Chapter 3 Transport Layer

Layering in Internet protocol stack

Applications

… built on …

Reliable (or unreliable) transport

… built on …

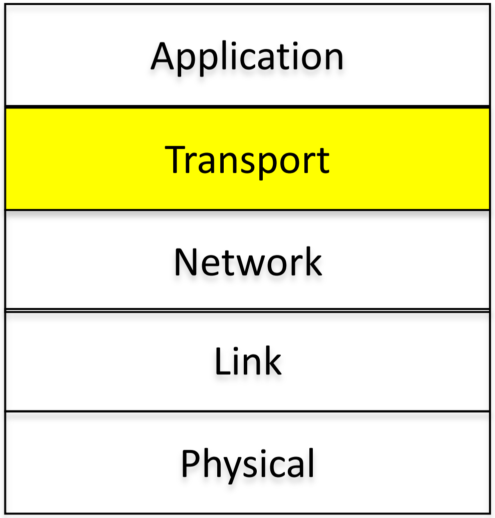
Best-effort global packet delivery

… built on …

Best-effort local packet delivery

… built on …

Physical transfer of bits



Application

Transport

Network

Link

Physical

Modified from Scott Shenker (UC Berkeley): The Future of Networking, and the Past of Protocols

Chapter 3: Our Goals

- understand principles behind transport layer services:
  - multiplexing, de-multiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Learn about transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

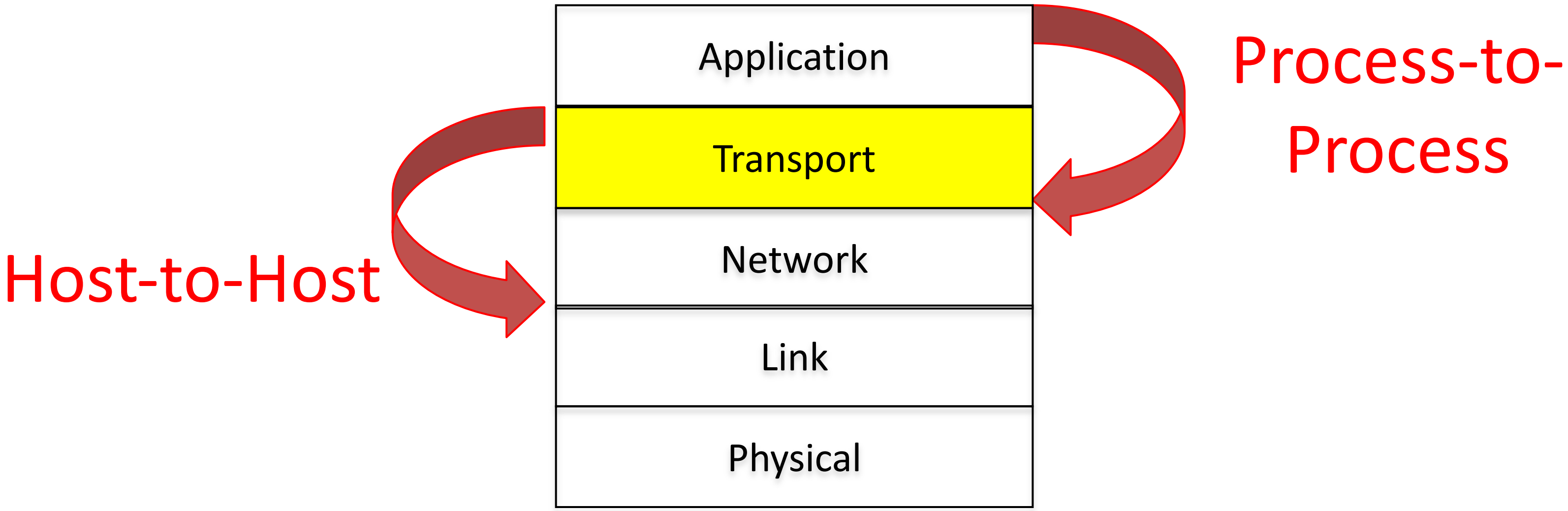
3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Upper and Lower Layers



Application

Transport

Network

Link

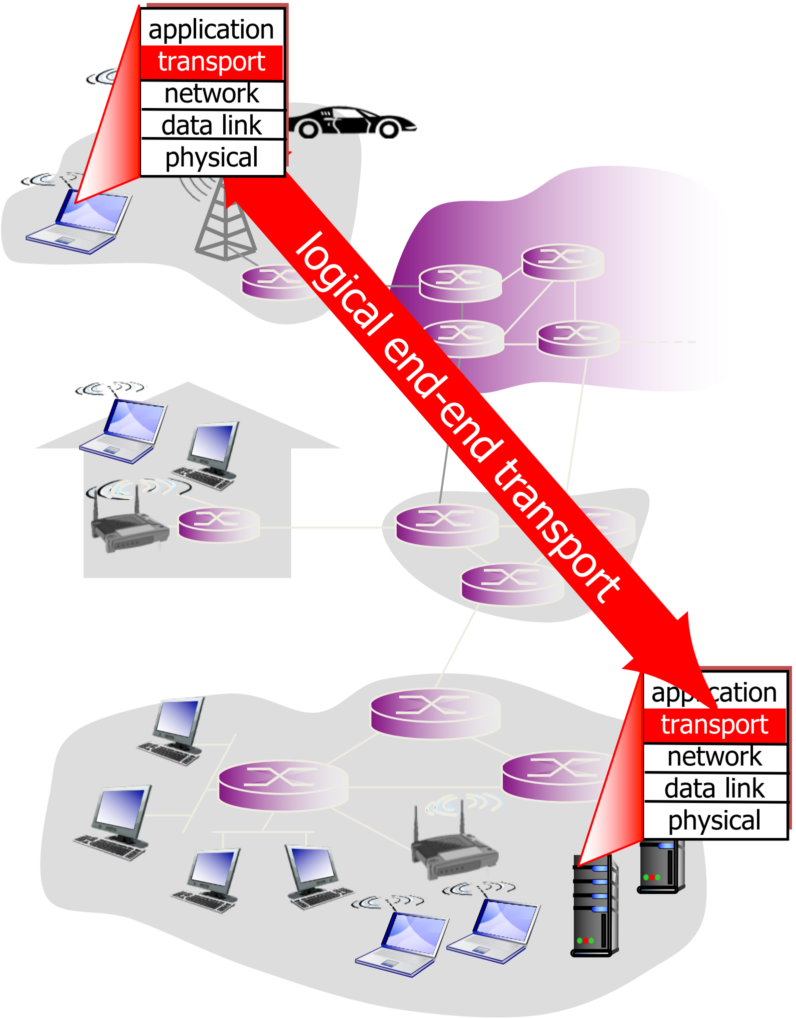
Physical

Process-to-Process

Host-to-Host

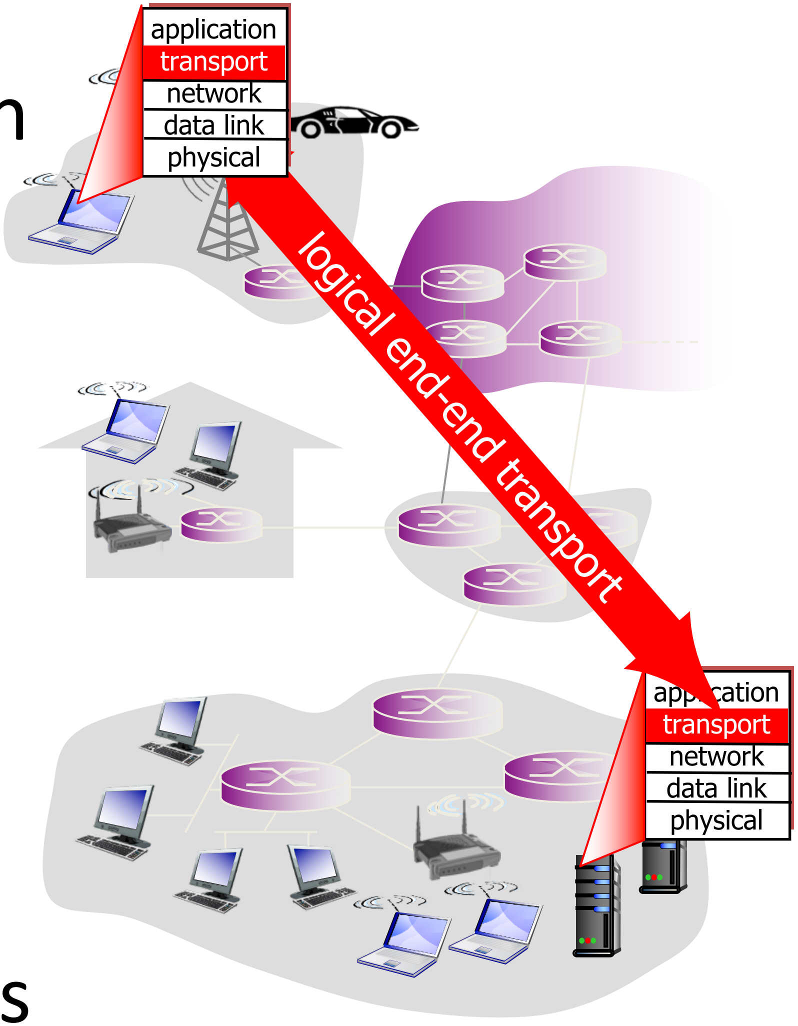
Transport services and protocols

- provide *logical communication* between app processes running on different hosts



Transport services and protocols

- 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

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

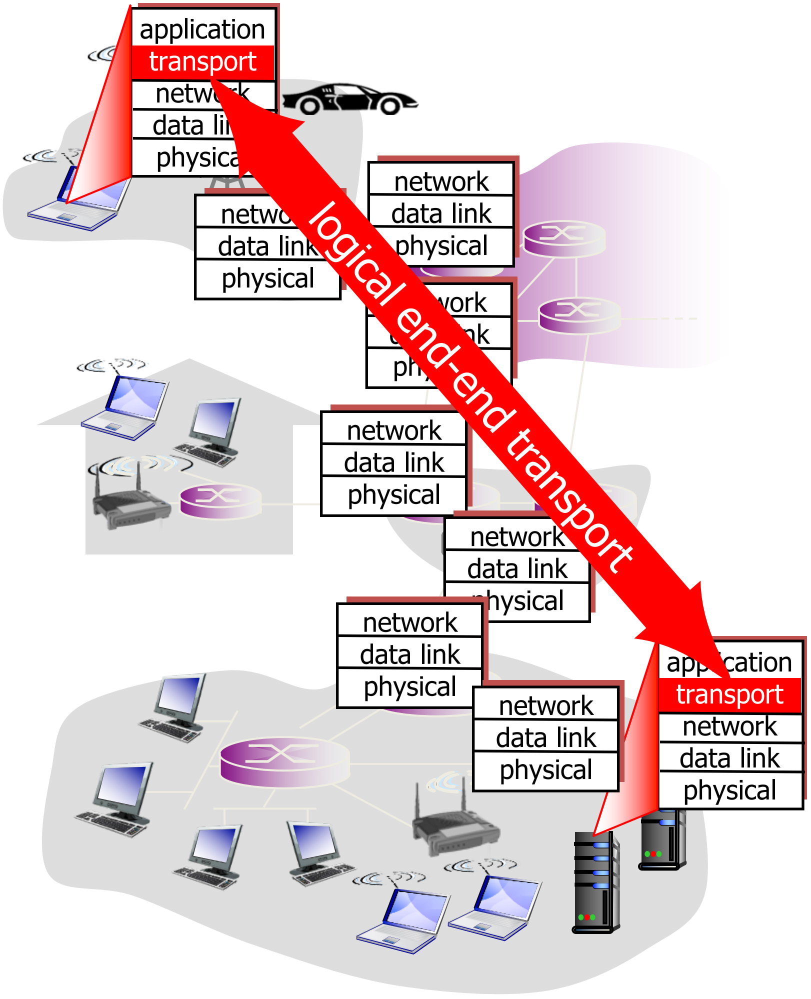
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

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



Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

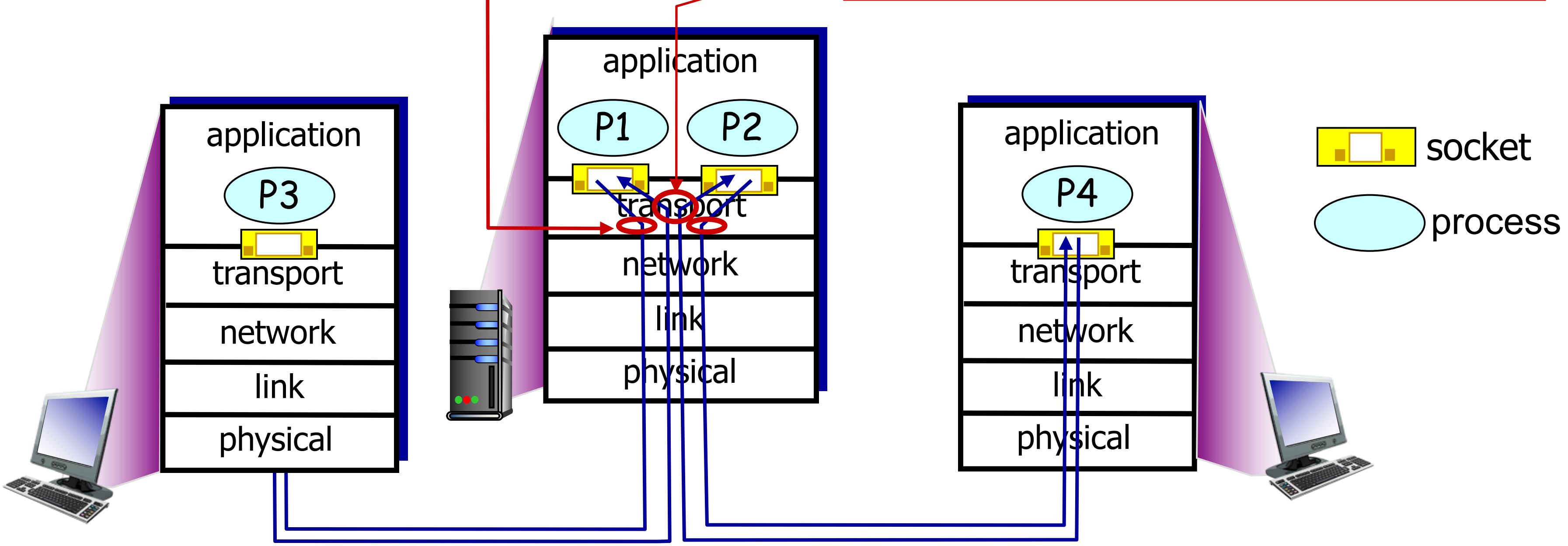
3.6 Principles of congestion control

3.7 TCP congestion control

Multiplexing/demultiplexing

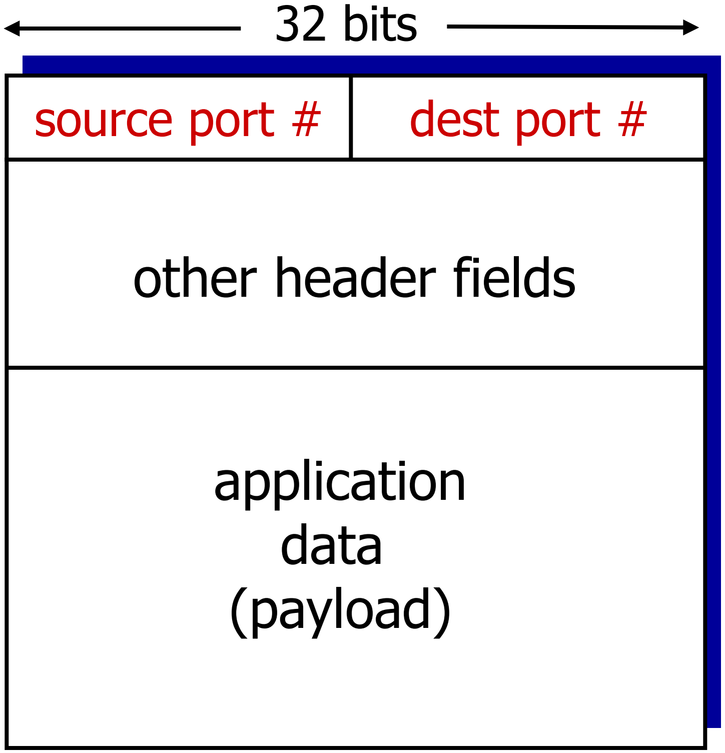
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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



How demultiplexing works

- 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:* create and bind a socket:

```
sockfd = socket(AF_INET , SOCK_DGRAM,1)  //
serv_addr.sin_port = htons(portno);  // local port # is specified
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

Note: both destination IP address and destination port # are used

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

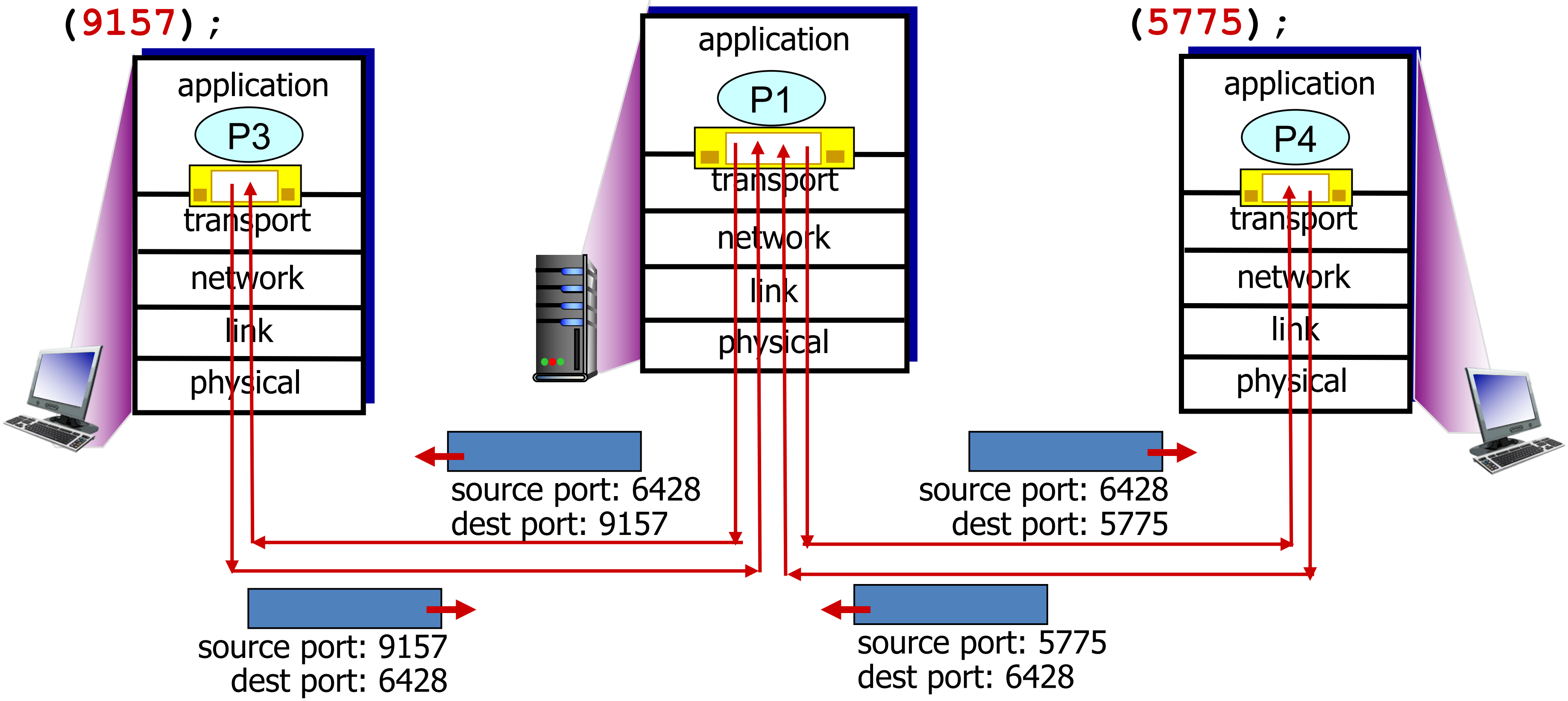
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
  (9157);
```

```
DatagramSocket
  serverSocket = new
  DatagramSocket
  (6428);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



source port: 6428
dest port: 9157

source port: 6428
dest port: 5775

source port: 9157
dest port: 6428

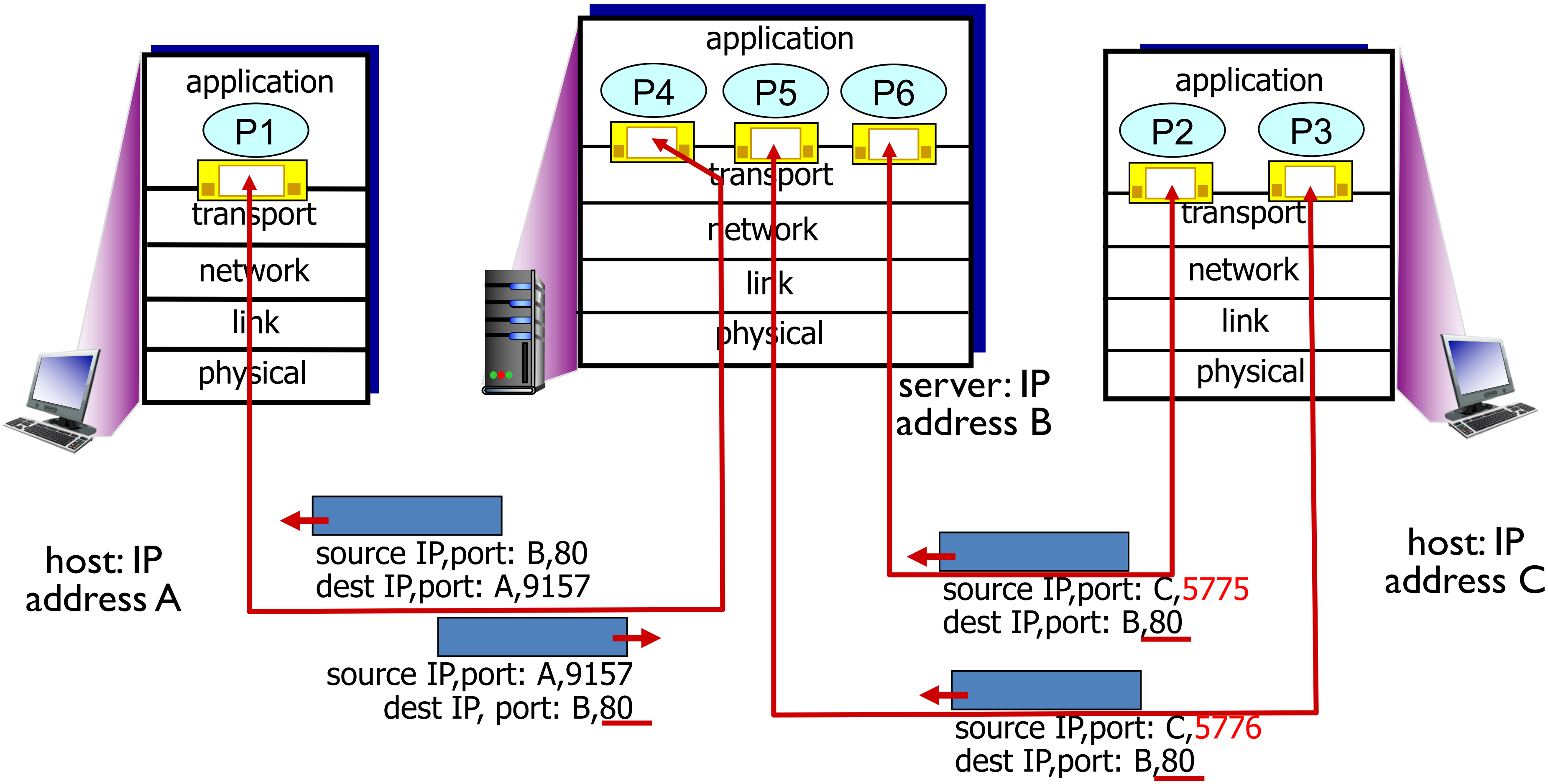
source port: 5775
dest port: 6428

Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

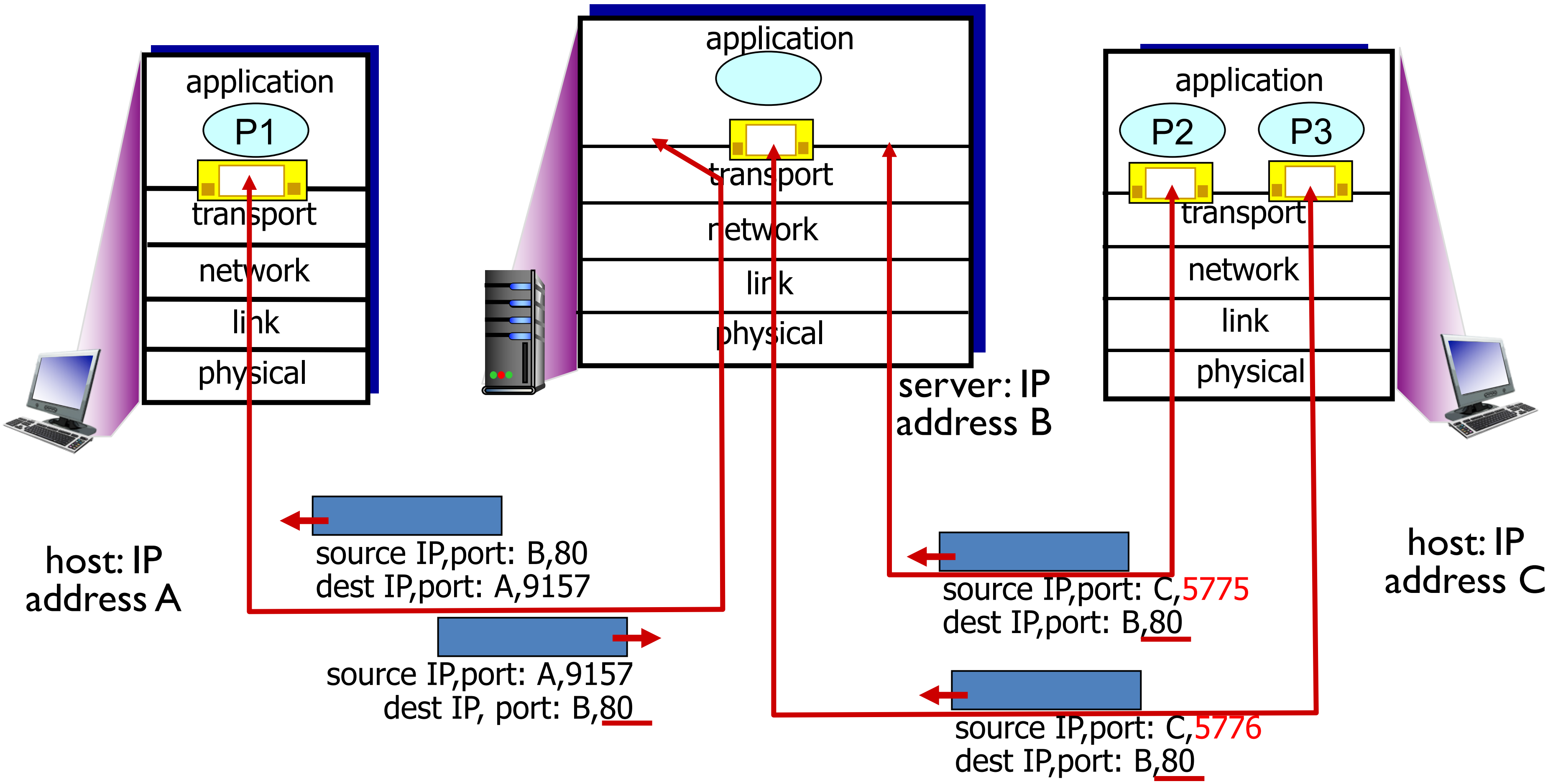
Connection-oriented demux: example



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

Question: What if UDP

How many sockets on the server side?



Question:
why no checking of dest IP?

- check port# only in the UDP example
- Check port# + source IP in the TCP example
  - Correctness of IP address is ensured on the Networking layer
  - Destination IP: (not delivered to the node)

Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

UDP: User Datagram Protocol

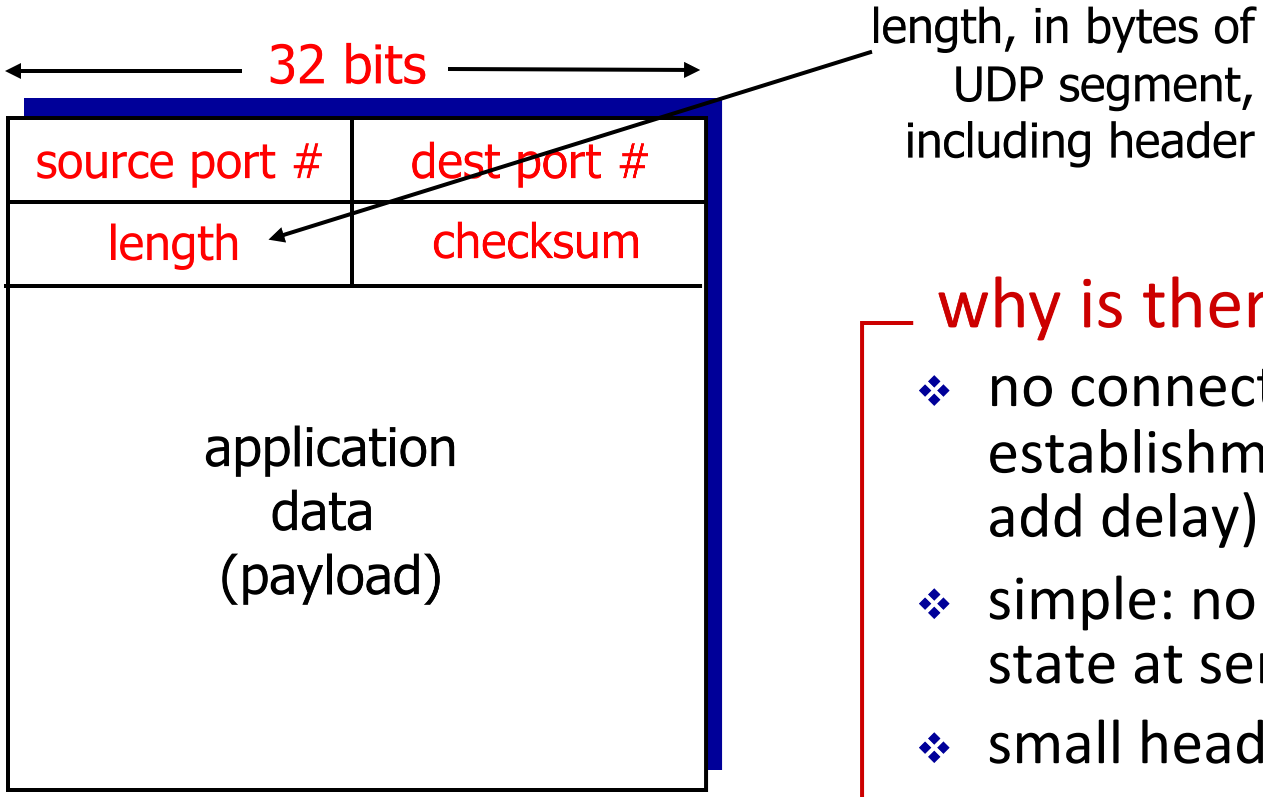
[RFC 768]

- "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

UDP: User Datagram Protocol

- UDP usage:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP?
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

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

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 nonetheless?* More later ….

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
```

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

After-class practice: UDP checksum

- 1st: 0110
- 2nd: 0101
- 3rd: 1000
- Calculate UDP checksum of 1st + 2nd + 3rd
- sum = 10011, -> 0011 + 1 (carryout) = 0100
- checksum = 1s complement = 1011
- Check: receiving 1011?
- Check: receiving 1001?
- Errors if receiving 1011??
  - See the notes for this slide for answers

Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

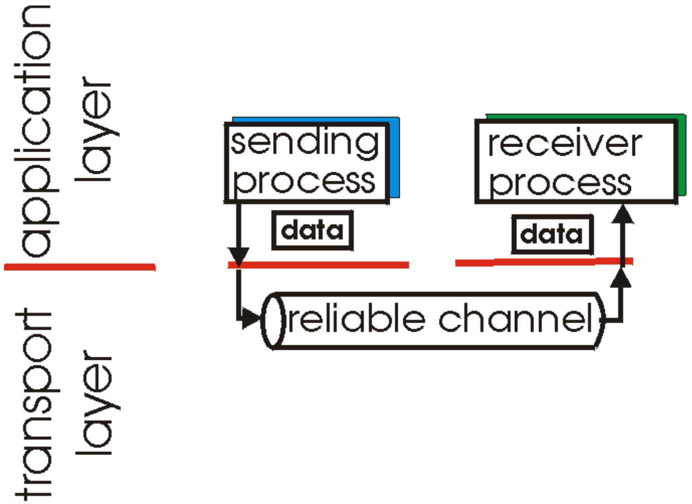
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

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

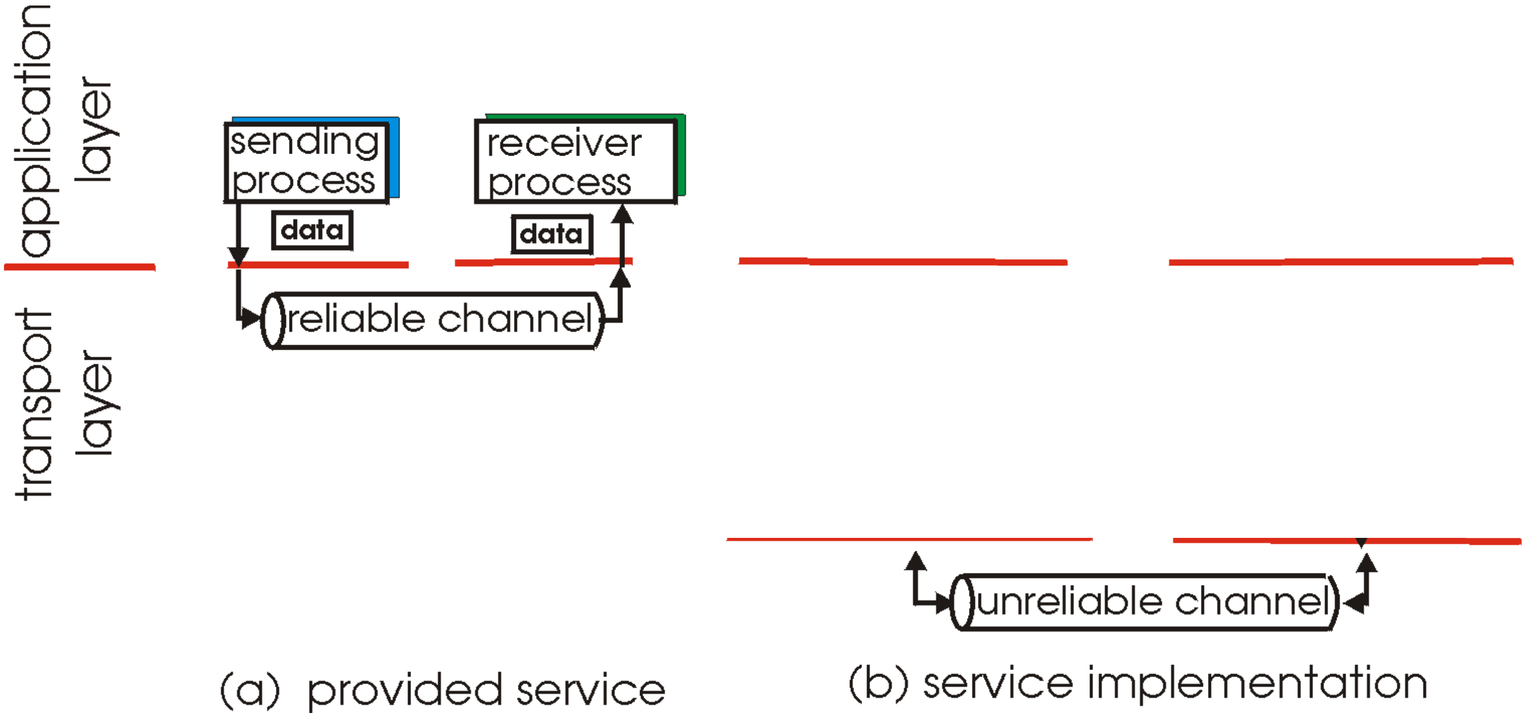


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

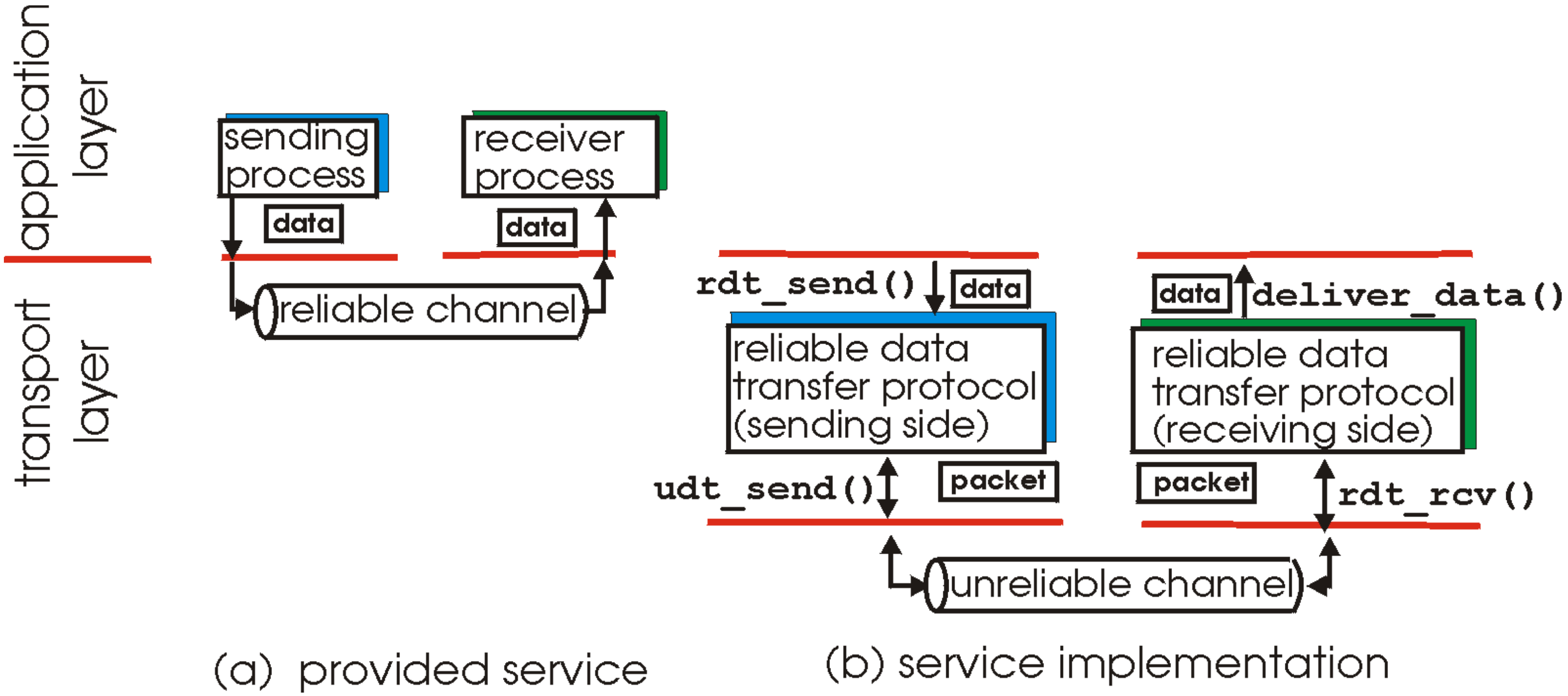
- important in application, transport, link layers
  - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

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

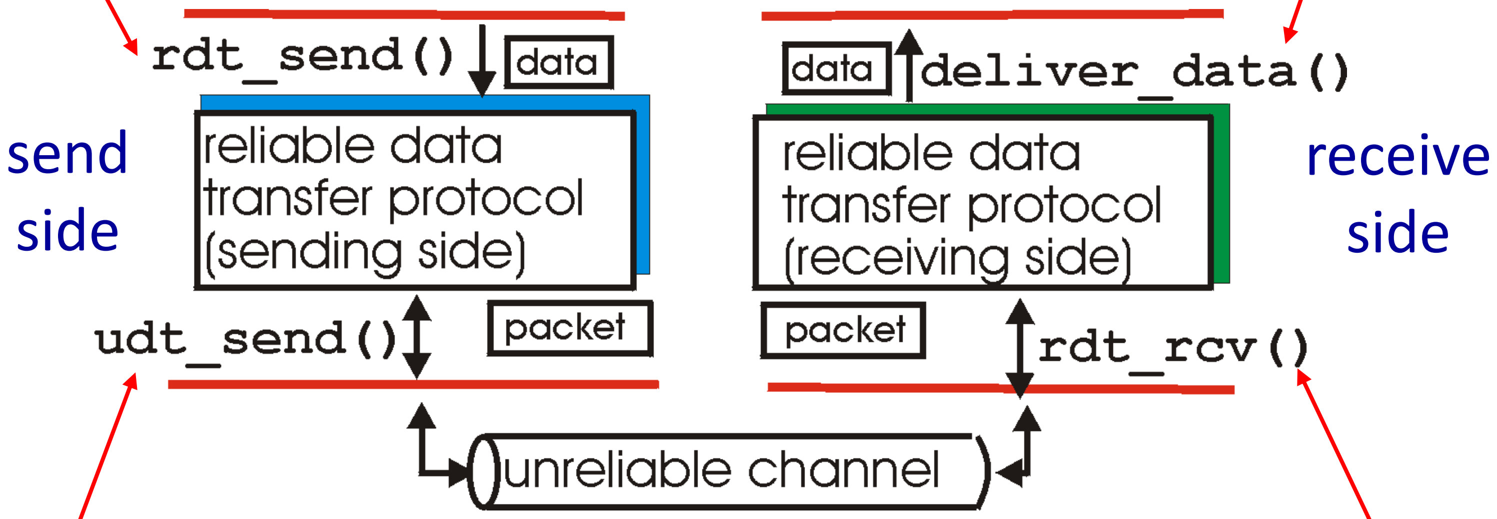


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

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

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



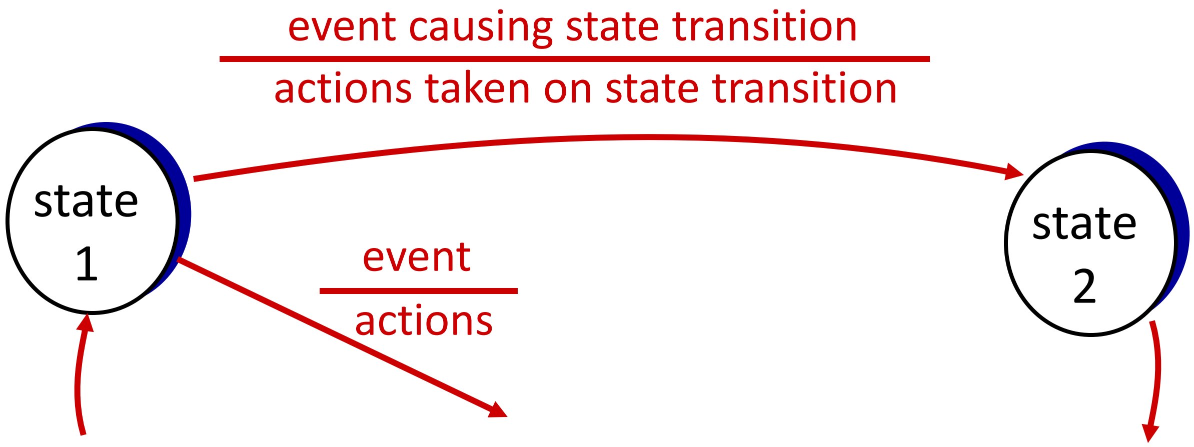
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

we' ll:

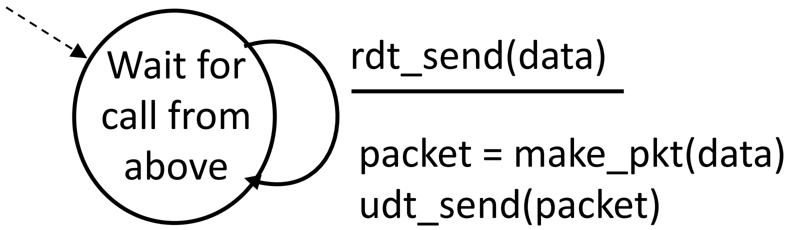
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver


event causing state transition
actions taken on state transition
state 1
event
actions
state 2

state: when in this "state" next state uniquely determined by next event

rdt1.0: reliable transfer over a reliable channel

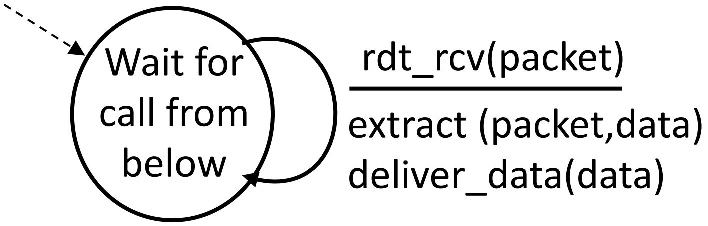
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel


Wait for call from above

rdt_send(data)

packet = make_pkt(data)
udt_send(packet)

sender


Wait for call from below

rdt_rcv(packet)

extract (packet,data)
deliver_data(data)

receiver

rdt2.0: channel with bit errors

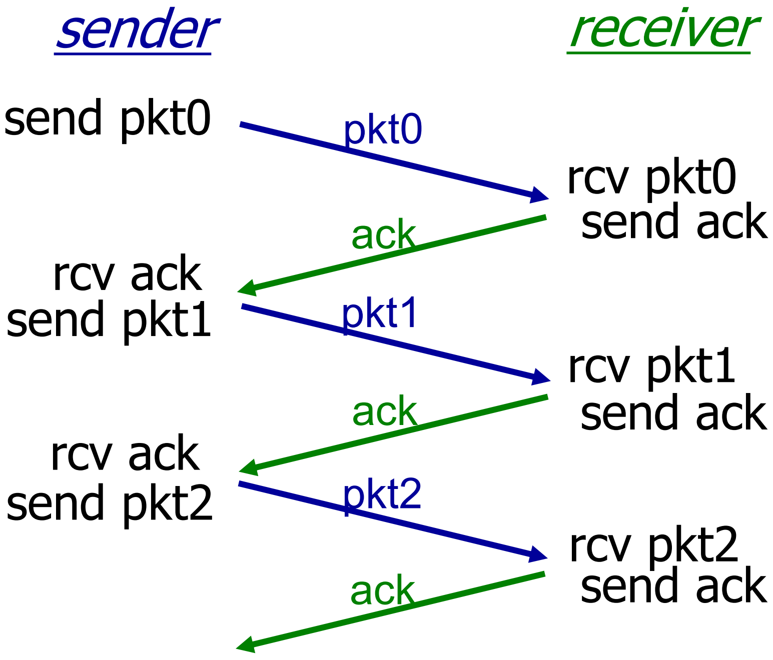
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- question: how to recover from errors?

How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- 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
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - Error detection at receiver
  - Feedback from receiver: control msgs (ACK,NAK) from receiver to sender
  - Retransmission at the sender

rdt2.0 in action



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack

ack

rcv ack
send pkt1

pkt1

rcv pkt1
send ack

ack

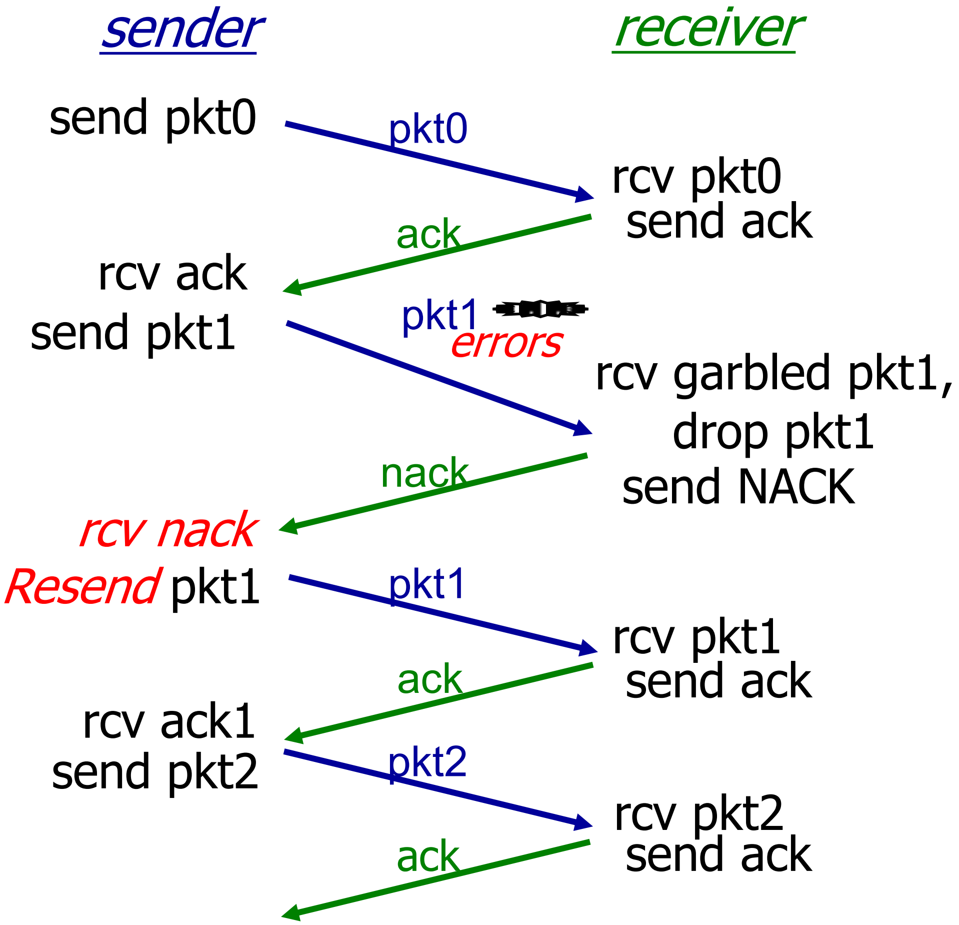
rcv ack
send pkt2

pkt2

rcv pkt2
send ack

ack

(a) no error



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack

ack

rcv ack
send pkt1

pkt1

errors

rcv garbled pkt1,
drop pkt1
send NACK

nack

rcv nack
Resend pkt1

pkt1

rcv pkt1
send ack

ack

rcv ack1
send pkt2

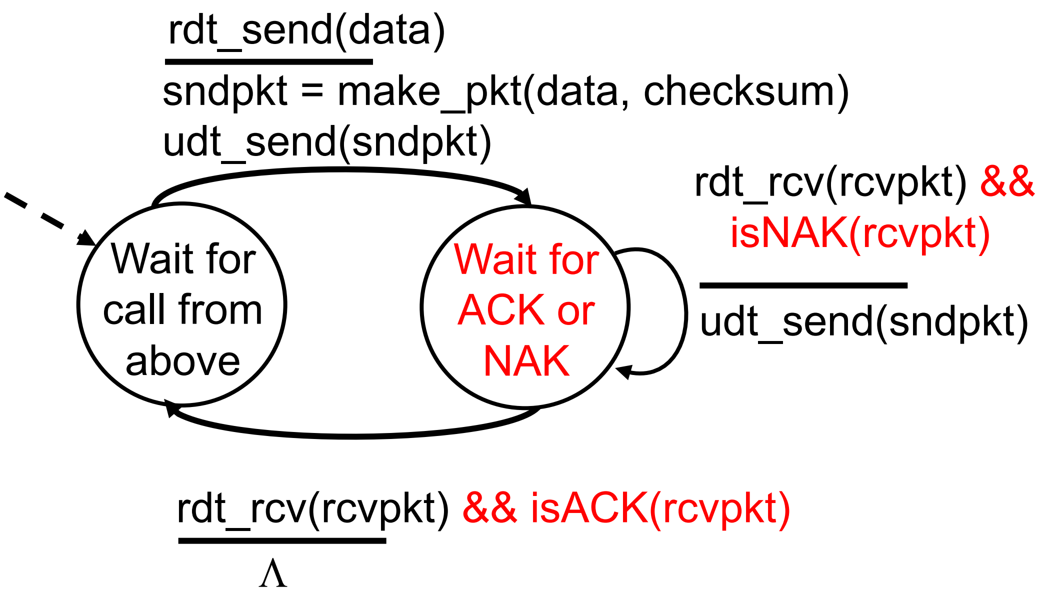
pkt2

rcv pkt2
send ack

ack

(b) packet with bit errors

rdt2.0: FSM specification



rdt_send(data)

sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

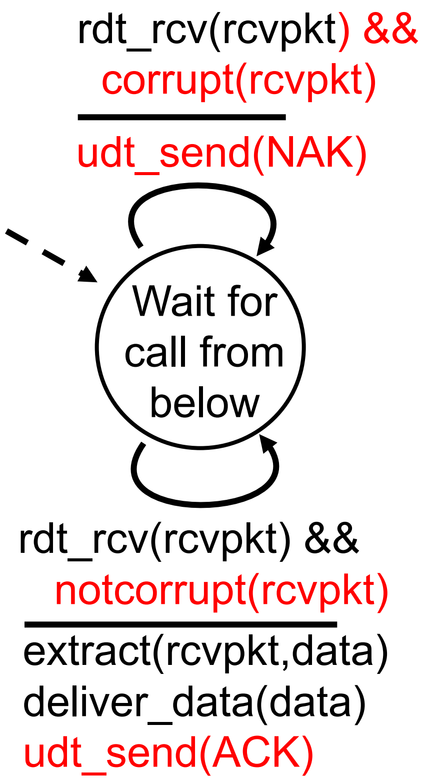
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

sender

receiver



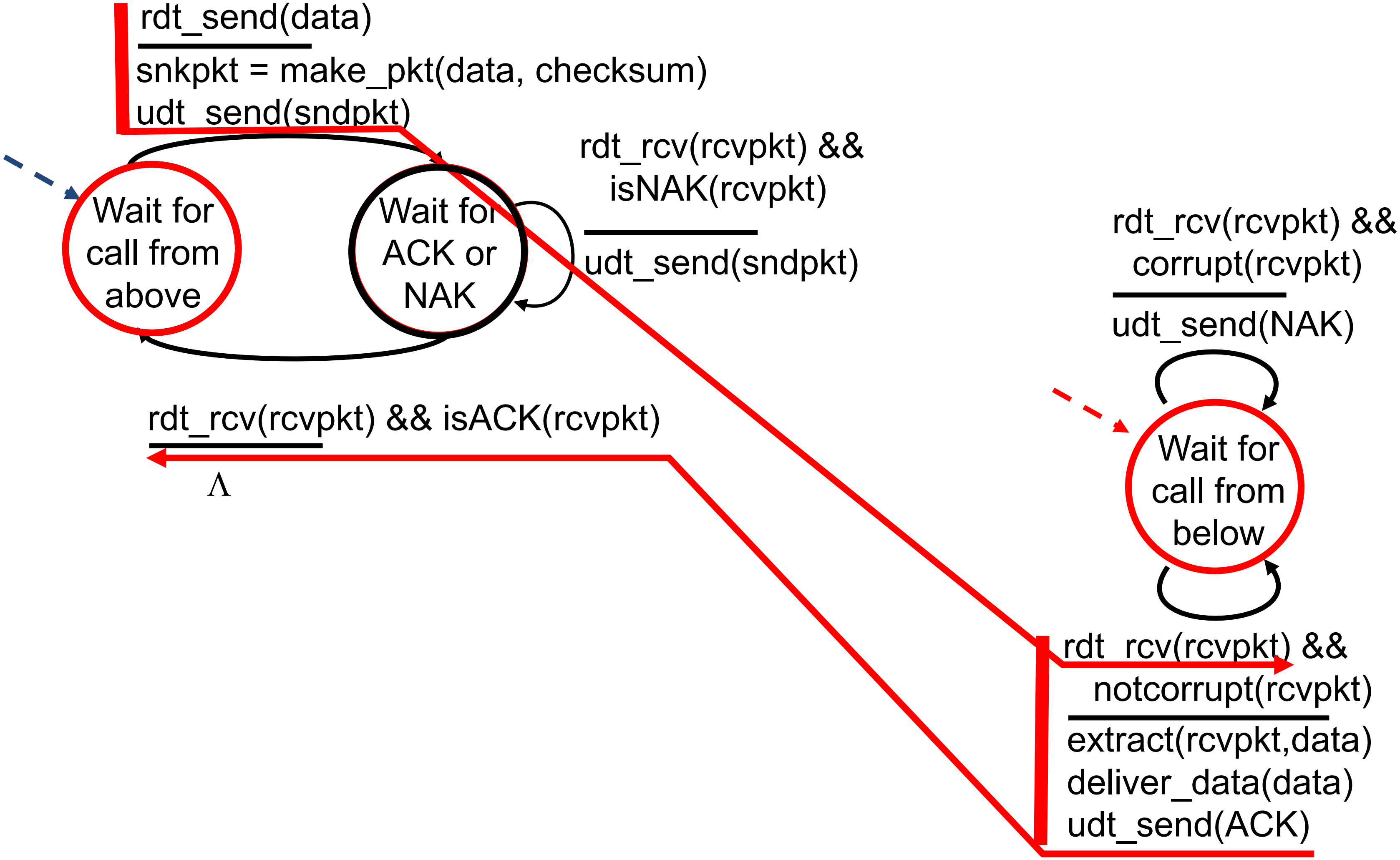
rdt_rcv(rcvpkt) && corrupt(rcvpkt)

udt_send(NAK)

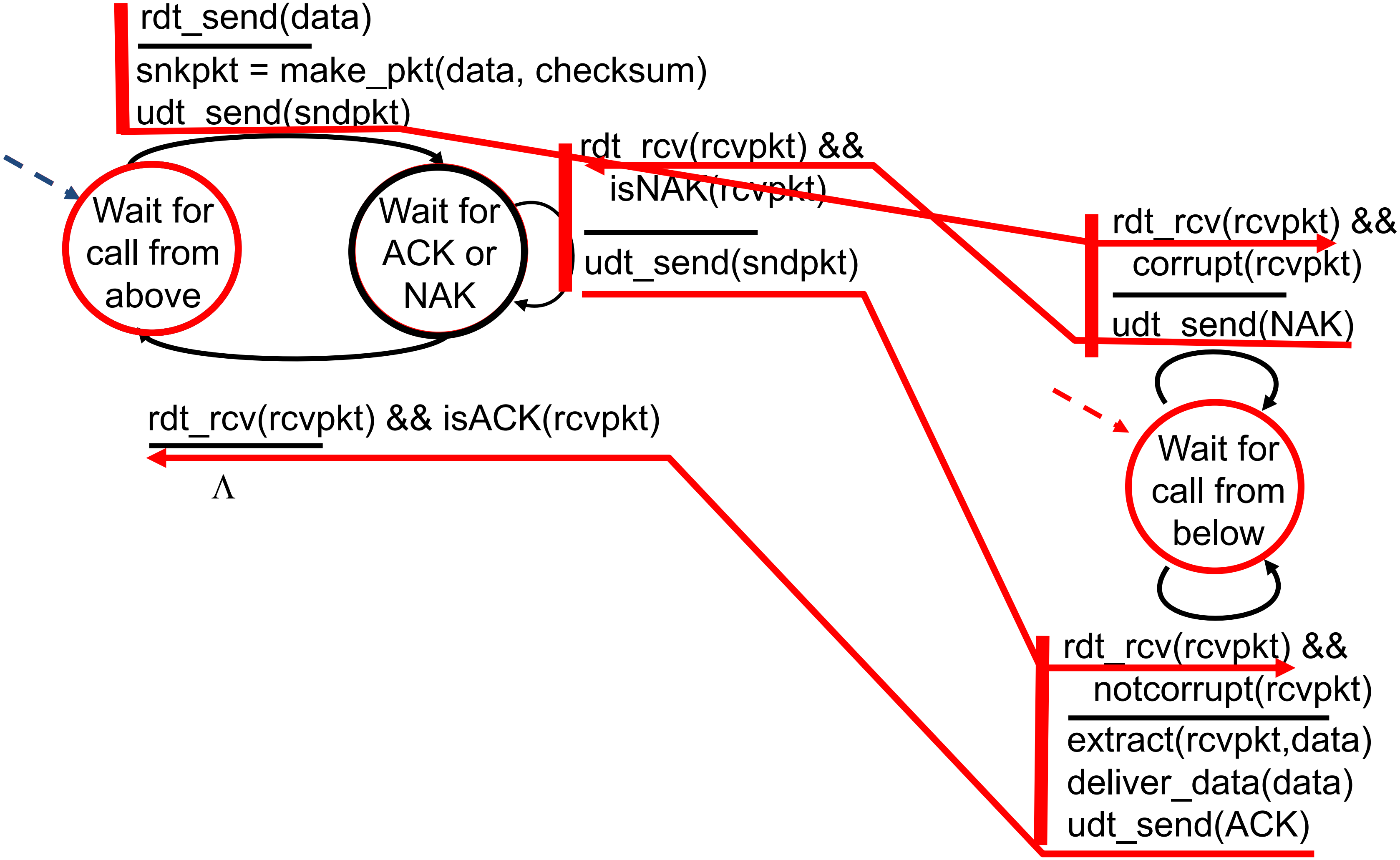
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: operation with no errors



rdt2.0: error scenario



rdt_send(data)

snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

udt_send(NAK)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

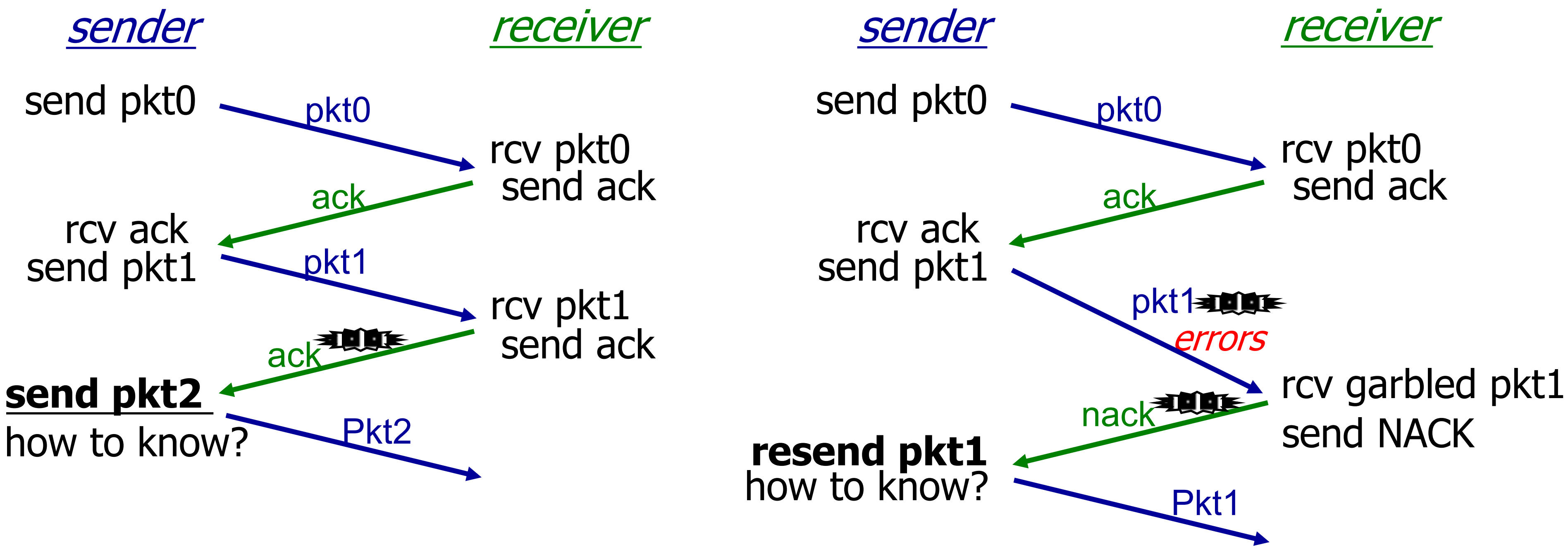
handling duplicates:

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

stop and wait

sender sends one packet, then waits for receiver response

rdt2.0's flaw: garbled ACK/NACK

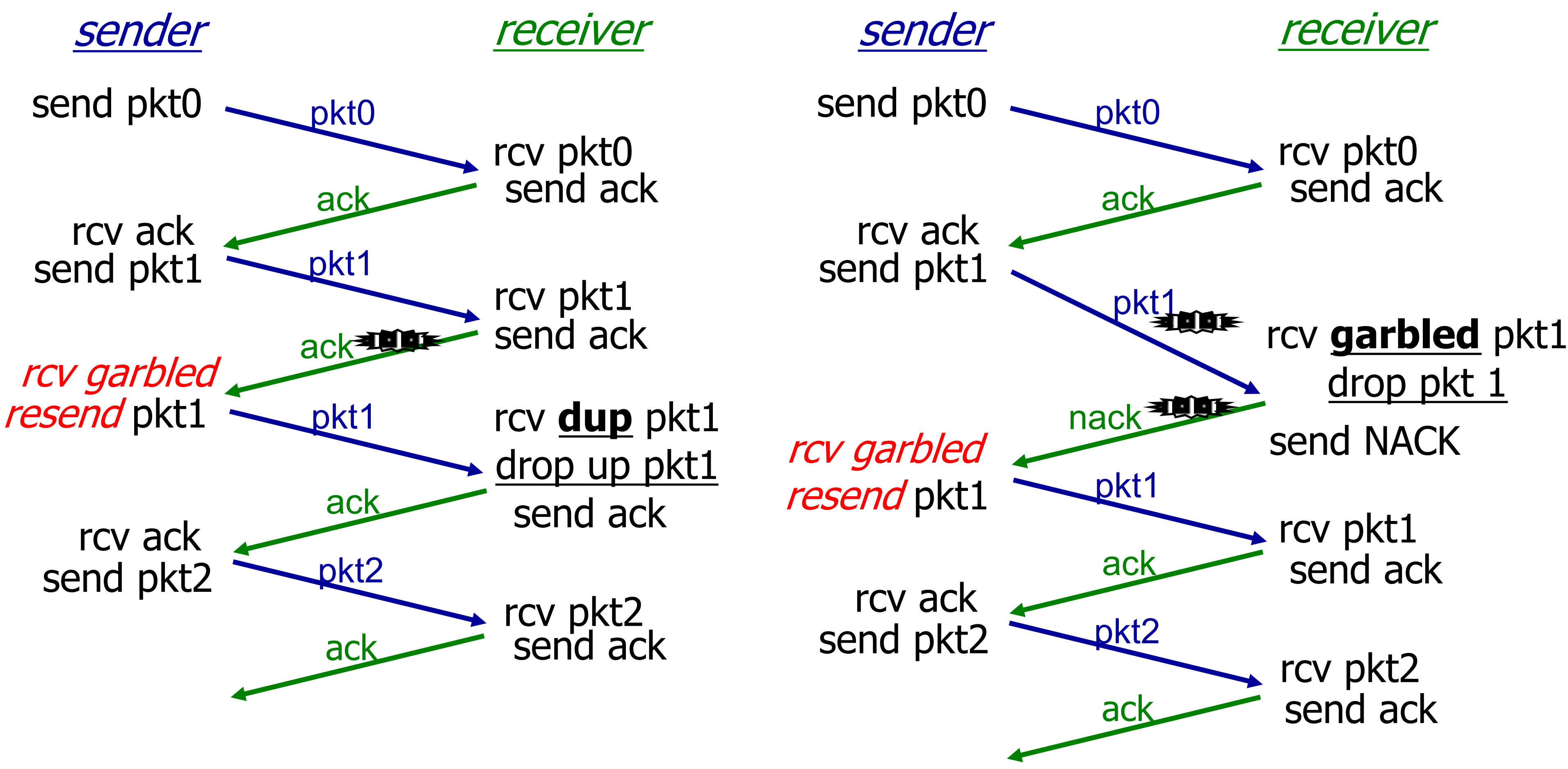


(a) Corrupted ack

(b) Corrupted NACK

Sender cannot tell whether it is corrupted ACK or NACK!

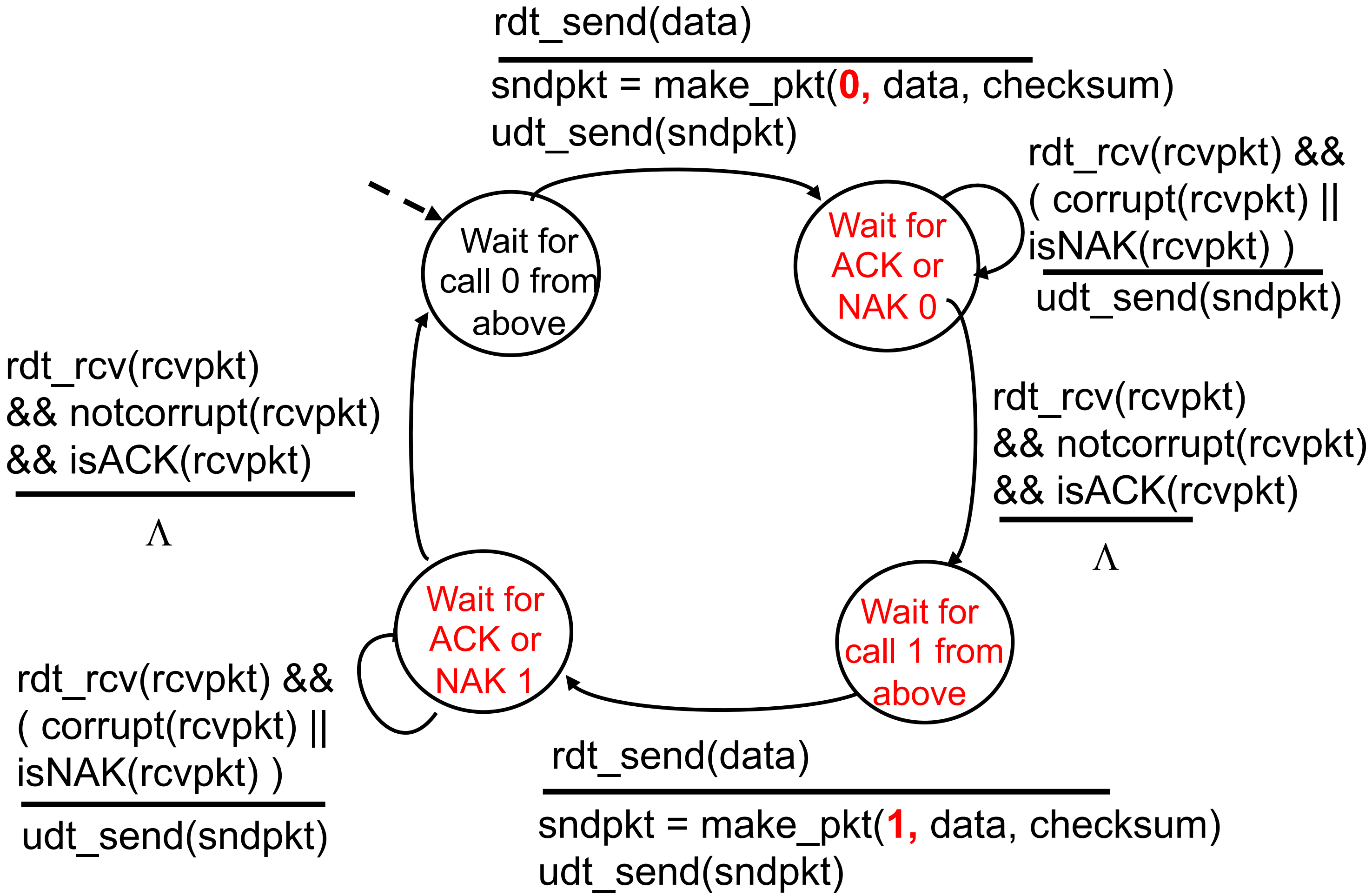
rdt2.1: need seq #!



(a) Corrupted ack

(b) Corrupted NACK

rdt2.1: sender, handles garbled ACK/NAKs



Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1

rdt_send(data)

sndpkt = make_pkt(**0,** data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt_send(data)

sndpkt = make_pkt(**1,** data, checksum)
udt_send(sndpkt)

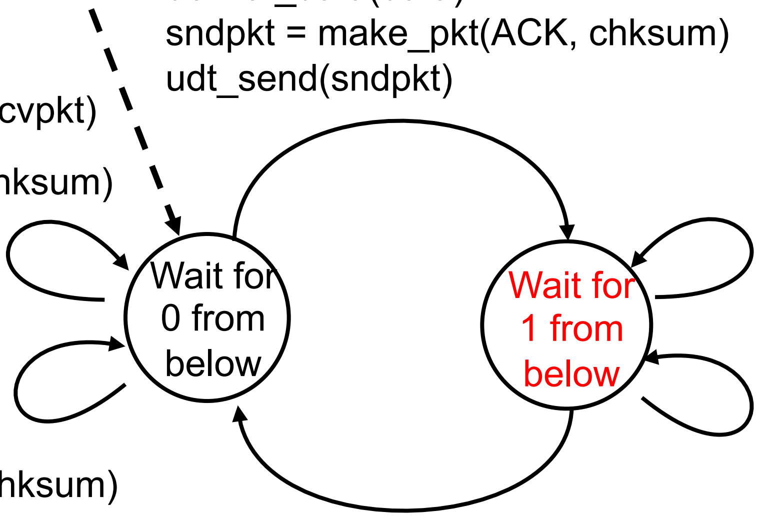
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt2.1: receiver, handles garbled ACK/NAKs



Wait for 0 from below

Wait for 1 from below

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

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

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

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

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

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

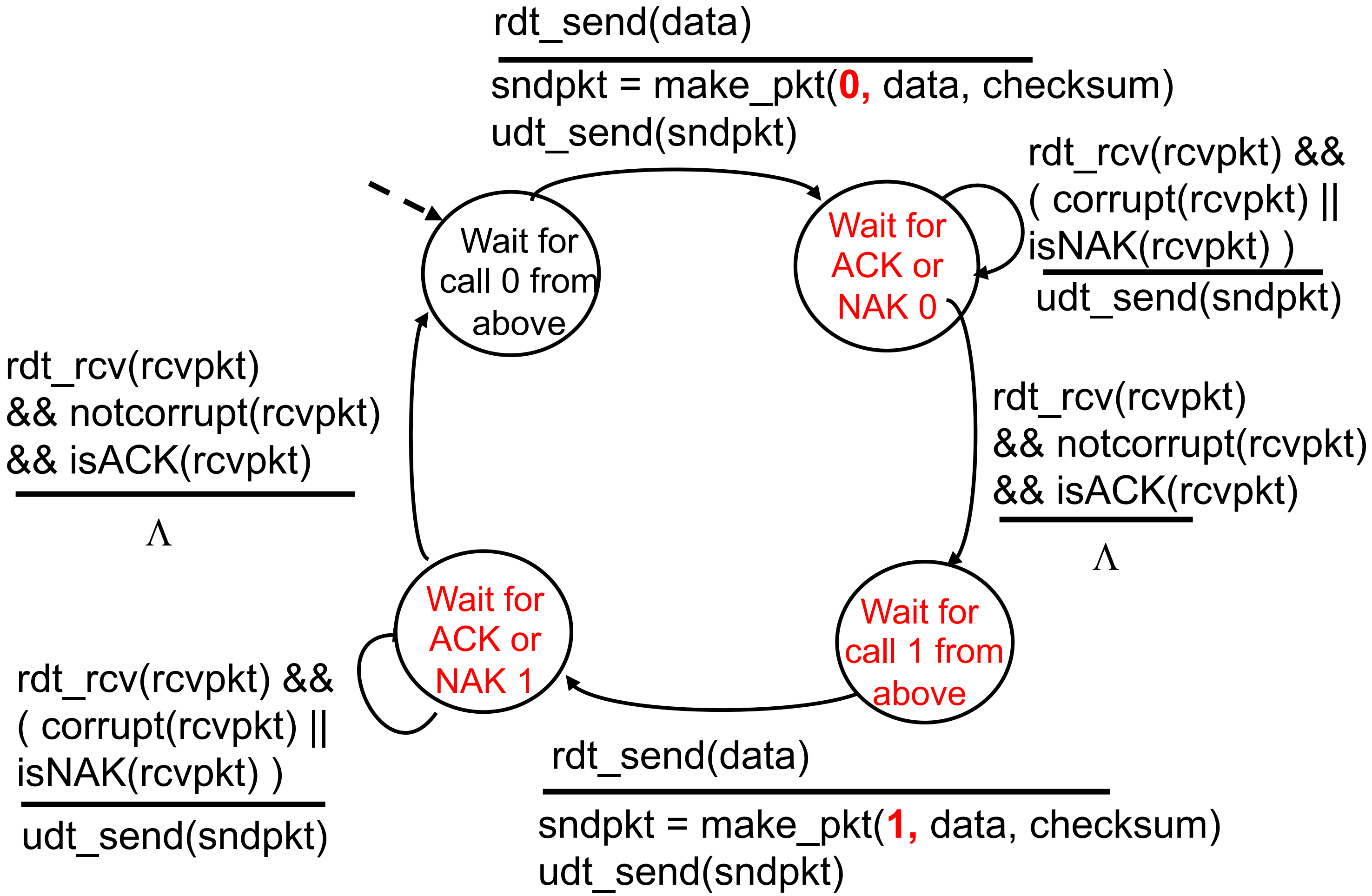
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

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

Summary: reliable data transfer

Version	Channel	Mechanism
rdt1.0	Reliable channel	nothing
rdt2.0	bit errors (no loss)	(1)error detection via checksum (2)receiver feedback (ACK/NAK) (3)retransmission
rdt2.1	Same as 2.0	handling fatal flaw with rdt 2.0: (4) need seq #.

rdt2.1: sender, handles garbled ACK/NAKs



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)

Λ

Wait for call 1 from above

rdt_send(data)

sndpkt = make_pkt(**1,** data, checksum)
udt_send(sndpkt)

Wait for ACK or NAK 1

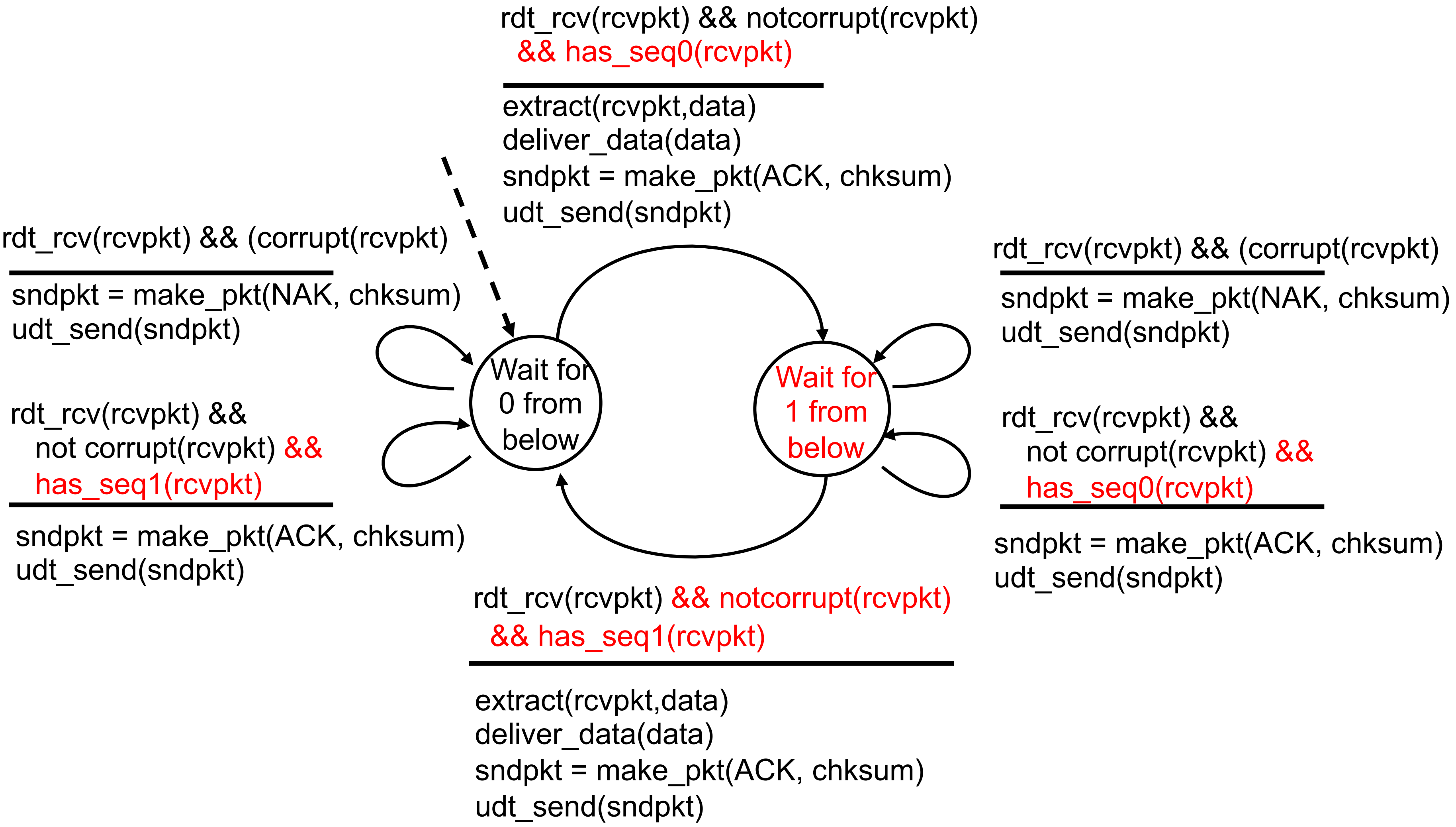
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

Λ

rdt2.1: receiver, handles garbled ACK/NAKs



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

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

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

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

Wait for
0 from
below

Wait for
1 from
below

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, 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)

Rdt2.1 discussion

- Rdt2.1 mechanisms
  - Error detection (checksum)
  - Feedback (ACK and NAK)
  - Retransmission
  - Seq number (fresh or duplicate packets)
- Q1: How many bits are needed for seq#?
  - two seq. #' s (0,1) will suffice. Why?
  - Under various scenarios to send 3 packets: (1) all ACK, no error, (2) ACK 0 (1st time) corrupted, (3) NAK 0 (1st time) corrupted, (4) ACK 0 and ACK 1, both corrupted for the first time
- Q2: Do we still need NAK? If not, how?

Rdt2.1 discussion: how many bits for a seq number?

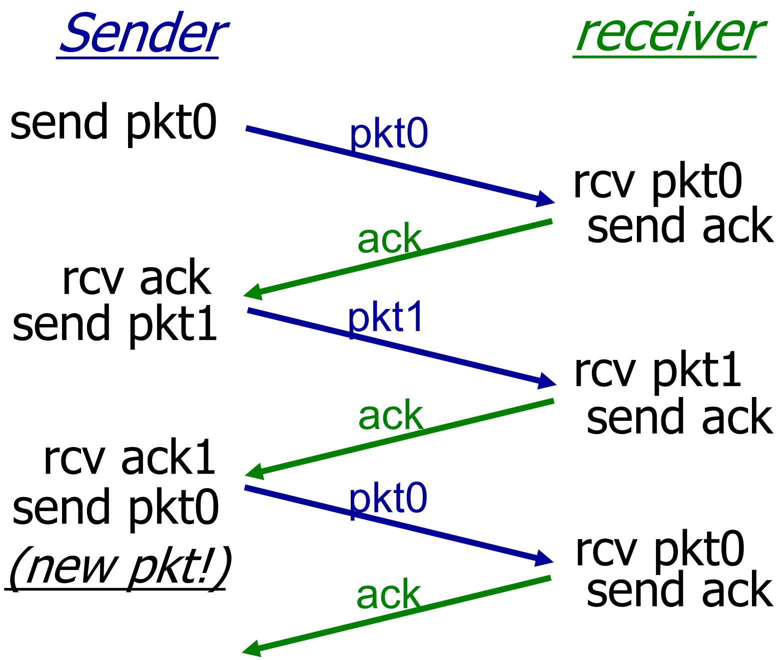
sender:

- seq # added to pkt
- two seq. #’ s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “expected” pkt should have seq # of 0 or 1

receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- Note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.1: 1-bit seq # is enough!



Sender

receiver

send pkt0

pkt0

rcv pkt0
send ack

ack

rcv ack
send pkt1

pkt1

rcv pkt1
send ack

ack

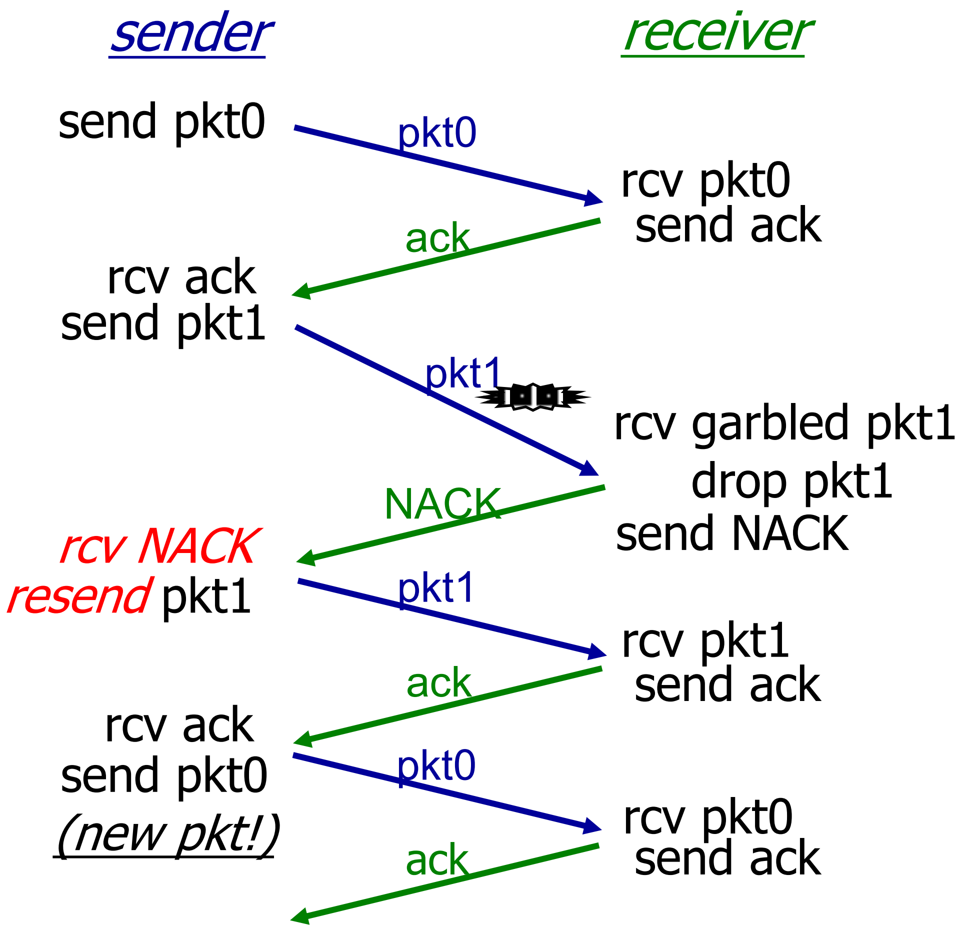
rcv ack1
send pkt0
(new pkt!)

pkt0

rcv pkt0
send ack

ack

(a) no error



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack

ack

rcv ack
send pkt1

pkt1

rcv garbled pkt1
drop pkt1
send NACK

NACK

rcv NACK
resend pkt1

pkt1

rcv pkt1
send ack

ack

rcv ack
send pkt0
(new pkt!)

pkt0

rcv pkt0
send ack

ack

(b) packet with bit errors

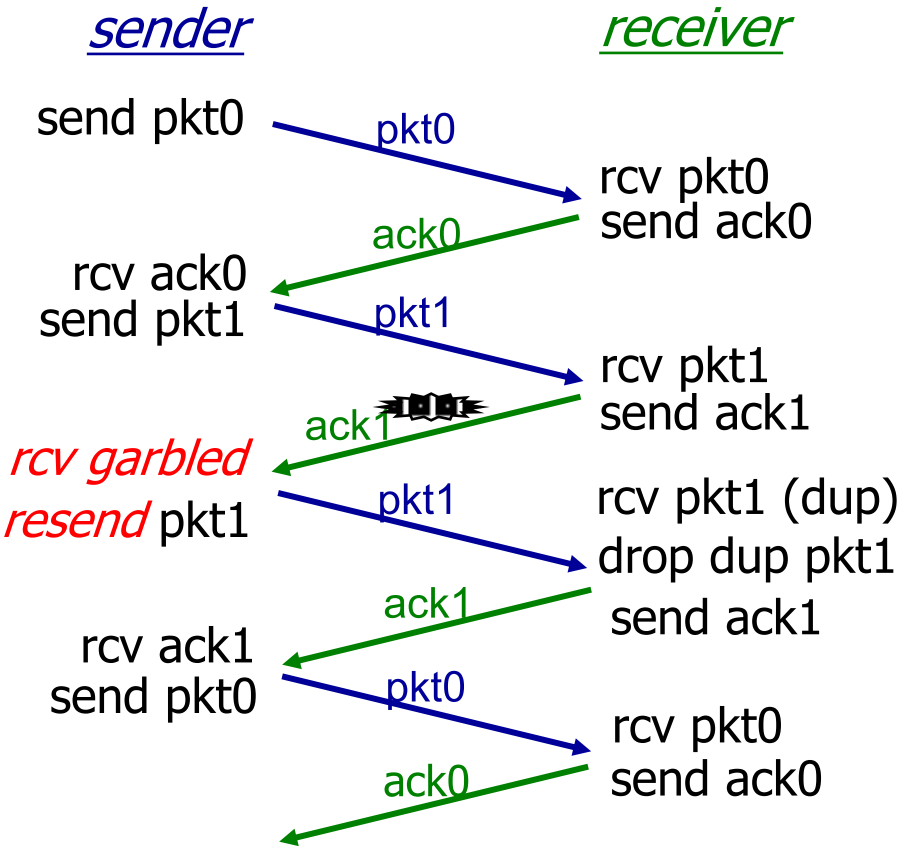
rdt2.2: a NAK-free protocol

- 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

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

rcv pkt1
send ack1

ack1

rcv garbled

resend pkt1

pkt1

rcv pkt1 (dup)
drop dup pkt1
send ack1

ack1

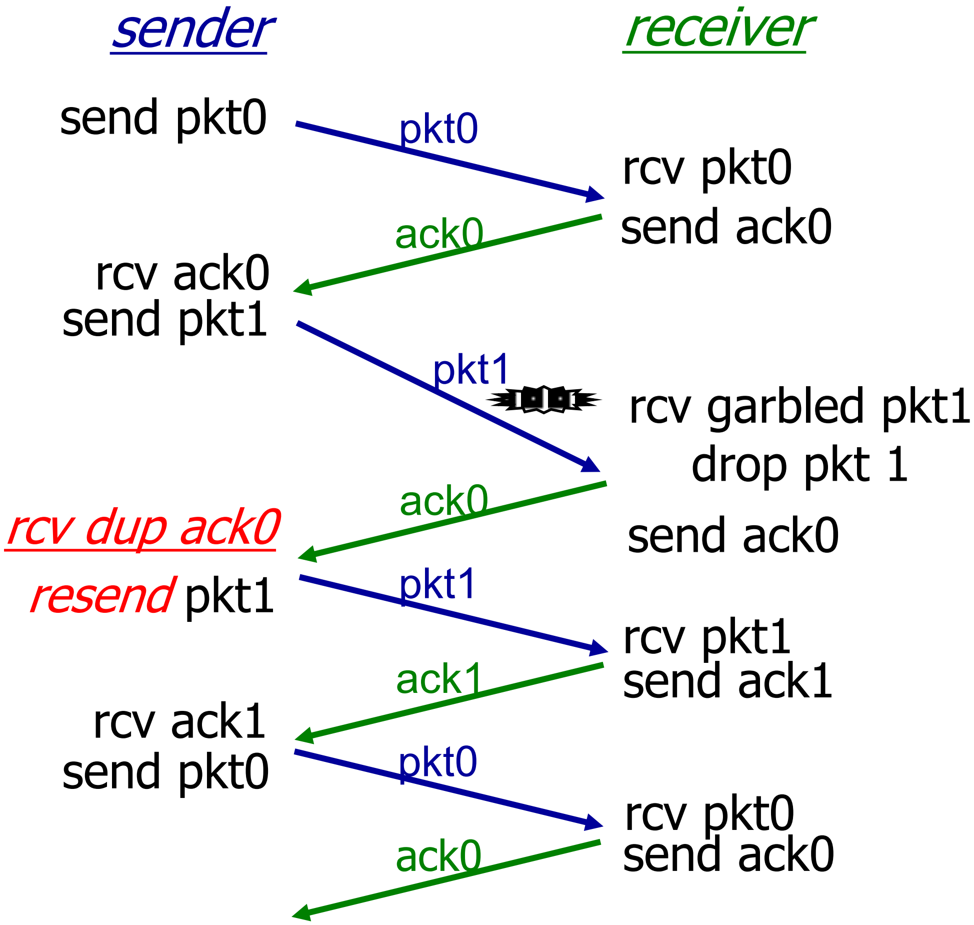
rcv ack1
send pkt0

pkt0

rcv pkt0
send ack0

ack0

(a) Corrupted ack



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

rcv garbled pkt1
drop pkt 1

send ack0

ack0

rcv dup ack0

resend pkt1

pkt1

rcv pkt1
send ack1

ack1

rcv ack1
send pkt0

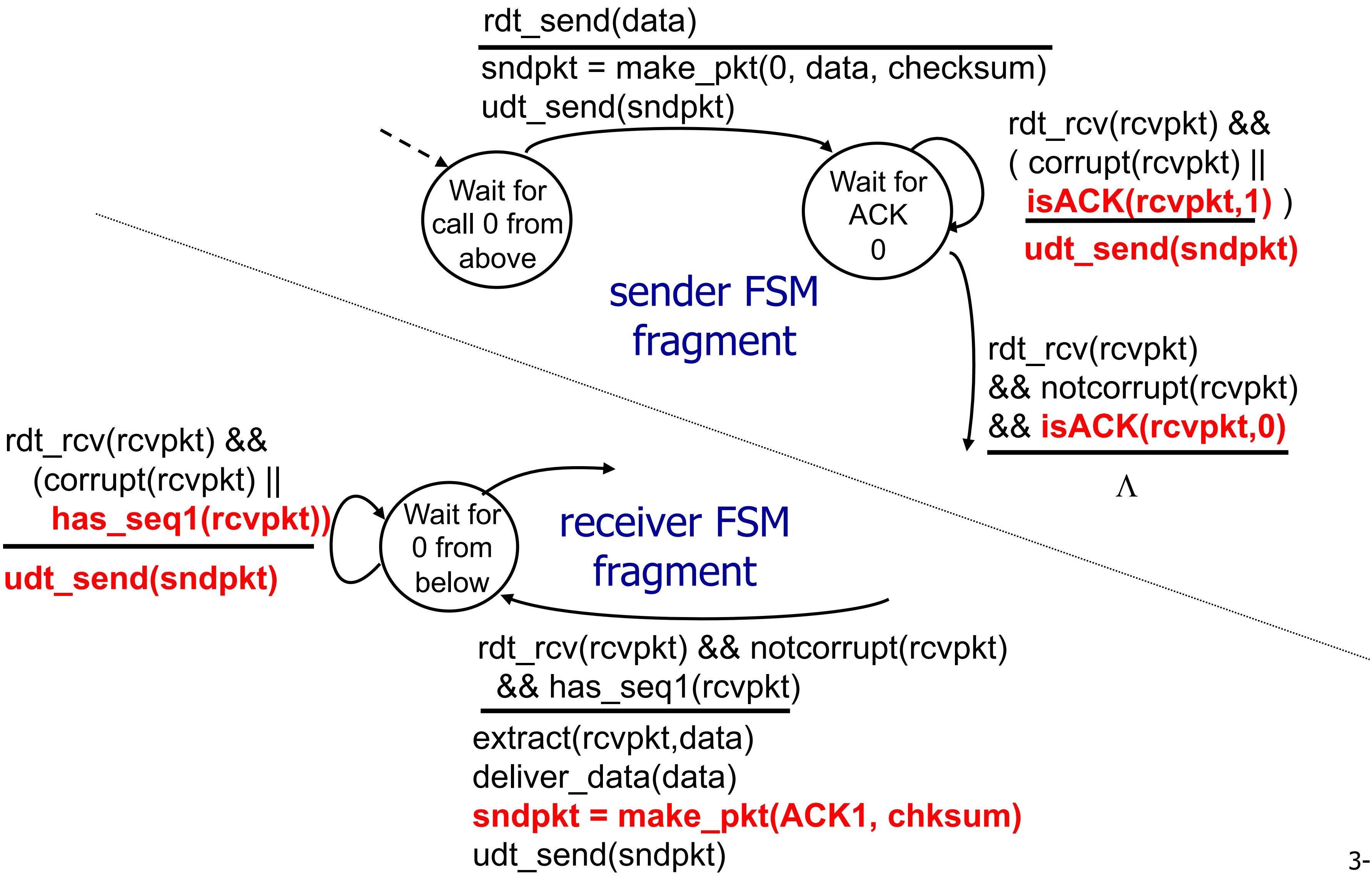
pkt0

rcv pkt0
send ack0

ack0

(b) dup ack for garbled pkt

rdt2.2: sender, receiver fragments



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)

Summary: reliable data transfer

Version	Channel	Mechanism
rdt1.0	Reliable channel	nothing
rdt2.0	bit errors (no loss)	(1)error detection via checksum (2)receiver feedback (ACK/NAK) (3)retransmission
rdt2.1	Same as 2.0 (fatal flaw)	(4)Seq# (1 bit, 0/1)
rdt2.2	Same as 2.0	A variant to rdt2.1 (no NAK) Duplicate ACK = NAK

rdt3.0: channels with errors *and* loss

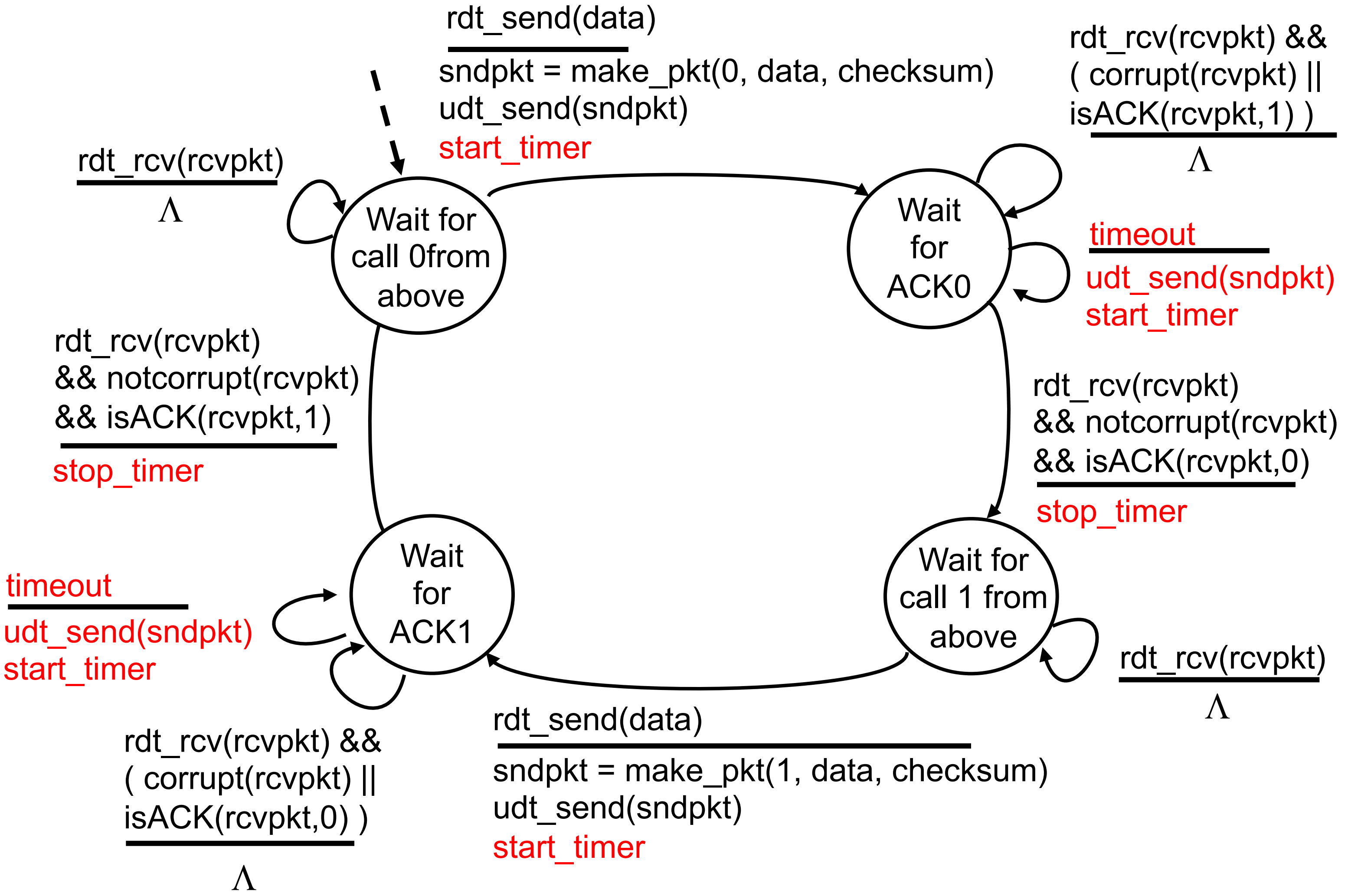
new assumption: underlying channel can also lose packets (data, ACKs)

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

approach: sender waits "reasonable" amount of time for ACK (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
- requires countdown timer

rdt3.0 sender



rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)

Λ

Wait for call 0from above

Wait for ACK0

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))

Λ

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

timeout

udt_send(sndpkt)
start_timer

Wait for ACK1

Wait for call 1 from above

rdt_rcv(rcvpkt)

Λ

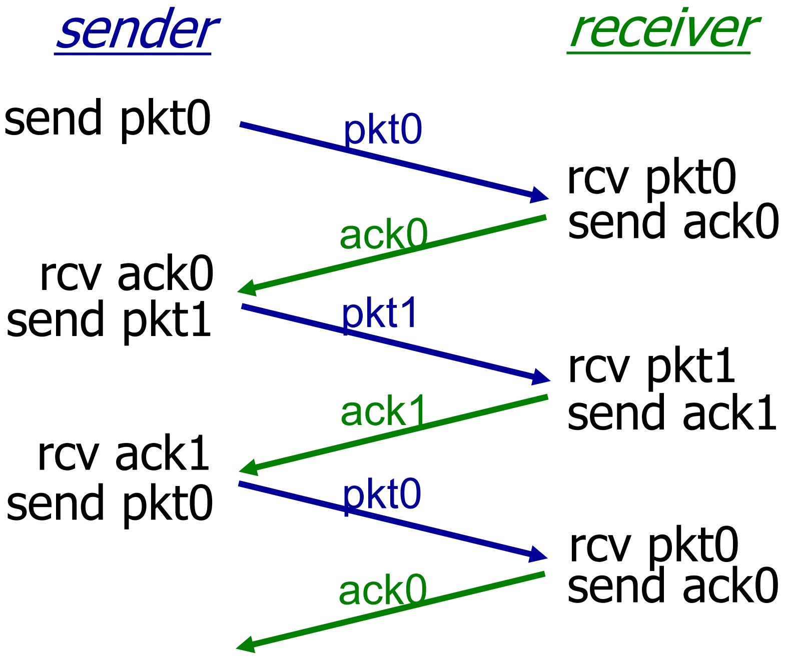
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))

Λ

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 in action



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

rcv pkt1
send ack1

ack1

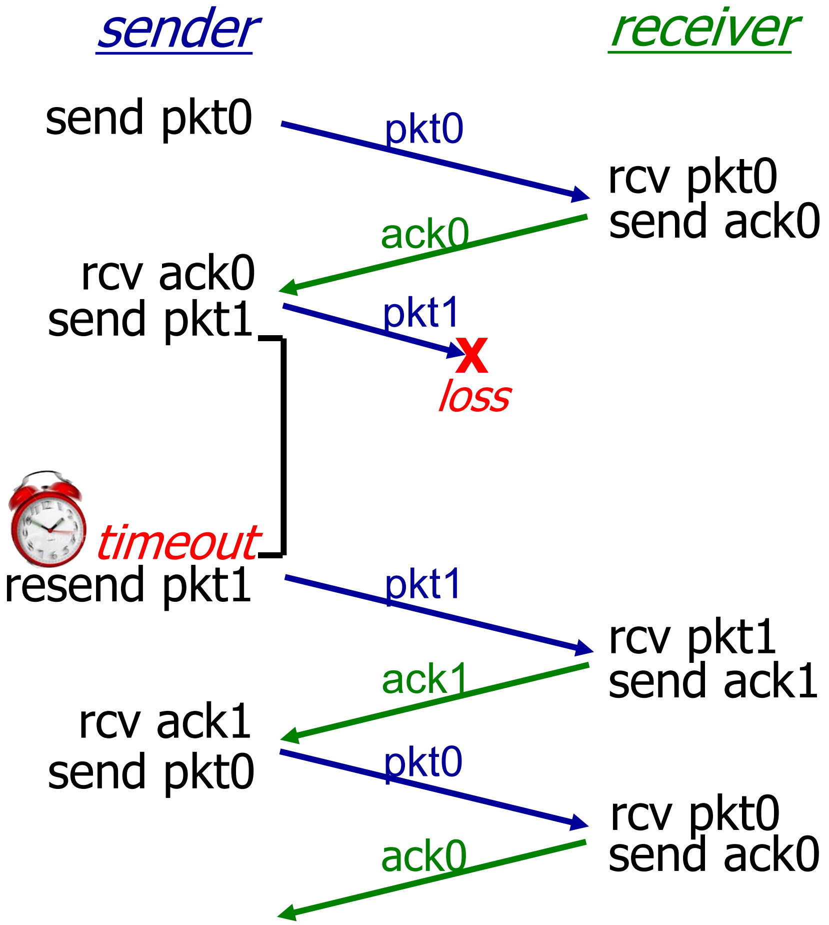
rcv ack1
send pkt0

pkt0

rcv pkt0
send ack0

ack0

(a) no loss



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

loss

timeout

resend pkt1

pkt1

rcv pkt1
send ack1

ack1

rcv ack1
send pkt0

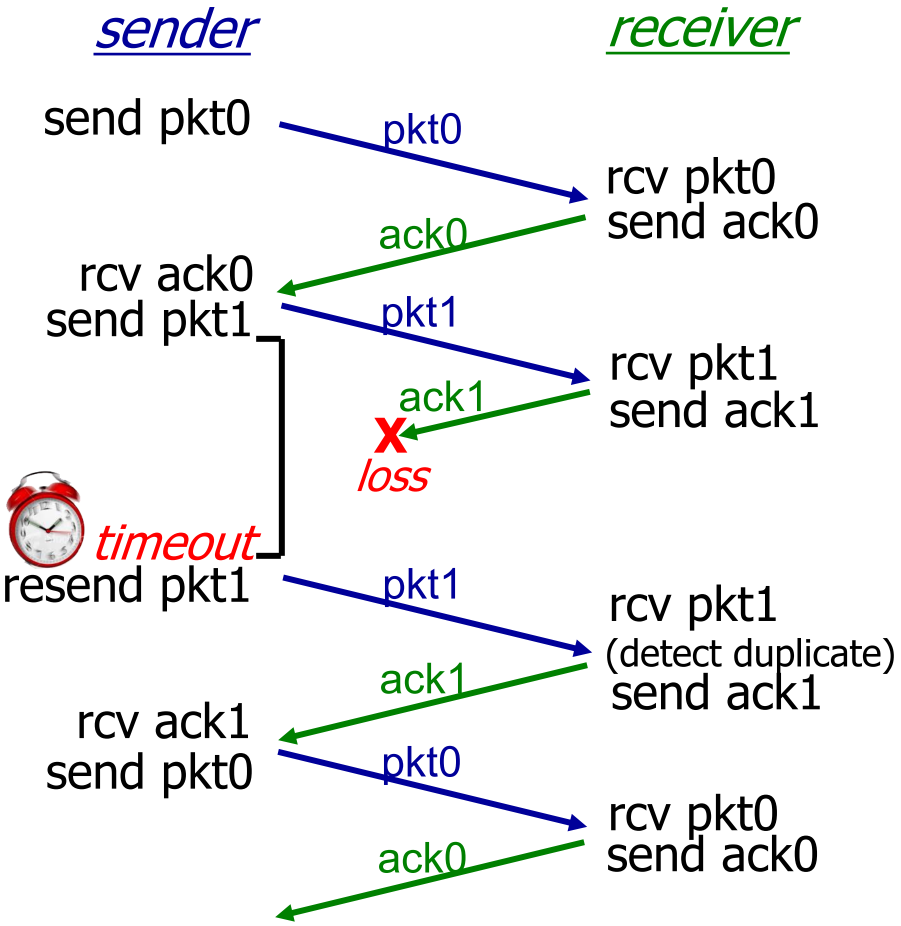
pkt0

rcv pkt0
send ack0

ack0

(b) packet loss

rdt3.0 in action


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

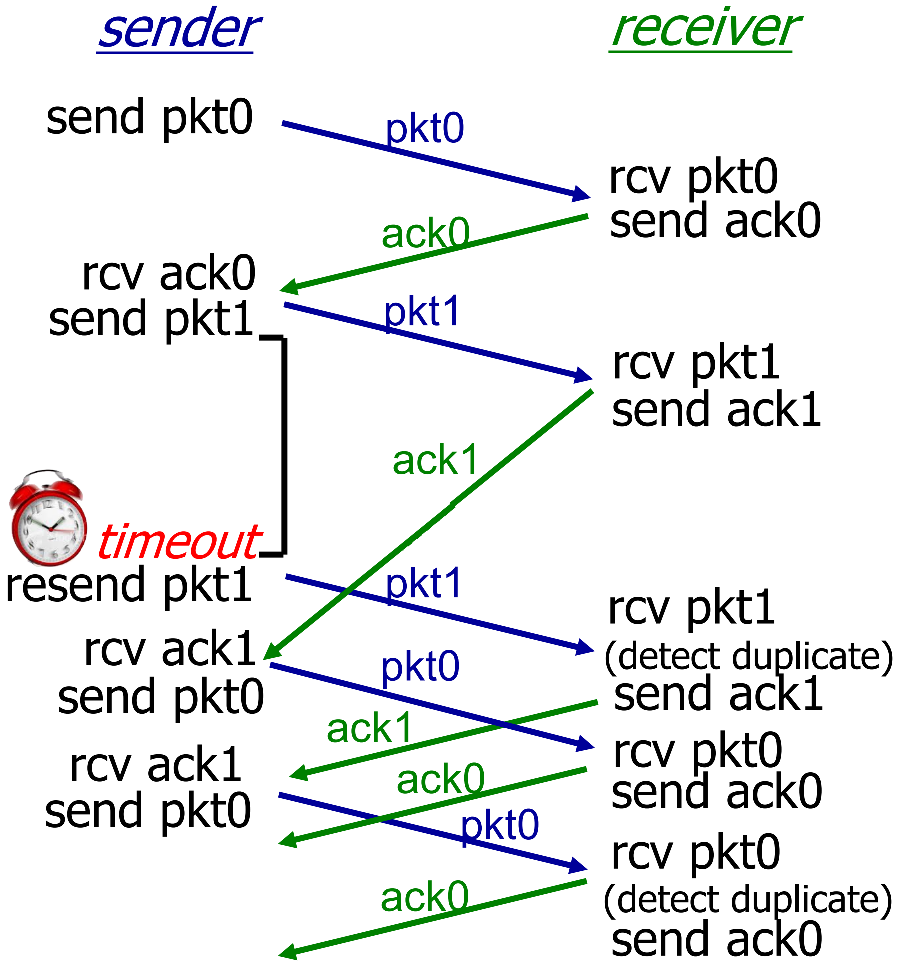
rcv ack1
send pkt0

pkt0

rcv pkt0
send ack0

ack0

(c) ACK loss


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

ack0

rcv ack1
send pkt0

pkt0

rcv pkt0
(detect duplicate)
send ack0

ack0

(d) premature timeout/ delayed ACK

Summary: reliable data transfer

Version	Channel	Mechanism
rdt1.0	Reliable channel	nothing
rdt2.0	bit errors (no loss)	(1)error detection via checksum (2)receiver feedback (ACK/NAK) (3)retransmission
rdt2.1	Same as 2.0	(4)Seq# (1 bit)
rdt2.2	Same as 2.0	A variant to rdt2.1 (no NAK) Unexpected ACK = NAK ACK0 = ACK for pkt0, NAK for pkt1
Rdt3.0	Bit errors + loss	(5) Retransmission timer No NAK, only ACK

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

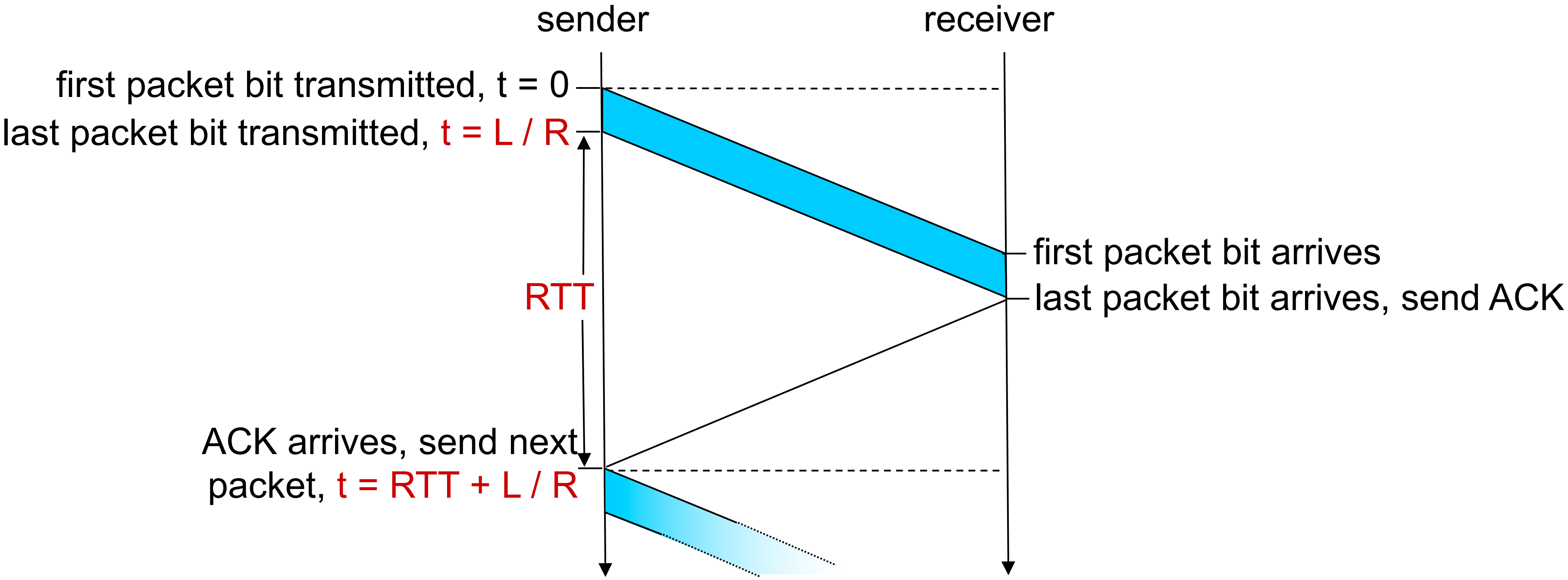
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - 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 thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: **stop-and-wait** operation

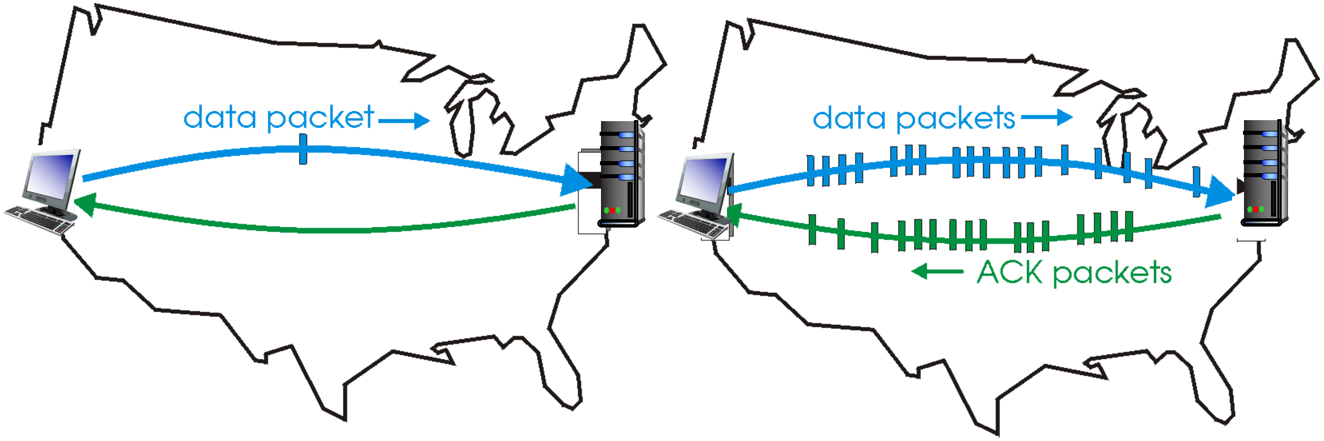

sender
receiver
first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

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

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

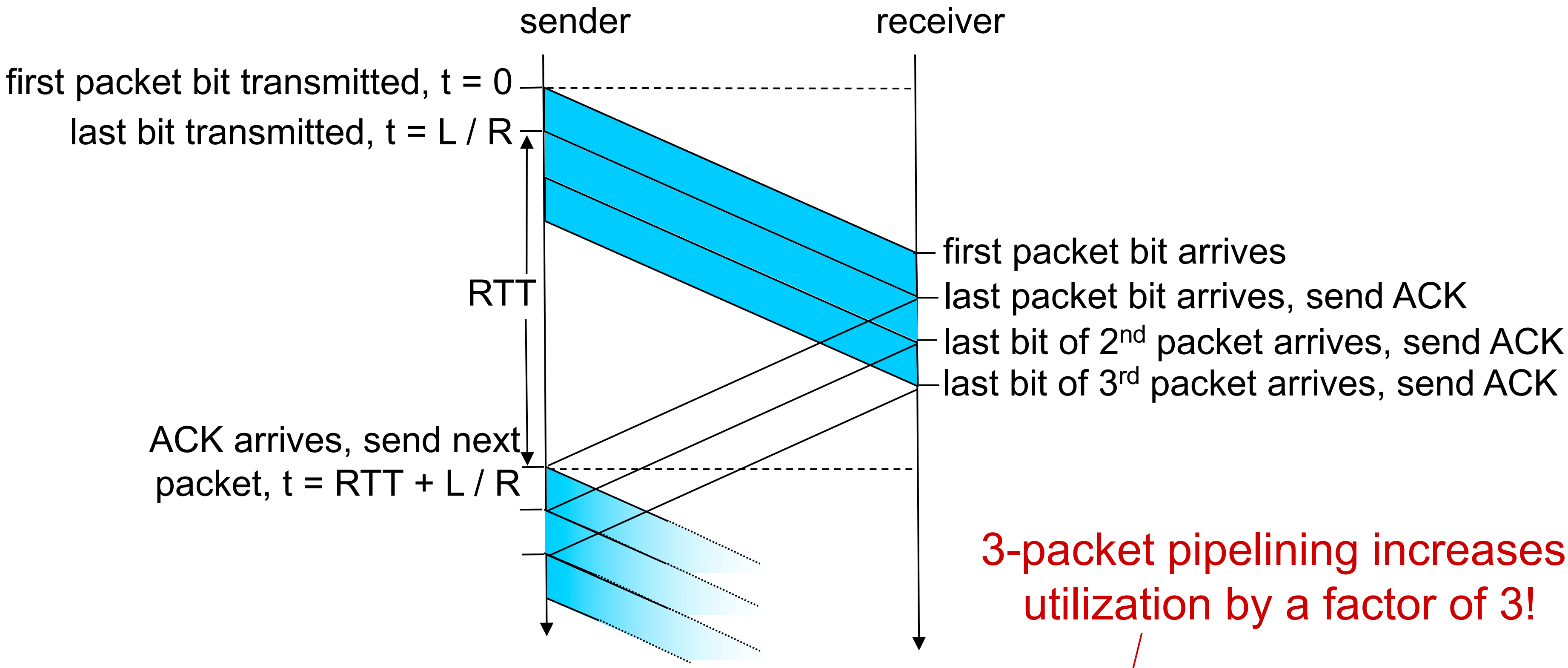


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

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:

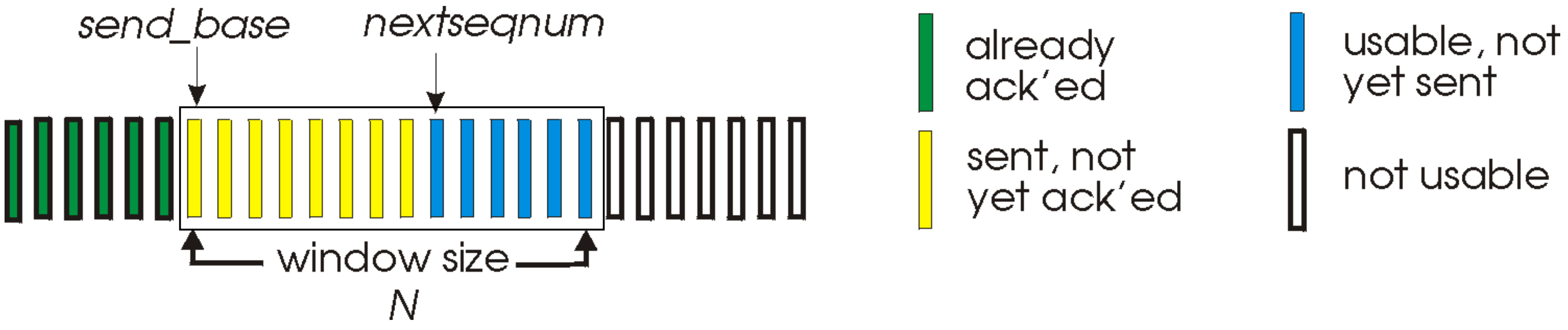
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unack’ed packets in pipeline
- rcvr 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

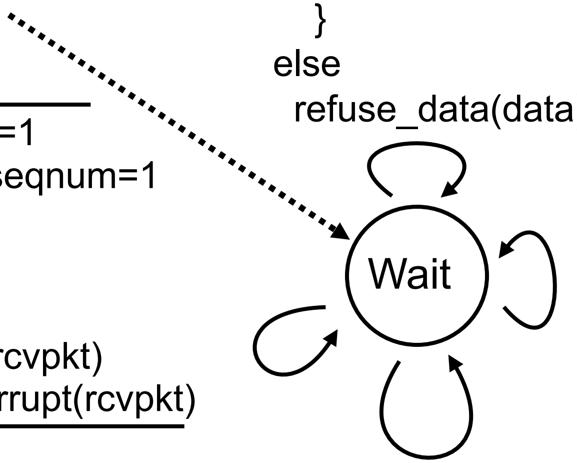


- ACK(n): ACKs all pkts up to, including seq # n - *“cumulative ACK”*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN: Sender

- sender can have up to N unacked packets in pipeline
- if data from the above AND nextseqnum < base+N,
  - Send(packet)
  - Nextseqnum++
  - Start timer (for one oldest unacked packet)
- If Timeout, retransmit *all* unacked packets
  - [base, nextseqnum-1]
- If receiving ACK, update base
  - base = getacknum(rcvpkt)+1
  - If base==nextseqnum, stop timer
- If receiving corrupted ACK, do nothing

GBN: sender extended FSM



```
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)
```

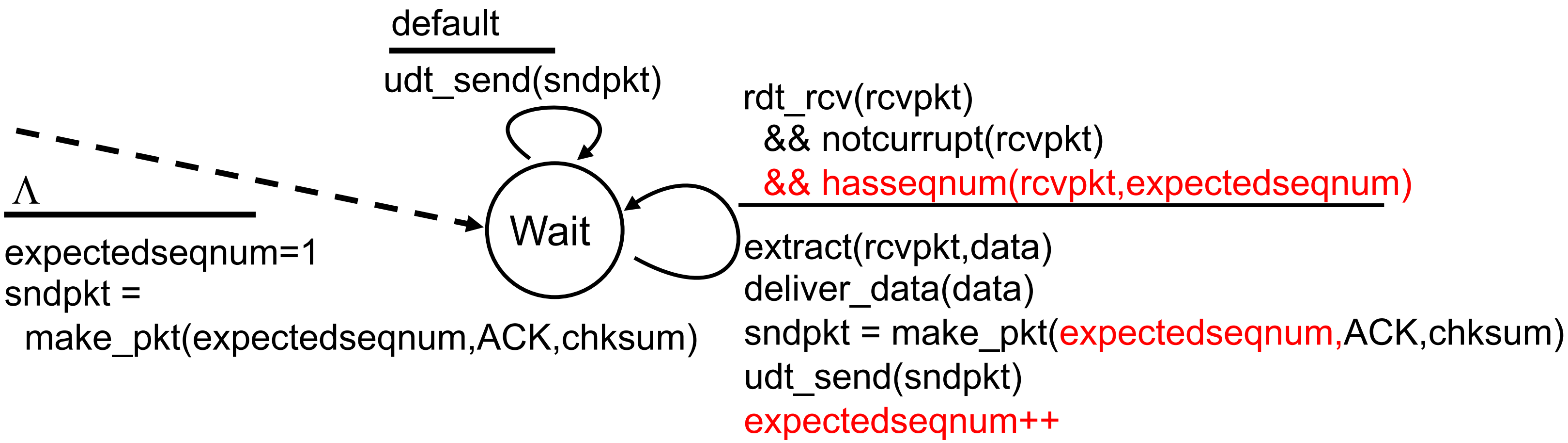
```
Λ
______________
base=1
nextseqnum=1
```

```
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 extended FSM

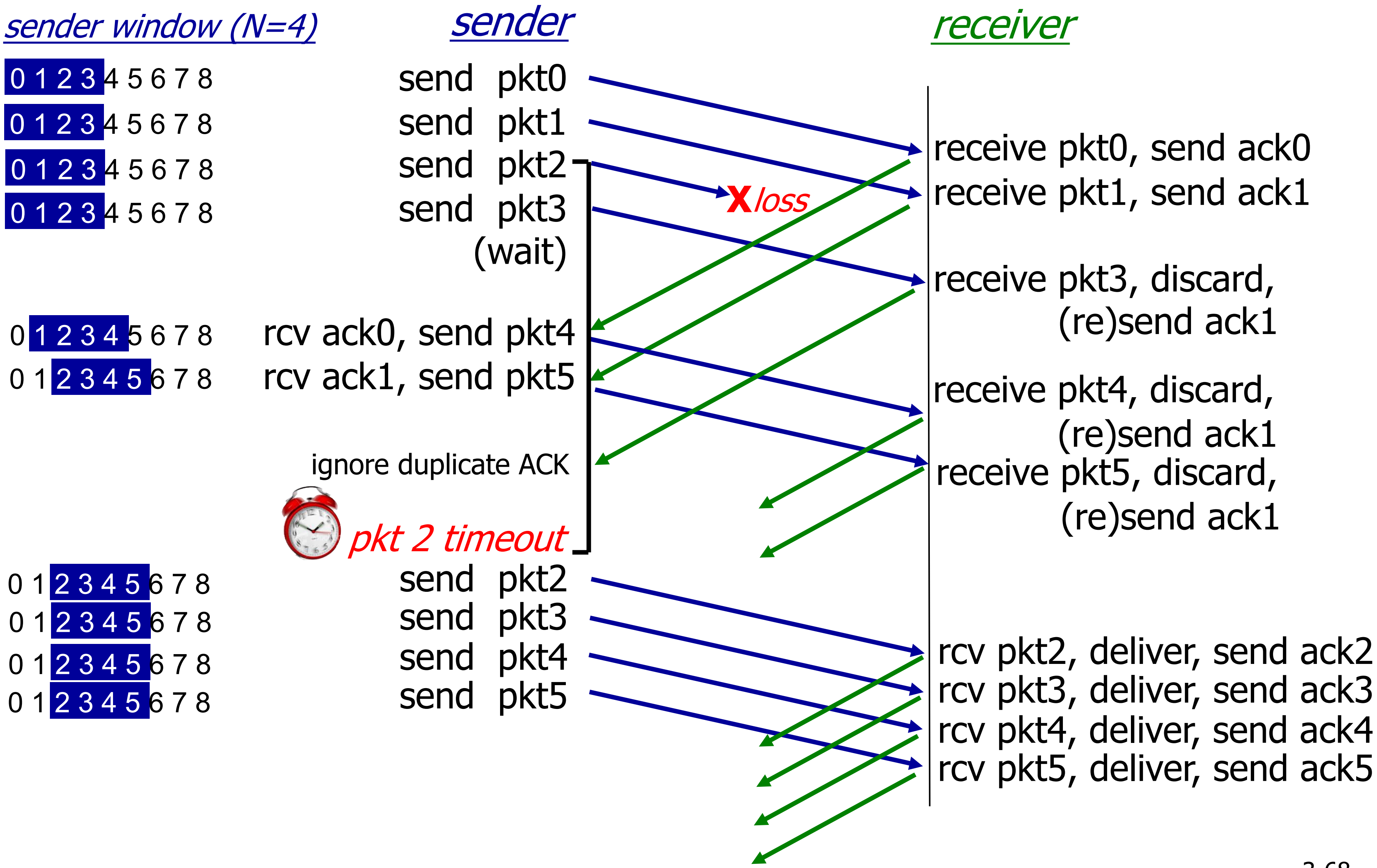


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

- out-of-order pkt:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

GBN in action



sender window (N=4)

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

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

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

ignore duplicate ACK

pkt 2 timeout

send pkt2

send pkt3

send pkt4

send pkt5

receiver

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, discard,
(re)send ack1

receive pkt4, discard,
(re)send ack1

receive pkt5, discard,
(re)send ack1

rcv pkt2, deliver, send ack2

rcv pkt3, deliver, send ack3

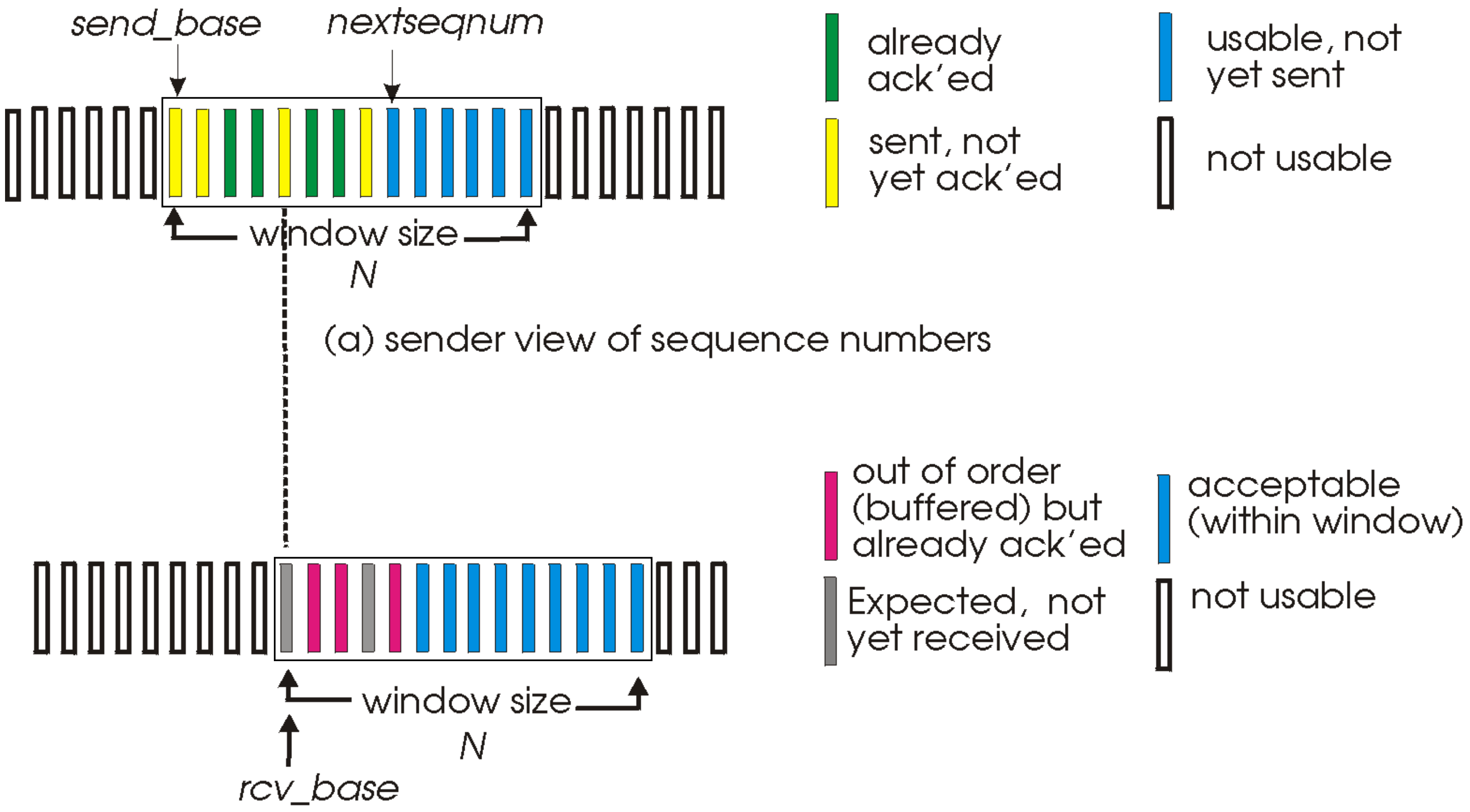
rcv pkt4, deliver, send ack4

rcv pkt5, deliver, send ack5

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



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

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

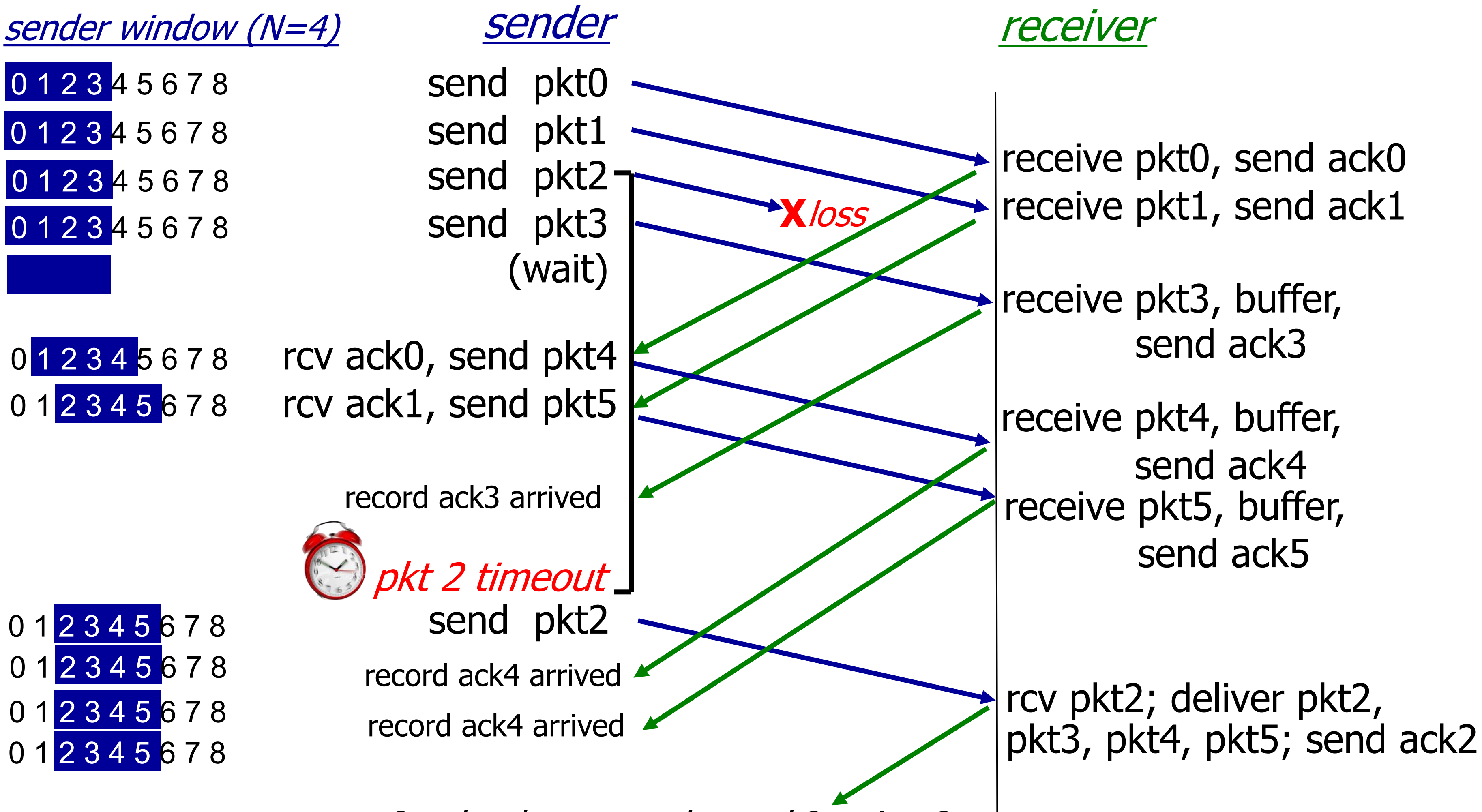
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



sender window (N=4)

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

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

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

record ack3 arrived

pkt 2 timeout

send pkt2

record ack4 arrived

record ack4 arrived

receiver

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer, send ack3

receive pkt4, buffer, send ack4

receive pkt5, buffer, send ack5

rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?

After-class Practice: GBN vs SR

- How many unique seq# may appear in GBN and SR, respectively?
  - N = 2
  - GBN: sender [4,5], what is the expected number at the receiver? **[4, 5, 6]**
    - No error
    - ACK 4 is lost
    - ACK 4 and ACK 5 are lost

GBN: give the expected number x, the sender window will be [x-2, x-1], [x-1, x], [x, x+1]

  - Given the expected number 6, how to infer the sender window?
- How about SR (expected window)? **[4,5], [5,6], [6,7]**
- What if we have N+1 sequence numbers for SR?

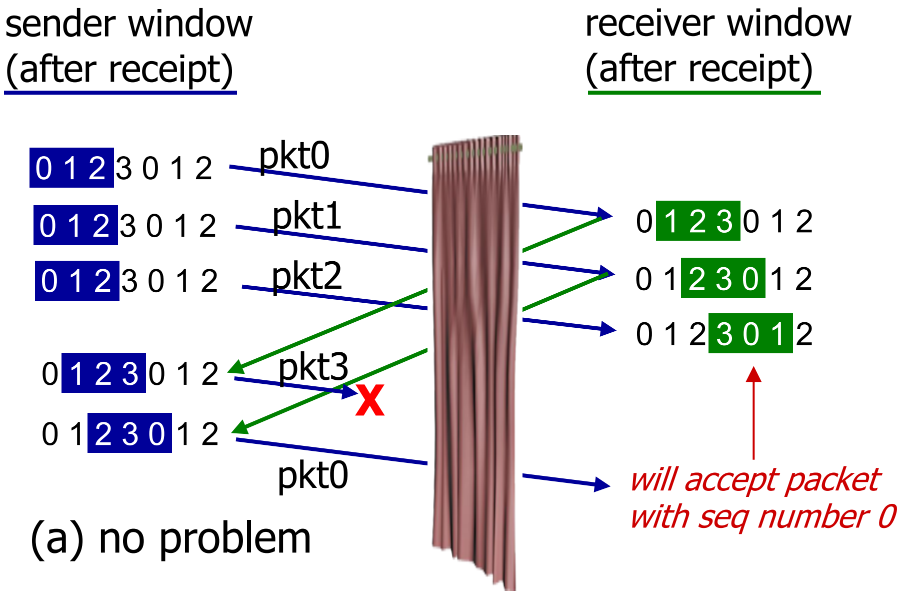
Selective repeat: dilemma (N+1)

example:

- window size=3
- seq #' s: 0, 1, 2, 3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

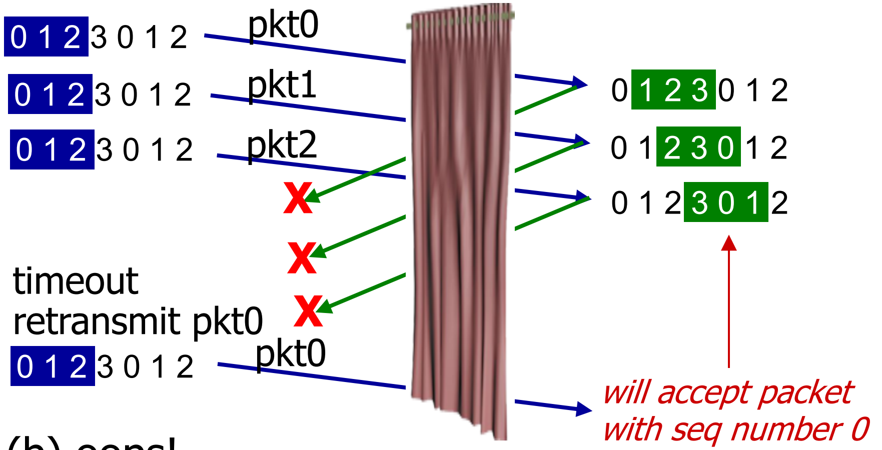
Q: what relationship between seq # size and window size to avoid problem in (b)?

2N



(a) no problem

receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!



(b) oops!

Summary: reliable data transfer

Version	Channel	Mechanism
rdt1.0	No error/loss	nothing
rdt2.0	bit errors (no loss)	(1)error detection via checksum (2)receiver feedback (ACK/NAK) (3)retransmission
rdt2.1	Same as 2.0	(4)Seq# (1 bit)
rdt2.2	Same as 2.0	(no NAK): Unexpected ACK = NAK
Rdt3.0	errors + loss	(5) Timer; ACK-only
Performance issue: low utilization		
Go-back-N	Same as 3.0	N sliding window (pipeline) Discard out-of-order pkts (recovery)
Selective Repeat	Same as 3.0	N sliding window, selective recovery

Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

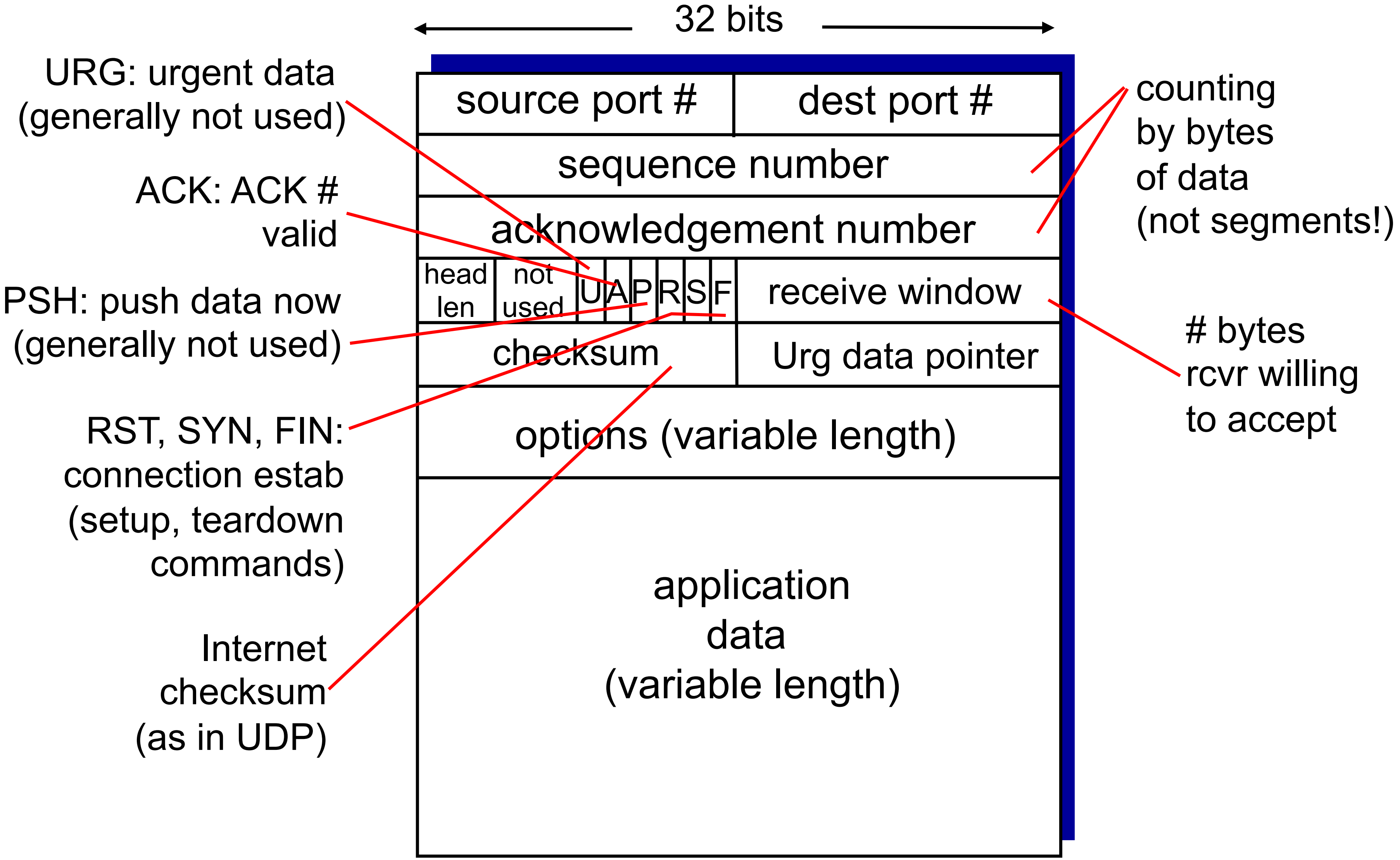
3.6 Principles of congestion control

3.7 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
  - MSS: maximum segment size
- 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

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)

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)

counting by bytes of data (not segments!)

# bytes rcvr willing to accept

TCP seq. numbers, ACKs

sequence number:

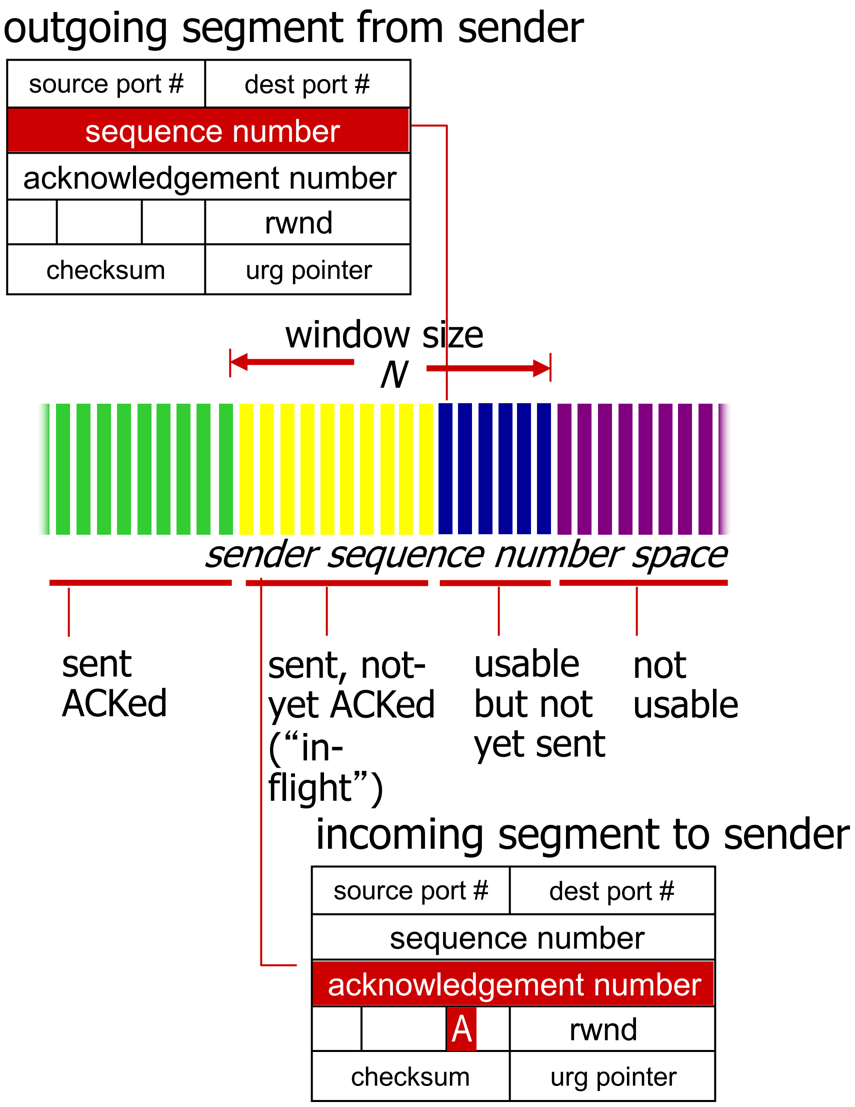
- byte stream "number" of first byte in segment's data

Acknowledgement #.:

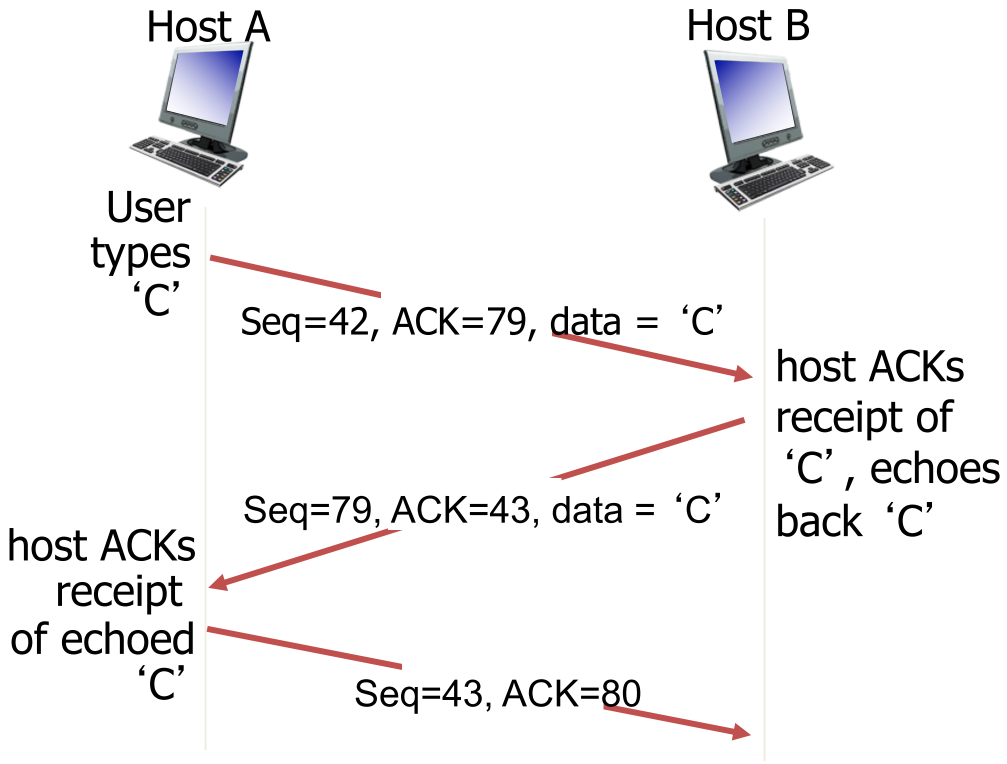
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementation



TCP seq. numbers, ACKs


Host A
Host B
User types 'C'
Seq=42, ACK=79, data = 'C'
host ACKs receipt of 'C', echoes back 'C'
Seq=79, ACK=43, data = 'C'
host ACKs receipt of echoed 'C'
Seq=43, ACK=80

simple telnet scenario

What if sending "ABC"?

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

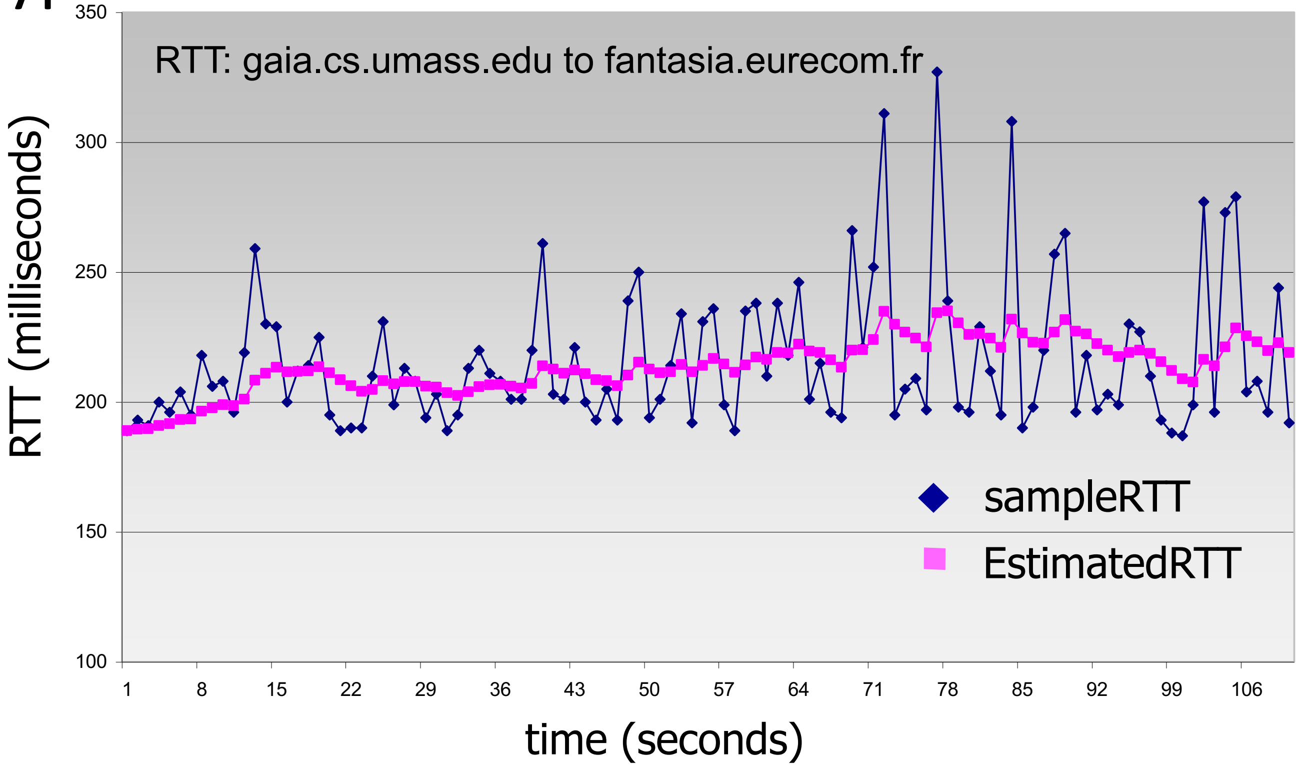
Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
    - Karn's algorithm: TCP ignores RTTs of retransmitted segments
    - Why? Avoid ACK ambiguity.
      - ACK for which transmitted segment? Original segment or retransmitted segment?
- **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
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus “safety margin”
  - large variation in **EstimatedRTT** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
                β*|SampleRTT-EstimatedRTT|
                (typically, β = 0.25)
```

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

estimated RTT

“safety margin”

Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

TCP reliable data transfer

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

let's initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

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**

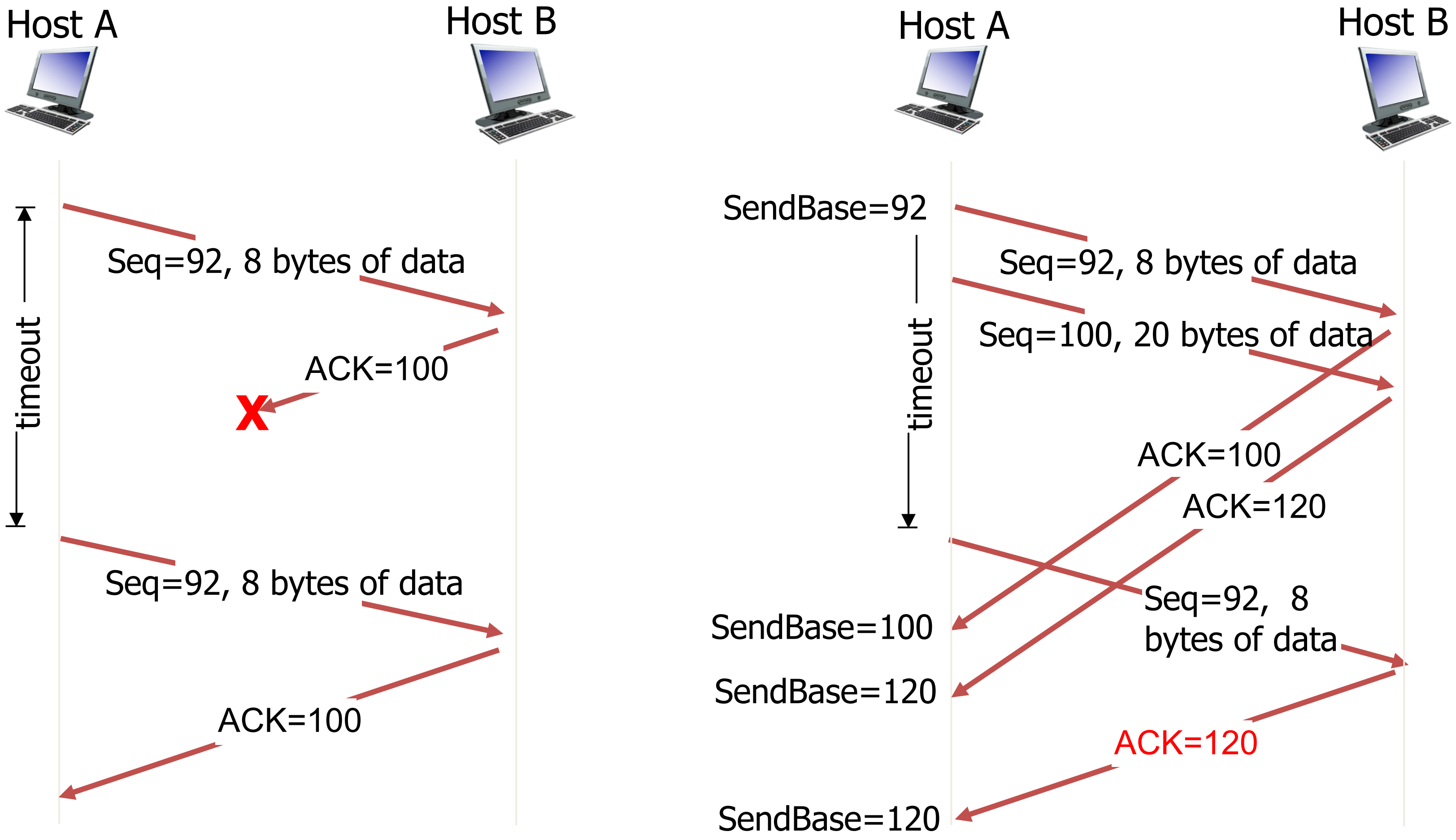
timeout:

- 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

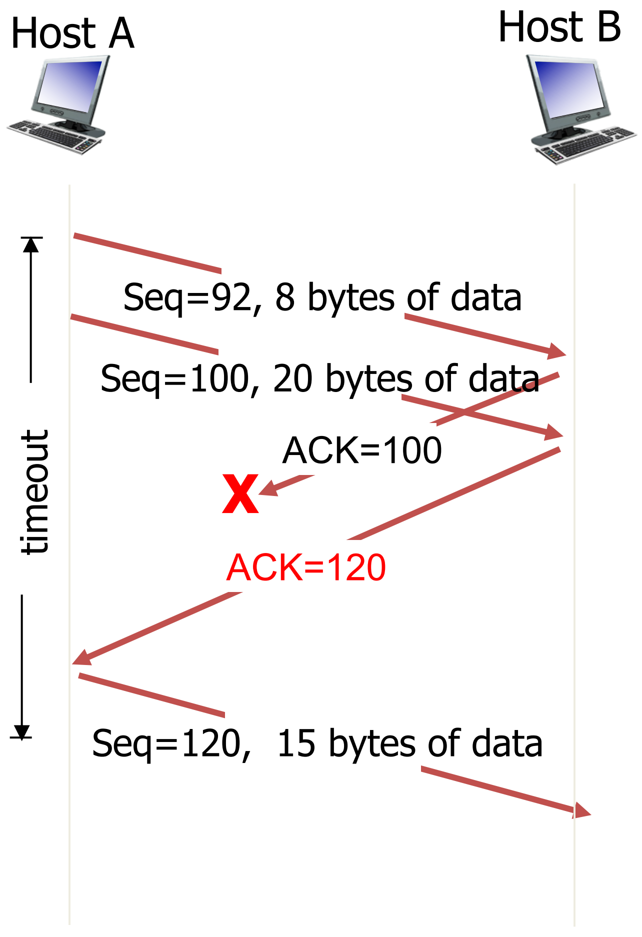
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP 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 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 segment is lost, there will likely be many duplicate ACKs.

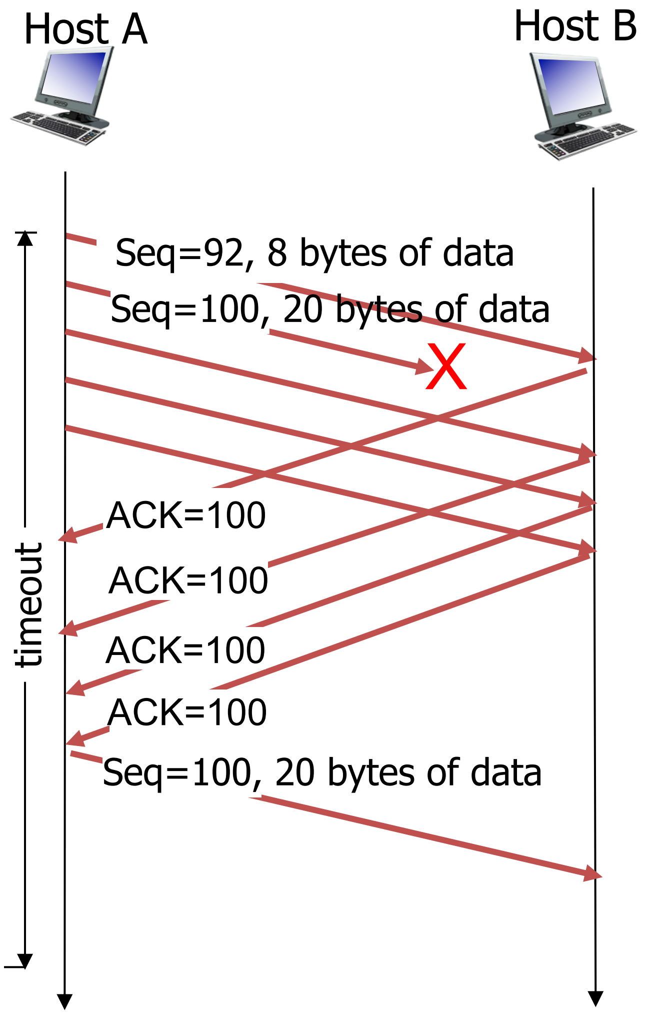
TCP fast retransmit

if sender receives 3 dup ACKs for same data

("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout
- Why wait for 3 dup ACKs? Why not respond upon 1st dup ACK?
  - Accommodate possible out of order segments

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

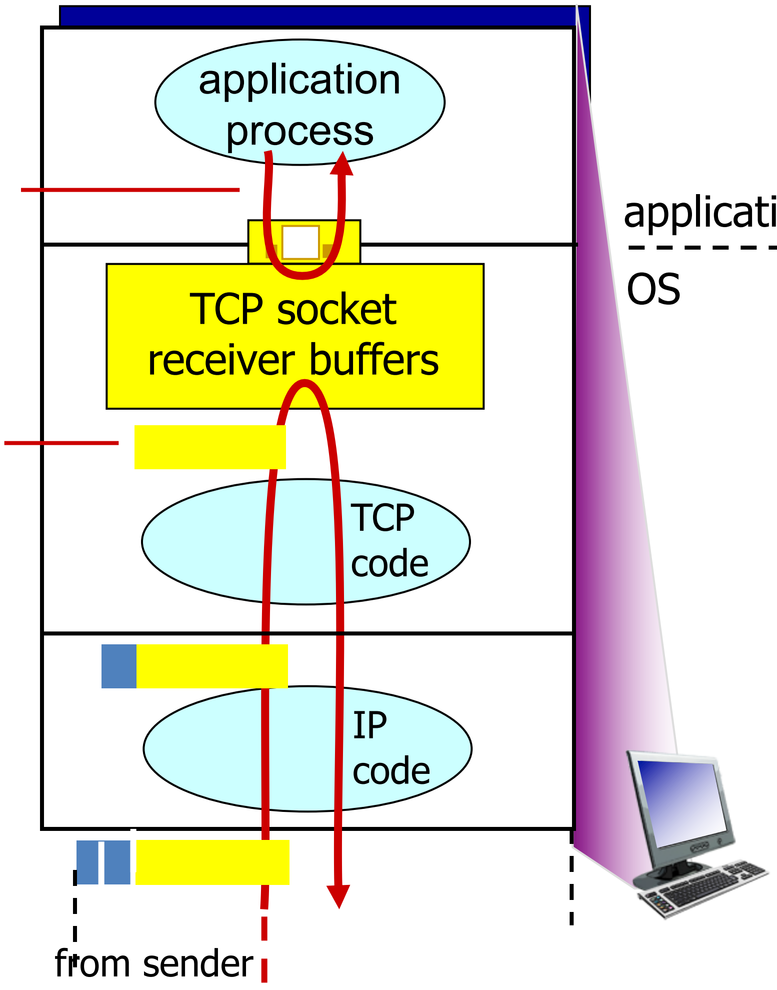
3.7 TCP congestion control

lectures (not textbook)

TCP flow control

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



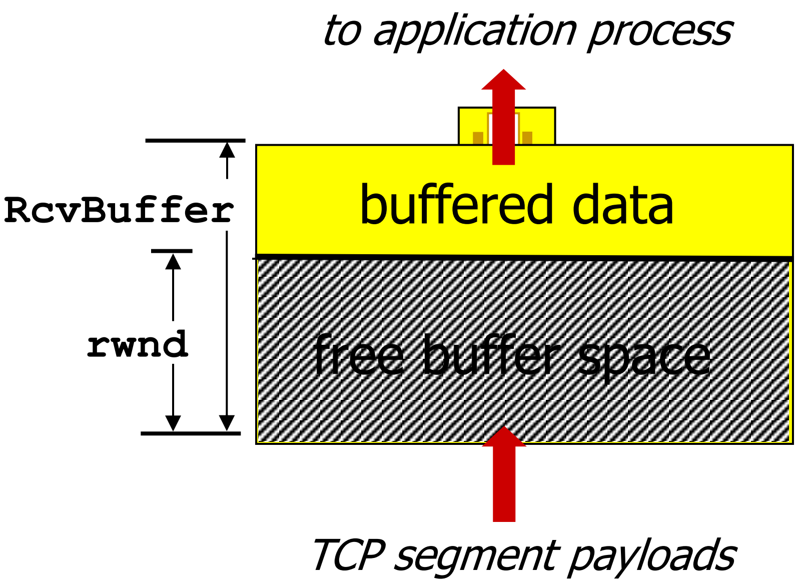
receiver protocol stack

flow control

receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast

TCP flow control

- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



receiver-side buffering

Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

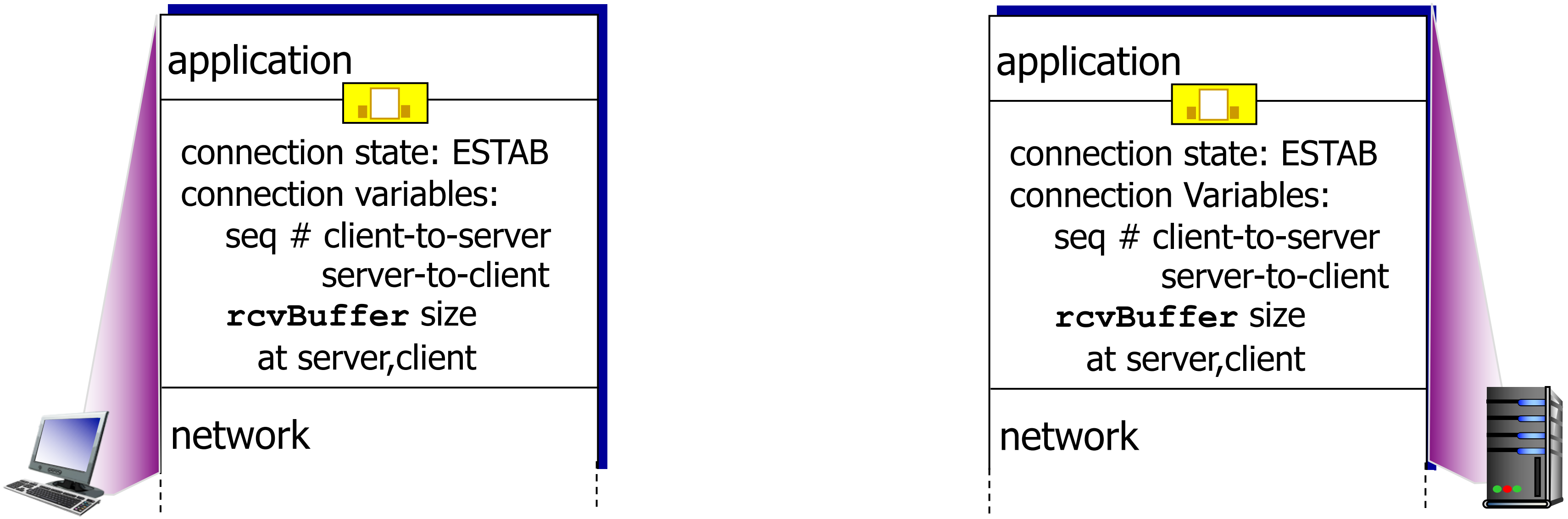
3.7 TCP congestion control

lectures (not textbook)

Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



application

connection state: ESTAB
connection variables:
seq # client-to-server
server-to-client
rcvBuffer size
at server,client

network

application

connection state: ESTAB
connection Variables:
seq # client-to-server
server-to-client
rcvBuffer size
at server,client

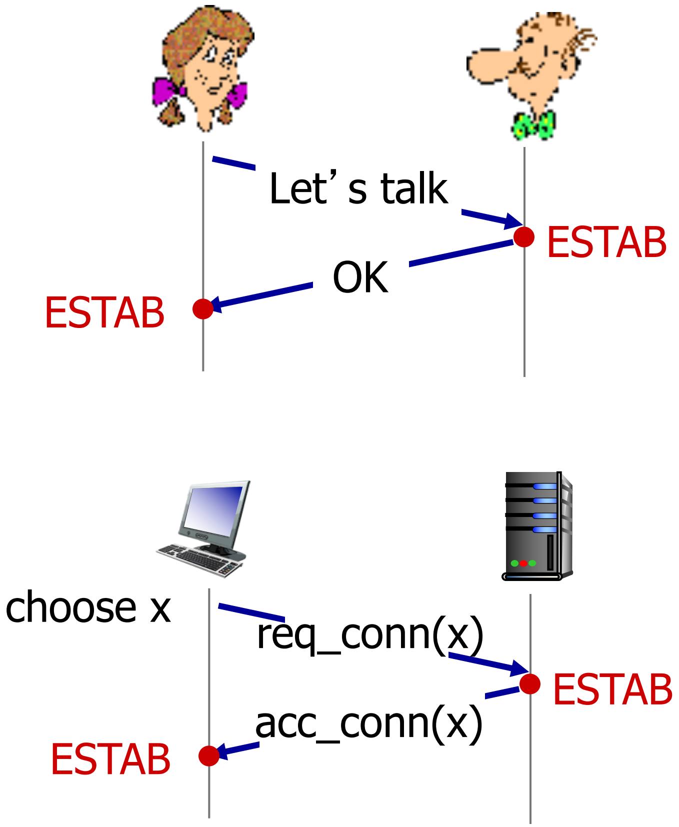
network

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
   welcomeSocket.accept();
```

Agreeing to establish a connection

2-way handshake:

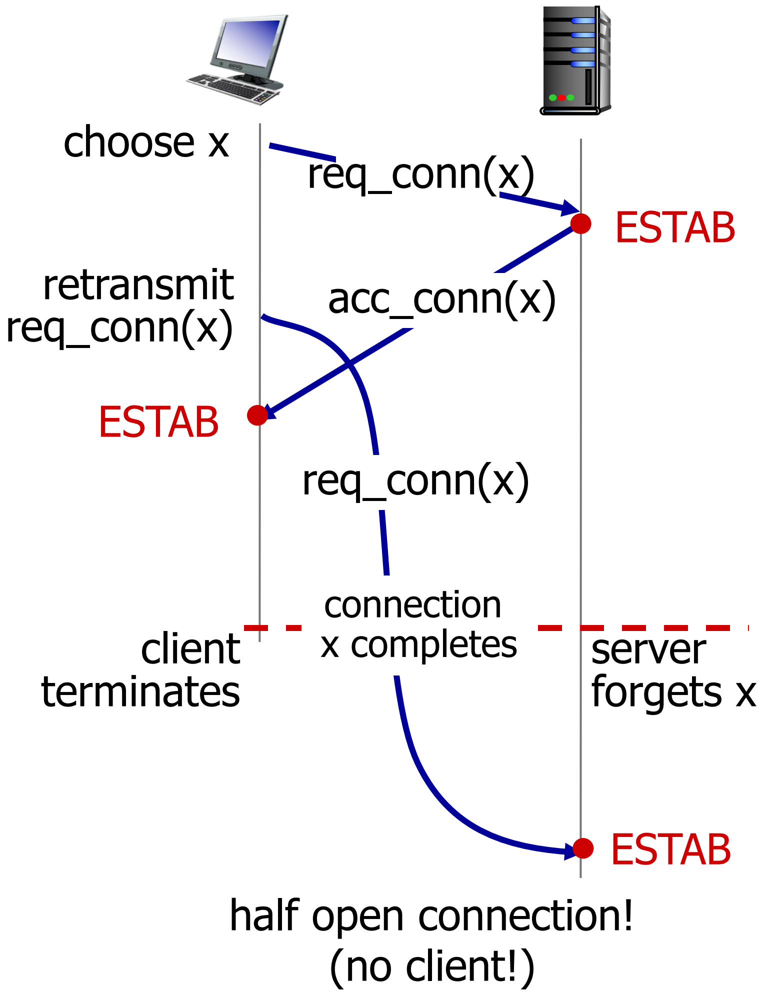


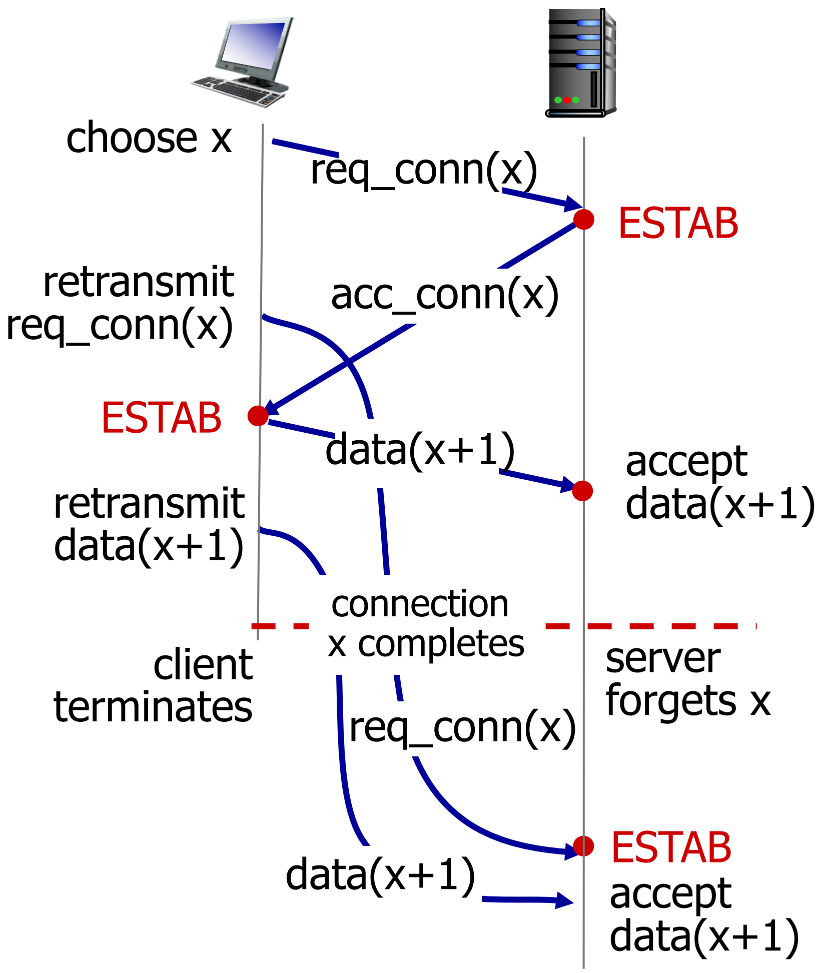
Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side

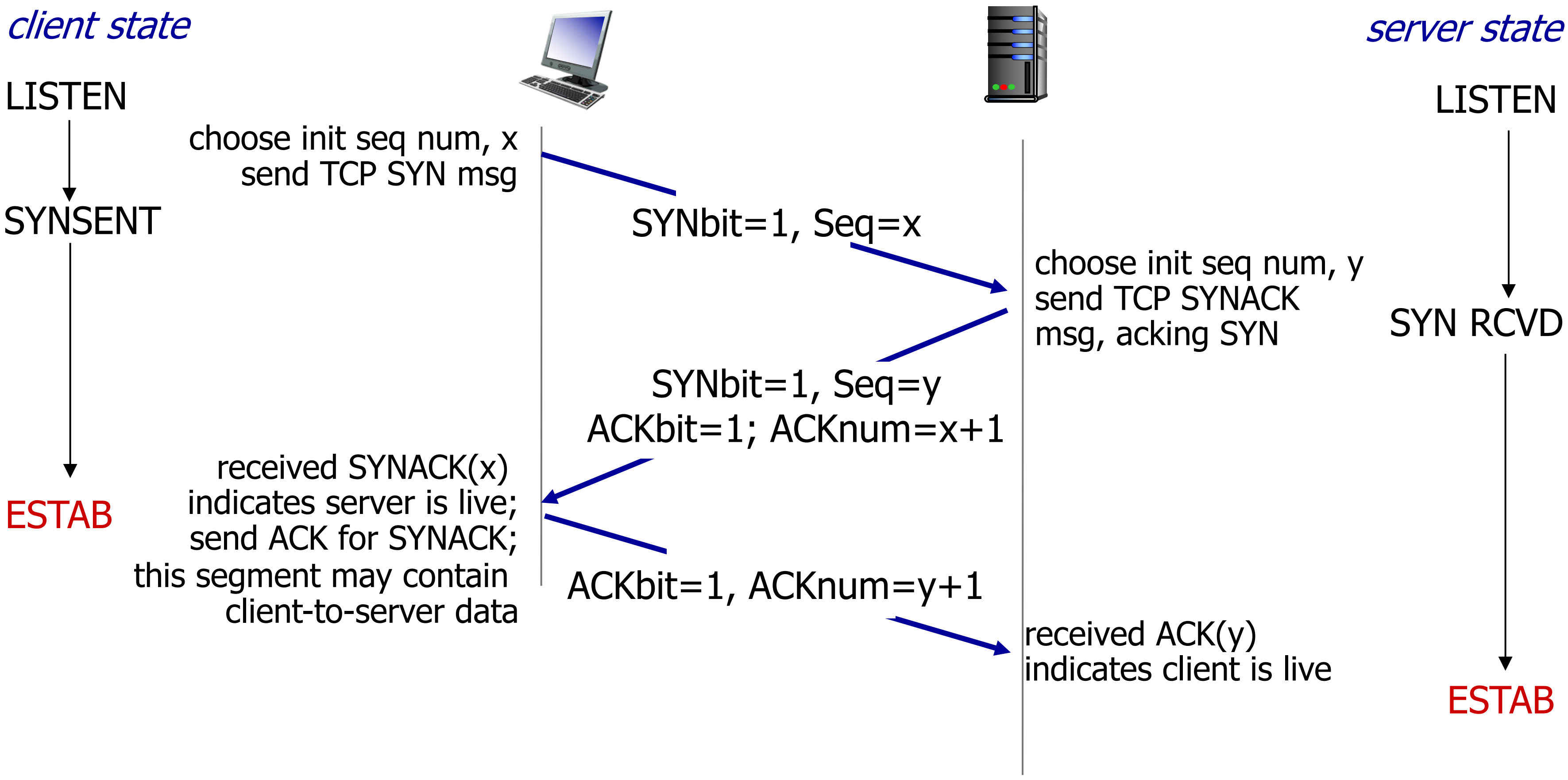
Agreeing to establish a connection

2-way handshake failure scenarios:





TCP 3-way handshake



client state

LISTEN

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

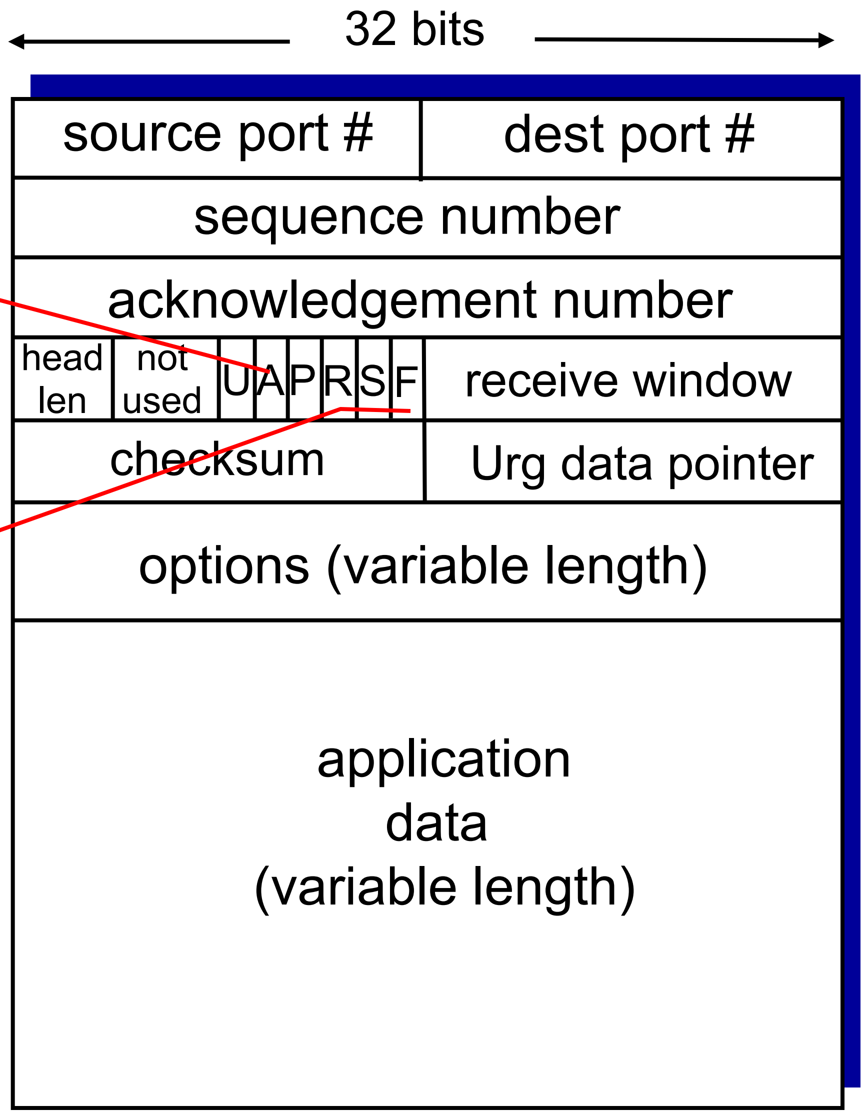
server state

LISTEN

SYN RCVD

ESTAB

How to set SYNC, ACK bit?



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)

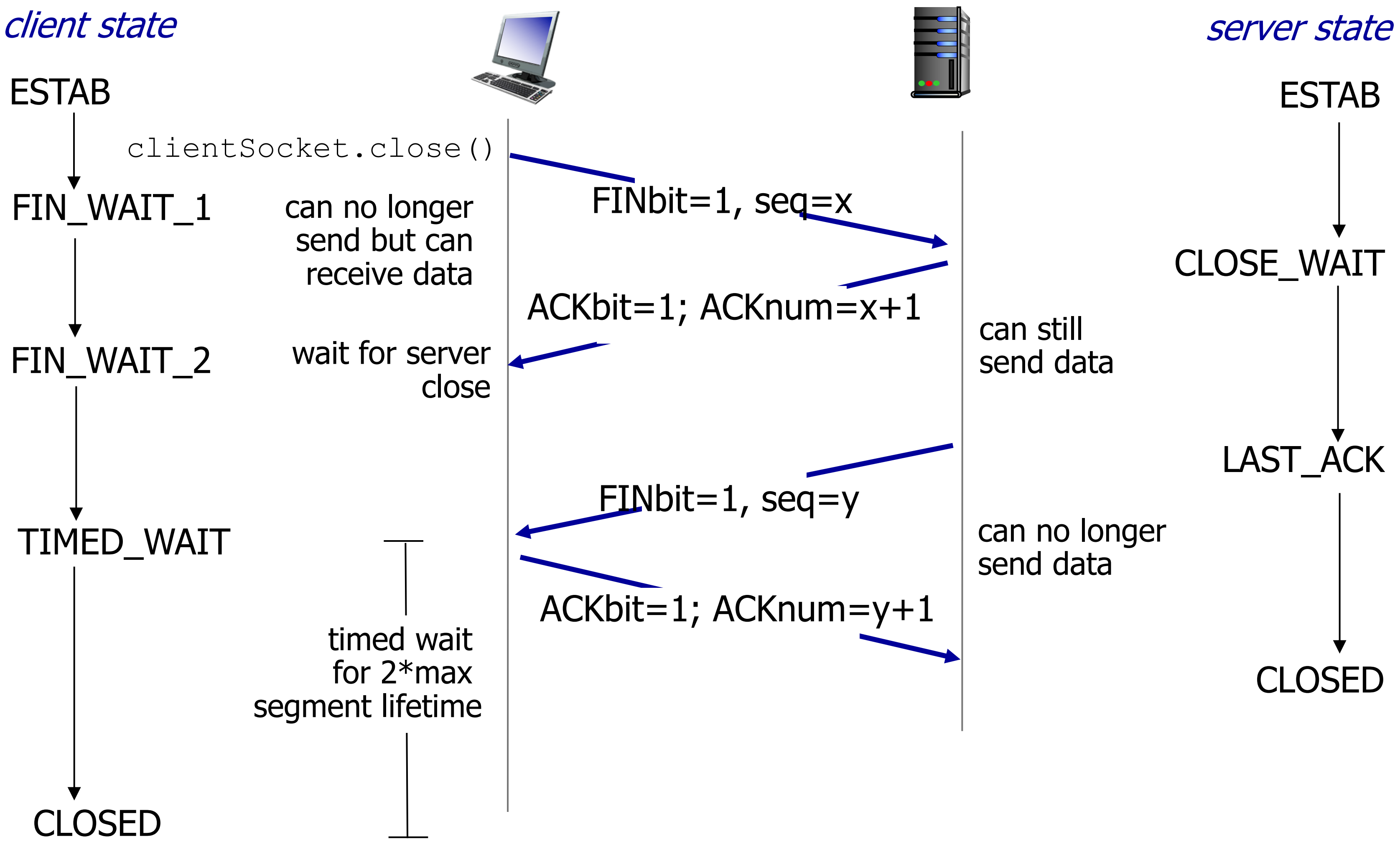
ACK: ACK # valid

RST, SYN, FIN: connection estab (setup, teardown commands)

TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection



client state

server state

ESTAB

ESTAB

clientSocket.close()

FIN_WAIT_1

can no longer send but can receive data

FINbit=1, seq=x

CLOSE_WAIT

ACKbit=1; ACKnum=x+1

can still send data

FIN_WAIT_2

wait for server close

LAST_ACK

FINbit=1, seq=y

can no longer send data

TIMED_WAIT

ACKbit=1; ACKnum=y+1

timed wait for 2*max segment lifetime

CLOSED

CLOSED

Chapter 3: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 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 (queuing in router buffers)

- a top-10 problem!

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

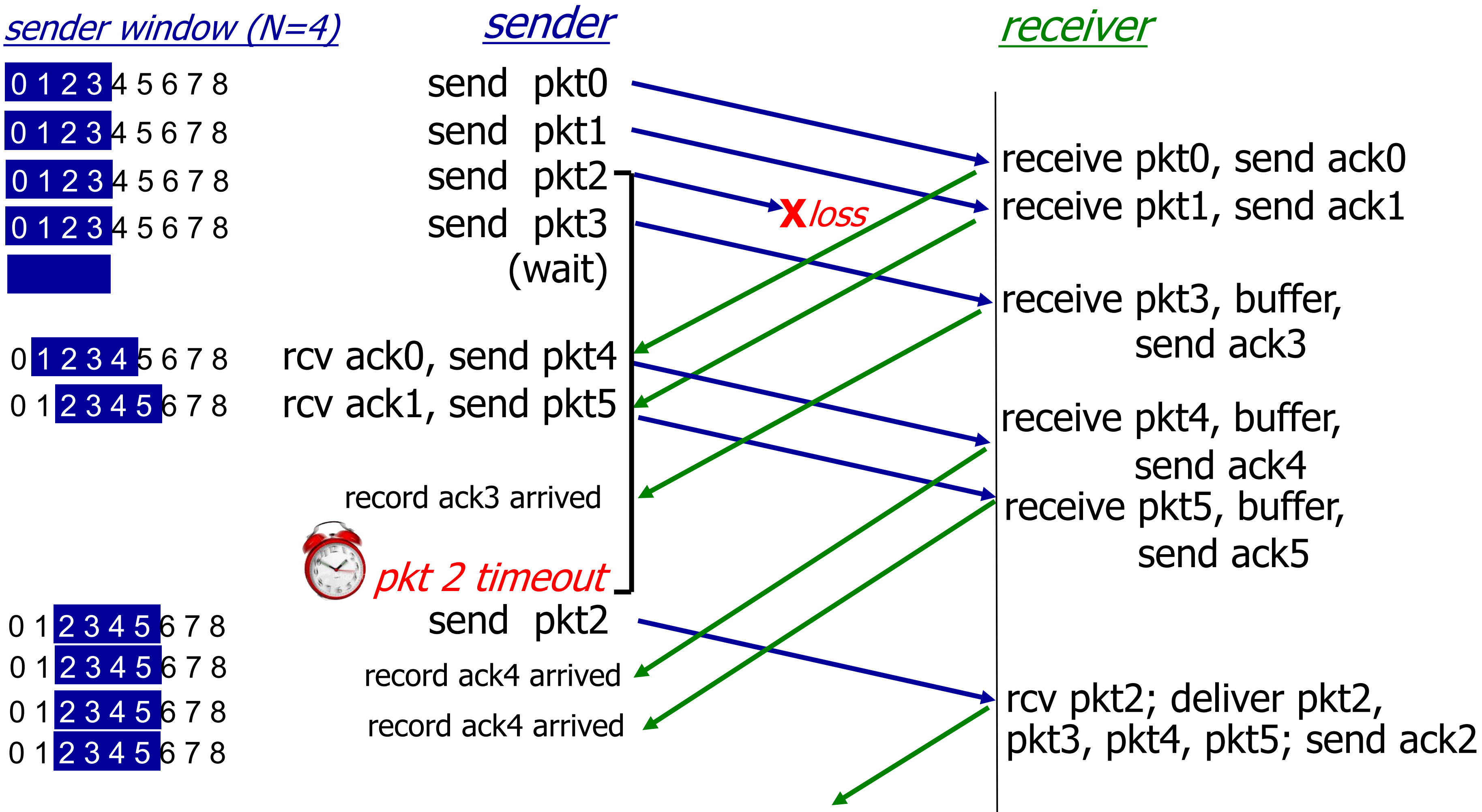
- routers provide feedback to end systems
    - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
    - explicit rate sender should send at

TCP Congestion Control

- Idea
  - Assumes best-effort network
  - Each source determines network capacity for itself
  - Implicit feedback via ACKs or timeout events
  - ACKs pace transmission (self-clocking)

- Challenge
  - Determining initial available capacity
  - Adjusting to changes in capacity in a timely manner

Recall in selective repeat protocol



sender window (N=4)

sender

receiver

0 1 2 3 4 5 6 7 8 send pkt0

0 1 2 3 4 5 6 7 8 send pkt1

0 1 2 3 4 5 6 7 8 send pkt2

0 1 2 3 4 5 6 7 8 send pkt3

(wait)

receive pkt0, send ack0

receive pkt1, send ack1

X *loss*

receive pkt3, buffer, send ack3

0 1 2 3 4 5 6 7 8 rcv ack0, send pkt4

0 1 2 3 4 5 6 7 8 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 send pkt2

0 1 2 3 4 5 6 7 8 record ack4 arrived

0 1 2 3 4 5 6 7 8 record ack4 arrived

0 1 2 3 4 5 6 7 8

rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

Congestion control: adjust the sender window size!

TCP Congestion Control

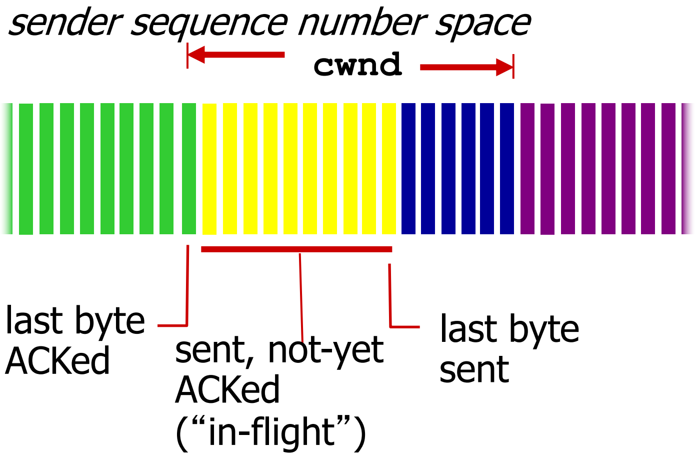
- Basic idea
  - Add notion of congestion window
    - Effective window (for selective repeat reliable transfer) is the smaller of
      - Advertised window (flow control) rwnd
      - Congestion window (congestion control) cwnd
  - Changes in congestion window size
    - Slow increases to absorb new bandwidth
    - Quick decreases to eliminate congestion

TCP Congestion Control

- sender limits transmission:

  LastByteSent-LastByteAcked

  ≤ cwnd



sender sequence number space
cwnd
last byte ACKed
sent, not-yet ACKed ("in-flight")
last byte sent

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

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate ACKs
- TCP sender reduces rate (**cwnd**) after loss event

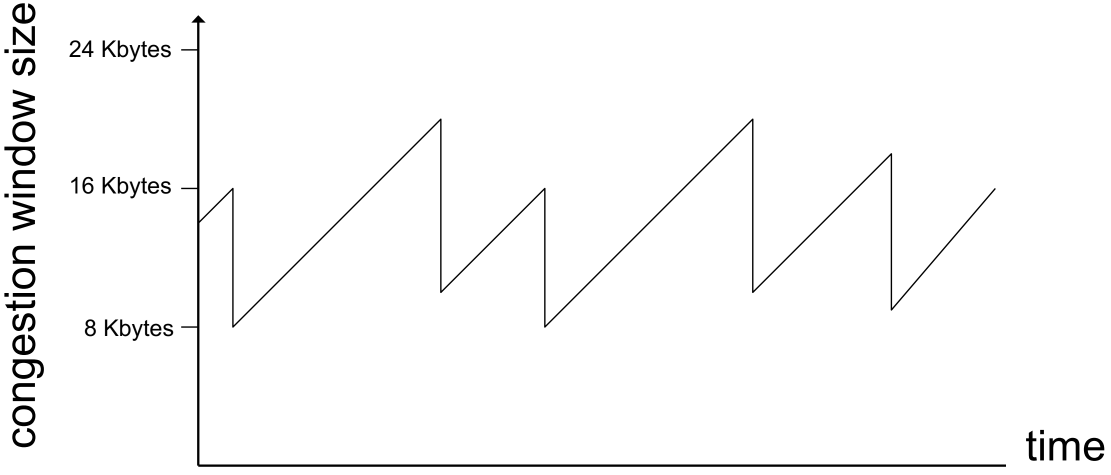
three mechanisms:

- AIMD: how to grow cwnd
- slow start: startup
- conservative after loss (timeout, duplicate ACKs) events

AIMD Rule: additive increase, multiplicative decrease

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **cwnd** by 50% after loss

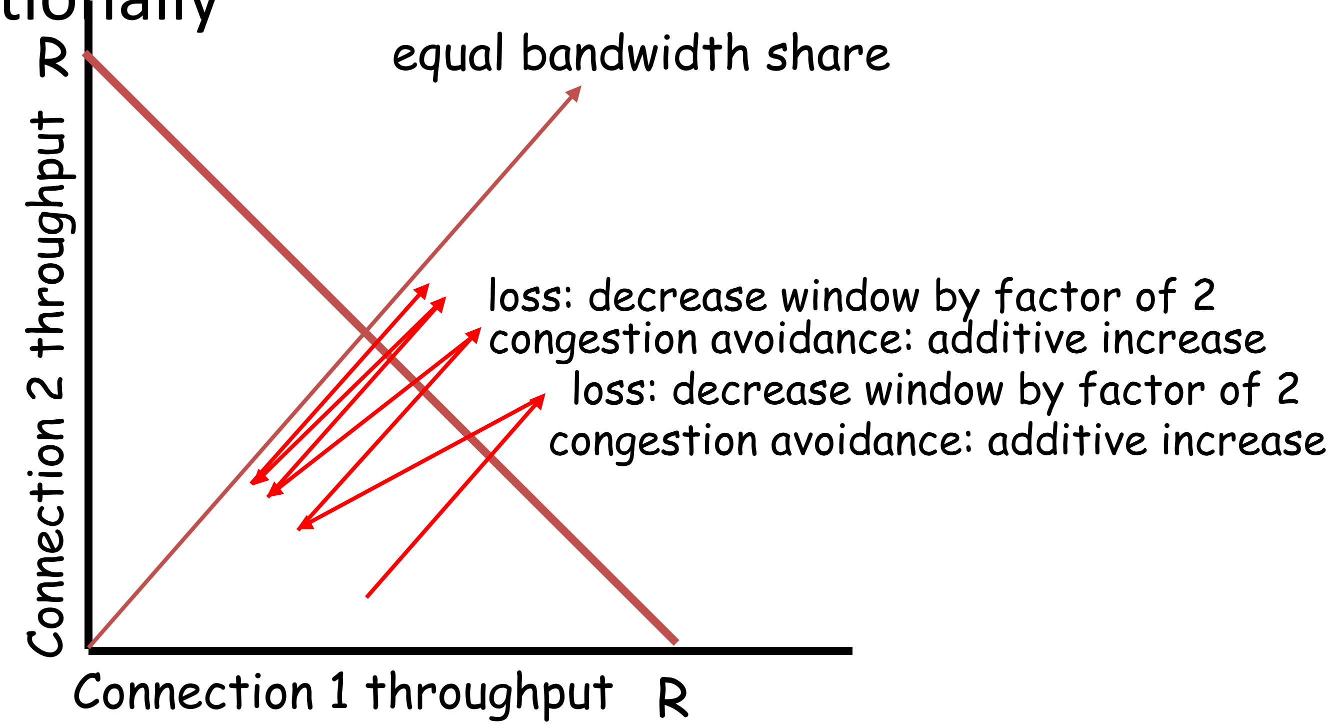
Saw tooth behavior: probing for bandwidth



What AIMD? TCP Fairness

Two competing sessions:

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


R
equal bandwidth share
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 2 throughput
Connection 1 throughput
R

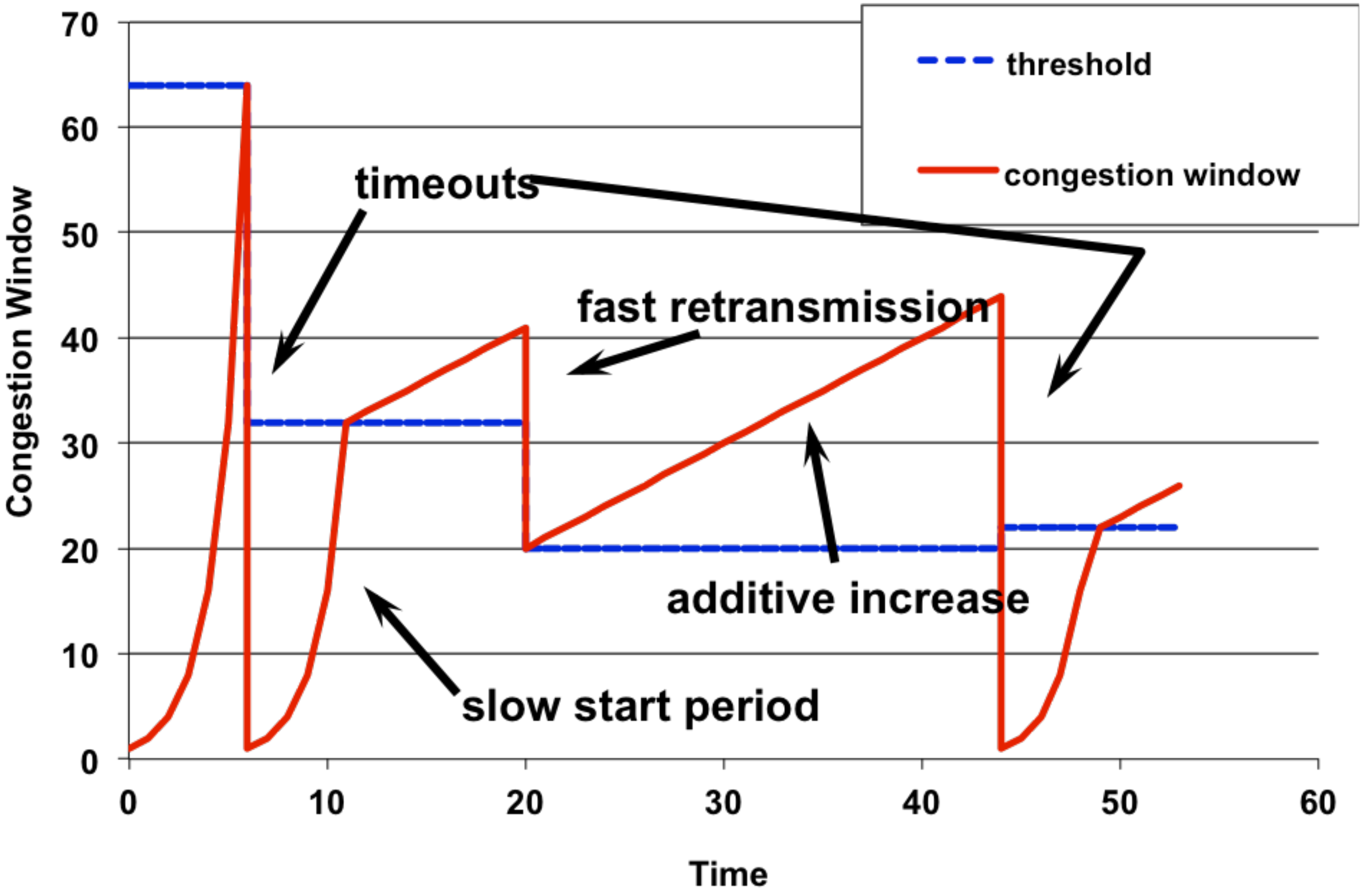
TCP Congestion Control (RFC 5681)

How to implement TCP Congestion Control?

Multiple algorithms work together:

- slow start: **how to jump-start**
- congestion avoidance: **additive increase**
- fast retransmit/fast recovery: recover from single packet loss: **multiplicative decrease**
- retransmission upon timeout: **conservative loss/failure handling**

Trace for TCP Congestion Window



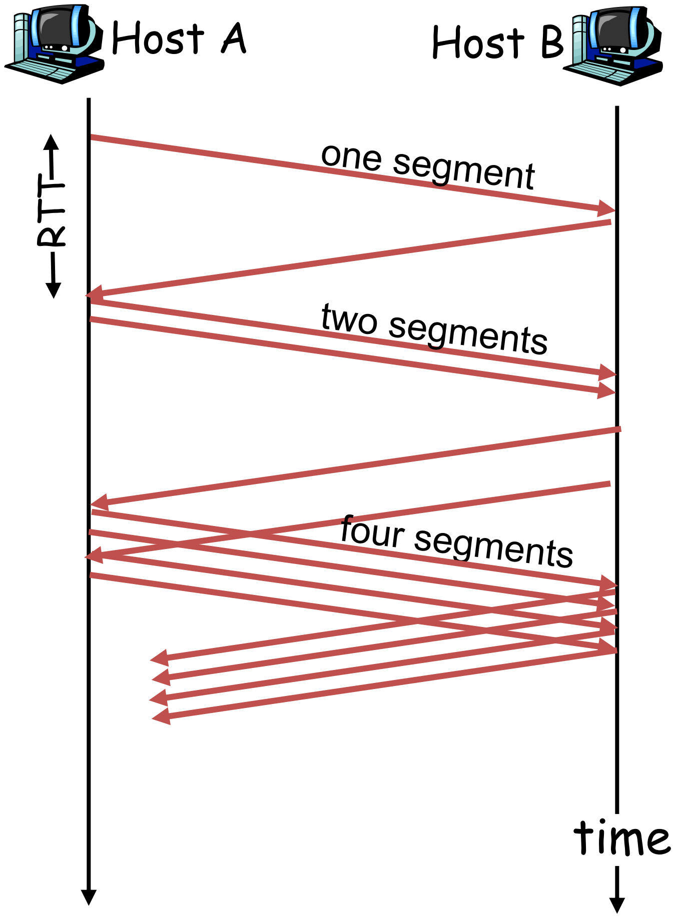
TCP Slow Start

- When connection begins, **cwnd ≤ 2 MSS**, typically, set cwnd = 1MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate

- When connection begins, increase rate exponentially fast until cwnd reaches a threshold value: slow-start-threshold *ssthresh*
  - cwnd < ssthresh

TCP Slow Start (more)

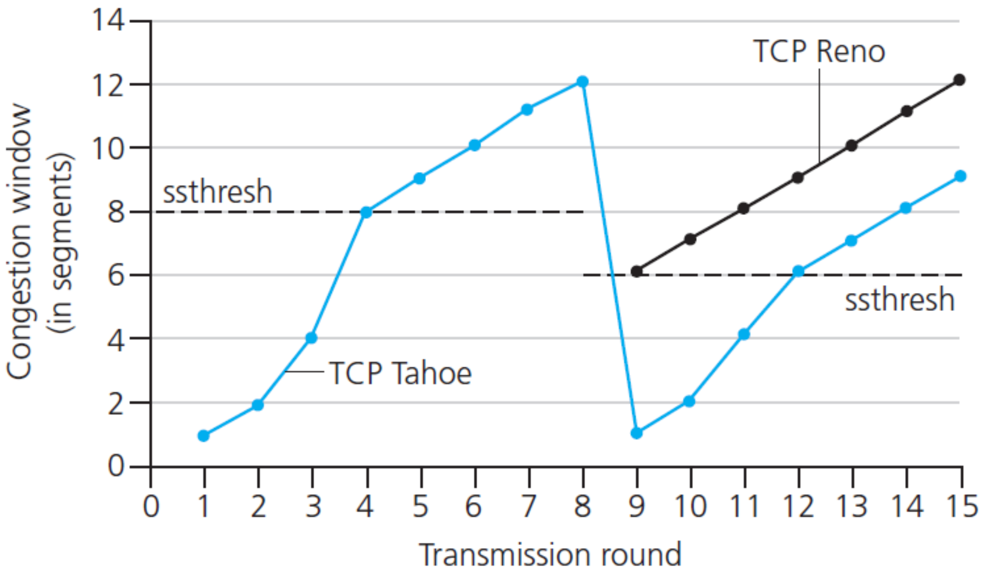
- When connection begins, increase rate exponentially when cwnd<ssthresh
  - Goal: double **cwnd** every RTT by setting
  - **Action: cwnd** += 1 MSS for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.



Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Congestion Avoidance

- Goal: increase cwnd by 1 MSS per RTT until congestion (loss) is detected
  - Conditions: when cwnd > ssthresh and no loss occurs
  - Actions: cwnd += (MSS/cwnd)*MSS (bytes) upon every incoming non-duplicate ACK

TCP Congestion Control

Algoritms	condition	Design	action
Slow Start	cwnd <= ssthresh;	cwnd doubles per RTT	cwnd+=1MSS per ACK
Congestion Avoidance	cwnd > ssthresh	cwnd++ per RTT (additive increase)	cwnd+=1/cwnd * MSS per ACK

When loss occurs

- Detecting losses and reacting to them:
  - through duplicate ACKs
    - fast retransmit / fast recovery
      - Goal: multiplicative decrease cwnd upon loss
  - through retransmission timeout
      - Goal: reset everything

Fast Retransmit/Fast Recovery

- fast retransmit: to detect and repair loss, based on incoming duplicate ACKs
  - **use** 3 duplicate ACKs to infer packet loss
  - set ssthresh = max(cwnd/2, 2MSS)
  - cwnd = ssthresh + 3MSS
  - retransmit the lost packet
- fast recovery: governs the transmission of new data until a non-duplicate ACK arrives
  - **increase** cwnd by 1 MSS upon every duplicate ACK

Philosophy:

- ❑ 3 dup ACKs to infer losses and differentiate from transient out-of-order delivery
- ❑ What about only 1 or 2 dup ACKs?
  - ❑ Do nothing; this allows for transient out-of-order delivery
- ❑ receiving each duplicate ACK indicates one more packet left the network and arrived at the receiver

Putting them together

- Initially, fastretx = false;
- If upon 3rd duplicate ACK
  - ssthresh = max (cwnd/2, 2*MSS)
  - cwnd = ssthresh + 3*MSS
    - why add 3 packets here?
  - retransmit the lost TCP packet
  - Set fastretx = true;
- If fastretx == true; upon each *additional* duplicate ACK
  - cwnd += 1 MSS
  - transmit a new packet if allowed
    - by the updated cwnd and rwnd
- If fastretx == true; upon a new (i.e., non-duplicate) ACK
  - cwnd = ssthresh
  - Fastretx = false; // After fast retx/fast recovery, cwnd decreases by half

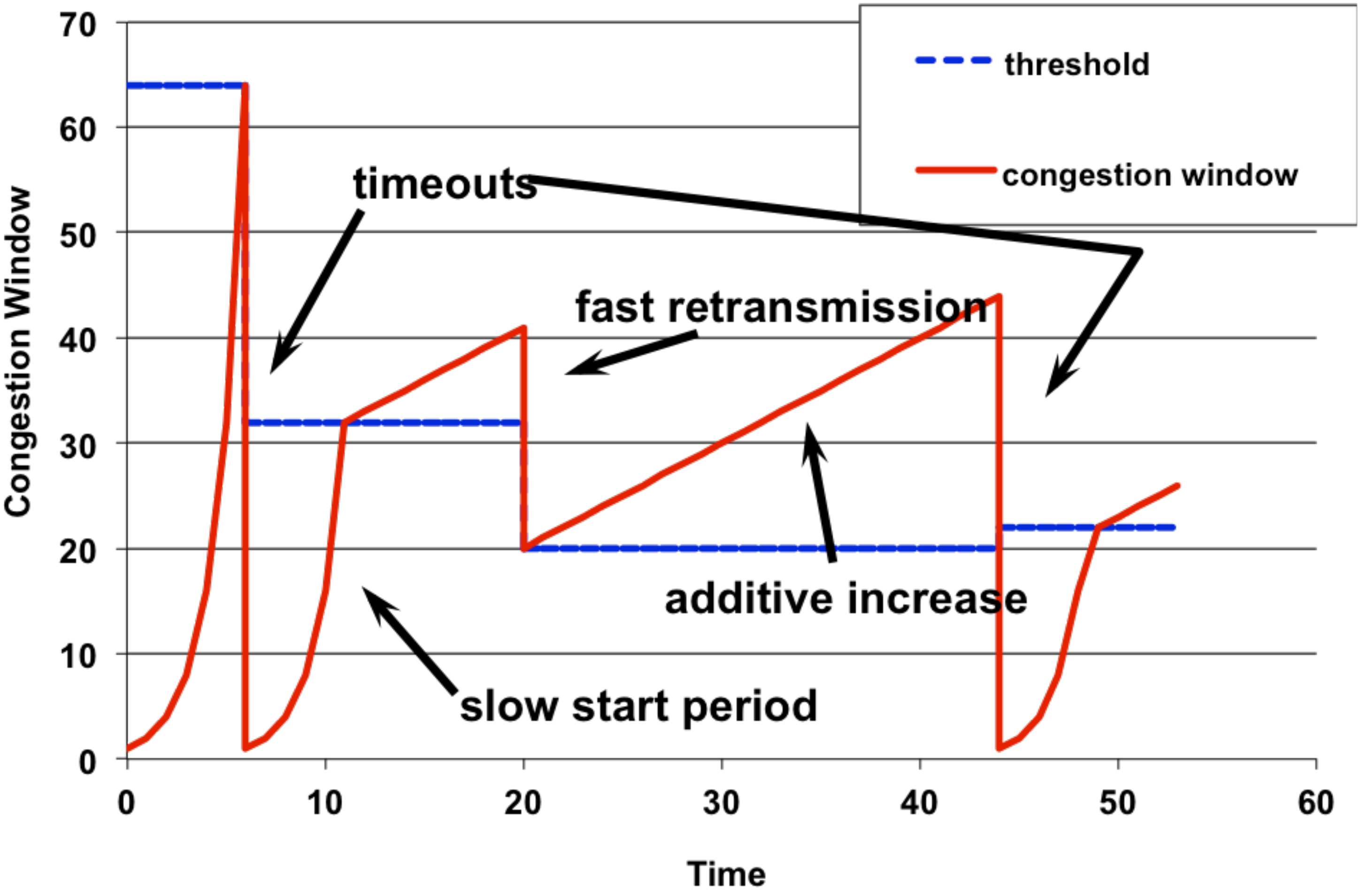
Retransmission Timeout

when retransmission timer expires

- ssthresh = max (cwnd/2, 2*MSS)
  - cwnd should be flight size to be more accurate
  - see RFC 2581
- cwnd = 1 MSS
- retransmit the lost TCP packet

- why resetting?
  - heavy loss detected

TCP Congestion Window Trace



TCP Congestion Control

Algoritms	condition	Design	action
Slow Start	cwnd <= ssthresh;	cwnd doubles per RTT	cwnd+=1MSS per ACK
Congestion Avoidance	cwnd > ssthresh	cwnd++ per RTT (additive increase)	cwnd+=1/cwnd * MSS per ACK

fast retransmit	3 duplicate ACK	reduce the cwnd by half (multicative decreasing)	ssthresh = max(cwnd/2,2)
			cwnd = ssthresh + 3 MSS;
			retx the lost packet

fast recovery	receiving a new ACK after fast retx	finish the 1/2 reduction of cwnd in fast retx/fast recovery	cwnd = ssthresh; tx if allowed by cwnd
	upon a dup ACK after fast retx before fast recovery	("transition phrase)	cwnd +=1MSS; Note: it is different from slow start.

RTO timeout	time out	Reset everything	ssthresh = max(cwnd/2,2)
			cwnd = 1;
			retx the lost packet

Practice

- The receiver acknowledges every segment, and the sender always has data to transmit.
- Initially ***ssthresh*** at the sender is set to 4. Assume ***cwnd*** and ***ssthresh*** are measured in segments.

- Assumptions for simplification
  - When ssthresh ==cwnd, use slow start algorithm
  - In congestion avoidance, let us set cwnd = cwnd + 1/[cwnd]
  - All data delivery is done in segments, so we can send the interger number of segments (for example, cwnd = 2.5MSS, we can send 2 segments)
  - All out-of-order segments will be buffered at the receiver side

Illustrative Example

Example Setting

- Use all following TCP congestion control algorithms:
  - Slow start
  - Congestion avoidance (CA)
  - Fast retransmit/fast recovery
  - Retransmission timeout (say, RTO=500ms)
- When cwnd=ssthresh, use slow start algorithm (instead of CA)
- Assume rwnd is always large enough, then the window size by selective repeat (SR) is =cwnd
- Assume 1 acknowledgement per packet, and we use TCP cumulative ACK (i.e., ACK # = (largest sequence # received in order at the receiver + 1))
- Assume each packet size is 1 unit (1B) for simple calculation
- TCP sender has infinite packets to send, 1, 2, 3, 4, 5,….
- Assume packet #5 is lost once

We will how TCP congestion control algorithms work together with SR

CC algorithm

SR after algo runs

slow start
ssh =4

cwnd =1

1				

slow start (upon ack2)

ssh =4

cwnd =1+1=2

2	3			

slow start (upon ack3)
ssh =4

cwnd =2+1=3

3	4	5		

slow start (upon ack4)
ssh =4

cwnd =3+1=4

4	5	6	7	

slow start (upon ack5)
ssh =4

cwnd =4+1=5

5	6	7	8	9	

Do nothing upon ack5 (1st dup)

Do nothing upon ack5 (2nd| dup)

Fast retransmit (upon 3 dup ack5)

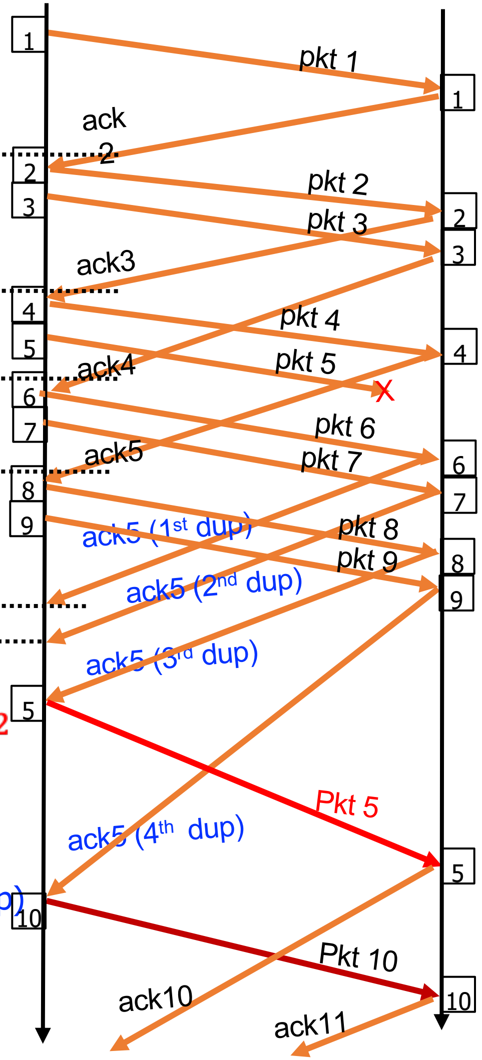
ssh = max(2, 5/2) = 2.5≈ 2
cwnd = ssh + 3 = 5
Retx pkt 5

5	6	7	8	9	

Fast recovery w/ additional dup ACK (upon 4th dup)

ssh = 2, cwnd = 5 +1 =6
send pkt 10

5	6	7	8	9	10



CC algorithm SR after algo runs

Fast recovery w/ additional dup ACK (upon 4th dup)

~~ssh = 2, cwnd = 5 +1 =6; send pkt 10~~

5	6	7	8	9	10

Fast recovery w/ a new ACK (upon ack10)

ssh =2 cwnd = ssh = 2

Fast retx/fast recovery is over

10	11				

Slow start also upon ack10

ssh =2 cwnd =2 + 1 = 3

Send new packet 12

10	11	12			

Congestion avoidance upon ack11

ssh =2 cwnd = 3 + 1/3 ≈ 3

Send packet 13

11	12	13			

Congestion avoidance upon ack 12

ssh =2 cwnd = 3 + 2/3≈ 3, send pkt14

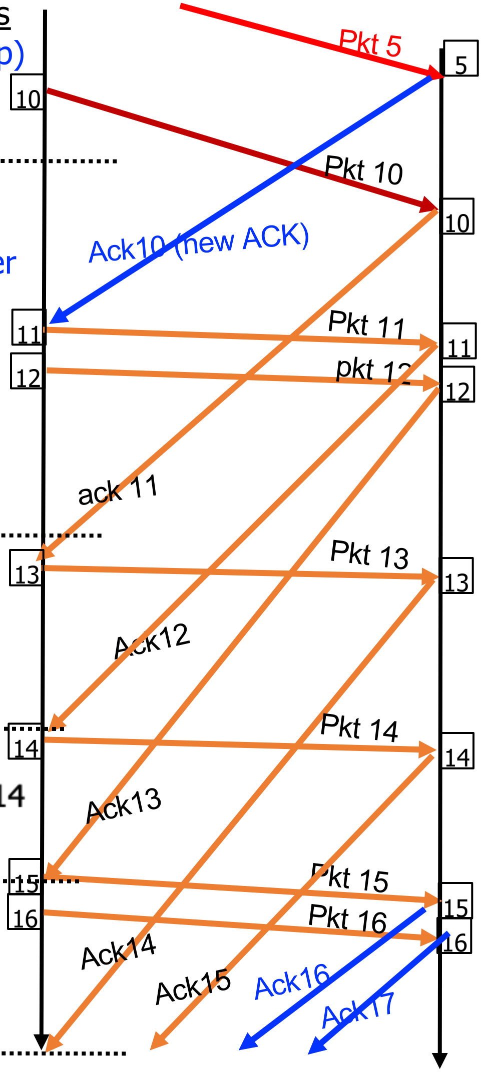
12	13	14			

Congestion avoidance upon ack 13

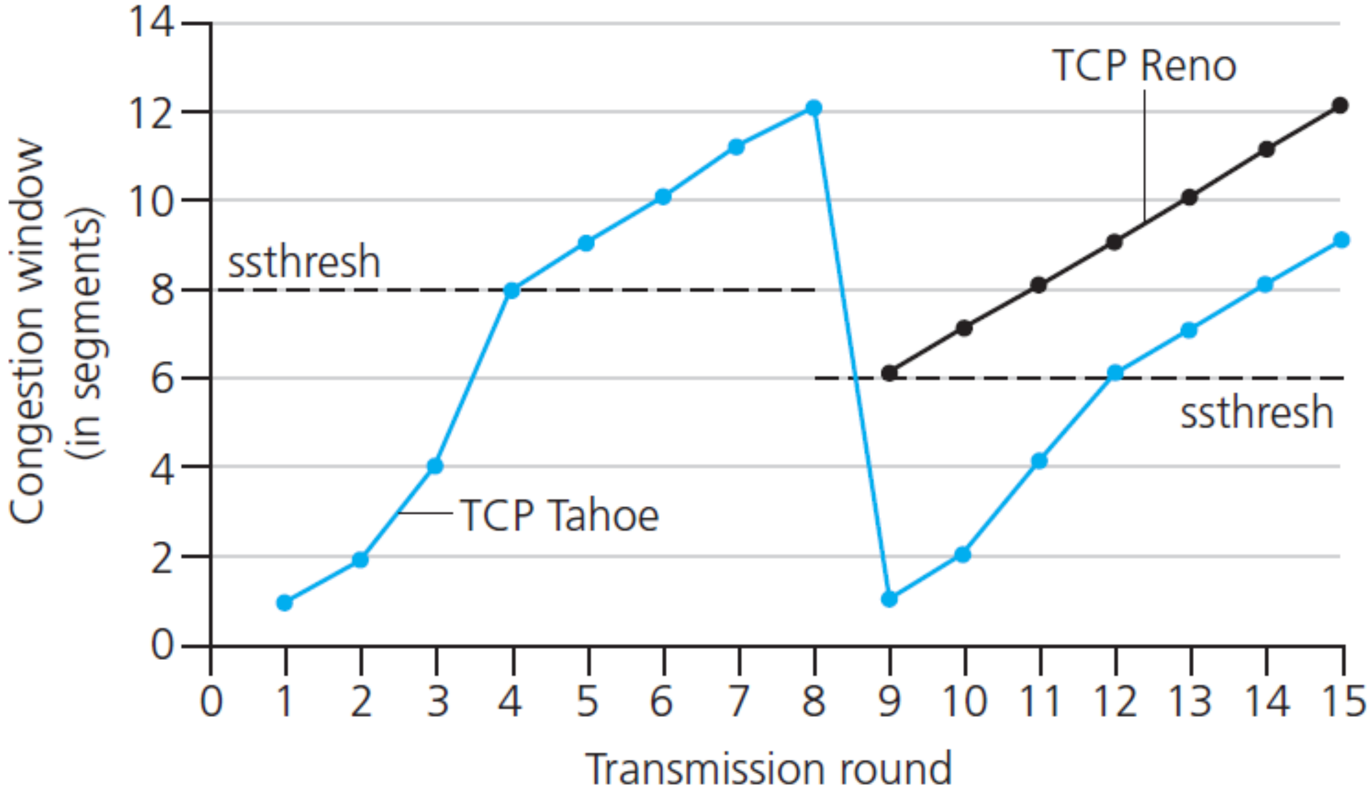
ssh =2 cwnd = 3 + 3/3=4

Send packets 15, 16

13	14	15	16		



Practice: TCP Congestion Window Trace



Putting Things Together in TCP

- use selective repeat to do reliable data transfer for a window of packets *win* at any time

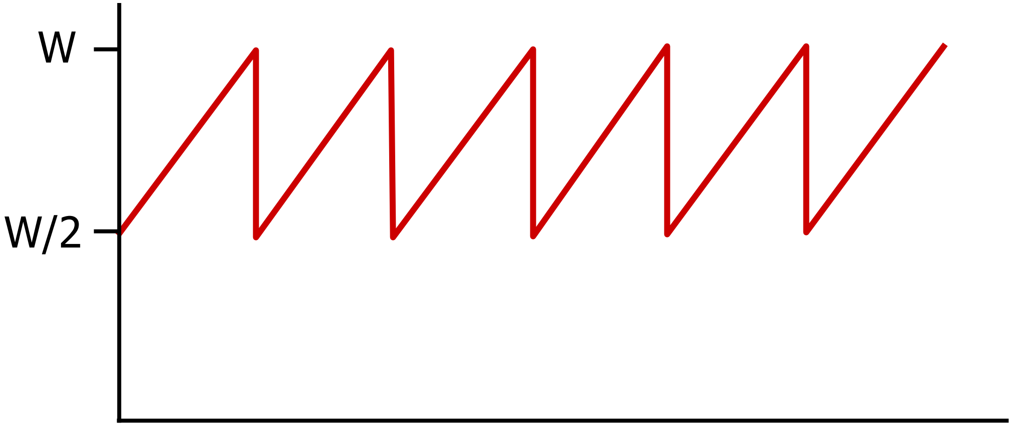
- update win = min (cwnd, rwnd)
  - cwnd is updated by TCP congestion control
  - rwnd is updated by TCP flow control

- Example: cwnd = 20; rwnd = 10
  - Then *win*=10

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) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\,\frac{W}{RTT}\ \text{bytes/sec}$$



TCP Futures: TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of L = $2 \cdot 10^{-10}$ *– a very small loss rate!*

- new versions of TCP for high-speed