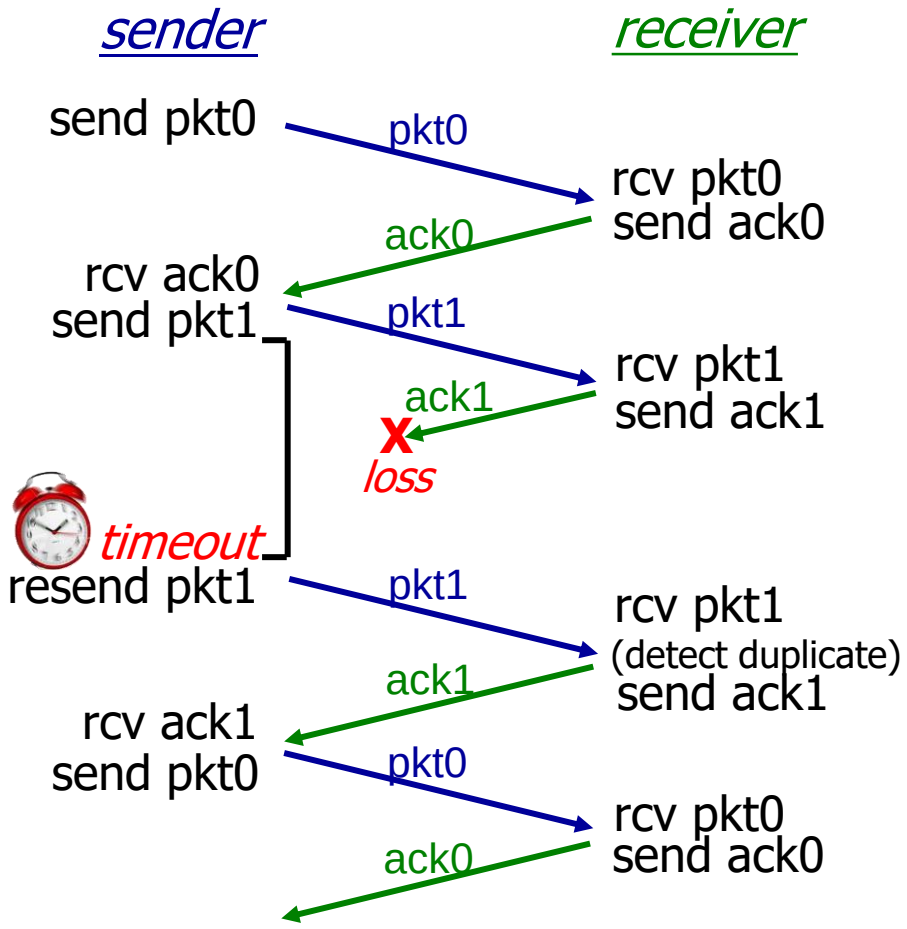


(a) no loss

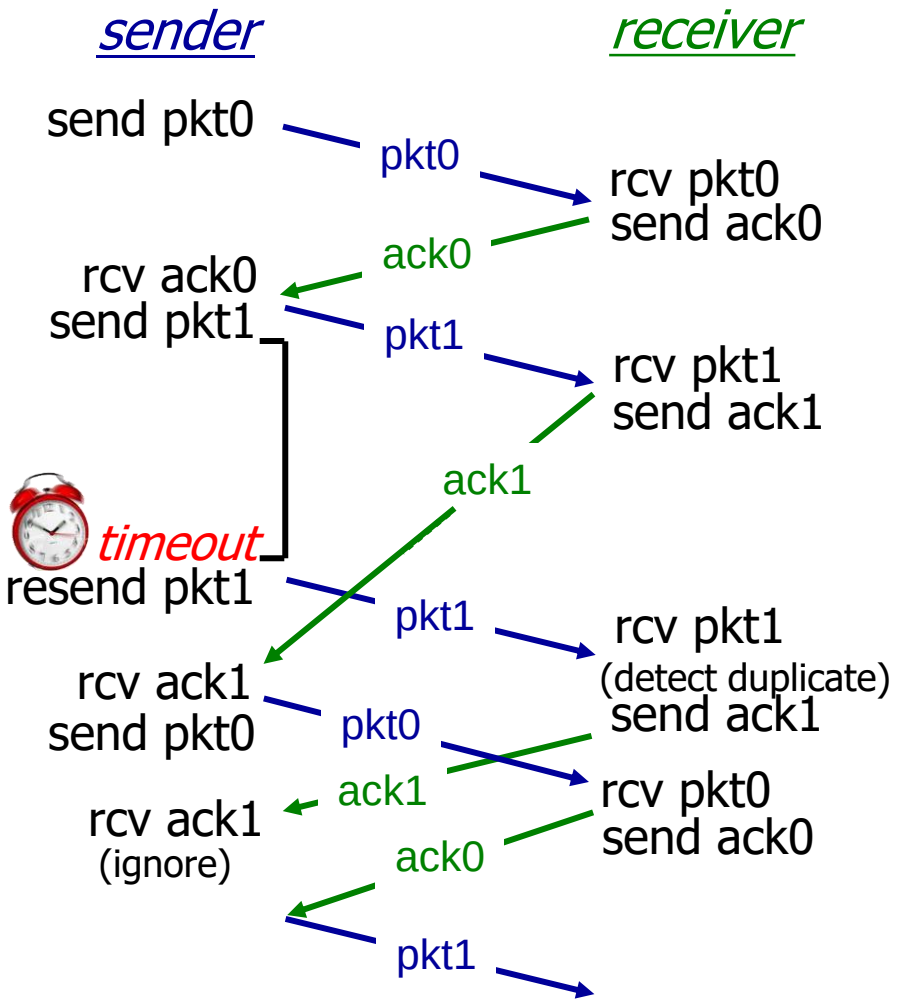


(b) packet loss

rdt3.0 in action



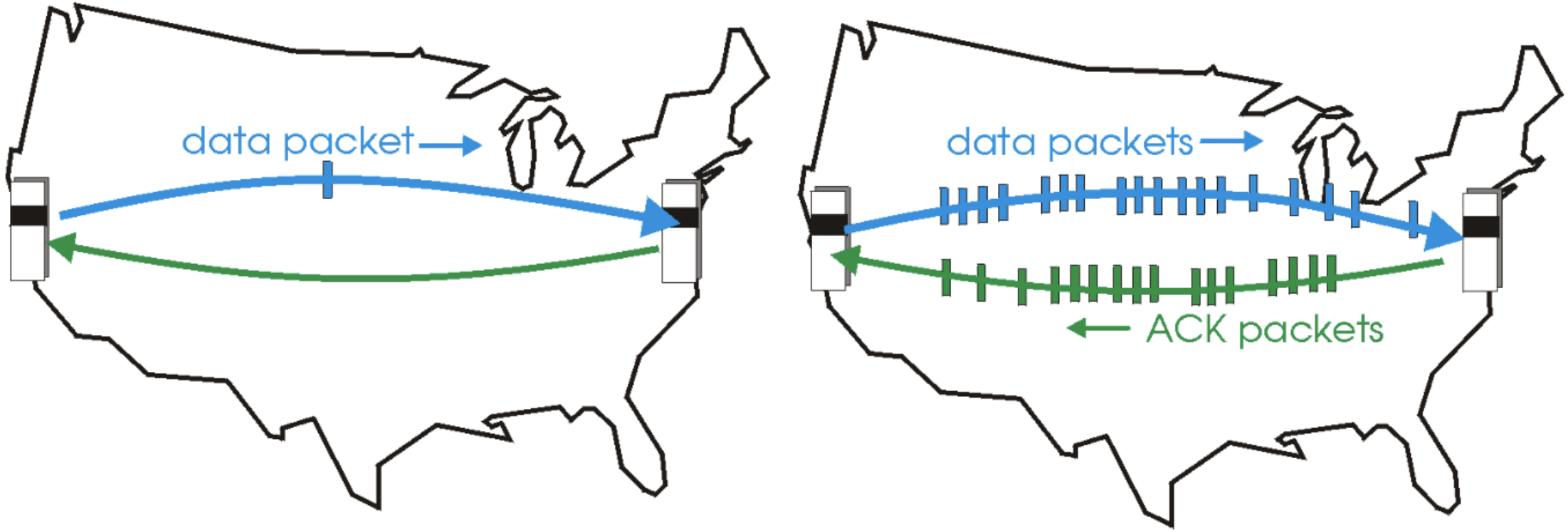
(c) ACK loss



(d) premature timeout/ delayed ACK

Sliding Window Protocol: Pipelining

- Pipelining: sender allows multiple, “in flight”, unack’d packets
- Question: What complexity is added?



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

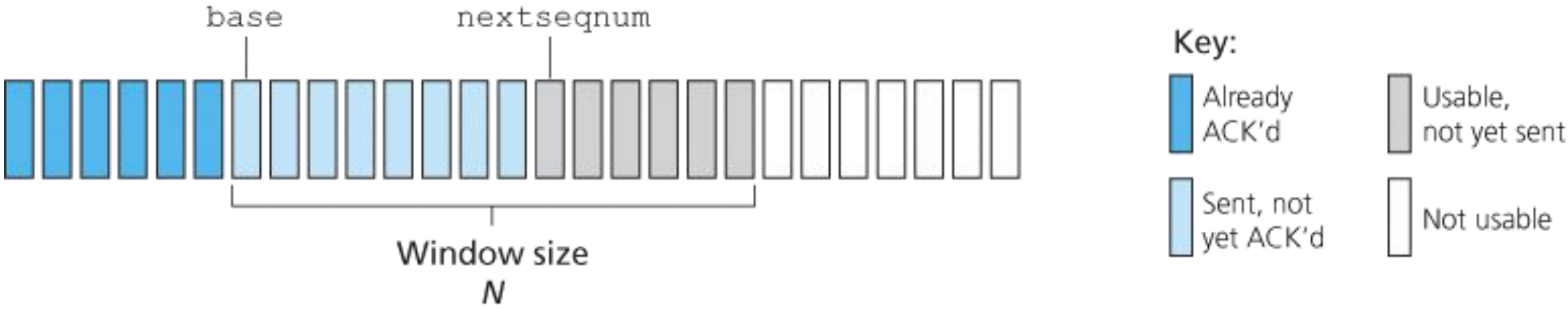
(b) a pipelined protocol in operation

Complexity Added

- Range of sequence numbers needs to be increased
- Unique numbers
- Packets on either side need to be buffered
- How many to send at a time?
- Error recovery
  - Two approaches: Go back N and selective repeat

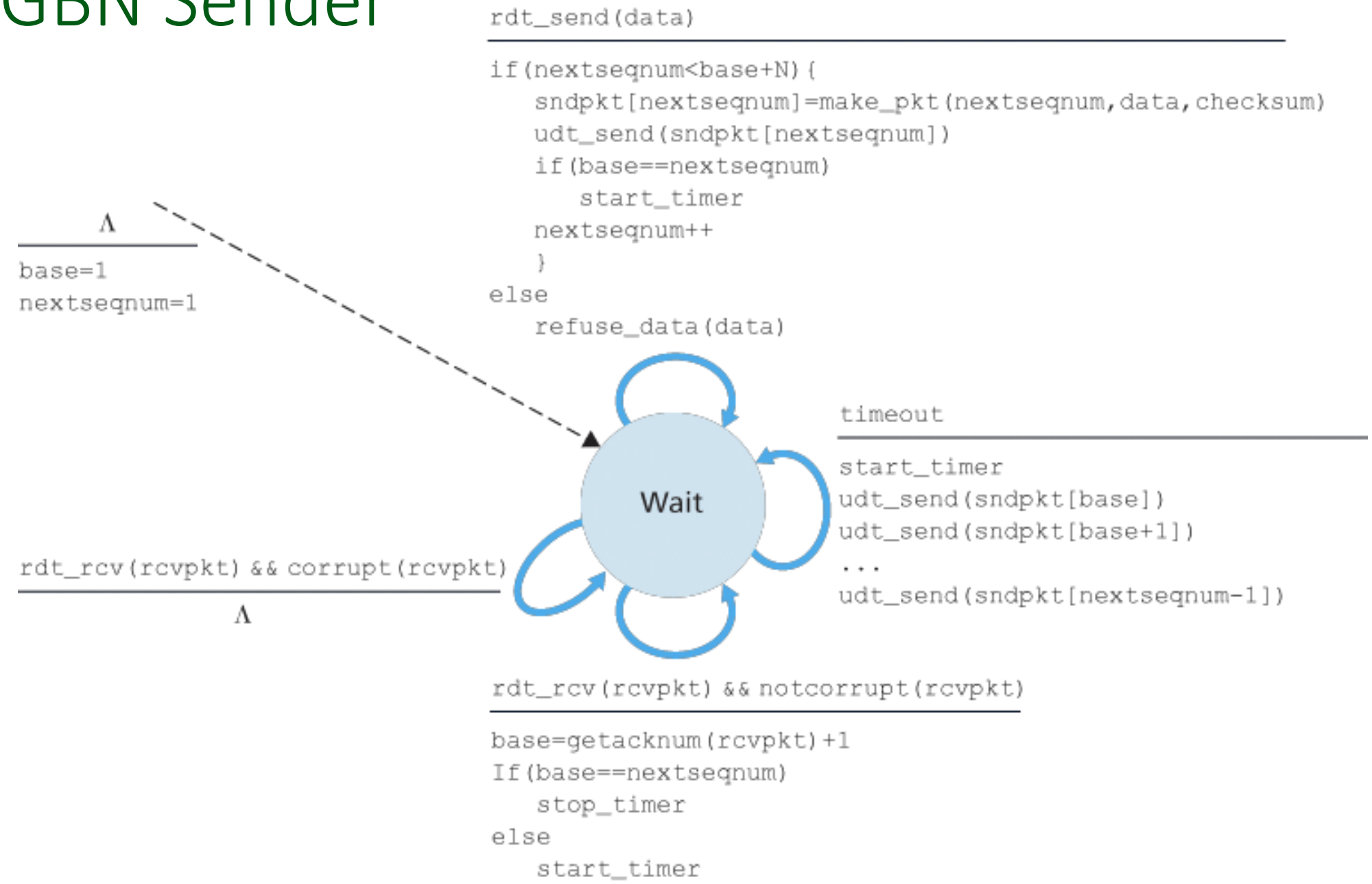
Realizing Sliding Window: Go-Back-n

- Sender:
- k-bit seq # in packet
- “window” up to N, conseq unack’d packets allowed

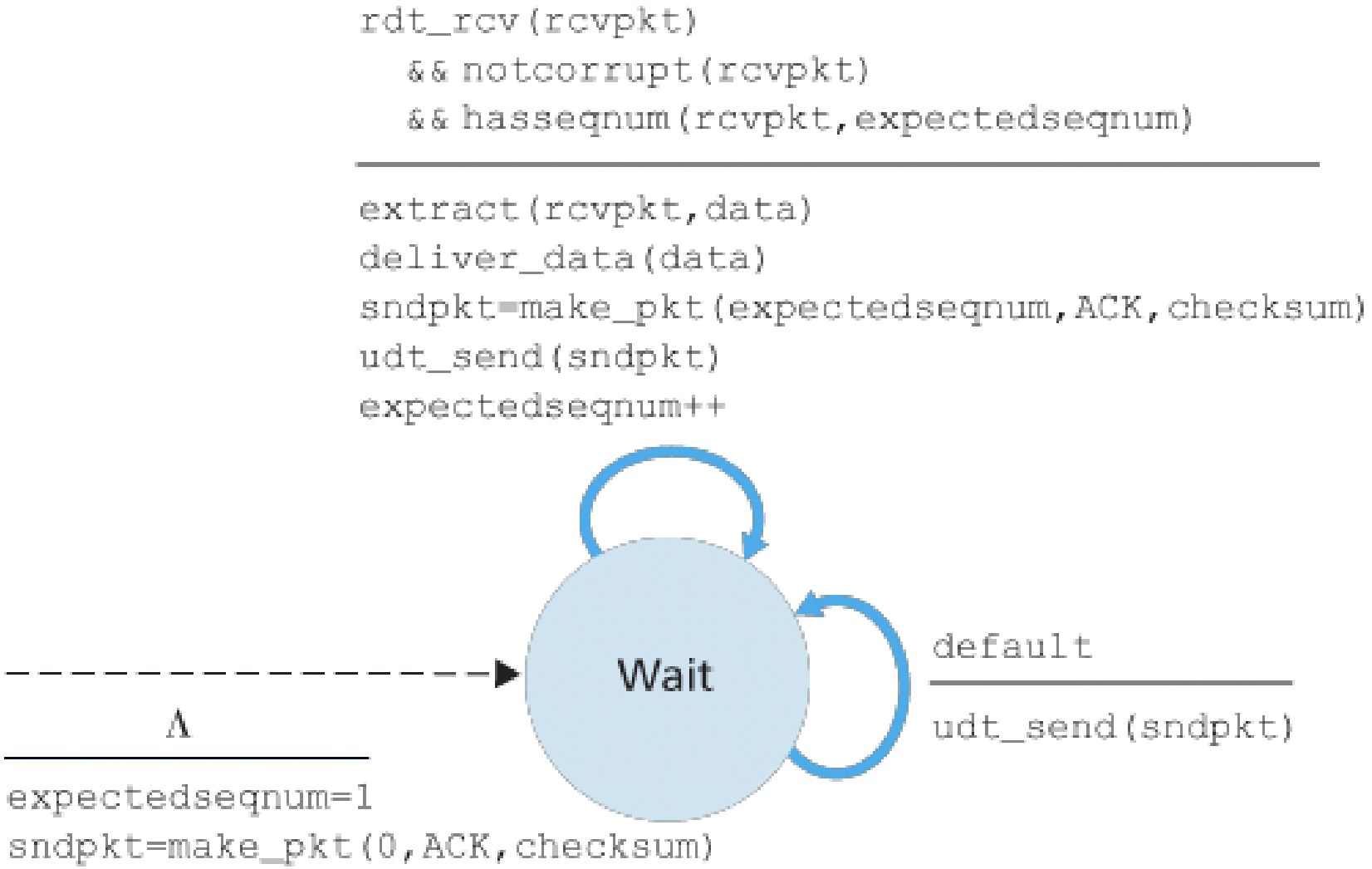


- ACK(n): ACK’s all packets up to n, including n. “cumulative ACK”
- Timer will run for the packet at the base.
- If timeout, resend base + all higher seq # packets in the window

GBN Sender

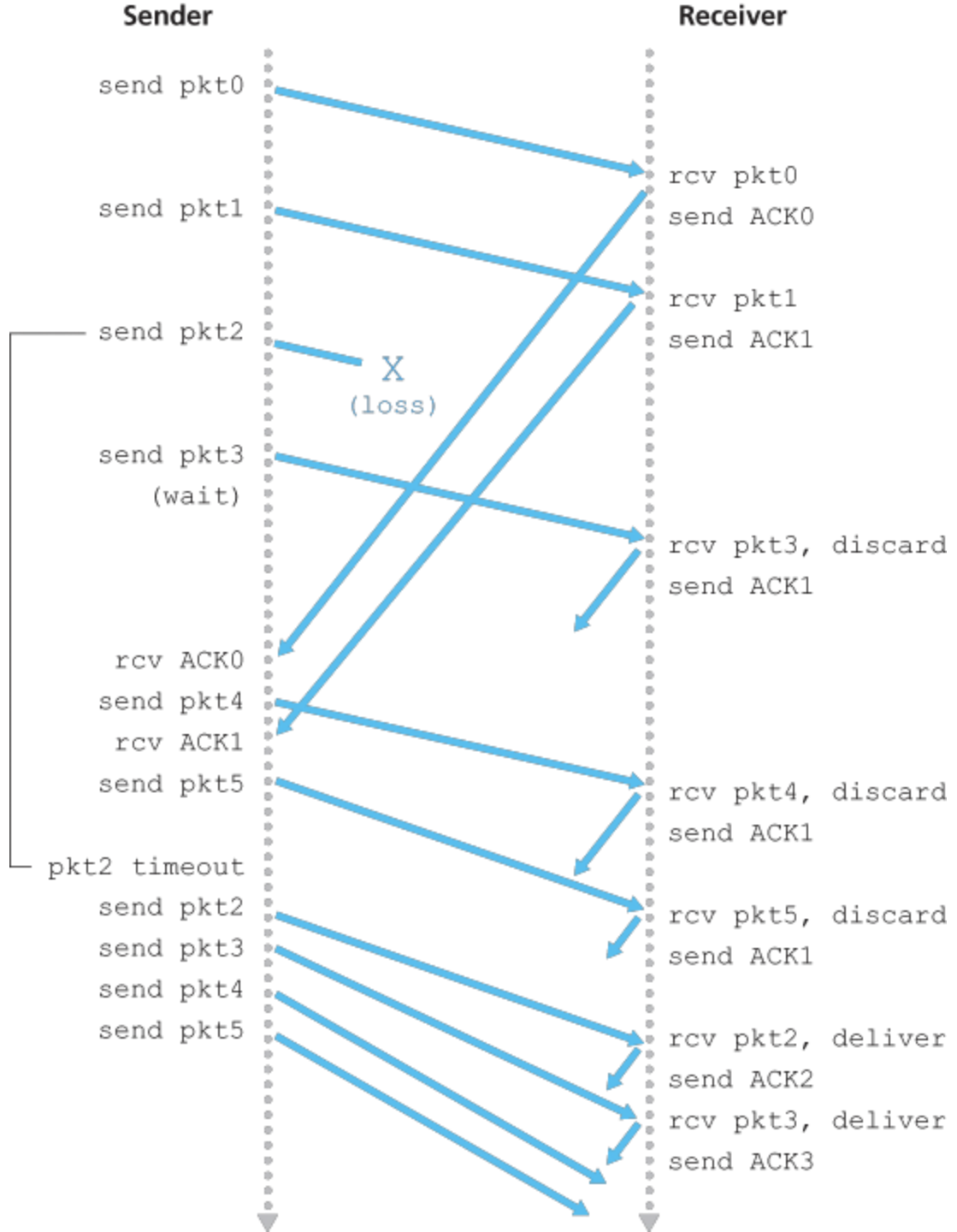

rdt_send(data)
if(nextseqnum<base+N){
sndpkt[nextseqnum]=make_pkt(nextseqnum,data,checksum)
udt_send(sndpkt[nextseqnum])
if(base==nextseqnum)
start_timer
nextseqnum++
}
else
refuse_data(data)
Λ
base=1
nextseqnum=1
Wait
timeout
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
...
udt_send(sndpkt[nextseqnum-1])
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



Flow Diagram

- Out of order packet:
  - Resend ACK for last in order recv'd
  - Discard packet! (No buffering)
  - May generate duplicate ACKs



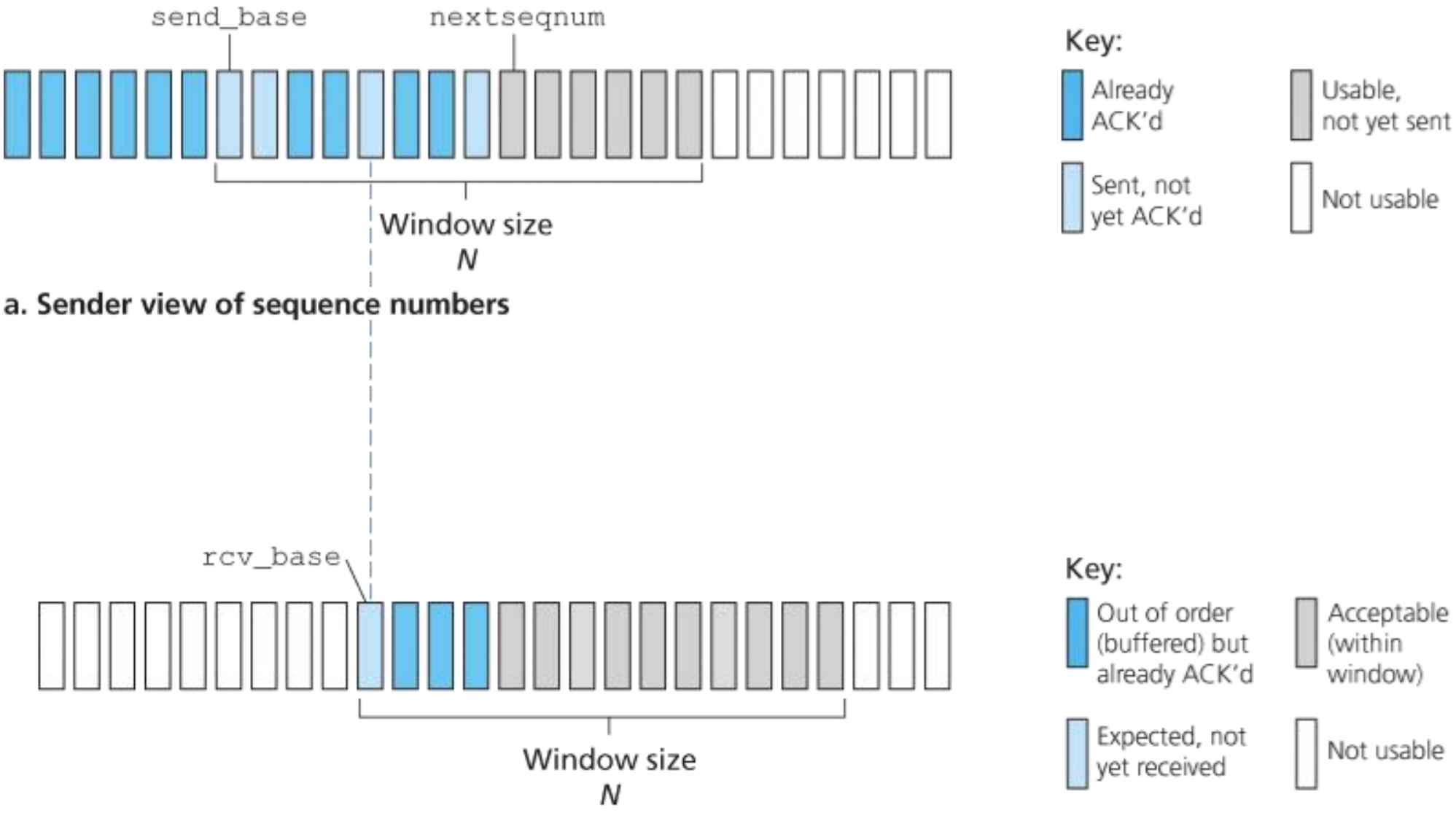
Sliding Window: Selective Repeat

- Sender Window: W just as before

- Receiver individually ACK's seq #
  - Buffer out of order packets
  - ACK(n) means ACK only individual packet n
  - Question: buffer size at the receiver?

- Sender only resends packets that have no ACK
  - Timers for all unACK'd packets

Sender and Receiver Windows



Selective Repeat

Sender

- Data from above
  - unACK'd packets is less than window size W, send packet and start timer
  - Otherwise, block app
- Timeout(n)
  - Resend packet n, restart timer
- ACK(n) where n inside W
  - Mark packet n was recv'd
  - Slide window to first unACK'd packet

Receiver

- Packet n where n is inside W
  - Send ACK(n)
  - If out of order
    - Mark recv'd on window
  - Else
    - Send all in-order packets to app
- Otherwise
  - Ignore

Summary of Reliable Data Transport Mechanisms

- Checksum
  - Detects flipped bits
- Timer
  - Used for timeouts/retransmits of packets (or their ACK) getting lost in the unreliable channel
  - Can cause duplicate transmissions
- Sequence Number
  - Helps the receiver identify out of order/duplicate packets
- ACKnowledgment
  - Individual vs. Cumulative
  - Let’s the receiver know that a packet or set of packets has been received
  - Typically carries the sequence number

Summary of Reliable Data Transport Mechanisms

- Negative ACKnowlegment
  - Informs the receiver a malformed packet has arrived
  - Typically contains the sequence number
- Window, pipelining
  - Allows multiple packets to be transmitted to avoid stop-and-wait
  - Sender restricts number of packets sent with use of a window
  - The size of the window will depend on the receivers ability to buffer messages, or the level of network congestion, or both.

TCP

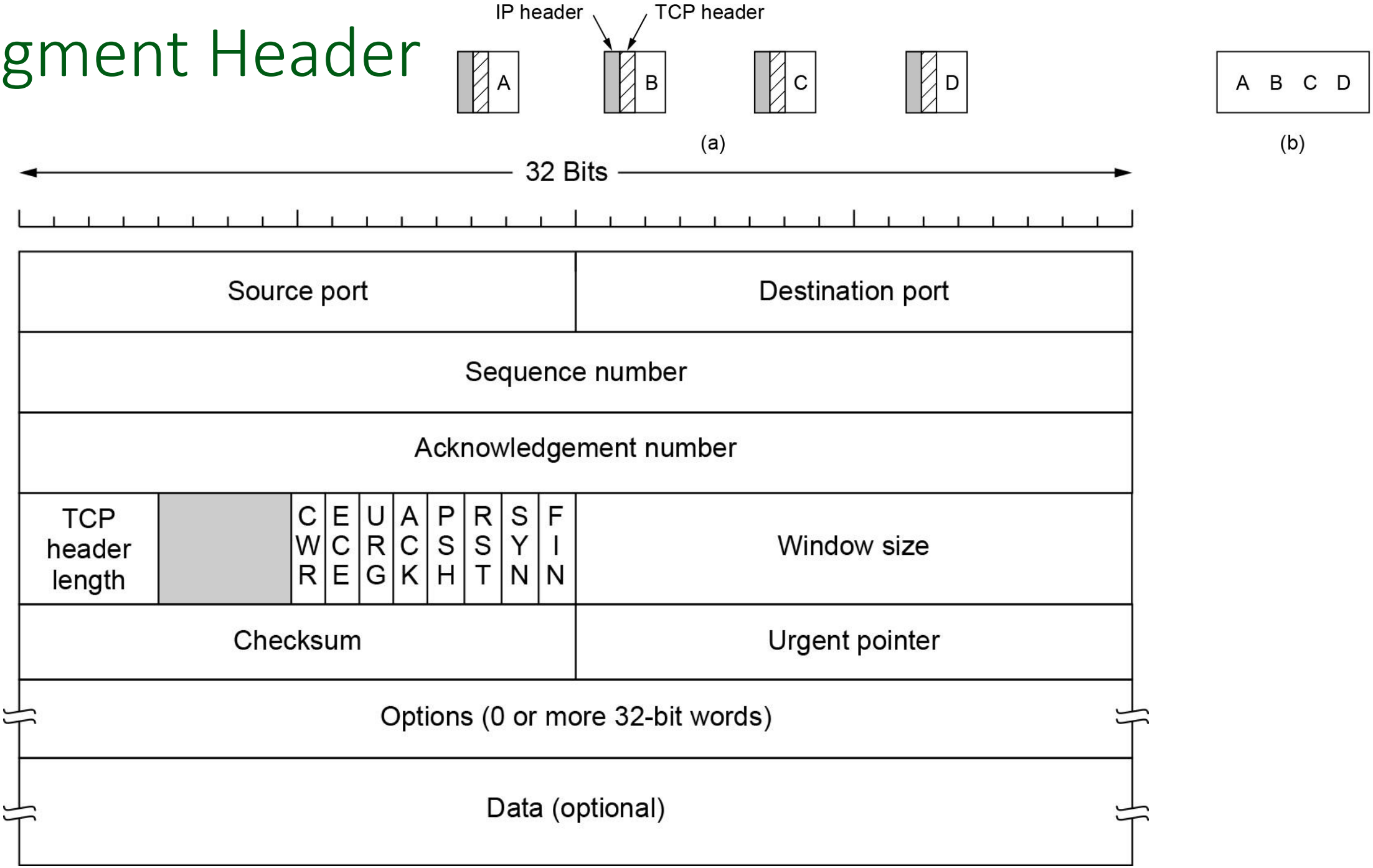
History of TCP

- 1981 – RFC 793
- RFC 1323, clarifications and bug fixes
- RFC 1112, extensions for high performance
- RFC 2018, selective acks added
- RFC 2581, congestion control
- RFC 2873, reproposing headers for quality of service
- RFC 2988, improved retransmission timers
- RFC 3168, explicity congestion notification
- RFC 4614, a guide to all of the RFCs for TCP

Modern TCP

- RFC 7414, updated roadmap for TCP
  - Which is updated by RFC 7805
- Lists 8 RFC’s for the core functionality of TCP
- And 14 RFC’s that are strongly recommended
- And another 20+ RFC’s for various things

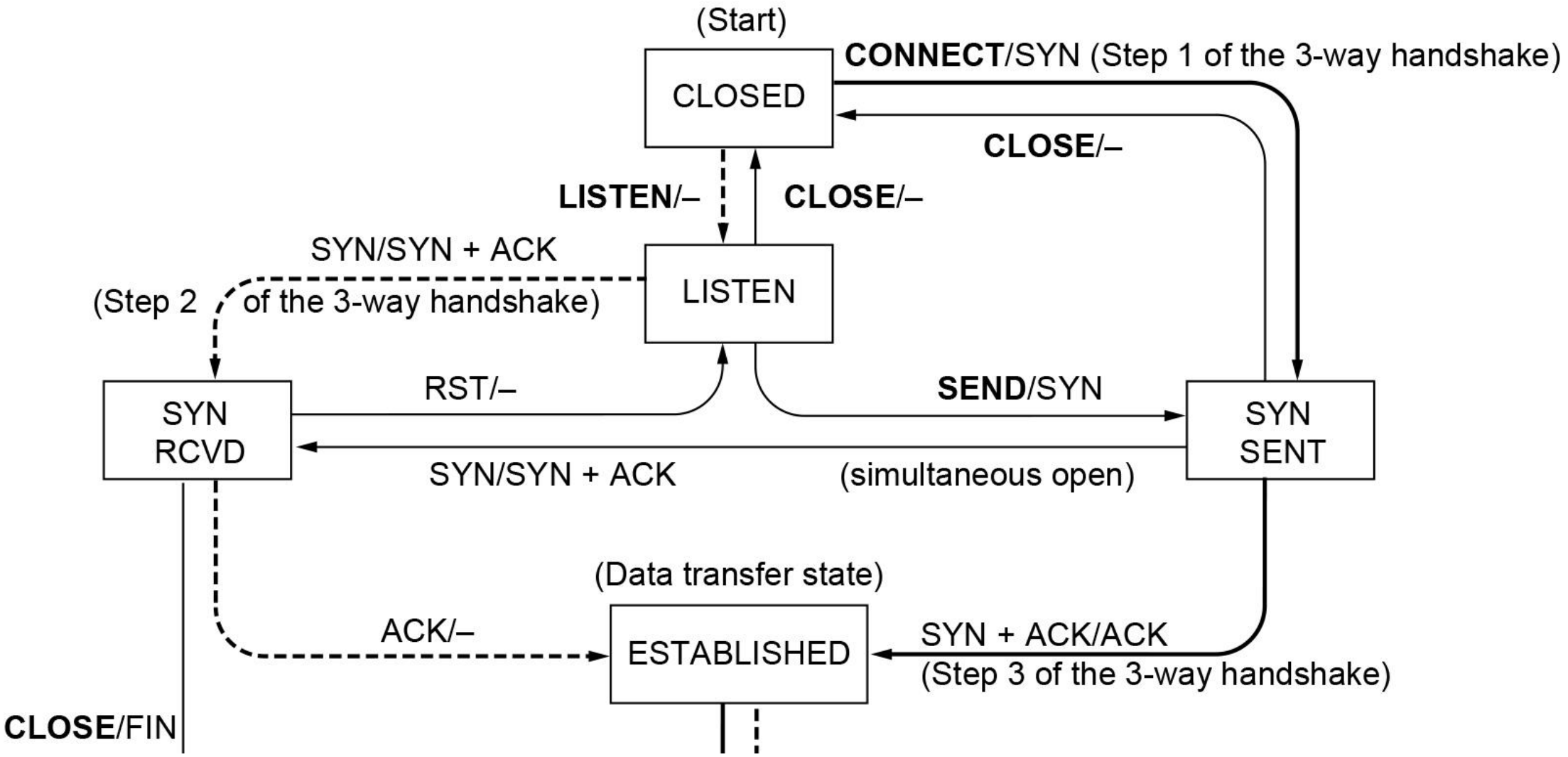
TCP Segment Header



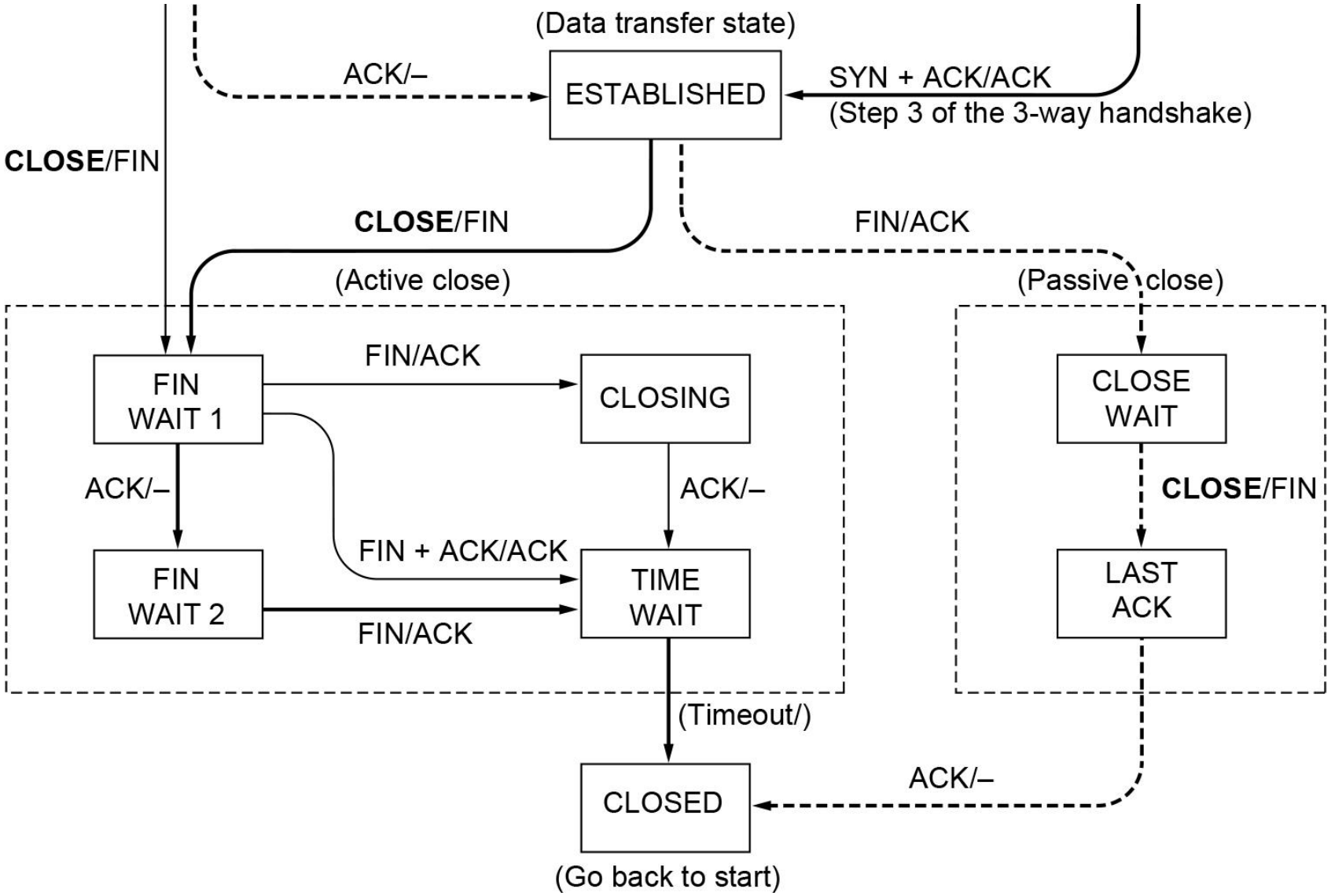
Many States of TCP

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

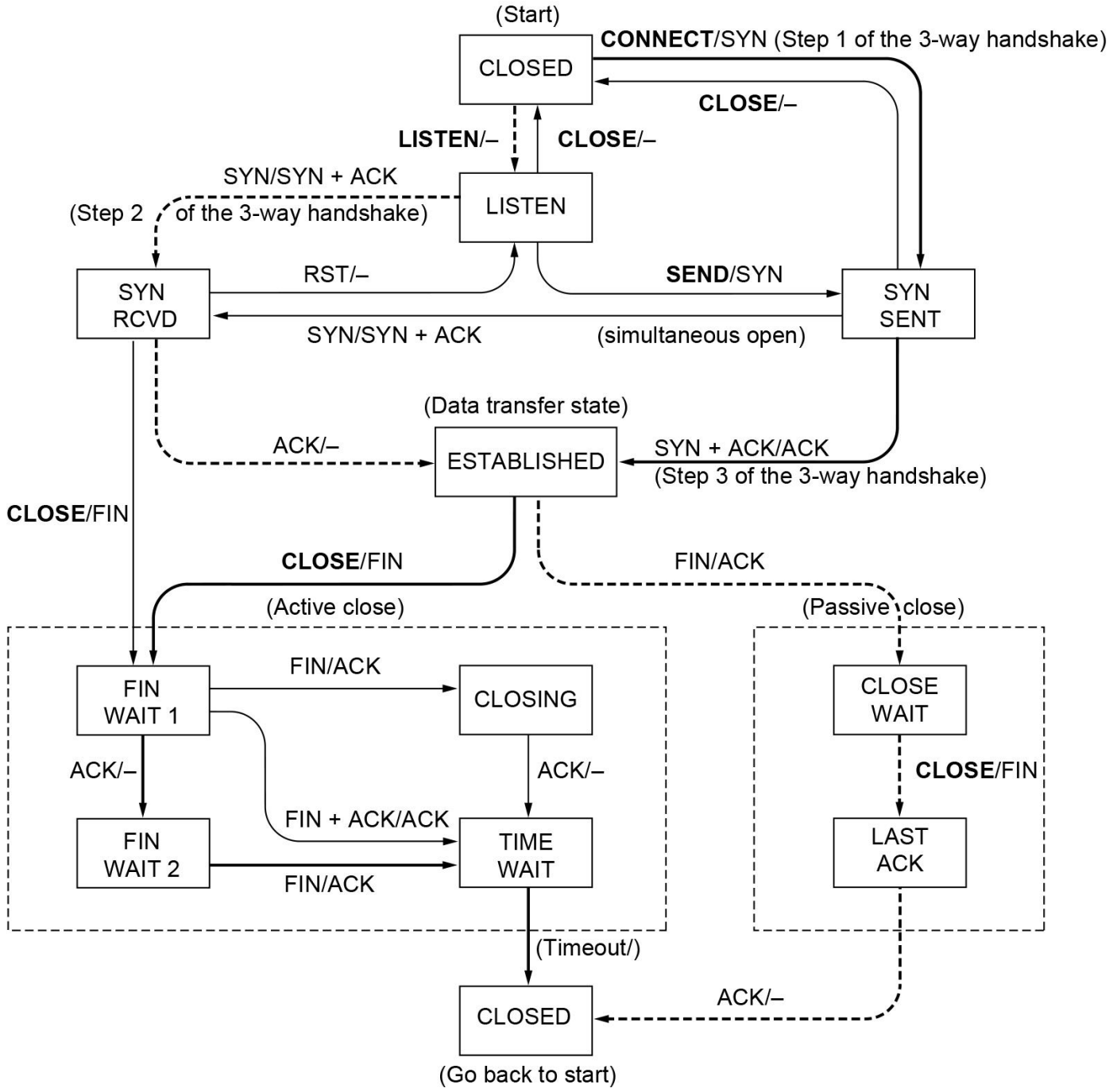
TCP State Machine (Opening/Estab half)



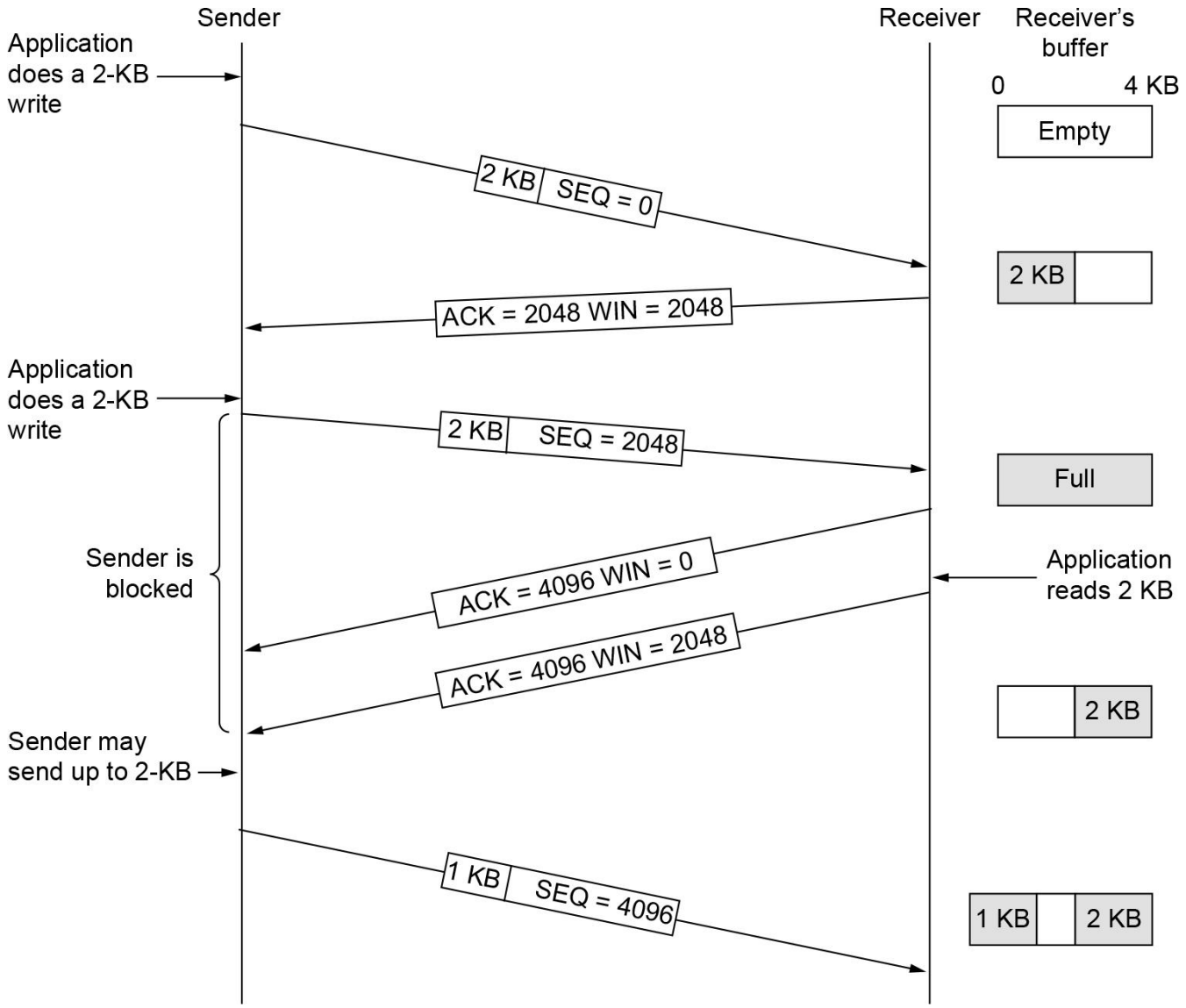
TCP State Machine (Estab/Closing Half)



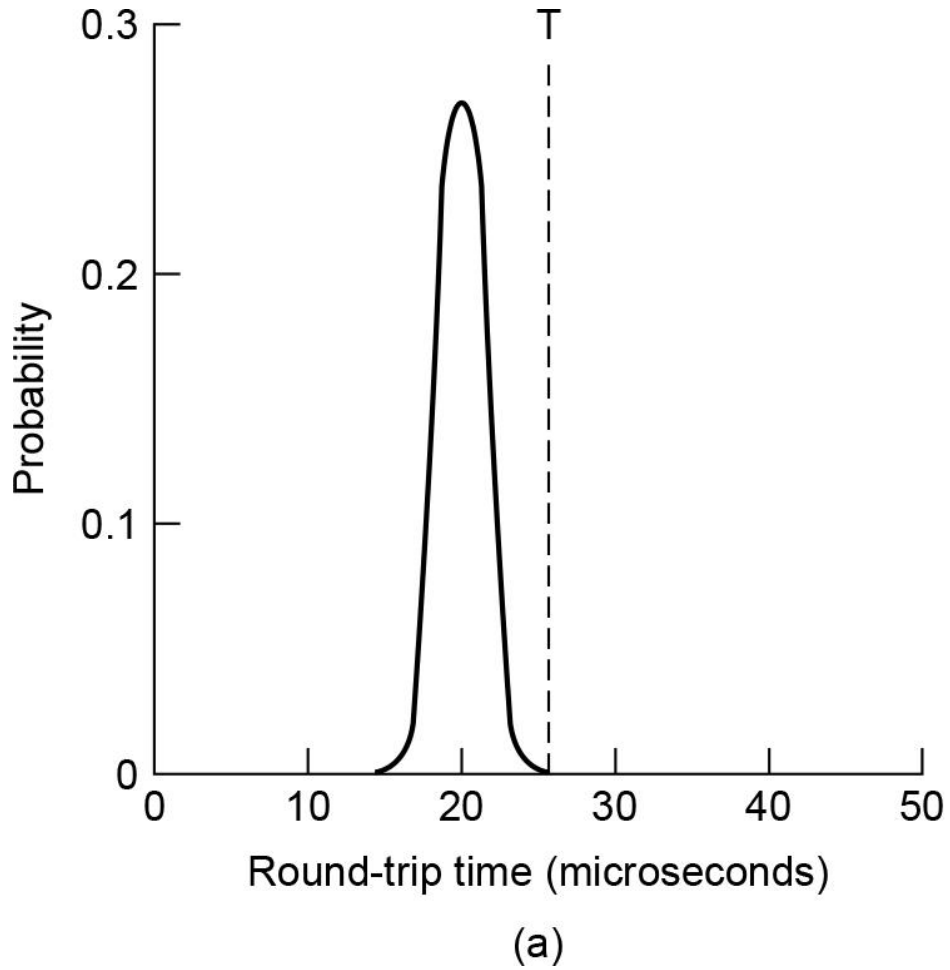
TCP State Machine (full view)

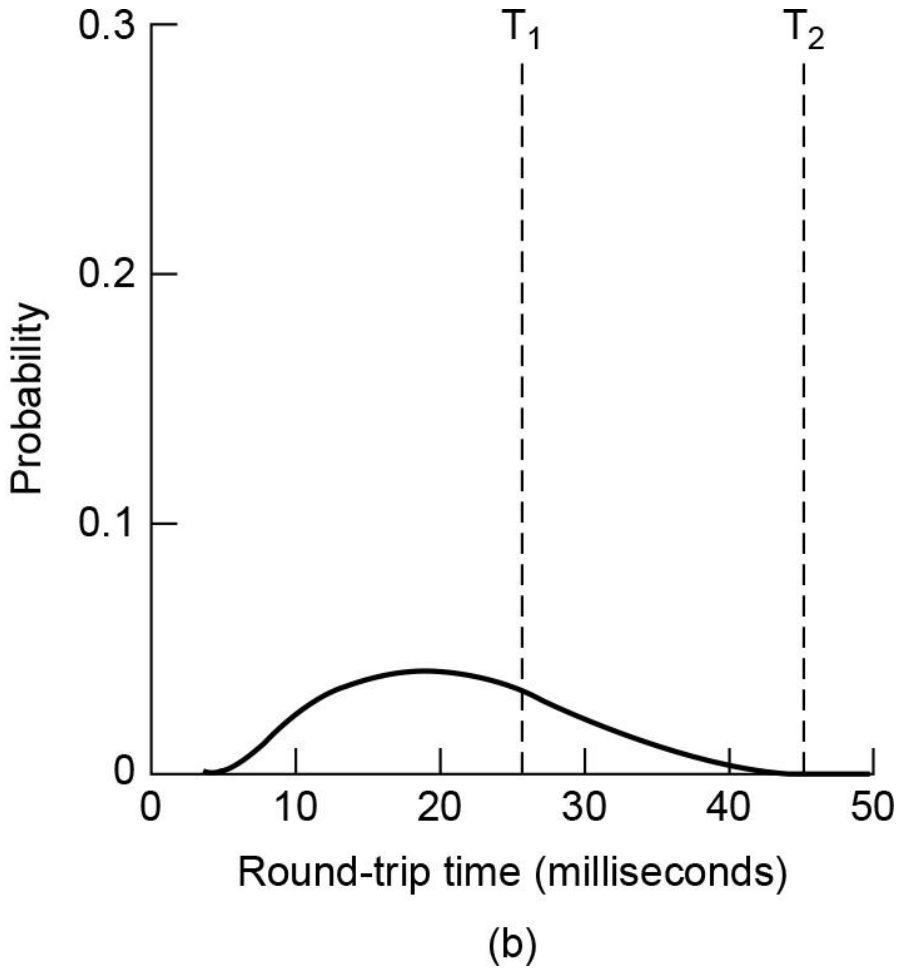


Sliding Window



TCP Timers


0.3
0.2
0.1
0
Probability
T
0
10
20
30
40
50
Round-trip time (microseconds)
(a)


0.3
0.2
0.1
0
Probability
T_1
T_2
0
10
20
30
40
50
Round-trip time (milliseconds)
(b)

Standard Method: mean deviation

- $Err = M - srtt$
- $srtt \leftarrow srtt + g(Err)$
- $rttvar \leftarrow rttvar + h(|Err| - rttvar)$
- **RTO** $= srtt + 4(rttvar)$

- Err – Difference between new measurement and previous
- M – new sample RTT (RTT_s previously)
- srtt – smoothed RTT
- g – gain, usually 1/8
- h – gain, usually 1/4
- *Essentially our rttvar is a better version of* β

Simple Implementation of Standard Method

```
void
rtt_update(struct rtt_info* rtt, int ms)
{
        int delta = ms - rtt->srtt;
        rtt->srtt += delta / 8;
        if(delta < 0)
                delta = -delta;
        rtt->rttvar += (delta - rtt->rttvar) / 4;
        rtt->rto = rtt->srtt + 4*rtt->rttvar;
}
```

$$Err = M - srtt$$

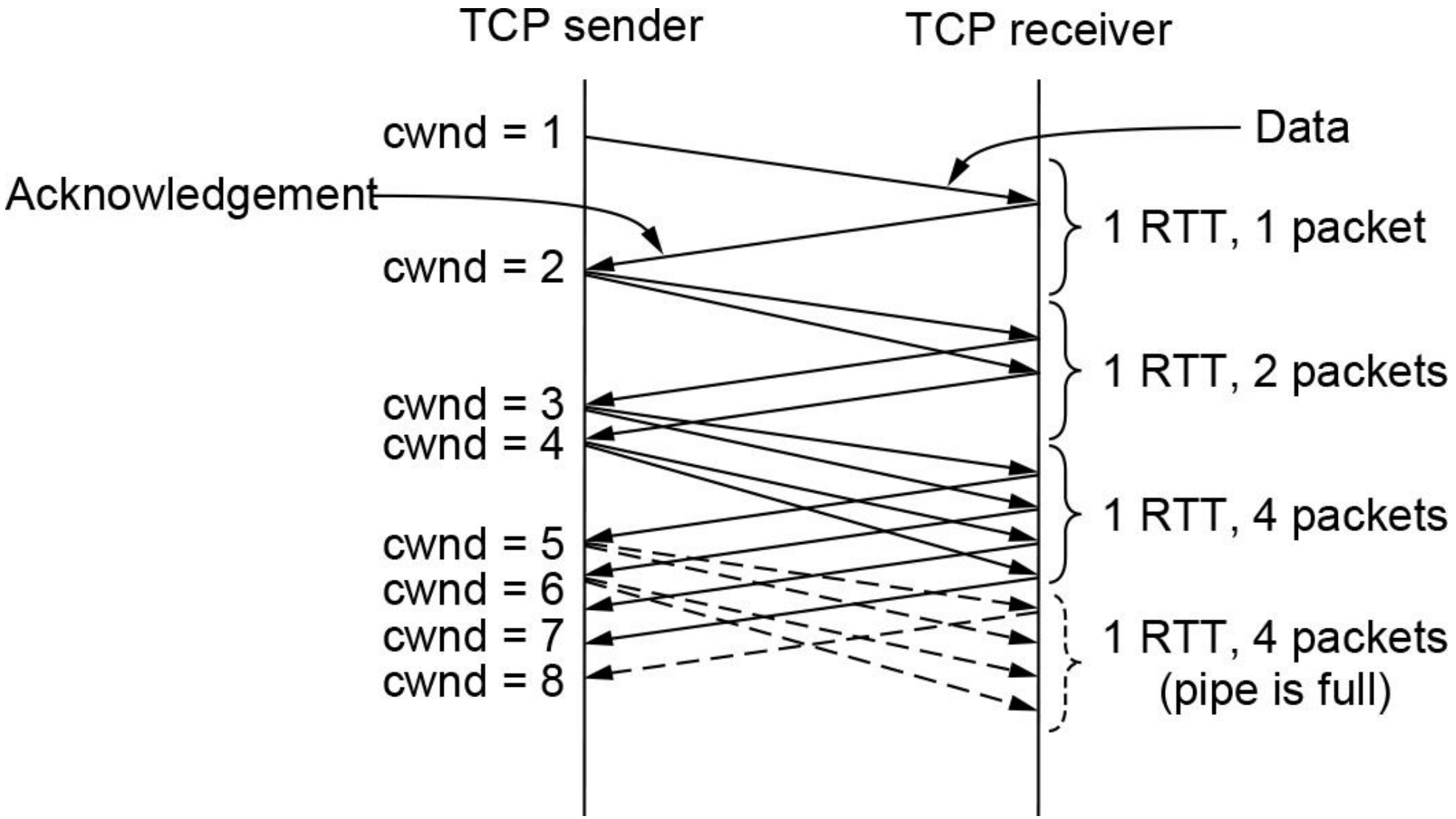
$$srtt \leftarrow srtt + g(Err)$$

$$rttvar \leftarrow rttvar + h(|Err| - rttvar)$$

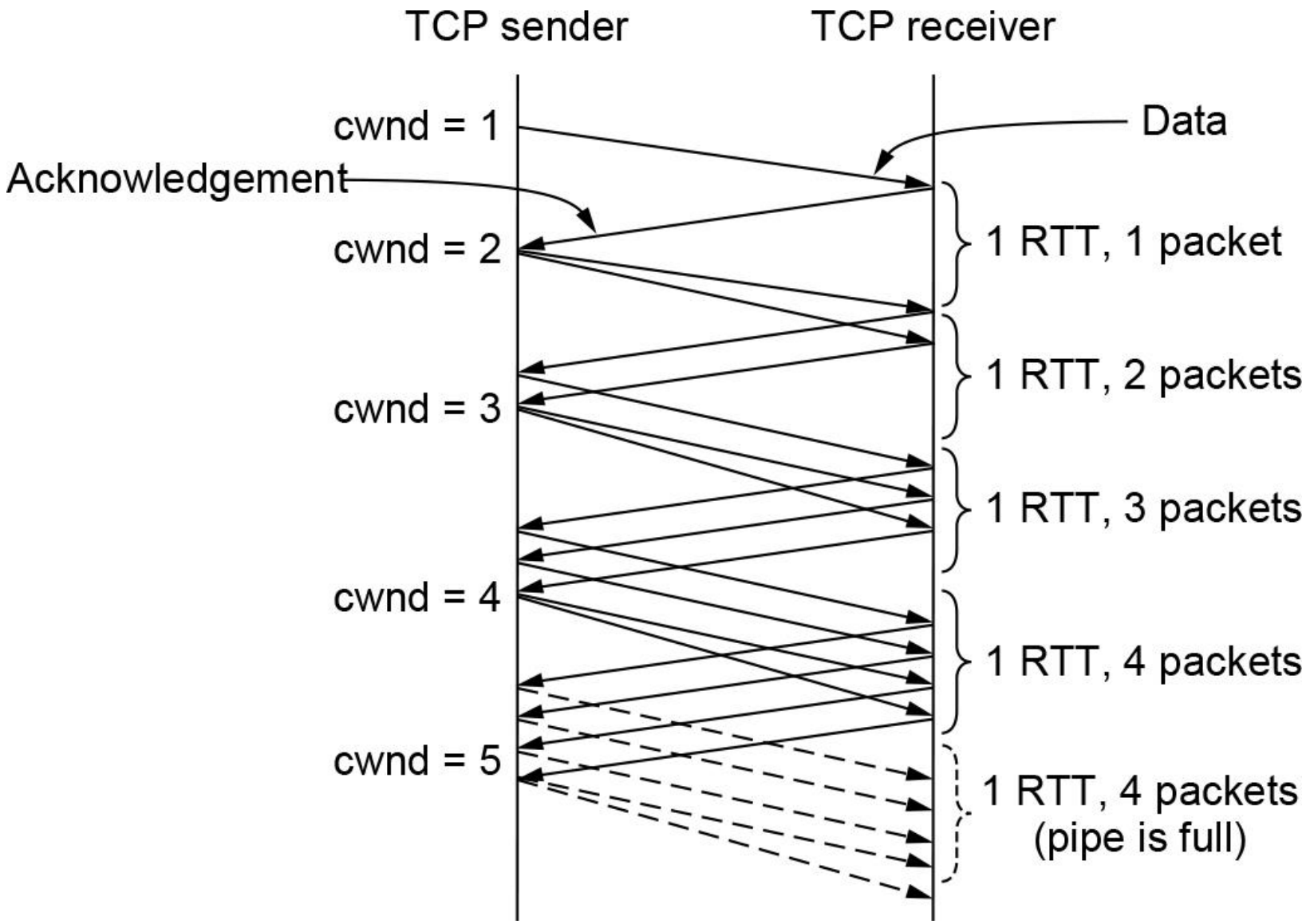
$$RTO = srtt + 4(rttvar)$$

- FreeBSD
  - https://github.com/freebsd/freebsd/blob/0b62c2b8b2539c9a06711e1bc40a4e5cdea0ac07/contrib/unbound/util/rtt.c
  - Interestingly enough, this code references our textbook:
  - `/* From Stevens, Unix Network Programming, Vol1, 3rd ed., p.598 */`

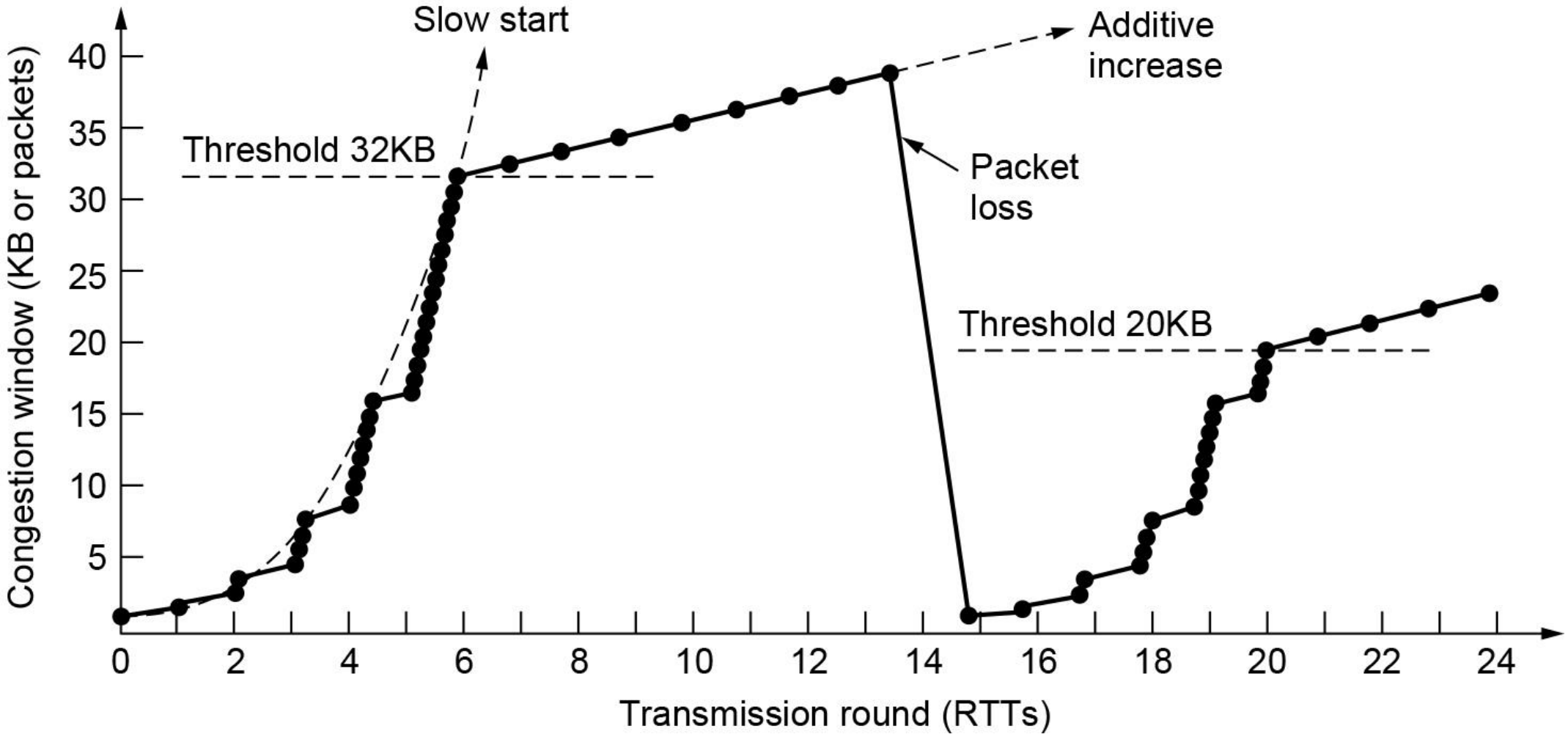
Congestion Control: Slow Start



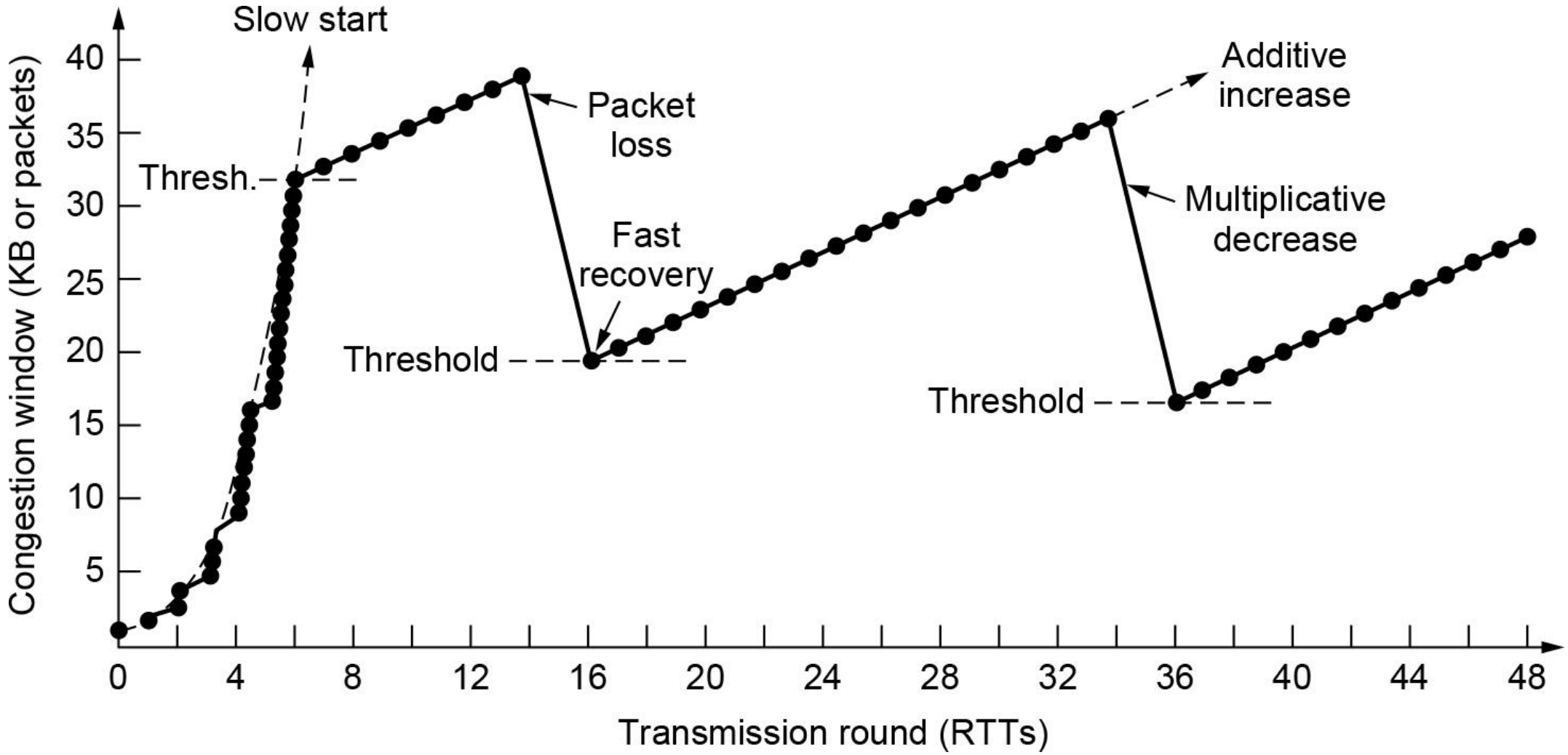
Congestion Control: Additive increase



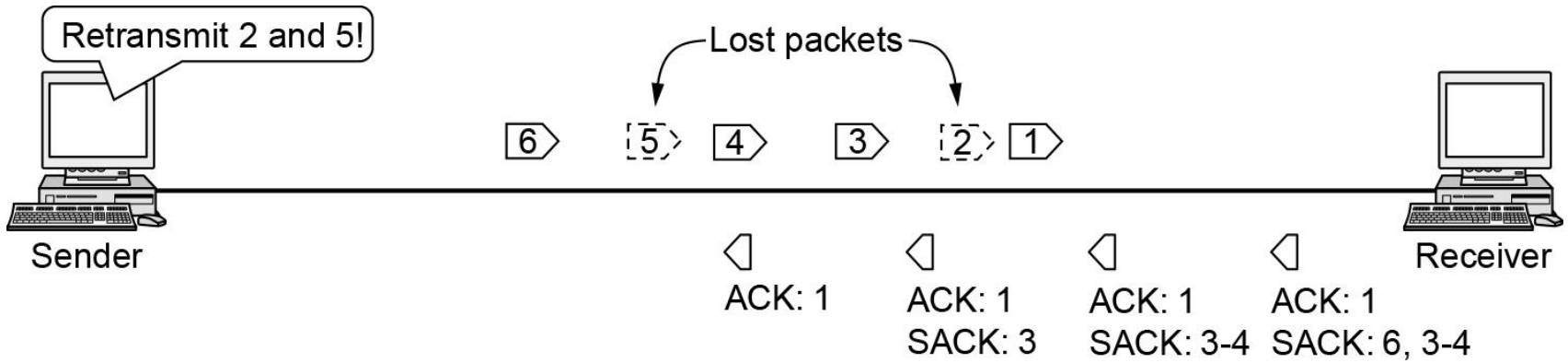
Congestion Control: packet loss is bad



Congestion Control: Fast Recovery



Selective Acknowledgements



TCP Lessons

- Recall how many RFCs we saw at the beginning
- TCP will continue to evolve
- In a way, it already has with QUIC

- “if it ain’t broke, don’t fix it” does not apply to TCP
- We are programmers
- “if it ain’t broke, [x]”
  - Change the settings
  - Make a better one
  - Say the new technology is better and that other people are dumb