Transport Layer

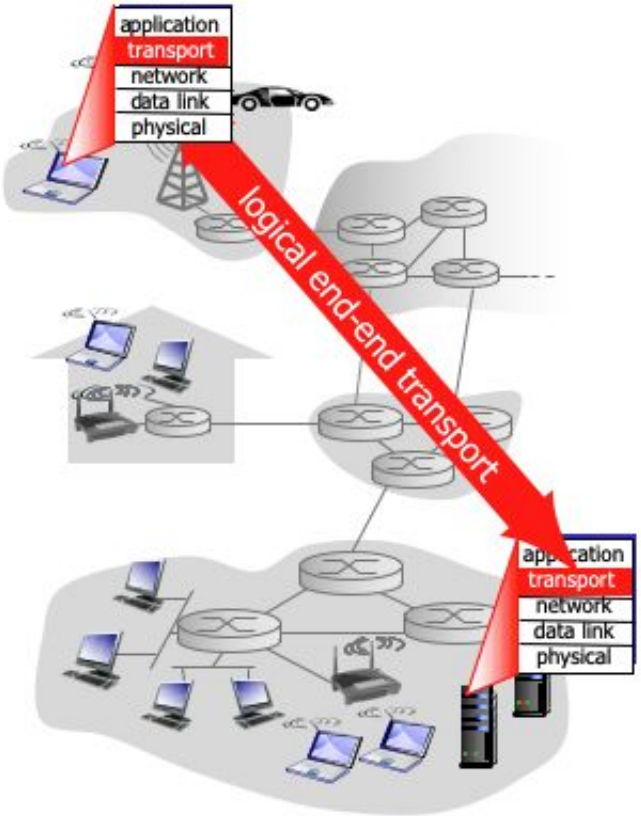
CS5700 Fall 2019

Agenda

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

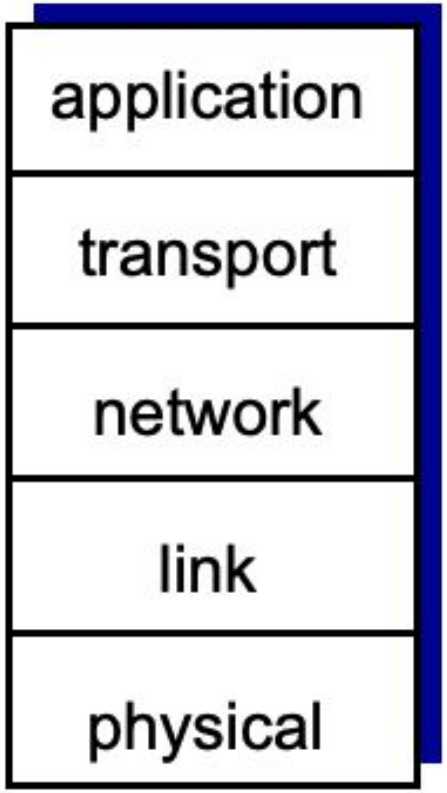
Transport services and protocols

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



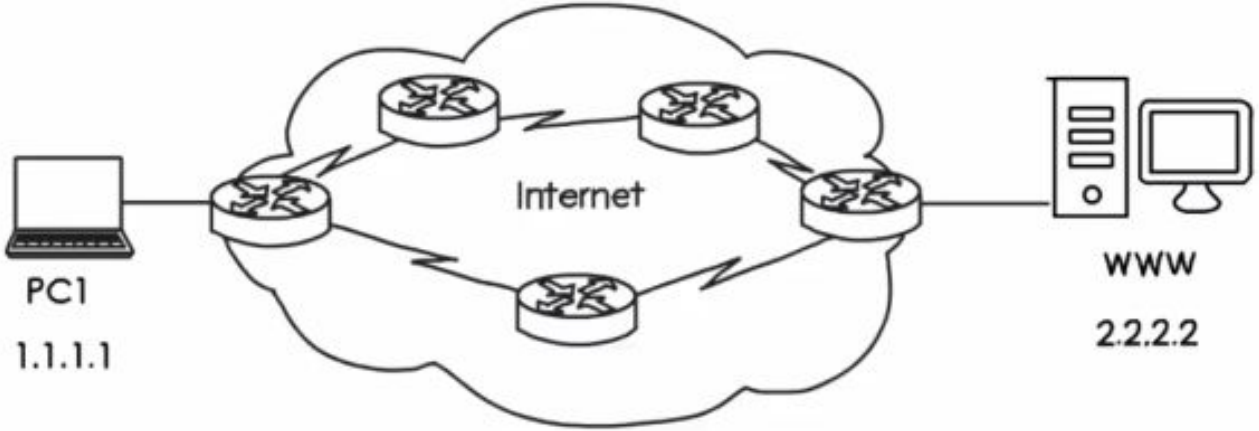
What does network layer do?

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



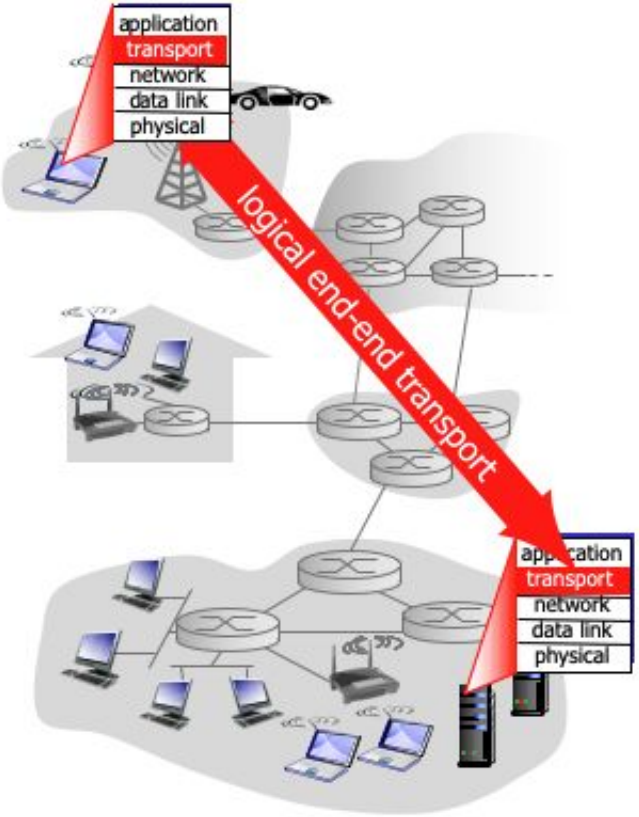
Network layer service model

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



Transport layer protocols

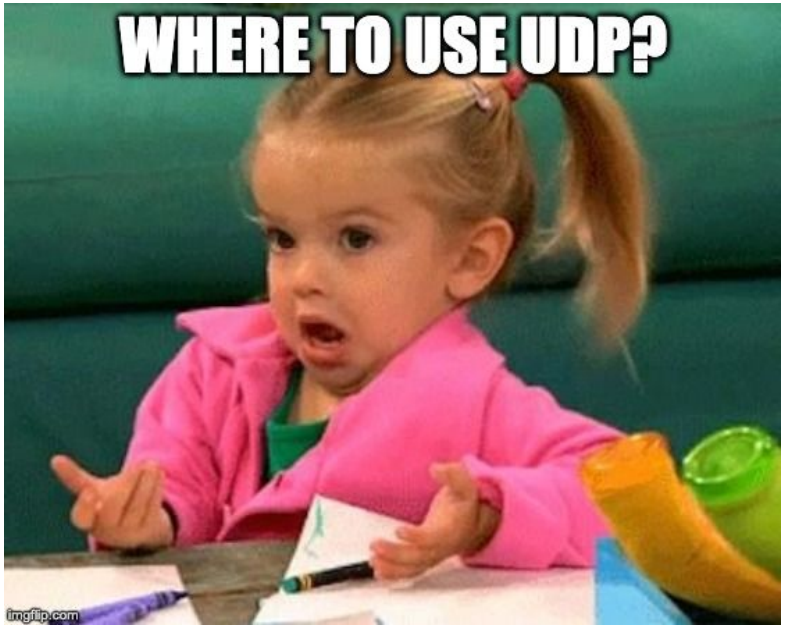
- TCP
  - Reliable in-order delivery
  - Connection oriented
  - Flow control
  - Congestion control
- UDP
- Services not available
  - Delay or bandwidth guarantee



UDP

UDP

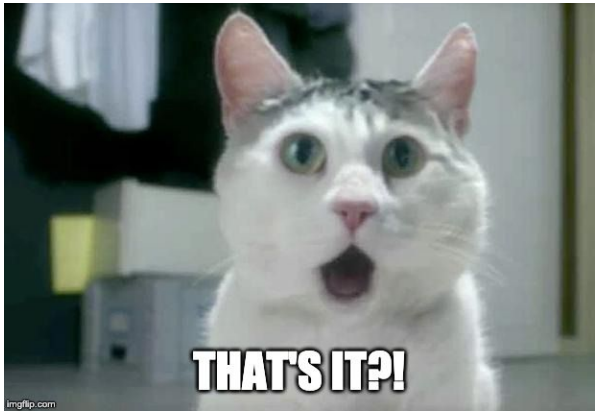
- User Datagram Protocol
  - Connection less
  - No guarantee of delivery
- Where do you see UDP used? Do you know why?



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

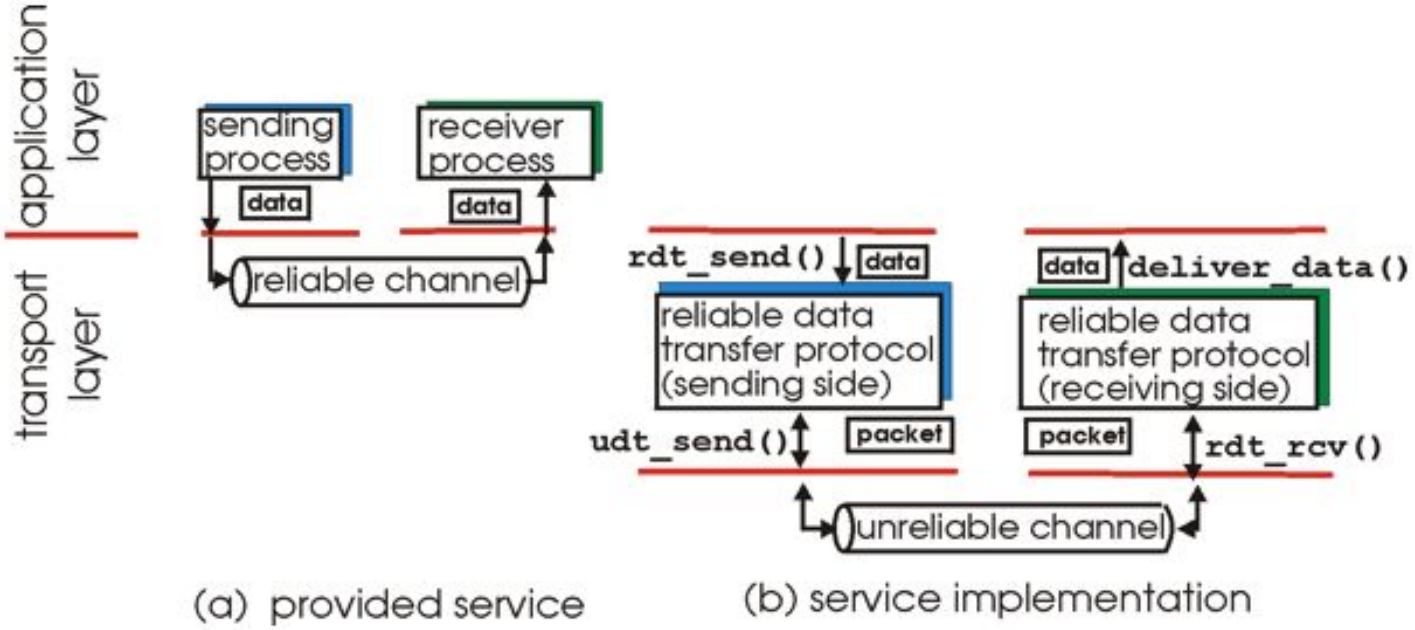
- Detect “errors” (e.g. flipped bits) in transmitted segment
- Sender
  - Treat data (include header) as seq of 16-bit integers
  - Add them up (1’s complement), call it checksum
  - Put checksum into UDP header
- Receiver
  - Same algorithm, compute checksum and compare

UDP socket

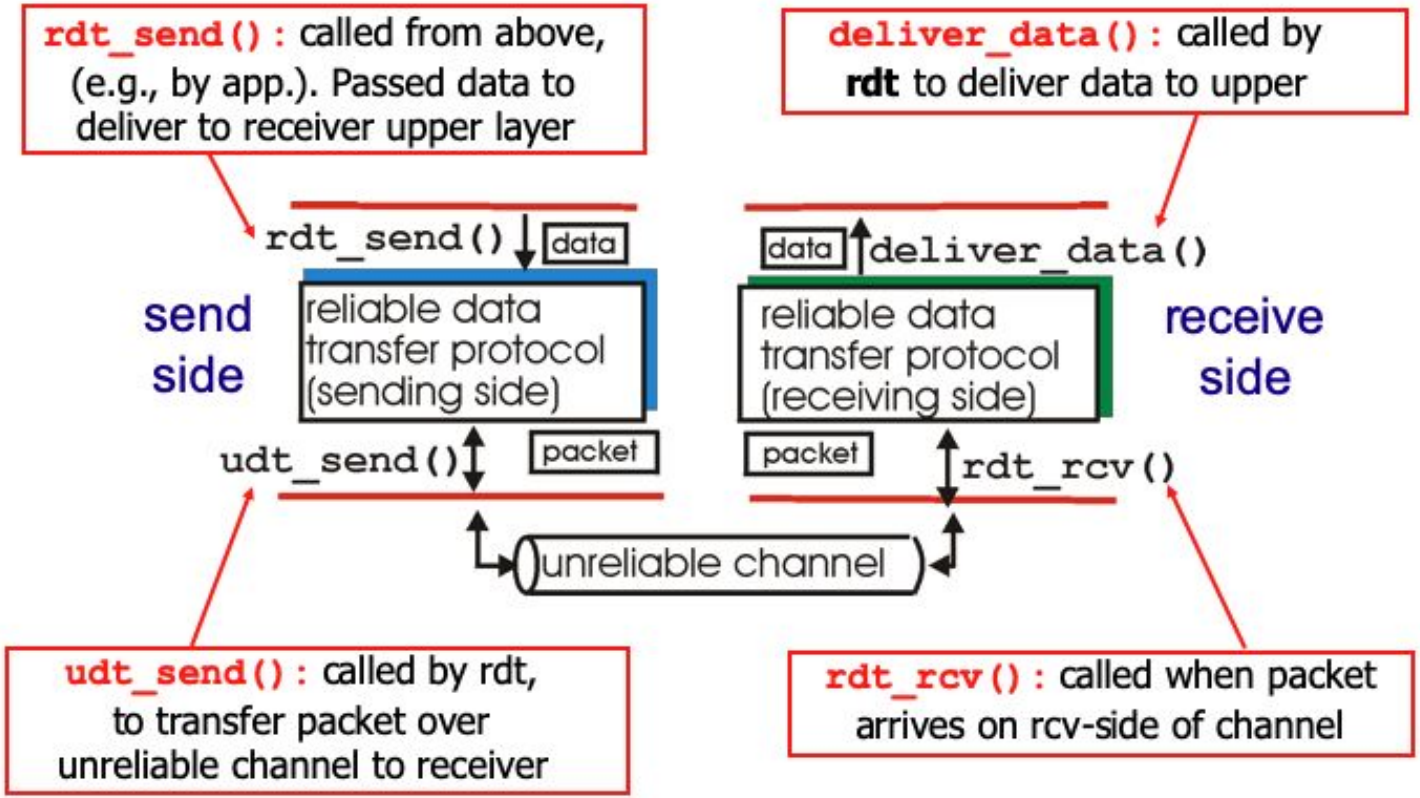
Reliable data transfer

Reliable data transfer

- Important in application, transport, and link layers

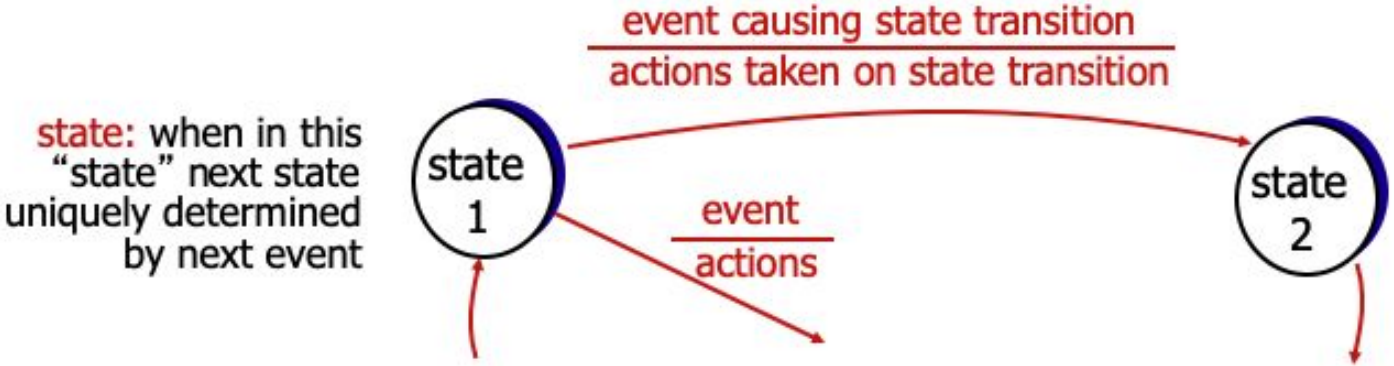


Reliable data transfer



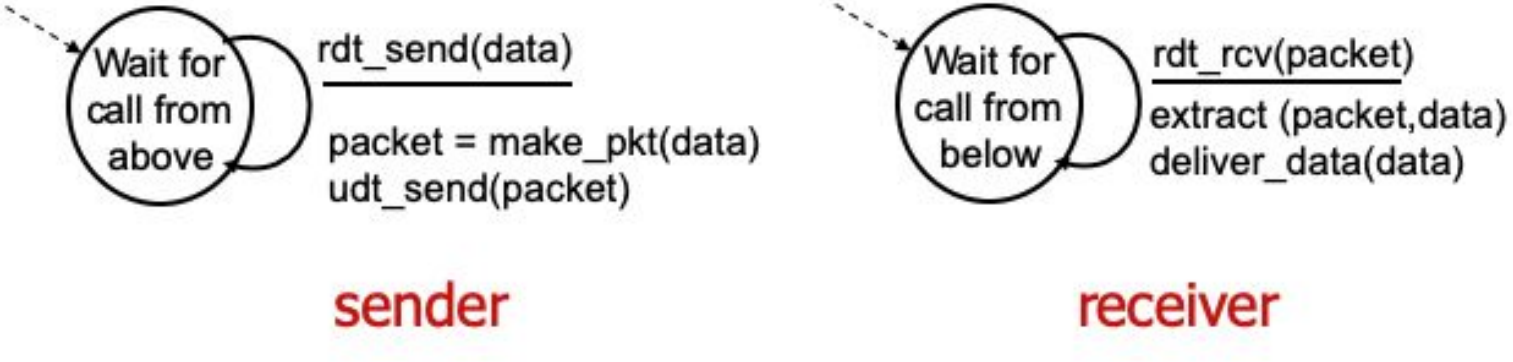
Reliable data transfer

- 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



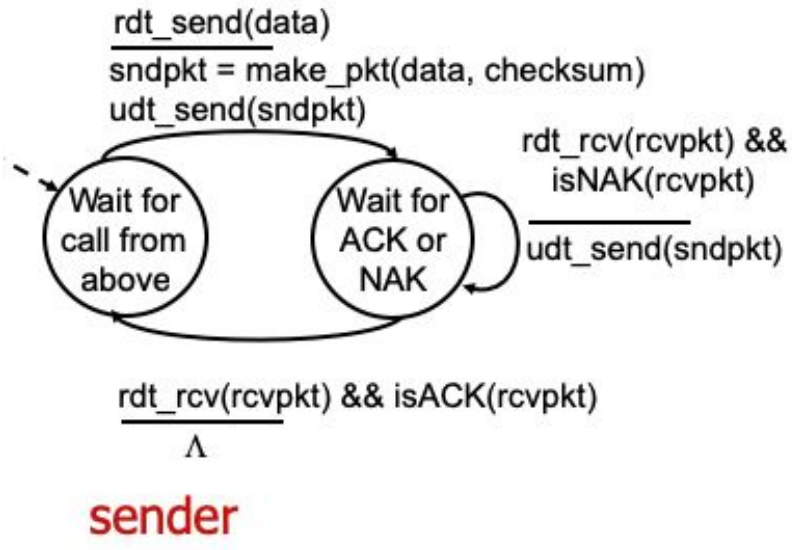
rdt2.0: channel with bit errors

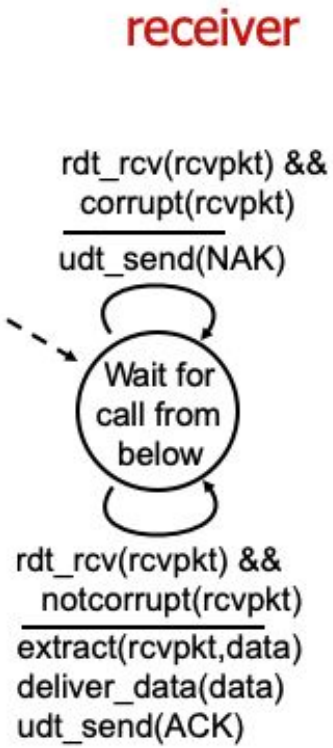
- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- How to recover from errors?

rdt2.0: channel with bit errors

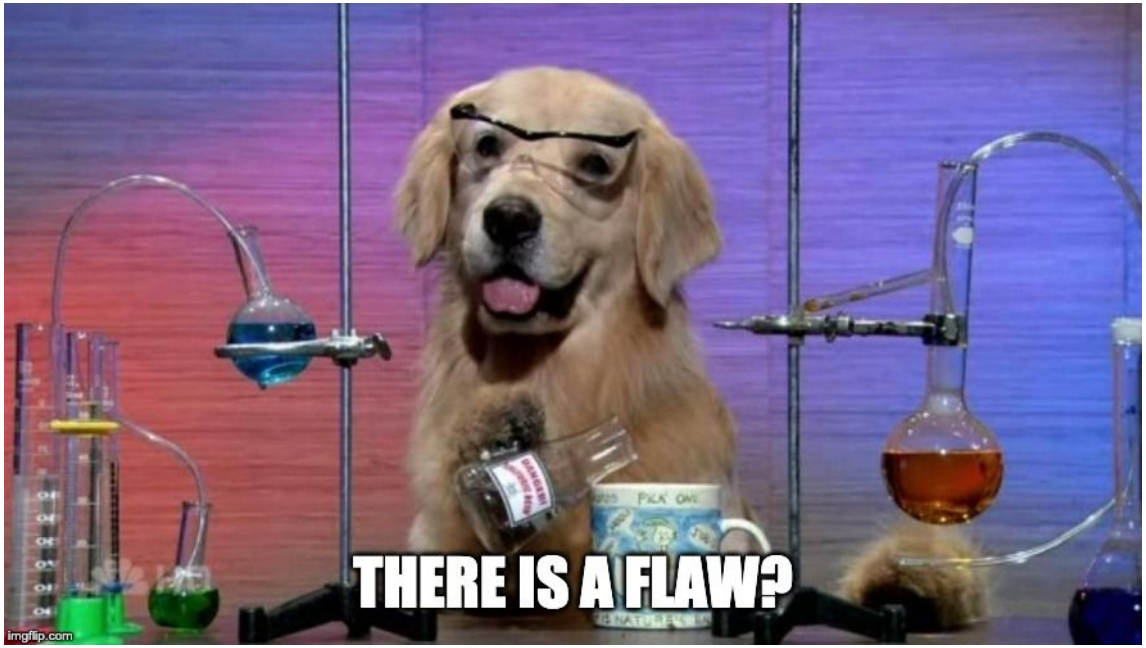
- ACK (acknowledgement)
  - Receiver explicitly tells sender that pkt received OK
- NAK (negative acknowledgement)
  - Receiver explicitly tells sender that pkt had errors
- Sender needs to retransmit pkt on receipt of NAK
- Summary
  - Error detection
  - Feedback with control message ACK and NAK

rdt2.0: FSM


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

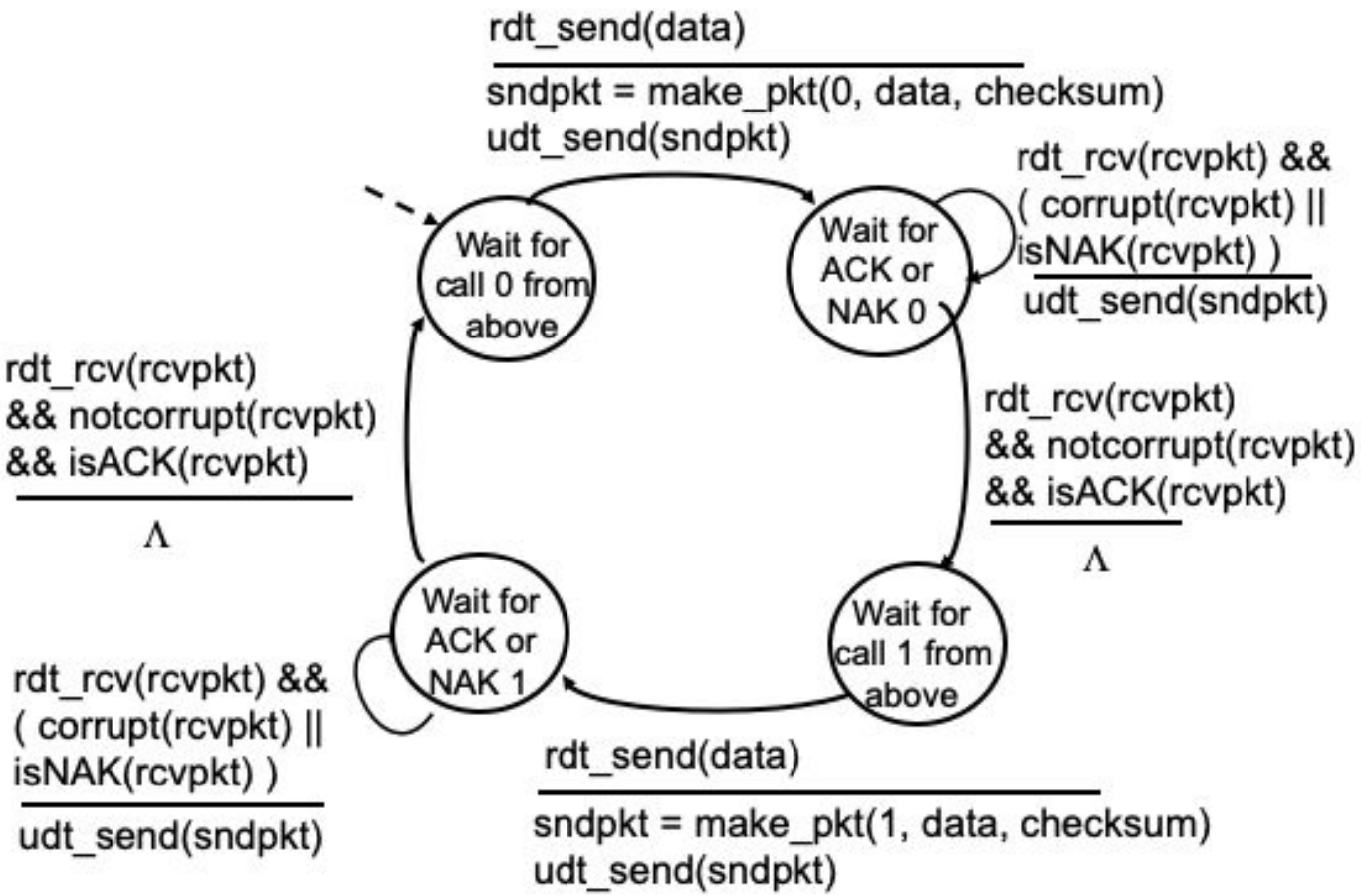
- What happens if ACK or NAK is corrupted?
  - Sender doesn’t know what happened at receiver
- Can sender just retransmit?

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

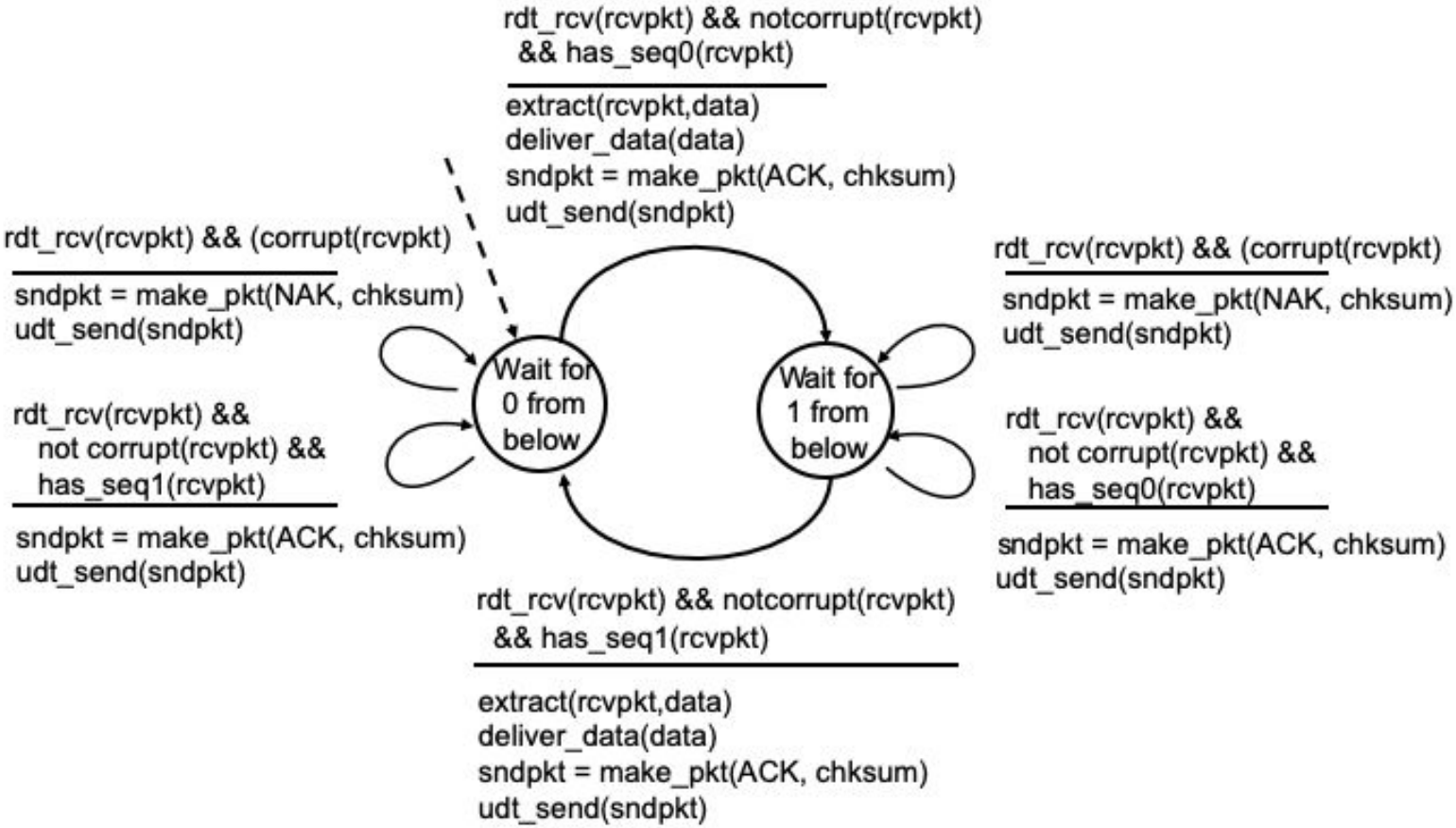
rdt2.1

Sender



rdt2.1

Receiver



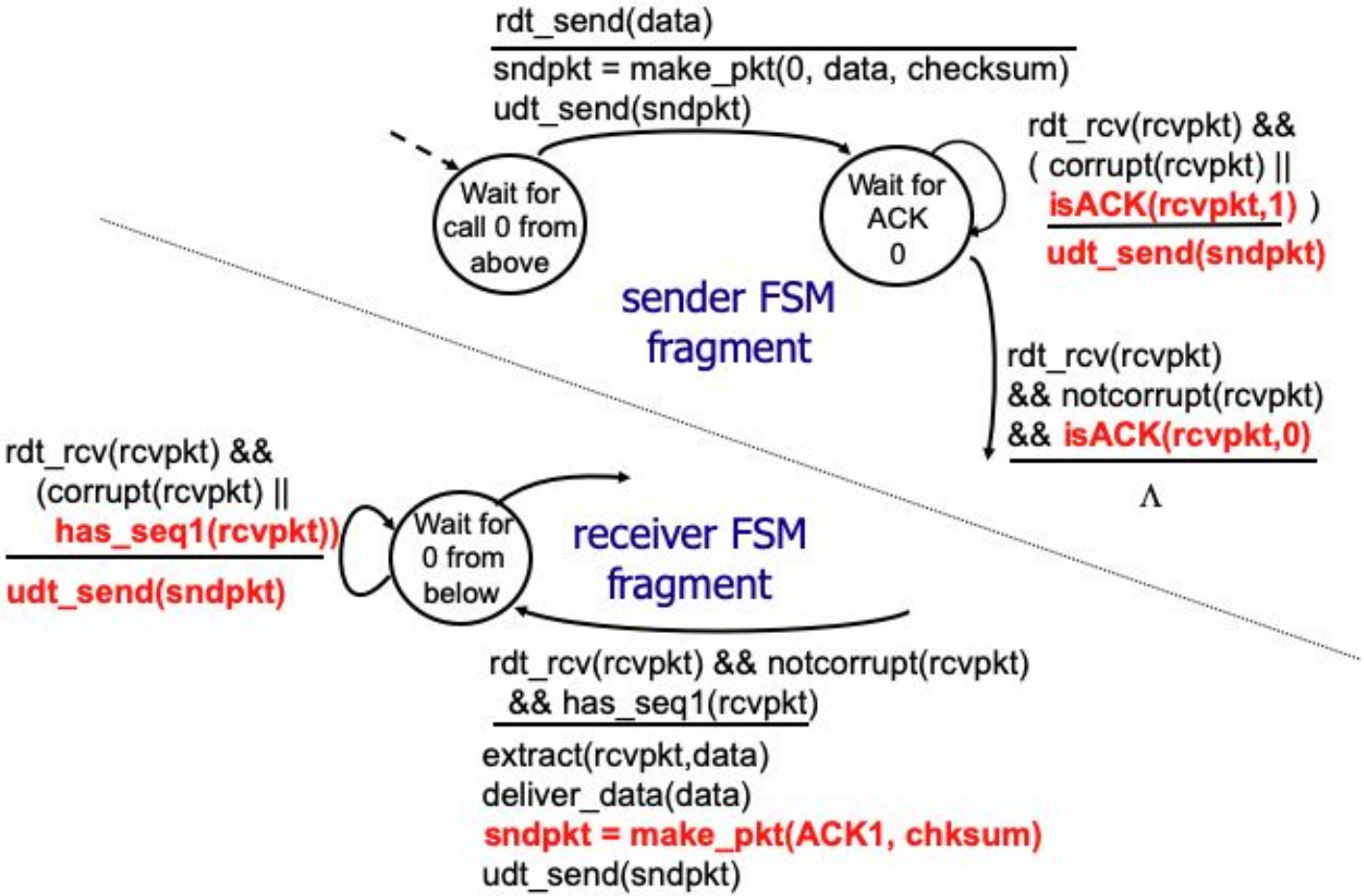
rdt2.1: summery

- 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

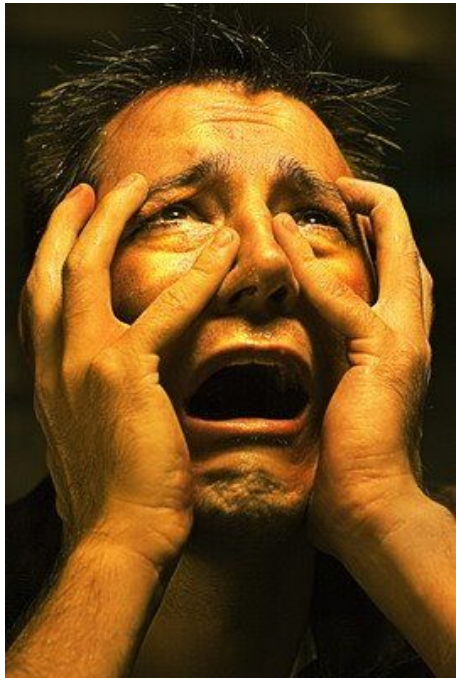
- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt
  - Receiver must explicitly include sequence number now in ACK message
- Duplicate ACK at sender results in same action as NAK
  - Retransmit current packet

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...

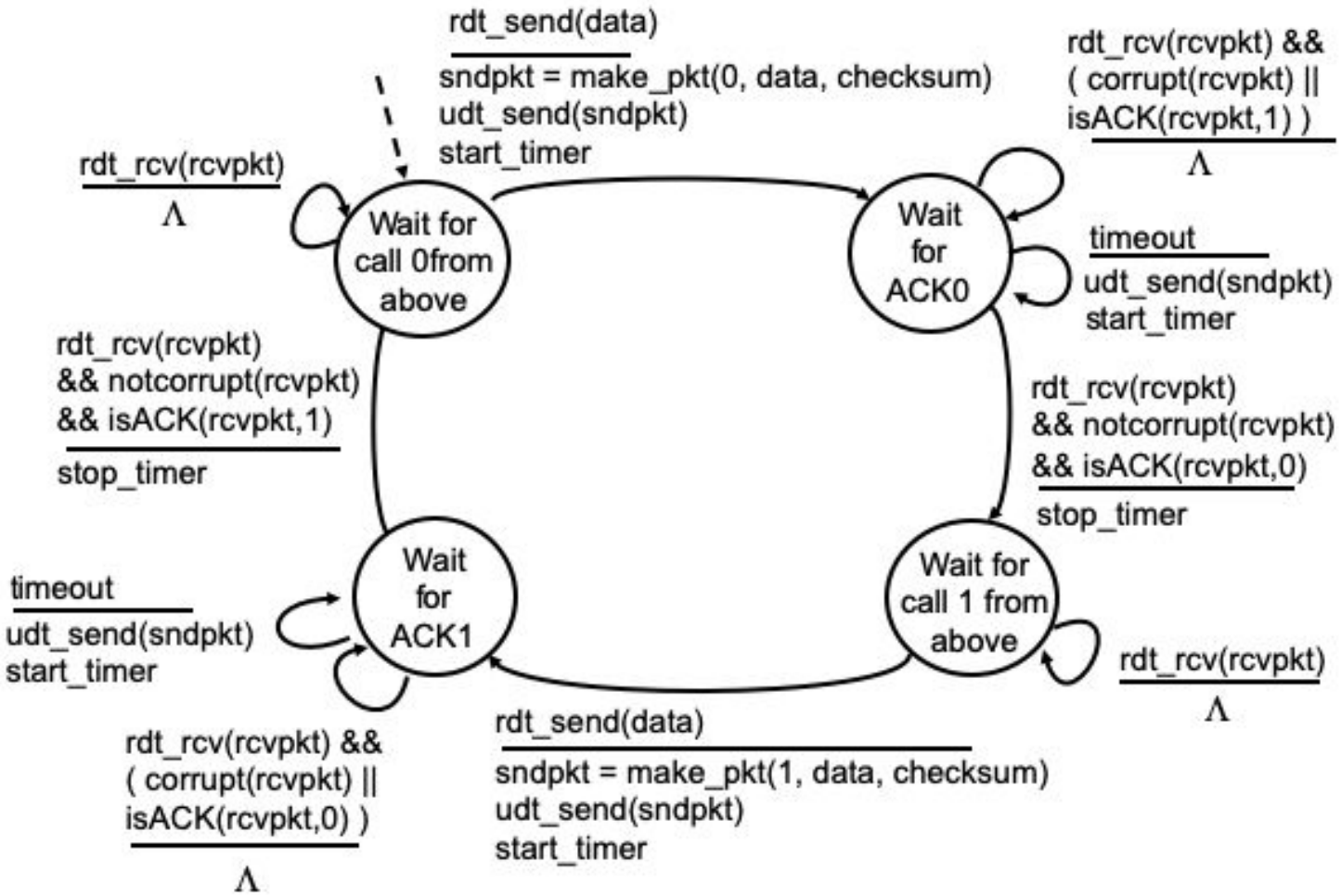


rdt3.0

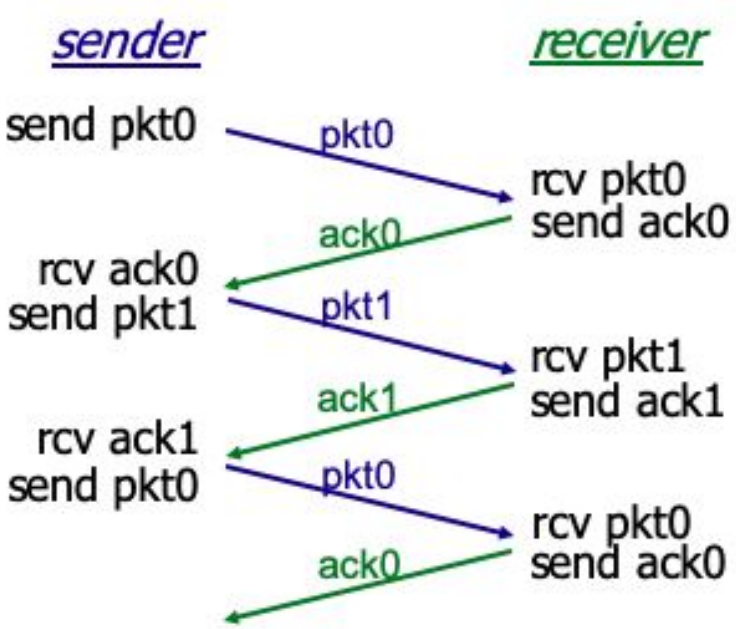
- Need timeout!
  - “Reasonable” amount of time for ACK
  - Retransmit if no ACK received in this time
    - Maybe delayed, maybe lost
  - Receiver must specify the sequence number of packet being ACKed

rdt3.0

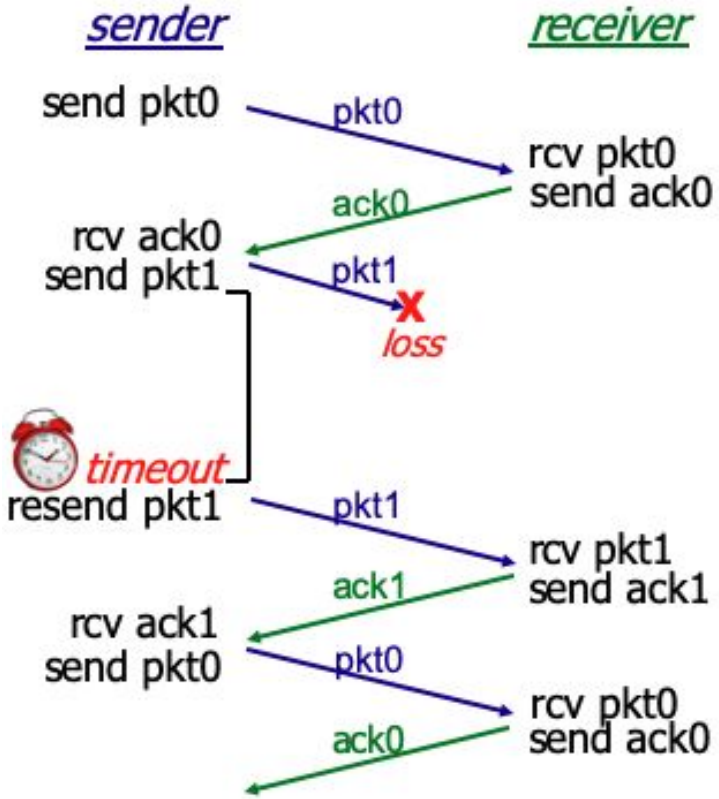
Sender



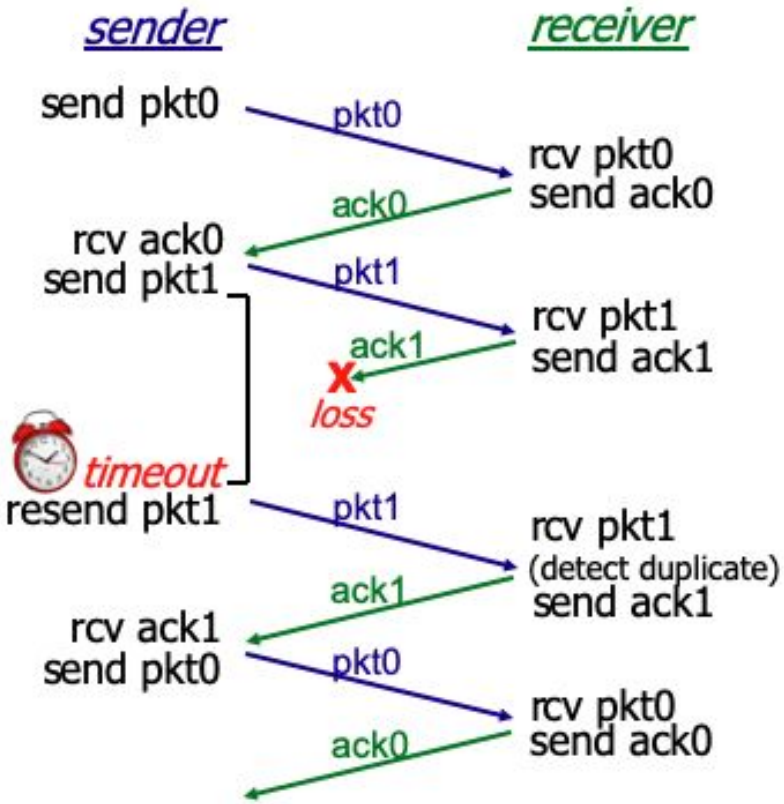
rdt3.0


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

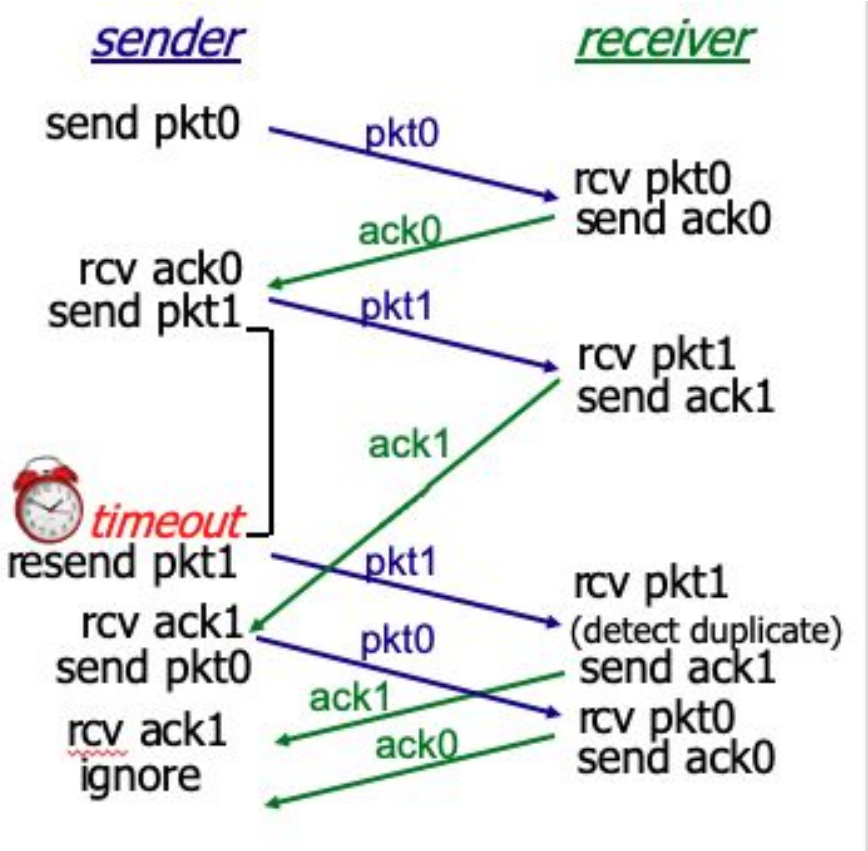
rdt3.0


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

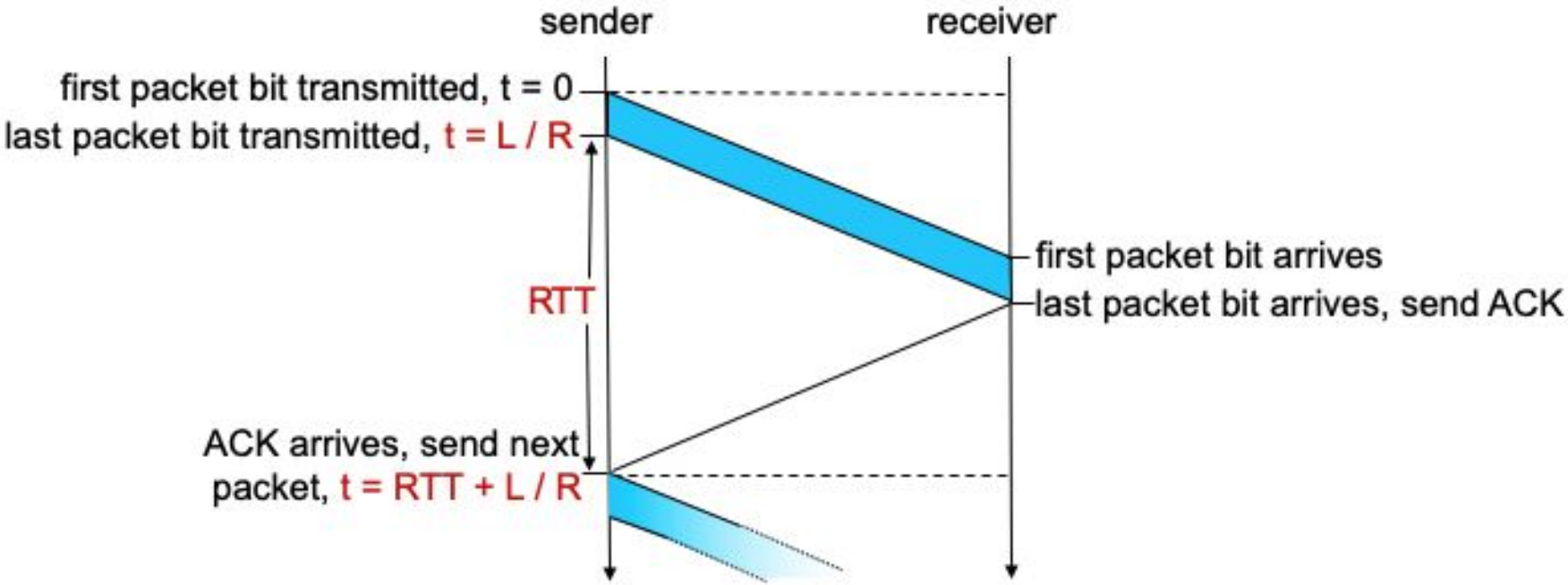
rdt3.0


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
rcv pkt0
send ack0
ack1
rcv ack1
ignore
ack0

rdt3.0: performance

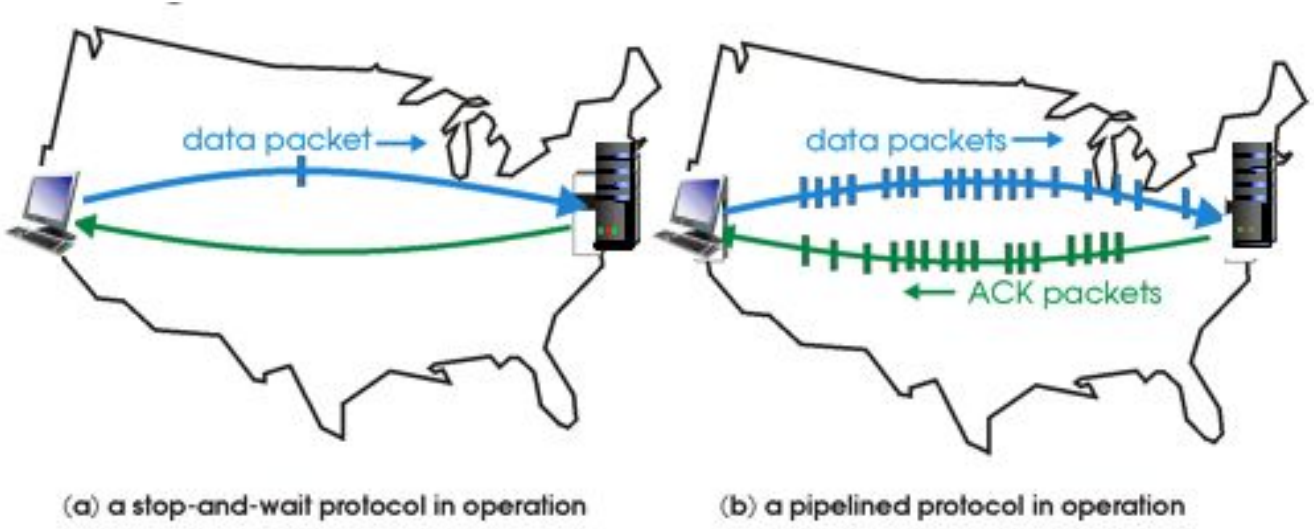
- R = 1Gbps link
- 15 ms propagation delay
- L = 8000 bit packet length
- Transmission delay: L/R = 8 micro seconds
- What's the throughput?
  - RTT = 30 ms, L = 8000 bits = 1KB
  - L/RTT = 1KB / 30ms = 33 KB/sec
  - But it is 1Gbps link!

rdt3.0: performance



Pipelined protocols

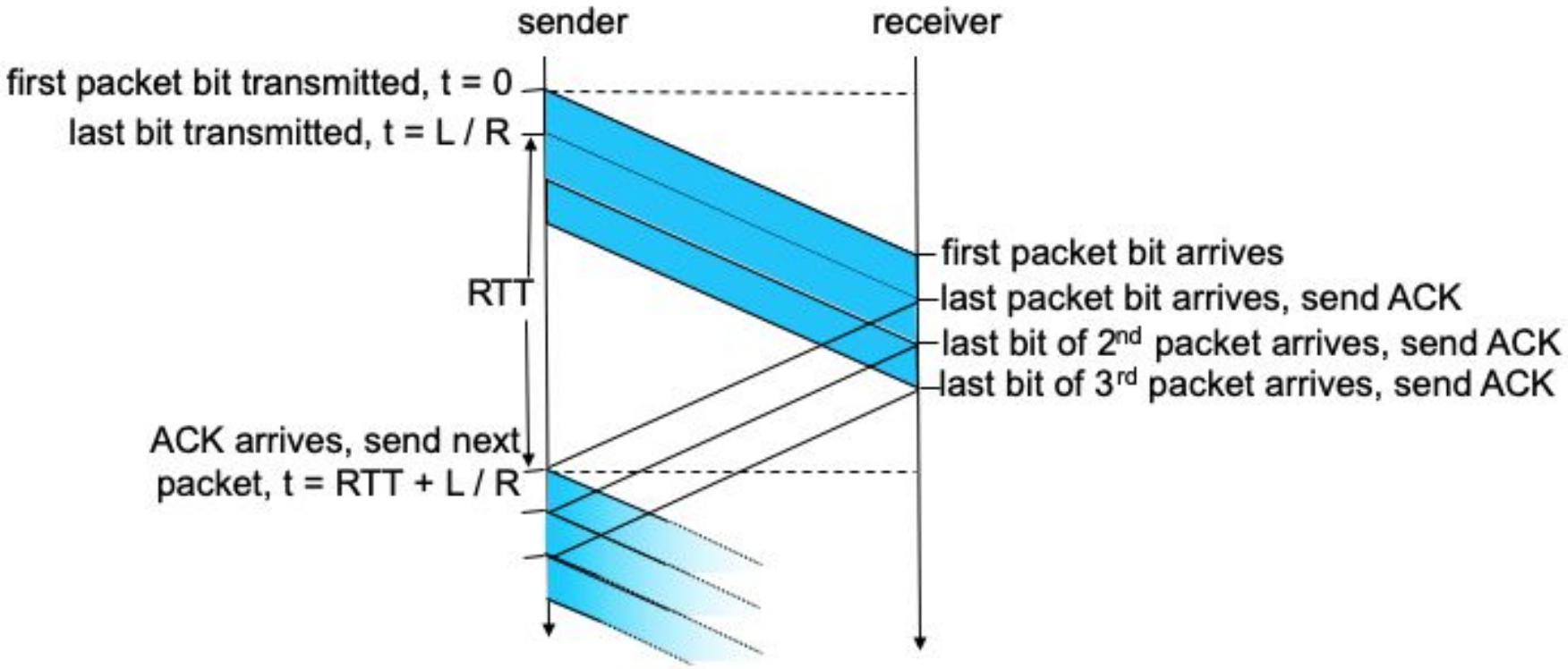
- Sender allows multiple “in-flight” (yet-to-be-acked) pkts
- Go-Back-N and Selective-Repeat



(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

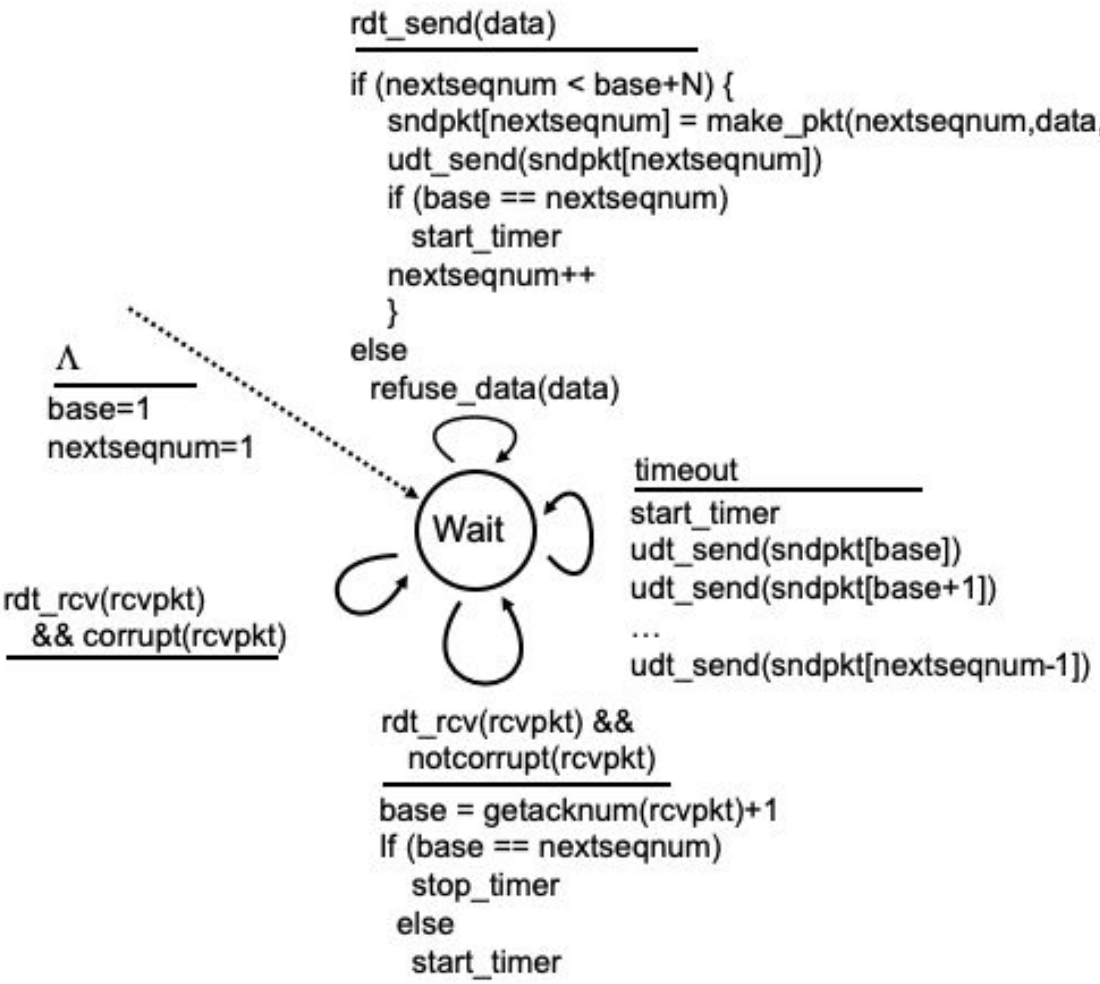
- Selective-Repeat
- Sender can have up to N un-ACKed packets
- Receiver sends individual ACK for each packet
- Sender maintains timer for each un-ACKed packet
  - When timer expires, retransmit only that un-ACKed packet

Which one do you like?

GBN: sender

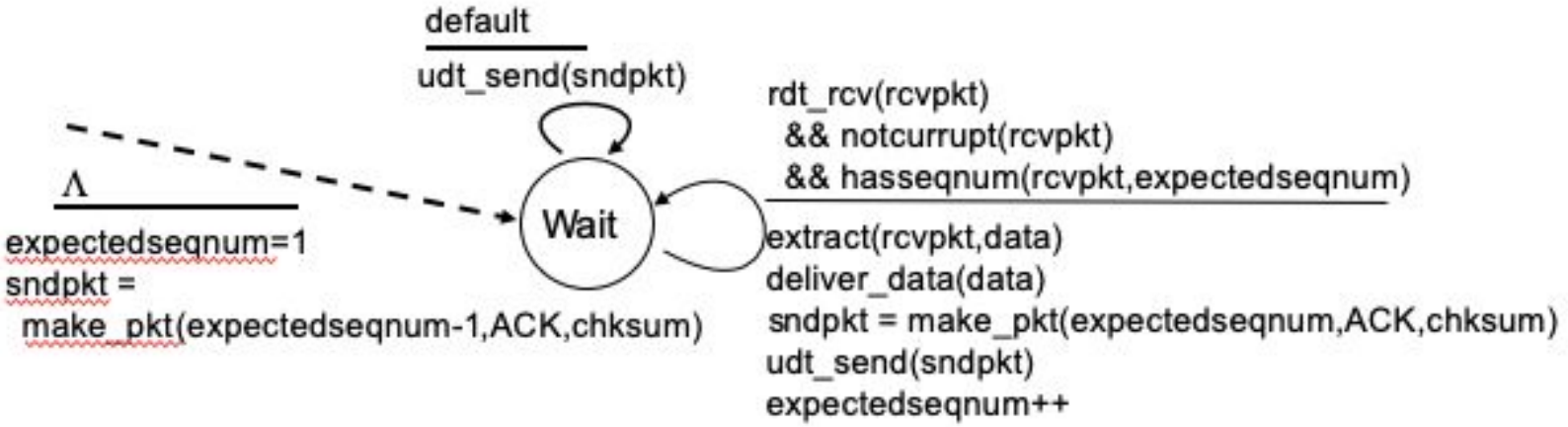
- Buffer of up to N, consecutive un-ACKed pkts allowed
- ACK(n): ACK all pkts up to sequence number n
- Timer for oldeast in-flight pkt
- Timeout(n): retransmit packet n and all higher sequence number pkts in the buffer

GBN

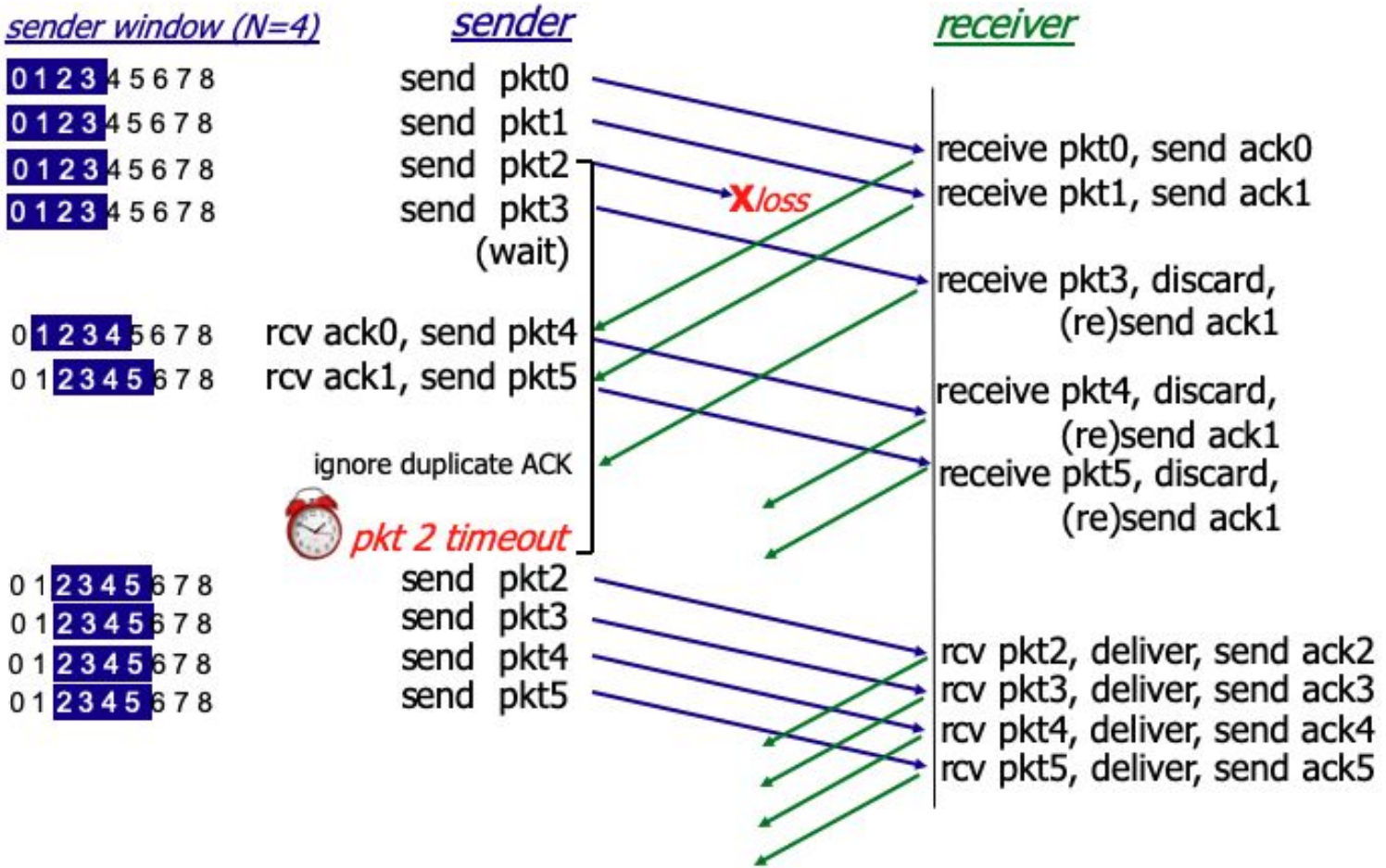

rdt_send(data)
if (nextseqnum < base+N) {
sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
udt_send(sndpkt[nextseqnum])
if (base == nextseqnum)
start_timer
nextseqnum++
}
else
refuse_data(data)
Λ
base=1
nextseqnum=1
Wait
timeout
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])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



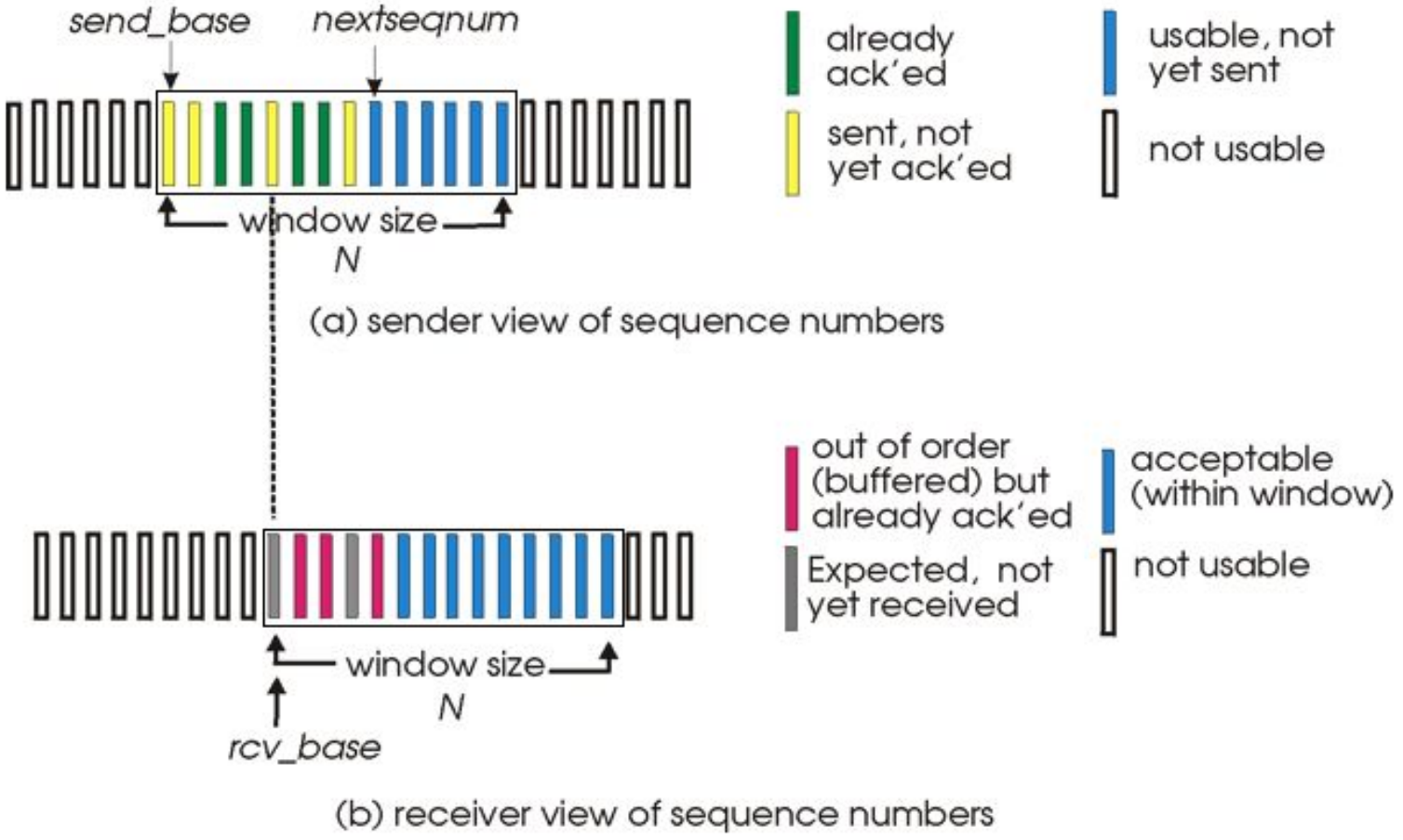
GBN


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

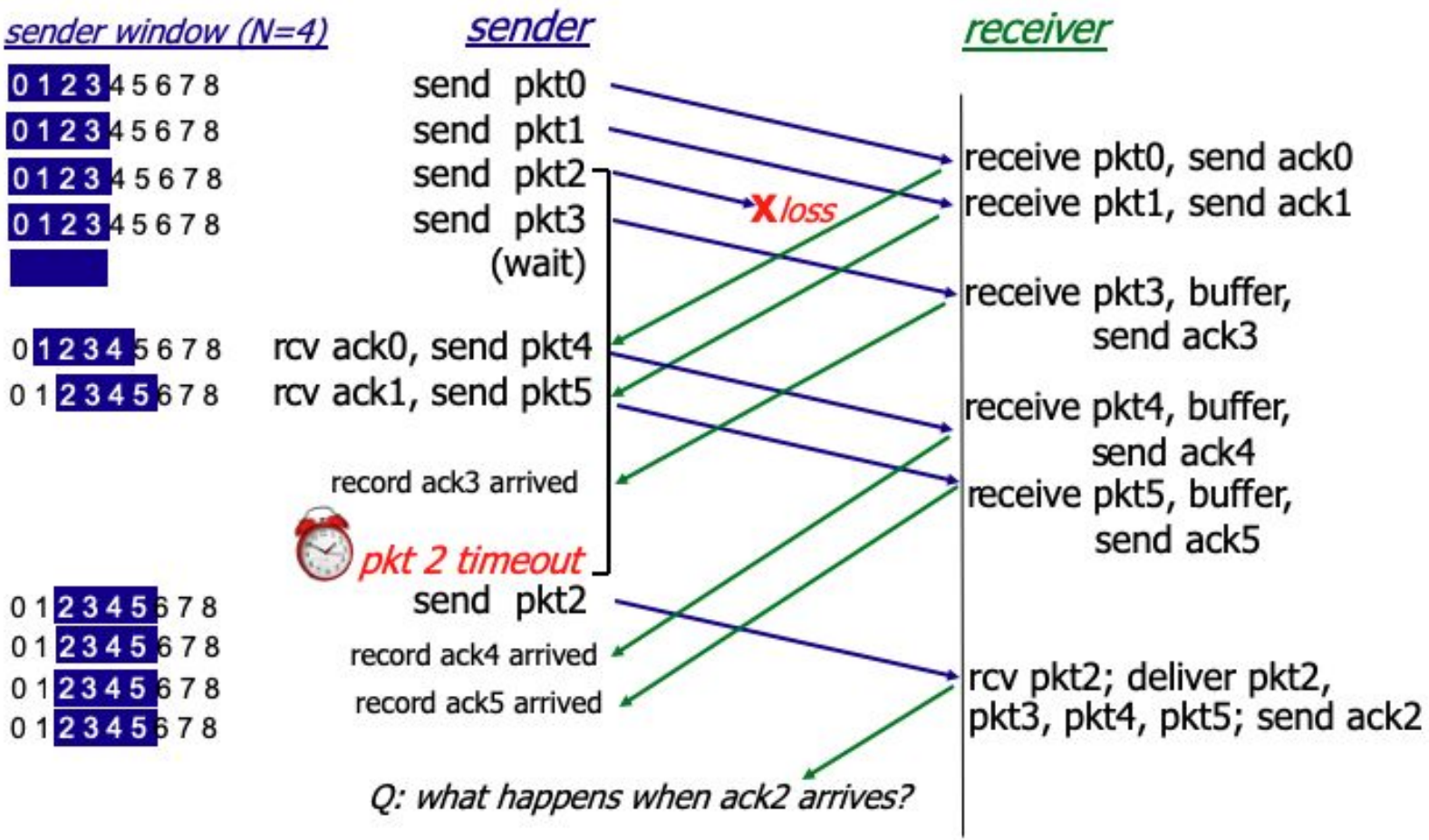
SR

- 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

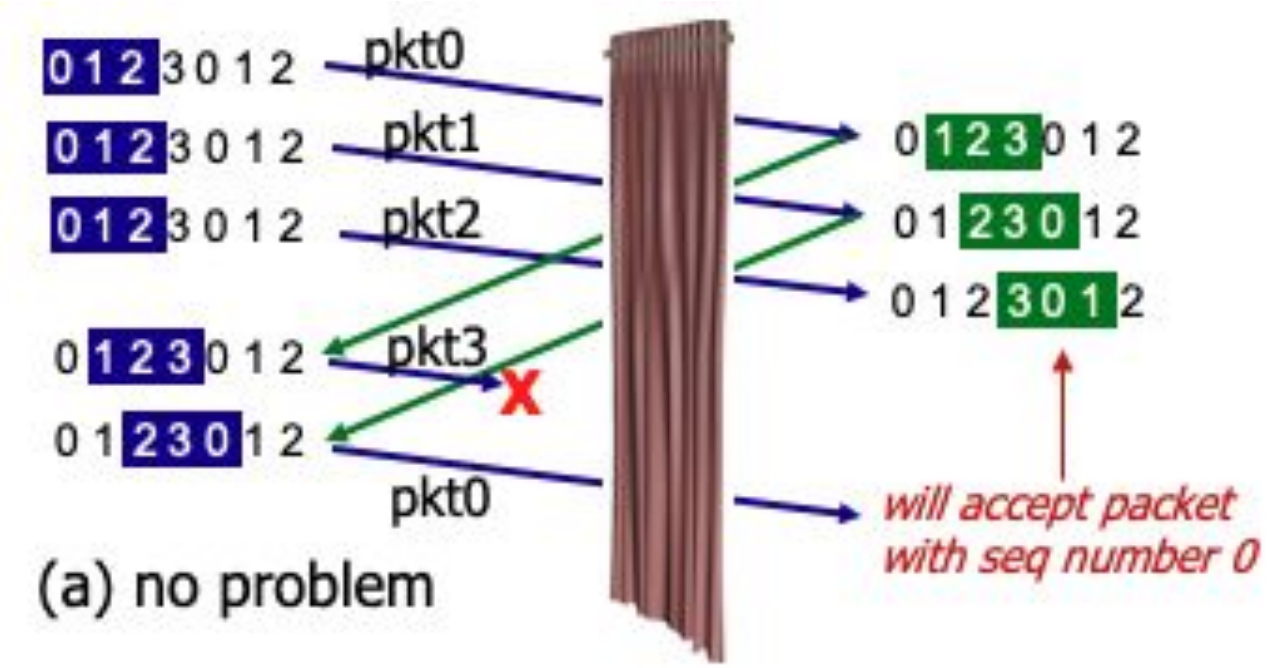
SR


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

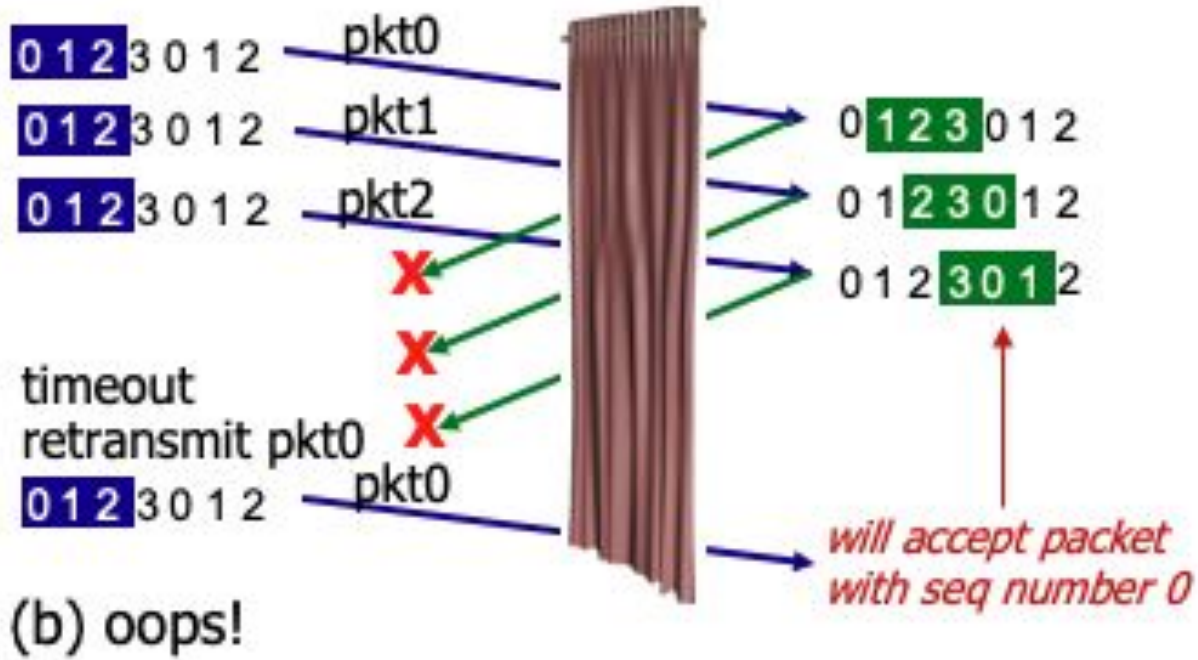
SR


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

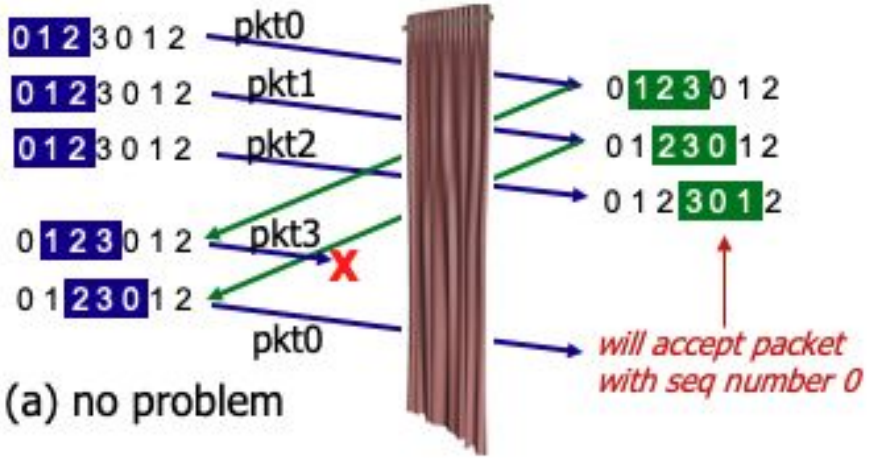
SR

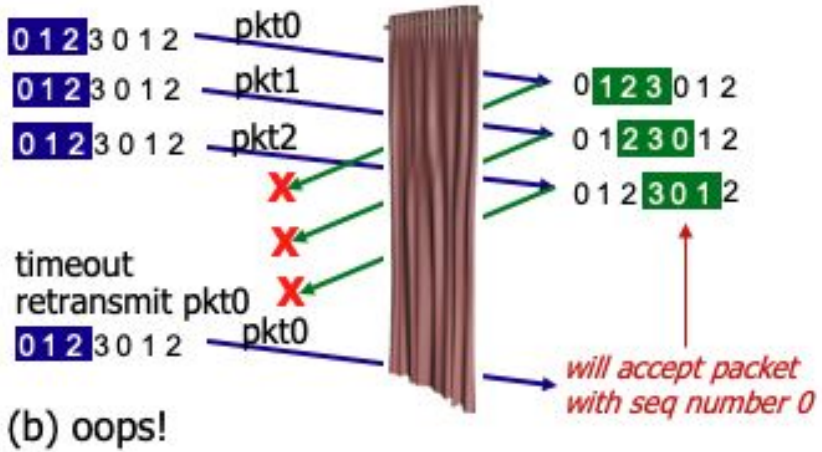

0 1 2 3 0 1 2
pkt0
0 1 2 3 0 1 2
pkt1
0 1 2 3 0 1 2
pkt2
0 1 2 3 0 1 2
pkt3
X
0 1 2 3 0 1 2
pkt0
(a) no problem
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
will accept packet
with seq number 0

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

Can receiver tell the difference?





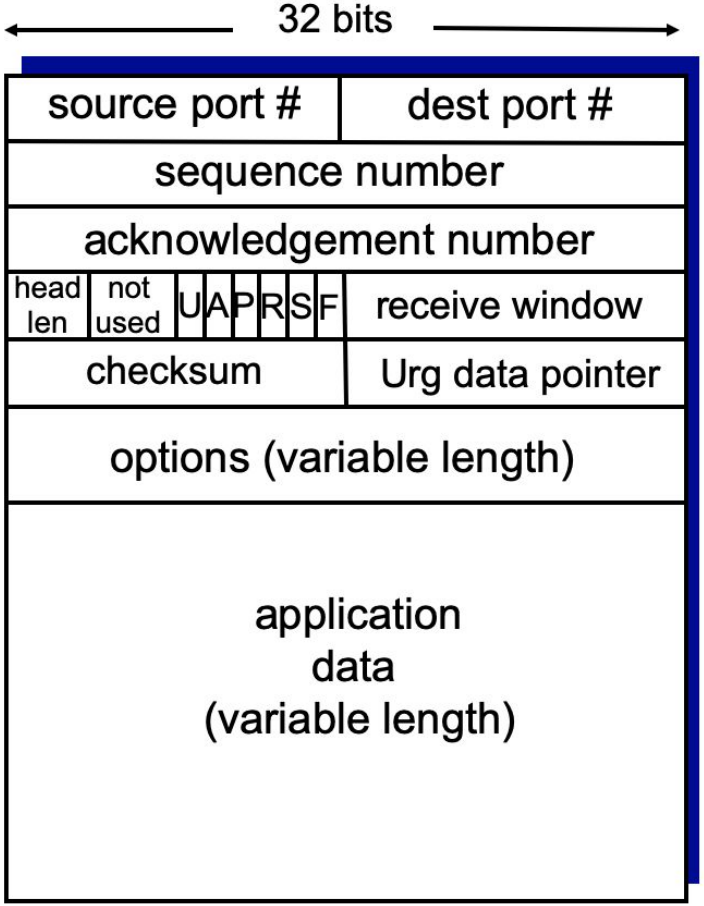
TCP

TCP overview

- Point to point (one sender, one receiver)
- Reliable, in-order, byte stream
  - No message boundaries
- Pipelined (like go-back-n and selective-repeat)
- Connection oriented
  - Handshake required before data exchange
- Flow control
- Congestion control

TCP segment structure

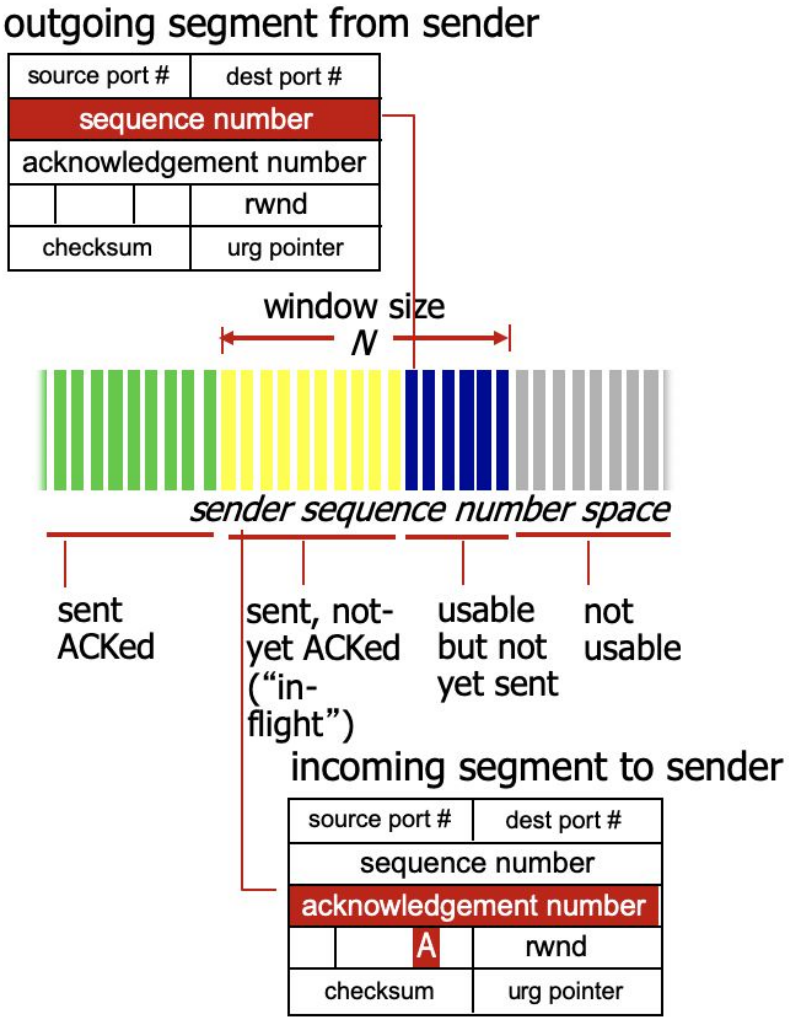
- You can see the familiar elements we designed in reliable data transfer protocol
  - Sequence number
  - ACK
  - Checksum
  - Window size



TCP: reliable data transfer

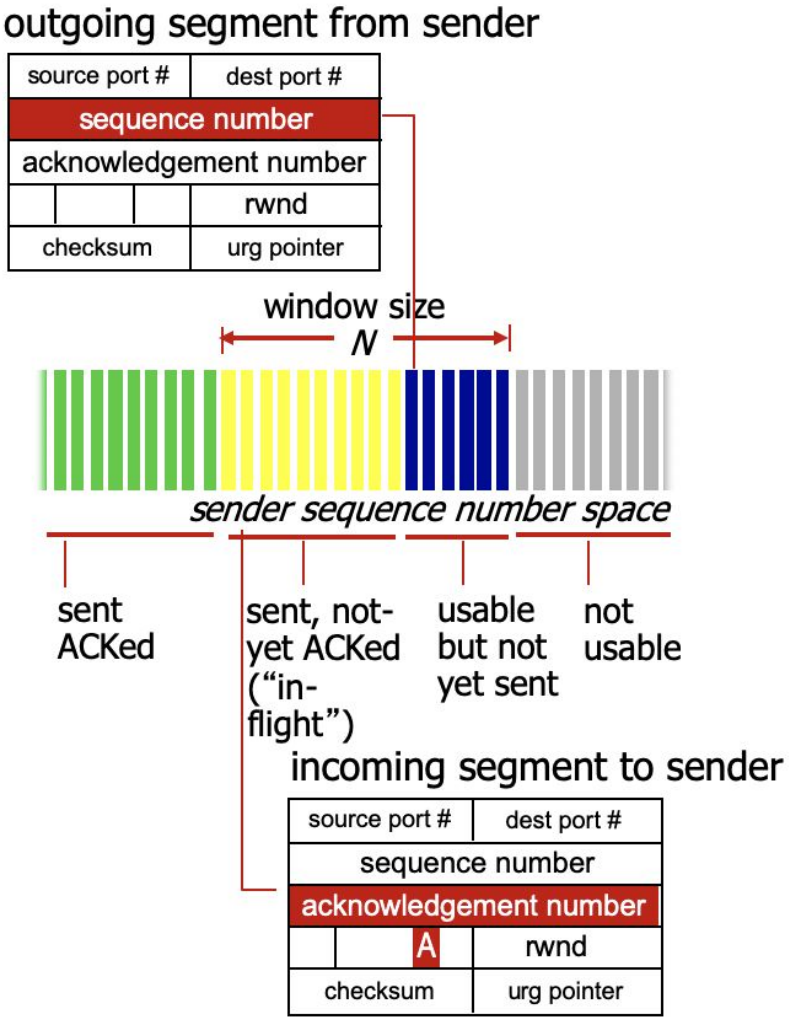
Sequence number

- Offset of the first byte in byte stream
- Is this the same as what we use in RDT?
- What's the benefit?

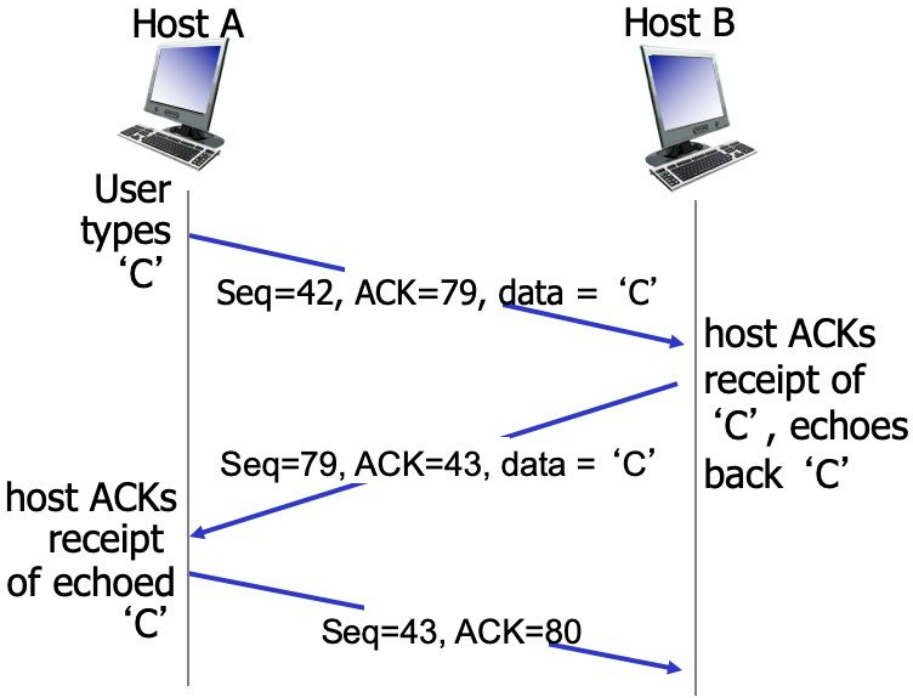


ACK

- Sequence number of next byte expected
- It is cumulative
- Why choose cumulative ACK? What's the benefit?



Sequence number and ACK example

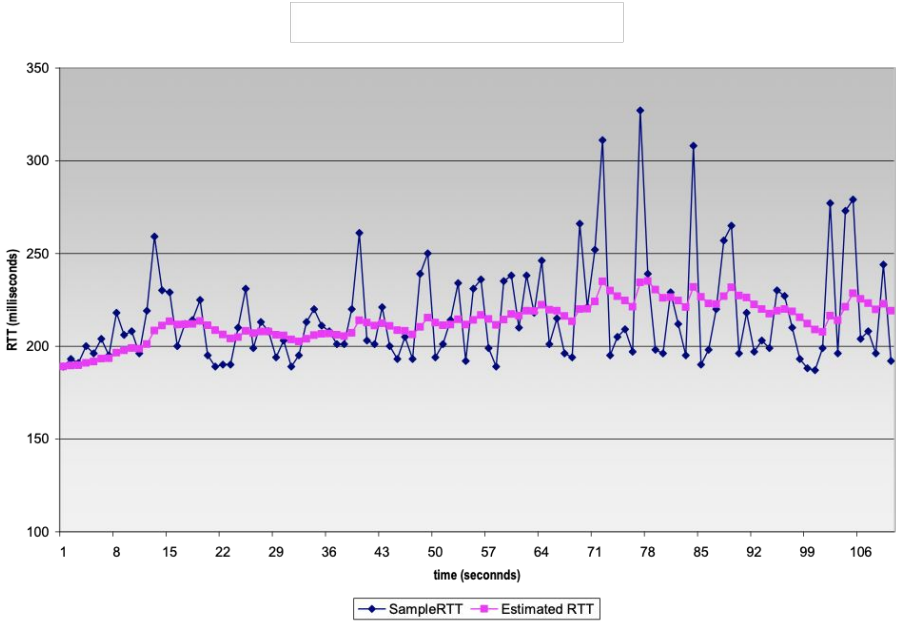


What about timer?

- We need timer to deal with possible packet loss in reliable data transfer
- How does TCP set timer?

RTT and timeout

- Timeout value should be longer than RTT
- But… RTT varies
- Too short: premature timeout, unnecessary retransmission
- Too long: slow reaction to data loss



Then how to estimate RTT?

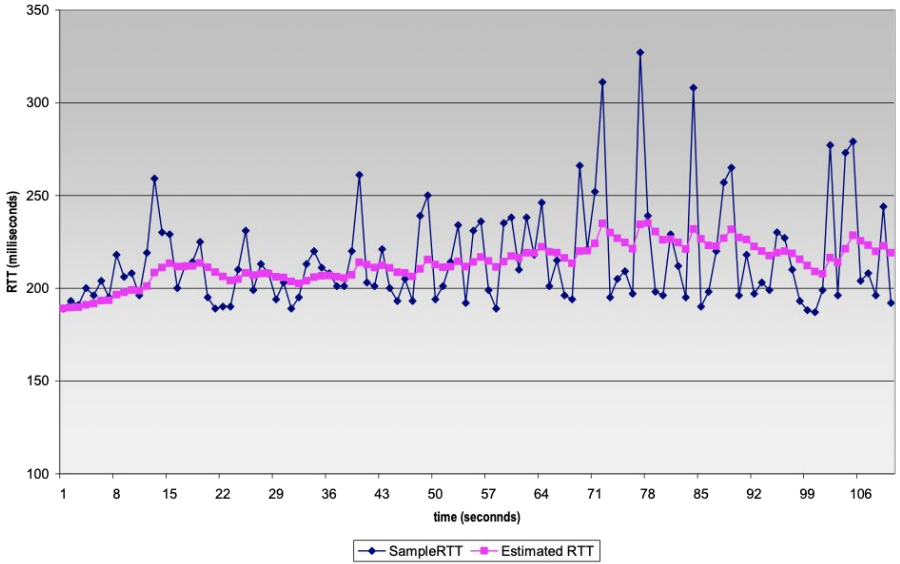
- Say you can collect sampled RTT over time
- Exponential weighted moving average
- Influence of past sample decrease exponentially fast
- Typical value $\alpha = 0.125$

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

Is EstimatedRTT enough to set timeout value?

RTT and timeout

- Only have the EstimatedRTT is not enough
- In order to set timeout you also need to add a “safety margin” to account for variance



Now estimate variance

- Use exponential weighted moving average
- Typical value $\beta = 0.25$

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

Now put everything together

- Combine EstimatedRTT and DevRTT to set timeout value

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

“safety margin”

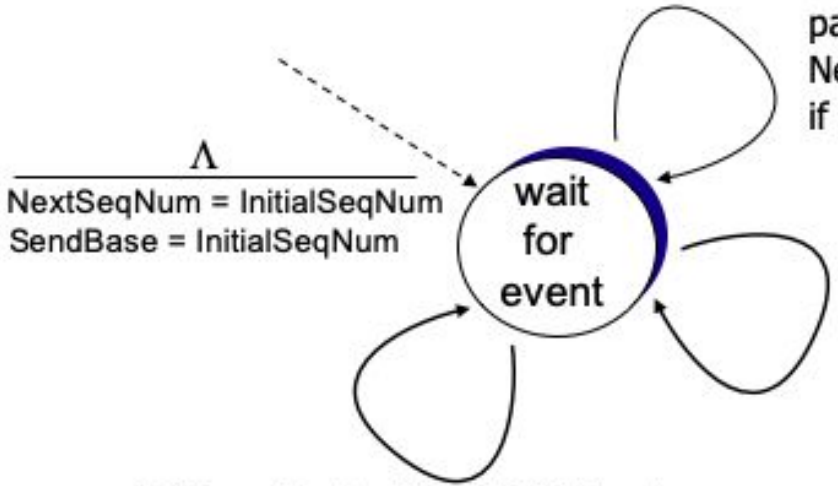
Quick recap so far

- Seq number is the offset of the first byte of the data
- ACK is the next sequence number it is expecting
  - It is cumulative
- How to set timeout value using EstimatedRTT and DevRTT

Reliable data transfer

- TCP creates rdt service on top of IP’s best effort service
  - pipelined segments
  - cumulative ACKs
  - single retransmission timer
- Retransmission triggered by
  - timeout events

Sender



data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

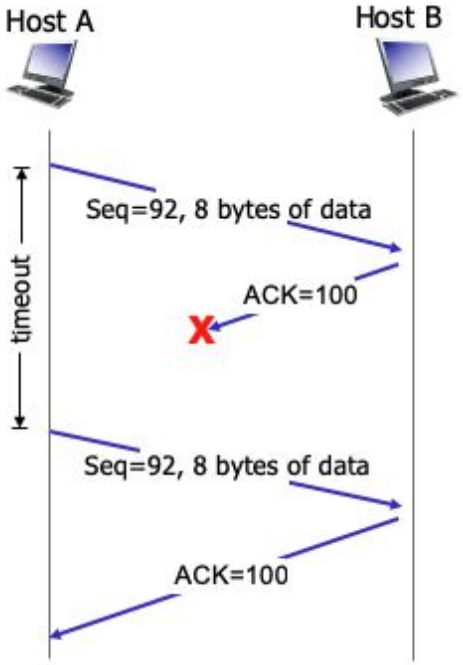
timeout

retransmit not-yet-acked segment
        with smallest seq. #
start timer

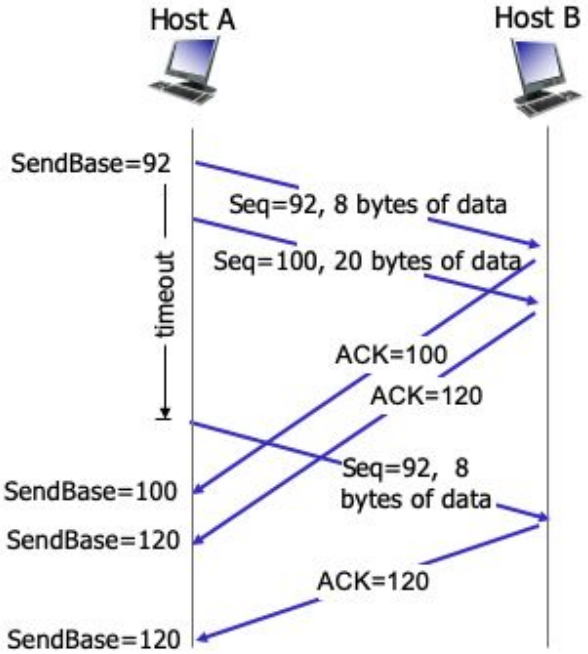
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
      else stop timer
    }
```

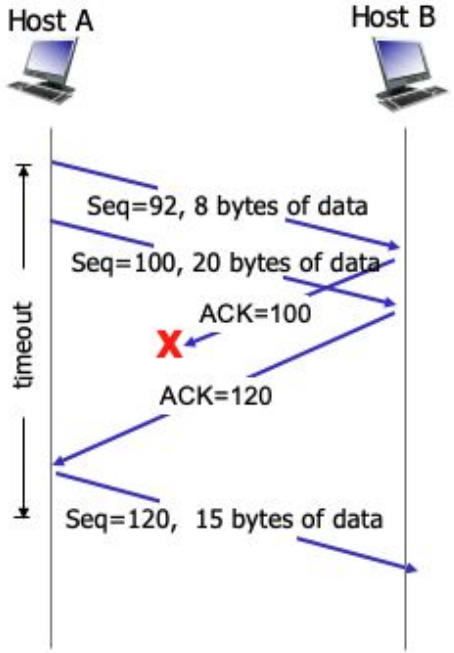
Examples



lost ACK scenario



premature timeout



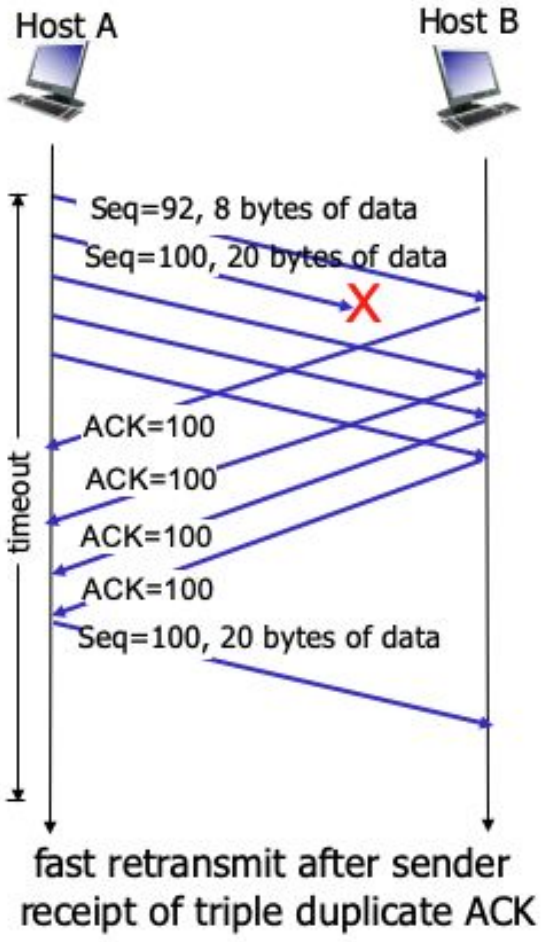
cumulative ACK

Fast retransmit

- Timeout period is often relatively long
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
  - How many duplicate ACKs to resend?

Fast retransmit

- If sender receives three duplicate ACKs, it resends unACKed segment with smallest sequence number

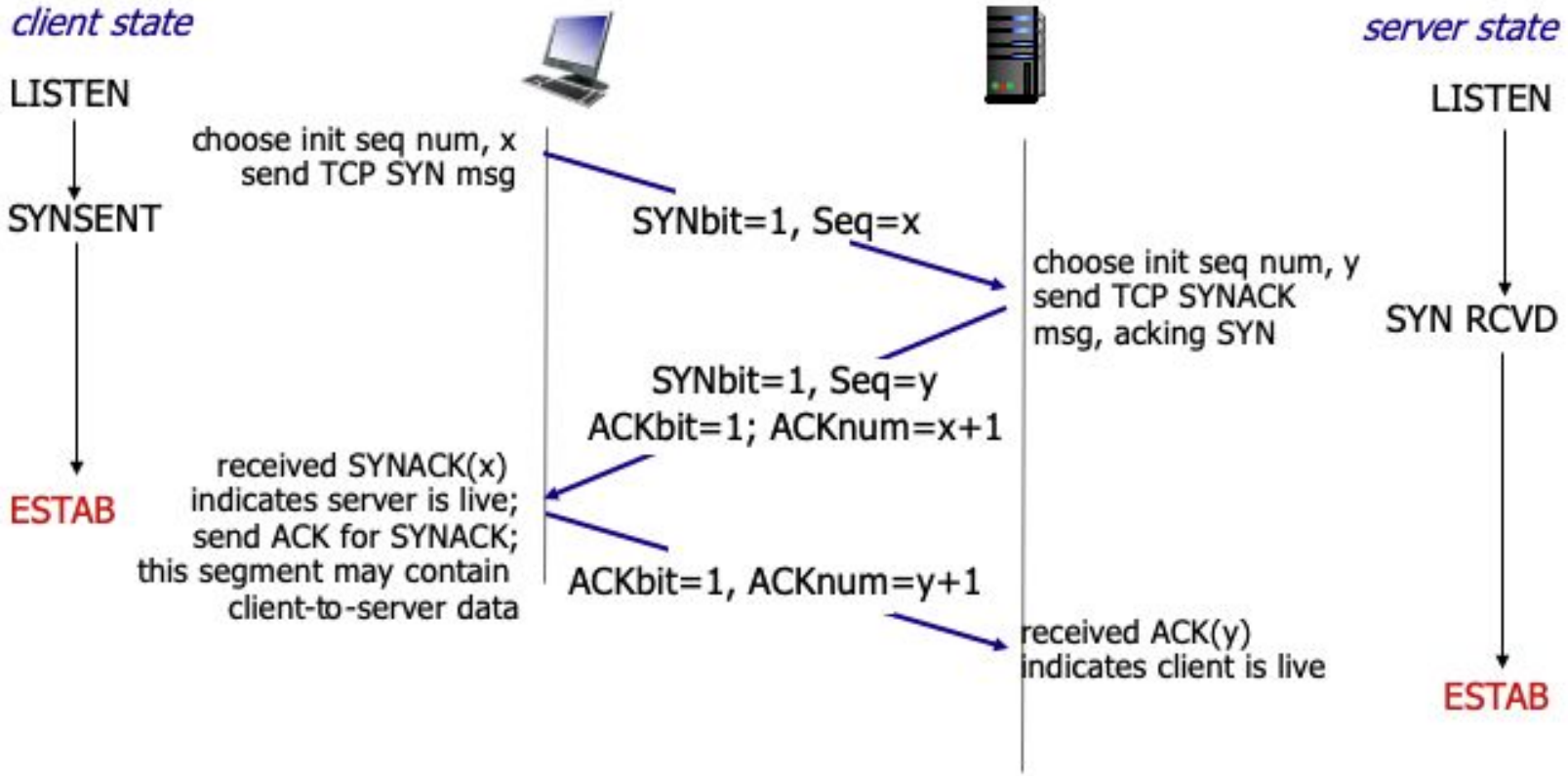


TCP: connection management

Setup connection

- Before exchanging data, sender/receiver “handshake”
  - Agree to establish connection
  - Agree on connection parameters

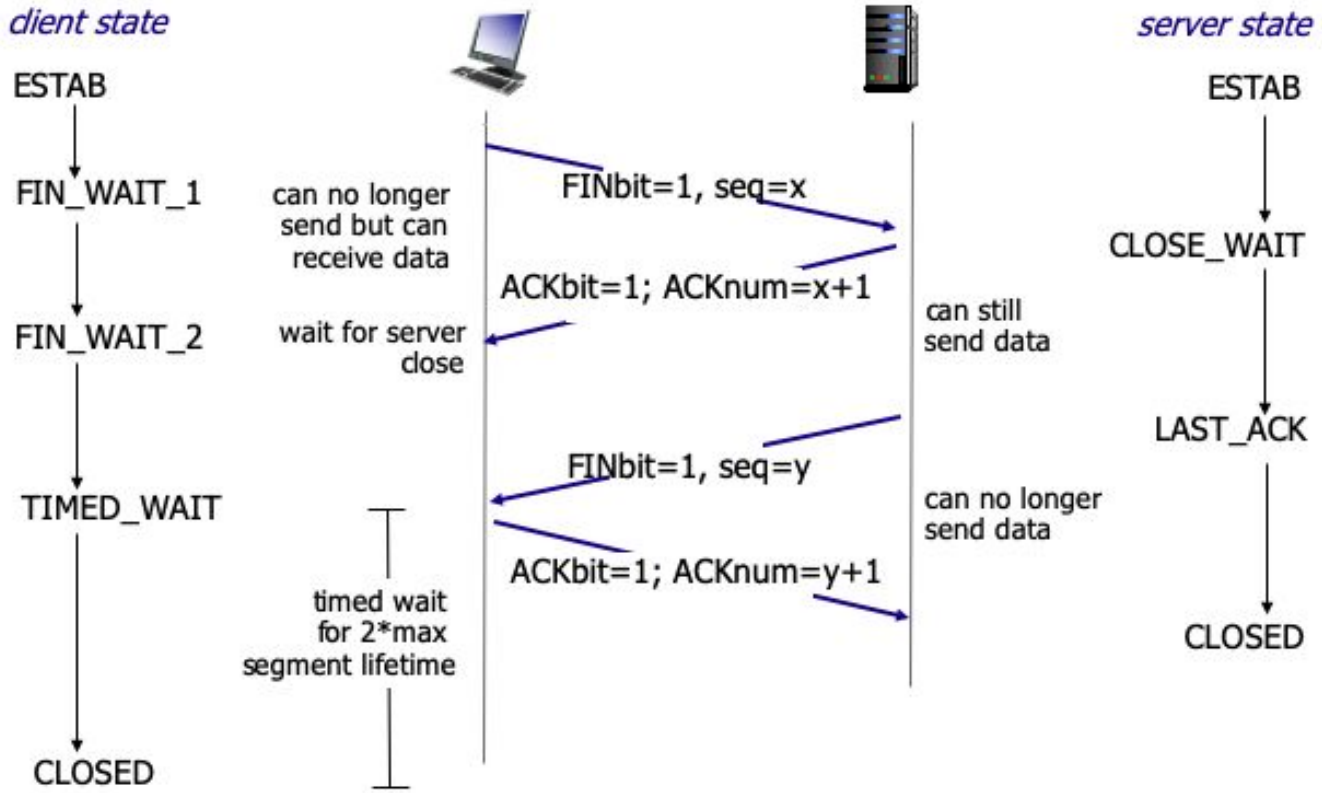
3-way handshake



Close connection

- Client and server each close their side of connection
  - send TCP segment with FIN bit set
- Respond to received FIN with ACK
  - on receiving FIN, the ACK can be combined with its own FIN

Close connection



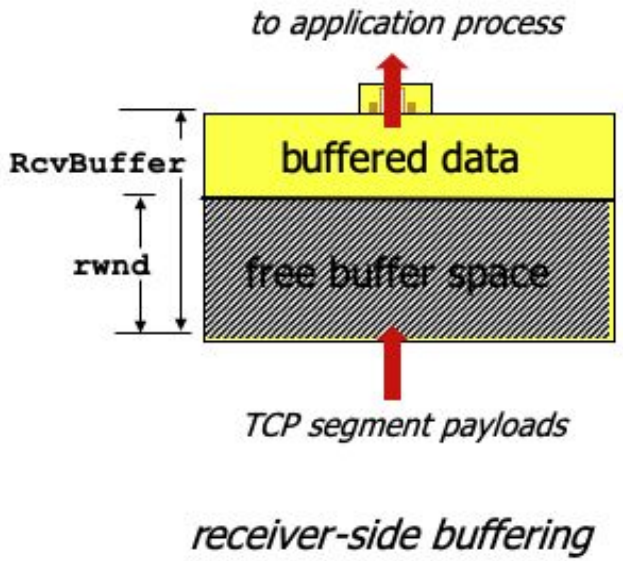
TCP: flow control

Flow control

- Avoid sending data too quickly

Flow control

- Receiver “advertises” free buffer space by including ***rwnd*** value in TCP header
- Sender limits amount of un-ACKed (in-flight) data to receiver’s ***rwnd*** value
- Guarantees receiver buffer will not overflow



TCP: congestion control

Congestion control

- Too many sources sending too much data too fast for network to handle
- Different from flow control
- Manifestations
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)

How to detect congestion?

Congestion signs

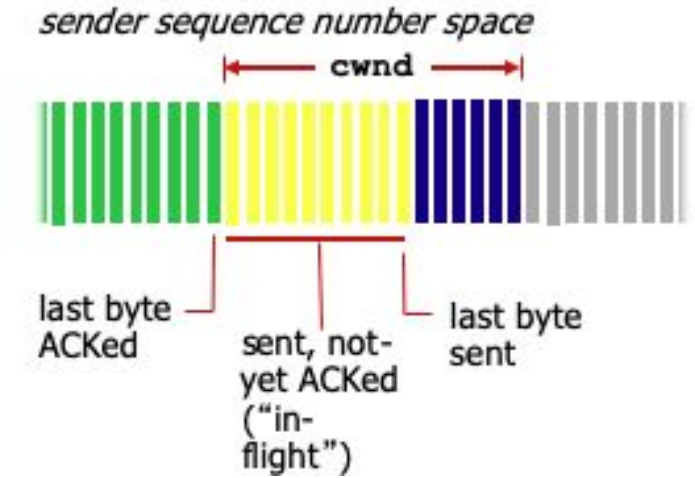
- When you lose a packet, it is a sign of congestion
- How does TCP detection packet loss?
  - Timeout
  - Duplicate ACKs

How to control data rate?

cwnd

- **cwnd** controls window size
- Dynamic function of perceived network congestion
- Data rate: roughly send cwnd bytes, wait RTT for ACKs, then send more bytes

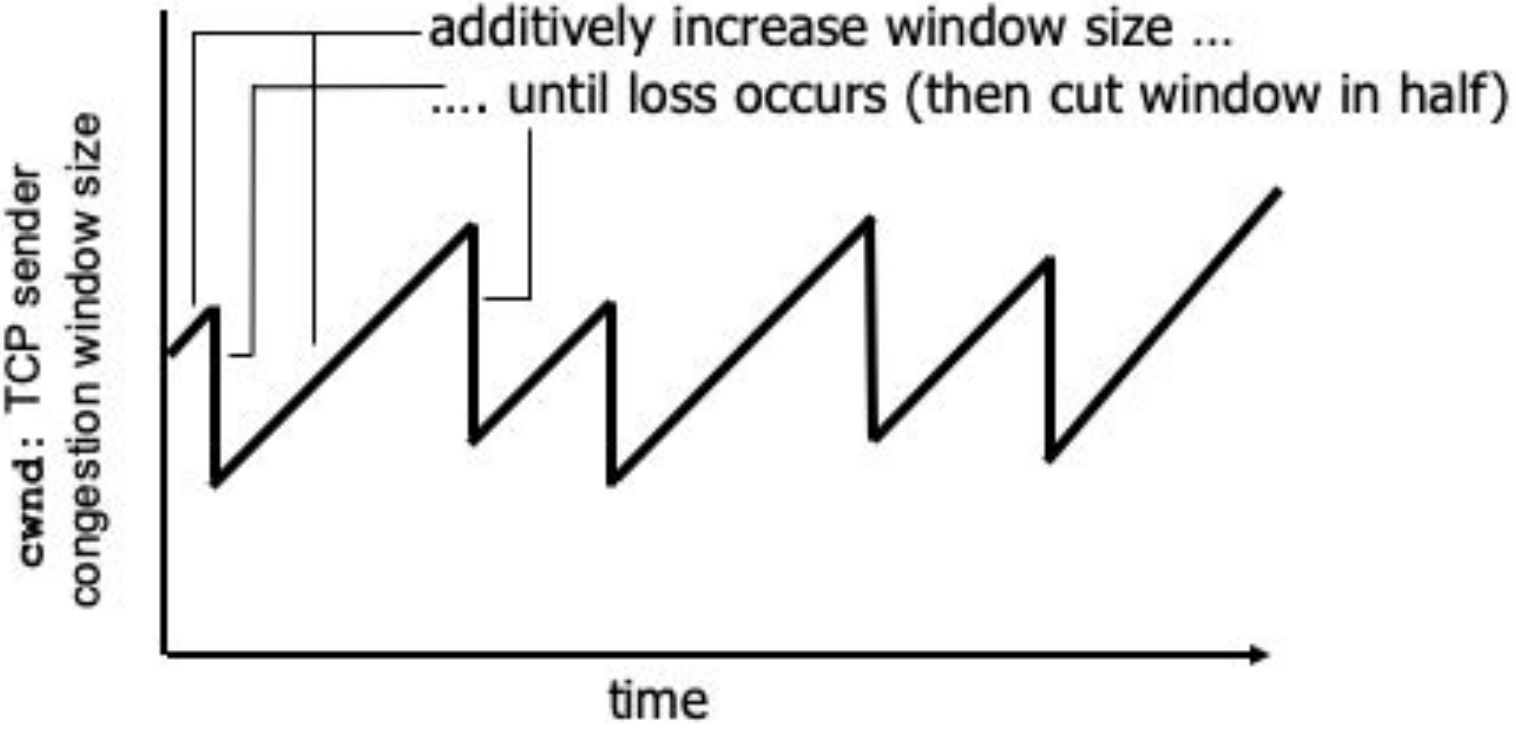
$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$



TCP's approach for congestion control

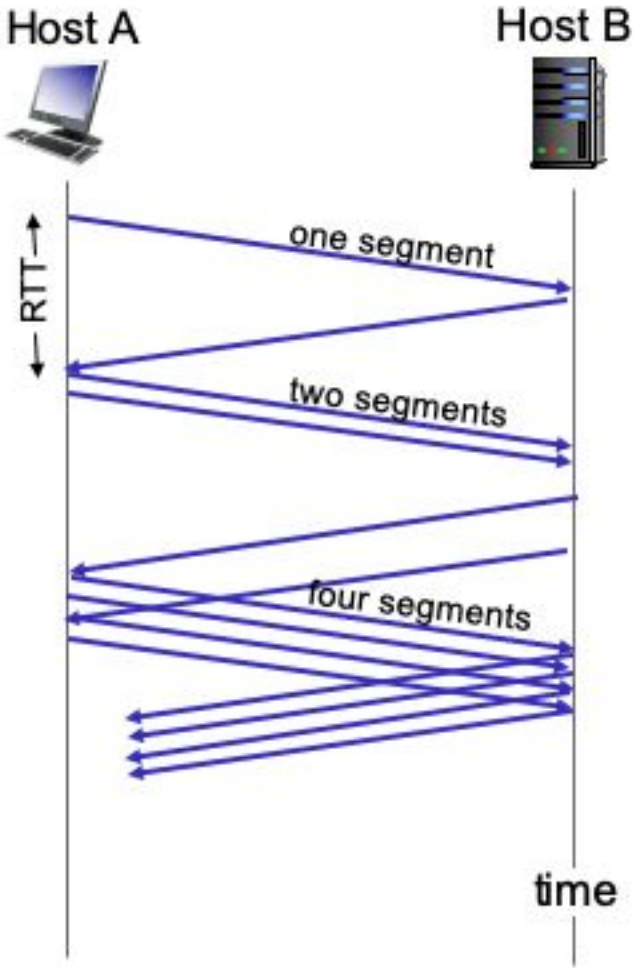
- Sender increase transmission rate (window size), probing for usable bandwidth, until loss occurs
- Additive increase: increase ***cwnd*** by 1 MSS every RTT until loss detected
  - MSS: maximum segment size
- Multiplicative decrease: cut ***cwnd*** by half after loss

Impact on cwnd



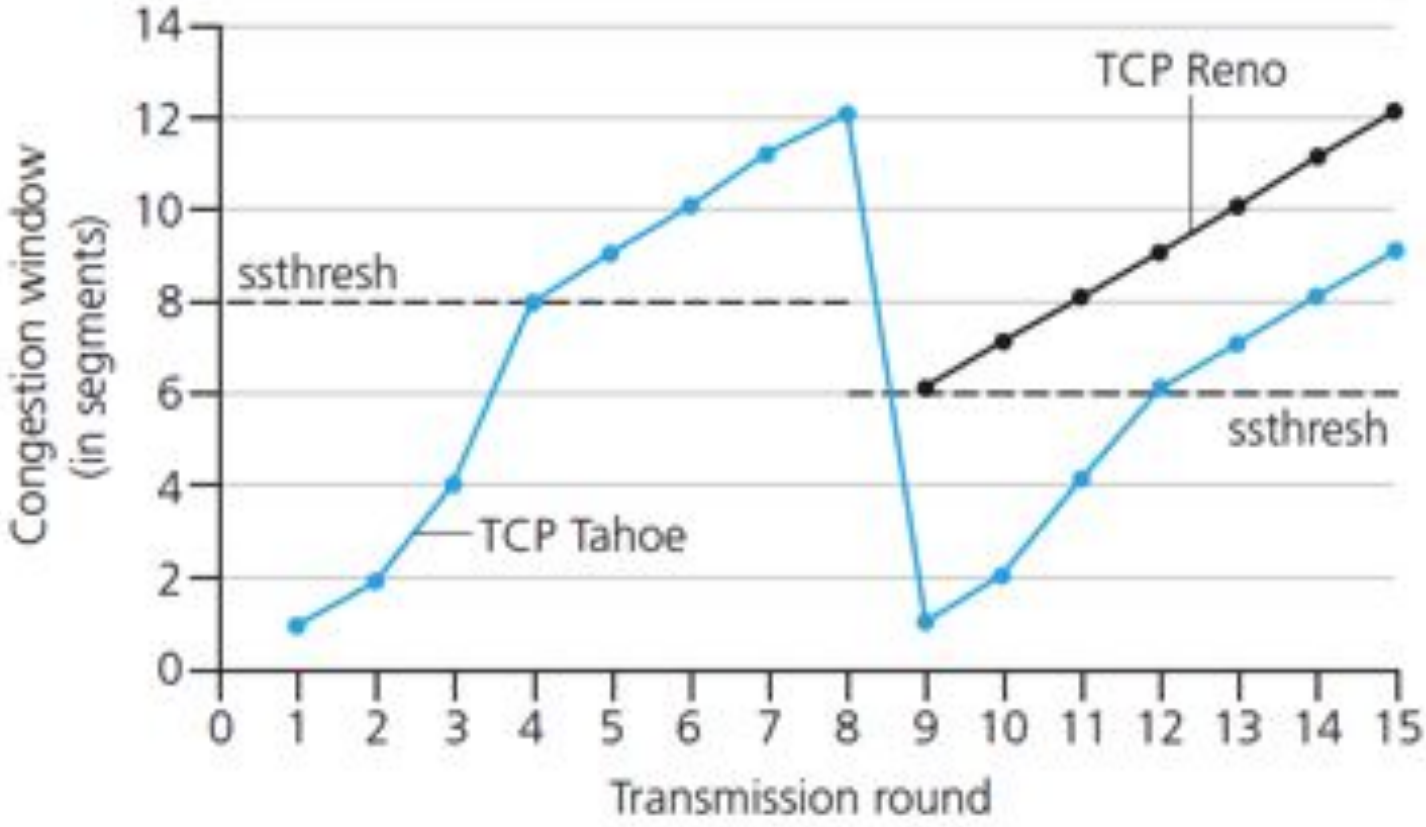
“Slow” start

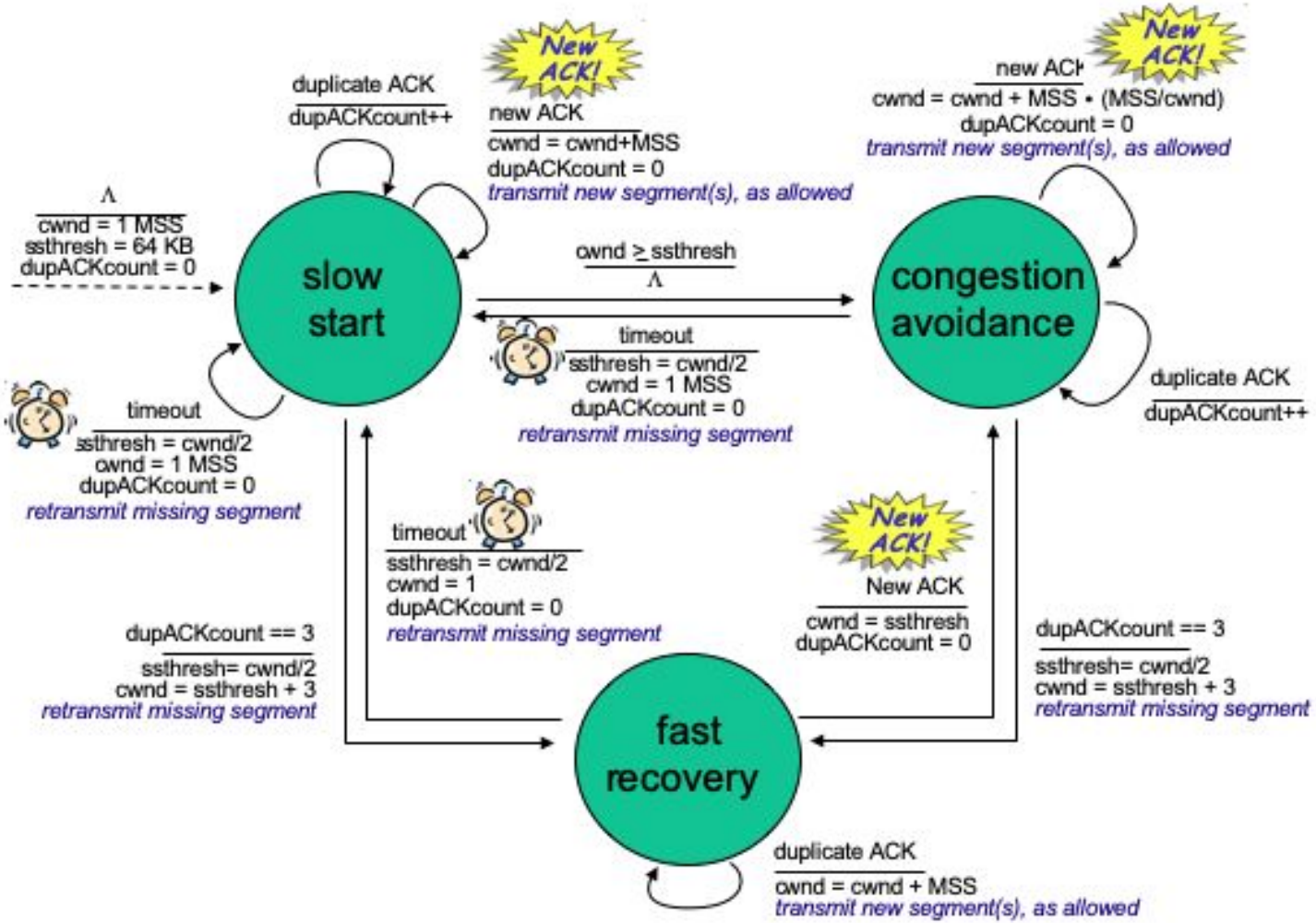
- When connection begins, increase rate exponentially till first loss
  - initial cwnd = 1 MSS
  - double cwnd every RTT
  - done by increasing cwnd by 1 MSS for every ACK
- Initial rate is slow, but ramps up exponentially fast



Reaction to loss

- Loss indicated by timeout
  - cwnd set to 1 MSS
  - cwnd then grows exponentially (as in slow start) to threshold, then grows linearly
- Loss indicated by 3 duplicate ACKs
  - Op 1: cwnd is cut in half then grows linearly [Reno]
  - Op 2: same as timeout [Tahoe]


Congestion window (in segments)
14
12
10
8
6
4
2
0
ssthresh
TCP Reno
ssthresh
TCP Tahoe
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Transmission round


slow start
congestion avoidance
fast recovery
duplicate ACK
dupACKcount++
New ACK!
new ACK
cwnd = cwnd+MSS
dupACKcount = 0
transmit new segment(s), as allowed
Λ
cwnd = 1 MSS
ssthresh = 64 KB
dupACKcount = 0
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment
cwnd ≥ ssthresh
Λ
new ACK
cwnd = cwnd + MSS • (MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed
duplicate ACK
dupACKcount++
timeout
ssthresh = cwnd/2
cwnd = 1
dupACKcount = 0
retransmit missing segment
New ACK
cwnd = ssthresh
dupACKcount = 0
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment
duplicate ACK
cwnd = cwnd + MSS
transmit new segment(s), as allowed

Questions?

Demo UDP socket

Backups

2-way handshake failure scenarios