Transport Layer

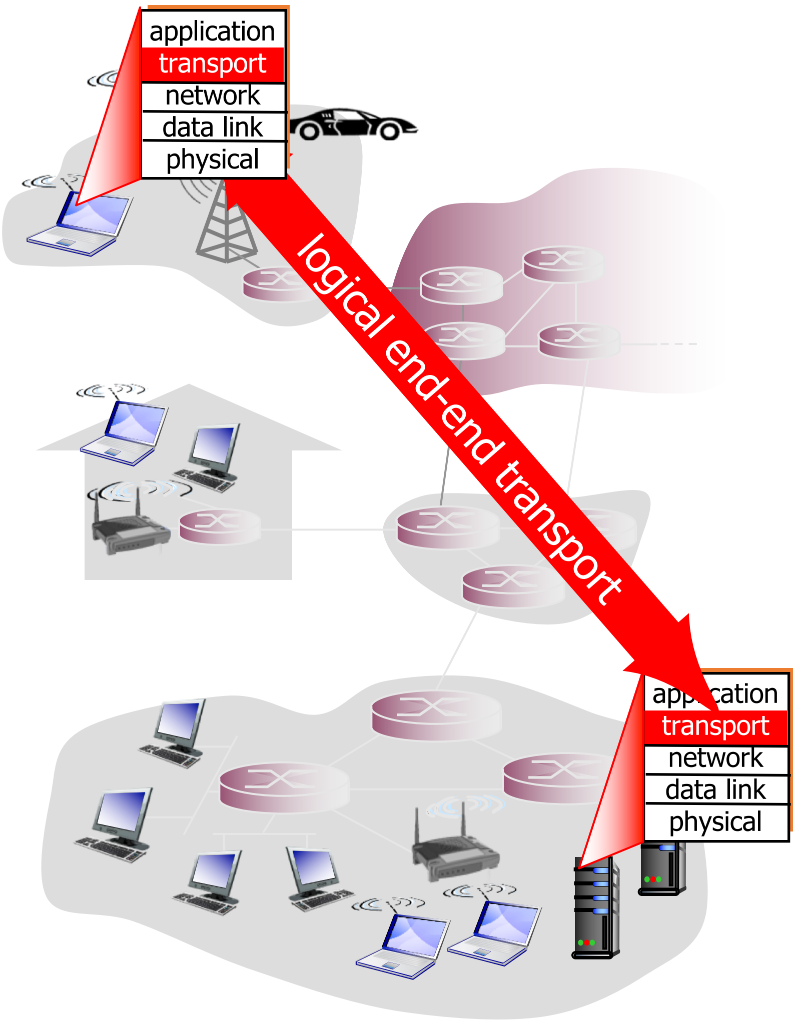
CMPS 4750/6750: Computer Networks

Outline

- Overview of transport-layer services
- Connectionless Transport: UDP
- Principles of reliable data transfer
- Connection-Oriented Transport: TCP
- TCP congestion control
- Network utility maximization

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

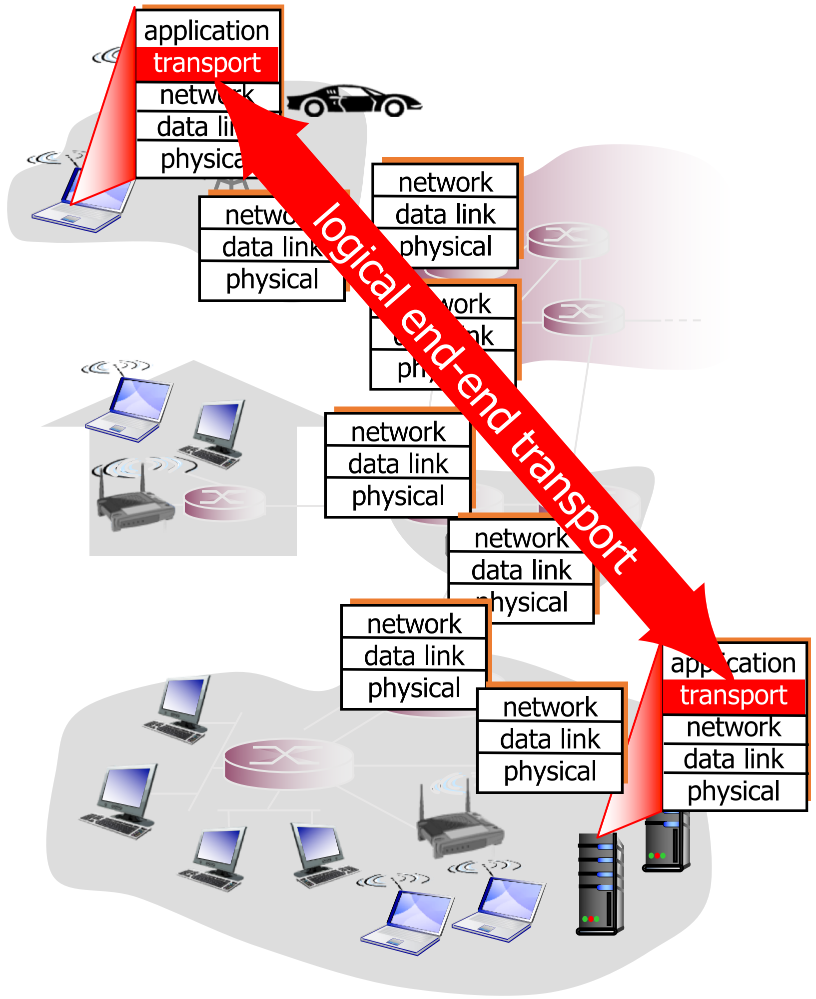


Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

Internet transport-layer protocols

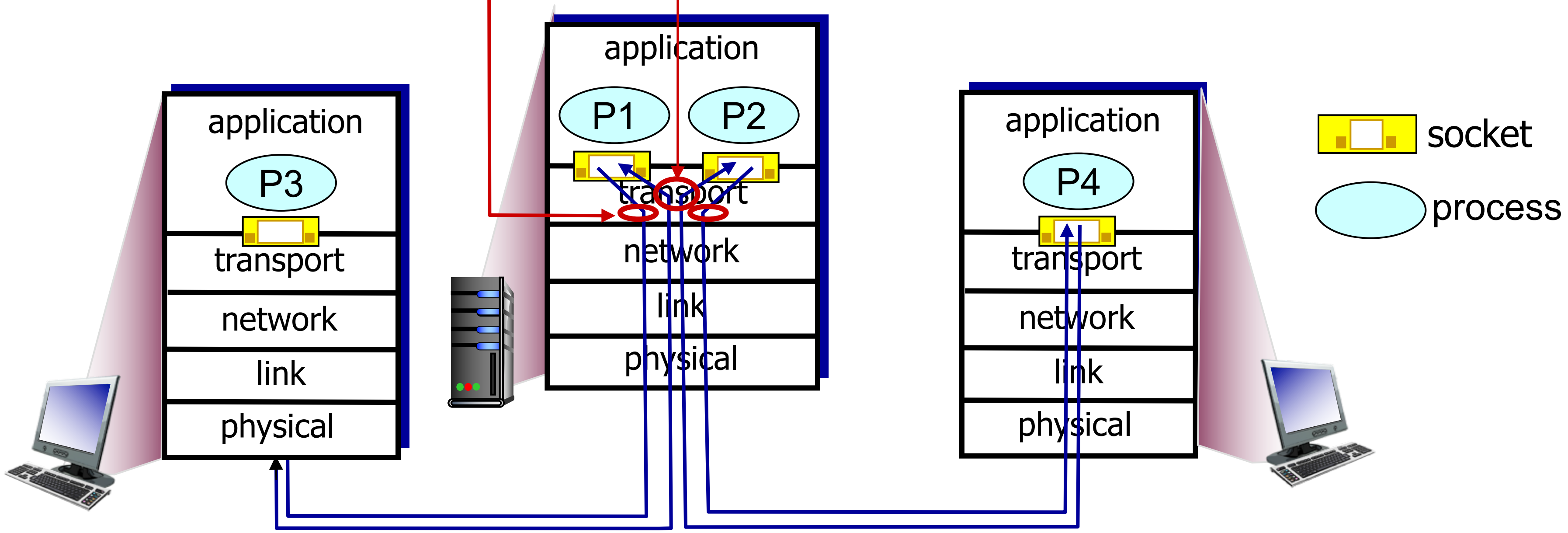
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- reliable, in-order delivery (TCP)
  - connection setup
  - flow control
  - congestion control
- services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing/demultiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

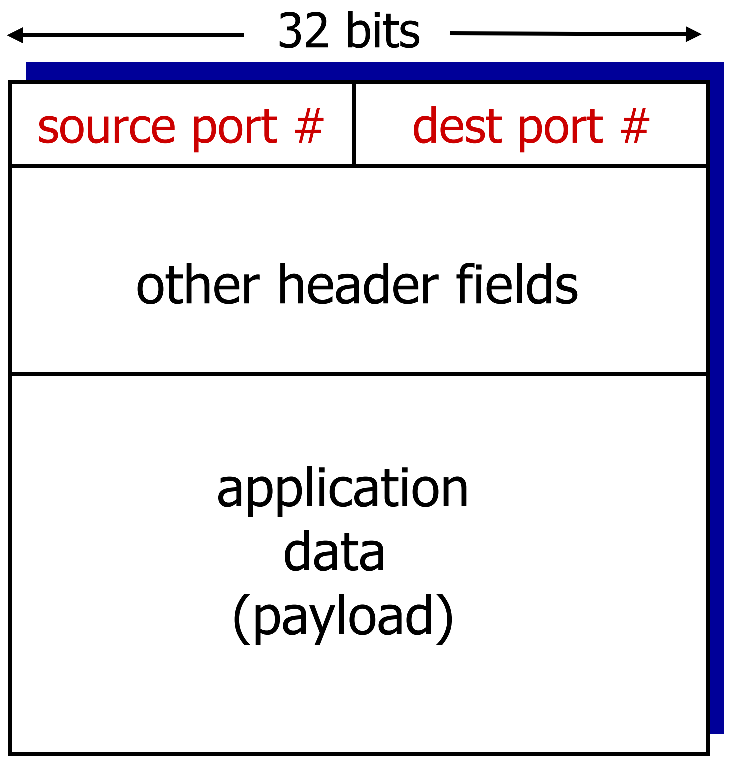
demultiplexing at receiver:
use header info to deliver received segments to correct socket



How demultiplexing works

Port number: 0 – 65535
Well-known port number: 0 - 1023

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

```
clientSocket =socket(AF_INET,SOCK_DGRAM)
clientSocket.bind(('', 19157));
```

- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

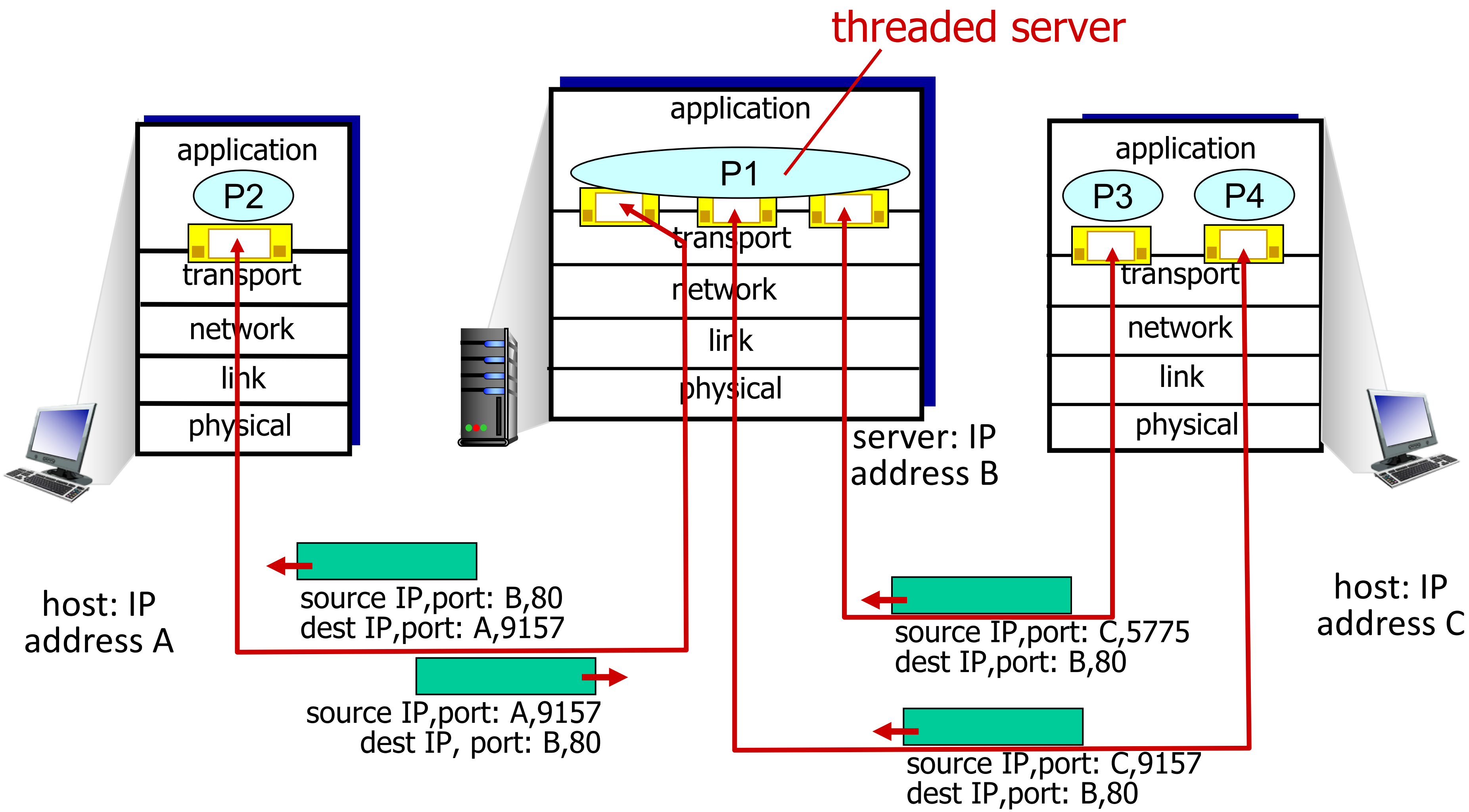
→ IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connection-oriented demux

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,12000))
serverSocket.listen(1)
while True:
    connectionSocket, addr = serverSocket.accept()
    ...
```

- TCP sockets waiting for connections identified by IP address and port number
- Other TCP sockets identified by 4-tuple:
  - source IP address, source port number
  - dest IP address, dest port number

Connection-oriented demux: example



Outline

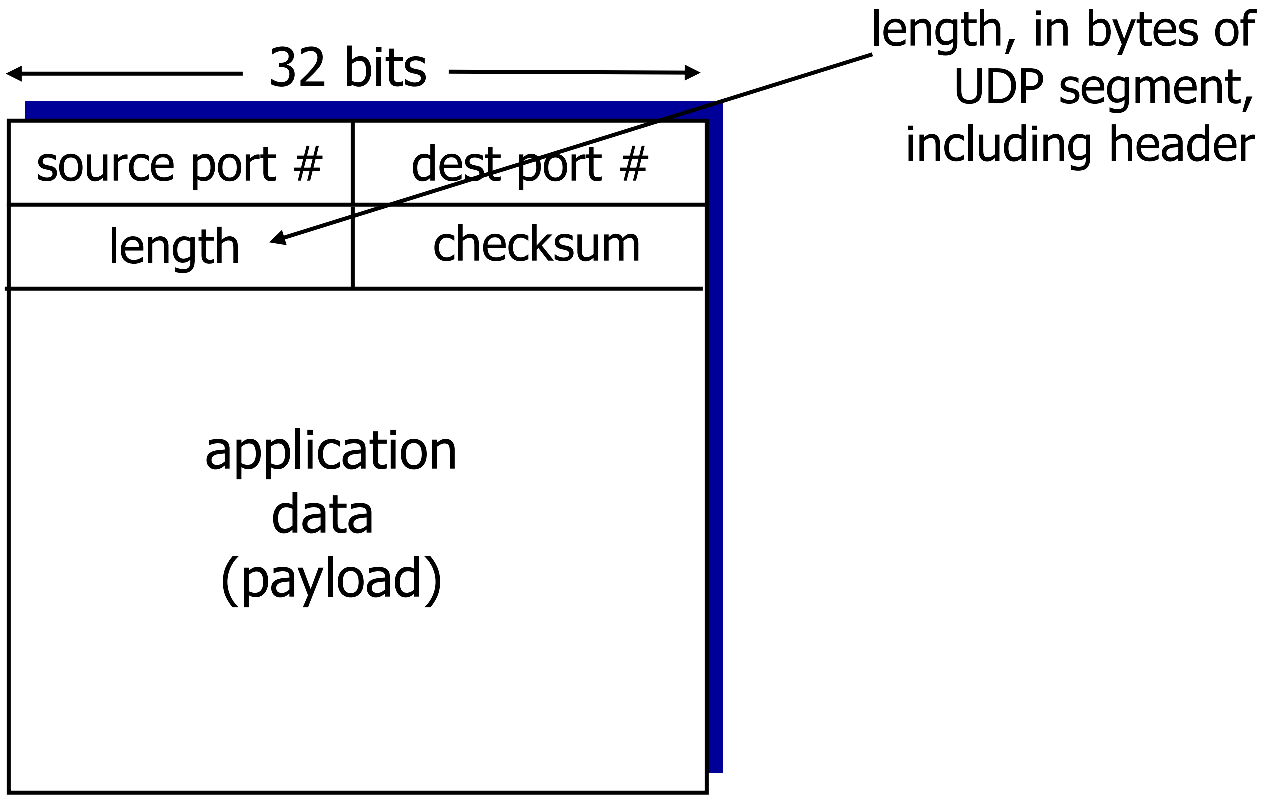
- Overview of transport-layer services
- Connectionless Transport: UDP
- Principles of reliable data transfer
- Connection-Oriented Transport: TCP
- TCP congestion control

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- why is there a UDP
  - no connection establishment (which can add delay)
  - simple: no connection state at sender, receiver
  - small header size
  - no congestion control: UDP can blast away as fast as desired
  - application-specific error recovery
- UDP use:
  - streaming multimedia apps, DNS, SNMP

UDP: segment header



UDP segment format

UDP checksum

- *Goal:* detect "errors" (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors*

Internet checksum: example

example: add two 16-bit integers

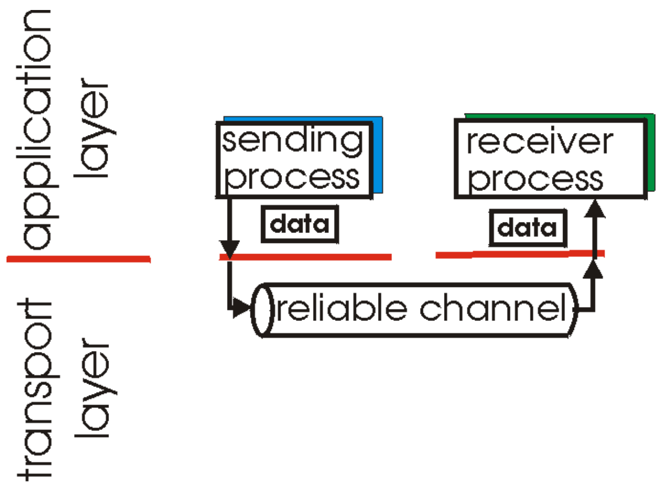
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Outline

- Overview of transport-layer services
- Connectionless Transport: UDP
- Principles of reliable data transfer
- Connection-Oriented Transport: TCP
- TCP congestion control

Principles of reliable data transfer

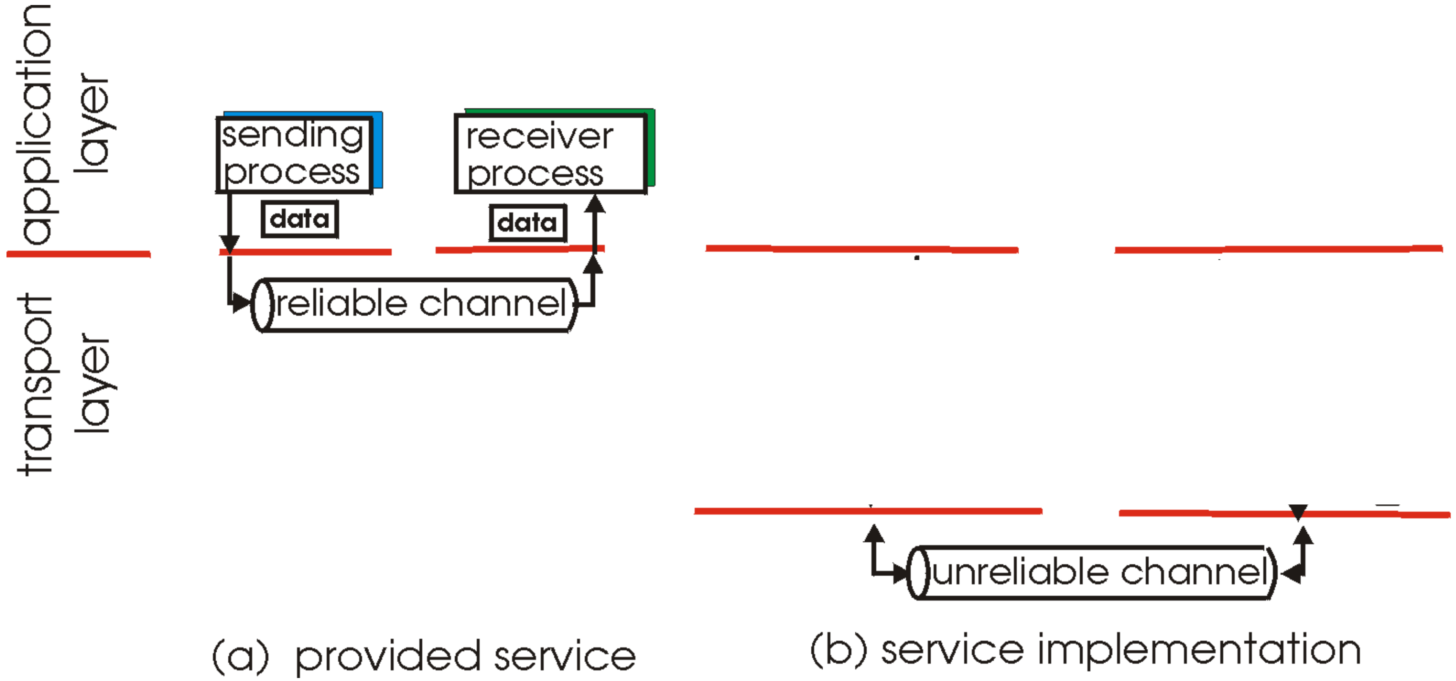
- important in application, transport, link layers
  - top-10 list of important networking topics!



(a) provided service

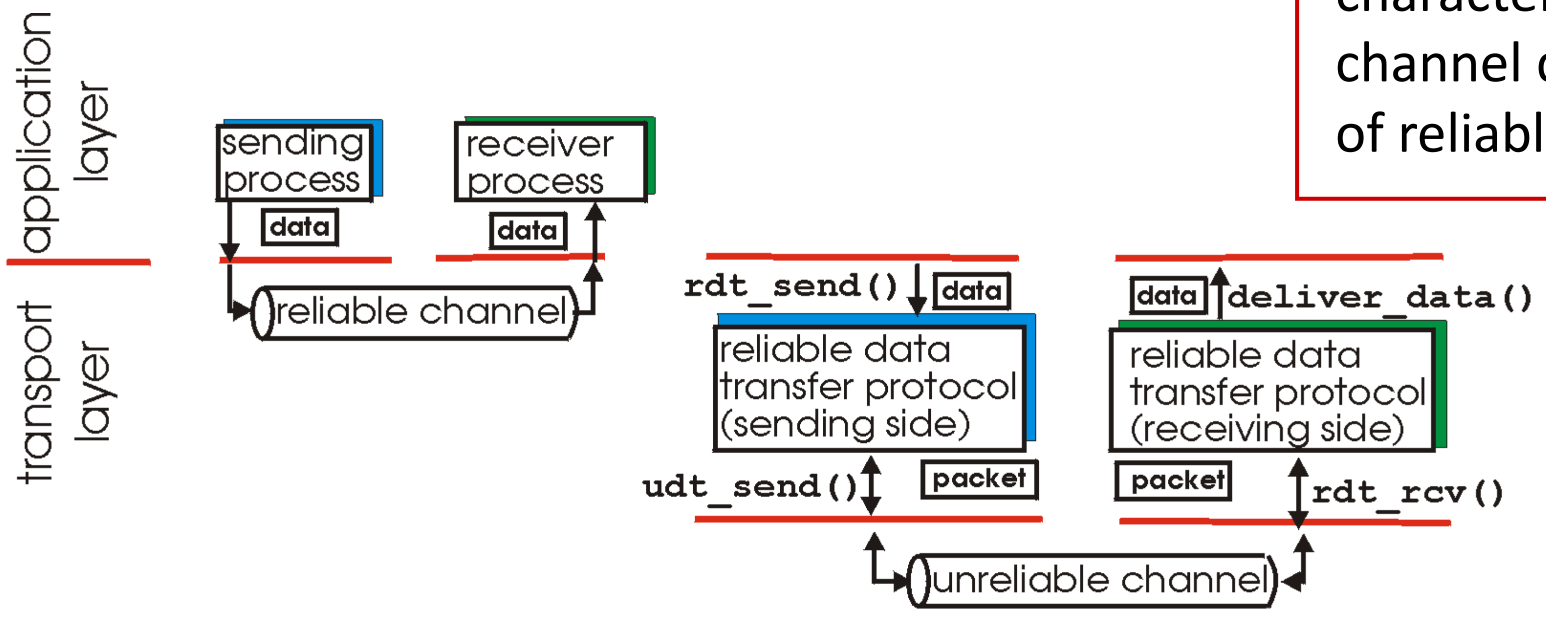
Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!



Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!



(a) provided service

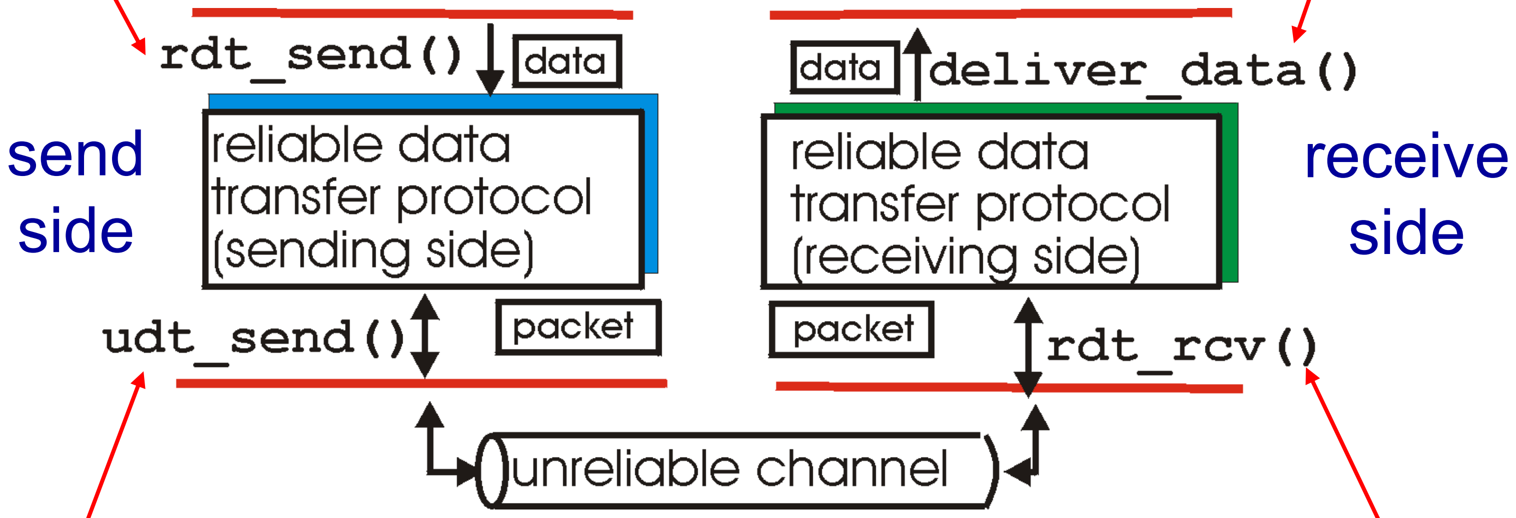
(b) service implementation

characteristics of unreliable channel determine complexity of reliable data transfer protocol

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver

deliver_data(): called by **rdt** to deliver data to upper



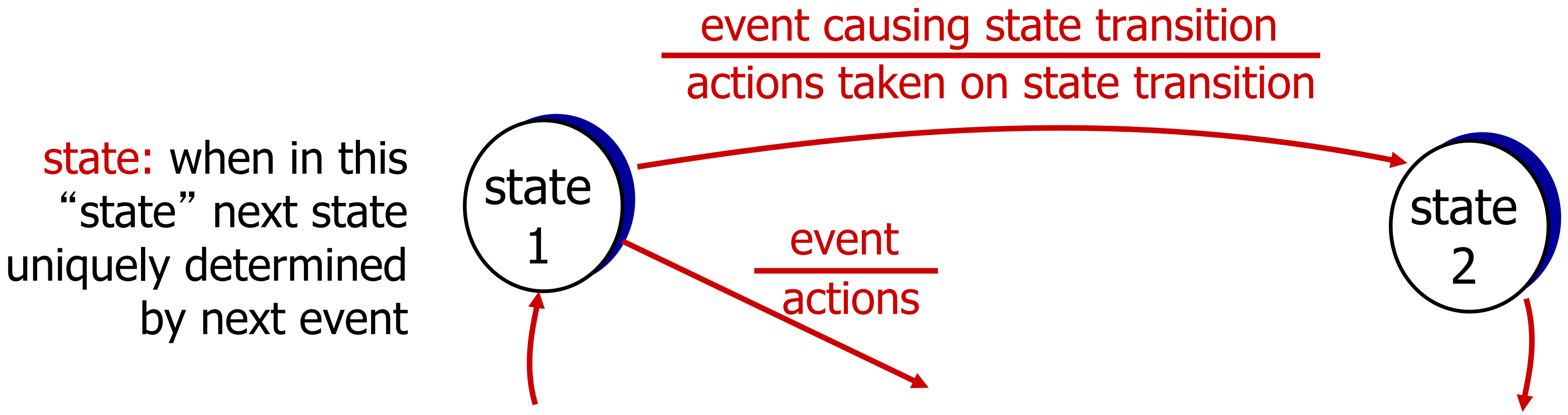
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

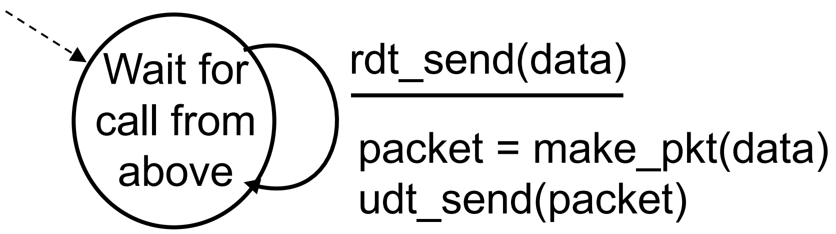
we' ll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

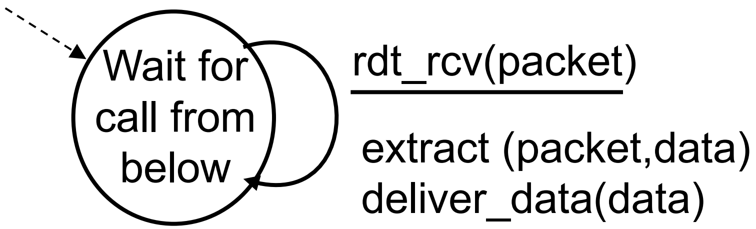


rdt1.0: reliable transfer over a reliable channel

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



sender



receiver

Potential Channel Errors

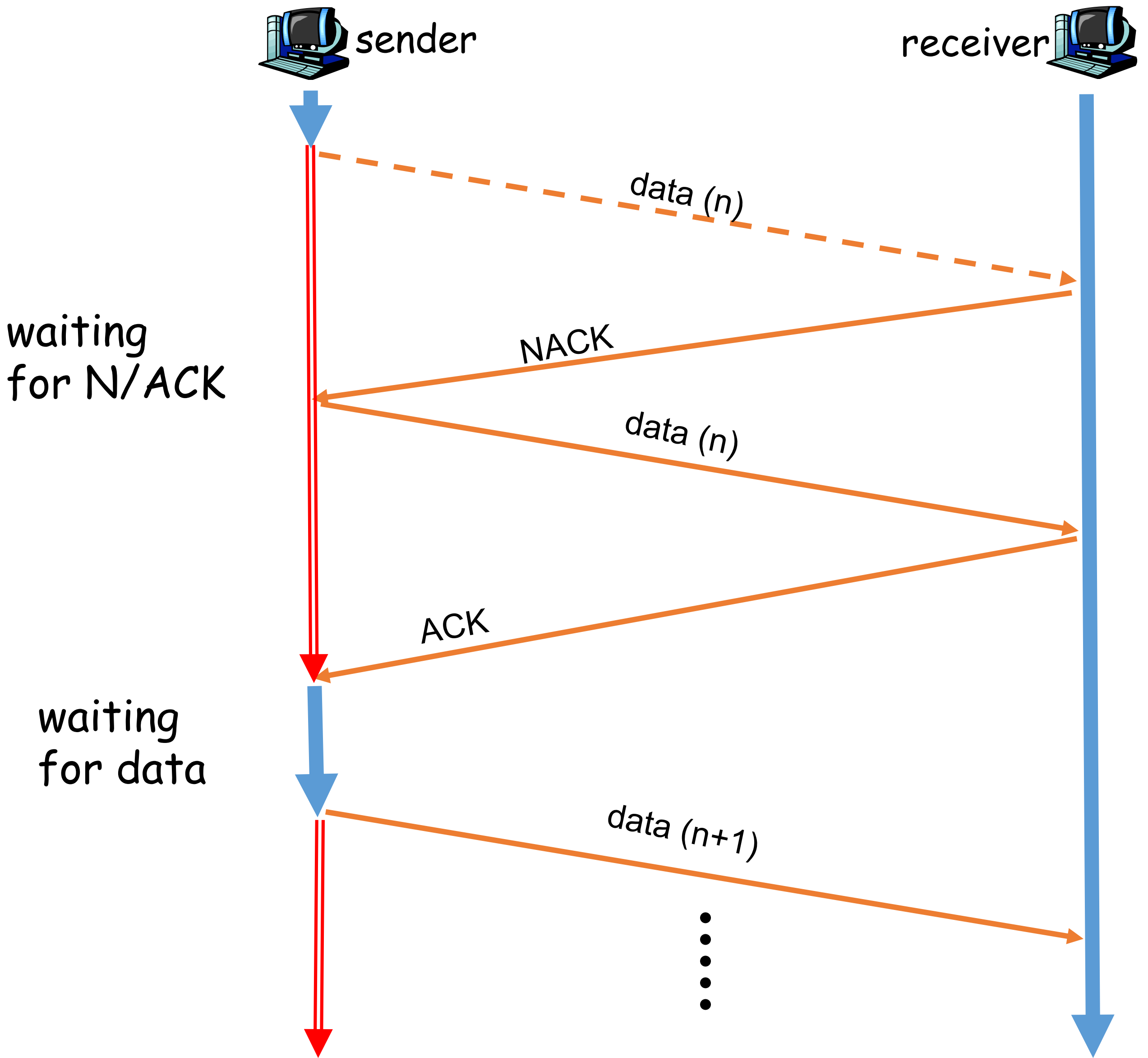
- bit errors
- loss (drop) of packets
- reordering or duplication

➢ characteristics of unreliable channel determine complexity of reliable data transfer protocol

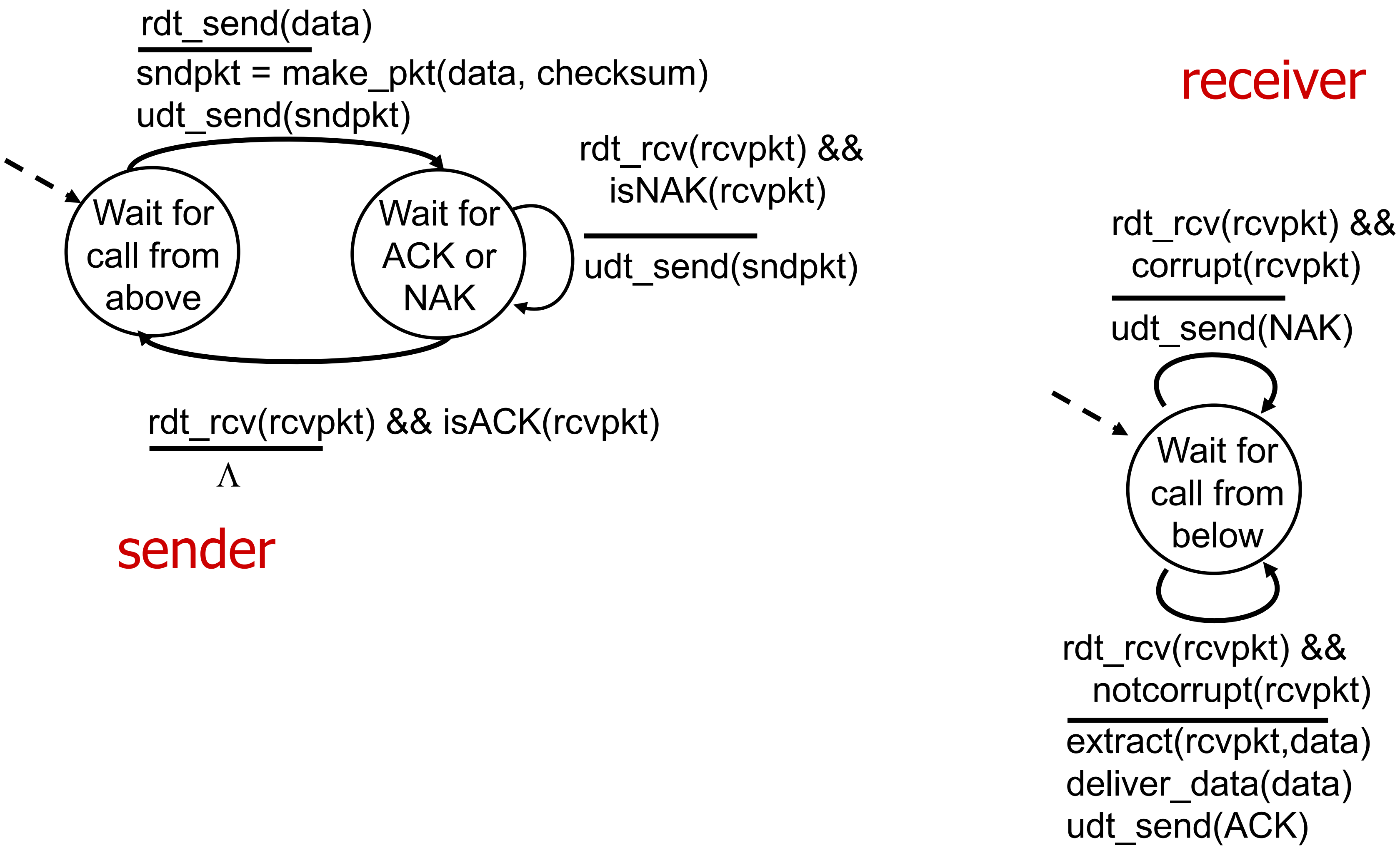
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - no loss of packets, no reordering of packets
- checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
  - Known as ARQ (Automatic Repeat reQuest) protocols
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

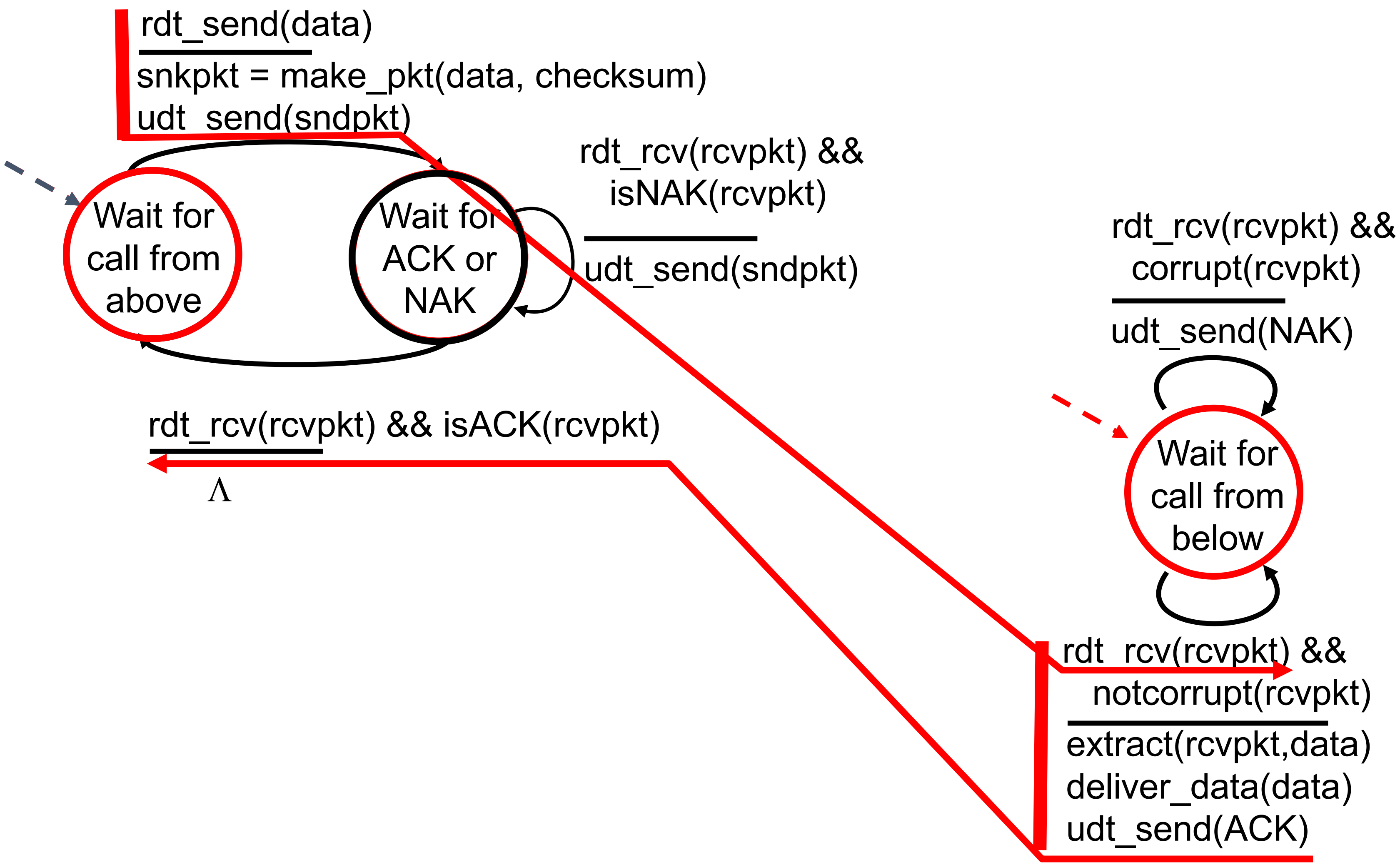
Big Picture of rdt2.0



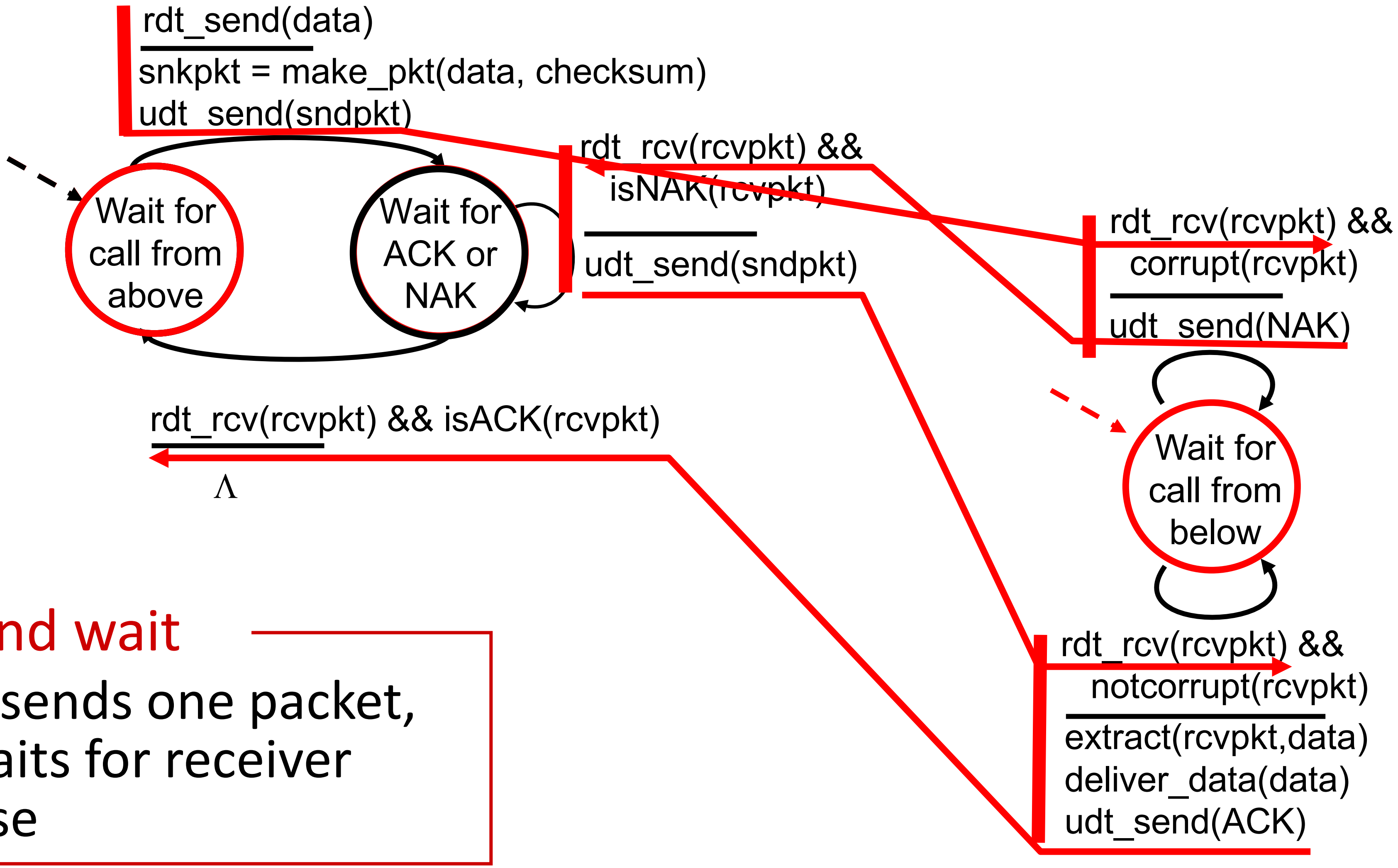
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



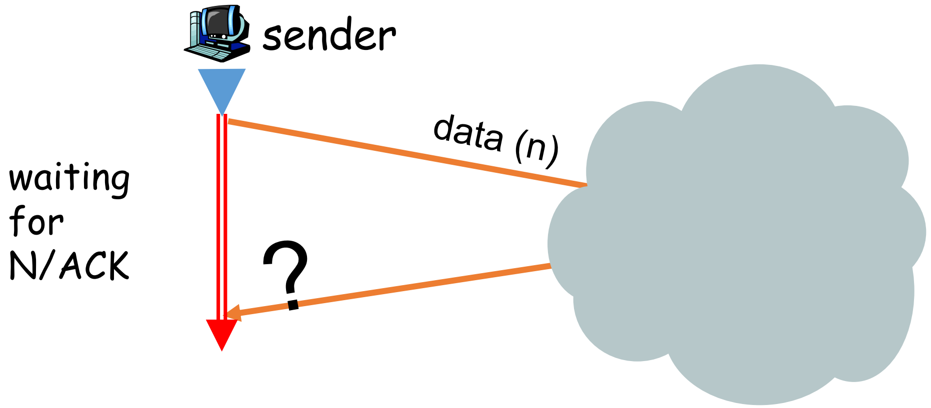
stop and wait

sender sends one packet, then waits for receiver response

rdt2.0 is Incomplete!

What happens if ACK/NAK corrupted?

- Although sender receives feedback, but doesn't know what happened at receiver!



Handle Control Message Corruption

It is always harder to deal with control message errors than data message errors

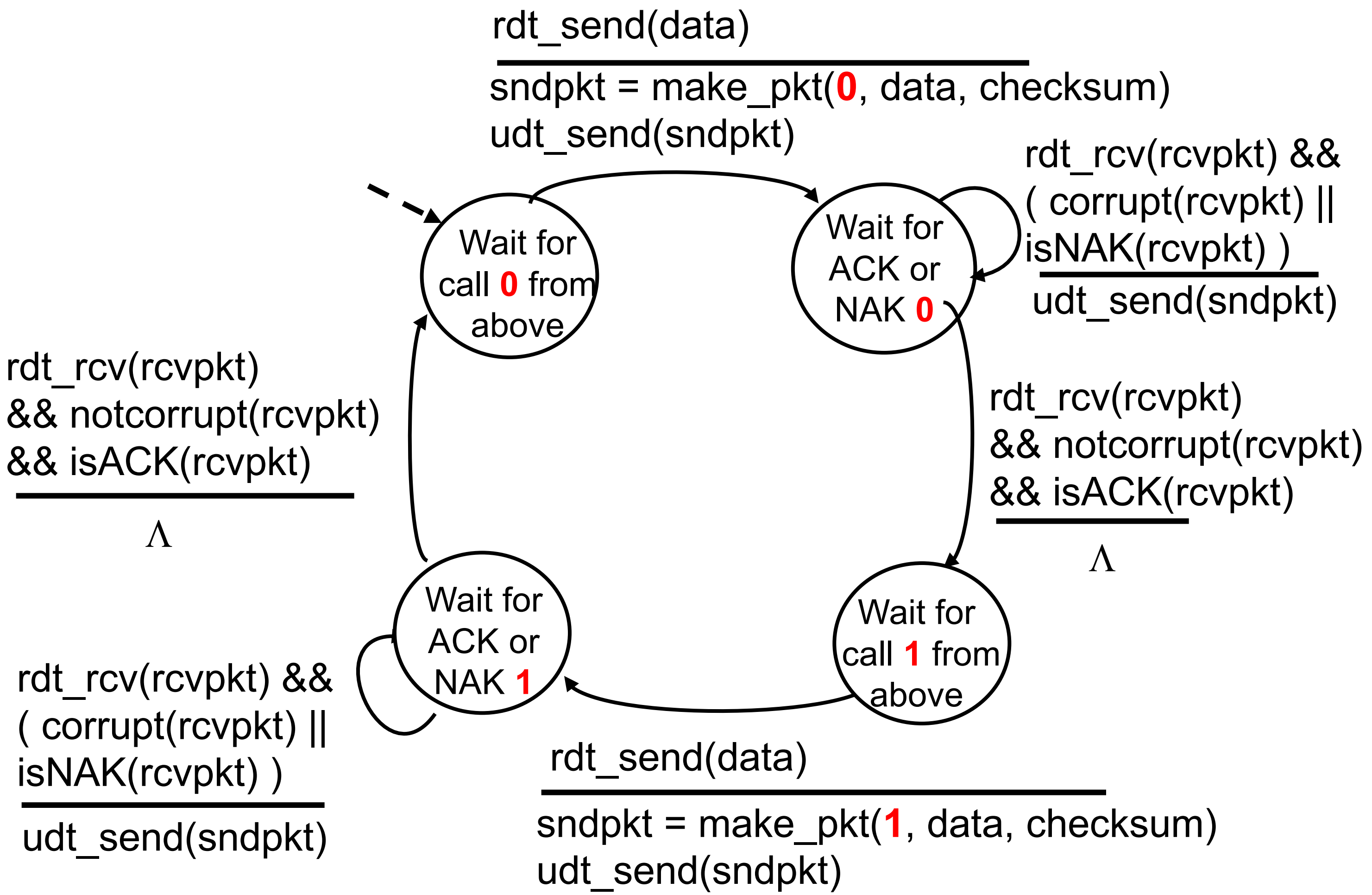
- sender can't just retransmit: possible duplicate
- neither can sender assumes received ok: possible missing packet

Handling duplicates:

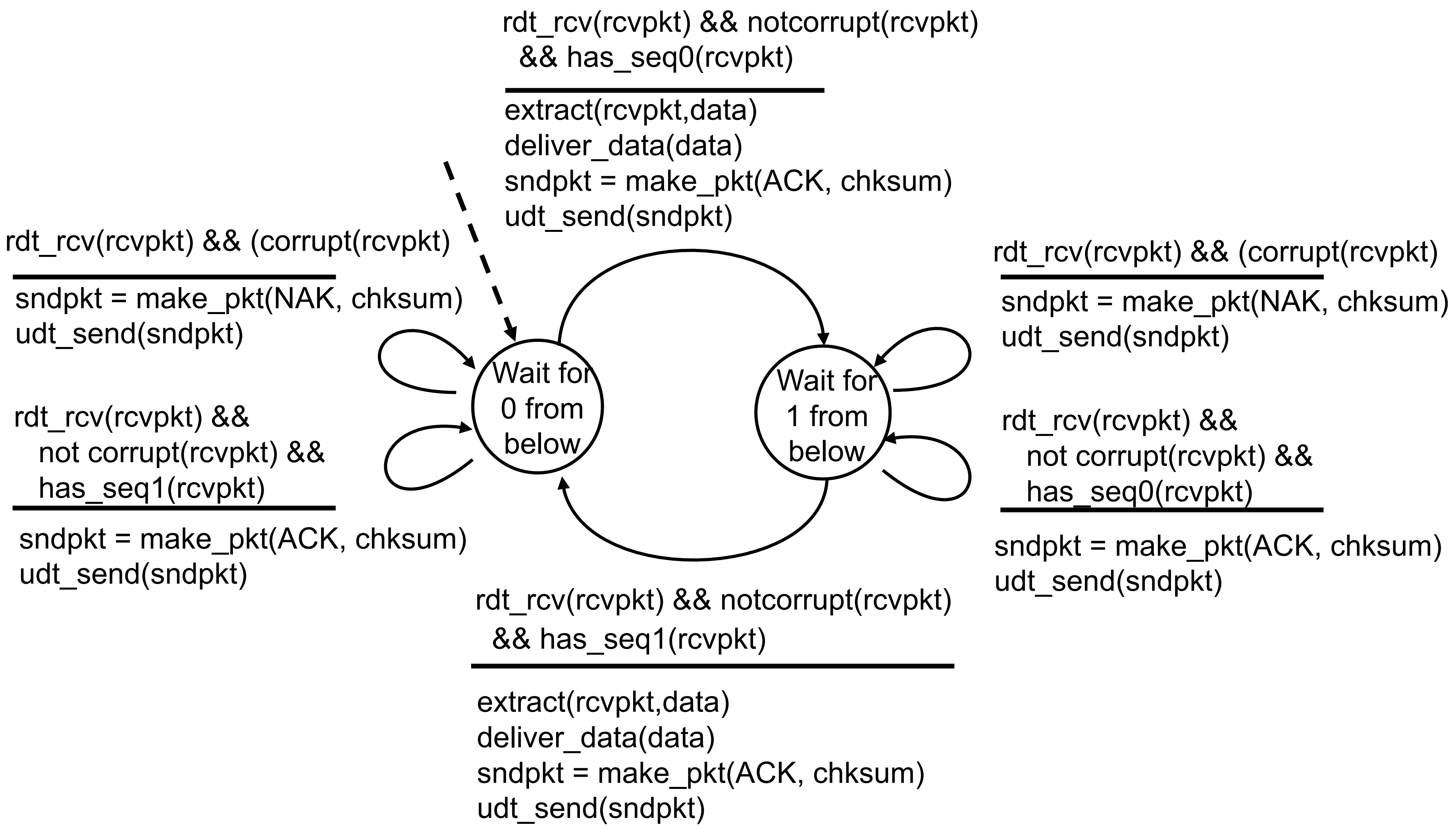
- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait
sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs: Using 1 bit (Alternating-Bit Protocol)



rdt2.1: receiver, handles garbled ACK/NAKs: Using 1 bit



rdt2.1: discussion

sender:

- state must "remember" whether "current" pkt has seq # of 0 or 1

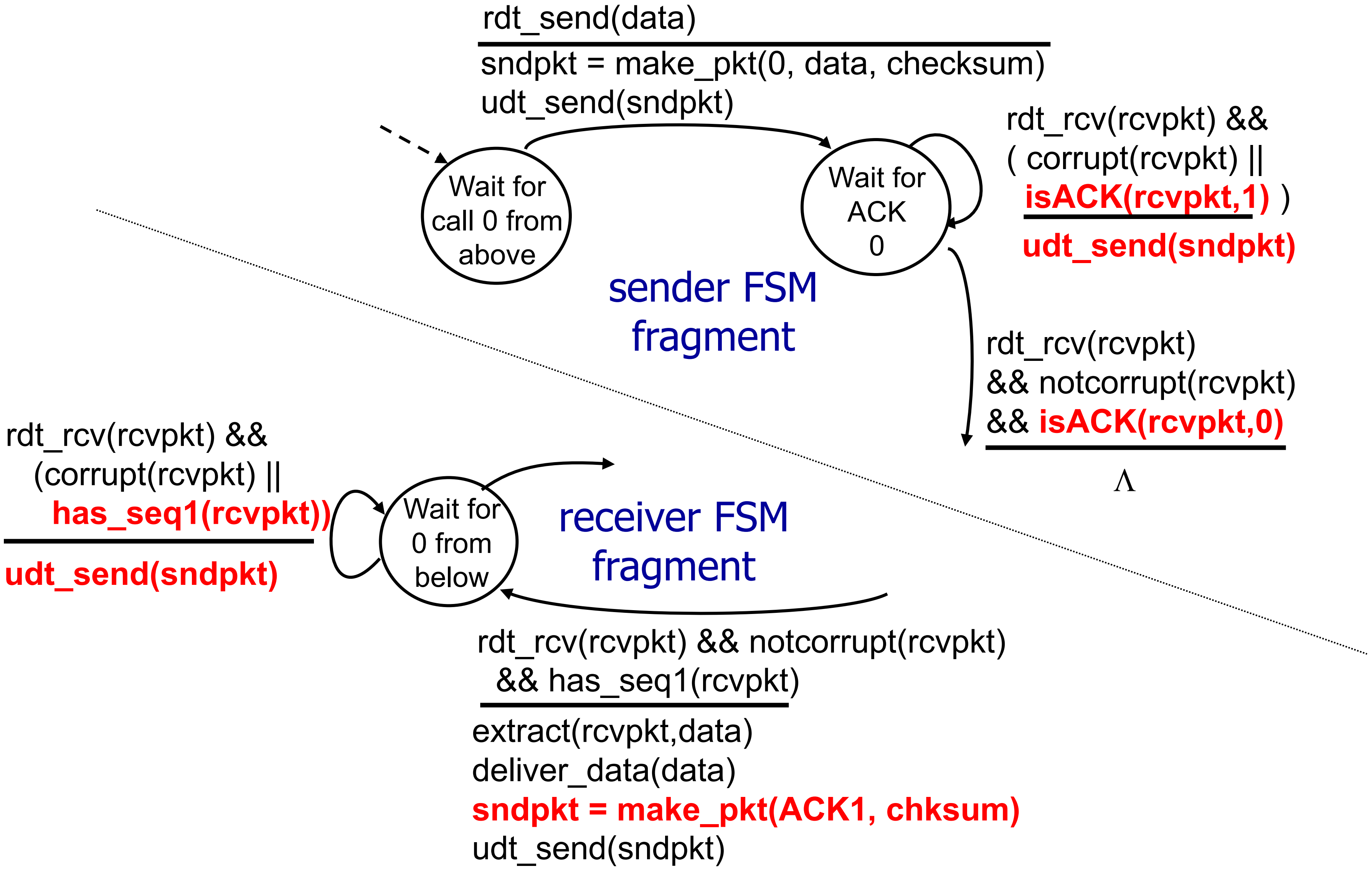
receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #

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: sender, receiver fragments



rdt3.0: channels with errors *and* loss

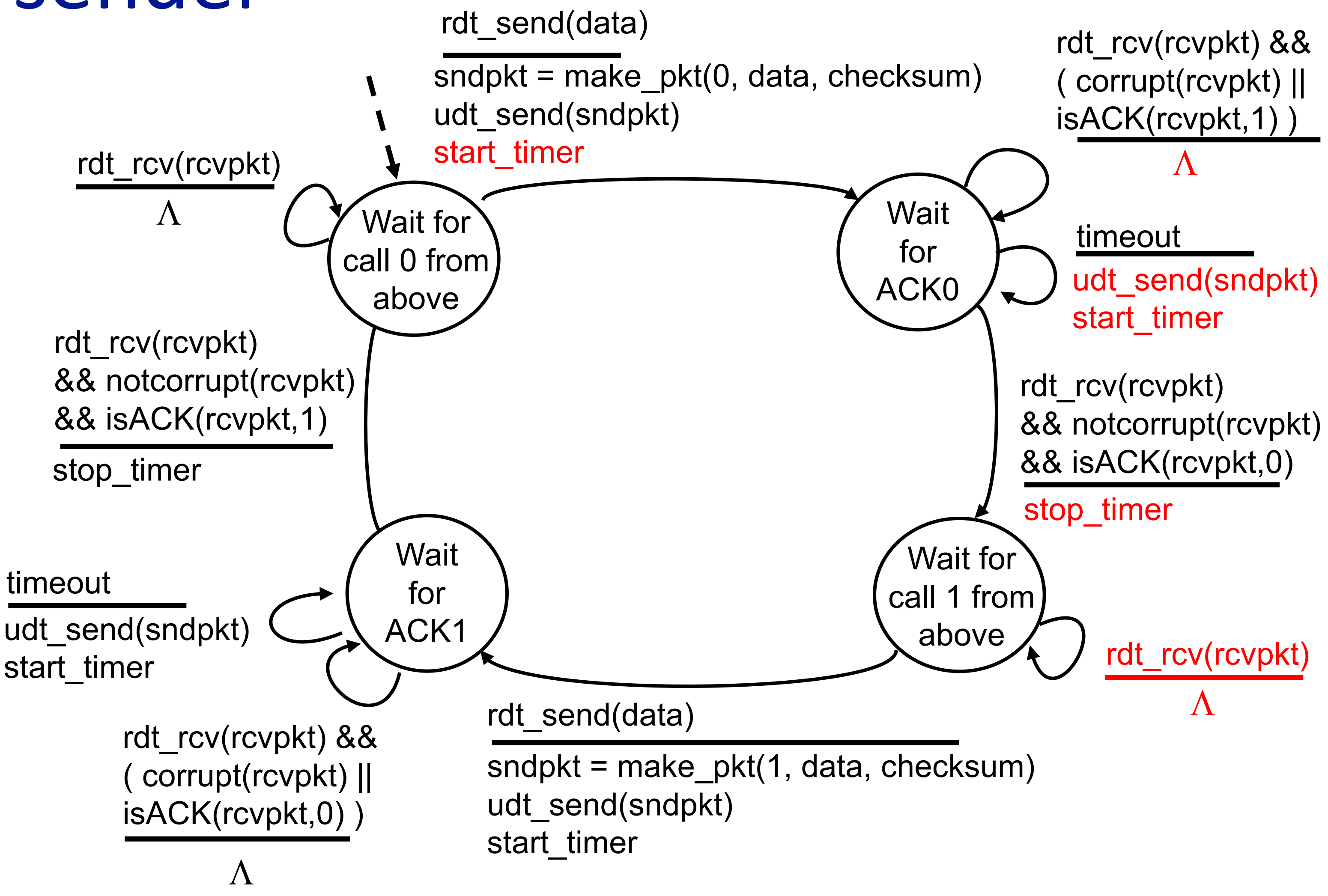
new assumption: underlying channel can also lose packets (data, ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

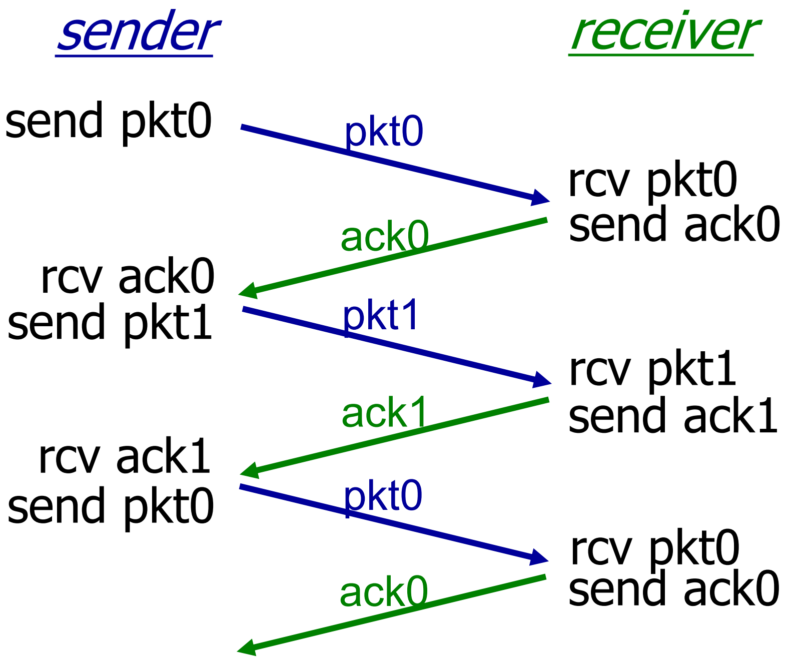
approach: sender waits "reasonable" amount of time for ACK

- requires countdown timer
- 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

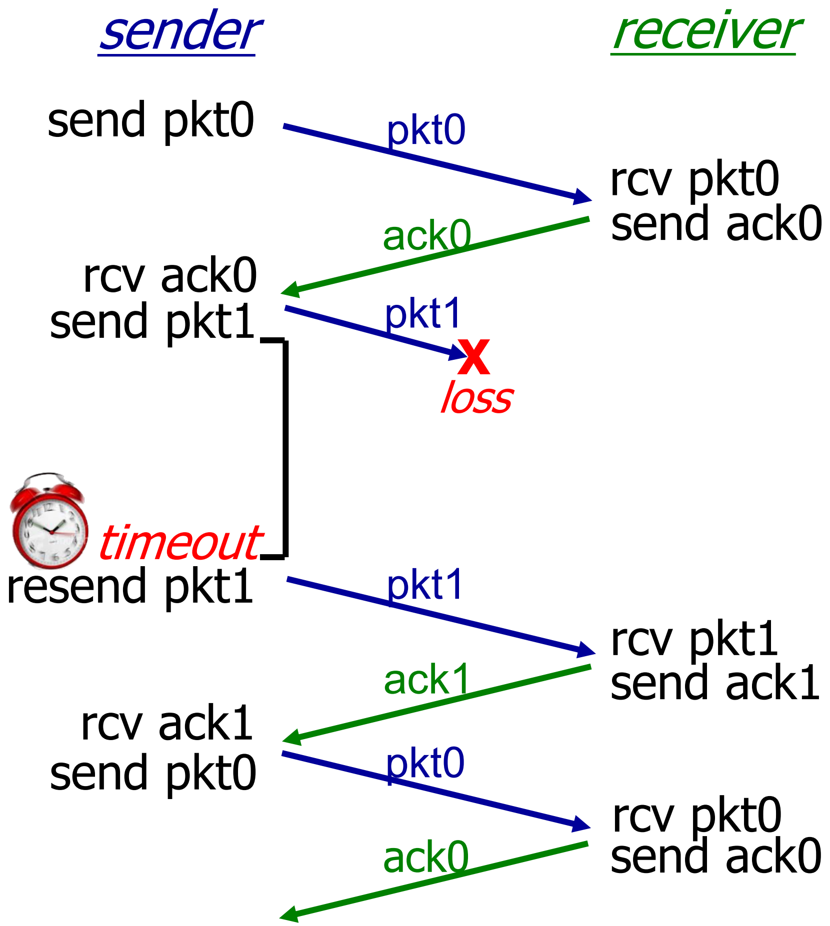
rdt3.0 sender



rdt3.0 in action

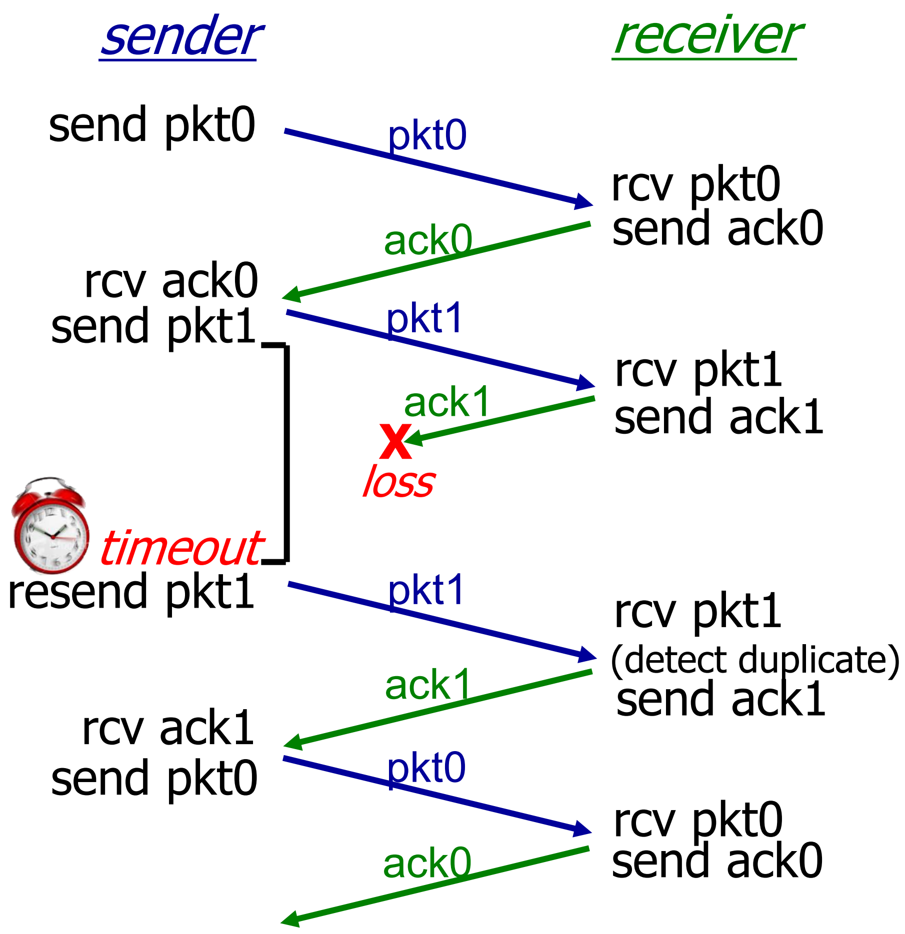


(a) no loss

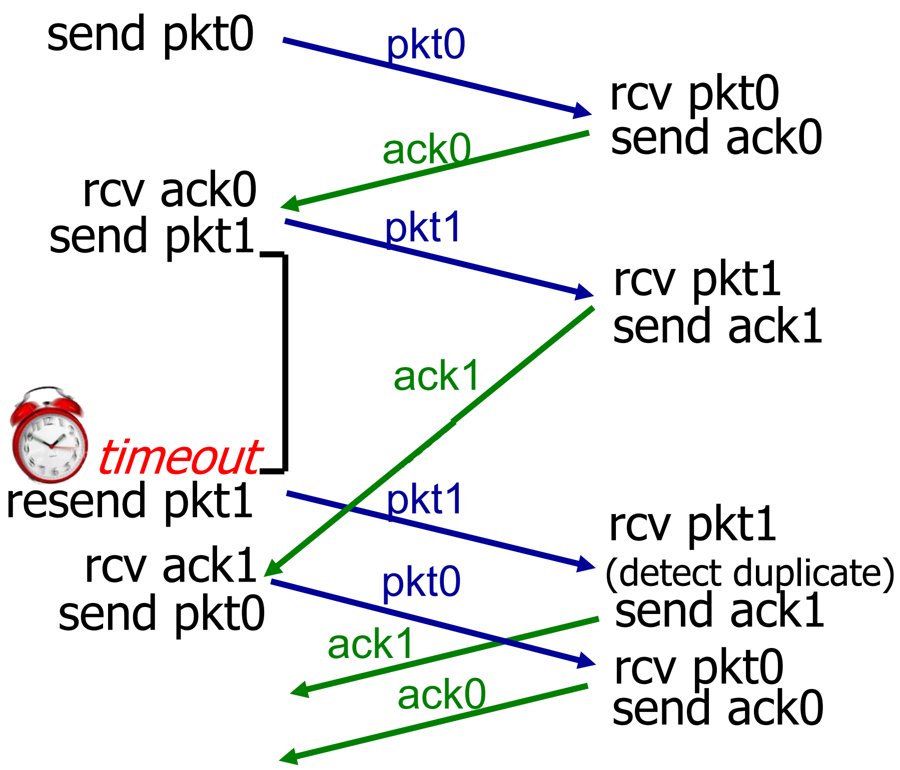


(b) packet loss

rdt3.0 in action



(c) ACK loss

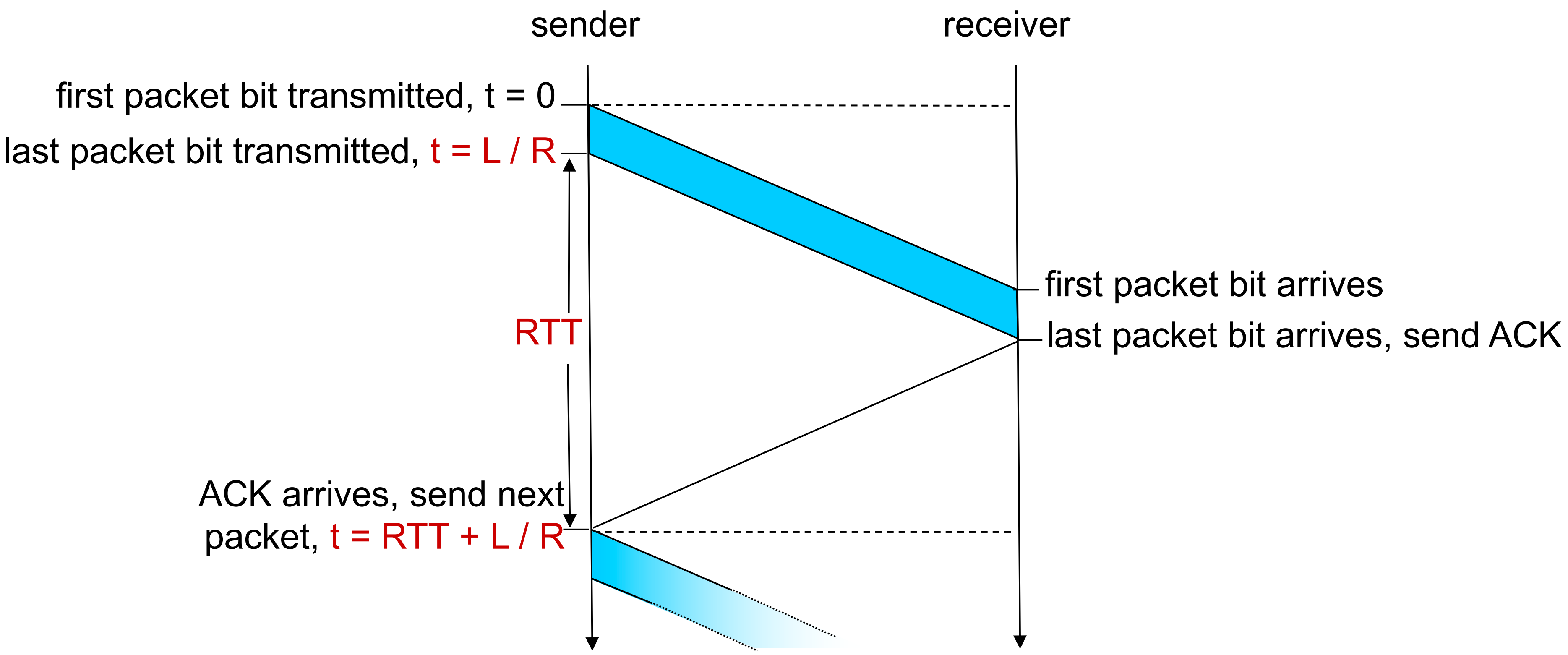


(d) premature timeout/ delayed ACK

A Summary of Questions

- How to improve the performance of rdt3.0?
- What if there are reordering and duplication?
- How to determine the “right” timeout value?

rdt3.0: stop-and-wait performance



What is the utilization of sender – fraction of time sender busy sending?

Assume: 1 Gbps link, 15 ms prop. delay, 1KB packet

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 1 KB packet:

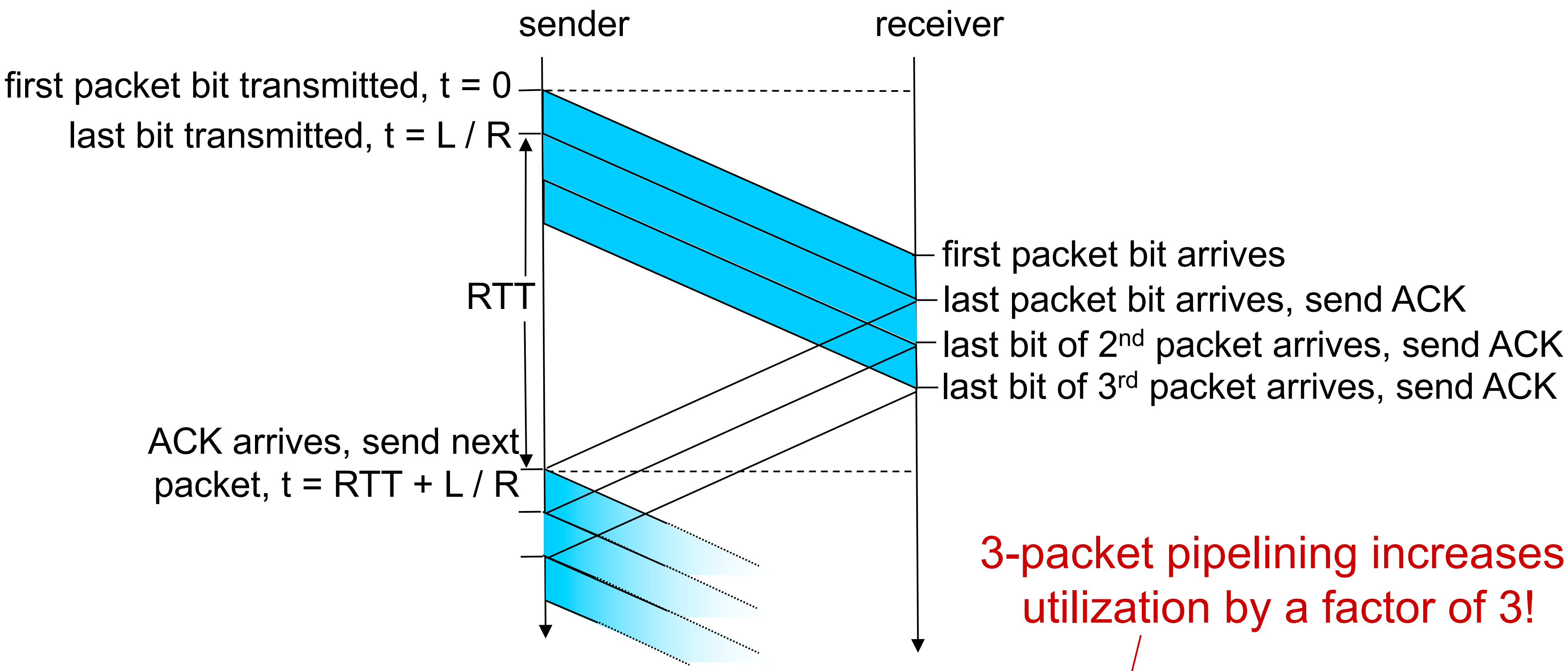
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U_{sender}: *utilization* – fraction of time sender busy sending

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

  - 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

Pipelining: increased utilization

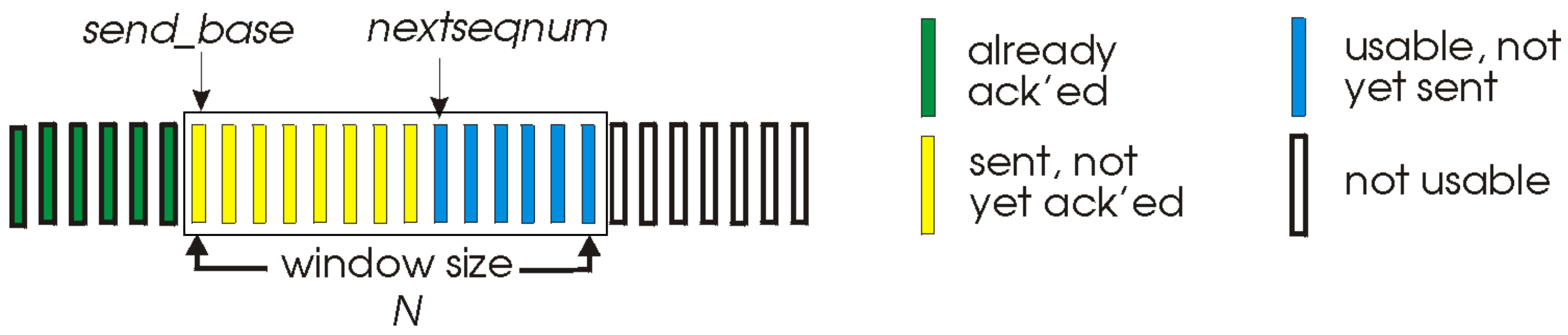


3-packet pipelining increases utilization by a factor of 3!

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

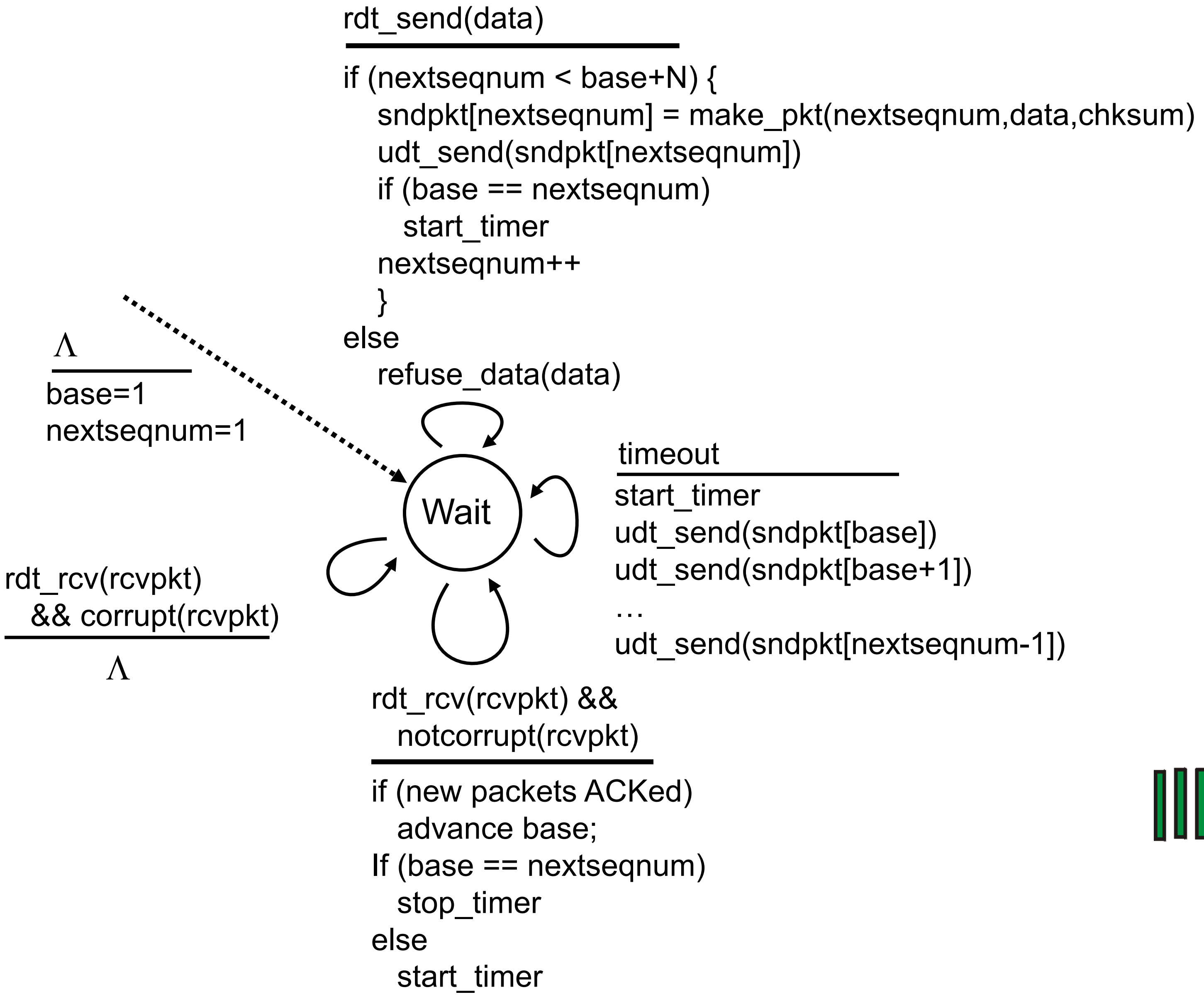
two generic forms of pipelined protocols: *go-Back-N, selective repeat*

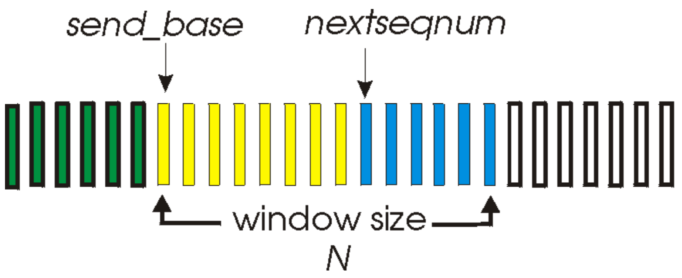
Go-Back-N Overview



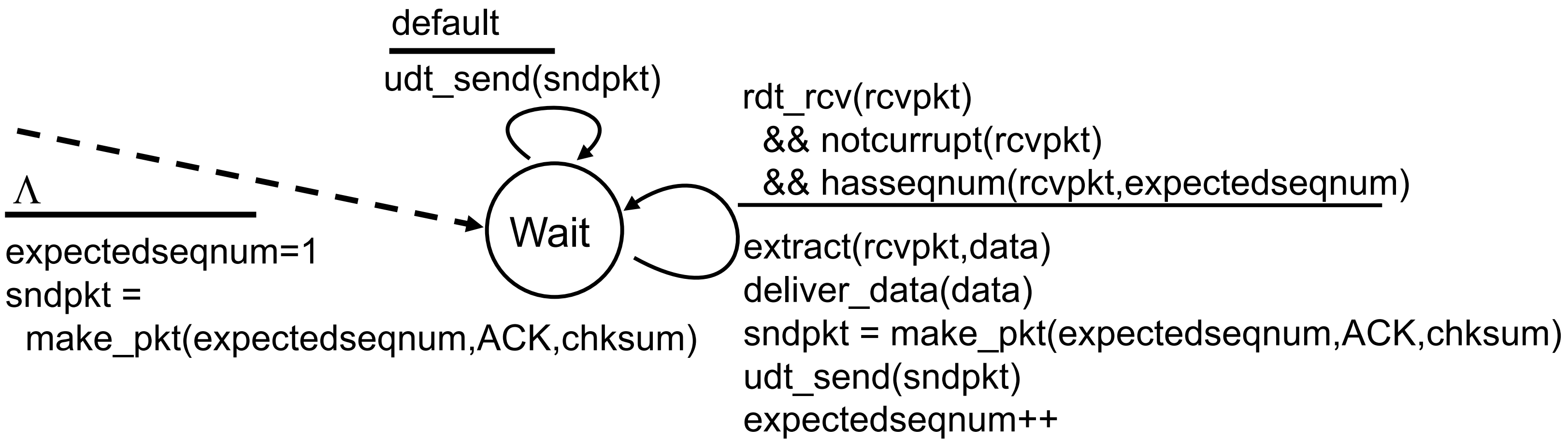
- sender keeps a window of packets
  - window represents a series of consecutive sequence numbers
  - window size *N* : number of un-ACKed packets allowed
- cumulative ACKs
  - ACK(*n*): acks packets up to and including *n*
  - sender may receive duplicate acks
- go-back-*N*
  - sender keeps a timer for the oldest in-flight packet
  - timeout(*n*) : retransmit packet *n* and all higher packets
  - no receiver buffering!

GBN: sender extended FSM



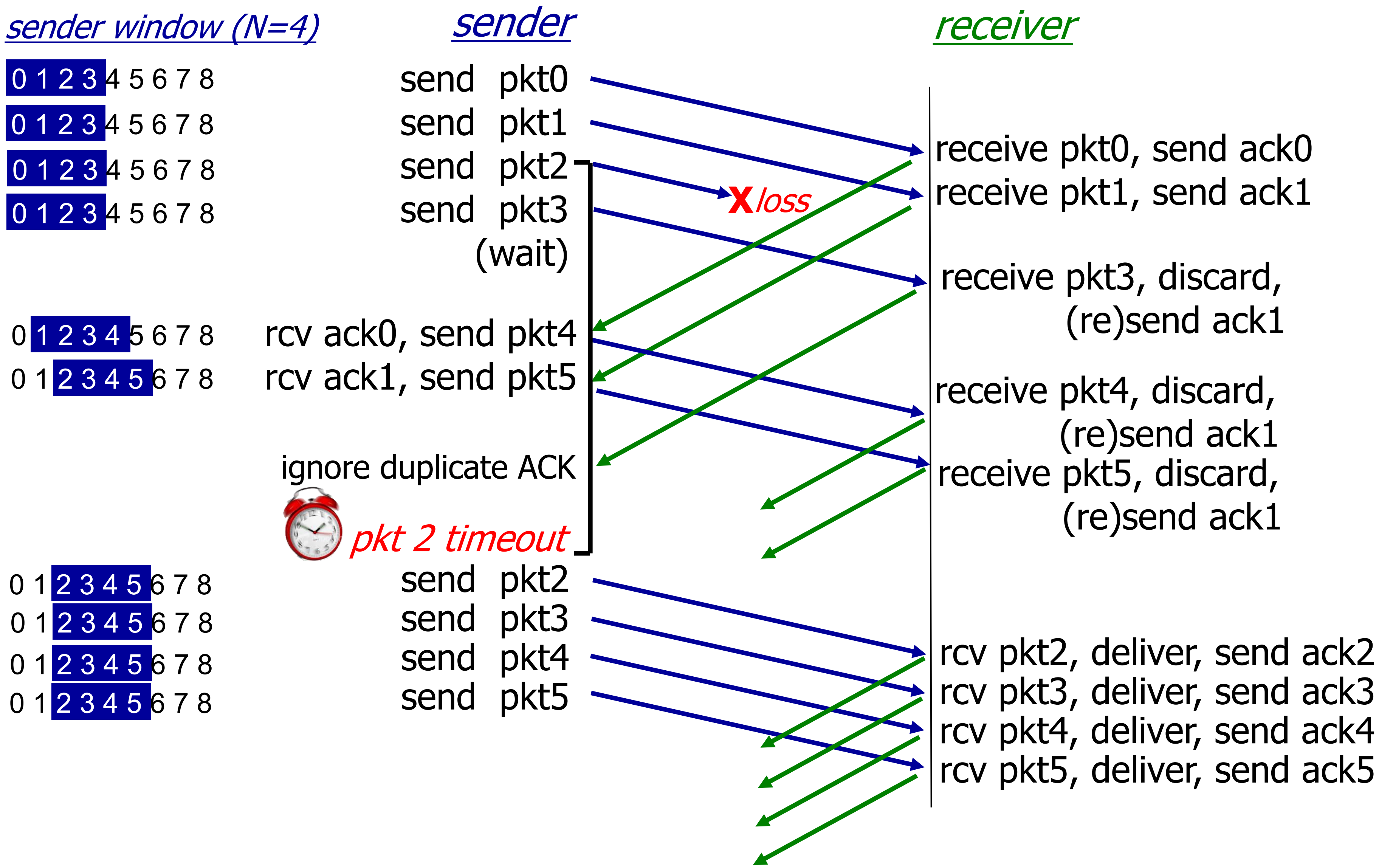


GBN: receiver extended FSM



- only state: **`expectedseqnum`**
- out-of-order packet:
  - discard: *no receiver buffering!*
  - re-ACK packet with highest in-order sequence number
  - may generate duplicate ACKs

GBN in action

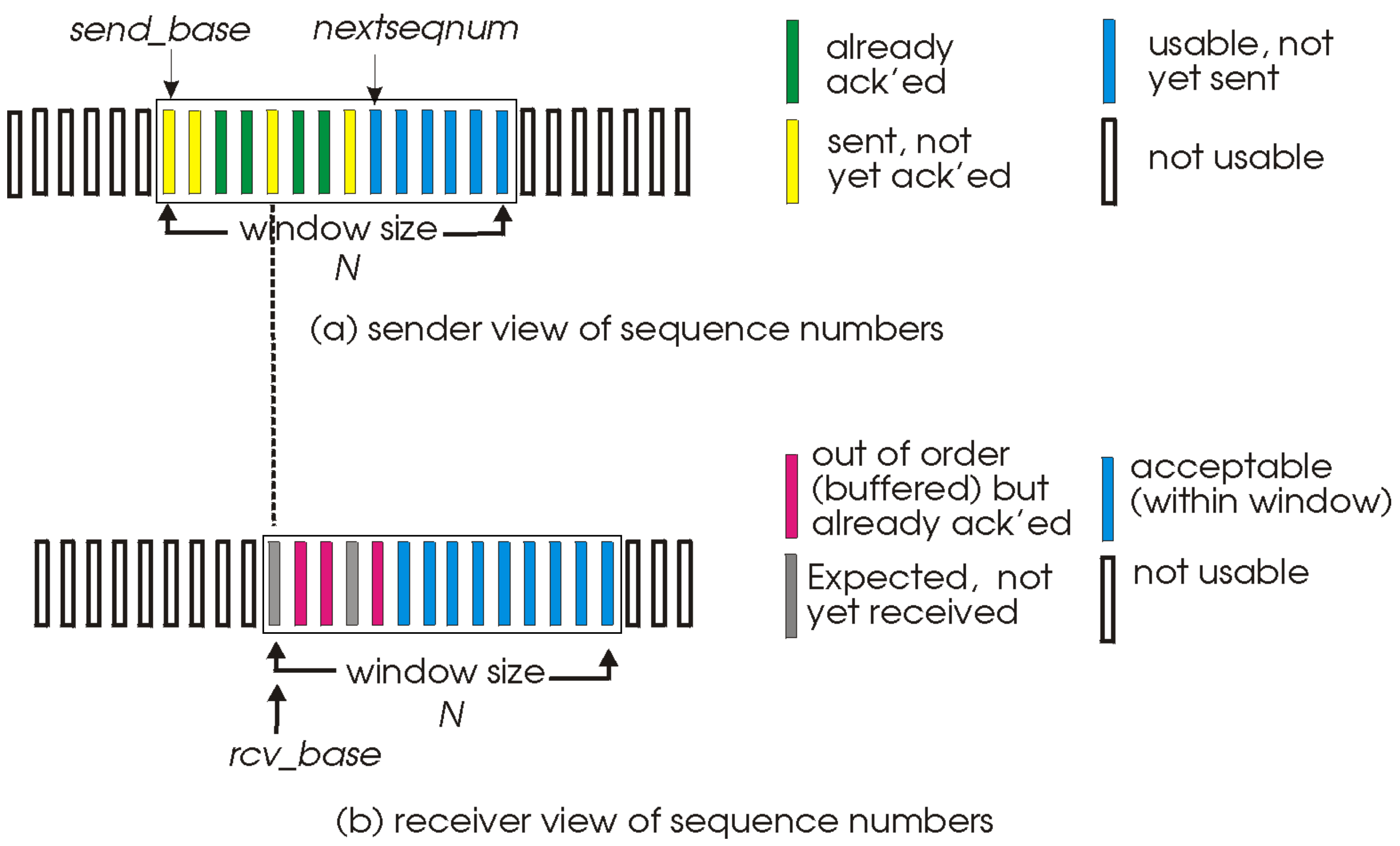


Selective repeat

- sender keeps a window of packets
  - window represents a series of consecutive sequence numbers
  - window size *N*: number of un-ACKed packets allowed

  ⇒ same as Go-Back-N
- selective ACKs
  - ACK(*n*): ACKs only sequence number *n*
- selective repeat
  - sender keeps a timer for each packet
  - timeout(*n*) : retransmit packet *n* only
  - receiver must buffer out-of-order packets!

Selective repeat: sender, receiver windows



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-1]:

- 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 buffered in-order pkts, advance window to next not-yet-received pkt

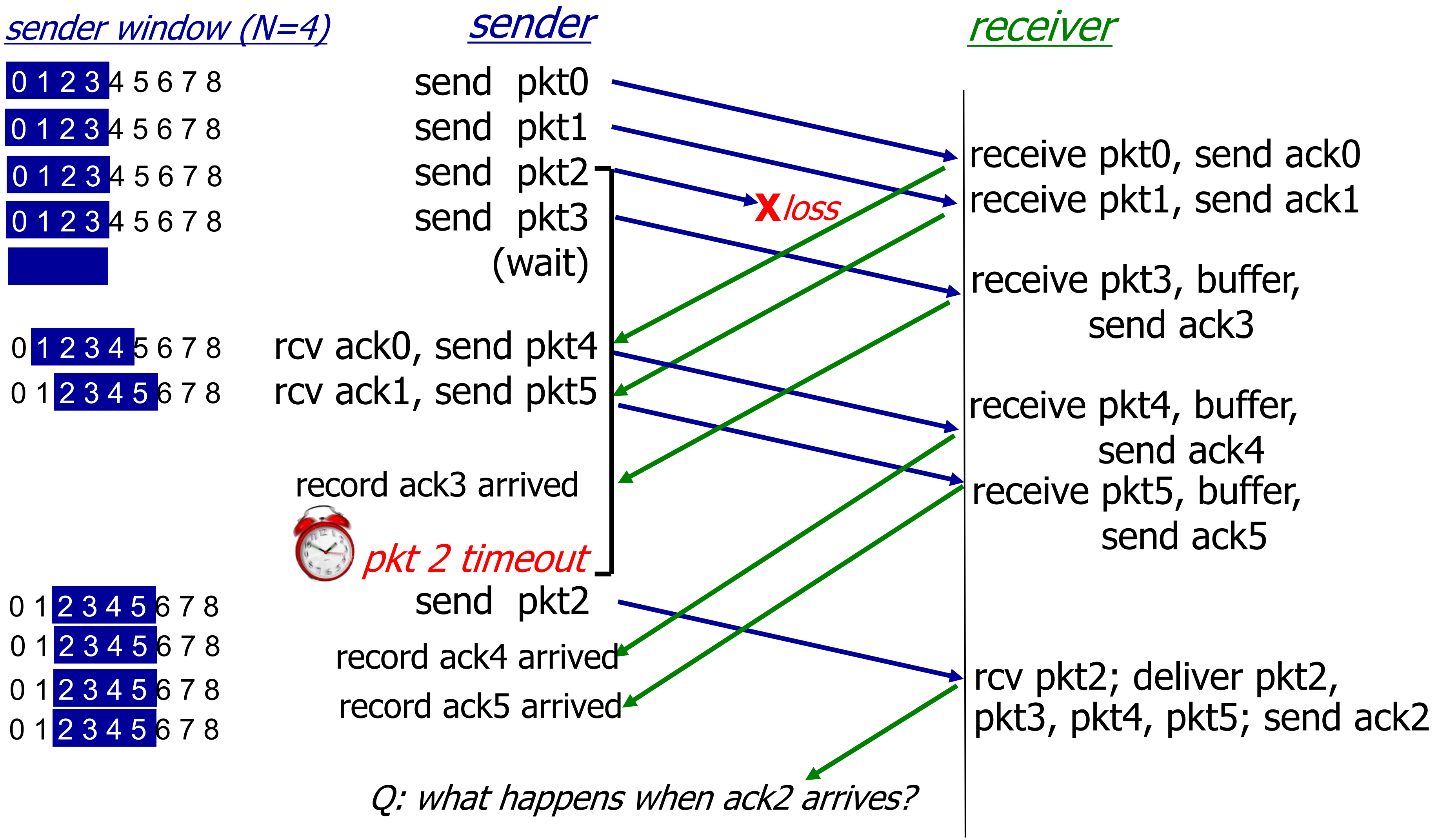
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



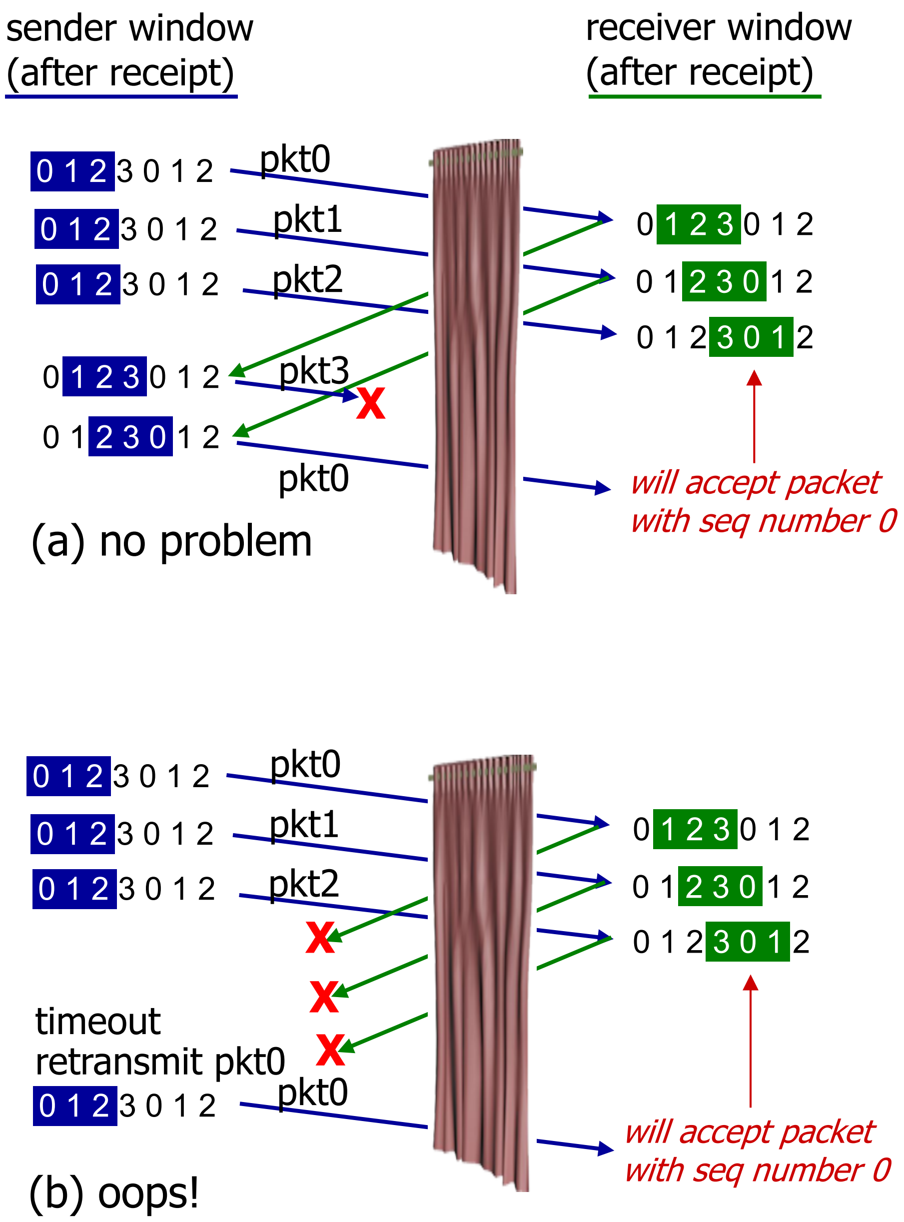
Selective repeat: dilemma

example:

- 2 bit sequence number
- window size=3

- receiver sees no difference in two scenarios!

- Q: how large should sequence space be?



Sliding Window Protocols: Go-back-*N* vs. Selective Repeat

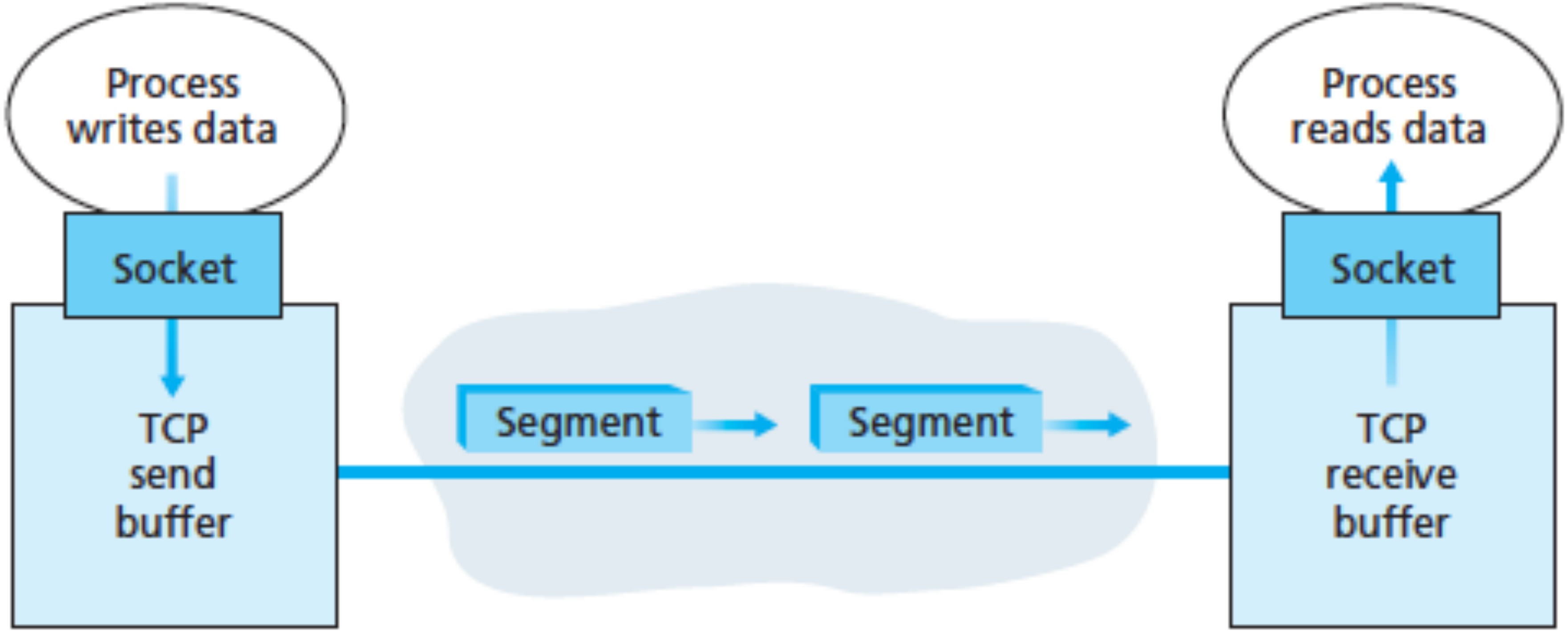
	Go-back-*N*	**Selective Repeat**
data bandwidth: sender to receiver	Less efficient	More efficient
ACK bandwidth (receiver to sender)	More efficient	Less efficient
Relationship between *M* (the number of seq#) and *N* (window size)	?	?
Buffer size at receiver	1	*N*
Complexity	Simpler	More complex

Outline

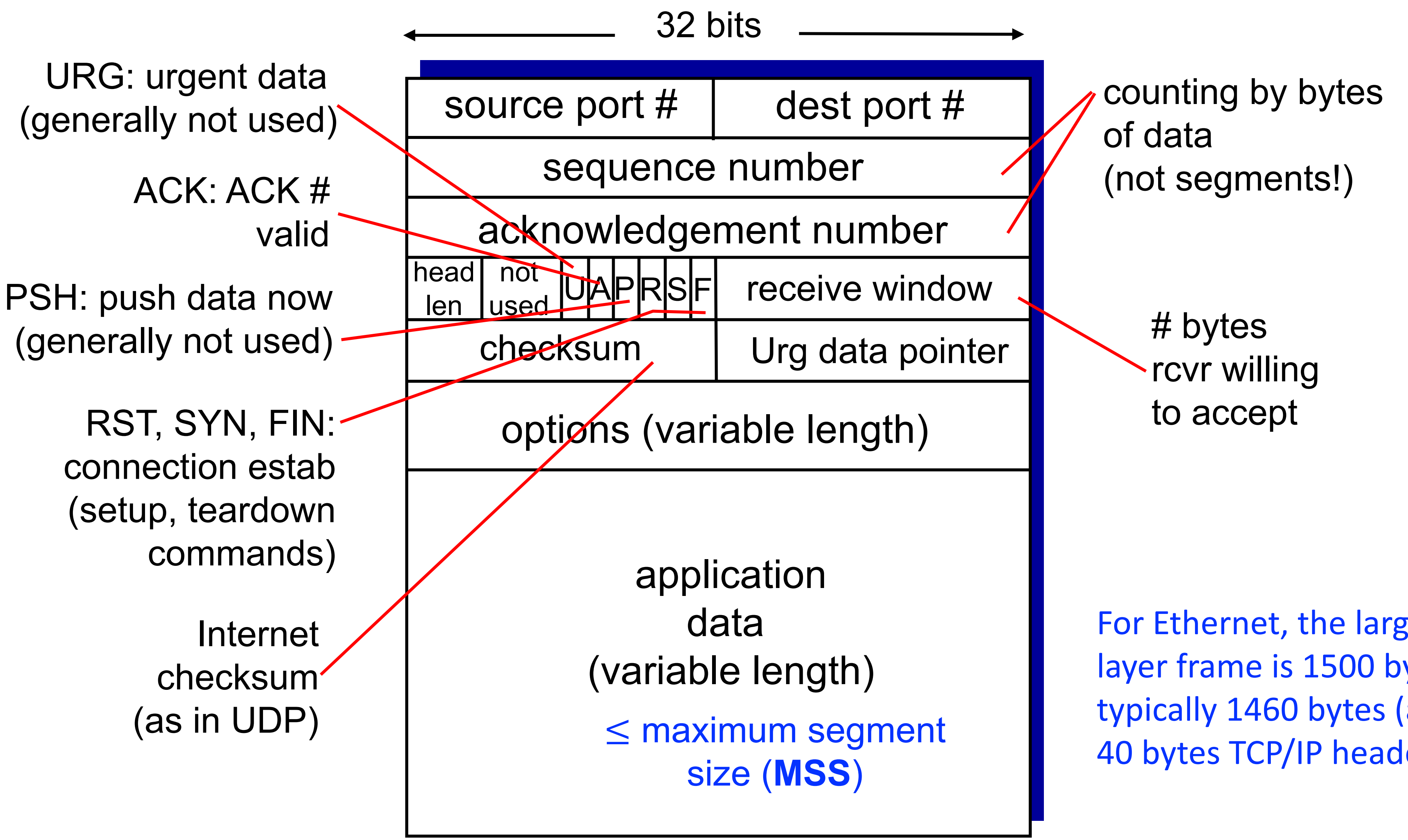
- Overview of transport-layer services
- Connectionless Transport: UDP
- Principles of reliable data transfer
- Connection-Oriented Transport: TCP
- TCP congestion control

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


Process
writes data
Socket
TCP
send
buffer
Segment
Segment
Process
reads data
Socket
TCP
receive
buffer

TCP segment structure



For Ethernet, the largest link layer frame is 1500 bytes, MSS is typically 1460 bytes (assuming 40 bytes TCP/IP header)

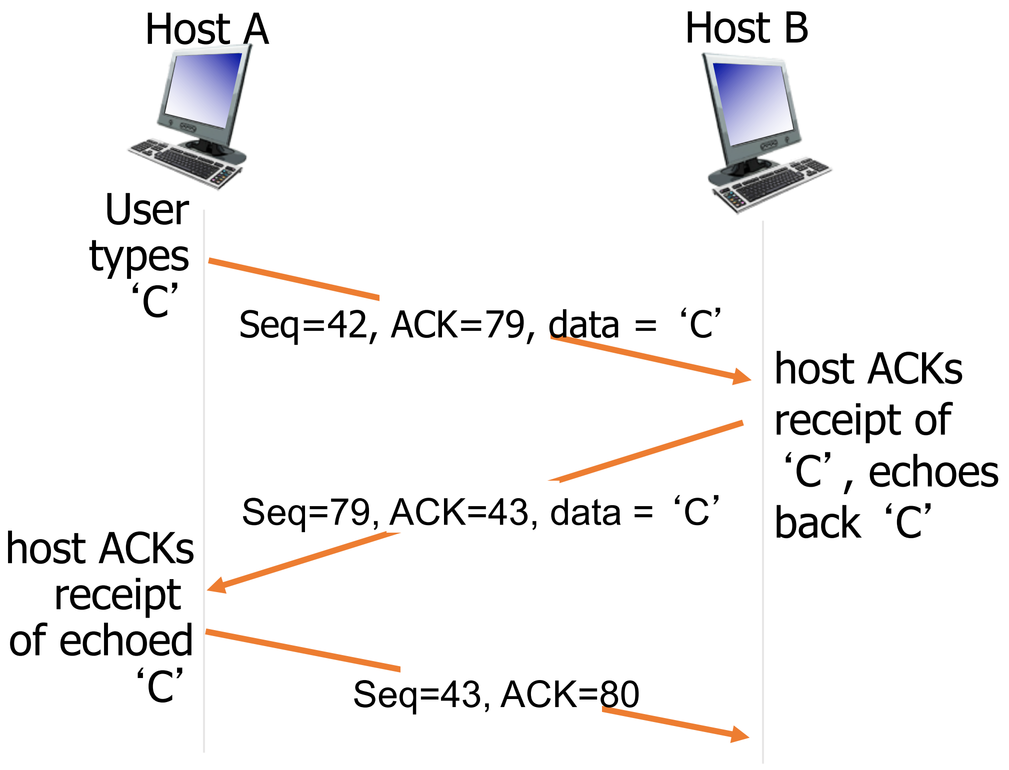
TCP seq. numbers, ACKs

sequence numbers:

- byte stream "number" of first byte in segment's data

acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK



simple telnet scenario

TCP round trip time, timeout

Q: 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

Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **SampleRTT**

TCP round trip time, timeout

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

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT- EstimatedRTT|
```

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

estimated RTT

“safety margin”

TCP reliable data transfer

- TCP creates reliable data transfer service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

TCP sender events

data rcvd from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`**

timeout:

- retransmit segment that caused timeout
- restart timer

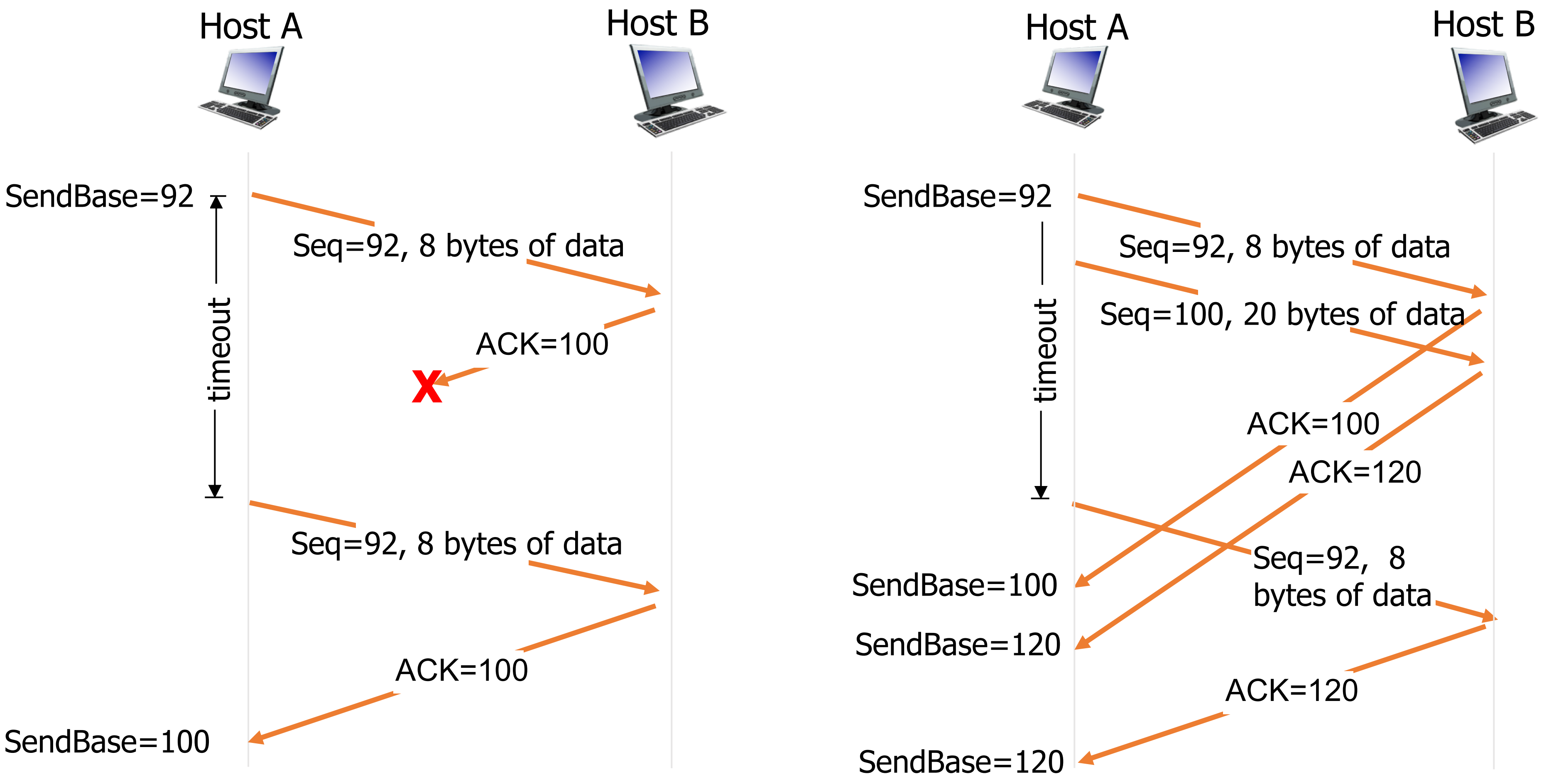
ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

TCP Receiver ACK generation [RFC 1122, RFC 2581]

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

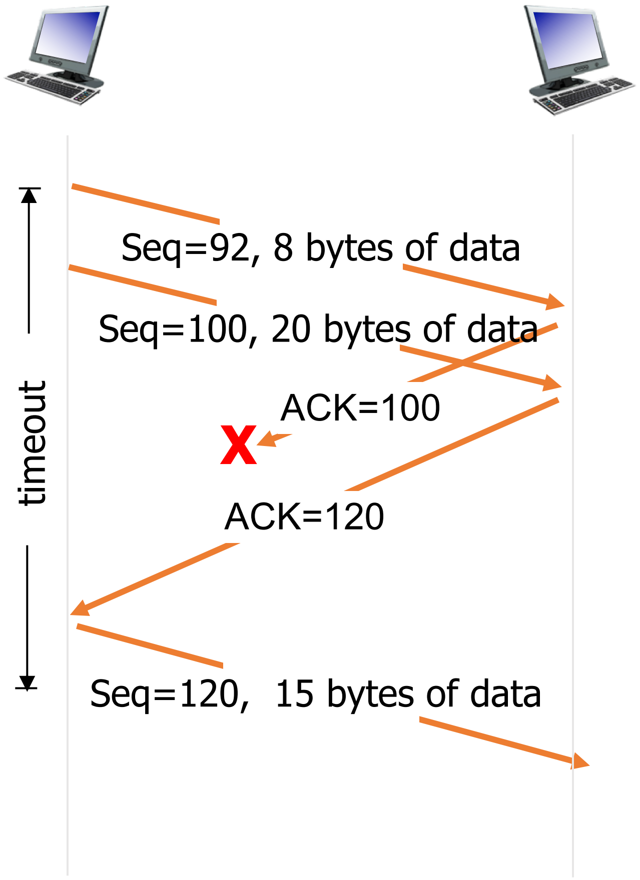
TCP: retransmission scenarios



lost ACK scenario

premature timeout

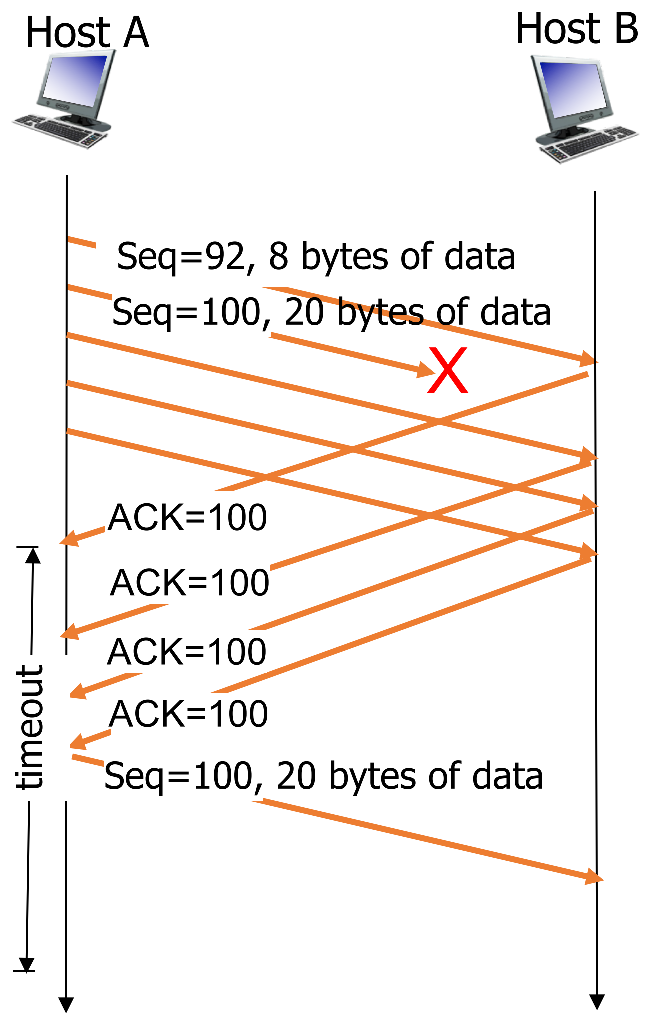
TCP: retransmission scenarios



cumulative ACK

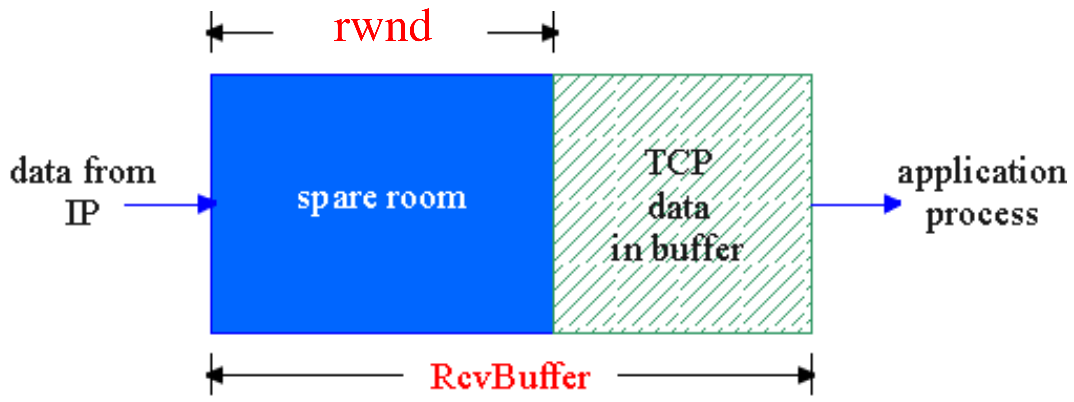
TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if a segment is lost, there will likely be many duplicate ACKs.
- *TCP fast retransmit*
  - if sender receives 3 duplicates ACKs for same data, resend unacked segment with smallest seq #
    - likely that unacked segment lost, so don't wait for timeout



Flow control

- receive side of a connection has a receive buffer:



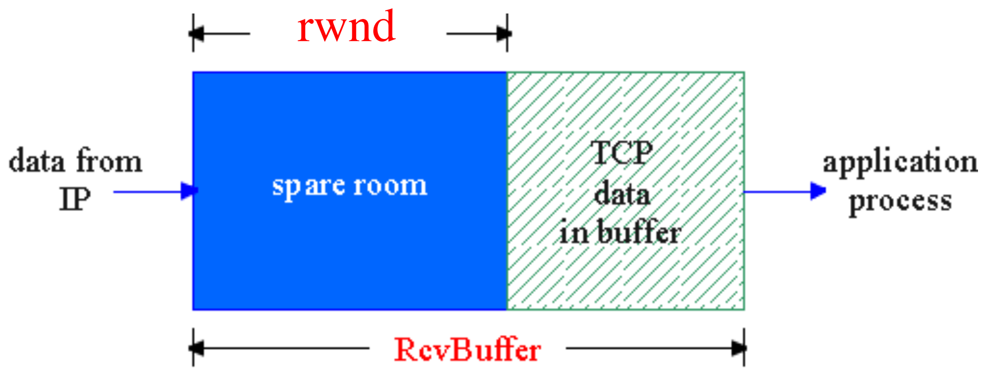
- app process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP flow control



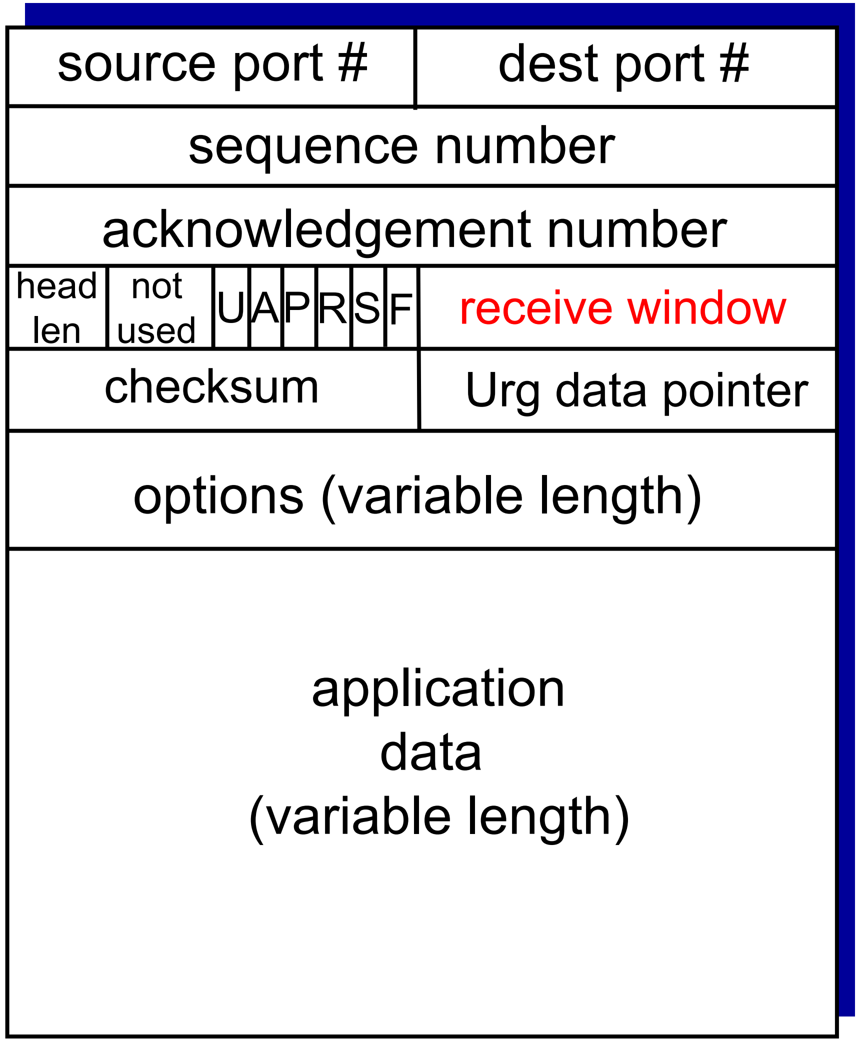
- spare room in buffer = **rwnd**

Receiver:

`rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]`

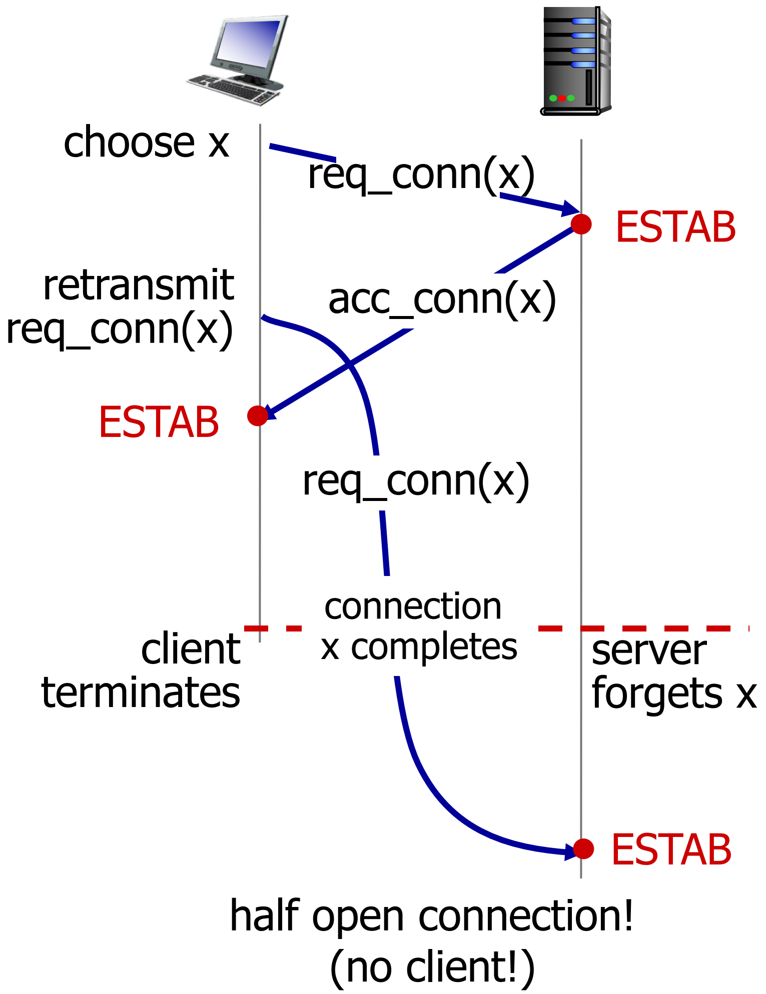
Sender:

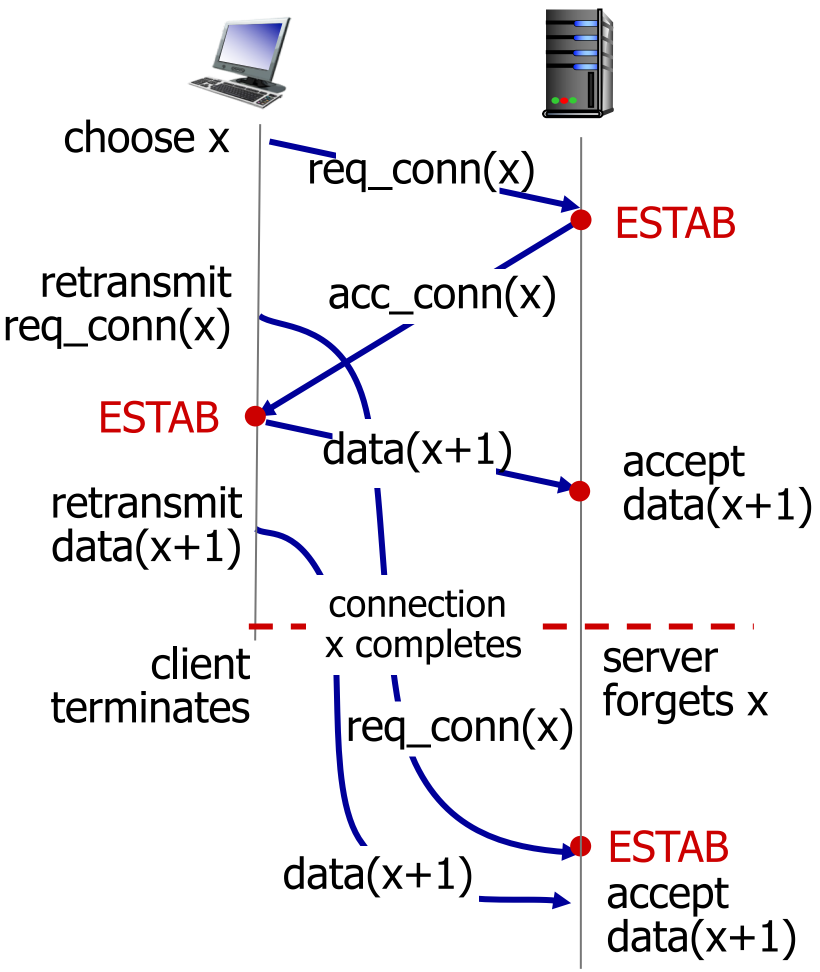
`LastByteSent – LastByteAcked ≤ rwnd`



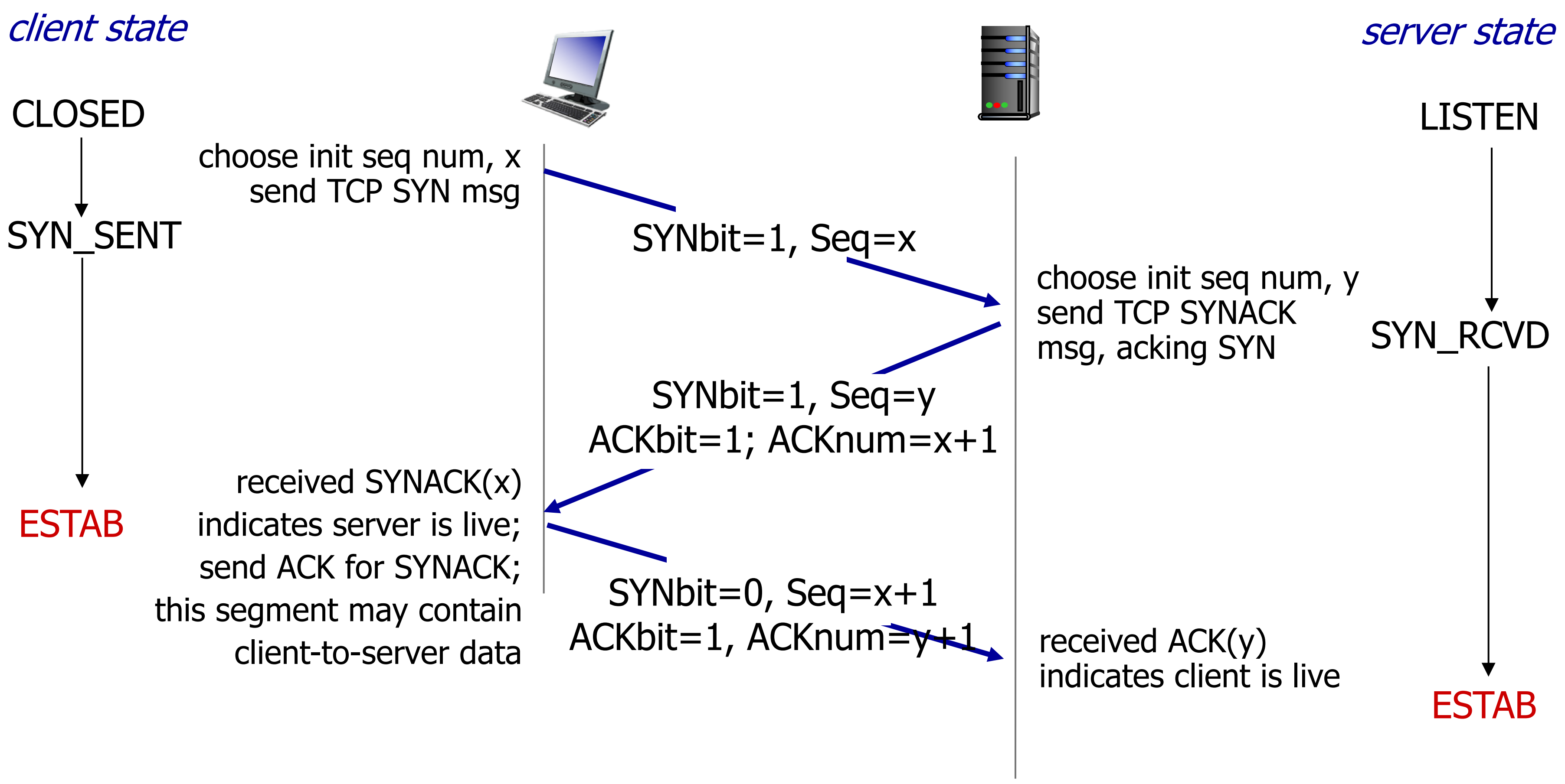
Connection Management

2-way handshake failure scenarios:

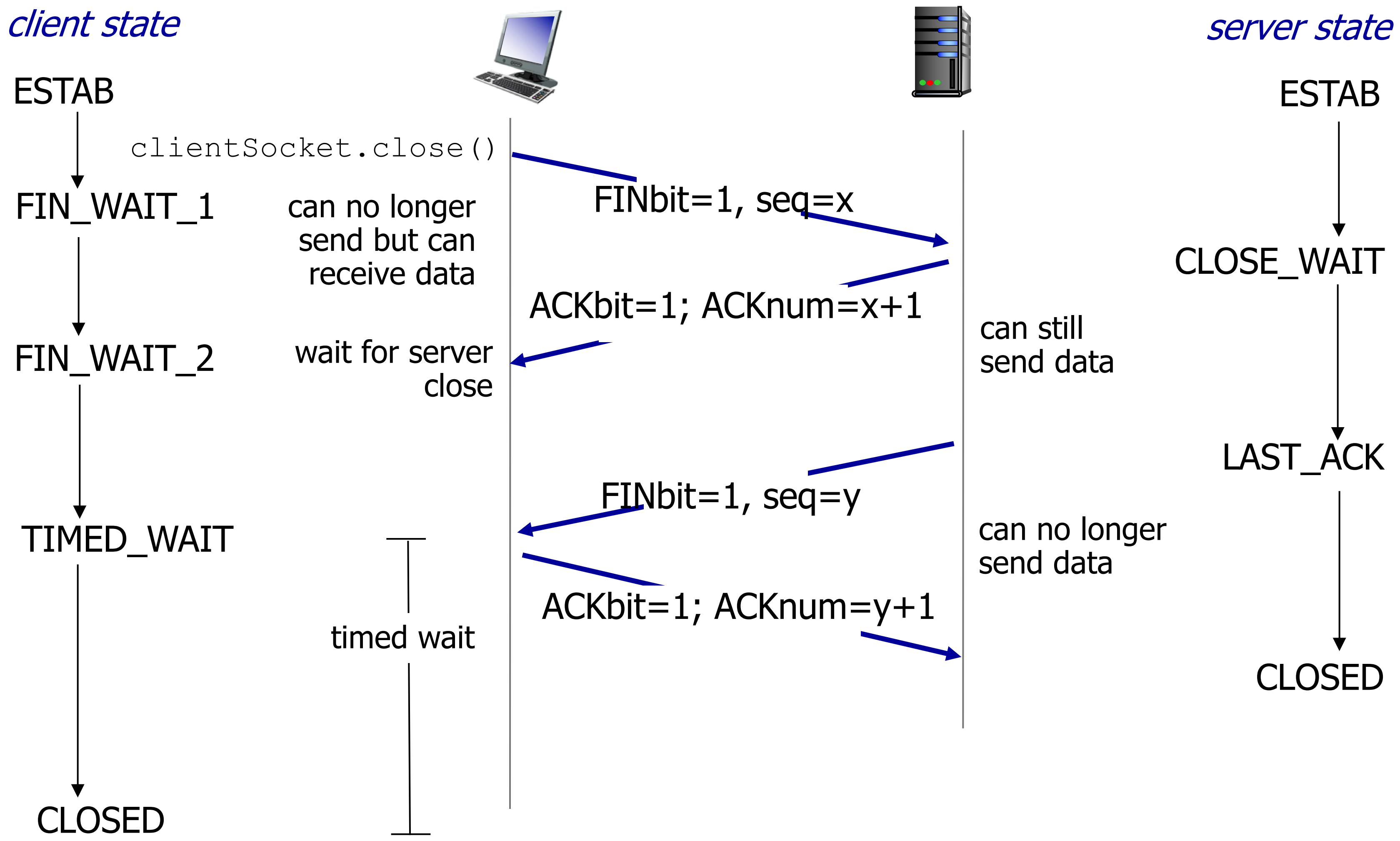




TCP 3-way handshake



TCP: closing a connection



Outline

- Overview of transport-layer services
- Connectionless Transport: UDP
- Principles of reliable data transfer
- Connection-Oriented Transport: TCP
- TCP congestion control

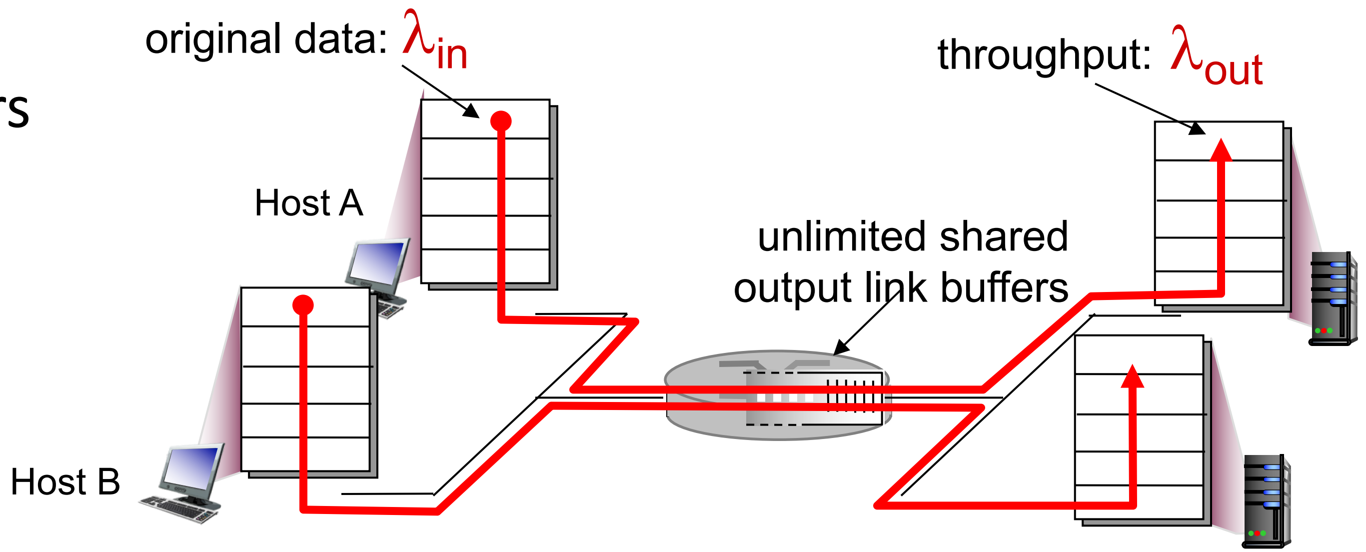
Principles of congestion control

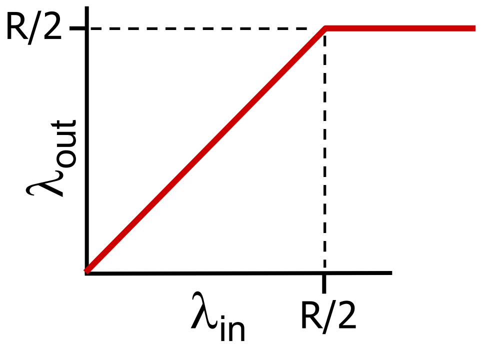
congestion:

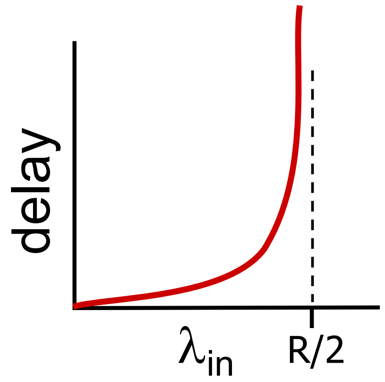
- informally: “too many sources sending too much data too fast for *network* to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffer
- output link capacity: R
- no retransmission





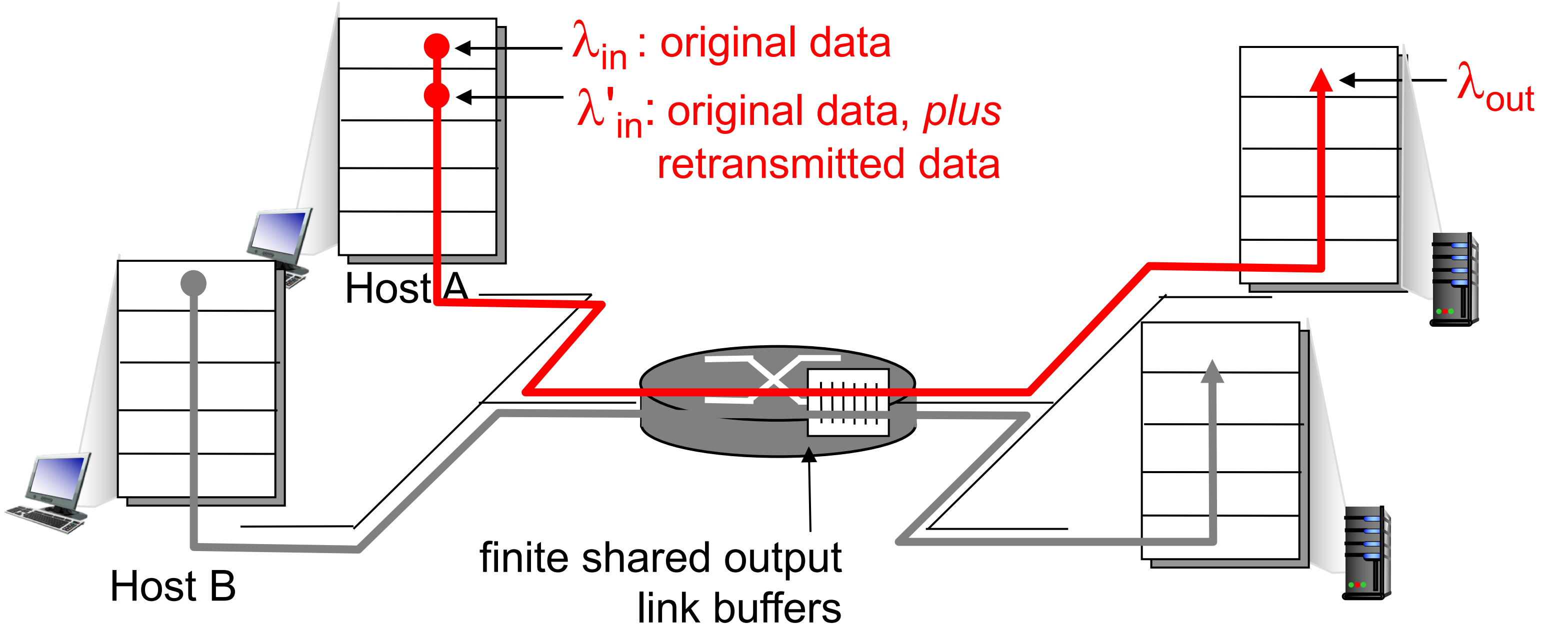


- maximum per-connection throughput: R/2

- large delays as arrival rate approaches capacity

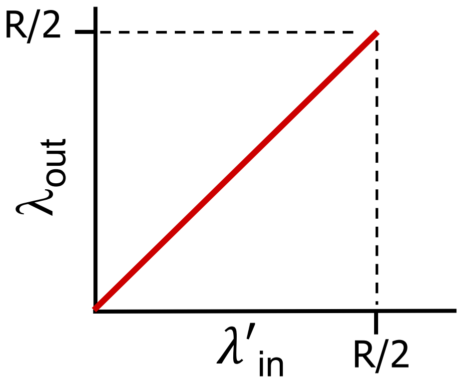
Causes/costs of congestion: scenario 2

- one router, *finite* buffer
- sender retransmits timed-out packets
  - transport-layer input includes *retransmissions*: $\lambda'_{in} \geq \lambda_{in}$



Causes/costs of congestion: scenario 2

- *Idealization: perfect sender*
  - sender sends only when router's buffer has free space
  - $\lambda'_{in} = \lambda_{in}$



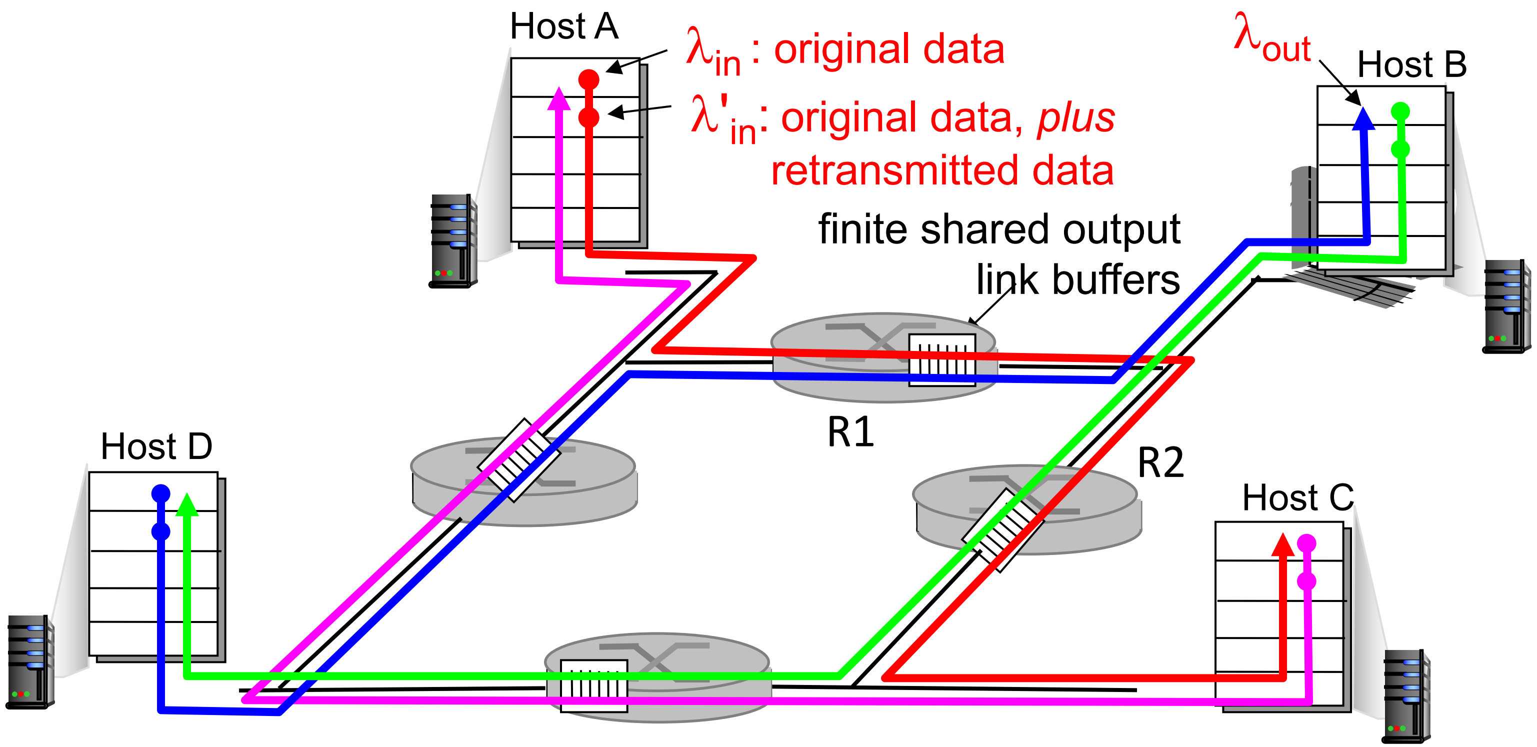
- *Idealization: known loss*
  - packets can be lost, dropped at router due to full buffer
  - sender only resends if packet *known* to be lost
  - cost of congestion: more work (retrans) for given "goodput", $\lambda'_{in} > \lambda_{out}$
- *Realistic: duplicates*
  - packets can be lost and sender may time out prematurely
  - cost of congestion: unneeded retransmissions (link carries multiple copies of pkt)

Causes/costs of congestion: scenario 3

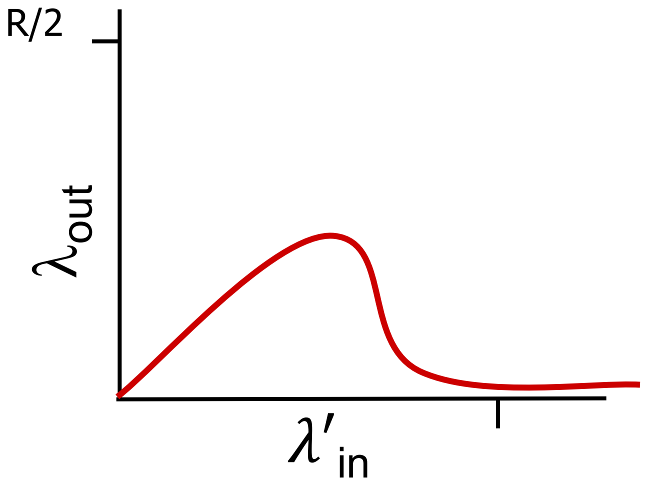
- four senders
- multihop paths
- timeout/retransmit

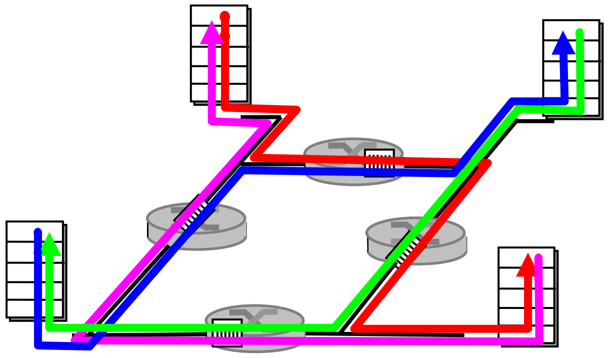
Q: what happens as λ_{in} and λ'_{in} increase ?

A: as B-D λ_{in} increases, all arriving A-C pkts are dropped at R2, A-C throughput goes to 0



Causes/costs of congestion: scenario 3





another "cost" of congestion:

- when packet dropped, any "upstream" transmission capacity used for that packet was wasted!

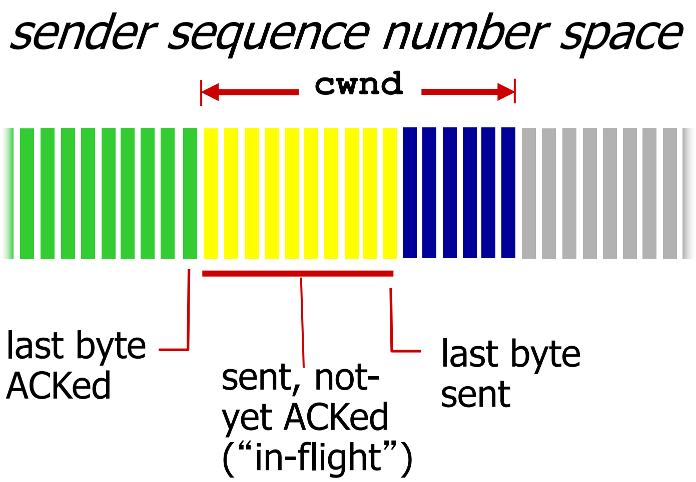
Approaches to Congestion Control

- End-to-end approach
  - No explicit feedback from the network layer
  - Indicators of network congestion
    - packet loss: indicated by time out or three duplicate ACKs
    - increasing round-trip delay
  - Implemented by TCP
- Network-assisted approach
  - Routers provide explicit feedback
  - Explicit Congestion Notification (ECN) has been proposed as extensions to TCP and IP

TCP Congestion Control

- End-to-end approach
- Have each sender limit the transmission rate as a function of perceived network congestion
  - How to limit rate — Sender’s congestion window
  - How to perceive congestion — A “loss event”: either a timeout or the receipt of three duplicate ACKs
  - What algorithm to change rate — Additive increase, multiplicative decrease (AIMD)

TCP Congestion Control: Congestion Window



TCP sending rate:

- *roughly:* send cwnd bytes, wait RTT for ACKs, then send more bytes

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

- sender limits transmission:

$$\texttt{LastByteSent - LastByteAcked} \leq \texttt{min\{cwnd,rwnd\}}$$

- **`cwnd`** is dynamic, function of perceived network congestion

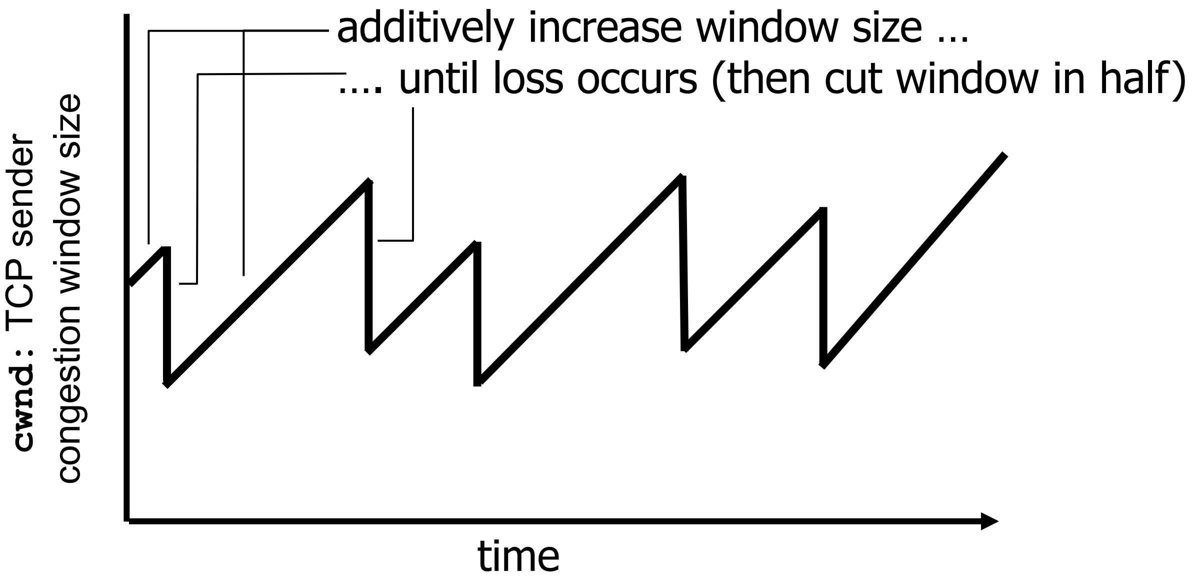
TCP Congestion Control: Self-Clocking

- A loss event as an indication of congestion
  - either a timeout or the receipt of three duplicate ACKs
- Arrival of ACKs of previously unacked segments as an indication that all is well
  - Increase congestion window size (and transmission rate) more quickly if ACKs arrive at a high rate

TCP Congestion Control: AIMD

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS (maximum segment size) every RTT until loss detected
  - *multiplicative decrease*: cut **`cwnd`** in half after loss

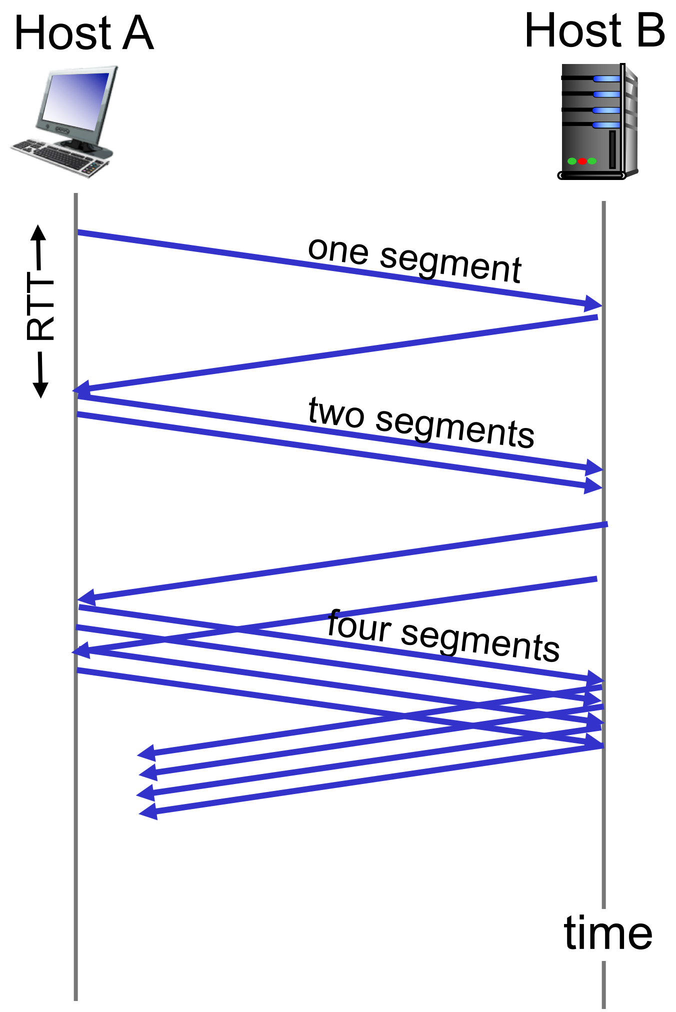
AIMD saw tooth behavior: probing for bandwidth



Each sender acts on local information asynchronously

Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** by 1 MSS every time a segment is first ACKed
- *summary:* initial rate is slow but ramps up exponentially fast



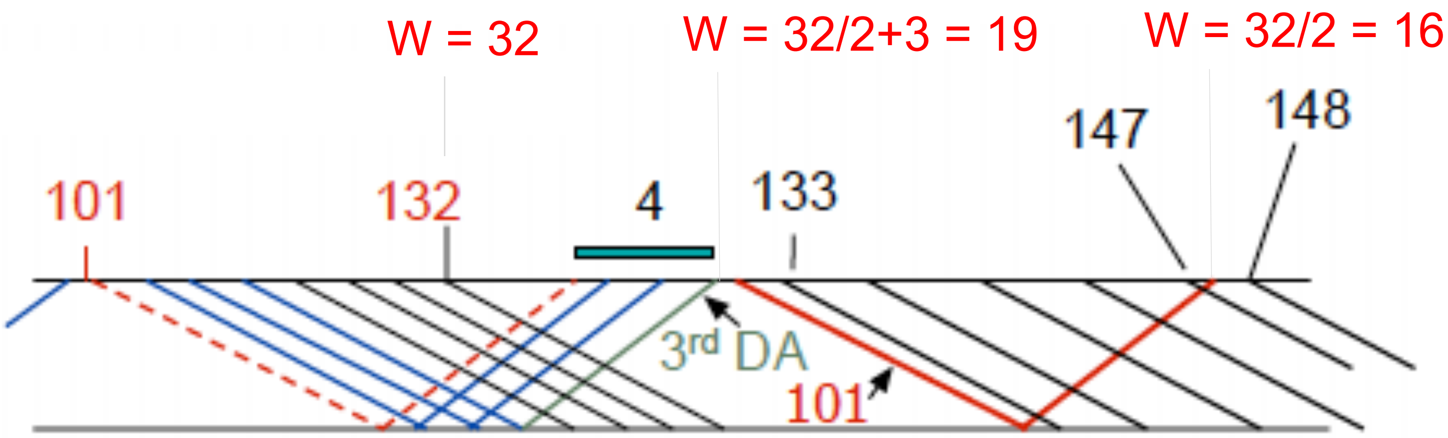
Congestion Avoidance

- loss indicated by timeout:
  - set **`ssthresh`** = **`cwnd`**/2, **`cwnd`** = 1 MSS
  - window then grows exponentially (as in slow start) to **`ssthresh`**
  - window then grows linearly (congestion avoidance)
    - increase **`cwnd`** by 1 MSS every RTT
    - done by incrementing **`cwnd`** by MSS × (MSS/**`cwnd`**) bytes for every new ACK received

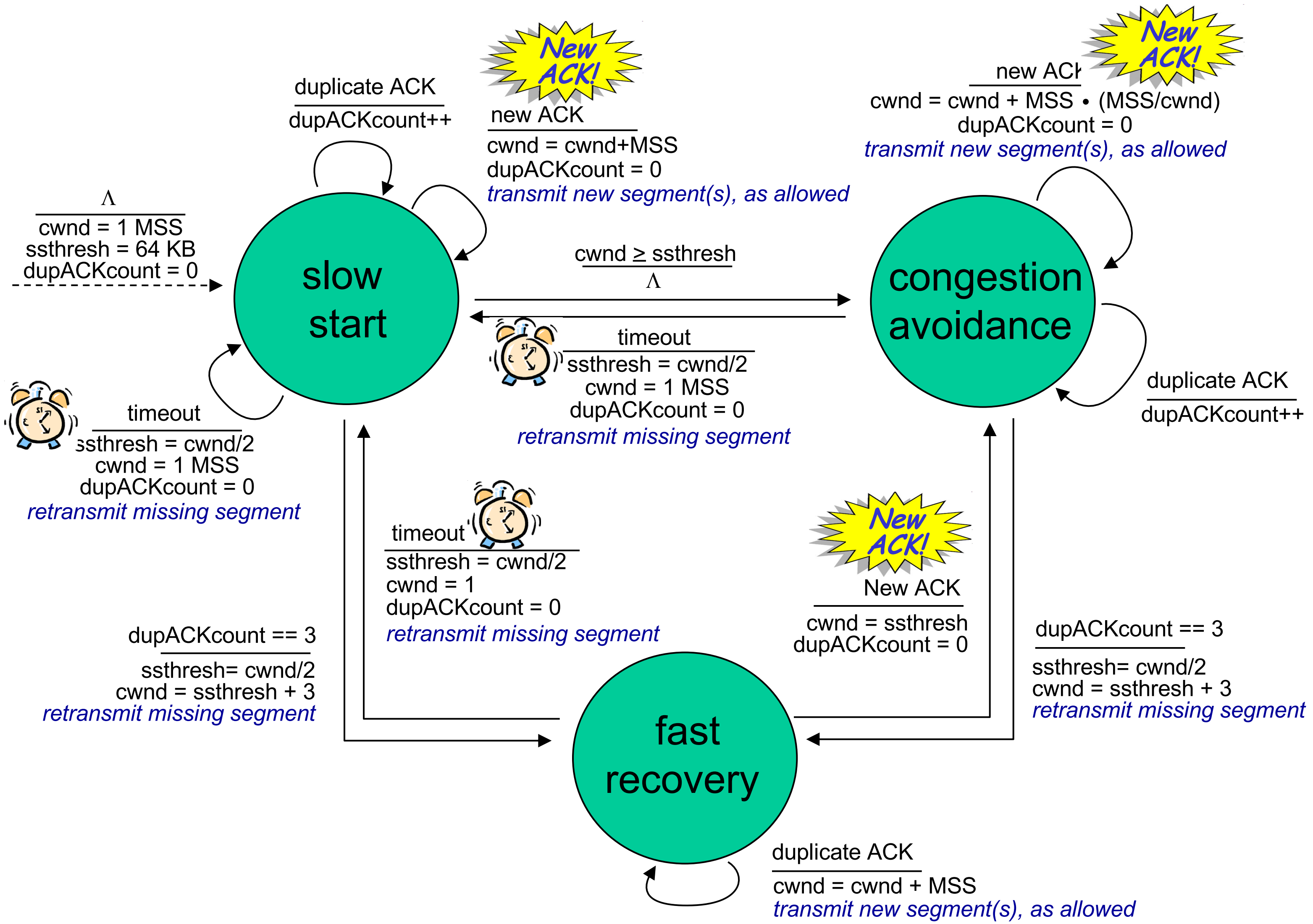
Fast Recovery

- loss indicated by 3 duplicate ACKs:
  - dup ACKs indicate network capable of delivering some segments
  - set **`ssthresh`** = **`cwnd`**/2, **`cwnd`** = **`cwnd`**/2 + 3 MSS
  - then increase **`cwnd`** by 1 MSS for every duplicate ACK received
  - when an ACK arrives for the missing segment, set **`cwnd`** = **`ssthresh`**, enters congestion avoidance
  - Implemented in TCP Reno
- TCP Tahoe always sets **`cwnd`** to 1 (timeout or 3 duplicate acks)

Fast Recovery: Example

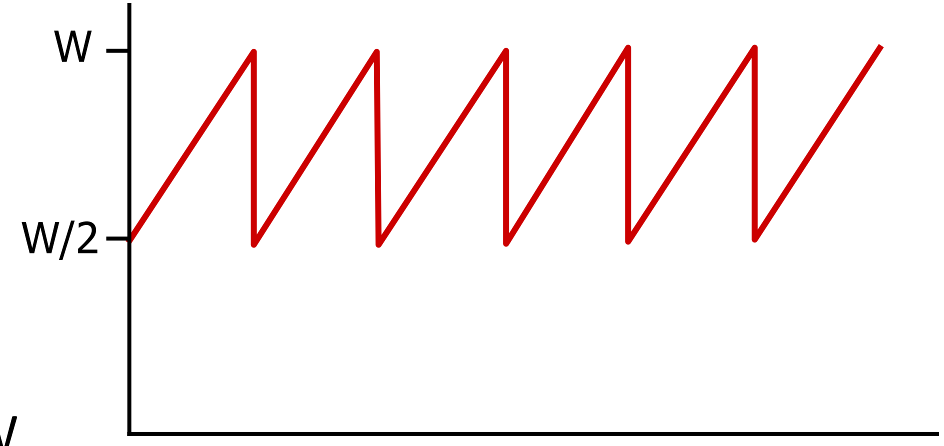


Summary: TCP Congestion Control



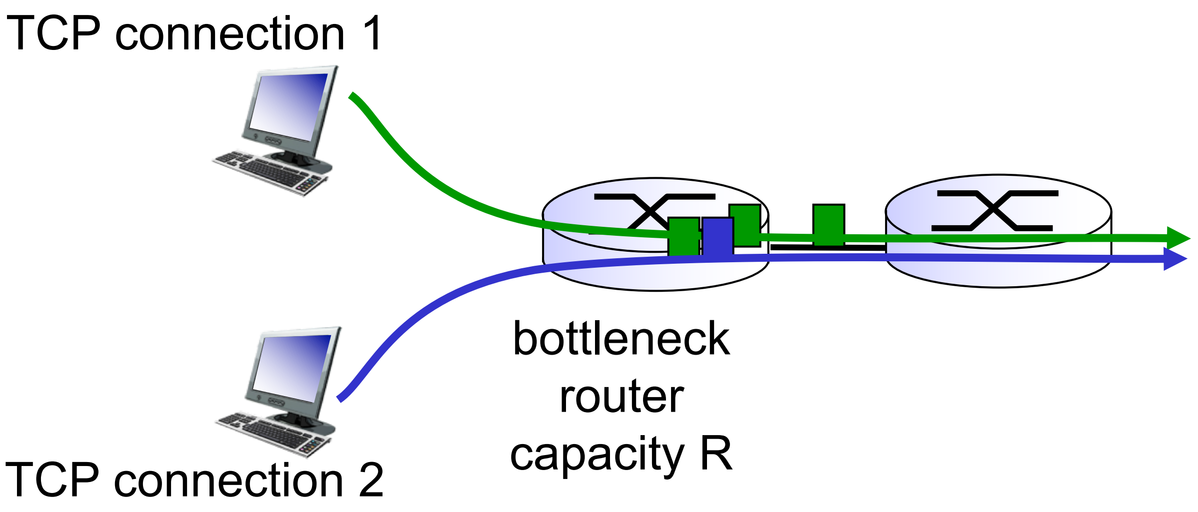
TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) when loss occurs
  - When loss occurs, window is cut in half and then increases by MSS every RTT until it again reaches W
  - avg. window size is 0.75 W
  - avg TCP thruput = $\frac{0.75W}{RTT}$ bytes/sec



TCP Fairness

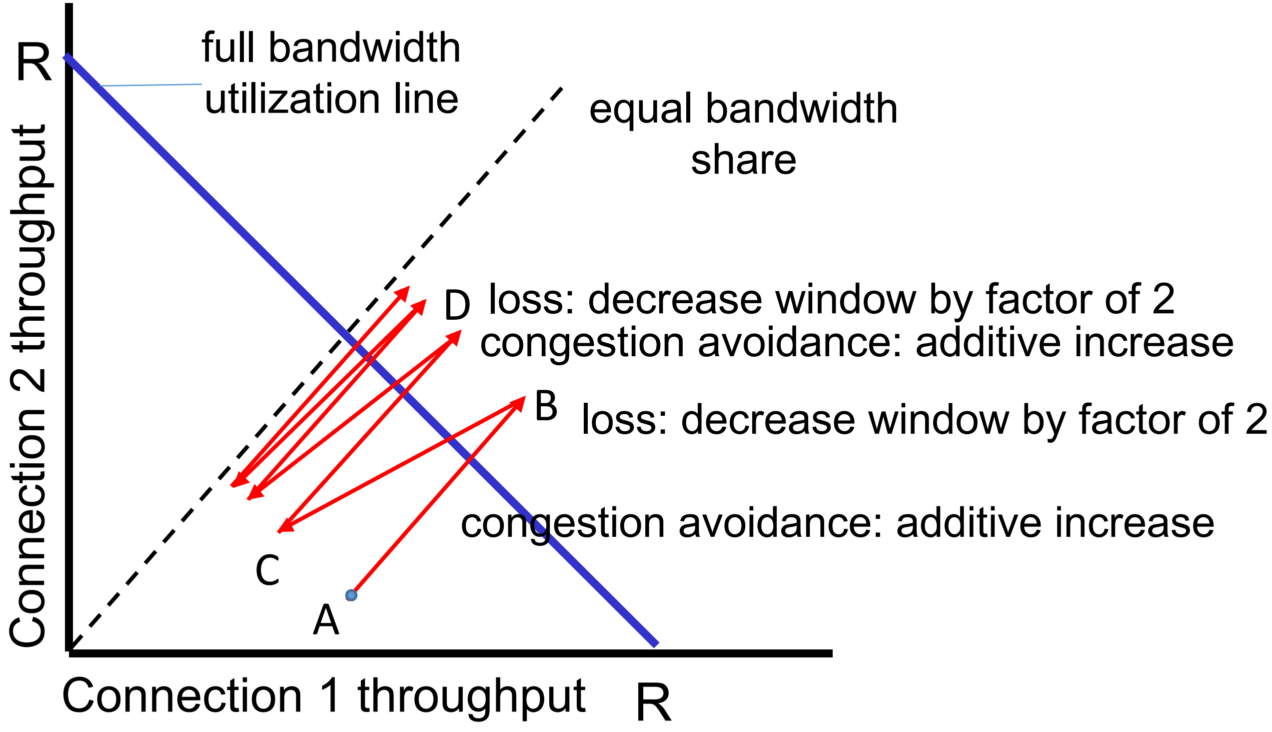
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

two competing sessions with same MSS and RTT:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Summary

- Transport-layer services
  - service model, multiplexing/demultiplexing
- Connectionless Transport: UDP
  - checksum
- Principles of reliable data transfer
  - rdt 3.0, GBN, SR
- Connection-Oriented Transport: TCP
  - reliable data transfer, flow control, connection setup
- TCP congestion control
  - TCP Reno, throughput, fairness