Transport Layer

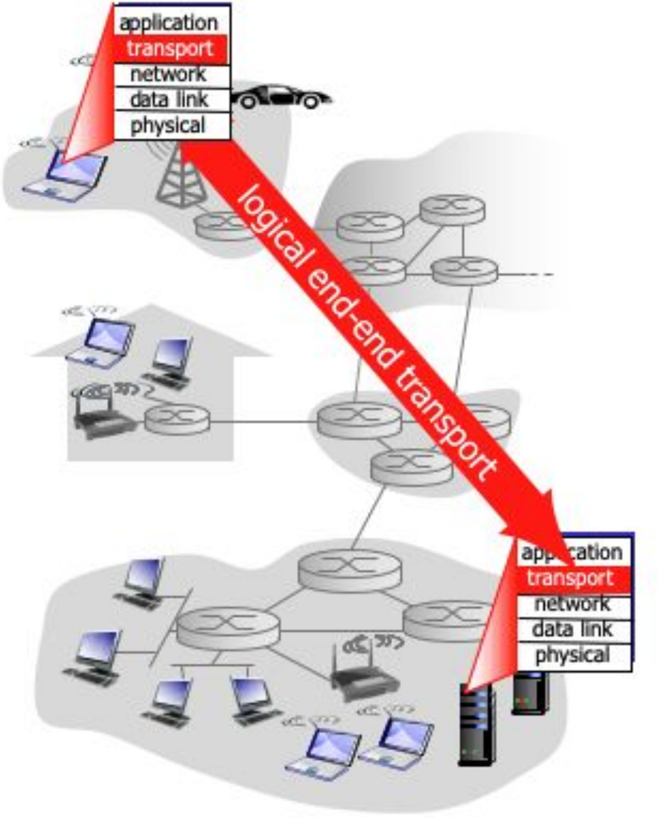
CS5700 Fall 2019

Agenda

- Transport layer services
- UDP
- Reliable data transfer
- TCP
- Congestion control

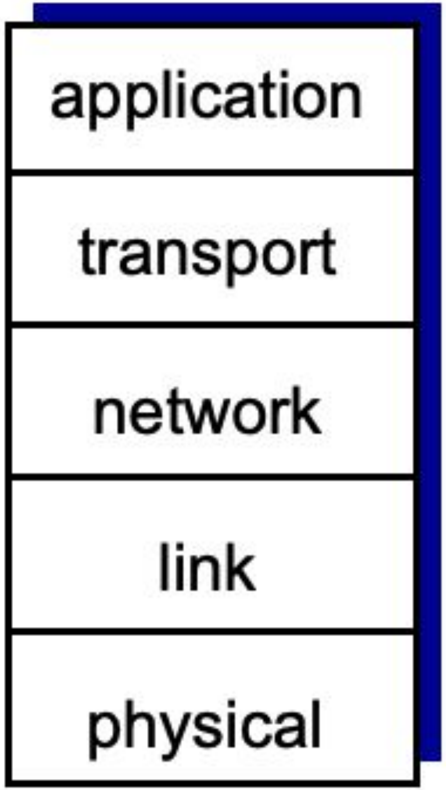
Transport services and protocols

- Provide logical communication between application processes
- Run in end systems (not the core)
- More than one transport protocol available to applications
  - TCP and UDP



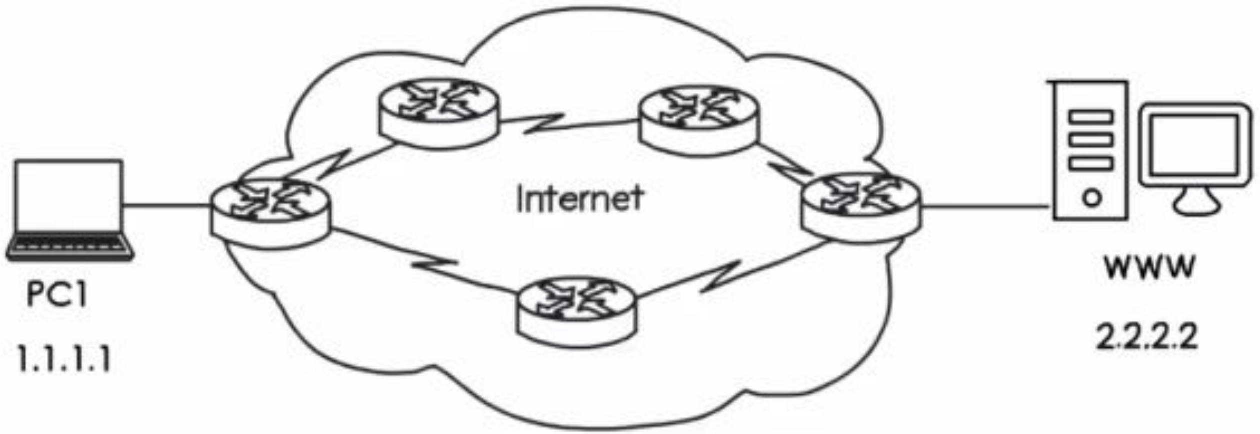
What does network layer do?

- What’s the difference between transport layer and network layer?
- What services are provided by network layer?



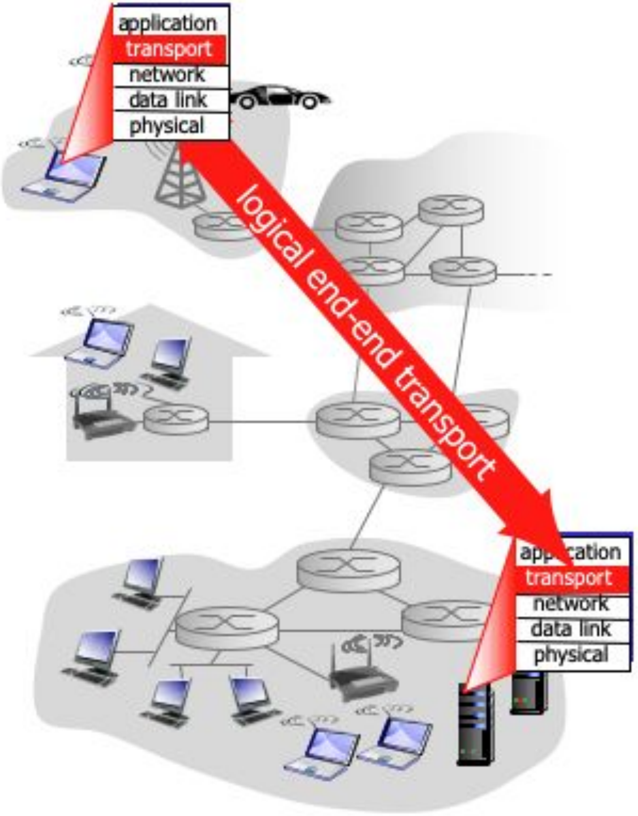
Network layer service model

- Logical communication between hosts
- Every packet is treated individually and separately
- ***Best effort***. No guarantee of delivery.



Transport layer protocols

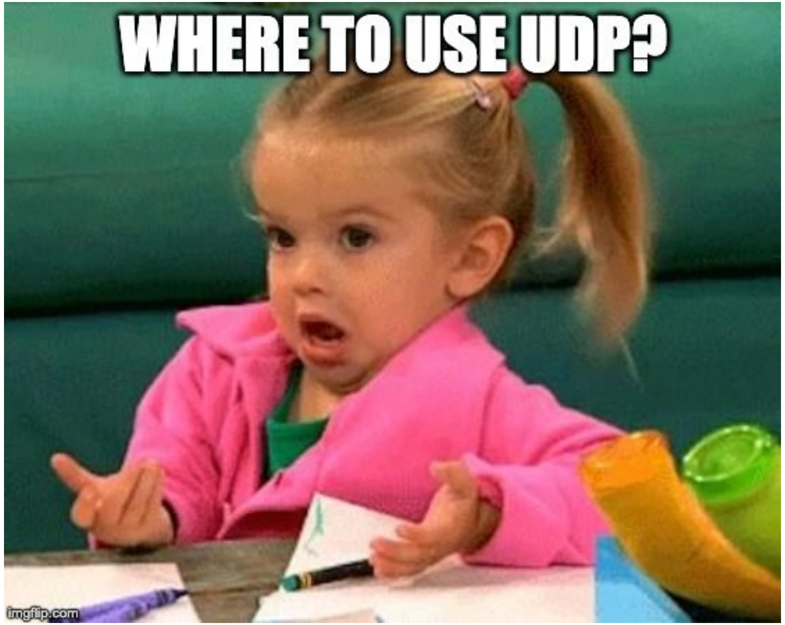
- TCP
  - Reliable in-order delivery
  - Connection oriented
  - Flow control
  - Congestion control
- UDP
- Services not available
  - Delay or bandwidth guarantee



UDP

UDP

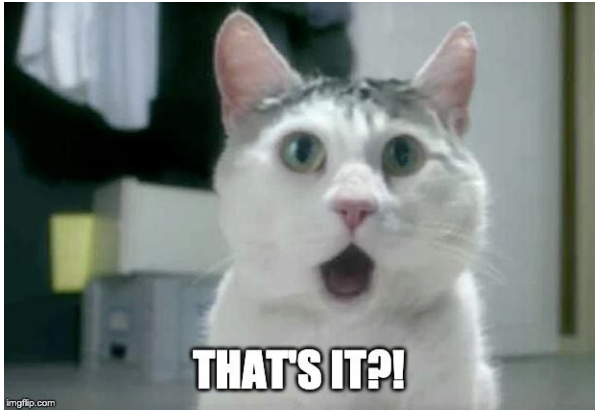
- User Datagram Protocol
  - Connection less
  - No guarantee of delivery
- Where do you see UDP used? Do you know why?



UDP header

- Do you know what's each field for?

16 bit source port	16 bit destination port
16 bit UDP length	16 bit UDP checksum
Data	



UDP checksum

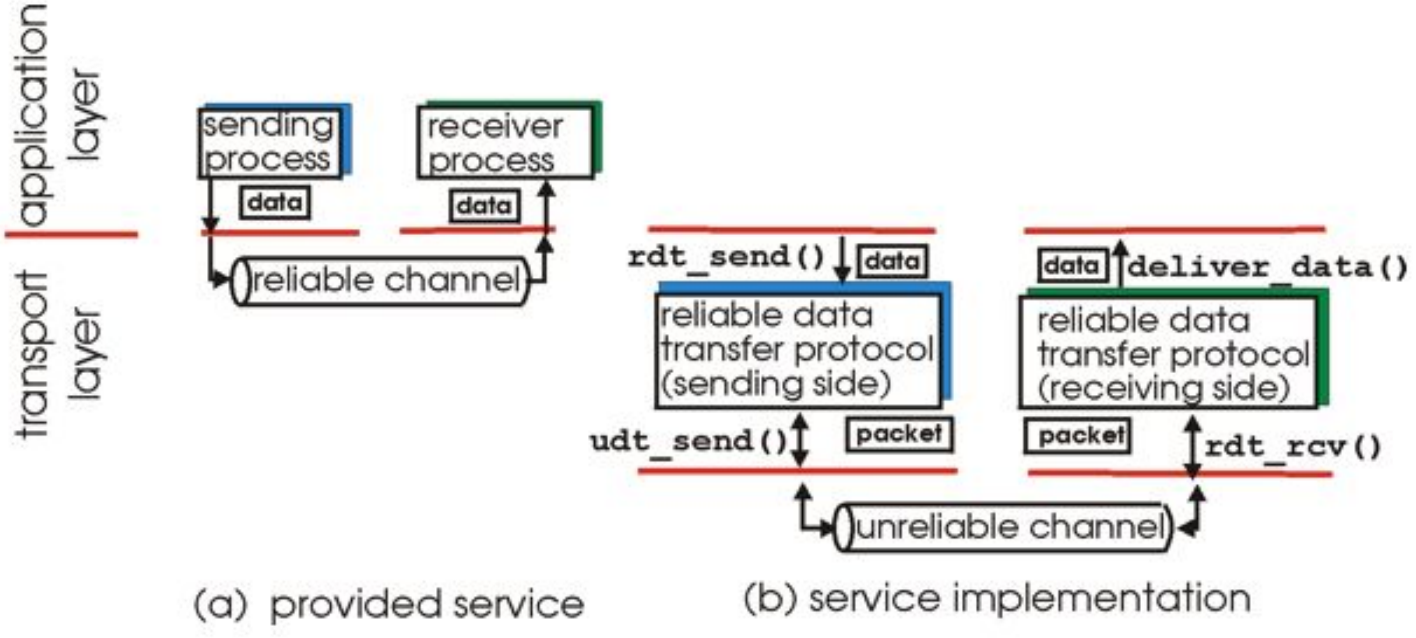
- Detect “errors” (e.g. flipped bits) in transmitted segment
- Sender
  - Treat data (include header) as seq of 16-bit integers
  - Add them up (1’s complement), call it checksum
  - Put checksum into UDP header
- Receiver
  - Same algorithm, compute checksum and compare

UDP socket

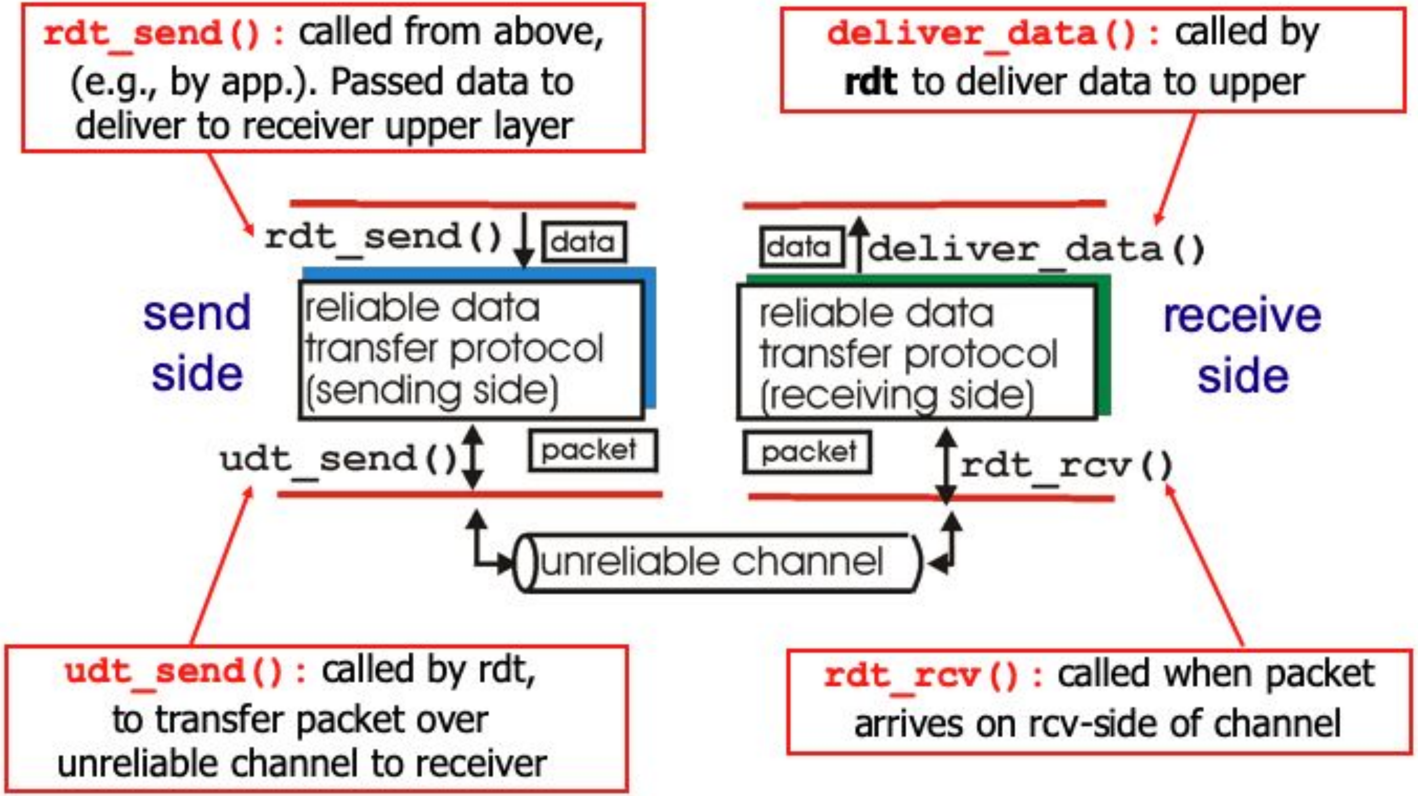
Reliable data transfer

Reliable data transfer

- Important in application, transport, and link layers

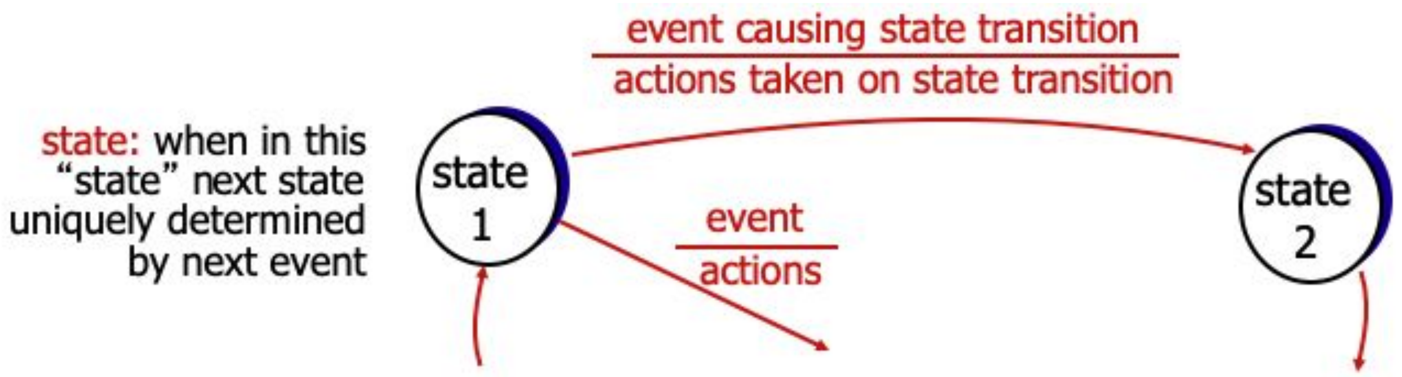


Reliable data transfer



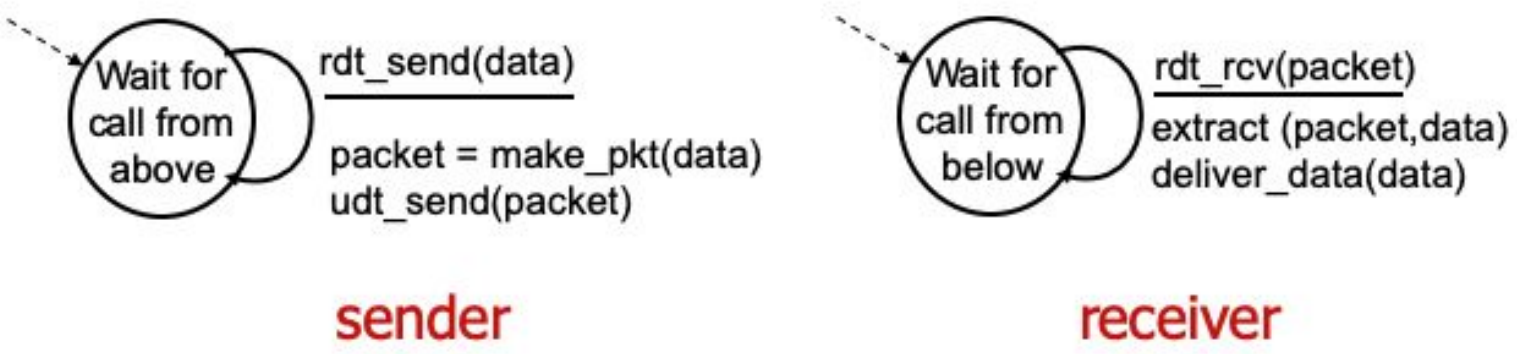
Reliable data transfer

- Incrementally design sender/receiver of rdt
- Consider only unidirectional data transfer
  - But control info will flow on both directions
- Use FSM (finite state machines) to design algorithm



rdt1.0: over a reliable channel

- Underlying channel is perfectly reliable
  - No bit errors
  - No loss of packets



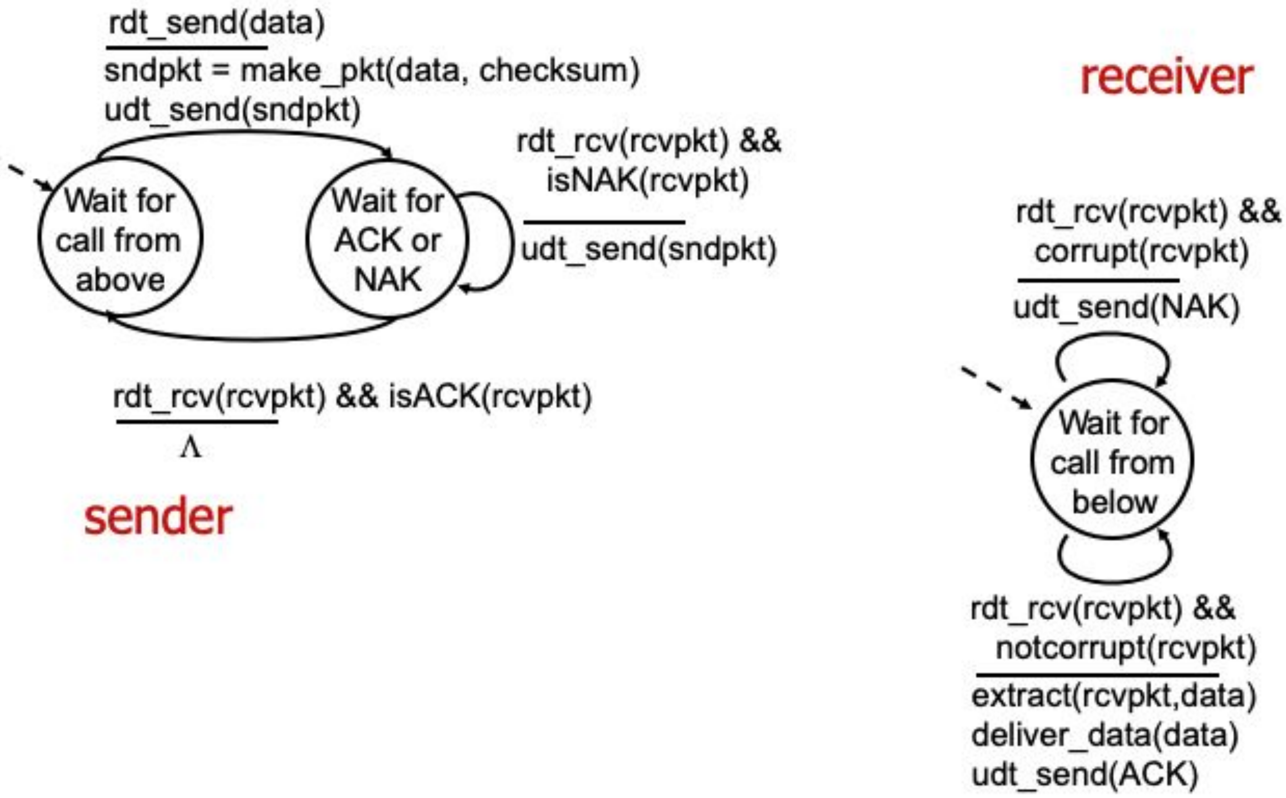
rdt2.0: channel with bit errors

- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- How to recover from errors?

rdt2.0: channel with bit errors

- ACK (acknowledgement)
  - Receiver explicitly tells sender that pkt received OK
- NAK (negative acknowledgement)
  - Receiver explicitly tells sender that pkt had errors
- Sender needs to retransmit pkt on receipt of NAK
- Summary
  - Error detection
  - Feedback with control message ACK and NAK

rdt2.0: FSM


rdt_send(data)
sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
sender
receiver
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)
udt_send(NAK)
Wait for call from below
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: anything looks wrong?

rdt2.0

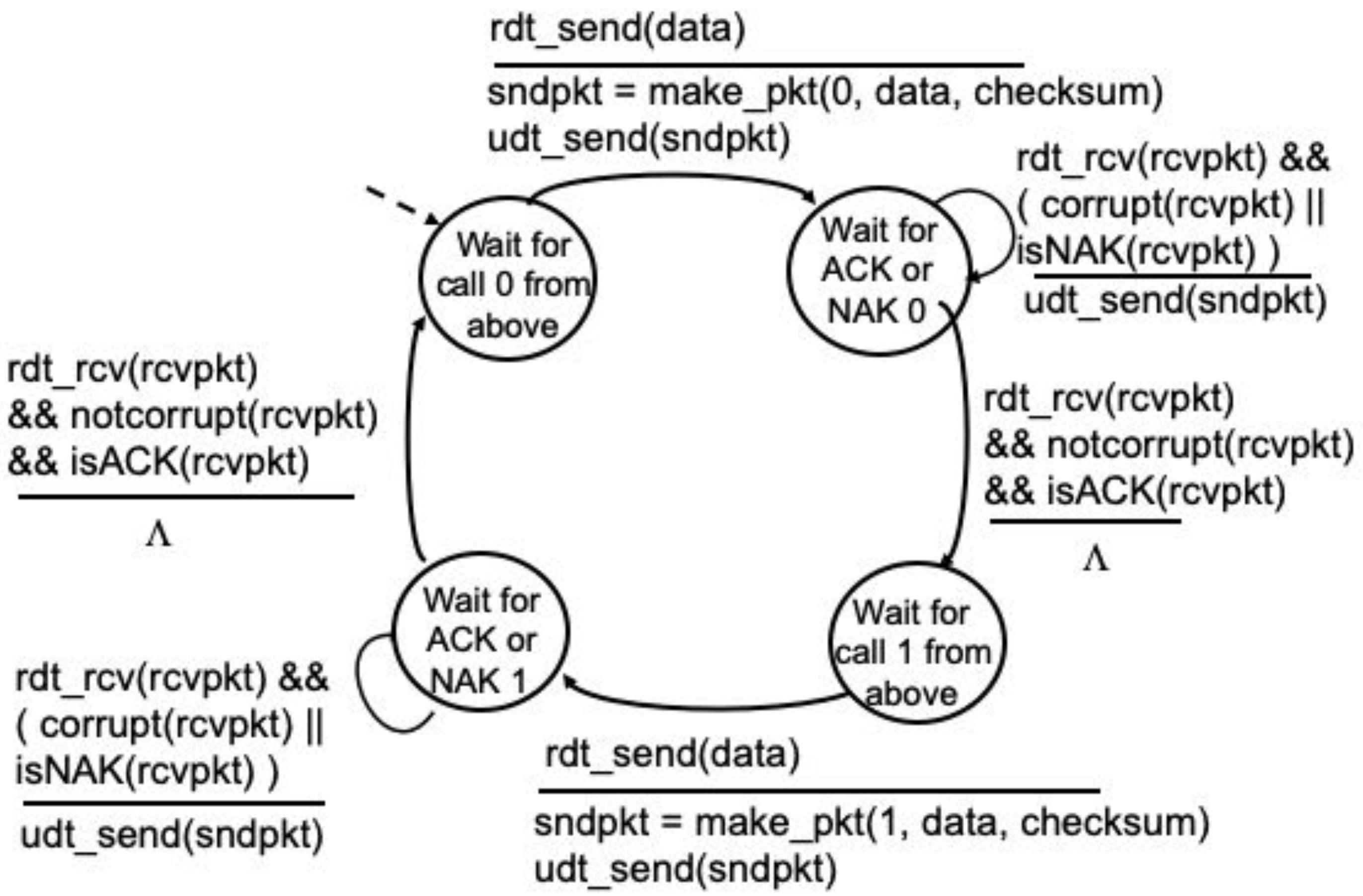
- What happens if ACK or NAK is corrupted?
    - Sender doesn’t know what happened at receiver
- Can sender just retransmit?

rdt2.0

- Receiver needs to handle duplicates when sender retransmit
- Need to use sequence number!
- Stop and wait algorithms
    - Sequence number either 0 or 1

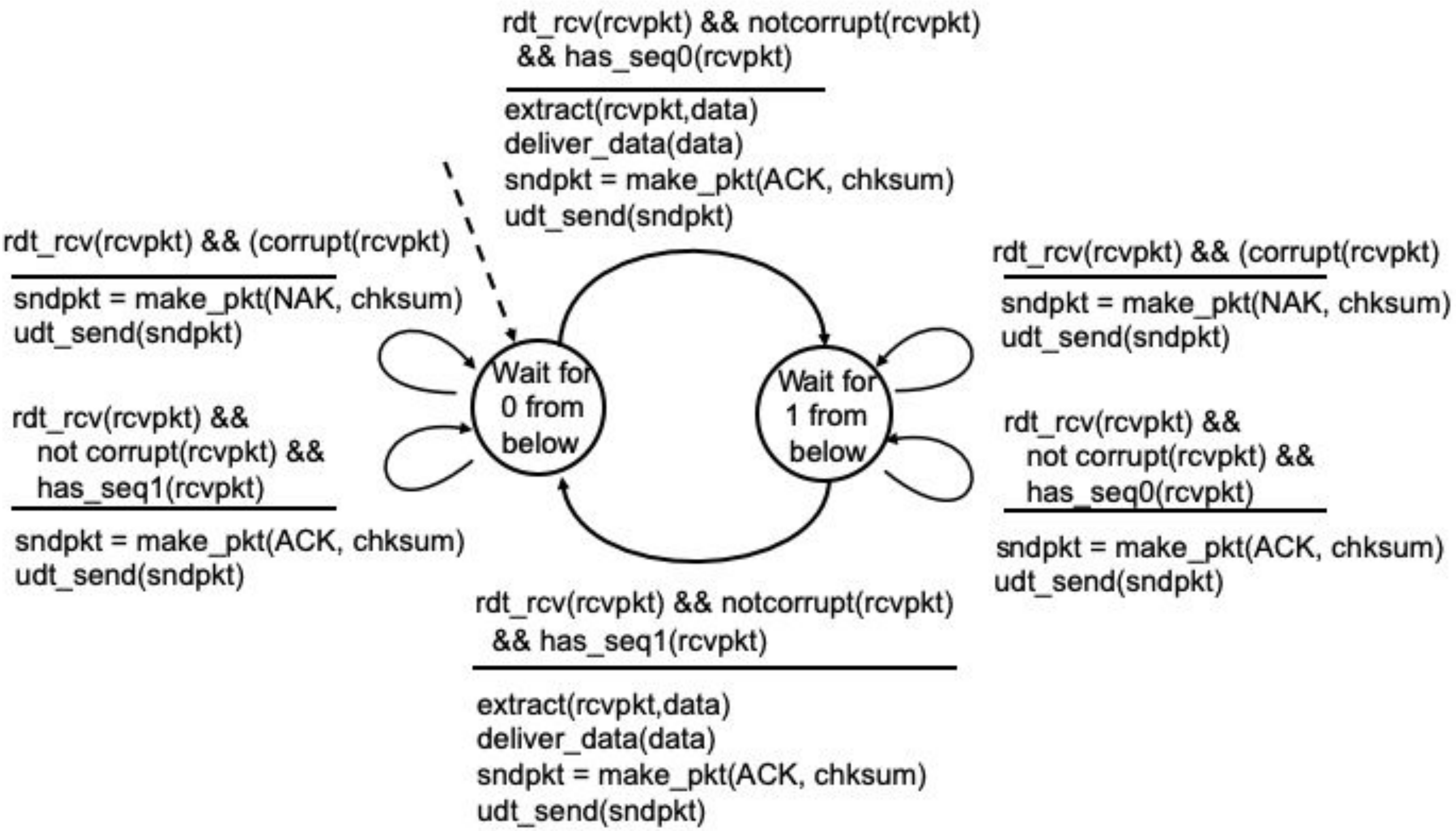
rdt2.1

Sender



rdt2.1

Receiver



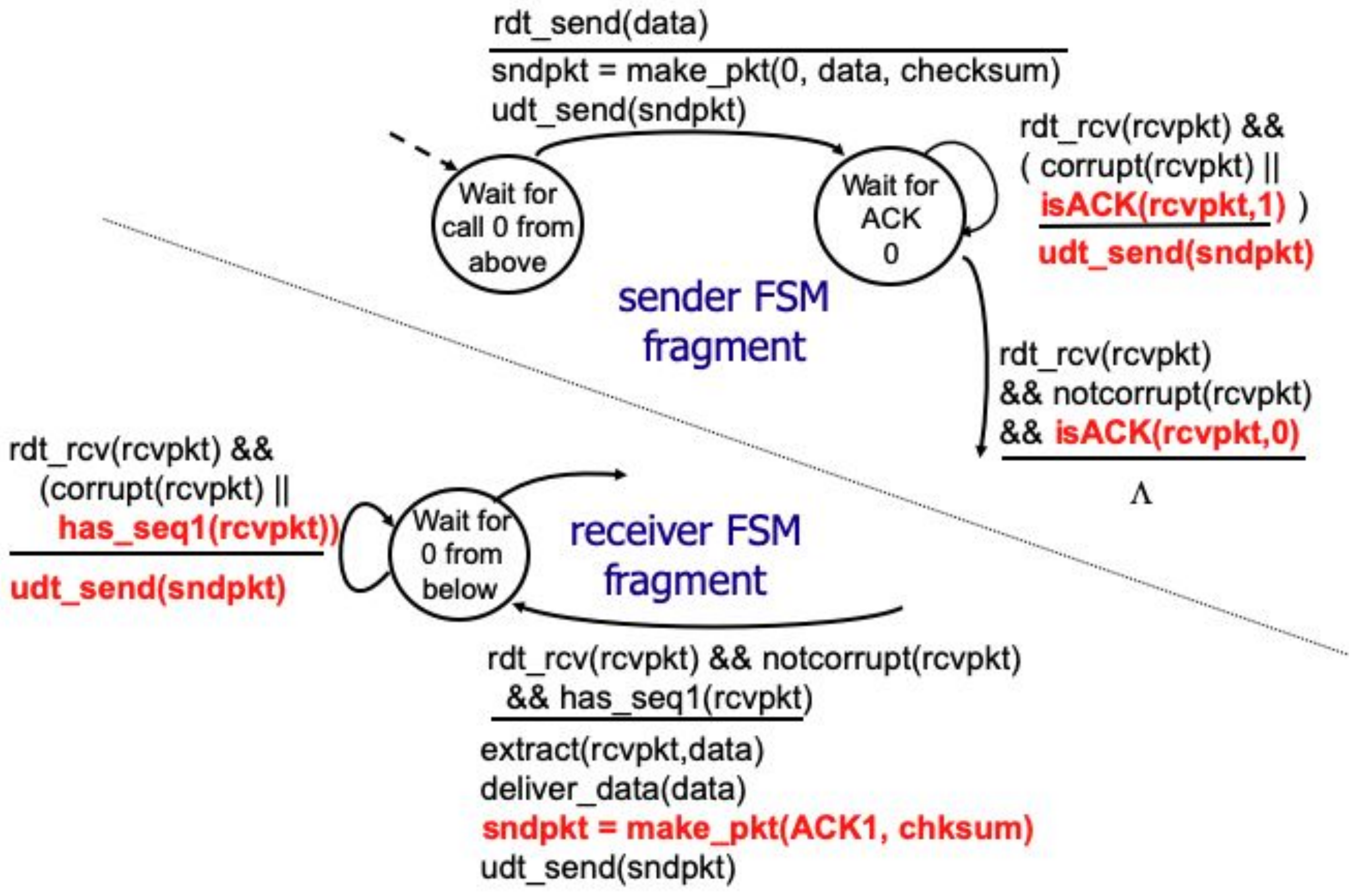
rdt2.1: summery

- Sender
  - Add sequence number to packets (either 0 or 1)
  - Retransmit if receives NAK
  - Retransmit if ACK/NAK is corrupted
- Receiver
  - Check if received packet is duplicate (use seq #)
  - Send ACK or NAK for each packet

rdt2.2: NAK-free protocol

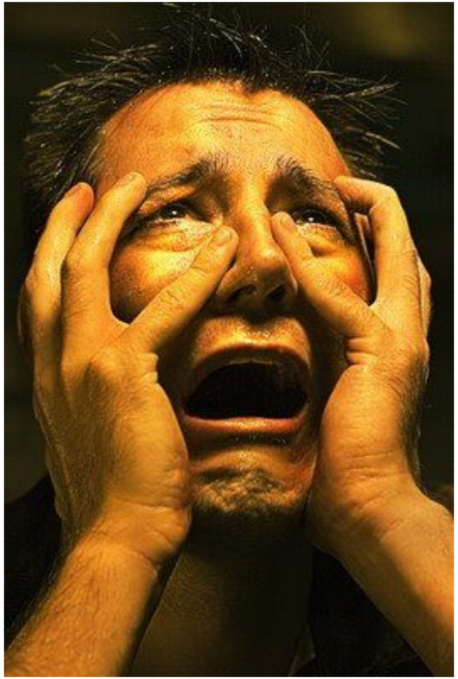
- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt
  - Receiver must explicitly include sequence number now in ACK message
- Duplicate ACK at sender results in same action as NAK
  - Retransmit current packet

rdt2.2: FSM


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
udt_send(sndpkt)
sender FSM fragment
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
Λ
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))
udt_send(sndpkt)
Wait for 0 from below
receiver FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

rdt3.0: channel with errors and loss

- Underlying channel can also lose packets (data or ACK)
- What now?
  - Sequence number
  - Checksum
  - ACKs
  - But not enough...

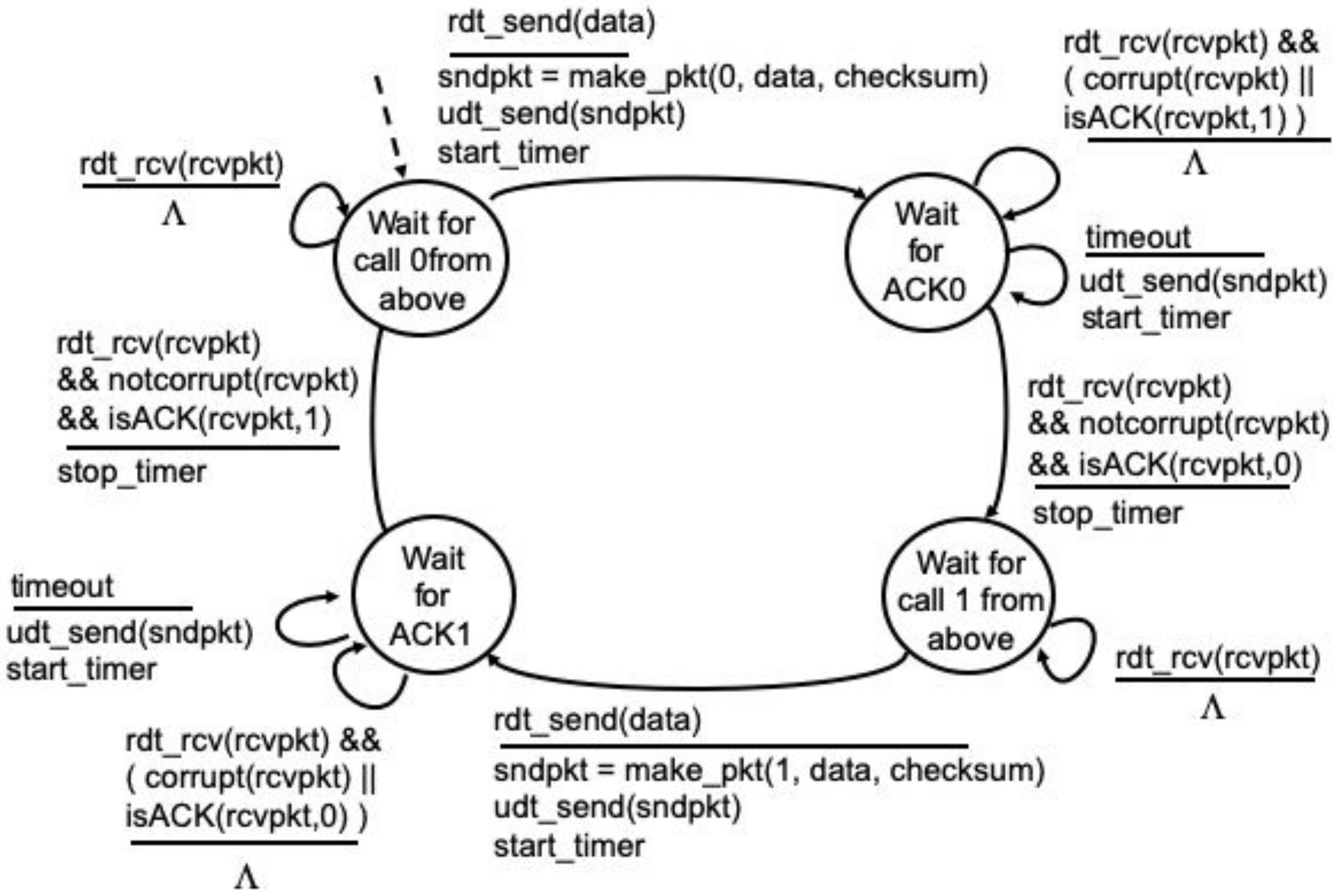


rdt3.0

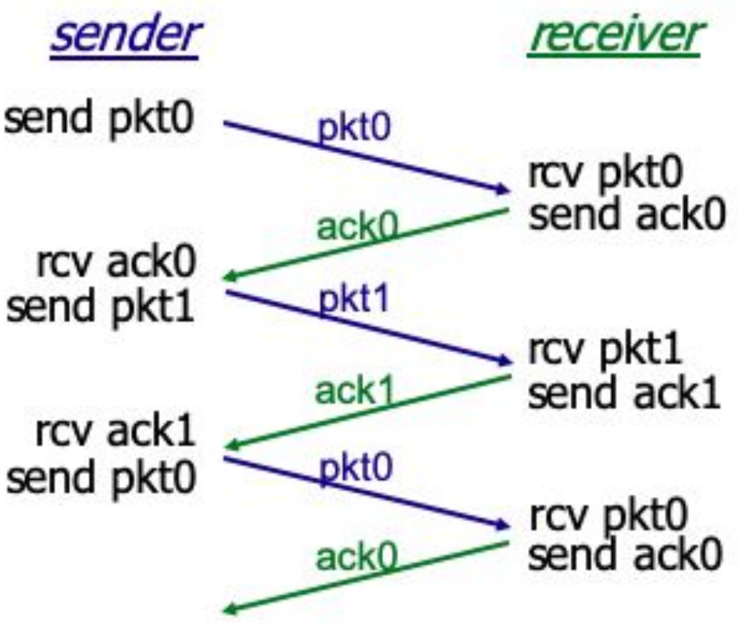
- Need timeout!
    - “Reasonable” amount of time for ACK
    - Retransmit if no ACK received in this time
        - Maybe delayed, maybe lost
    - Receiver must specify the sequence number of packet being ACKed

rdt3.0

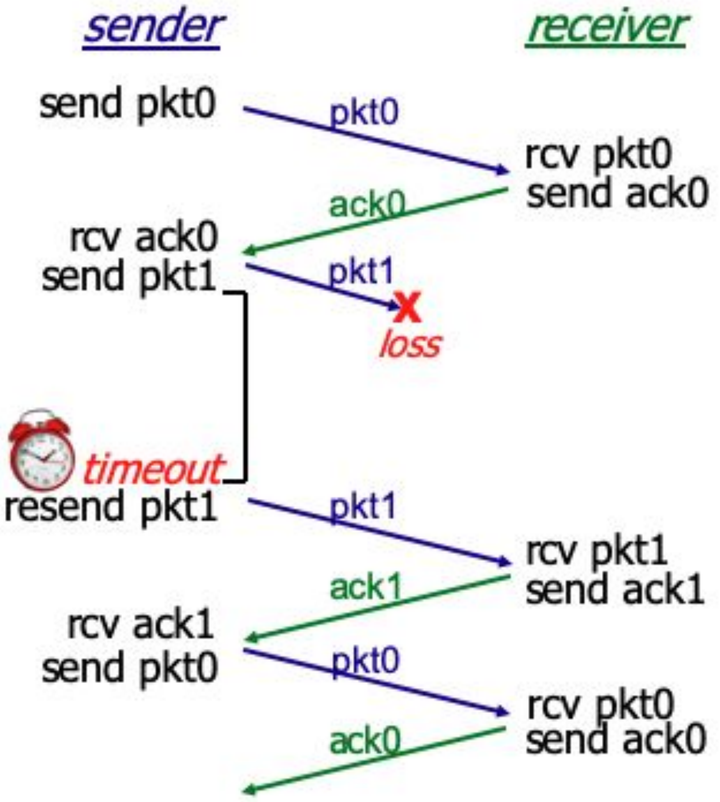
Sender



rdt3.0

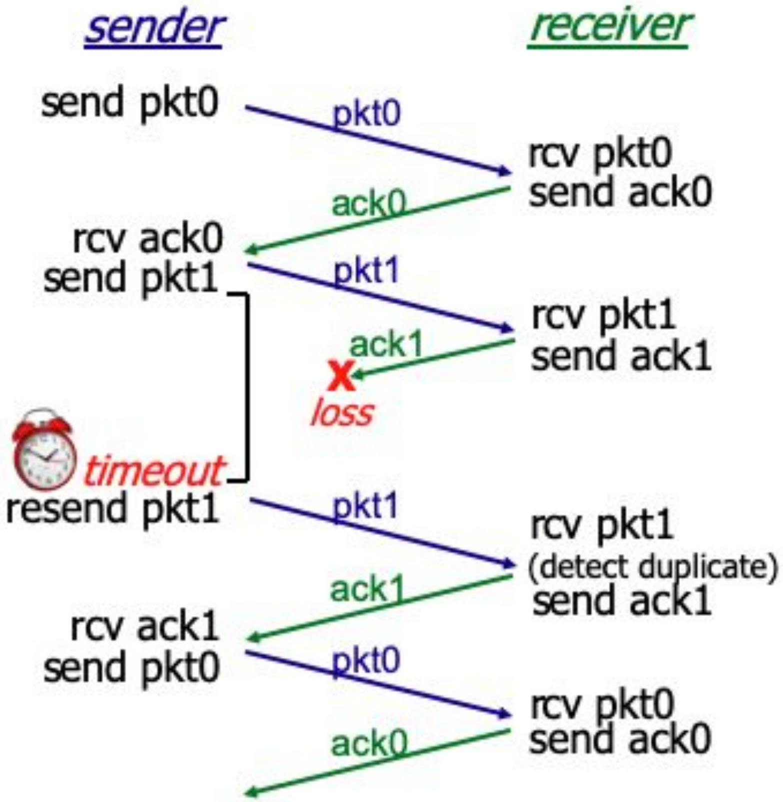

sender
receiver
send pkt0
pkt0
rcv pkt0
send ack0
ack0
rcv ack0
send pkt1
pkt1
rcv pkt1
send ack1
ack1
rcv ack1
send pkt0
pkt0
rcv pkt0
send ack0
ack0

rdt3.0

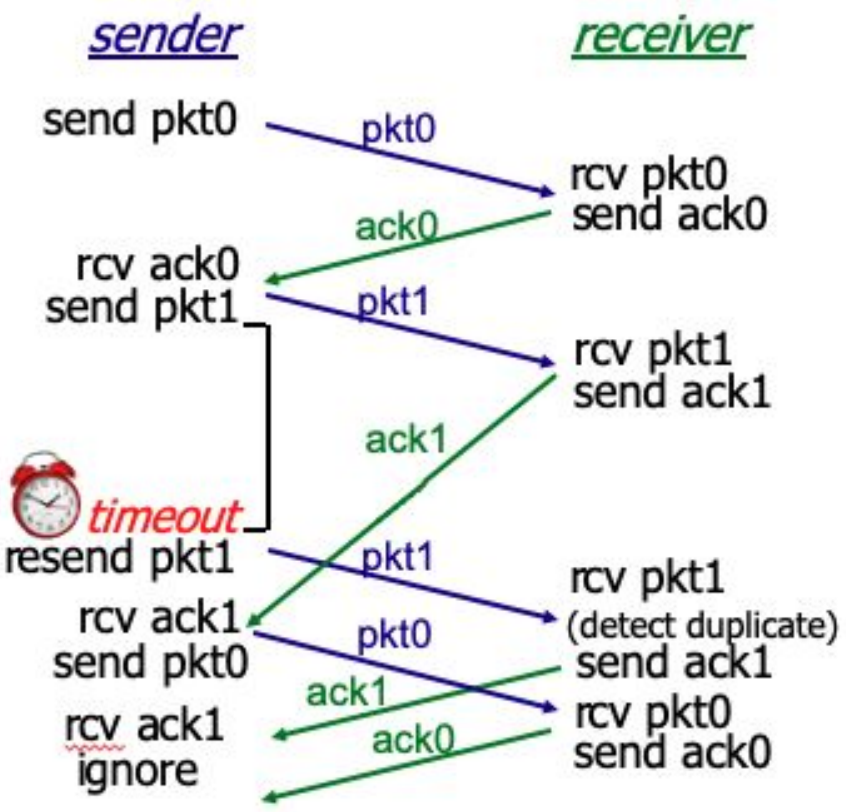


sender
receiver
send pkt0
pkt0
rcv pkt0
send ack0
ack0
rcv ack0
send pkt1
pkt1
X
loss
timeout
resend pkt1
pkt1
rcv pkt1
send ack1
ack1
rcv ack1
send pkt0
pkt0
rcv pkt0
send ack0
ack0

rdt3.0


sender
receiver
send pkt0
pkt0
rcv pkt0
send ack0
ack0
rcv ack0
send pkt1
pkt1
rcv pkt1
send ack1
ack1
X
loss
timeout
resend pkt1
pkt1
rcv pkt1
(detect duplicate)
send ack1
ack1
rcv ack1
send pkt0
pkt0
rcv pkt0
send ack0
ack0

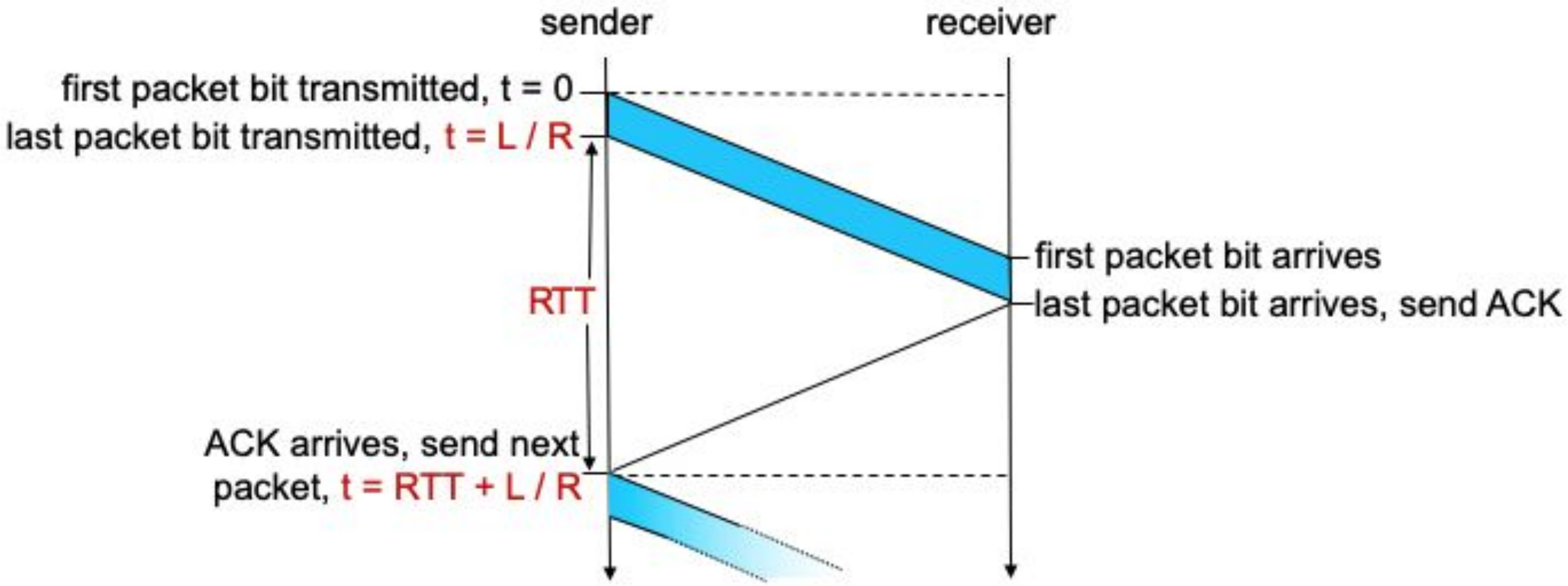
rdt3.0


sender
receiver
send pkt0
pkt0
rcv pkt0
send ack0
ack0
rcv ack0
send pkt1
pkt1
rcv pkt1
send ack1
ack1
timeout
resend pkt1
pkt1
rcv pkt1
(detect duplicate)
send ack1
rcv ack1
send pkt0
pkt0
ack1
rcv pkt0
send ack0
rcv ack1
ignore
ack0

rdt3.0: performance

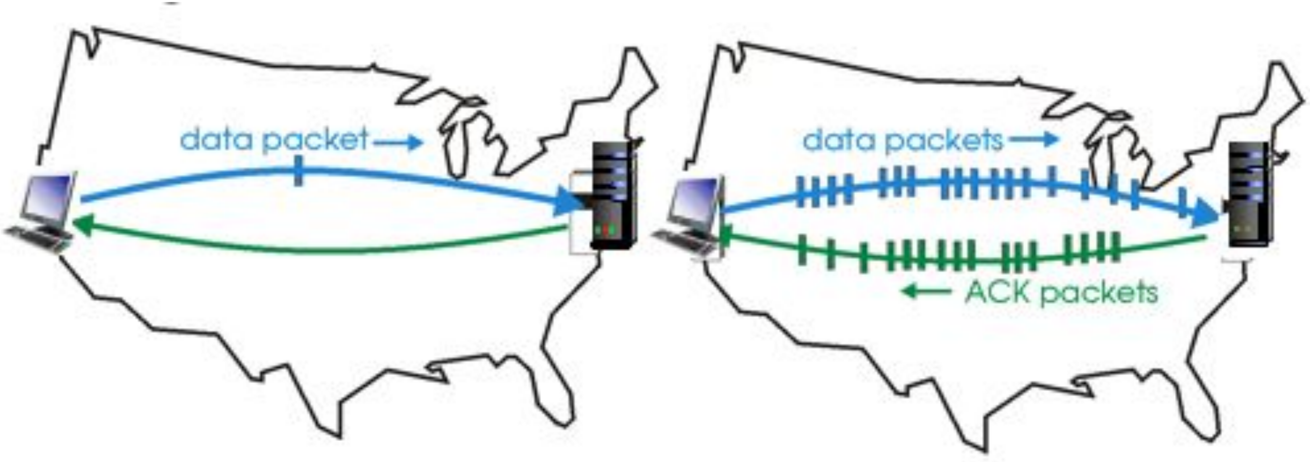
- R = 1Gbps link
- 15 ms propagation delay
- L = 8000 bit packet length
- Transmission delay: L/R = 8 micro seconds
- What's the throughput?
  - RTT = 30 ms, L = 8000 bits = 1KB
  - L/RTT = 1KB / 30ms = 33 KB/sec
  - But it is 1Gbps link!

rdt3.0: performance



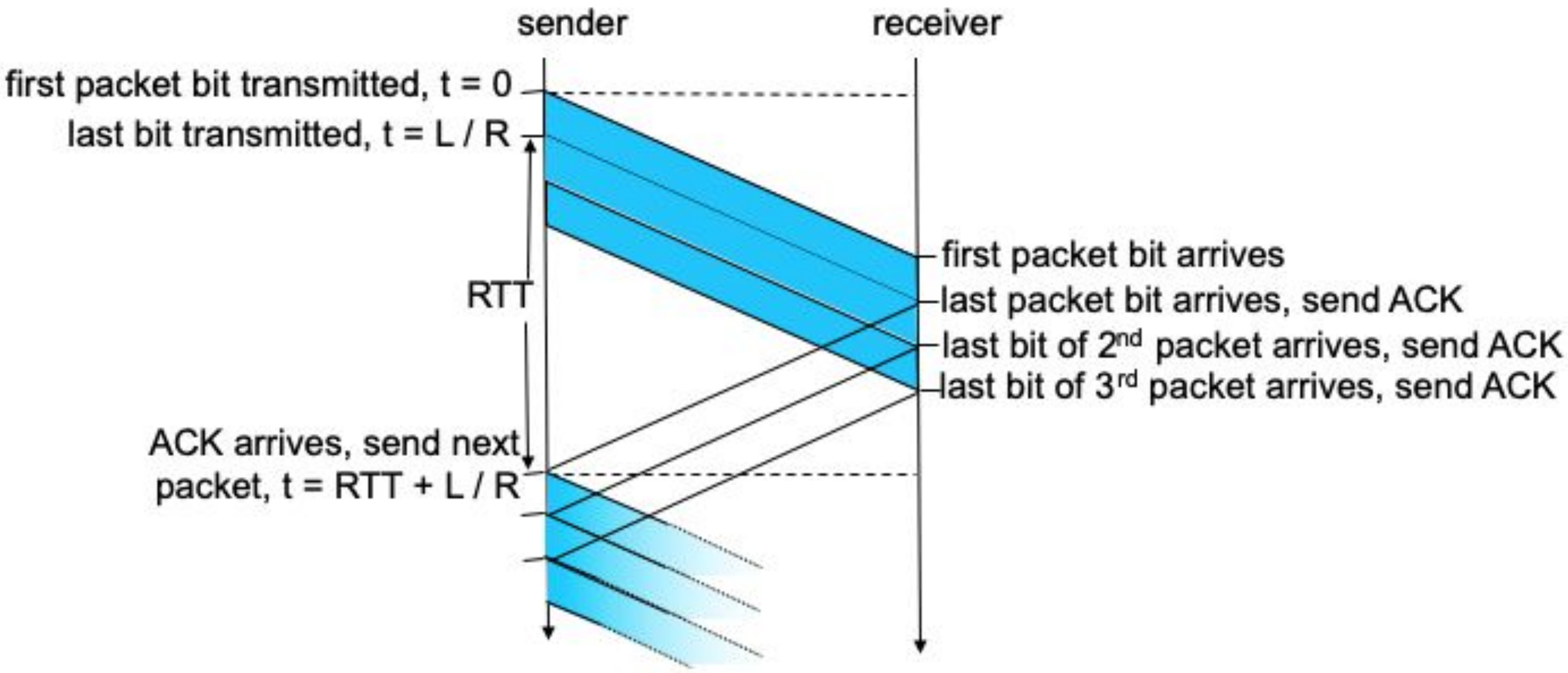
Pipelined protocols

- Sender allows multiple “in-flight” (yet-to-be-acked) pkts
- Go-Back-N and Selective-Repeat



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

Pipeline protocols



Pipelined protocols: GBN

- Go-Back-N
- Sender can have up to N un-ACKed packets
- Receiver only sends **cumulative ACK**
  - Doesn't ACK packet if there is a gap
- Sender has timer for oldest un-ACKed packet
  - When timer expires, retransmit all un-ACKed packets

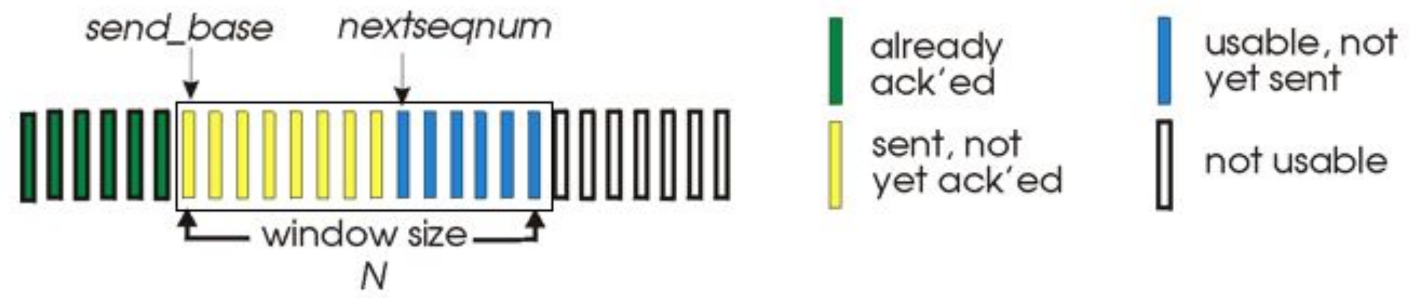
Pipelined protocols: SR

- Selective-Repeat
- Sender can have up to N un-ACKed packets
- Receiver sends individual ACK for each packet
- Sender maintains timer for each un-ACKed packet
    - When timer expires, retransmit only that un-ACKed packet

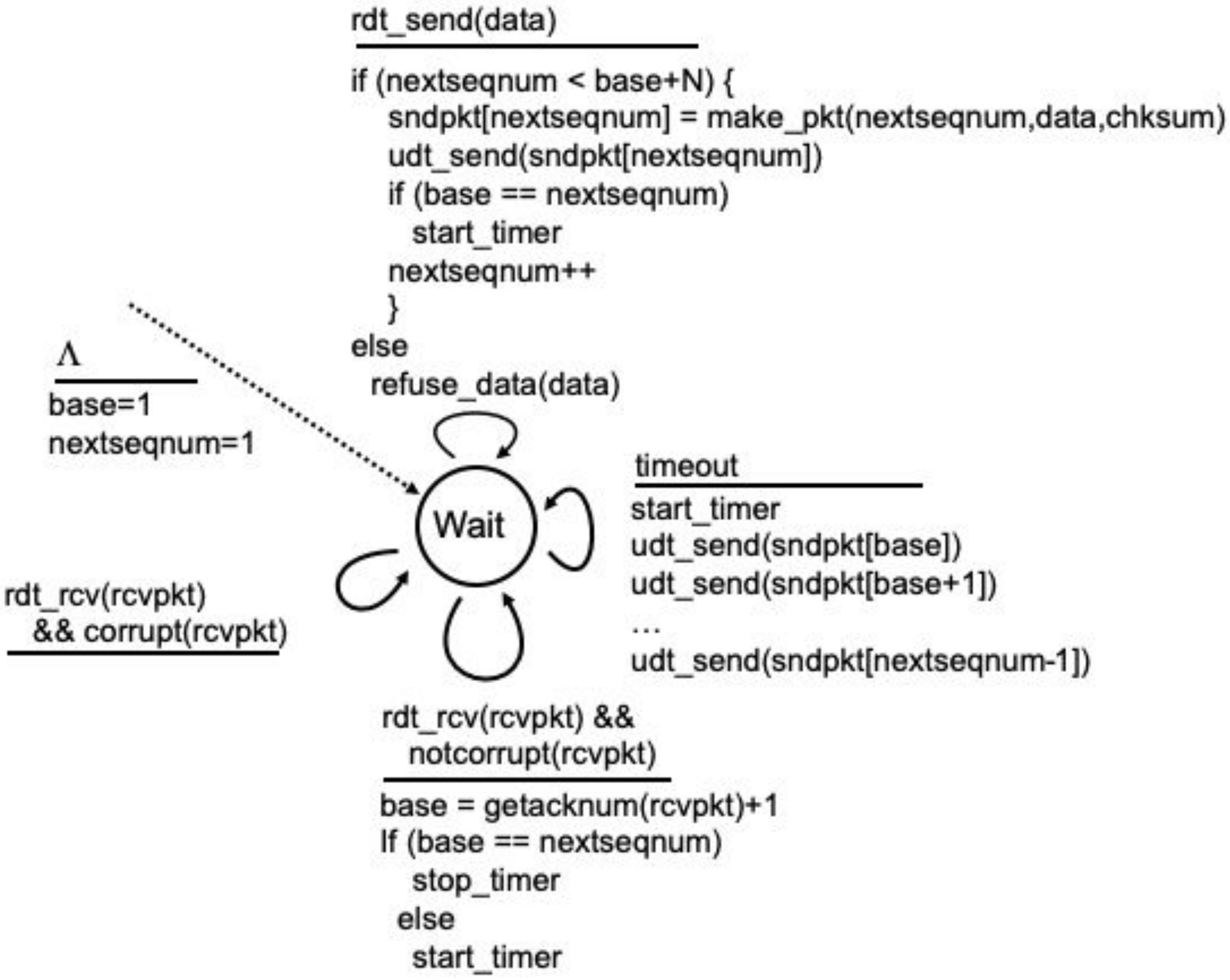
Which one do you like?

GBN: sender

- Buffer of up to N, consecutive un-ACKed pkts allowed
- ACK(n): ACK all pkts up to sequence number n
- Timer for oldeast in-flight pkt
- Timeout(n): retransmit packet n and all higher sequence number pkts in the buffer

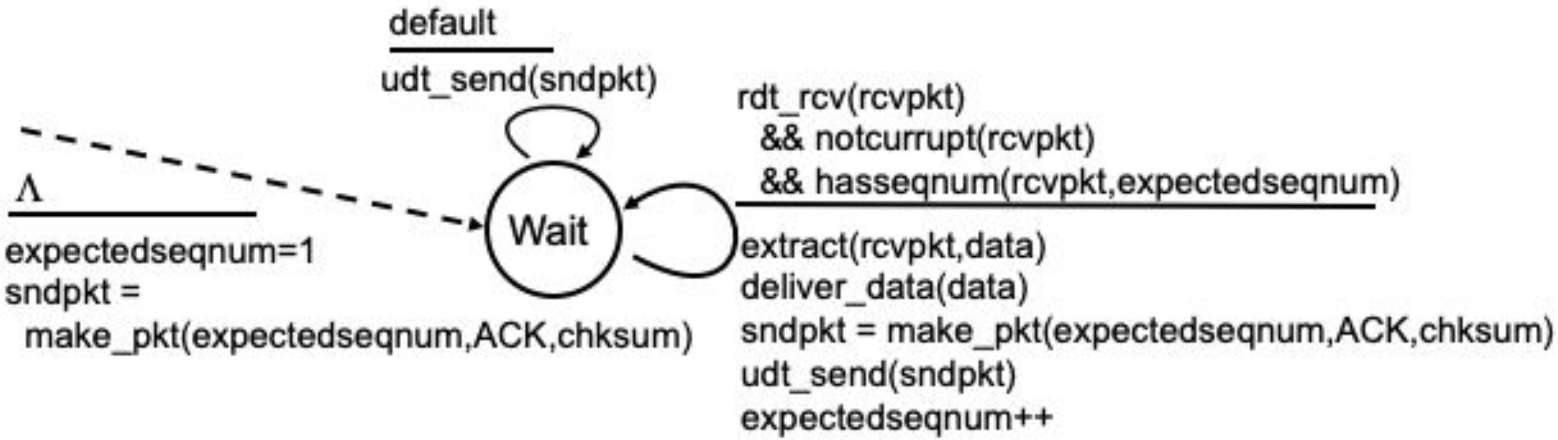


GBN

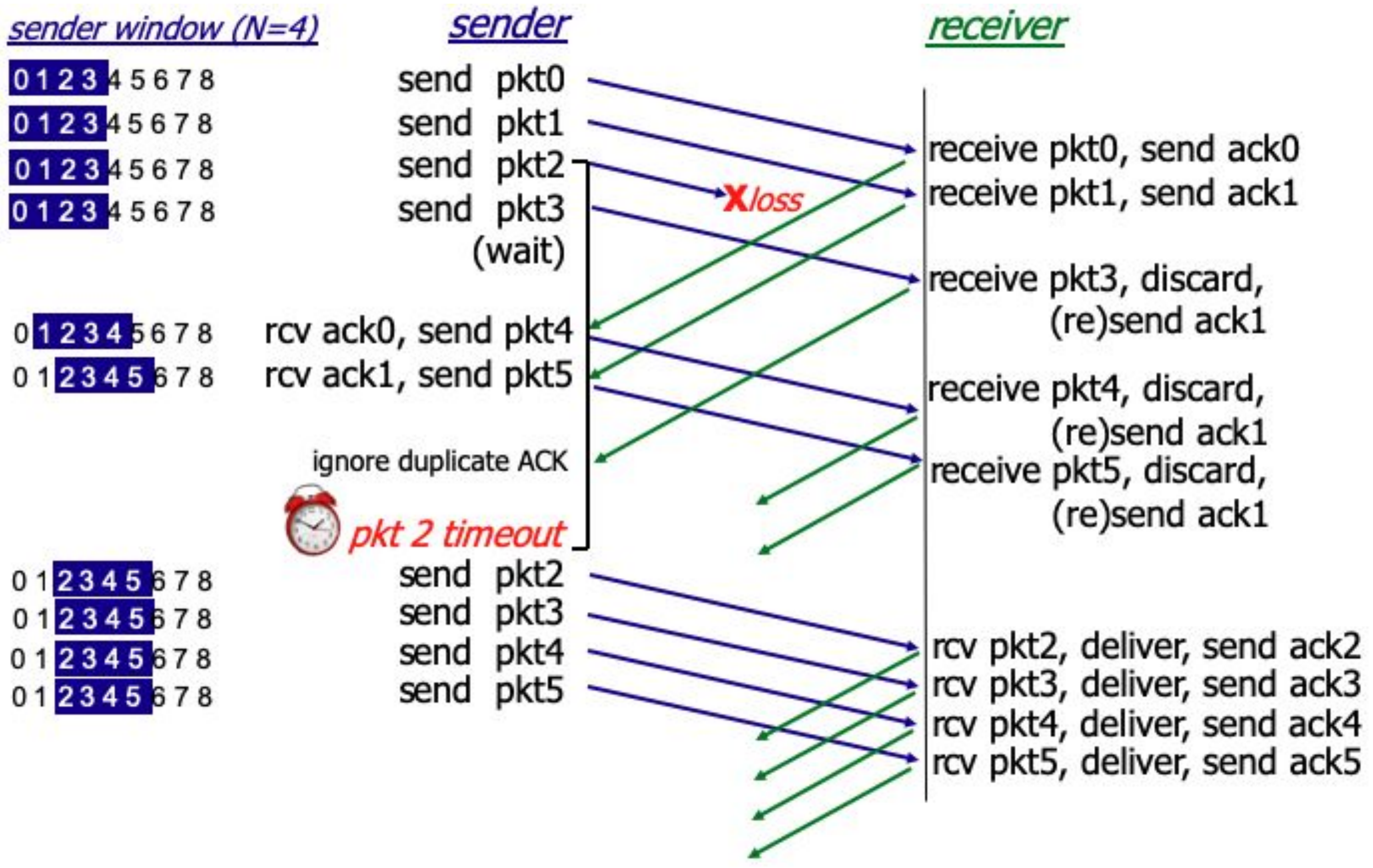

rdt_send(data)
if (nextseqnum < base+N) {
sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
udt_send(sndpkt[nextseqnum])
if (base == nextseqnum)
start_timer
nextseqnum++
}
else
refuse_data(data)
Λ
base=1
nextseqnum=1
Wait
timeout
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
...
udt_send(sndpkt[nextseqnum-1])
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
stop_timer
else
start_timer

GBN: receiver

- Always send ACK for correctly received pkt with highest sequence number
- No buffer, discard out-of-order pkt



GBN

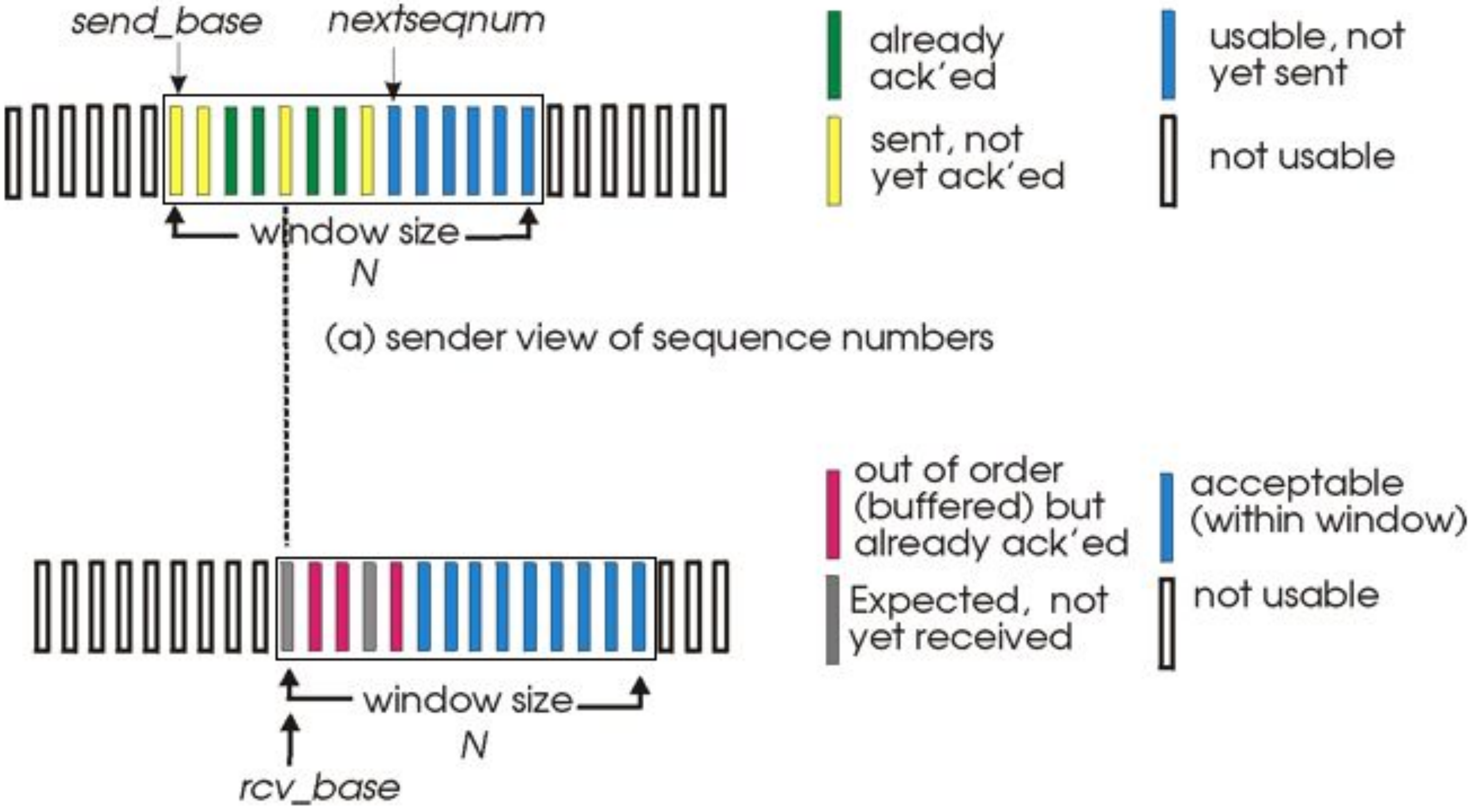


sender window (N=4)
sender
receiver
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, discard,
(re)send ack1
ignore duplicate ACK
receive pkt5, discard,
(re)send ack1
pkt 2 timeout
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

SR

- Receiver individually ACK all correctly received pkts
  - Buffer pkts as needed
- Sender only resends pkts for which ACK not received
  - One timer for each un-ACKed pkt
- Sender has buffer of size N
- Receiver has buffer of size N

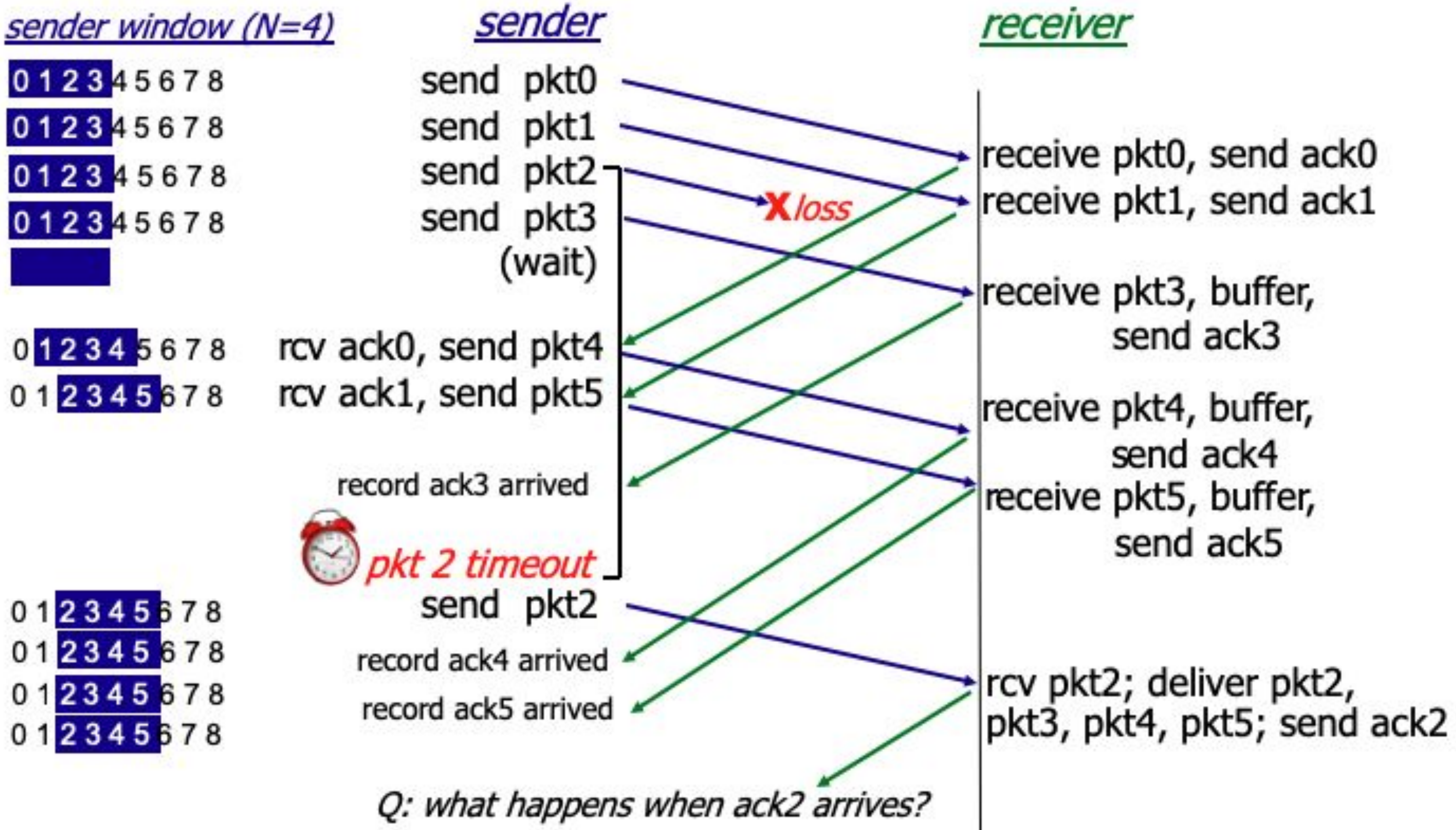
SR



(a) sender view of sequence numbers

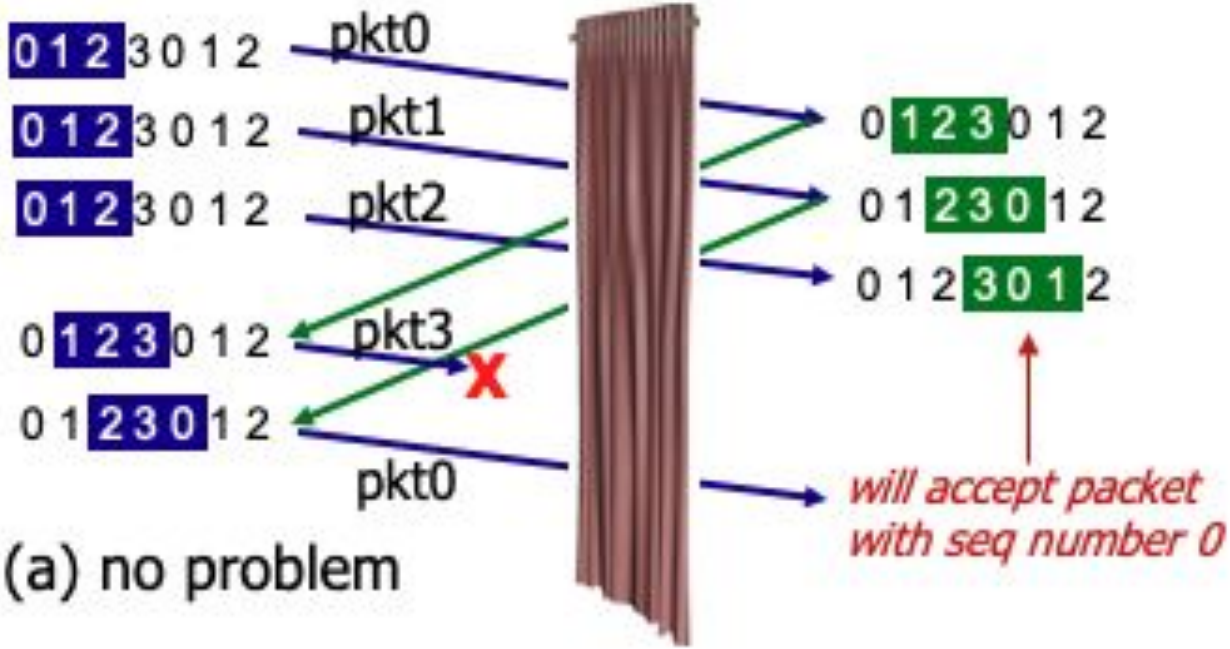
(b) receiver view of sequence numbers

SR



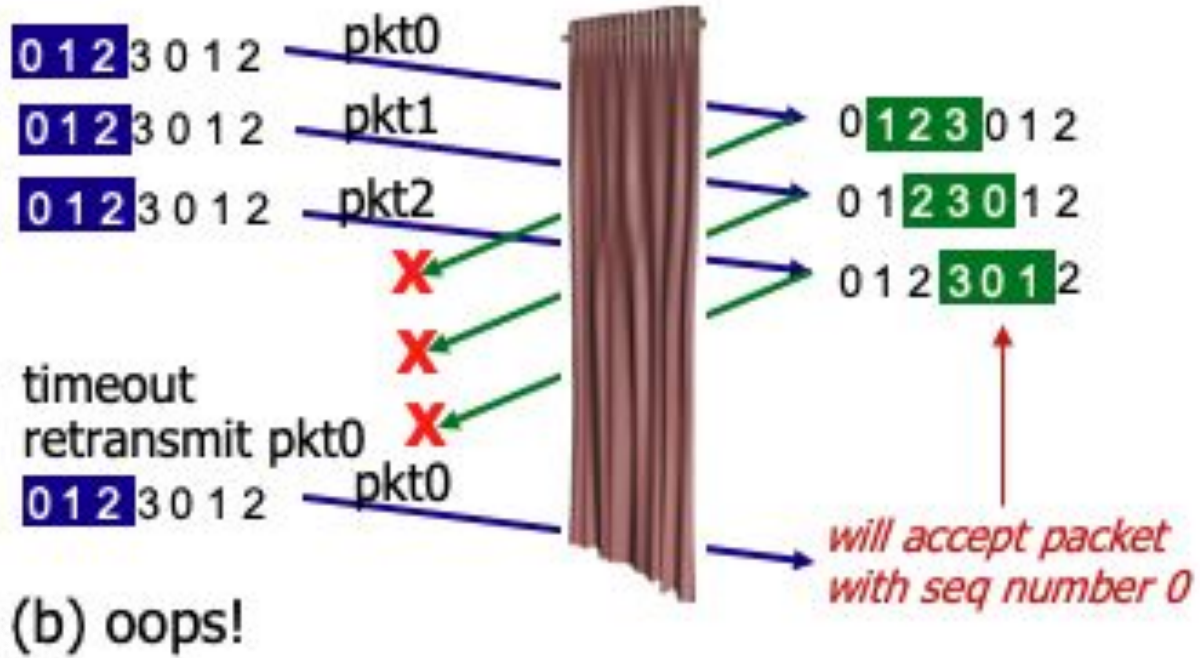
sender window (N=4)
sender
receiver
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,
send ack3
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, buffer,
send ack4
receive pkt5, buffer,
send ack5
record ack3 arrived
pkt 2 timeout
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
send pkt2
record ack4 arrived
record ack5 arrived
rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2
Q: what happens when ack2 arrives?

SR



(a) no problem

SR



0 1 2 3 0 1 2
pkt0
0 1 2 3 0 1 2
pkt1
0 1 2 3 0 1 2
pkt2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
X
X
X
timeout
retransmit pkt0
0 1 2 3 0 1 2
pkt0
will accept packet
with seq number 0
(b) oops!

Can receiver tell the difference?