Reliable data transfer - Noisy channel

CE 352, Computer Networks

Salem Al-Agtash

Lecture 9

Recap

- Transport protocols:
  - Reliable channel – Simple, rdt1.0
  - Unreliable channel – Stop-and-Wait
    - rdt2.0: channel with bit errors
    - rdt2.1: distorted ACK/NAK)
    - rdt2.2: NAK – free protocol

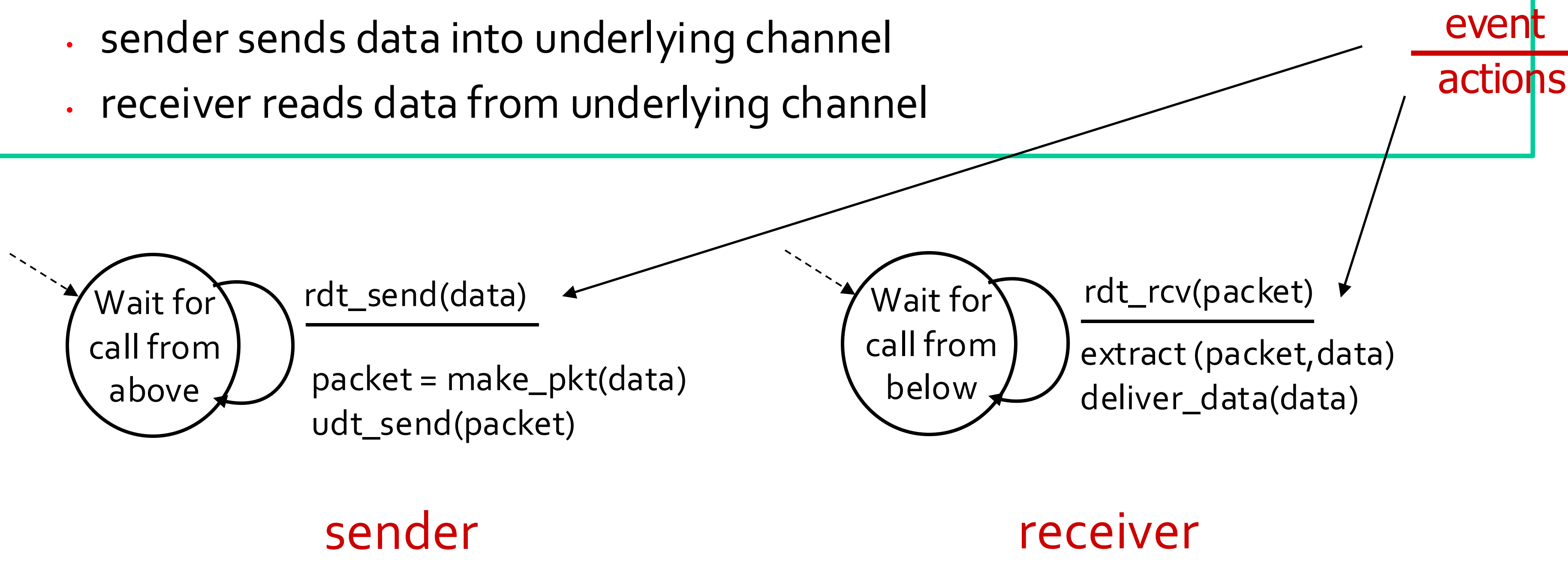
Today

    - rdt3.0: channel with errors and loss
  - Unreliable channel - pipelined protocols
    - Go-Back-N
    - Selective repeat
  - TCP protocol

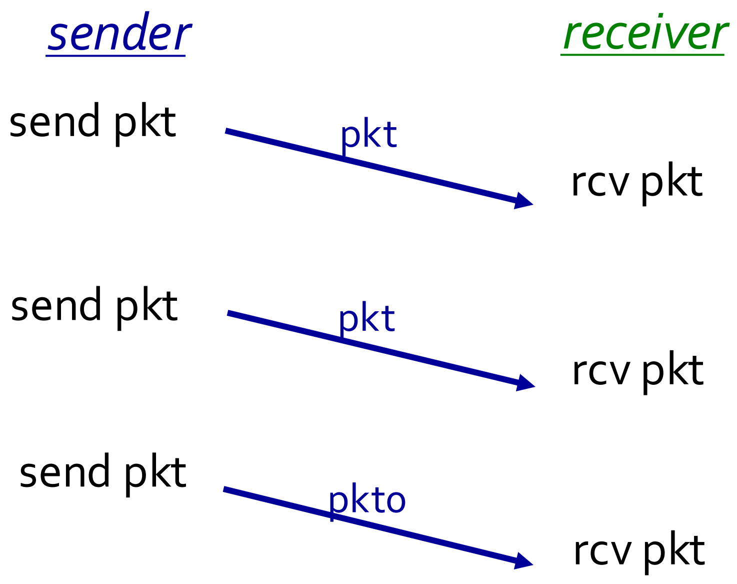
rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

Simple
provides neither
flow nor error control

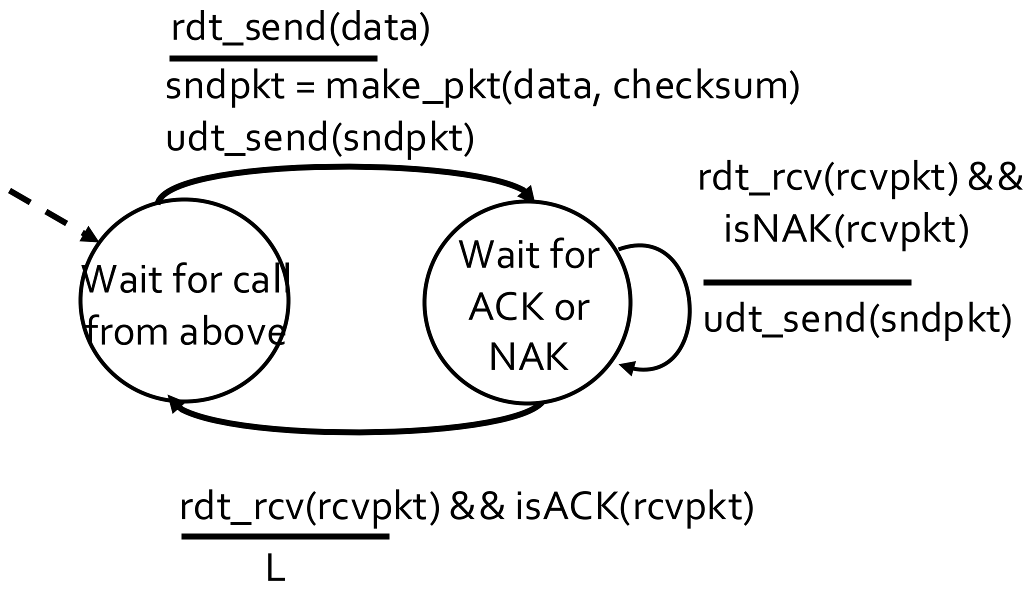


rdt1.0 in action

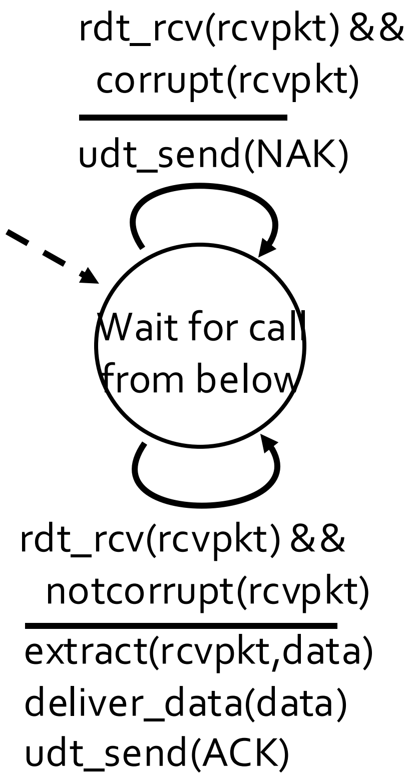


No error and no loss (reliable communication channel)

rdt2.0: channel with bit errors (unreliable channel)


rdt_send(data)
sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isACK(rcvpkt)
L

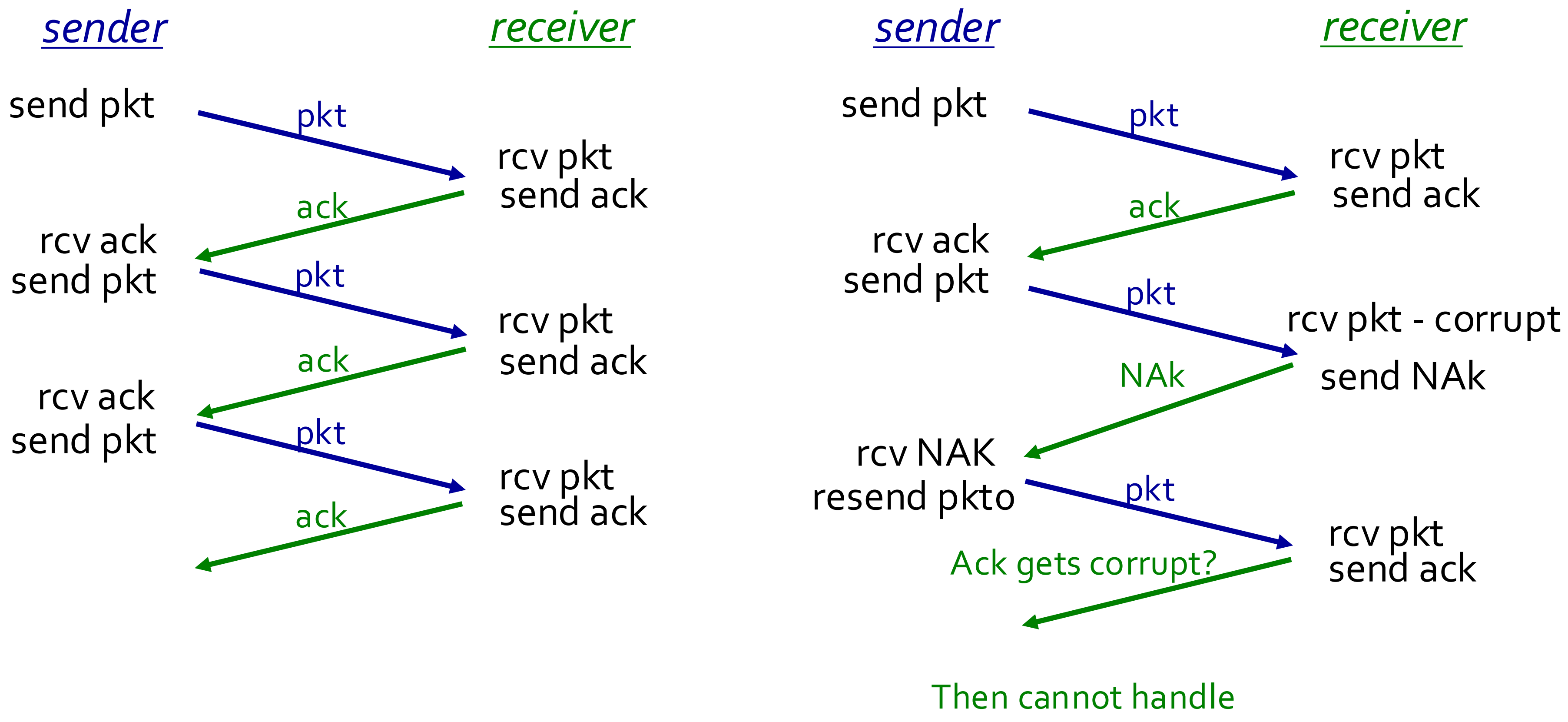
receiver



sender

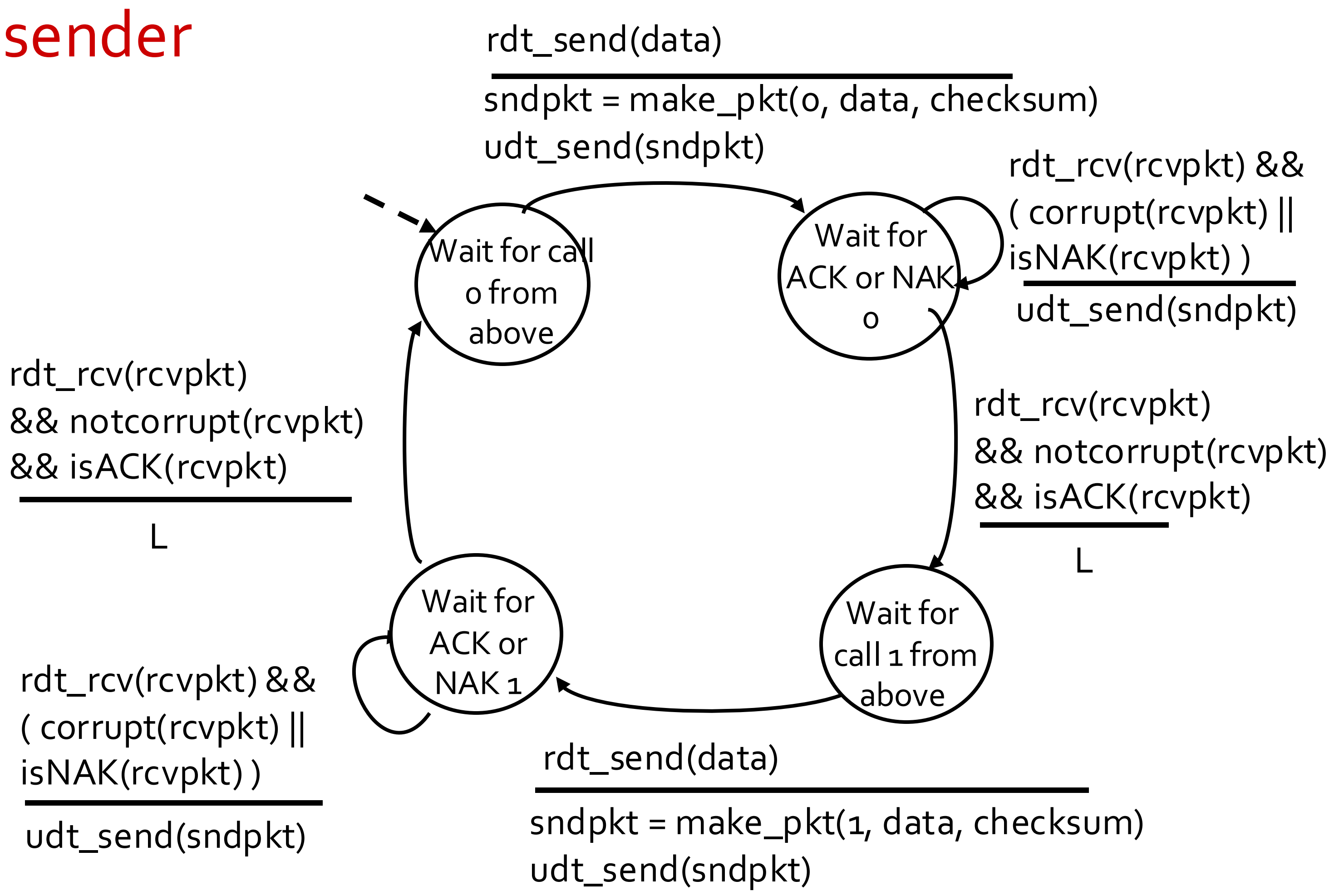
stop and wait

rdt2.0 in action

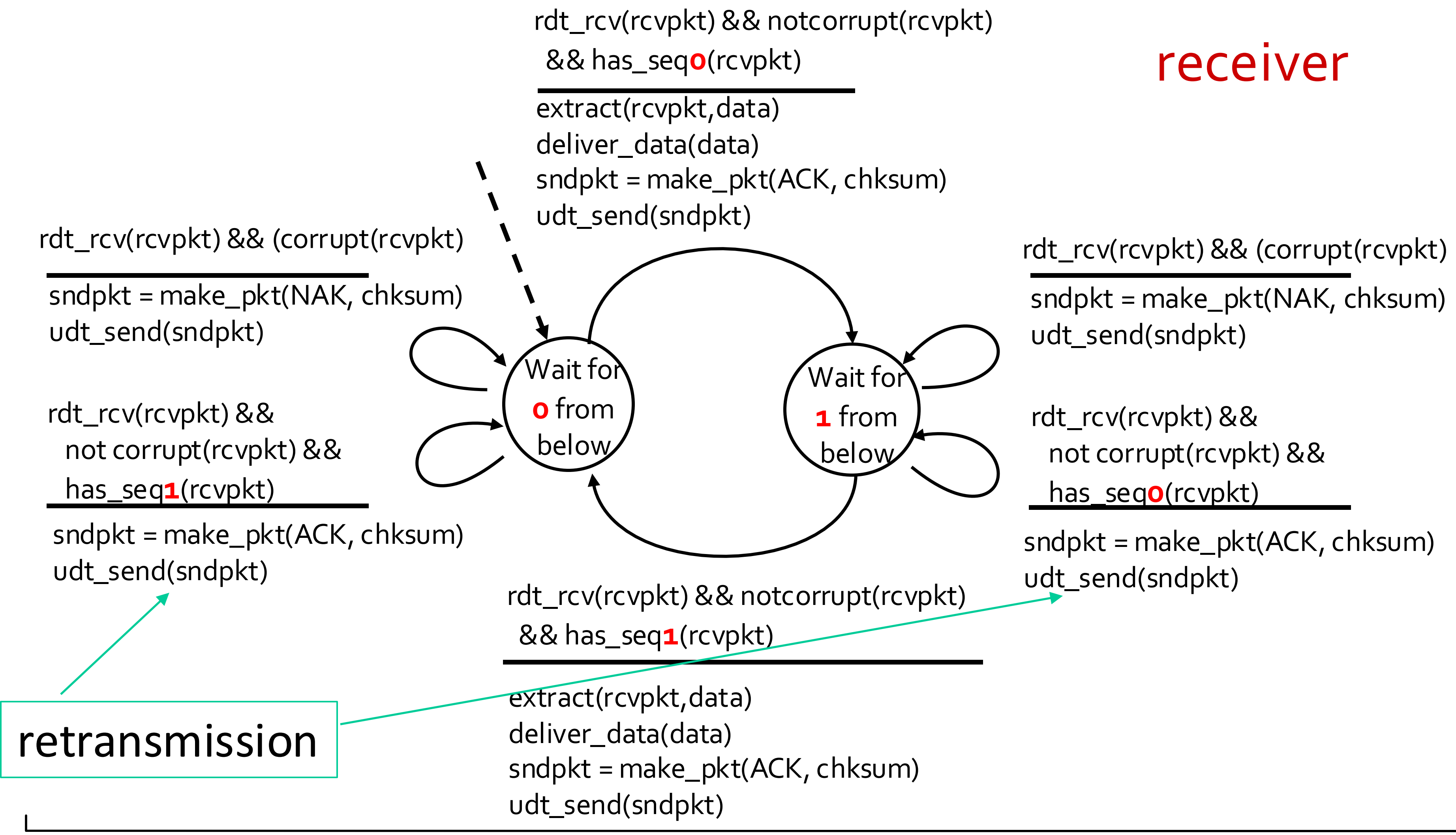


Possible error in Ack but still assuming no loss of packets

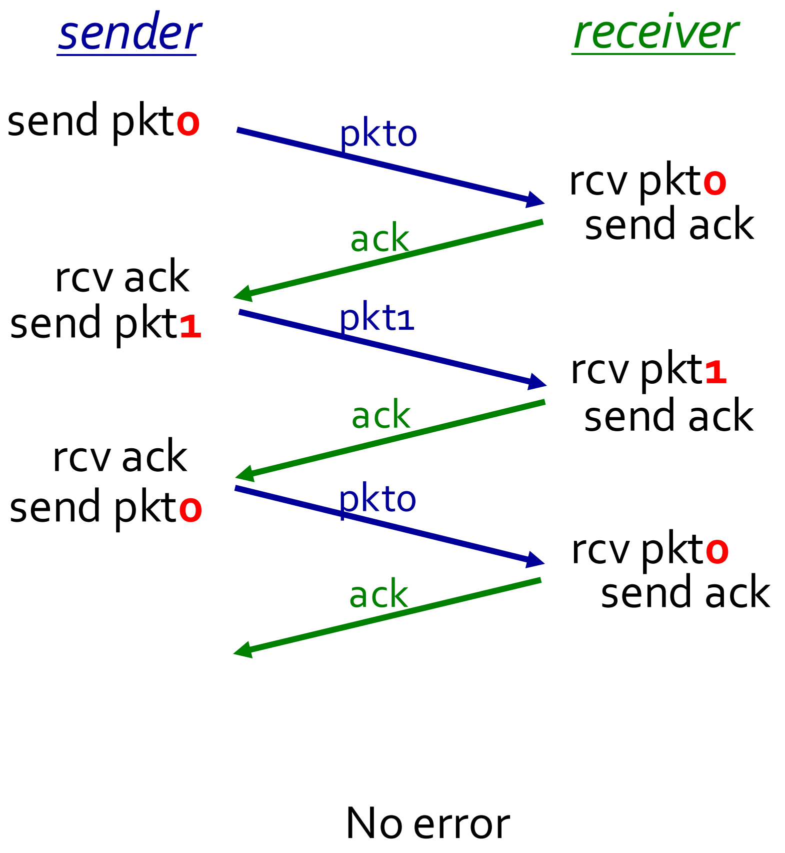
rdt2.1: sender handles distorted ACK/NAKs


sender
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
L
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
L
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

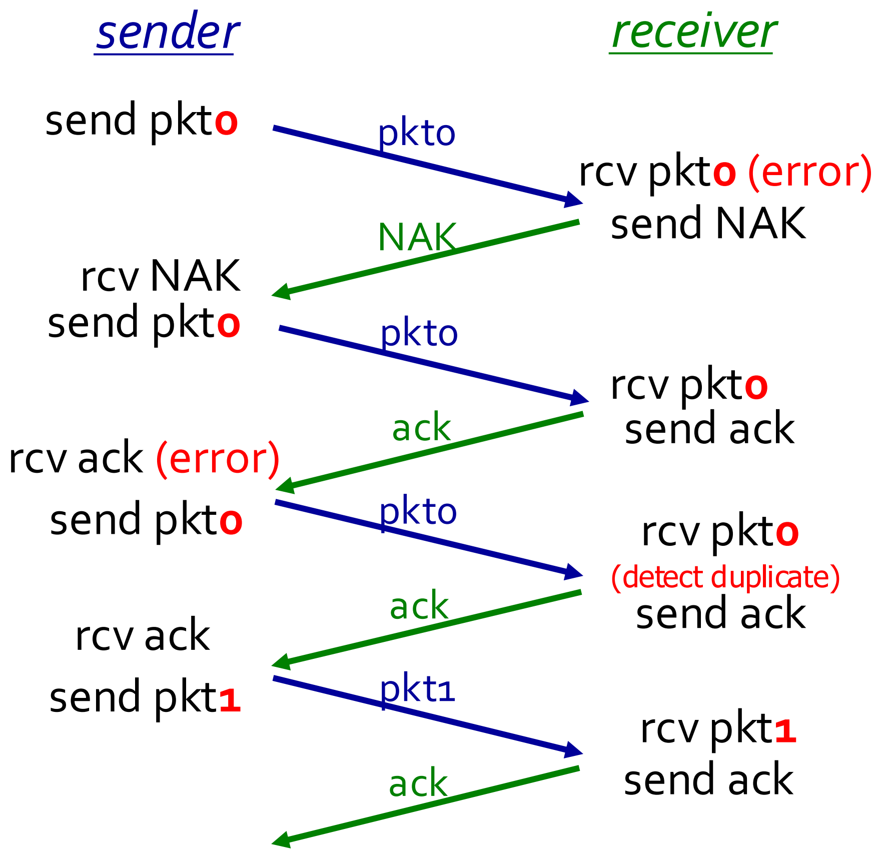
rdt2.1: receiver handles distorted ACK/NAKs


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
receiver
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, 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)
retransmission

rdt2.1 in action



No error

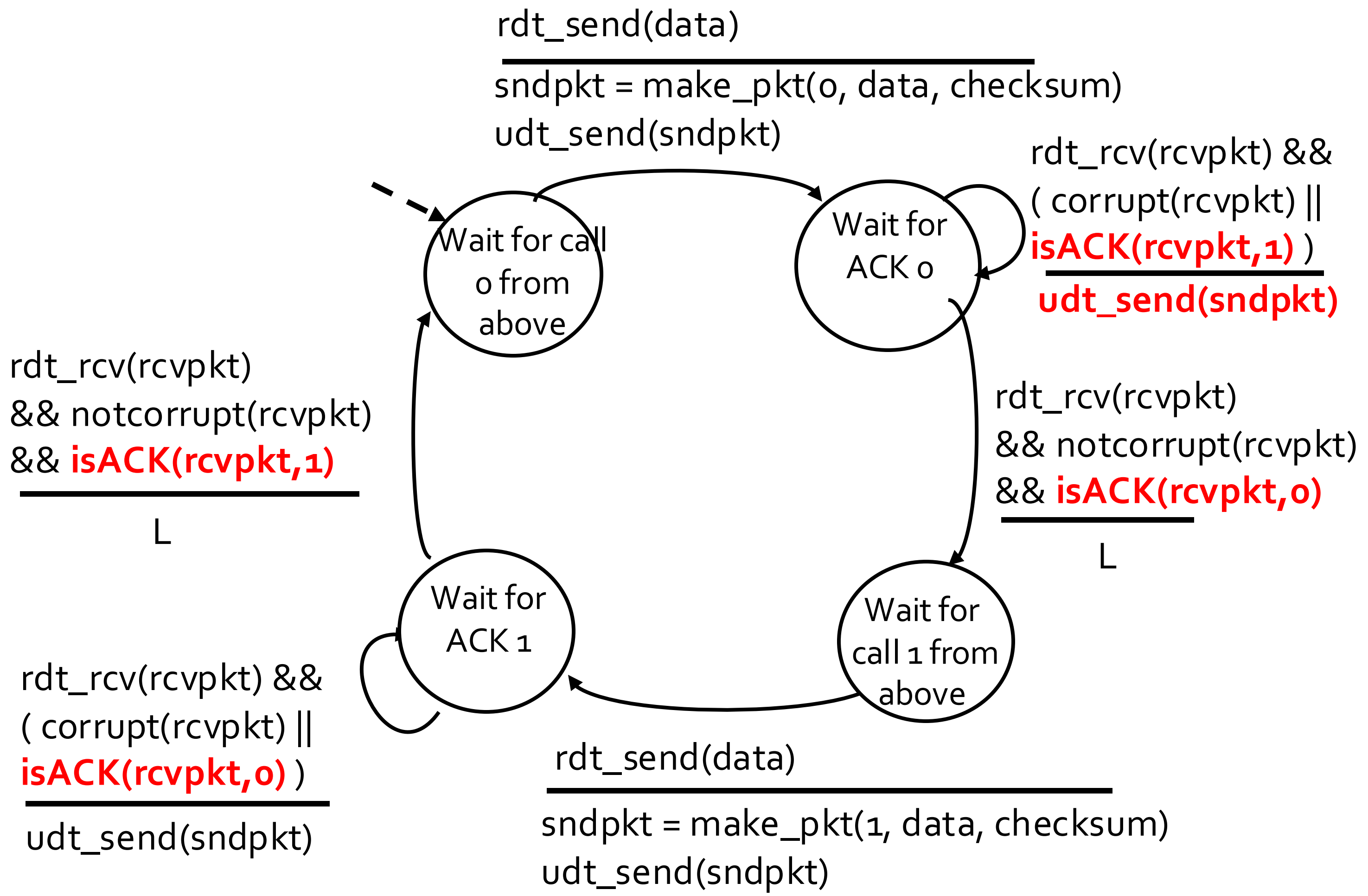


Error in pkt or ack/nak

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, 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: NAK-free sender


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
L
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
L
Wait for ACK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

rdt2.2: NAK-free receiver

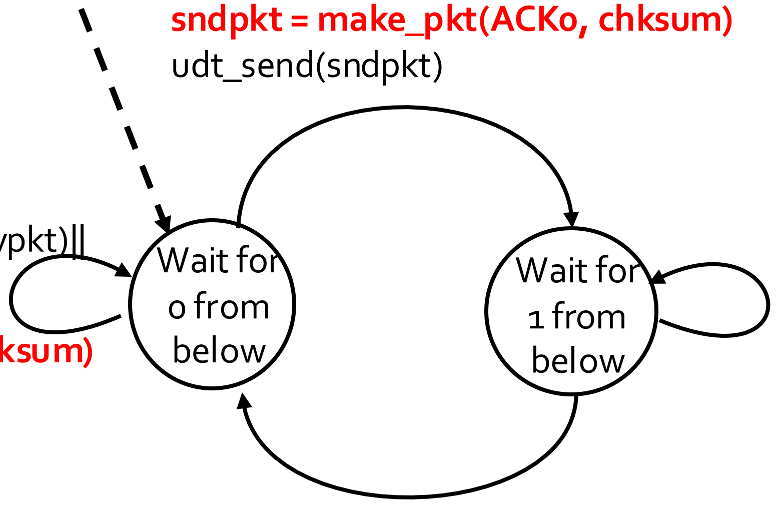
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK0, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)||
has_seq1(rcvpkt))

Sndpkt=make_pkt(ACK,1,chksum)
udt_send(sndpkt)



rdt_rcv(rcvpkt) && (corrupt(rcvpkt) ||
has_seq0(rcvpkt)

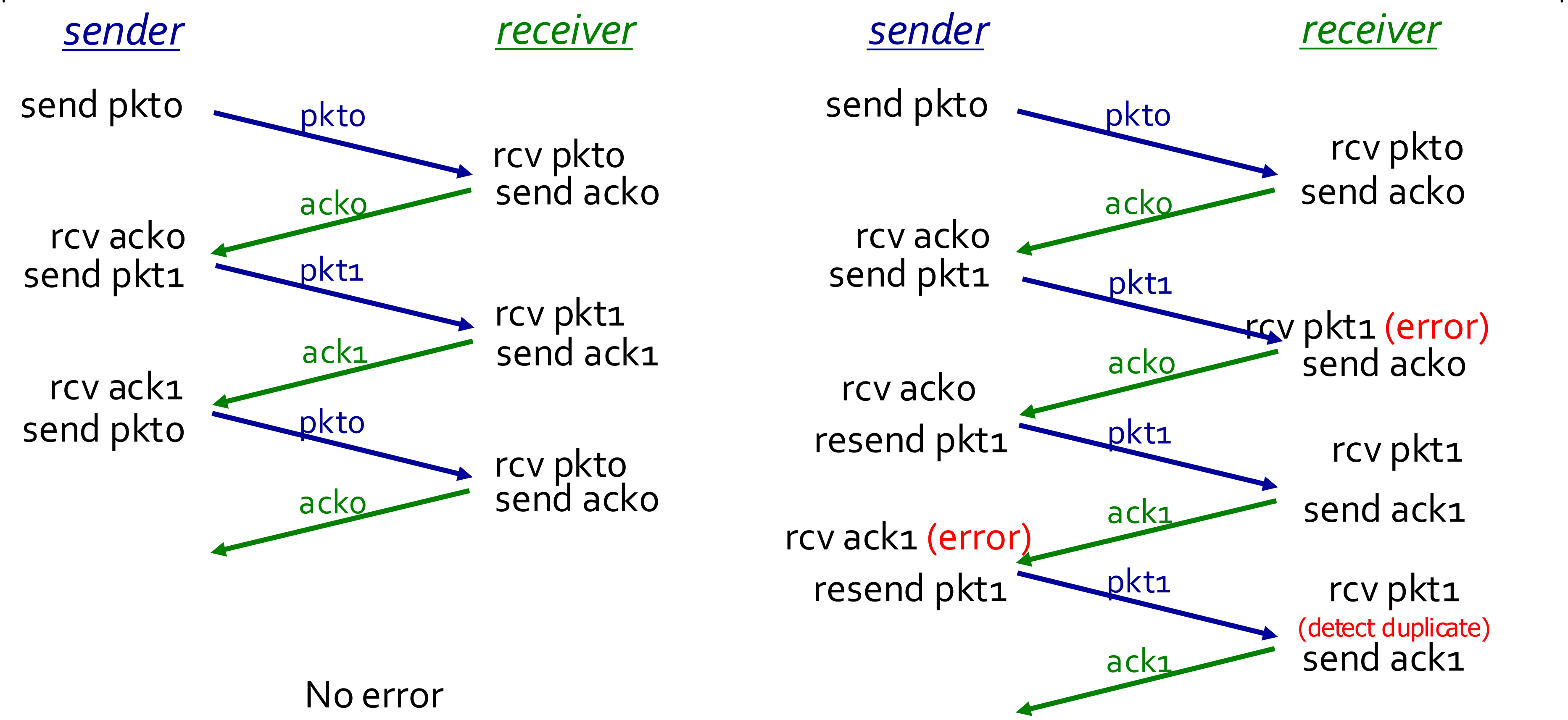
Sndpkt=make_pkt(ACK,0,chksum)
udt_send(sndpkt)

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

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

rdt2.2 in action



No error

Error in pkt or ack/nak

rdt3.0: channels with errors *and* loss

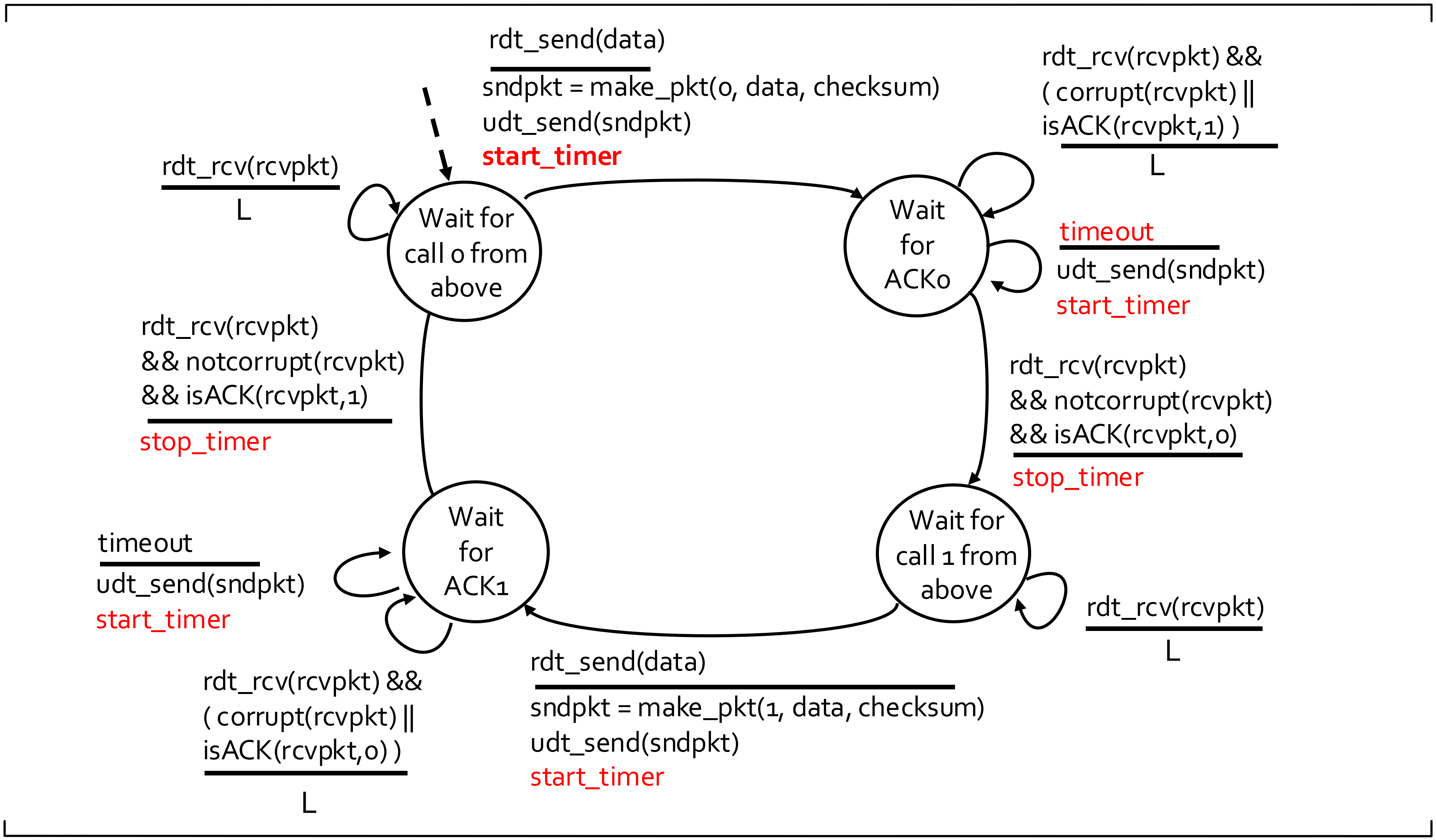
New assumption:

- underlying channel can also lose packets (data, ACKs)
  - checksum, seq. #, ACKs, retransmissions will be of help … but not enough

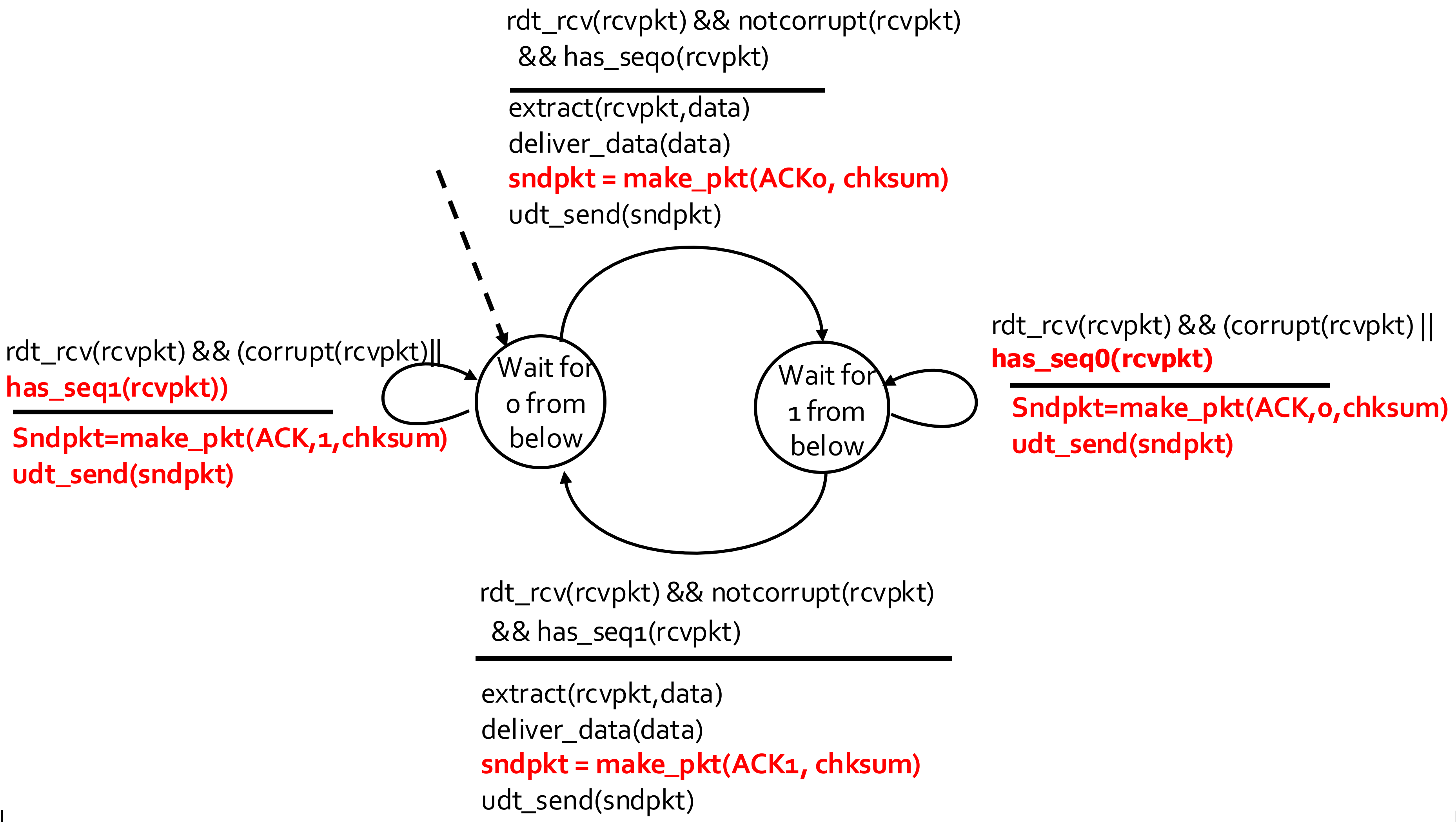
Approach:

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

rdt3.0 sender


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
L
rdt_rcv(rcvpkt)
L
Wait for call 0 from above
Wait for ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
Wait for ACK1
Wait for call 1 from above
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
L
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
L
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 receiver (rdt2.2 receiver)



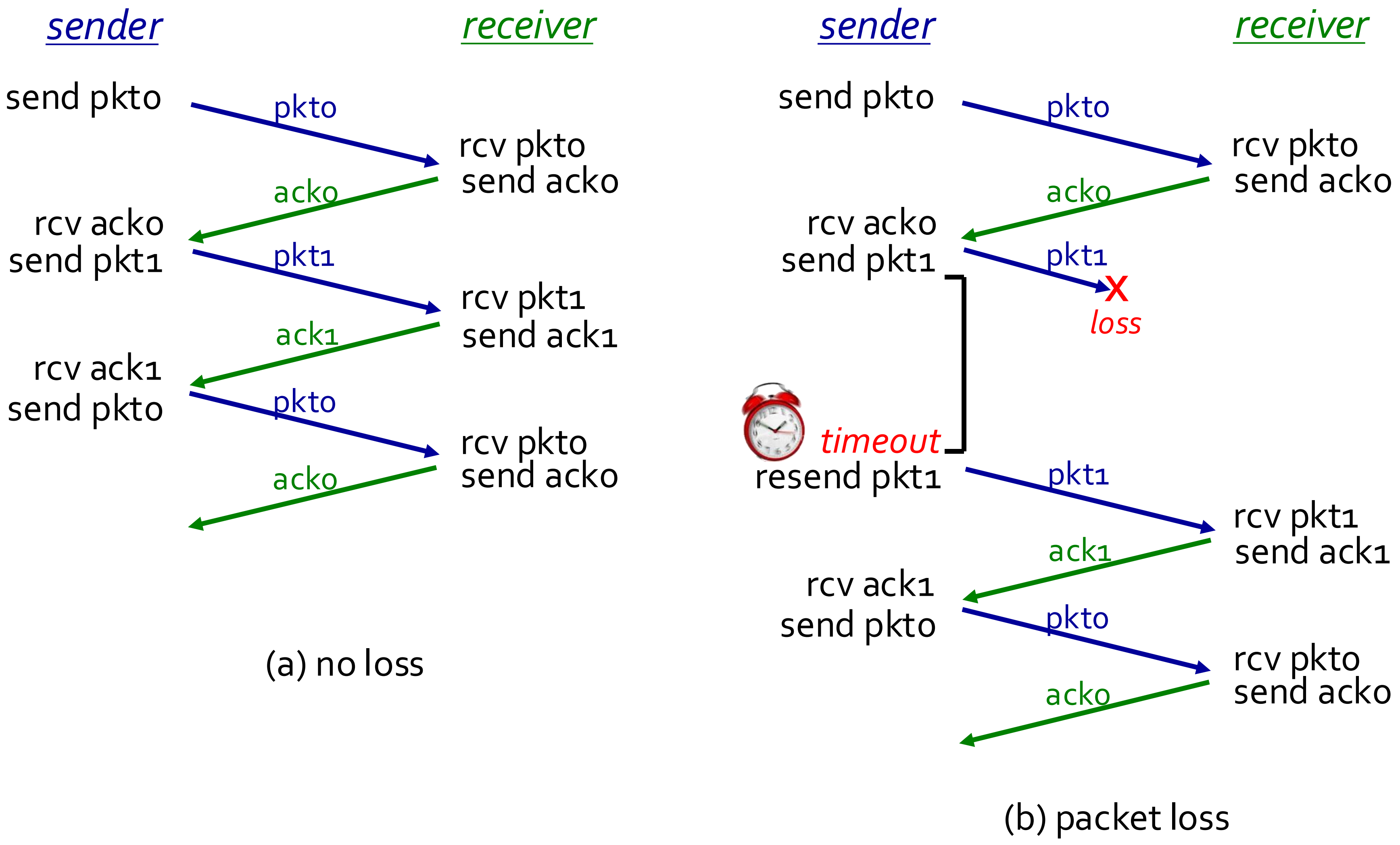
```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  && has_seq0(rcvpkt)
_____________________________________
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK0, chksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)||
has_seq1(rcvpkt))
_____________________________________
Sndpkt=make_pkt(ACK,1,chksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) ||
has_seq0(rcvpkt)
_____________________________________
Sndpkt=make_pkt(ACK,0,chksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  && has_seq1(rcvpkt)
_____________________________________
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)
```

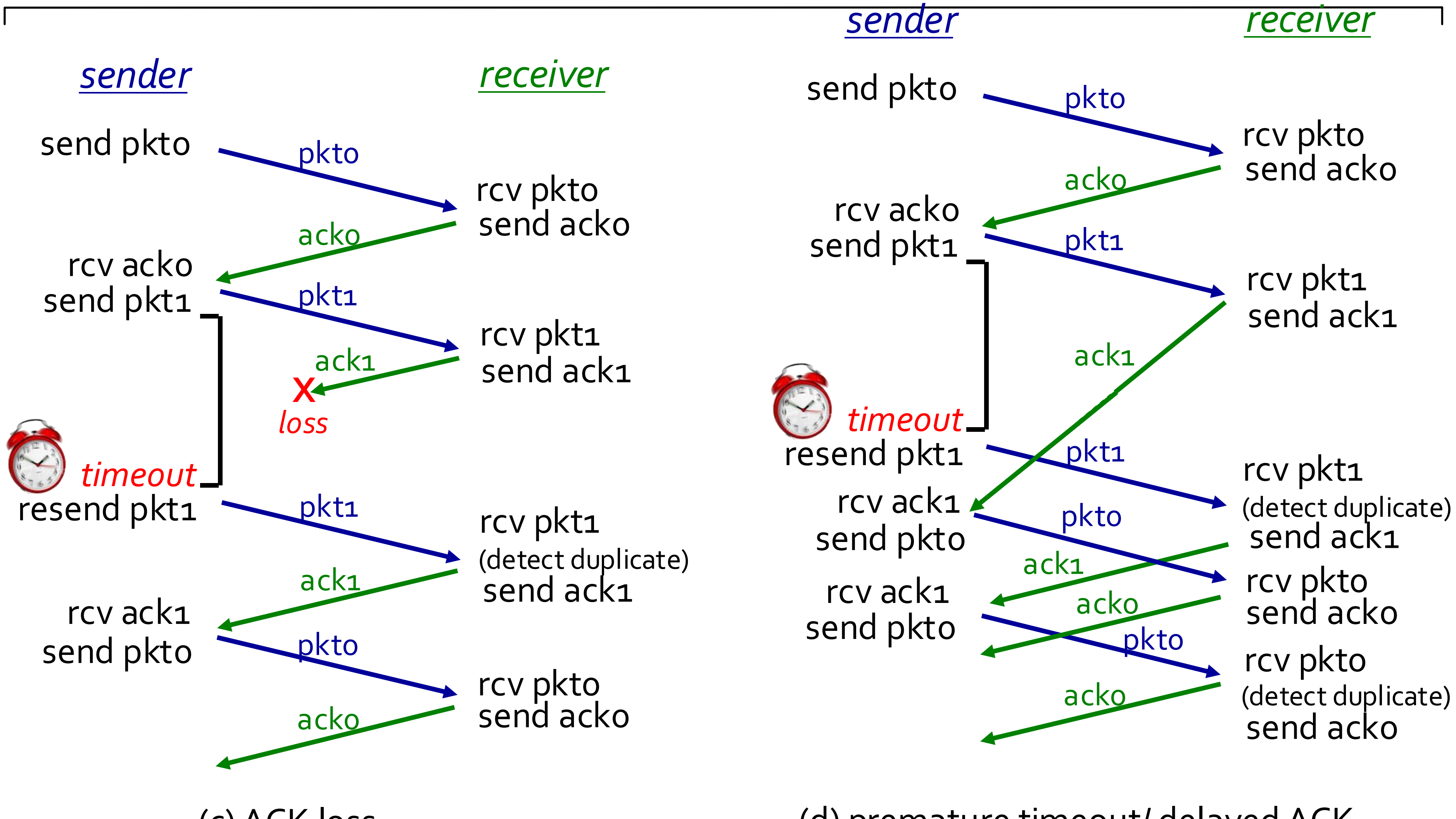
rdt3.0 in action



(a) no loss

(b) packet loss

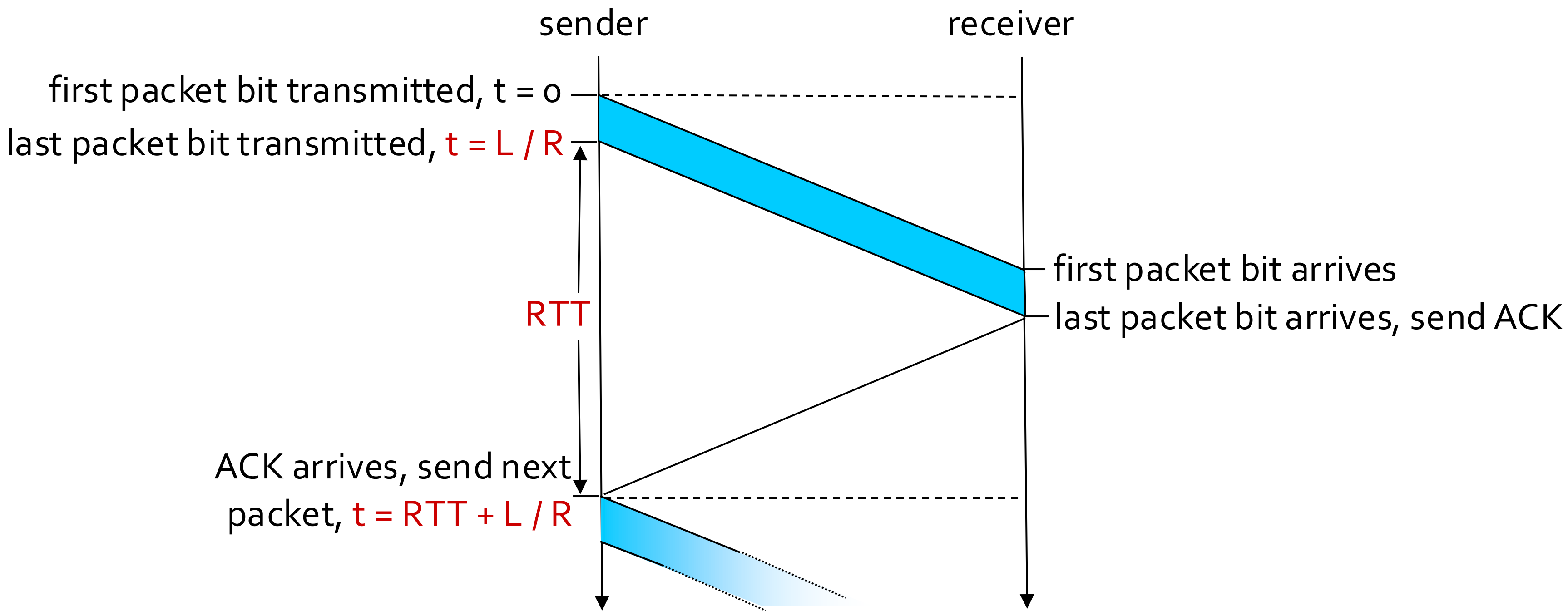
rdt3.0 in action



(c) ACK loss

(d) premature timeout/ delayed ACK

rdt3.0: stop-and-wait operation


sender
receiver
first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R
RTT
ACK arrives, send next packet, t = RTT + L / R
first packet bit arrives
last packet bit arrives, send ACK

$$U_{sender} = \frac{L / R}{RTT + L / R}$$

Performance of rdt3.0

- rdt3.0 is correct, but performance is not good
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

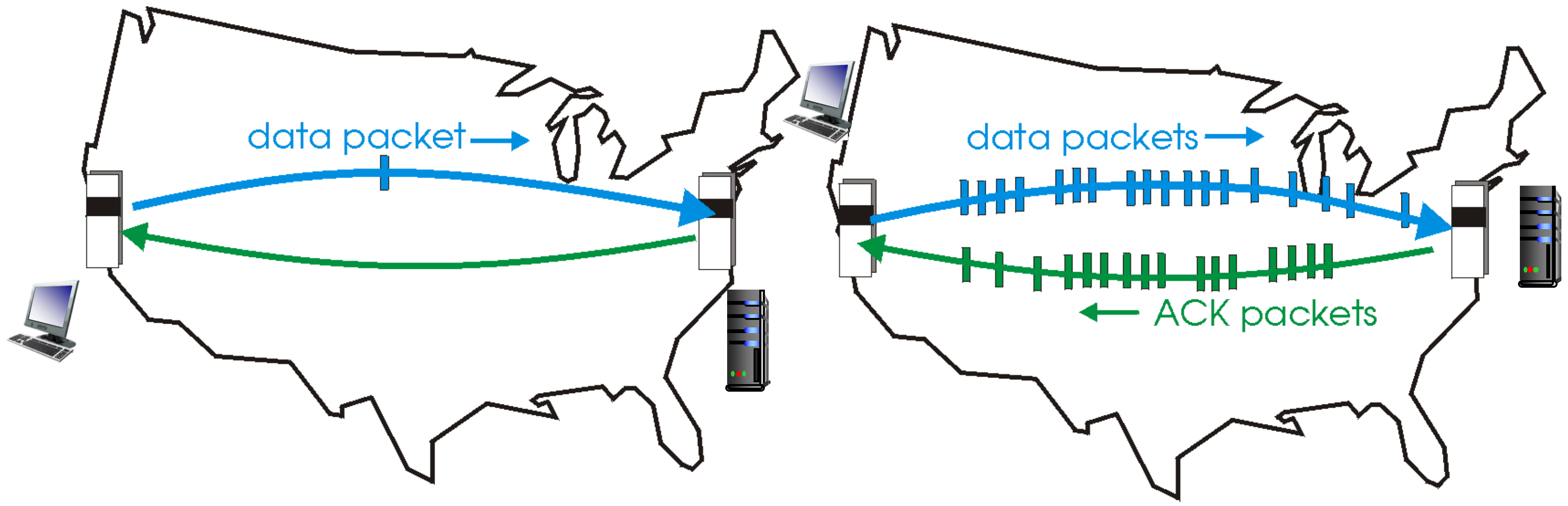
- U $_{sender}$: *utilization* – fraction of time sender busy sending

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

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec (267kbps) thruput over 1 Gbps link
- network protocol limits use of physical resources!

Pipelined protocols

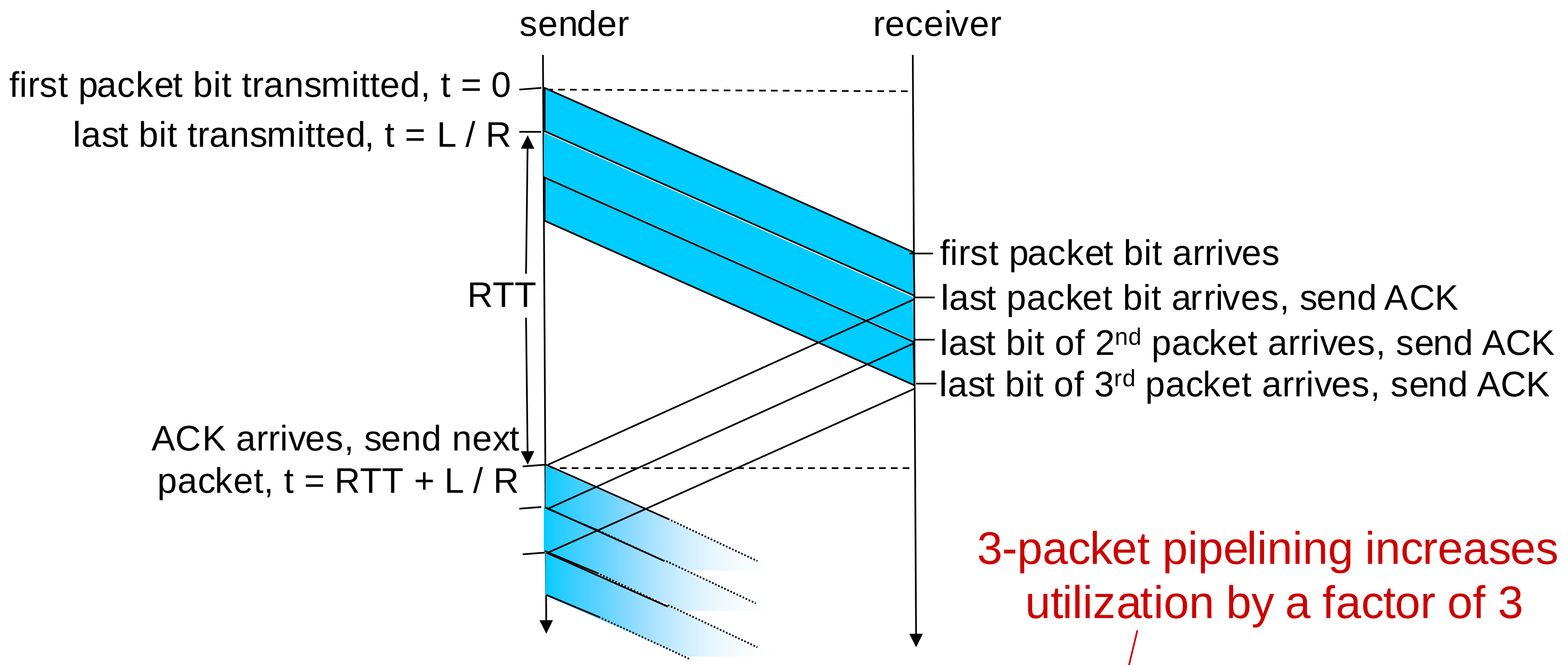
- pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver
- two generic forms of pipelined protocols: *go-Back-N, selective repeat*



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

(b) a pipelined protocol in operation

Pipelining: increased utilization


sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
ACK arrives, send next packet, t = RTT + L / R
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK

3-packet pipelining increases utilization by a factor of 3

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined protocols: overview

Go-back-N:

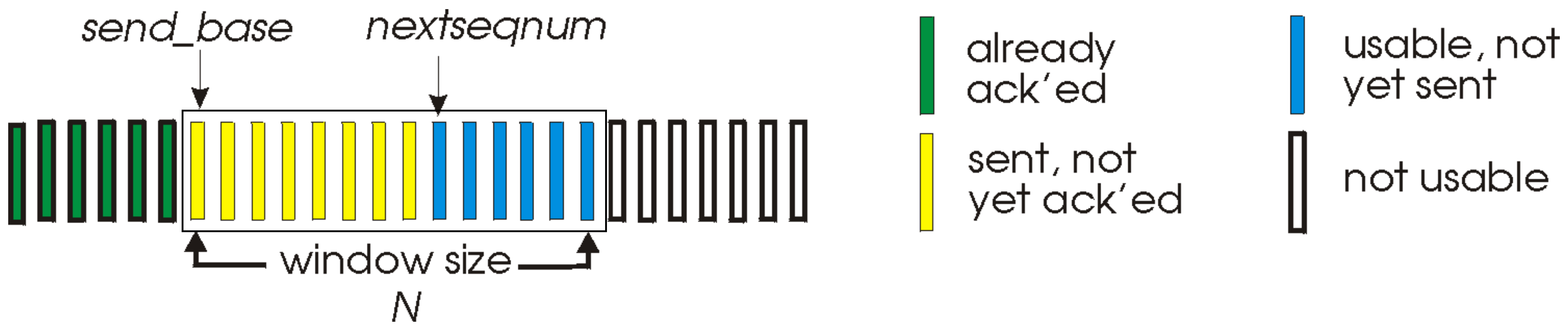
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

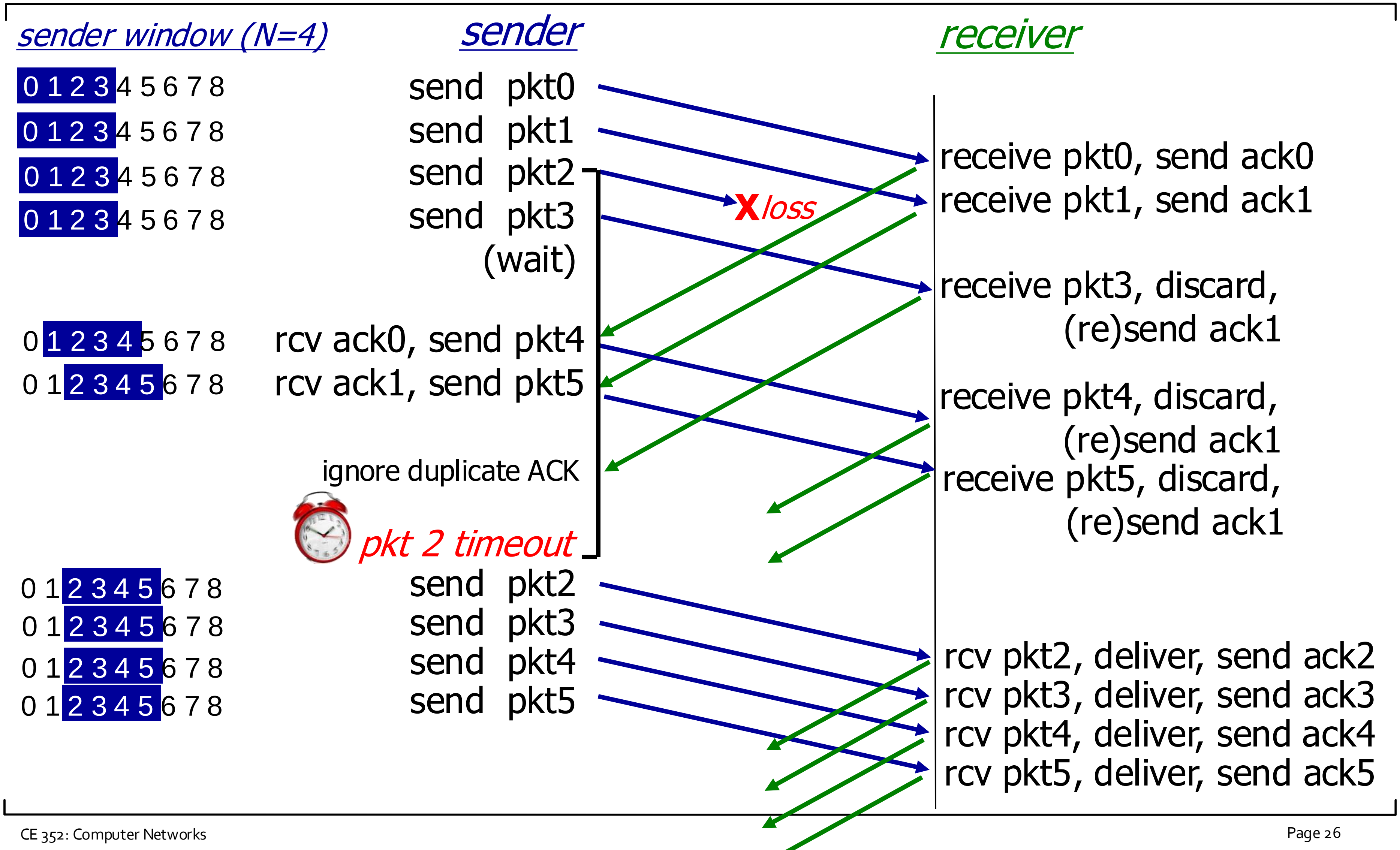
- sender can have up to N unack'ed packets in pipeline
- receiver sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

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 for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window



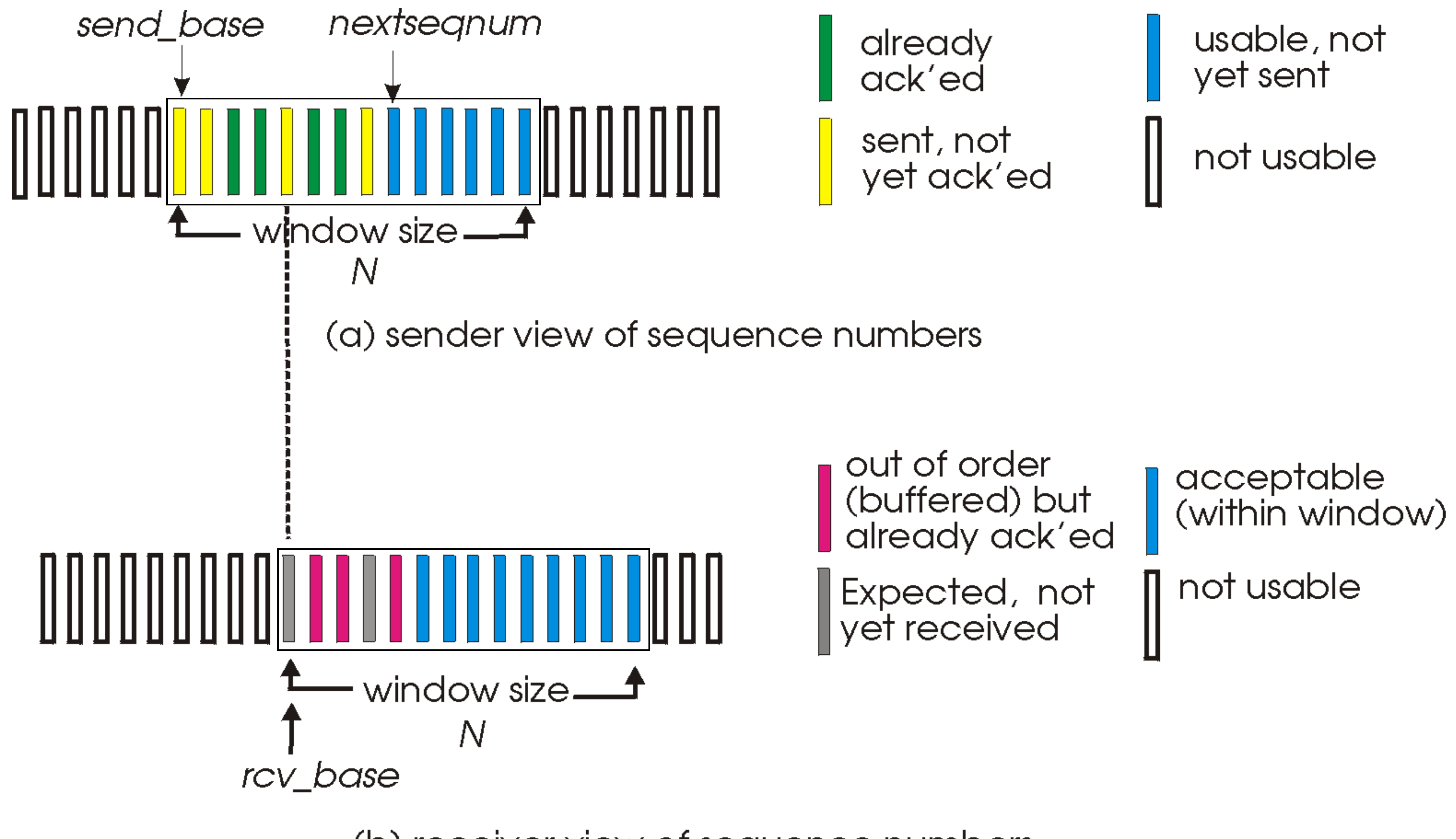
GBN in action



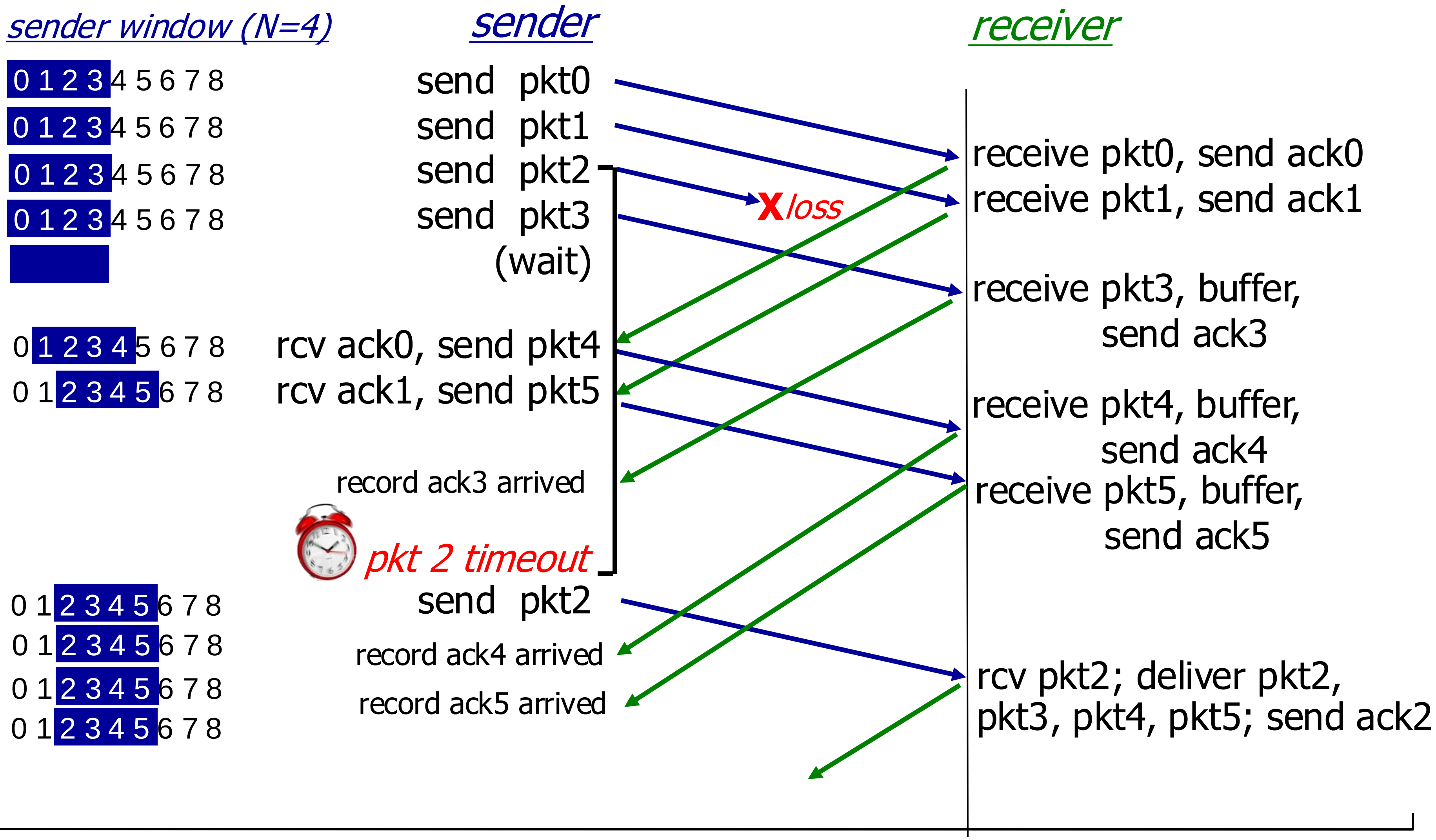
Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - 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
  - limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows


send_base
nextseqnum
already ack'ed
usable, not yet sent
sent, not yet ack'ed
not usable
window size N
(a) sender view of sequence numbers
out of order (buffered) but already ack'ed
acceptable (within window)
Expected, not yet received
not usable
window size N
rcv_base
(b) receiver view of sequence numbers

Selective repeat in action



Selective repeat

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)

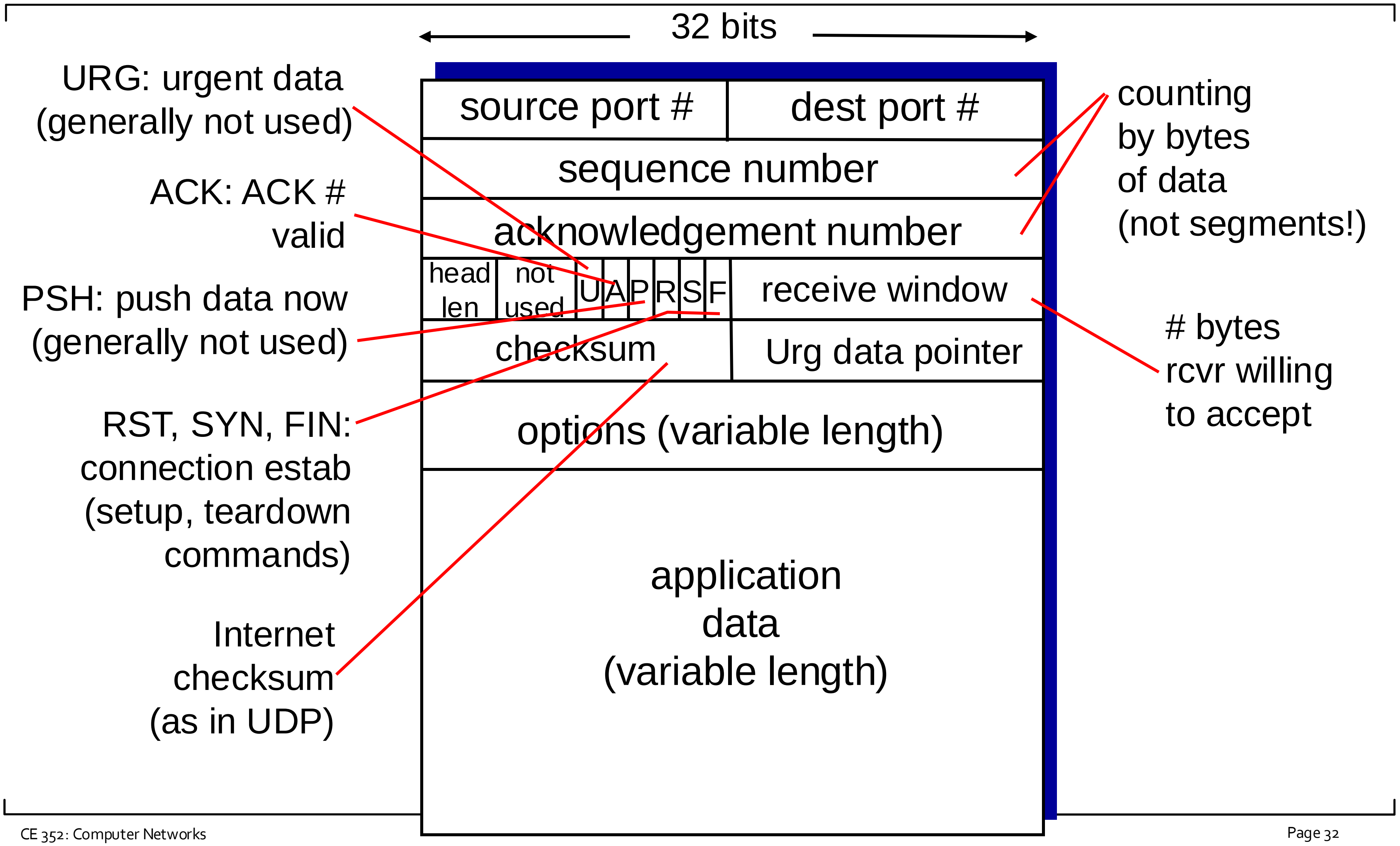
otherwise:

- ignore

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
- full duplex data:
  - bi-directional data flow in same connection
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

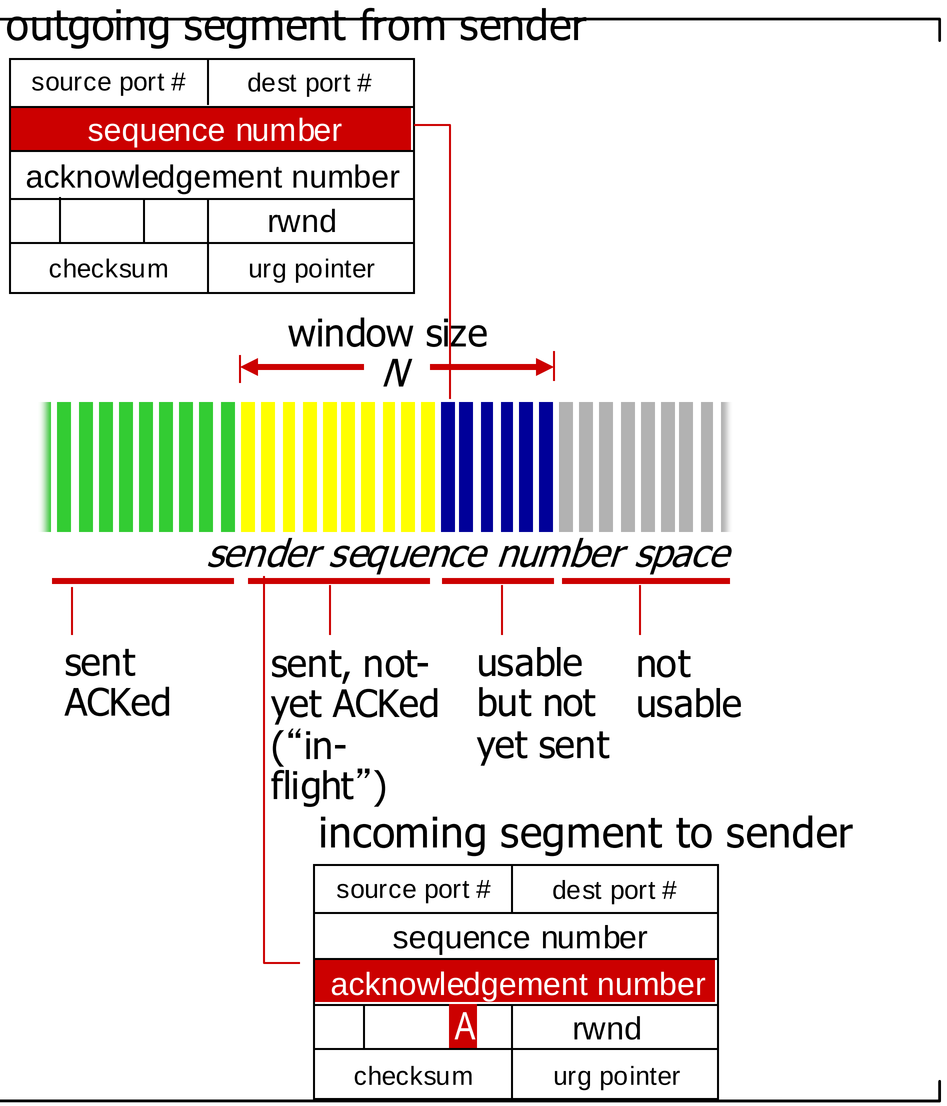
TCP segment structure



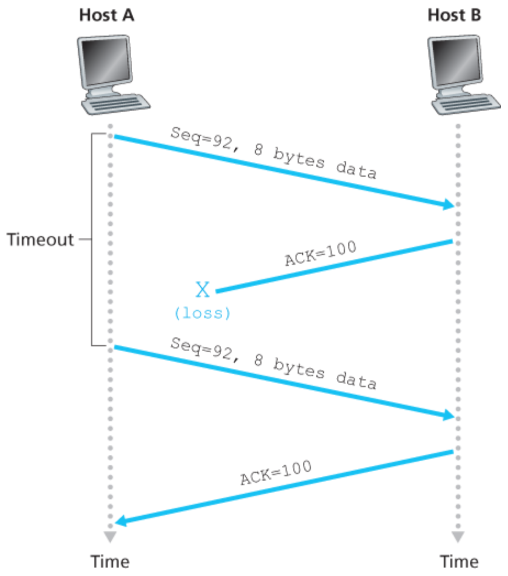
32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
URG: urgent data (generally not used)
ACK: ACK # valid
PSH: push data now (generally not used)
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
counting by bytes of data (not segments!)
# bytes rcvr willing to accept

TCP seq. numbers, ACKs

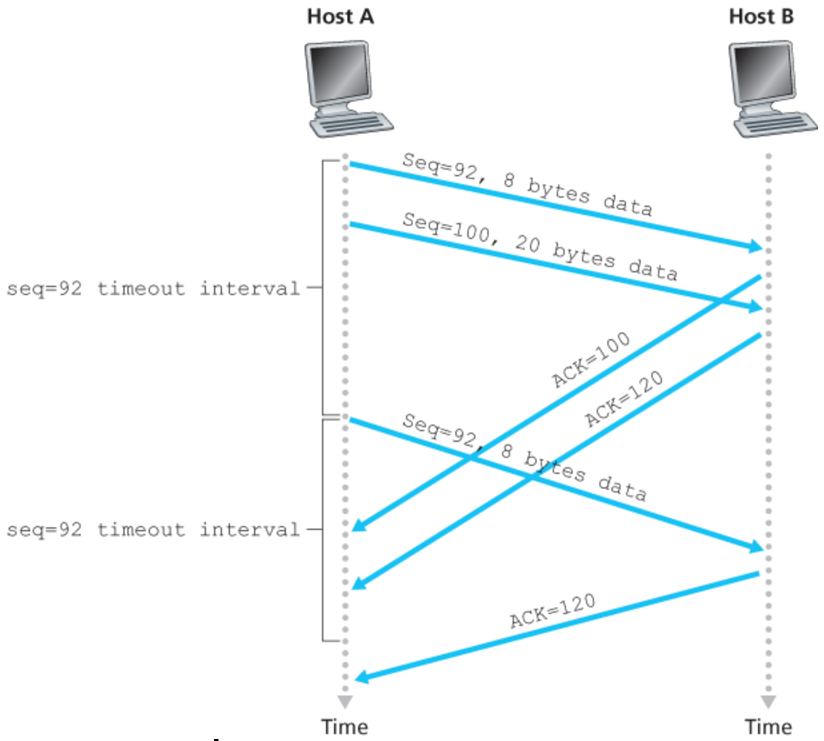
- Sequence numbers:
  - byte stream "number" of first byte in segment's data
- Acknowledgements:
  - seq # of next byte expected from other side
  - cumulative ACK

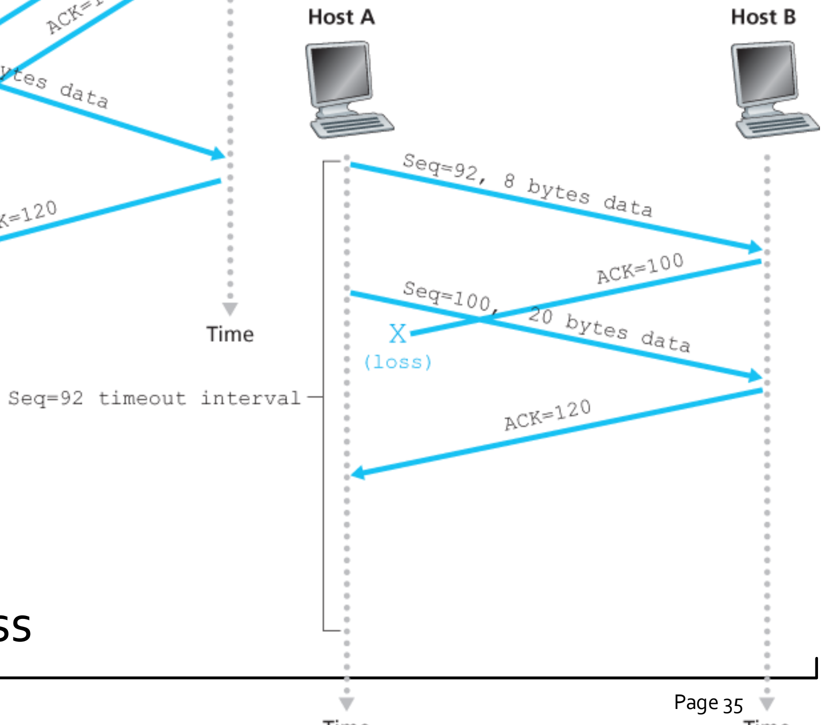


TCP Timeout



Retransmission due to a lost acknowledgment

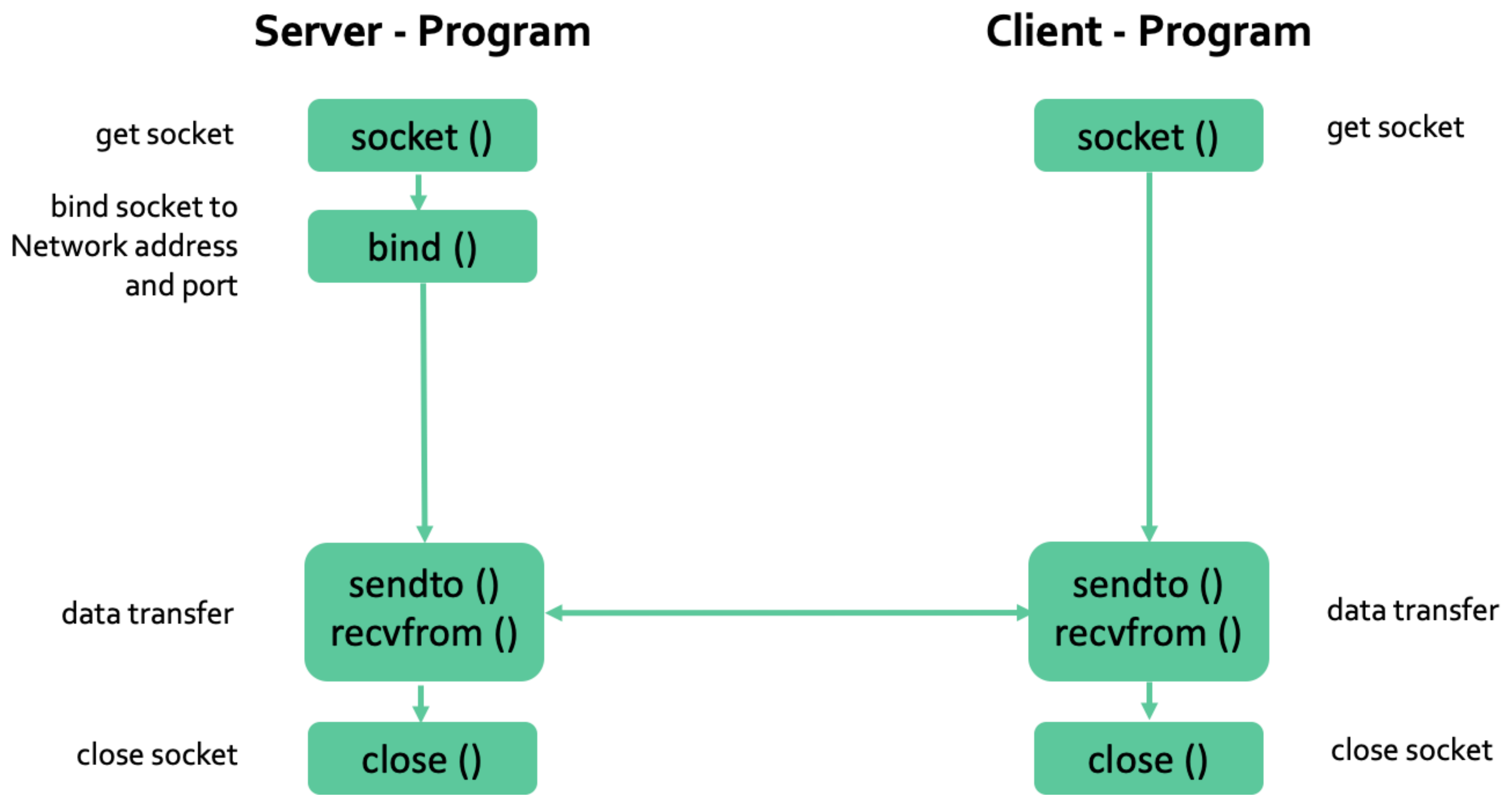




- How to set TCP timeout value?
- longer than RTT, but RTT varies
  - *too short:* premature timeout, unnecessary retransmissions
  - *too long:* slow reaction to segment loss

Bonus 3

- UDP/ IP socket programming
- Client - Server file transfer (connectionless and unreliable)



Summary

Today:

- Transport protocols
  - rtd3.0: channel with errors and loss
  - Pipelined protocols: Go-Back-N and Selective repeat
- TCP overview, segment structure, communication

Canvas discussion:

- Reflection
- Exit ticket

Next time:

- read 3.6 and 3.7 of K&R
- follow on Canvas! material and announcements

Any questions?