Chapter 3: Transport Layer

Applications

… built on …

Reliable (or unreliable) transport

… built on …

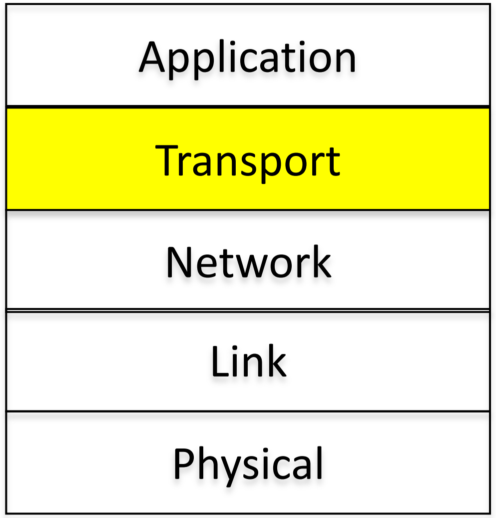
Best-effort global packet delivery

… built on …

Best-effort local packet delivery

… built on …

Physical transfer of bits



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

Chapter 3: Our Goals

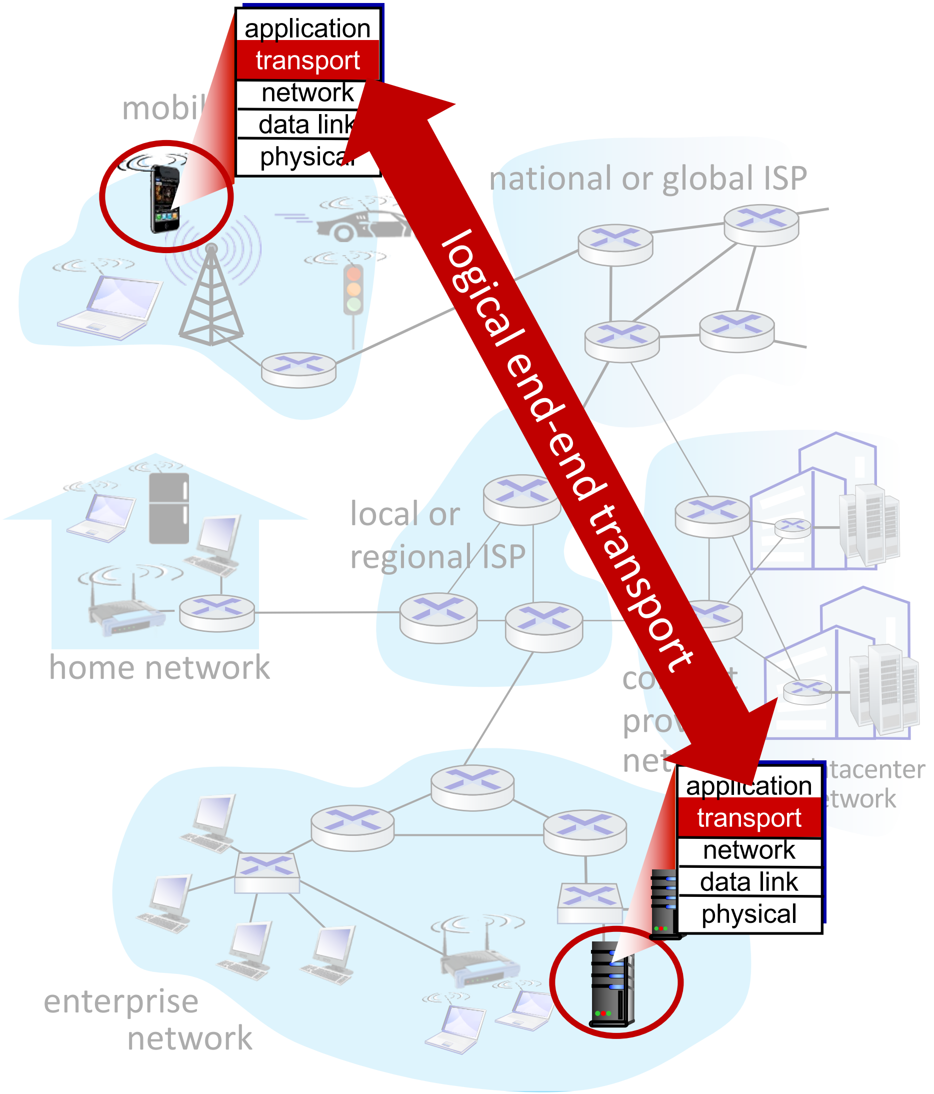
- understand principles behind transport layer services:
  - multiplexing, de-multiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Learn about transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Chapter 3: Outline

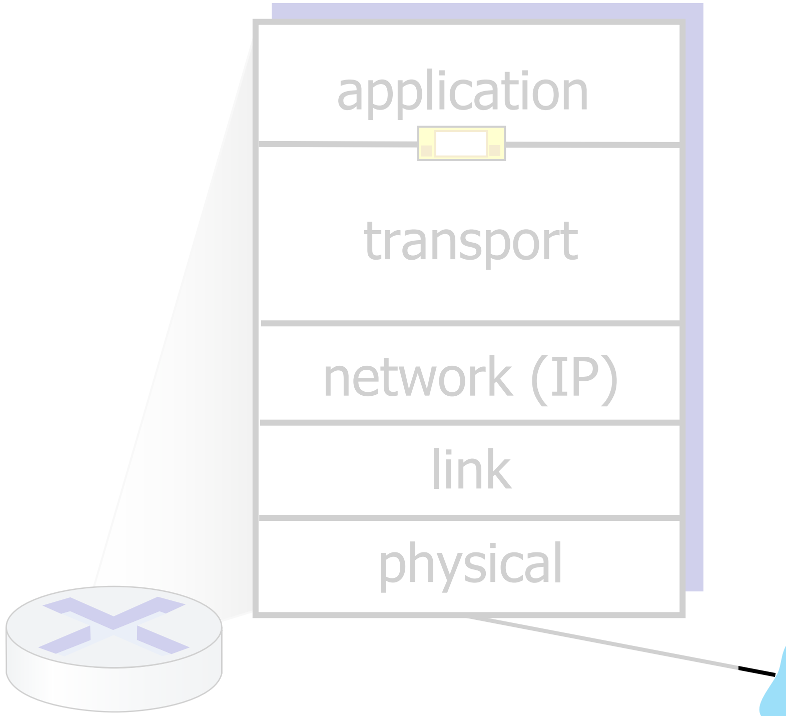
3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
3.6 Principles of congestion control
3.7 TCP congestion control
3.8 Evolution of transport-layer functionality

Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP

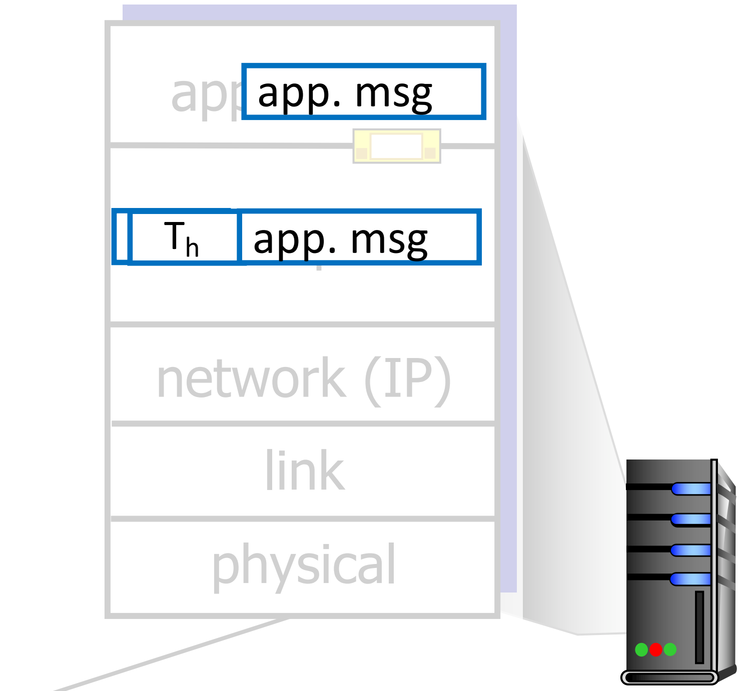


Transport Layer Actions

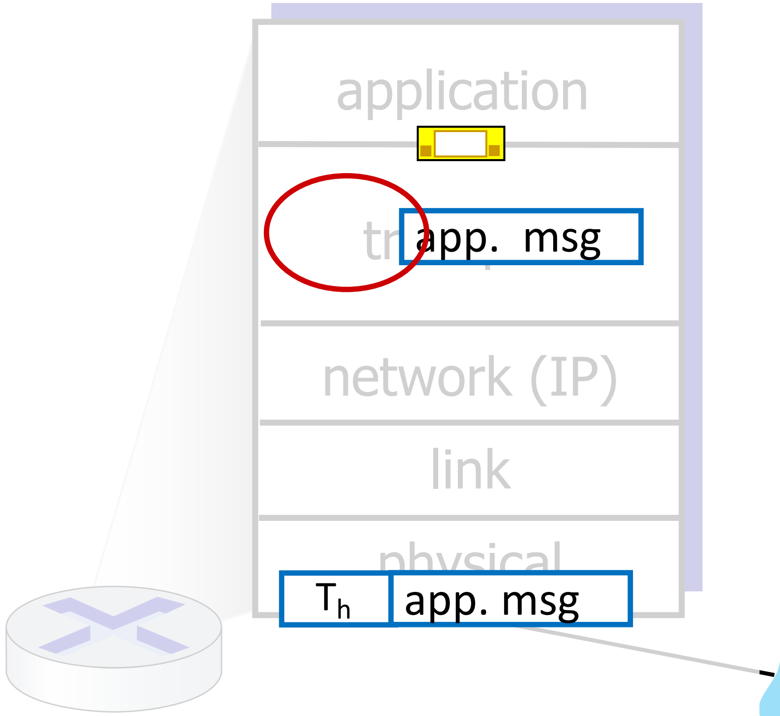


Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP

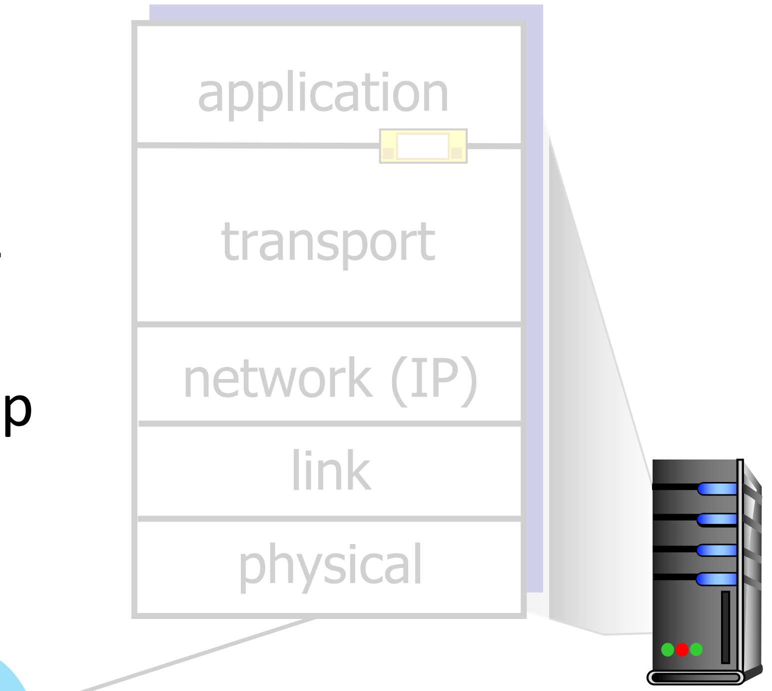


Transport Layer Actions



Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket



Transport vs. network layer services and protocols

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

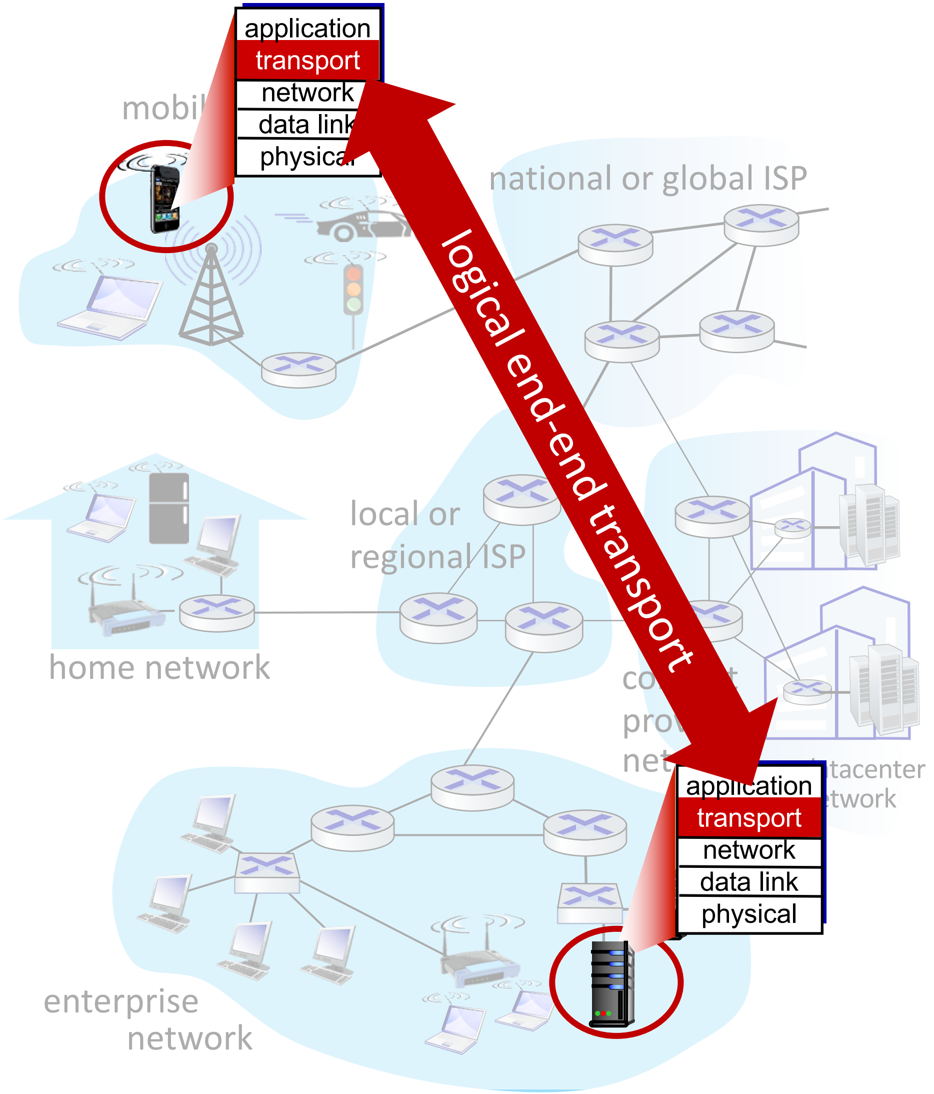
household analogy:

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

- hosts = houses
- processes = kids
- app messages = letters in envelopes

Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

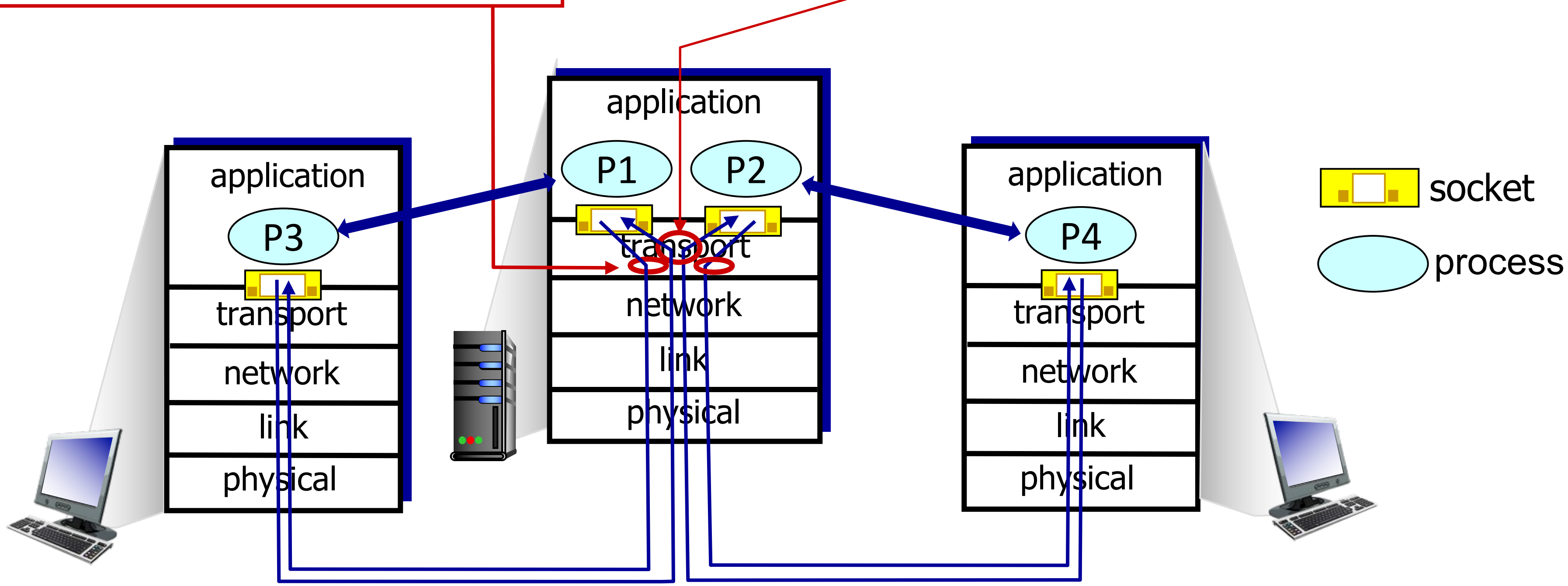
Multiplexing/demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

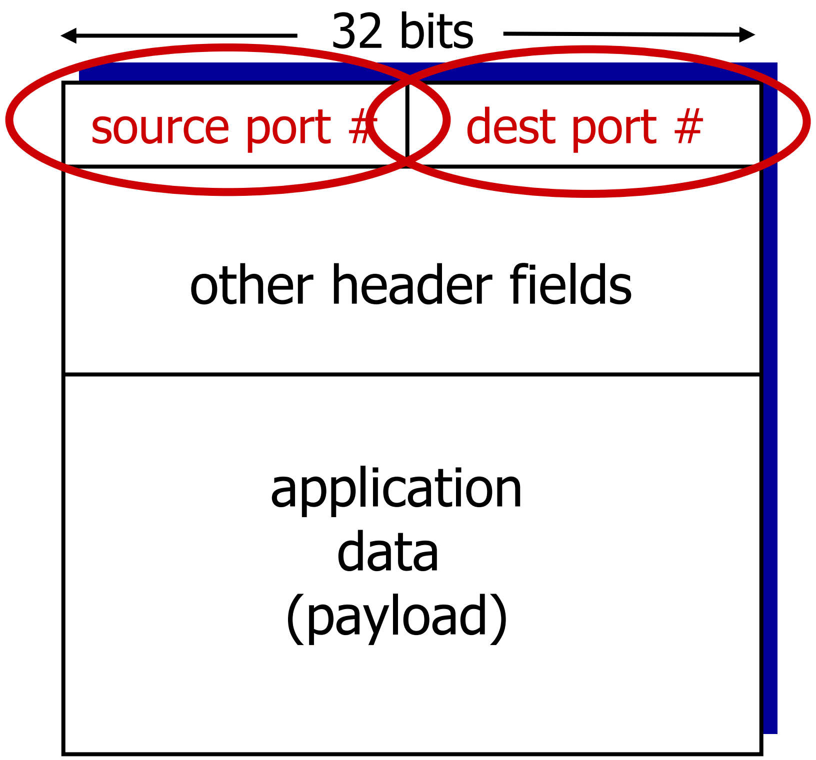
demultiplexing at receiver:

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:

- when creating socket, must specify *host-local* port #:

```
DatagramSocket mySocket1
 = new DatagramSocket(12534);
```

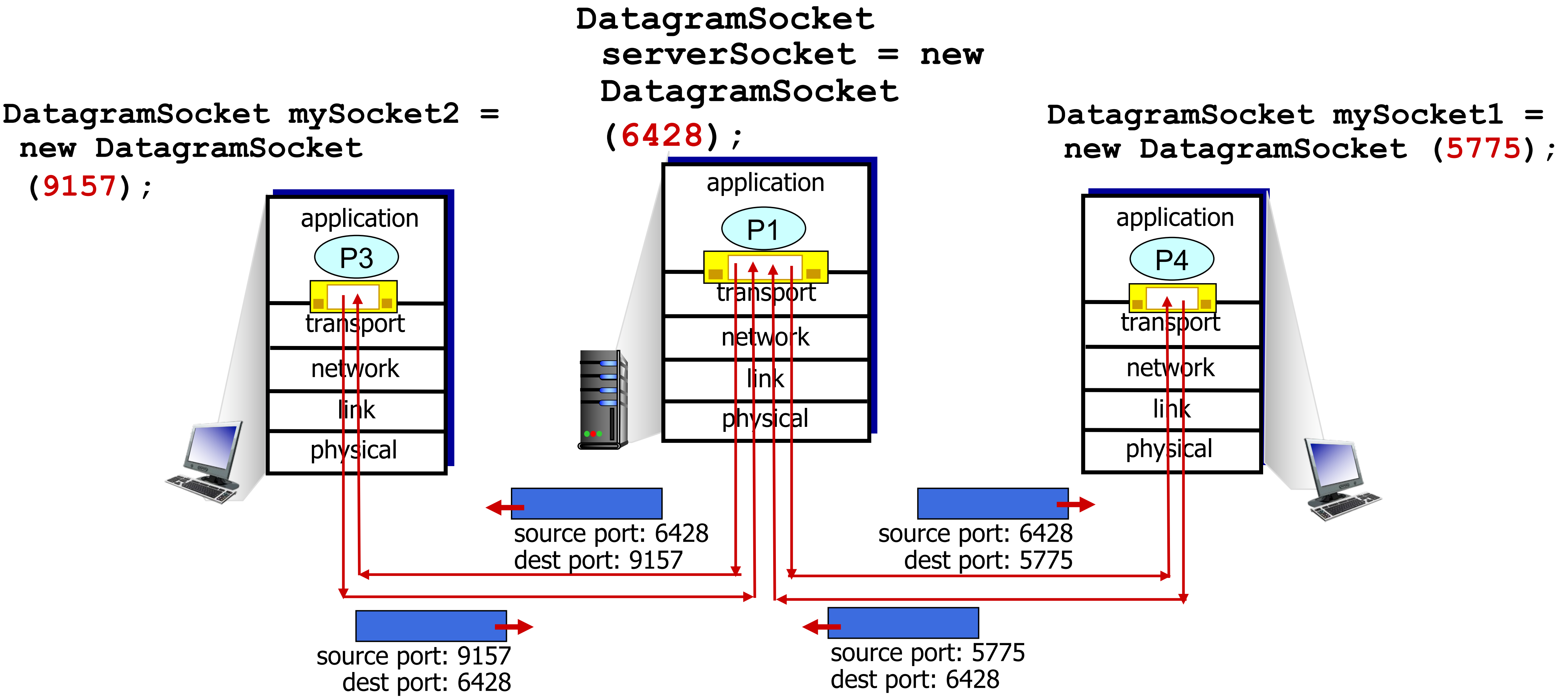
- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

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

Connectionless demultiplexing: an example

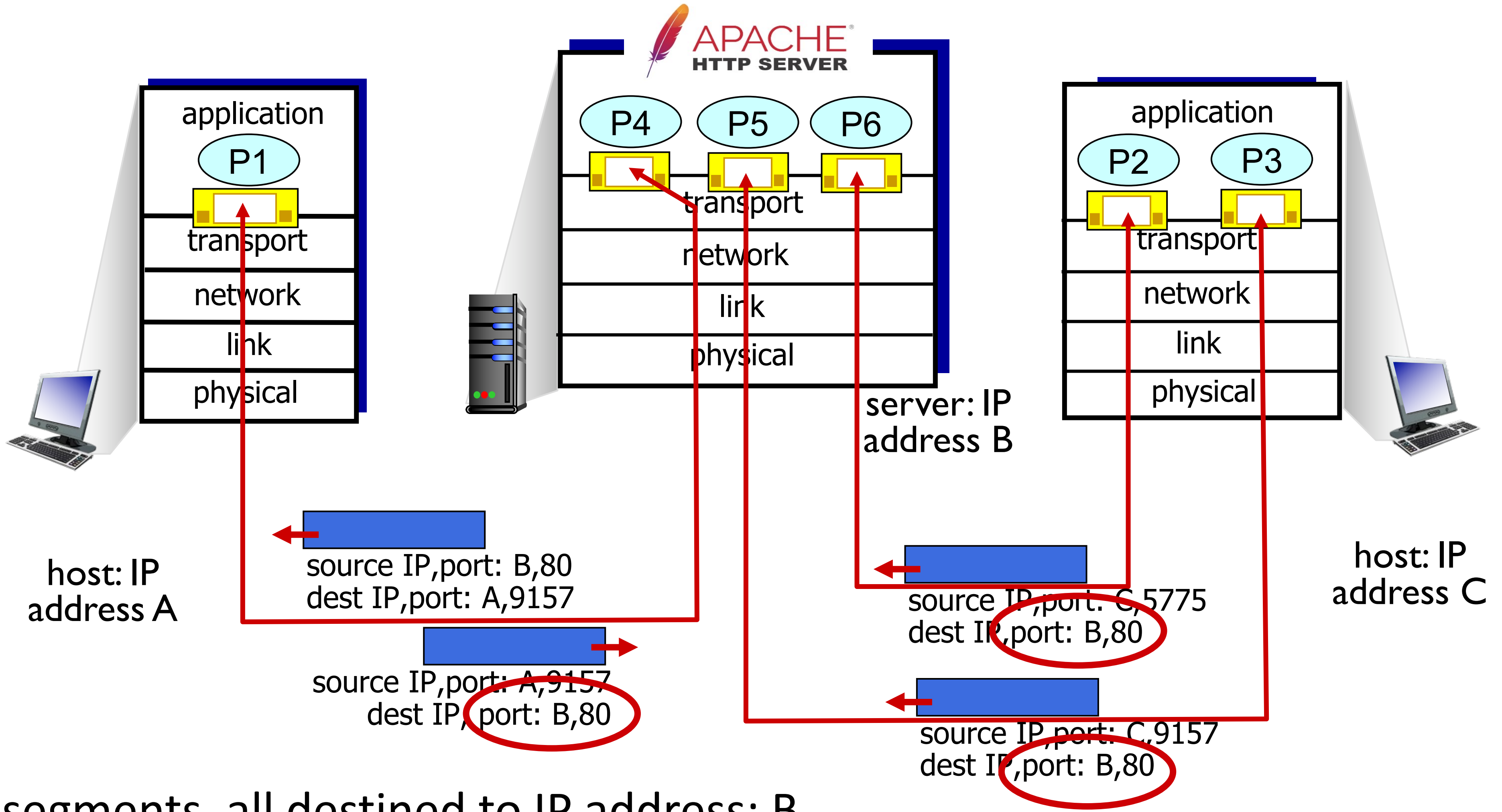


Connection-oriented demultiplexing

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

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

Connection-oriented demultiplexing: example



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

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

UDP: User Datagram Protocol

- "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

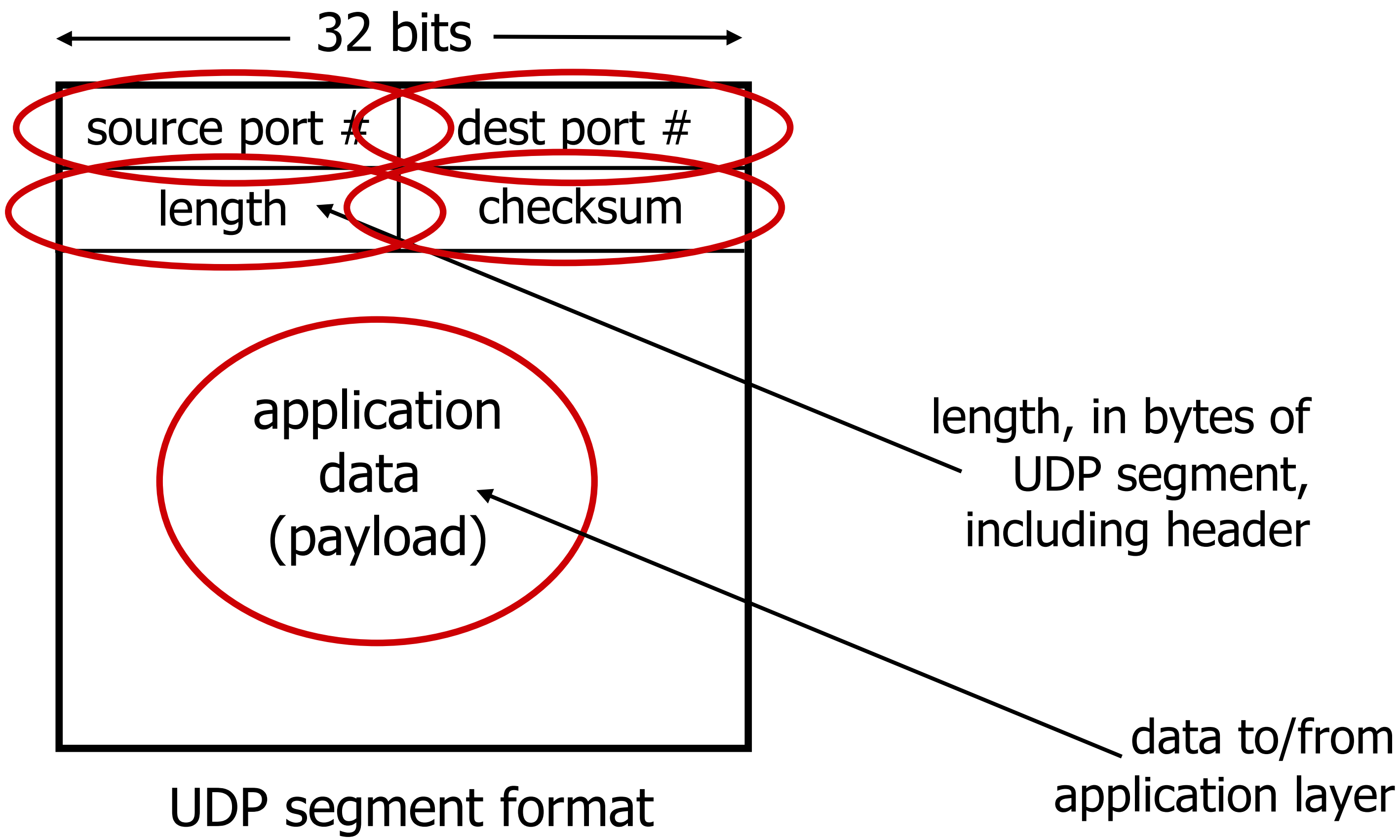
Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

UDP segment header



UDP segment format

UDP checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment

	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11

receiver-computed checksum $\neq$ sender-computed checksum (as received)

UDP checksum

Goal: detect errors (*i.e.*, flipped bits) in transmitted segment

sender:

- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: an 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

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

Internet checksum: weak protection!

example: add two 16-bit integers

	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0	0 1
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	1 0
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	

Even though numbers have changed (bit flips), *no* change in checksum!

In-class practice: UDP checksum

- 1^{st}: 0110
- 2^{nd}: 0101
- 3^{rd}: 1000
- Calculate UDP checksum of 1^{st} + 2^{nd} + 3^{rd}
- sum = 10011, -> 0011 + 1 (carryout) = 0100
- checksum = 1s complement = 1011
- Check: receiving 1011? Passed the check
- Check: receiving 1001? Failed. Error for sure.
- Errors if receiving 1011?? Maybe(if two bits flipped)

Summary: UDP

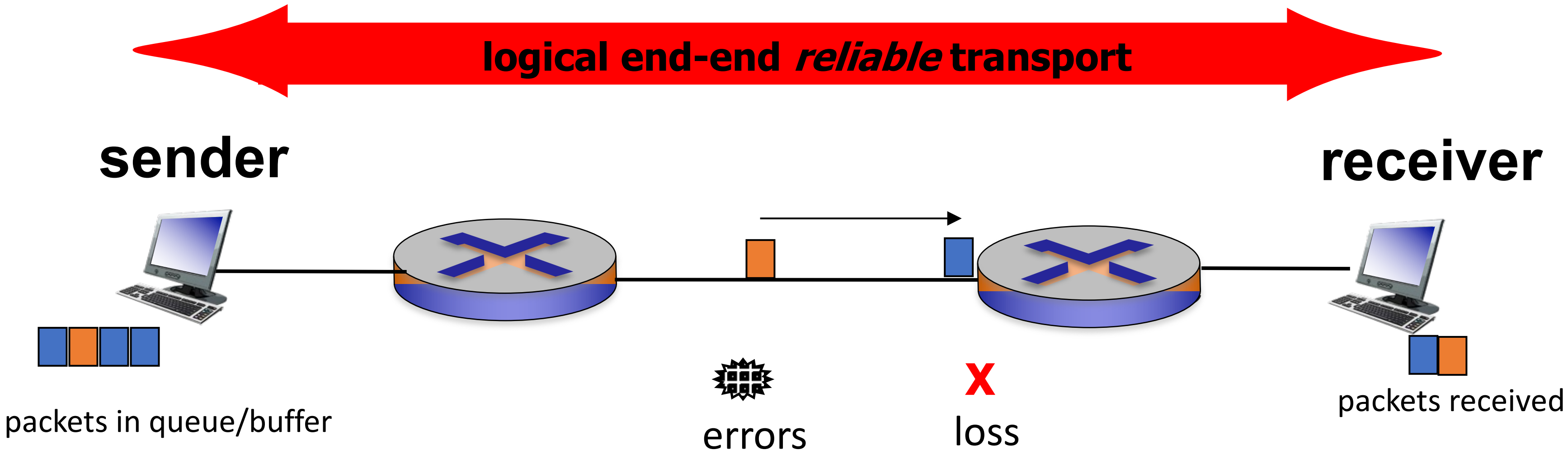
- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Principles of reliable data transfer

- important @ application, transport, link layers
  - Reliable transport of packets
    - A single sender and a single receiver
  - Packet delivery imperfect
    - With bit errors, dropping packets, out-of-order delivery, duplicate copies, long delay, ….

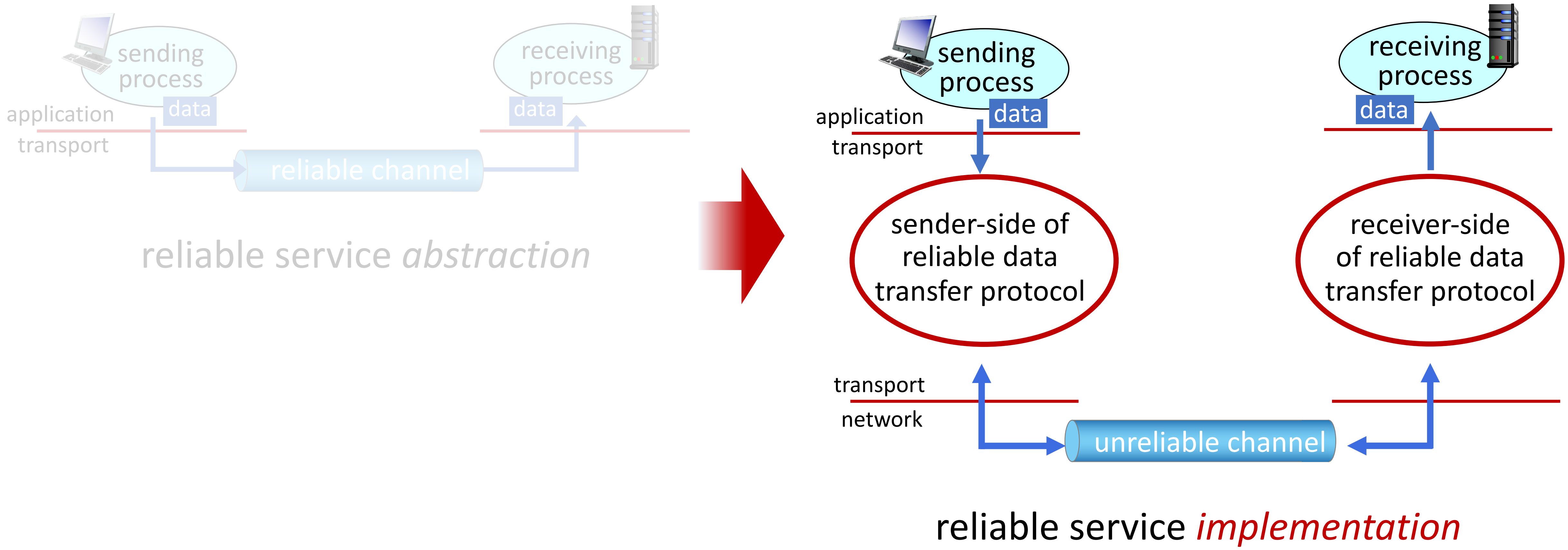


Packet delivery misbehaviors

Principles of reliable data transfer

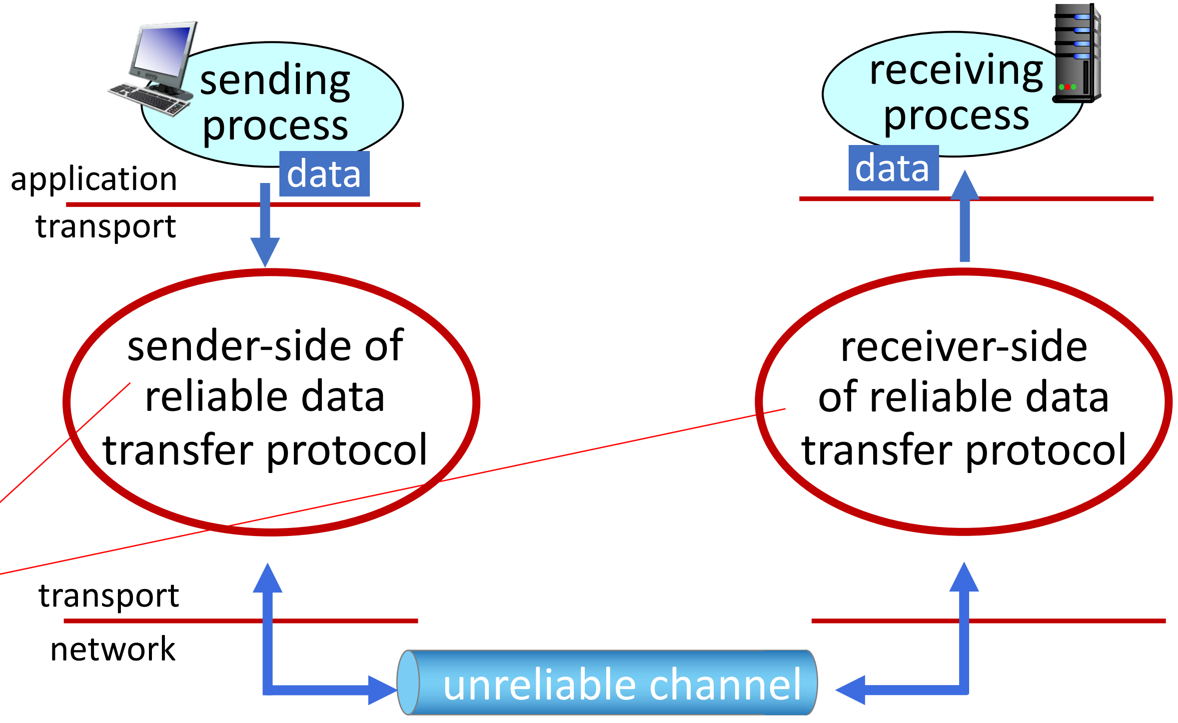
reliable service *abstraction*

Principles of reliable data transfer



Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

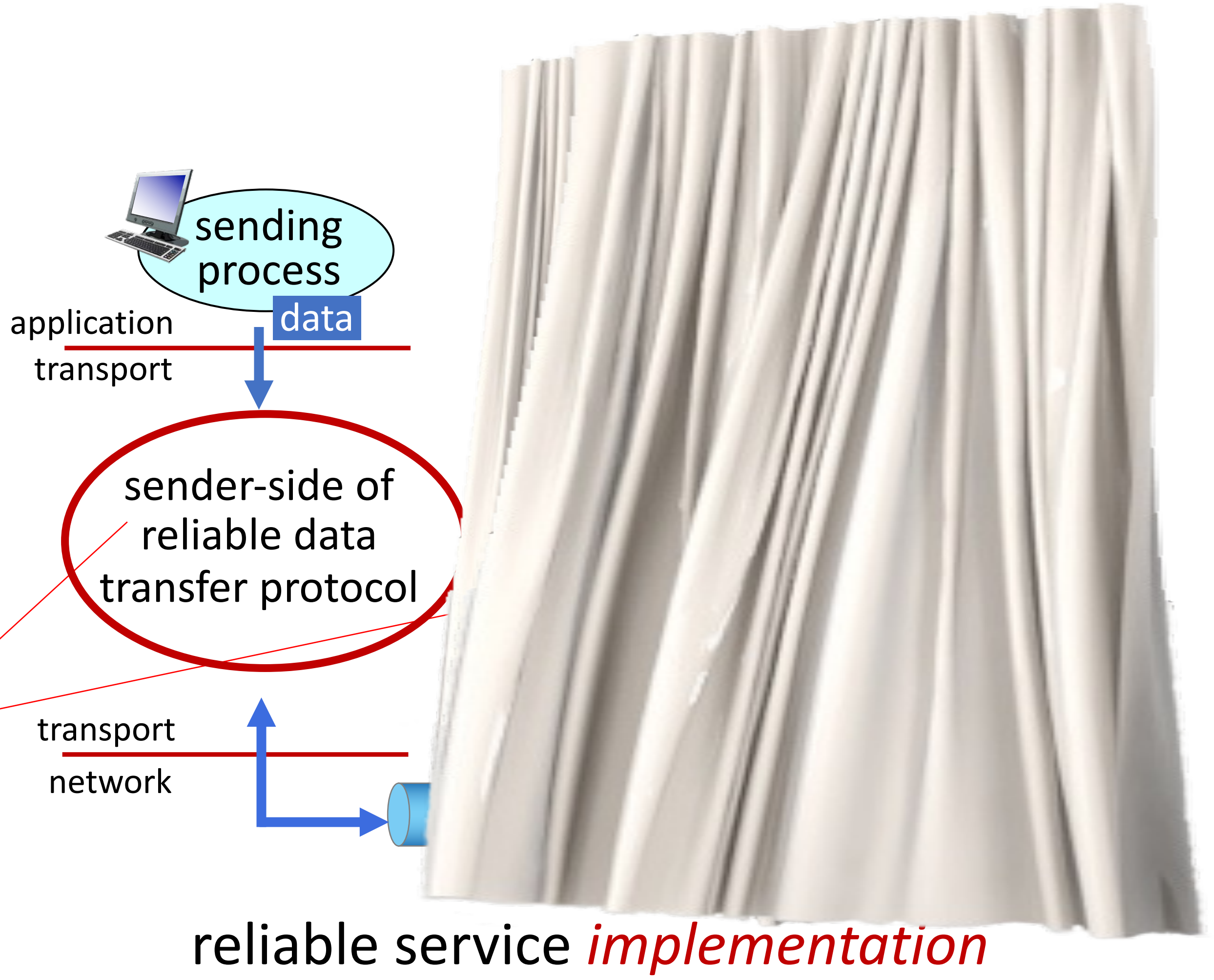


reliable service *implementation*

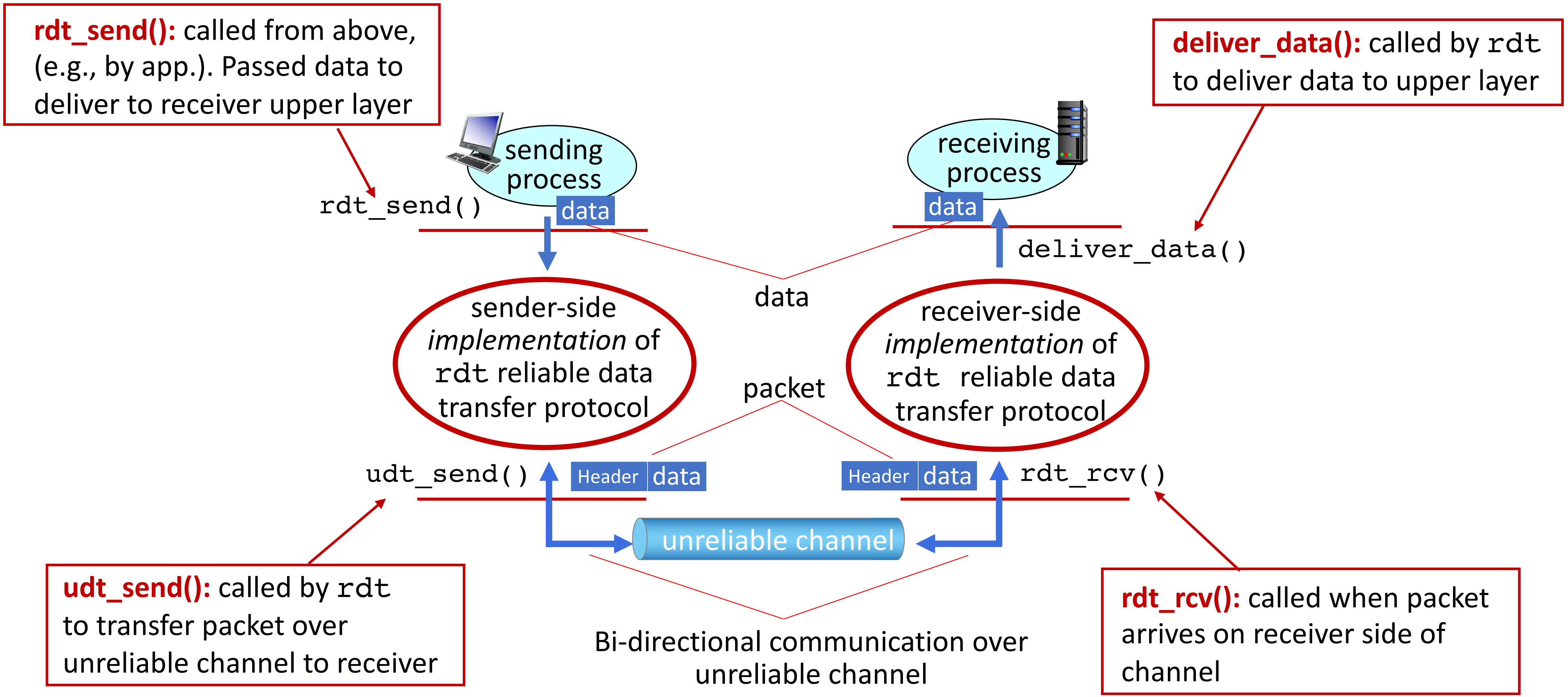
Principles of reliable data transfer

Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- unless communicated via a message



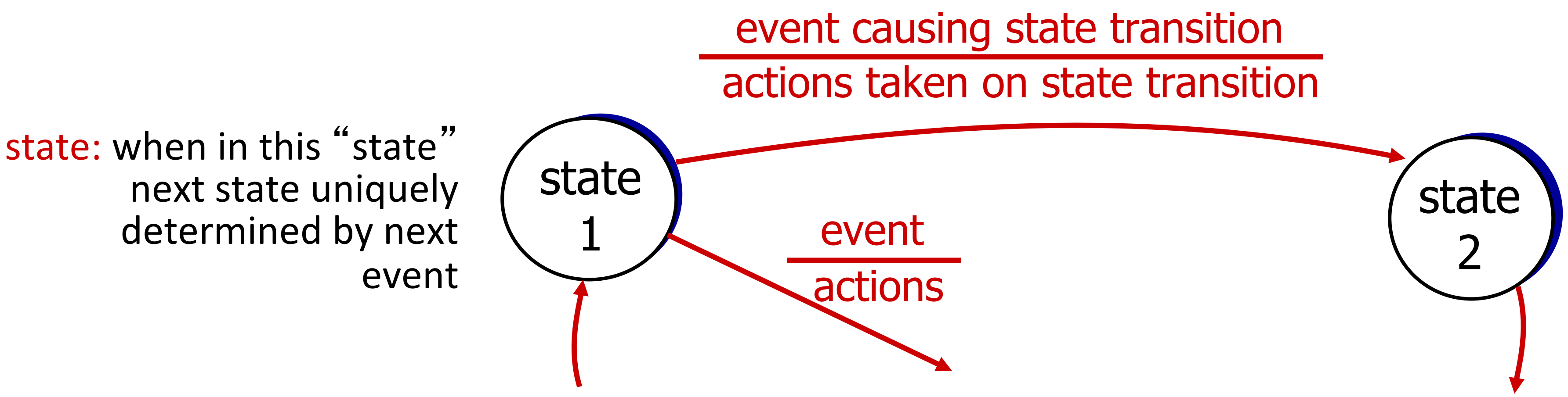
Reliable data transfer protocol (rdt): interfaces



Reliable data transfer: getting started

We will:

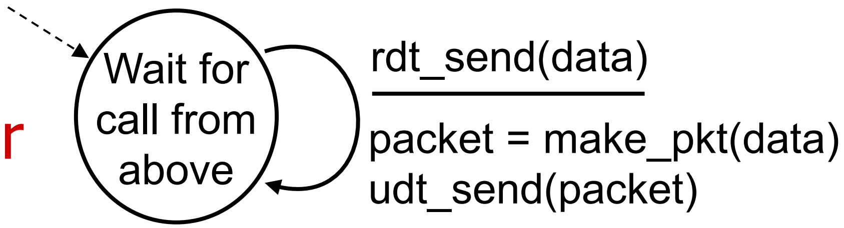
- incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



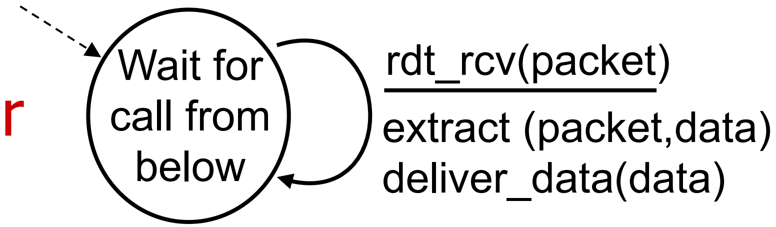
rdt1.0: reliable transfer over a reliable channel

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

sender

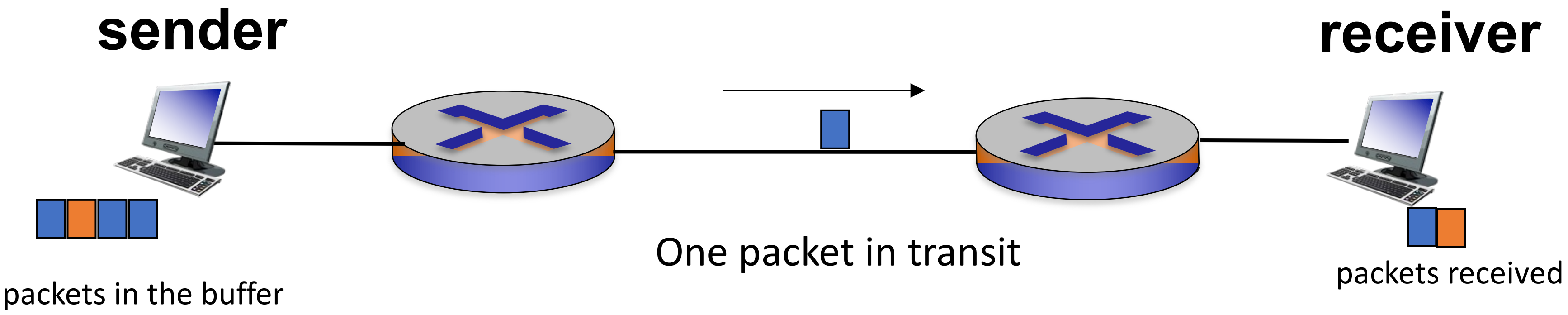


receiver



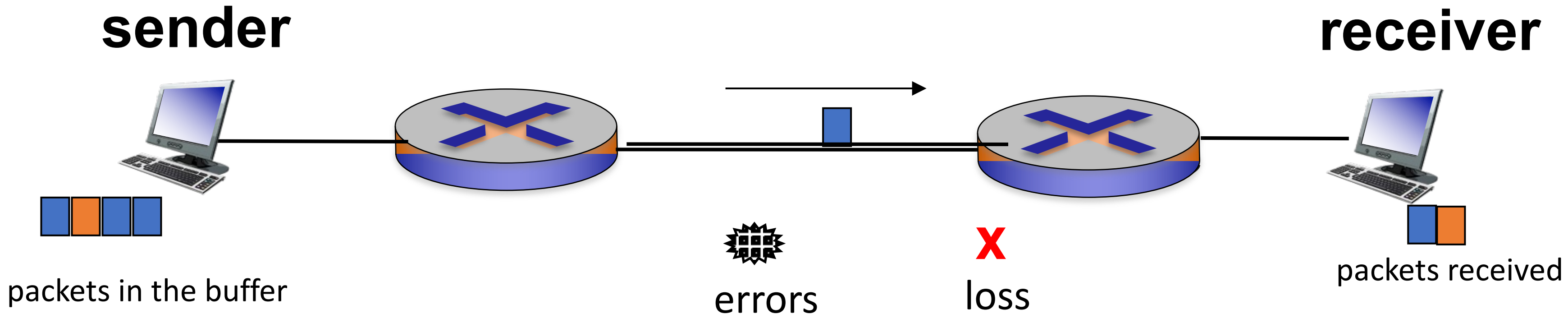
“Stop and Wait” Scenario

- Simple setting: <u>one packet at a time (stop and wait)</u>
  - One sender, one receiver
  - sender has infinite number of packets to transfer to the receiver
  - sender starts one-packet transmission at a time, and will not proceed with the next new packet transmission until the current packet has been successfully received & acknowledged by the receiver.



"Stop and Wait" Scenario

- We progressively consider more complex cases
  - Bit errors
  - Packet loss
  - Duplicate copies of the same packet
  - Long delay (thus also out of order)
  - ….
- Designs: rdt2.0 (initial) → rdt3.0 (stop & wait)



Packet delivery misbehaviors

rdt2.0: channel with bit errors

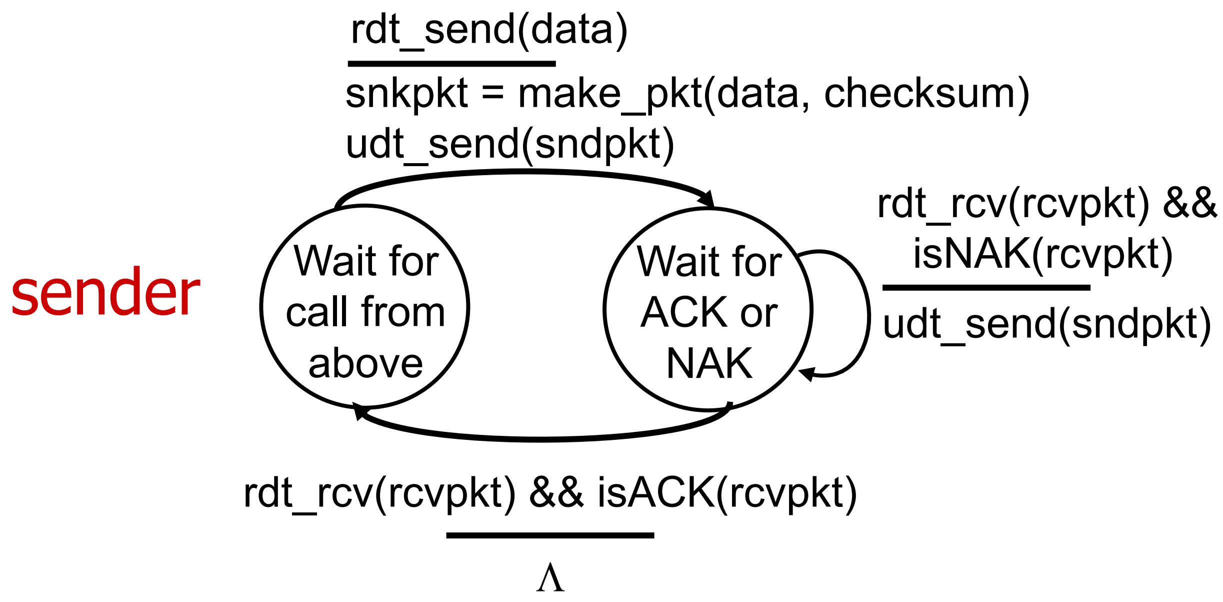
- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

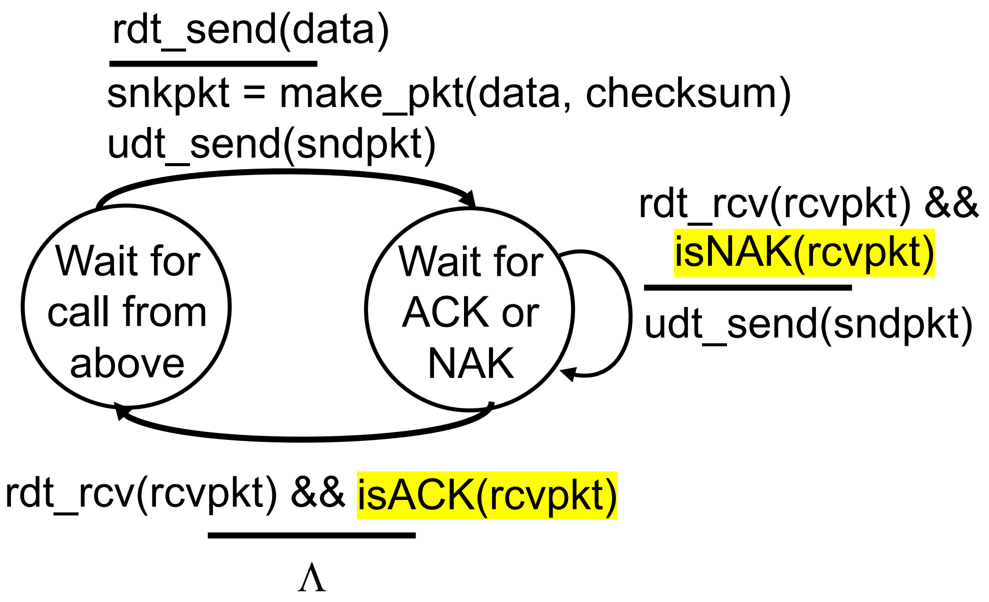
- How to detect bit errors in packet?
  - Internet checksum algorithm
- How to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits packet upon receiving NAK
- **new mechanisms in rdt2.0 (beyond rdt1.0):**
  - Error detection at receiver
  - Feedback from receiver: control messages (ACK,NAK) from receiver to sender
  - Retransmission at the sender upon NAK feedback

rdt2.0: FSM specifications



rdt2.0: FSM specification

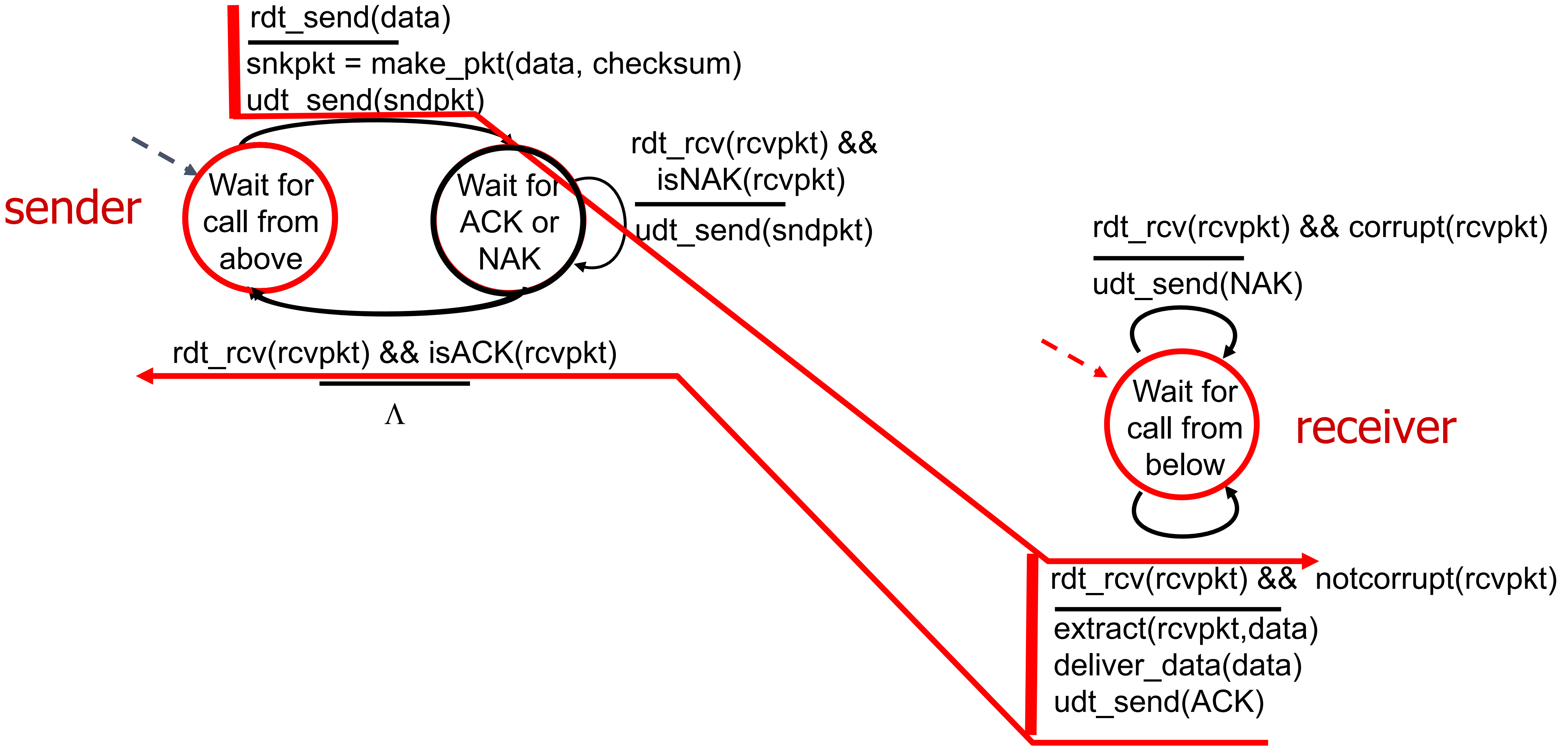
sender



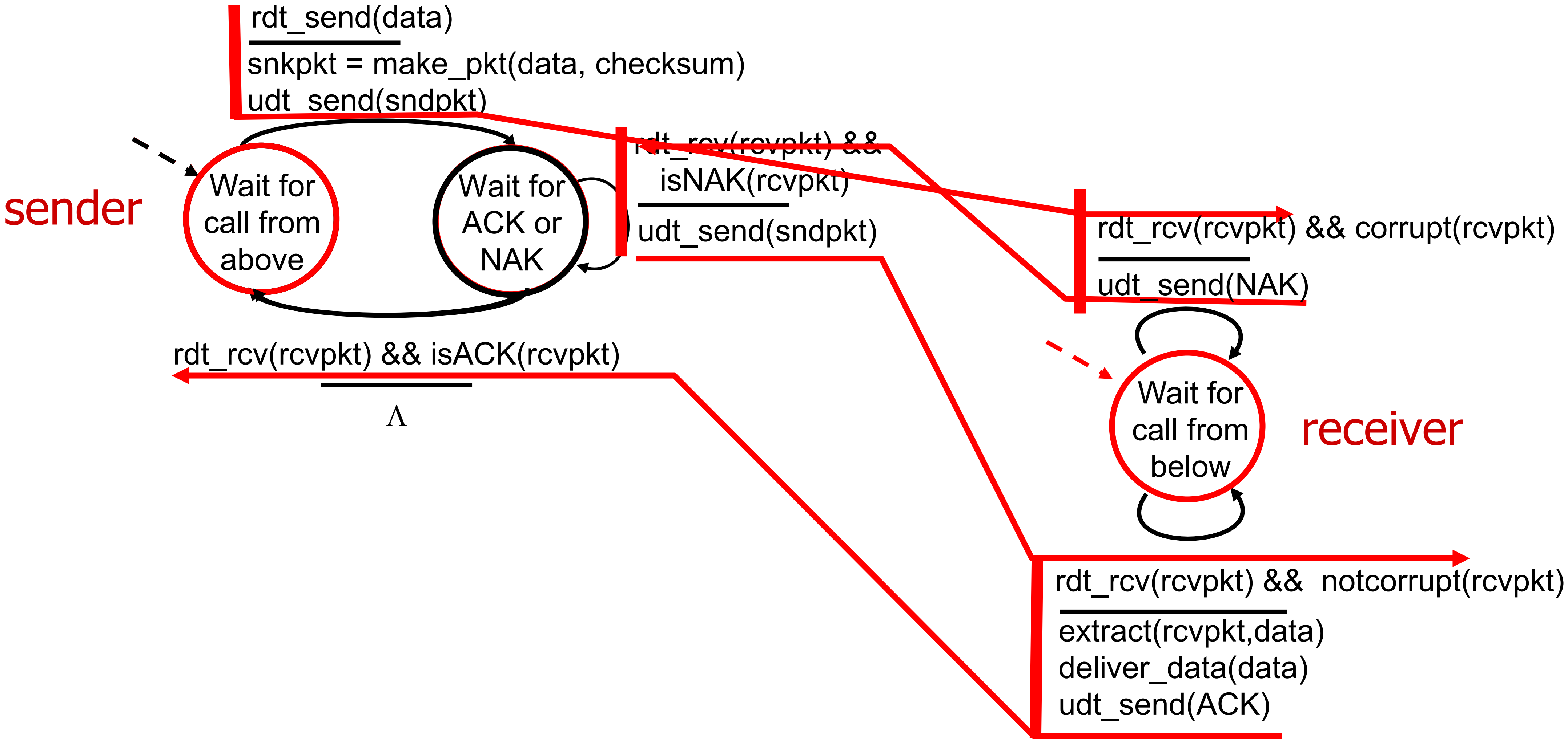
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol!

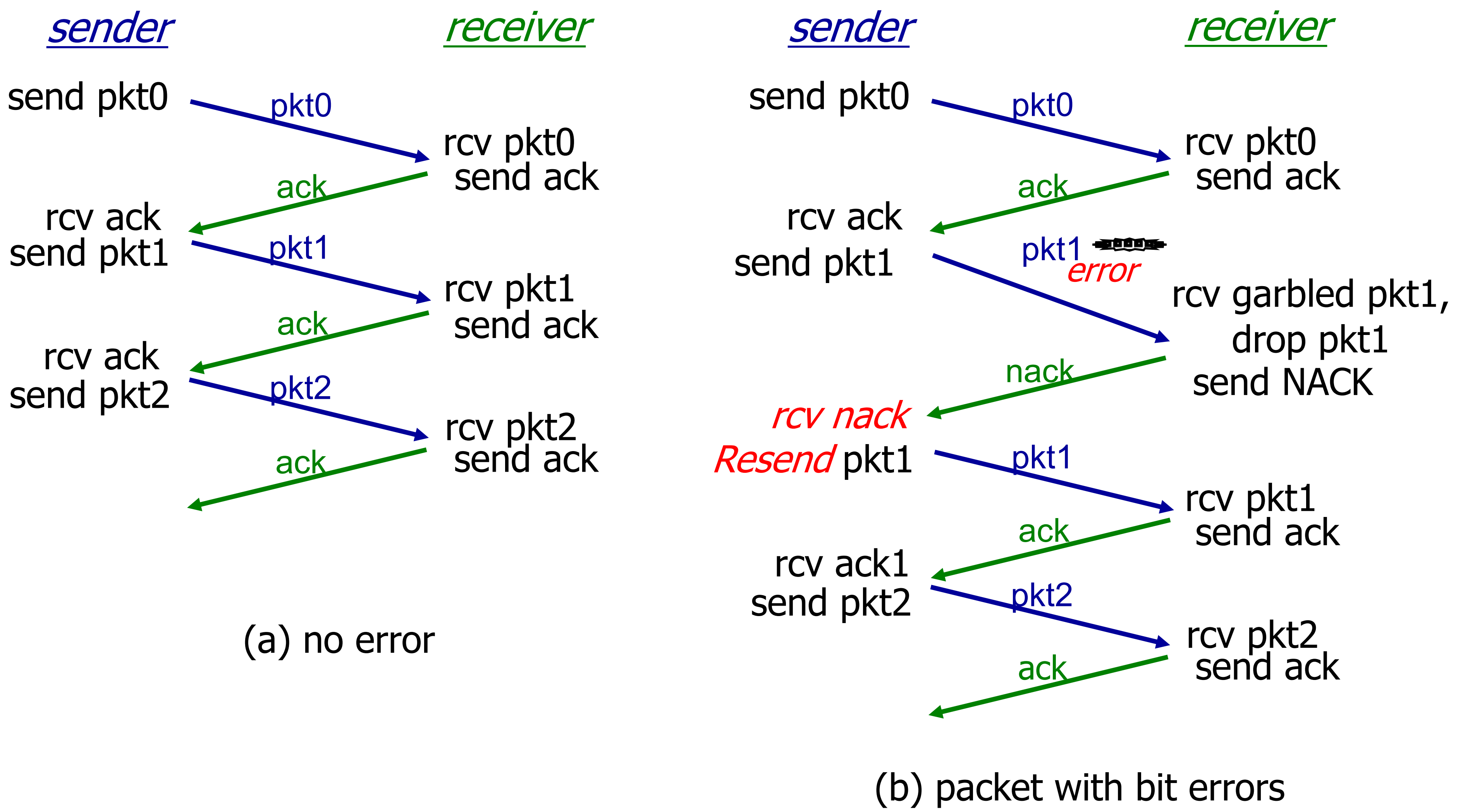
rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



rdt2.0 in action



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

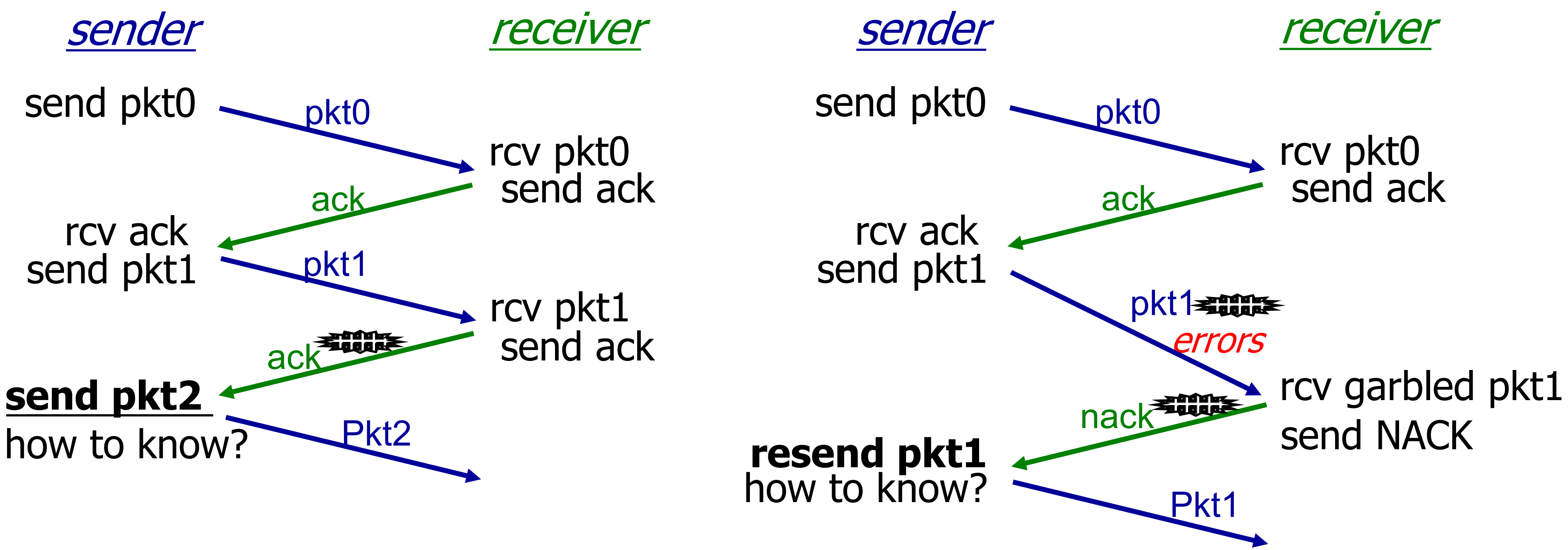
handling duplicates:

- 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

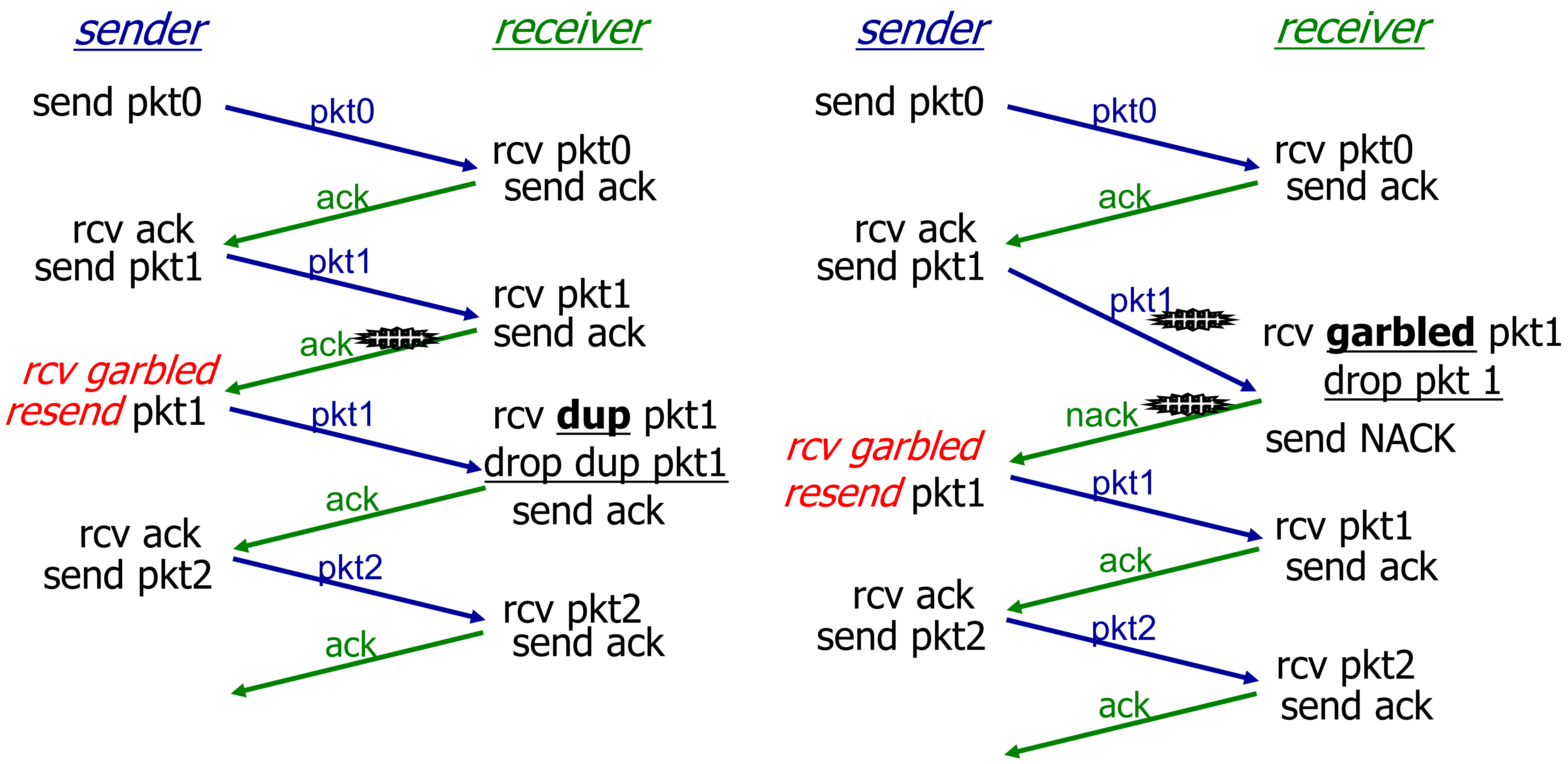
(b) Corrupted NACK

Simply retransmitting upon corrupted ACK/NACK is not sufficient!

Sender cannot tell whether the corrupted message is ACK or NACK!

Receiver cannot tell whether the received message is a new packet or a retransmitted packet!

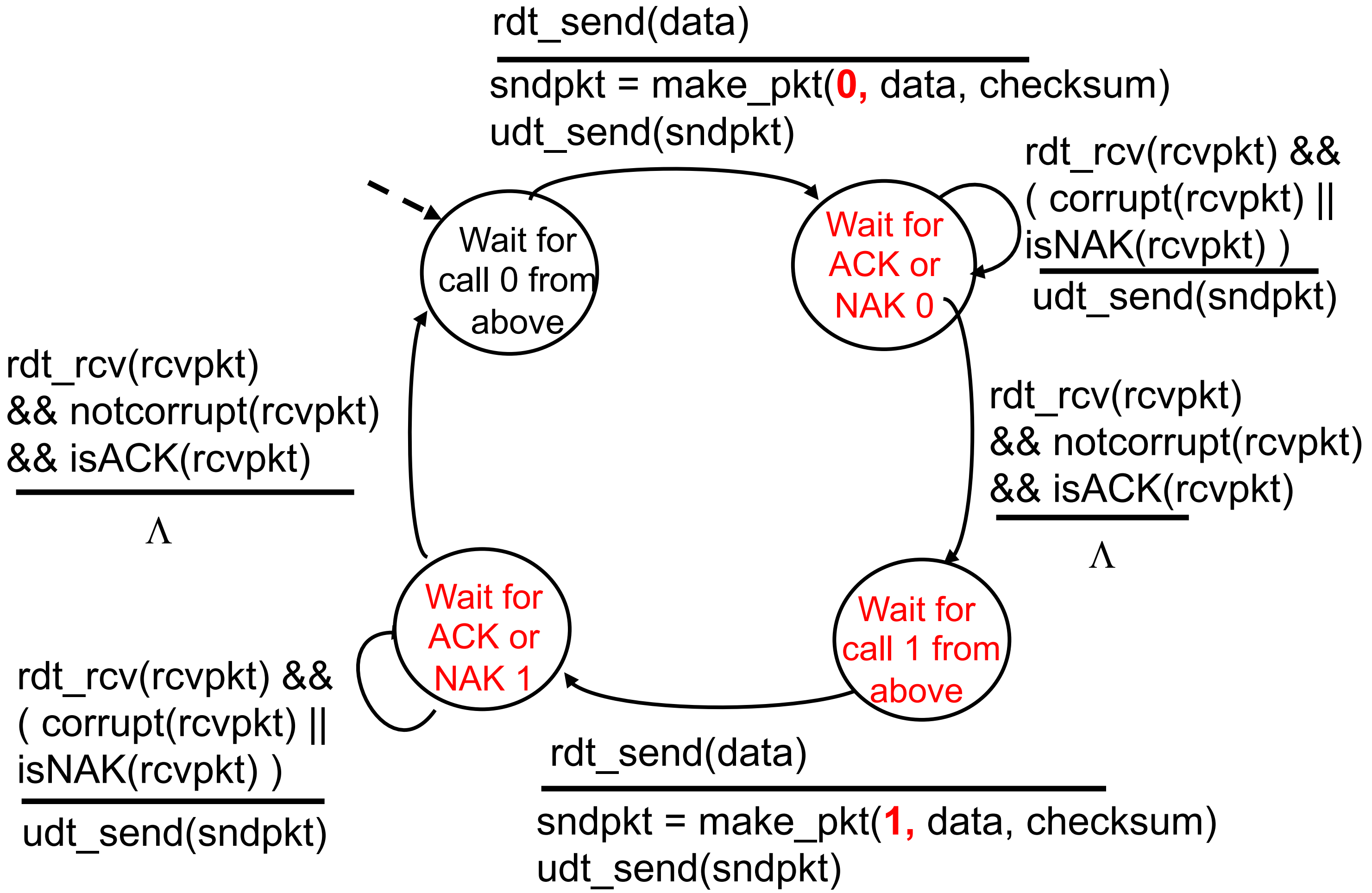
rdt2.1: need seq #!



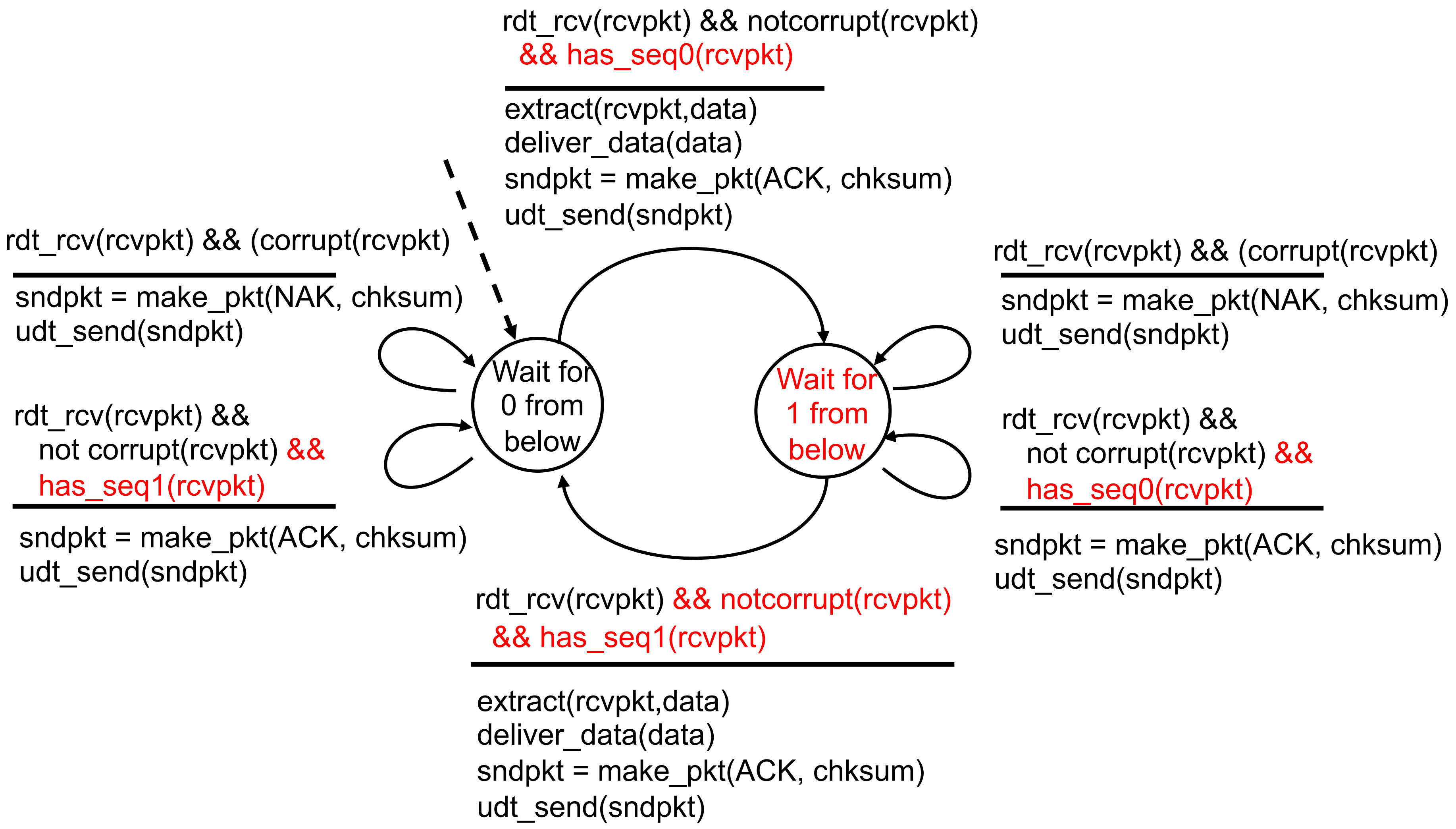
(a) Corrupted ack

(b) Corrupted NACK

rdt2.1: sender, handles garbled ACK/NAKs



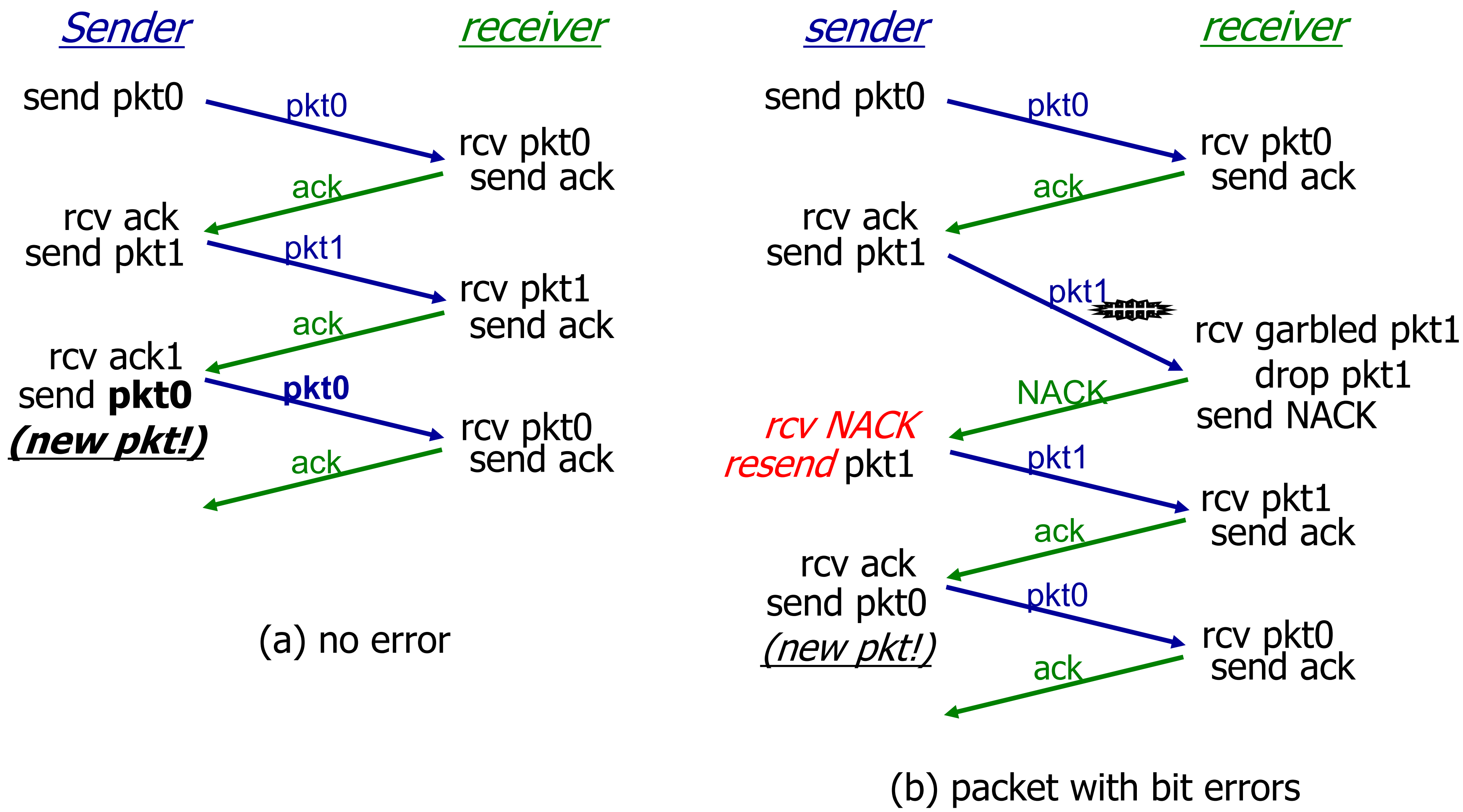
rdt2.1: receiver, handles garbled ACK/NAKs



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 upon NAK
rdt2.1	Same as 2.0	handling fatal flaw with rdt 2.0: (4)need seq #. for each packet

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

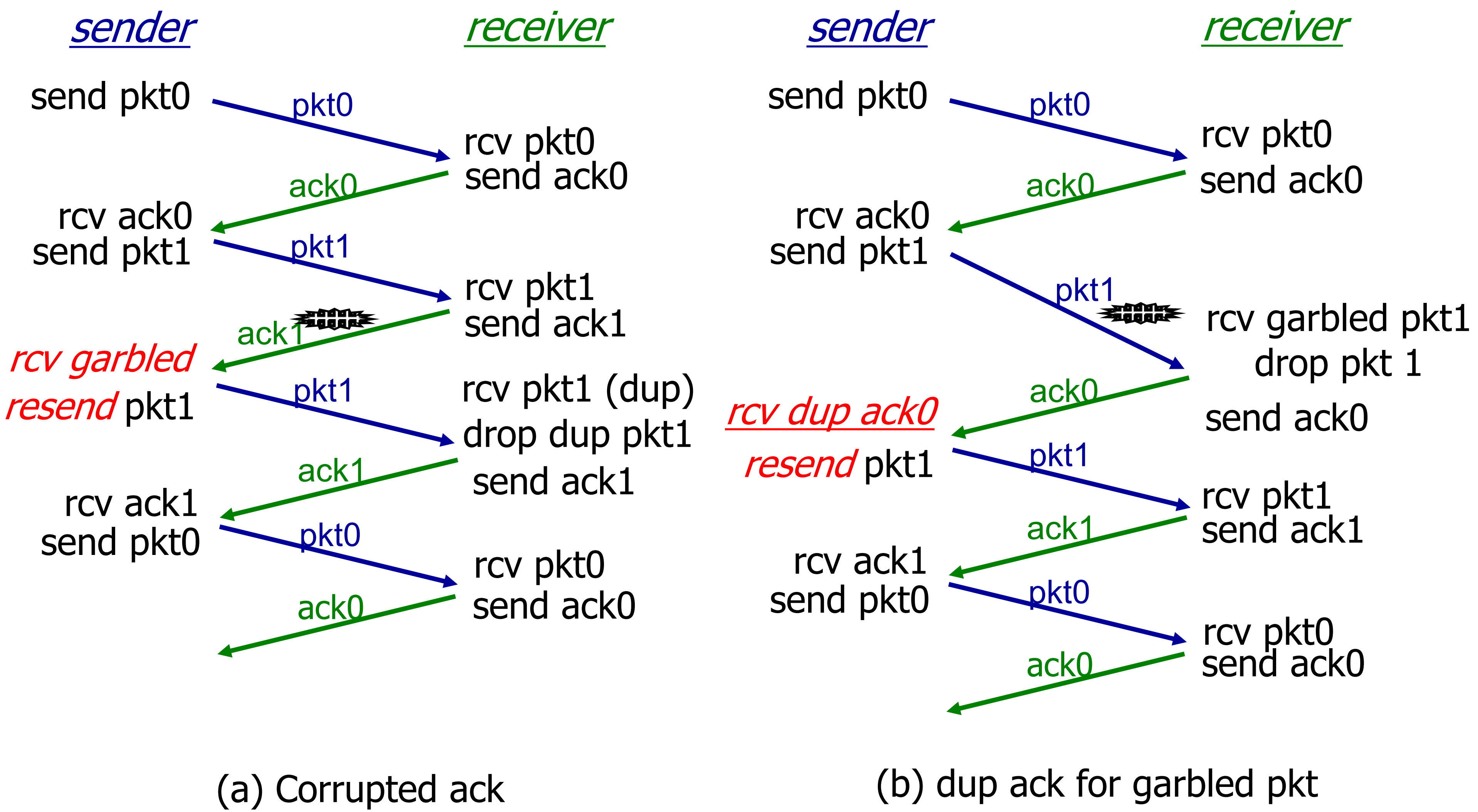


rdt2.2: a NAK-free protocol

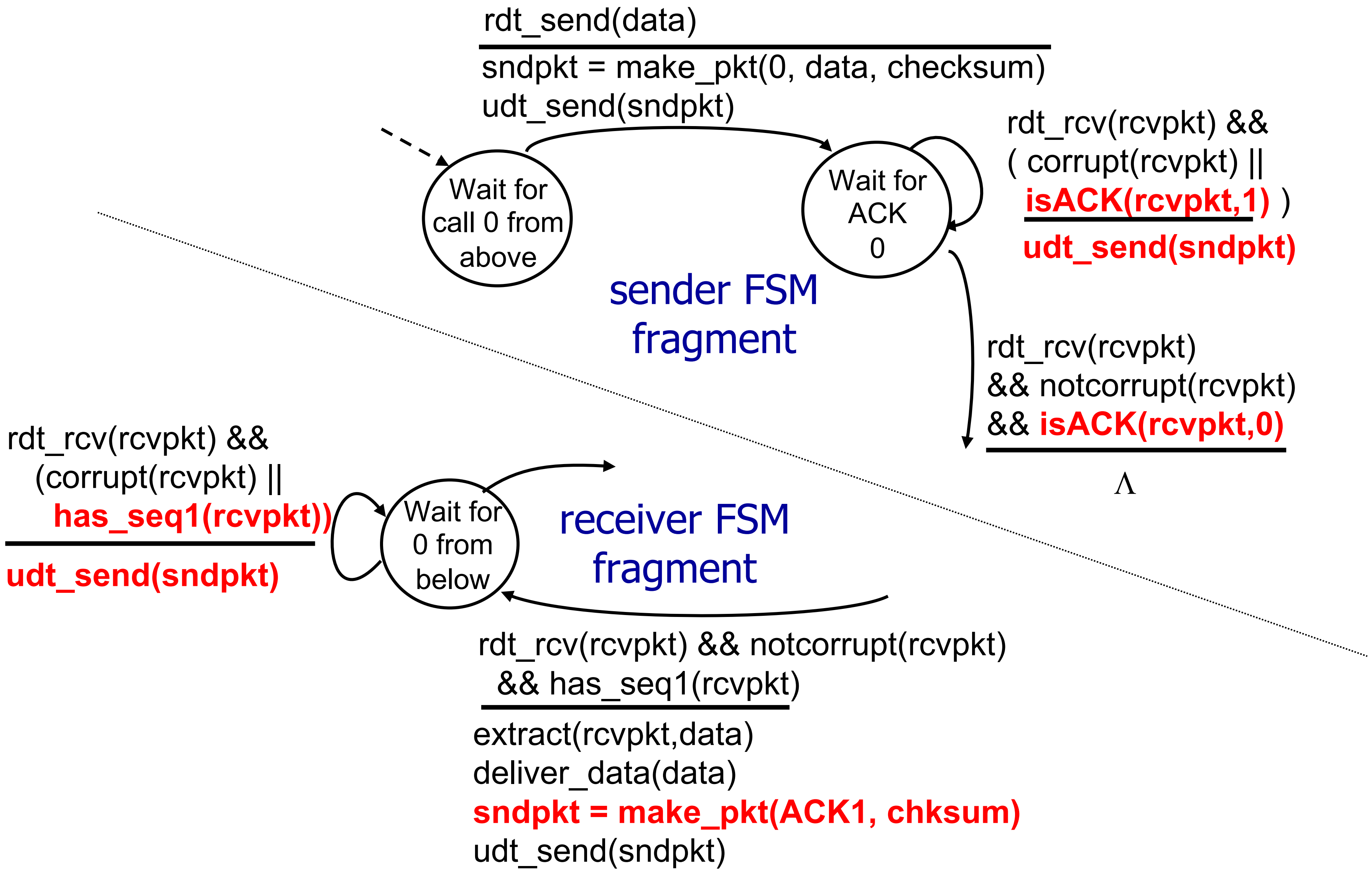
- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

As we will see, TCP uses this approach to be NAK-free

rdt2.2: NAK-free



rdt2.2: sender, receiver fragments



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 upon NAK
rdt2.1	Same as 2.0 (fatal flaw)	(4)seq# (1 bit, 0/1) for each pkt
rdt2.2	Same as 2.0	A variant to rdt2.1 (no NAK) Duplicate ACK = NAK

rdt3.0: channels with errors *and* loss

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

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

Q: How do *humans* handle lost sender-to-receiver words in conversation?

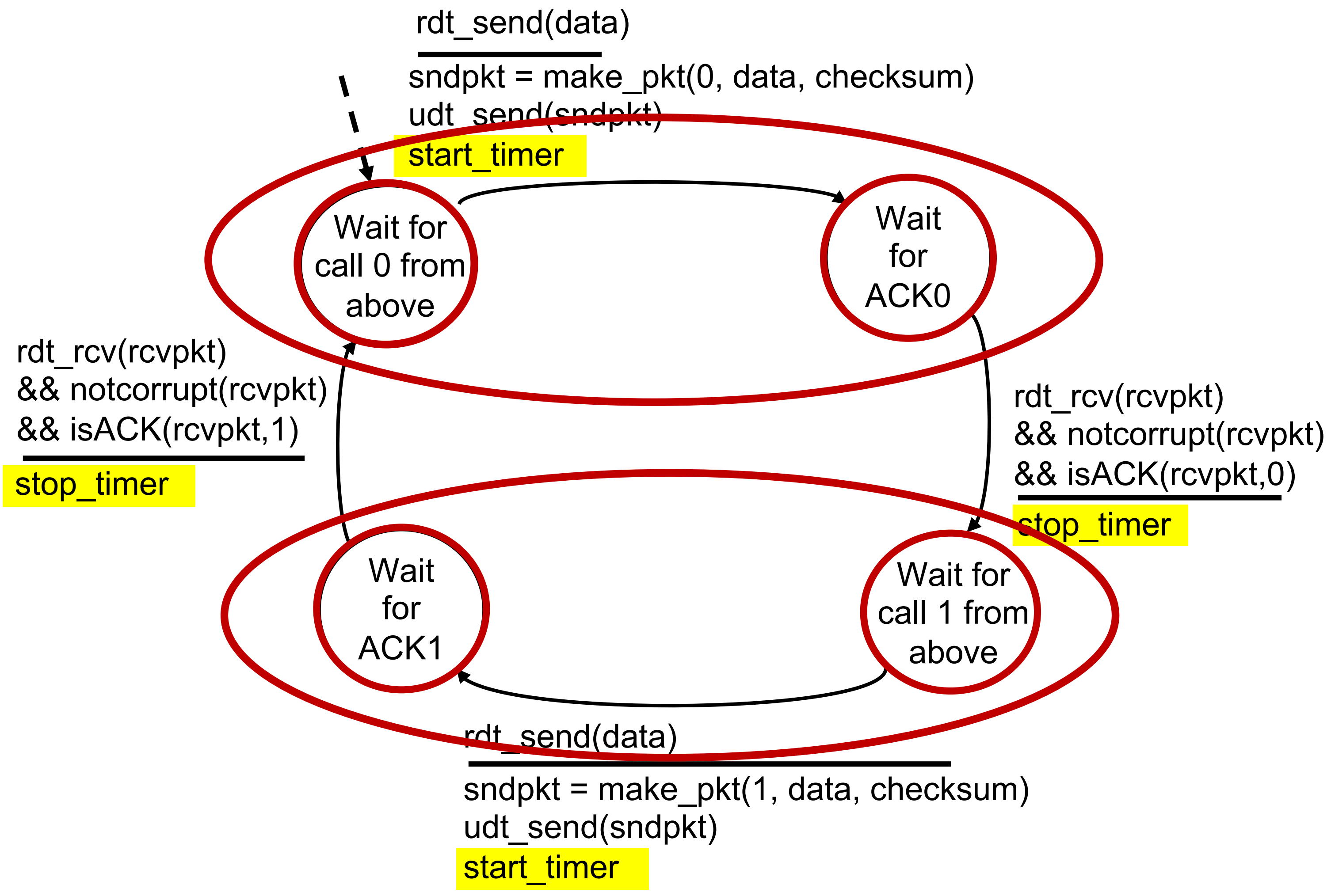
rdt3.0: channels with errors *and* loss

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

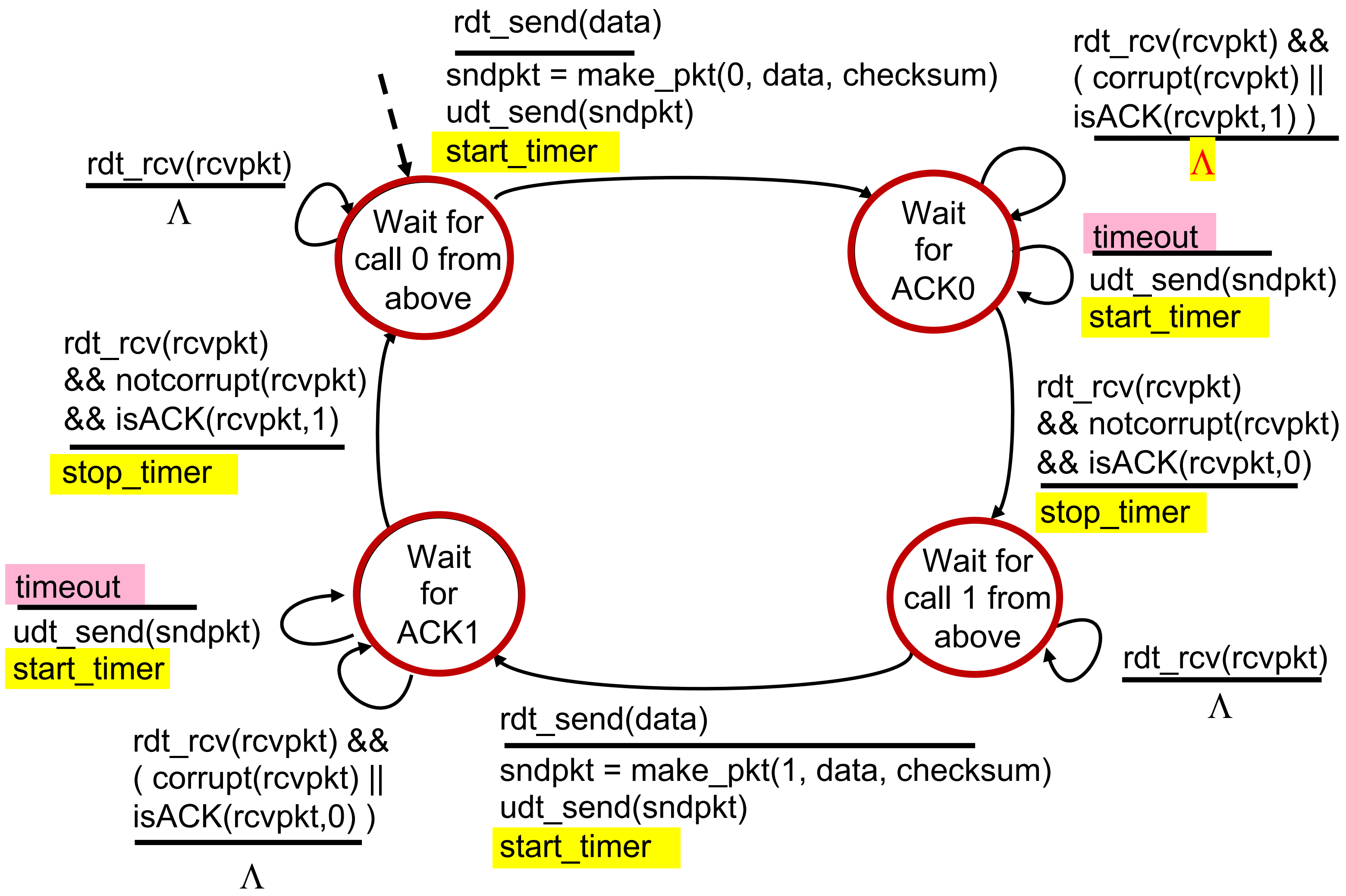
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

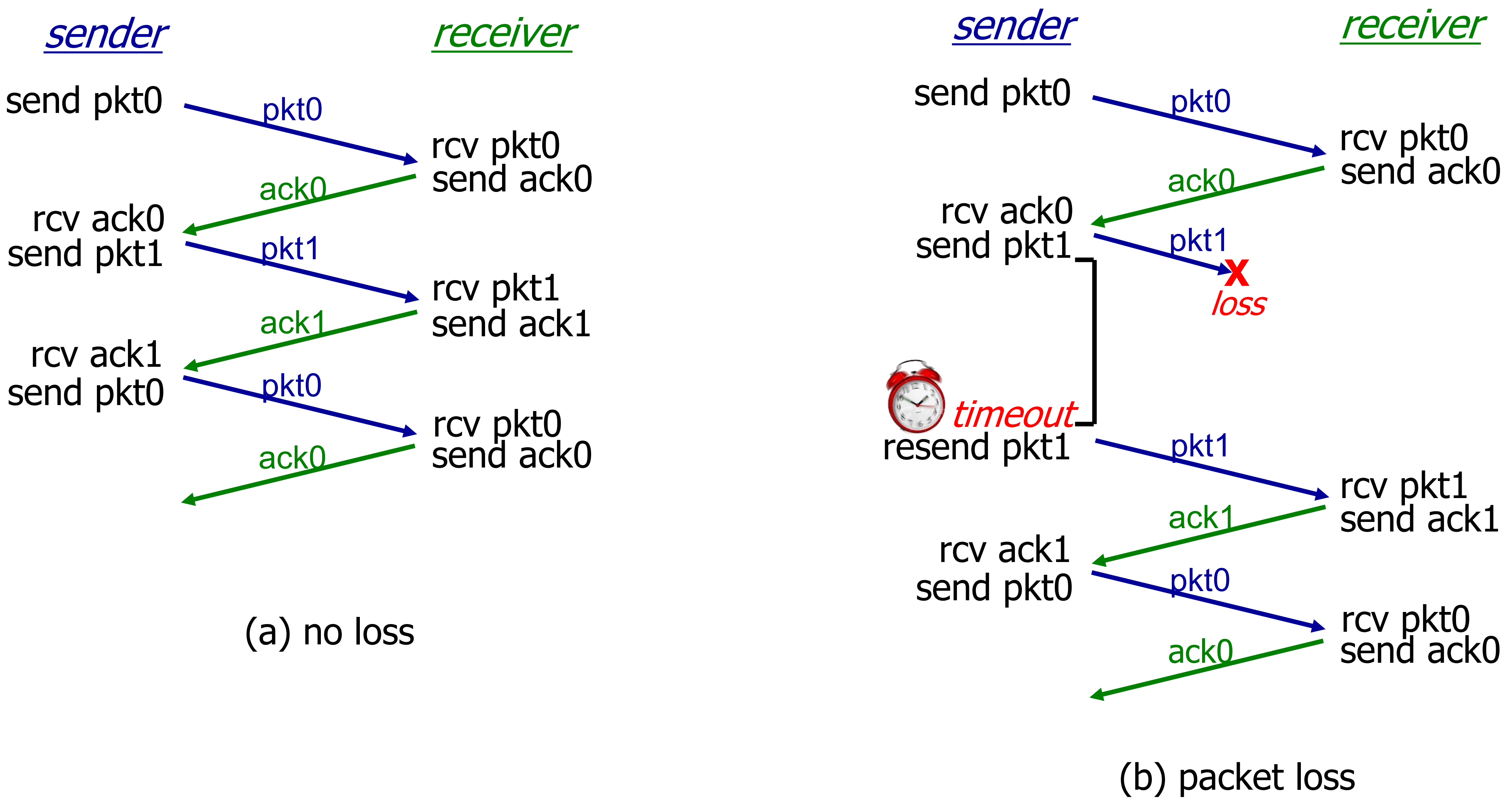
rdt3.0 sender



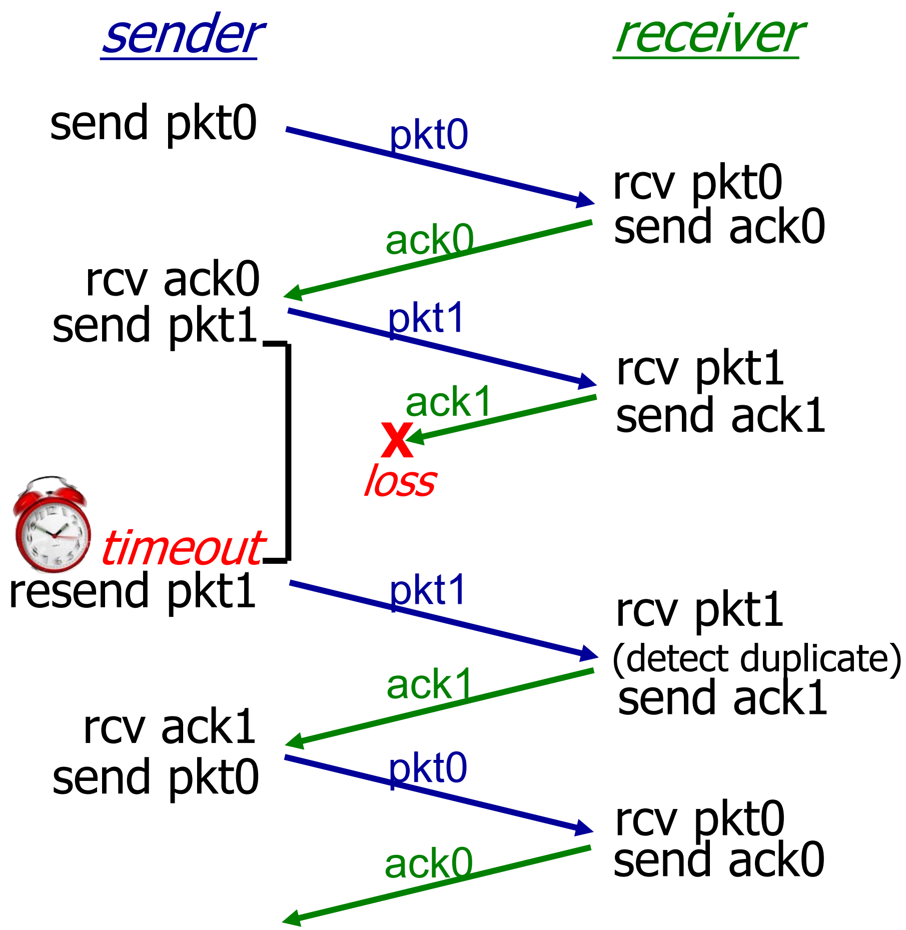
rdt3.0 sender



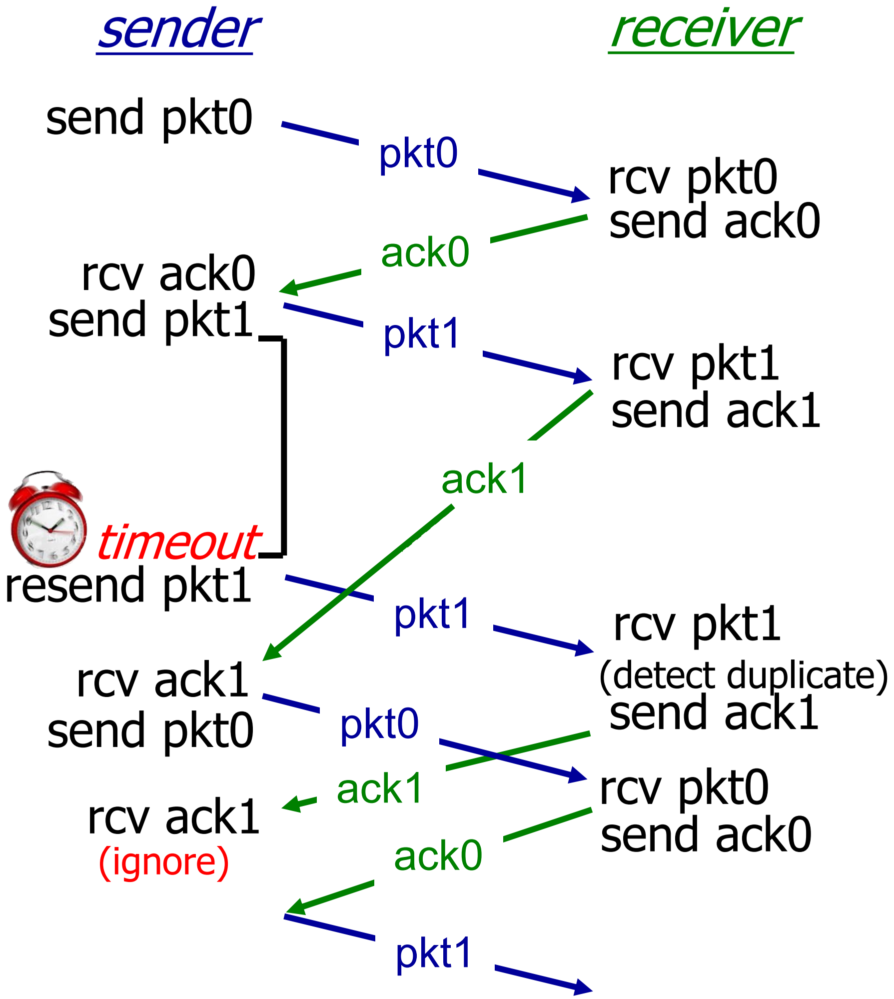
Example: rdt3.0 in action



rdt3.0 in action



(c) ACK loss



(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 upon NAK
rdt2.1	Same as 2.0	(4)seq# (1 bit) for each pkt
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 upon timeout No NAK, only ACK

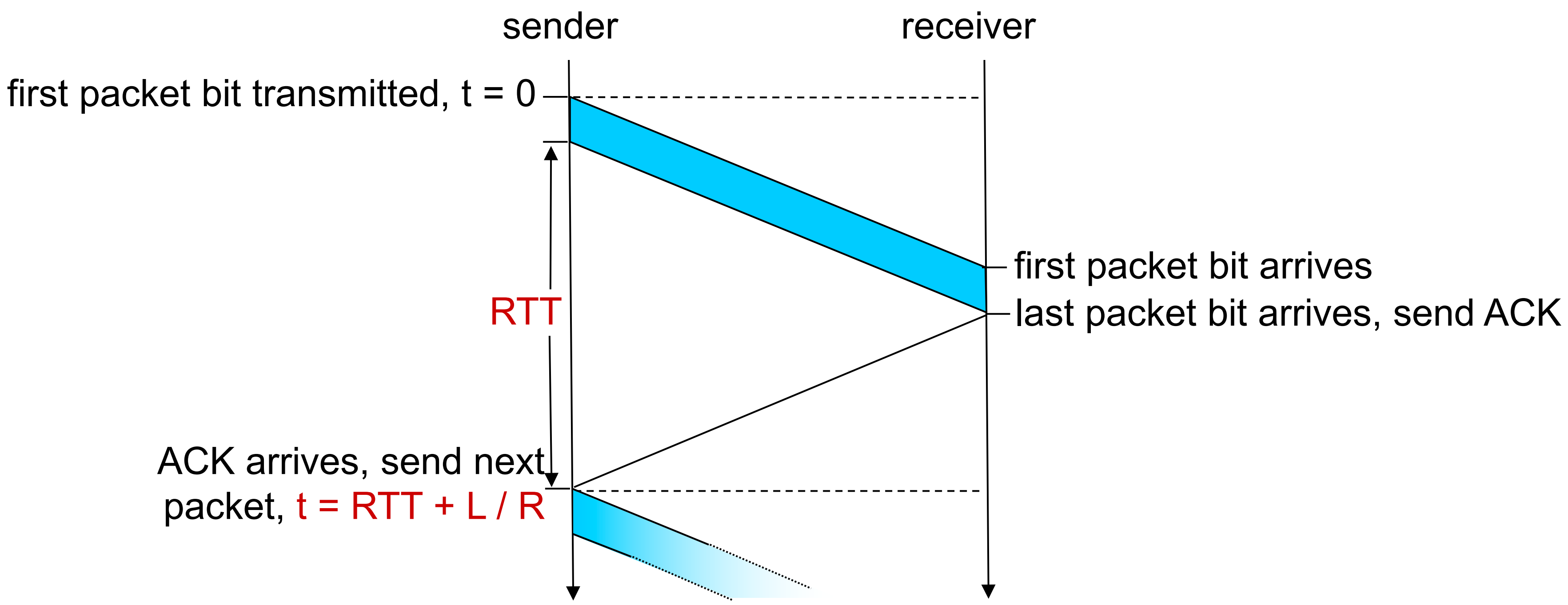
Performance of rdt3.0 (stop-and-wait)

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

- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

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

rdt3.0: stop-and-wait operation

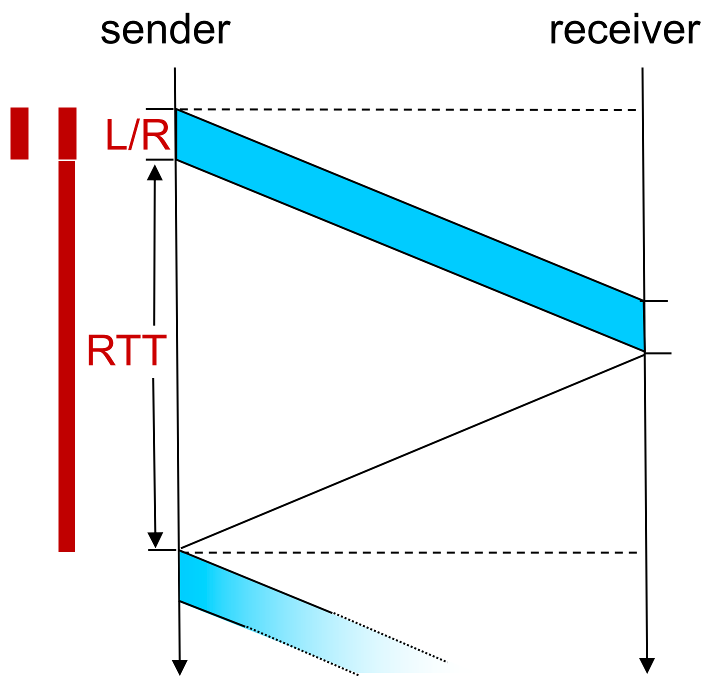


rdt3.0: stop-and-wait operation

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

$$= \frac{.008}{30.008}$$

$$= 0.00027$$



- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

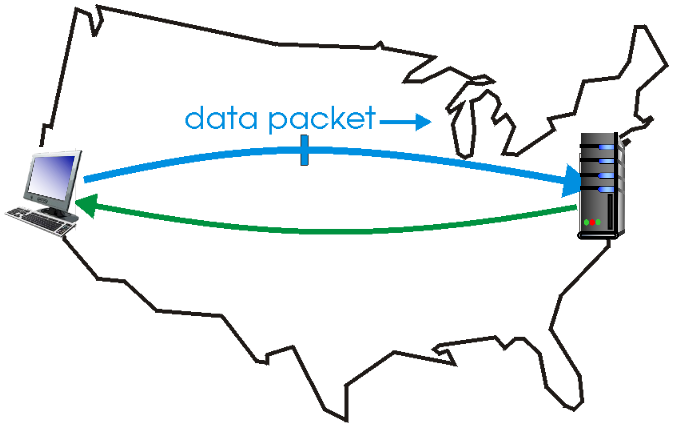
Mechanisms for reliable data transfer

- Error detection
  - via algorithms such as Internet checksum (in UDP), CRC (later in Chapter 6)
- Receiver feedback via (ACK + sequence #)
  - Duplicate ACK = negative acknowledgment
- Timer & sequence # for each transmitted packet
  - Number of seq. #: ≥ 2 for stop & wait protocol
  - Timeout not too small, not too big ($\approx RTT$)
- Retransmission upon timeout or duplicate ACK (i.e., negative ACK)

rdt3.0: pipelined protocols operation

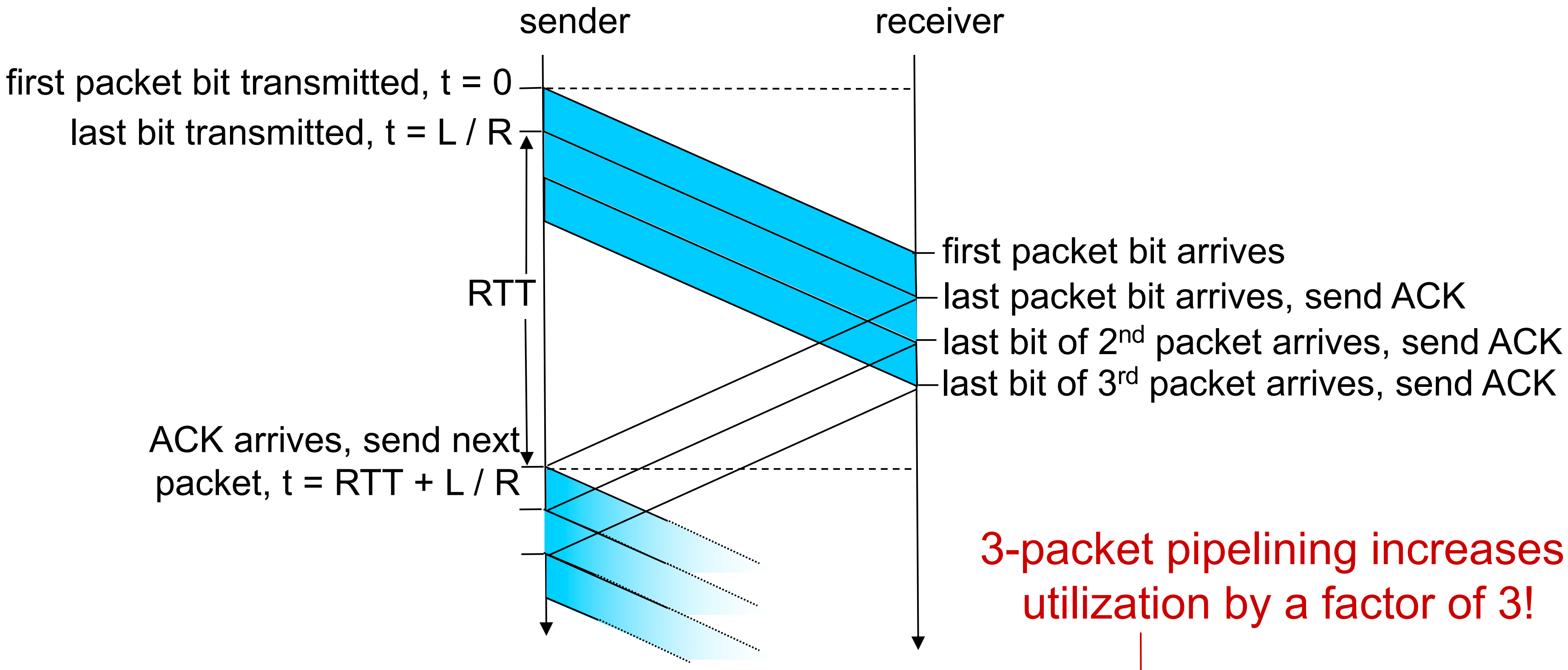
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

Pipelining: increased utilization

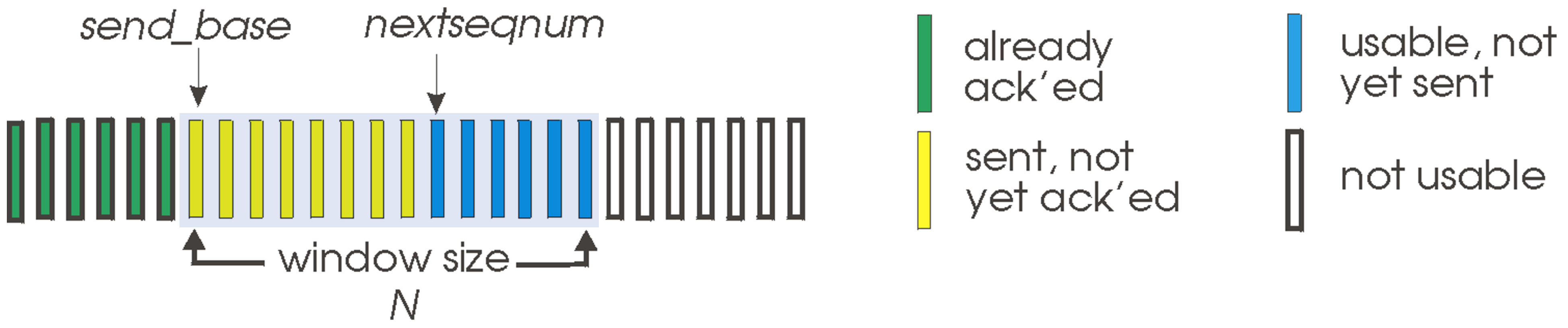


3-packet pipelining increases utilization by a factor of 3!

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

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

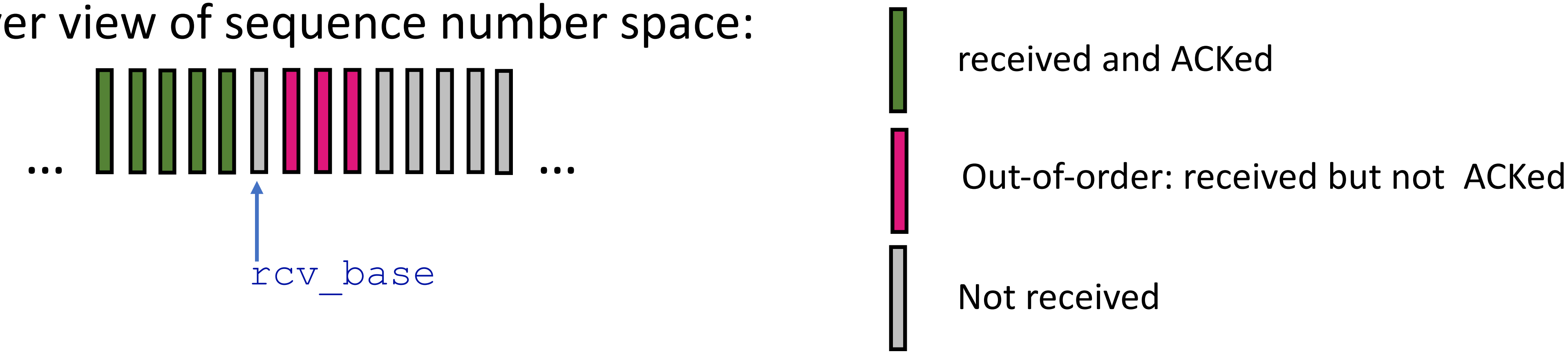


- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

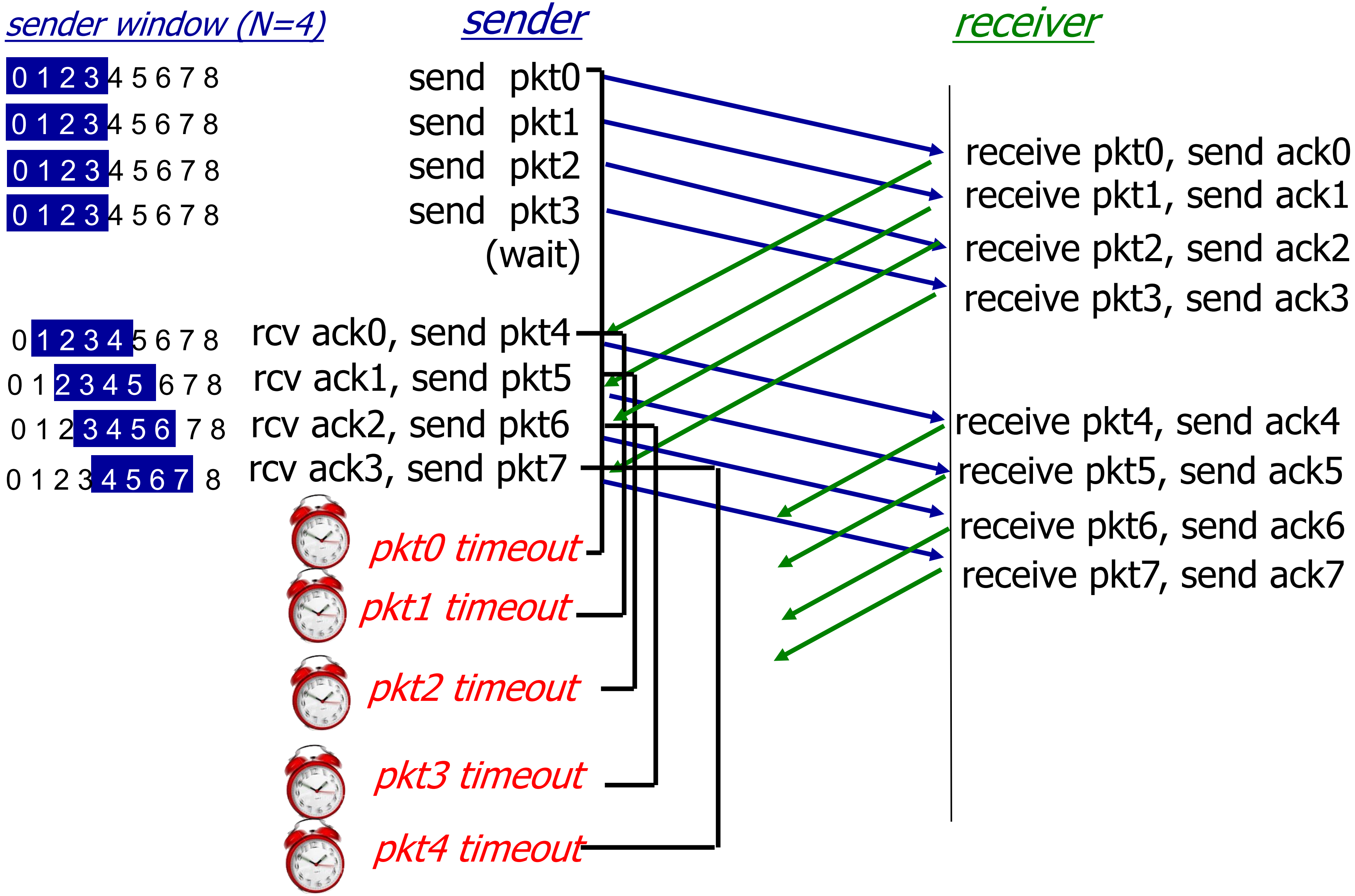
Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

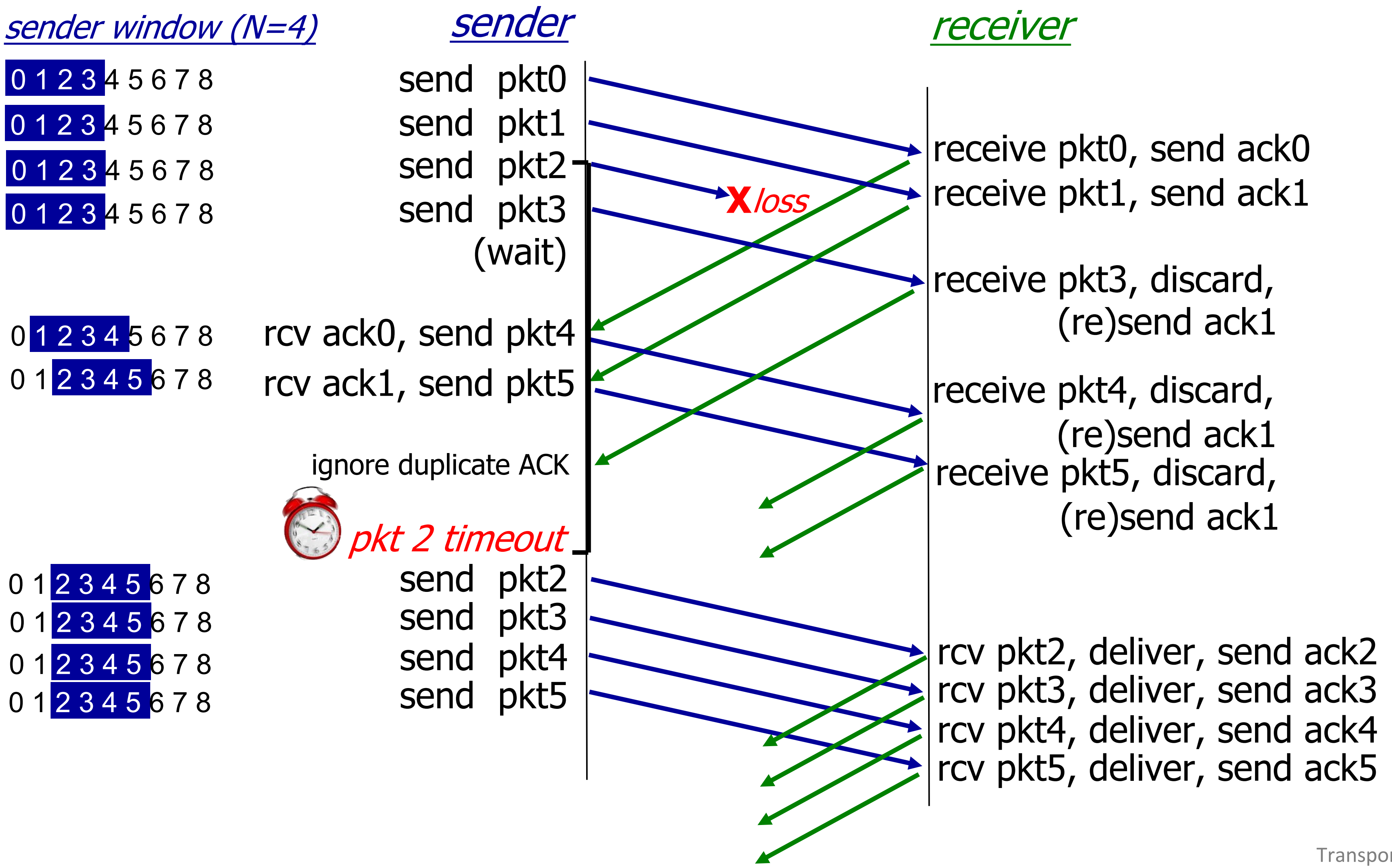
Receiver view of sequence number space:



Go-Back-N in action: No loss



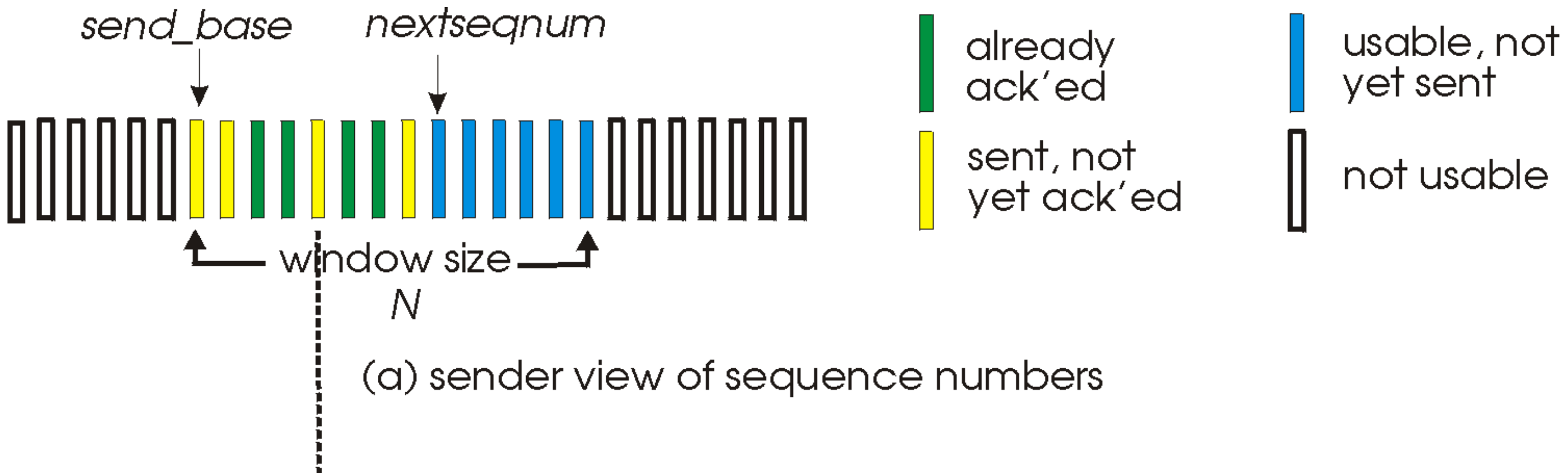
Go-Back-N in action: Loss



Selective repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N]:

- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

packet *n* in [rcvbase, rcvbase+N-1]

- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

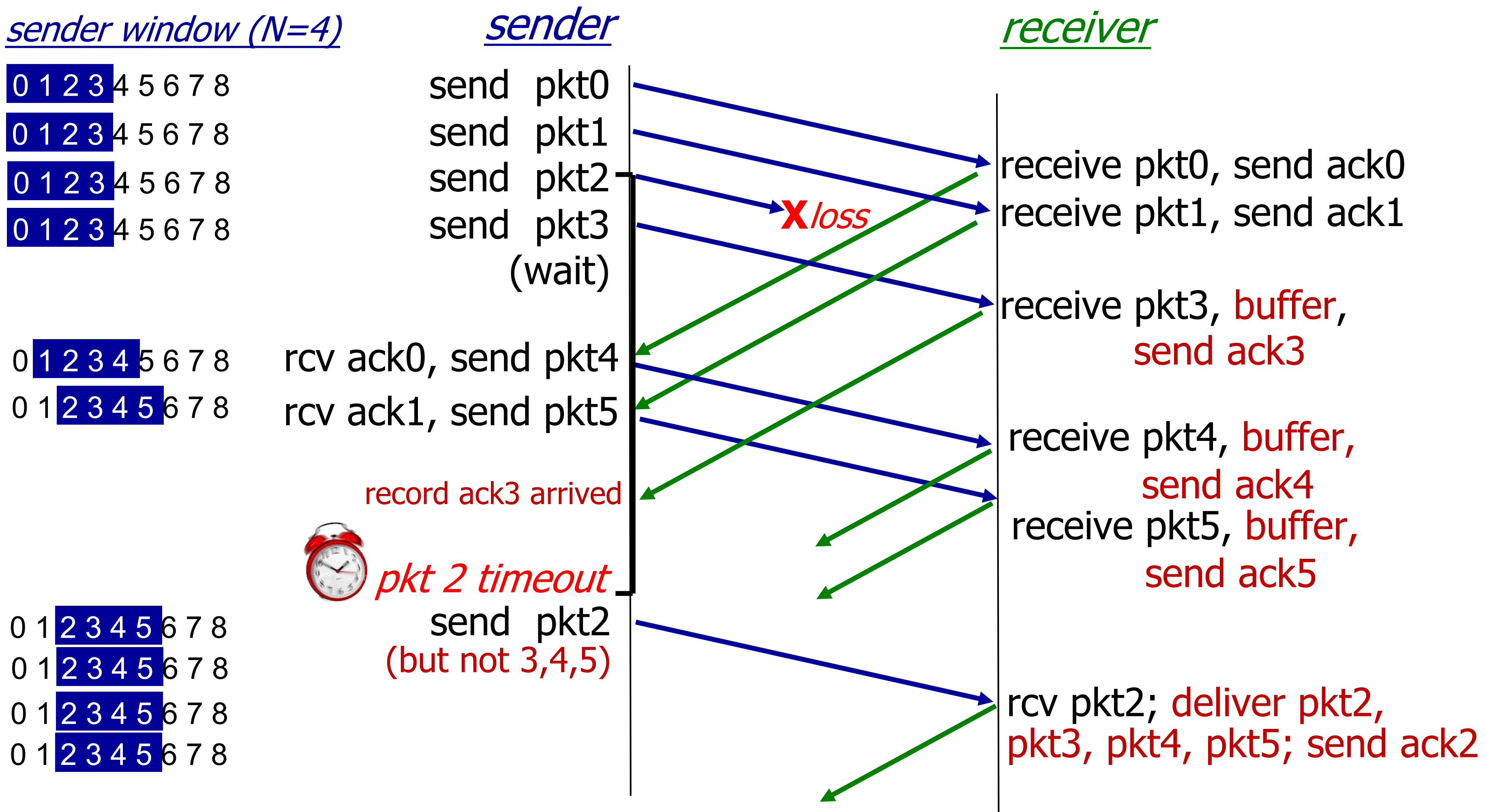
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action



Q: what happens when ack2 arrives?

In-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, or 6**
    - No error
    - ACK 4 is lost
    - ACK 4 and ACK 5 are lost

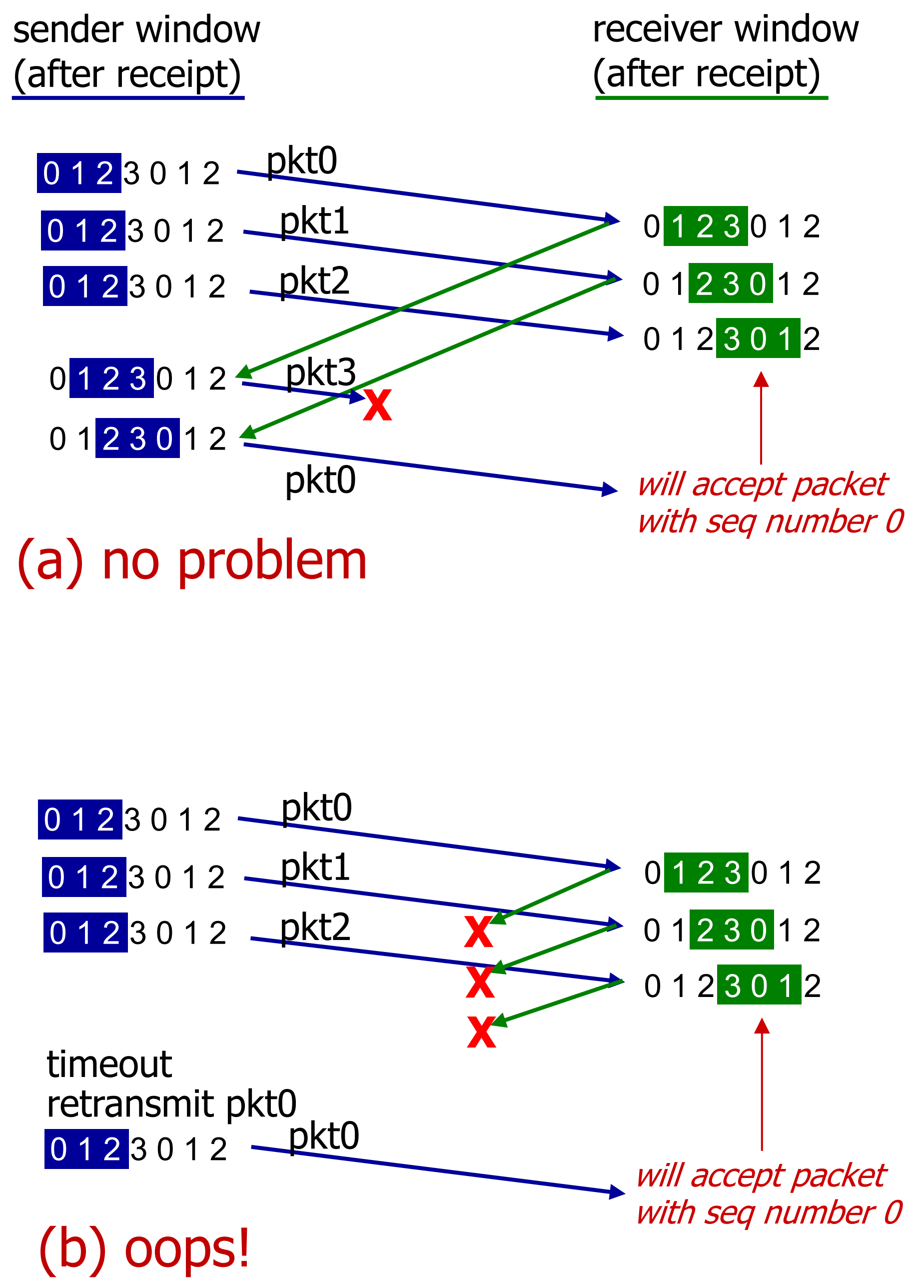
GBN: give the expected number x, the sender window will be [x-2, x-1], [x-1, x], [x, x+1]

  - 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: a dilemma!

example:

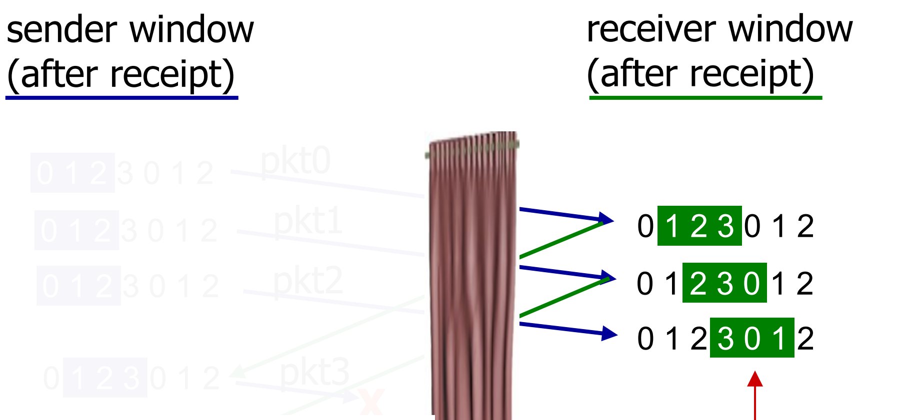
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



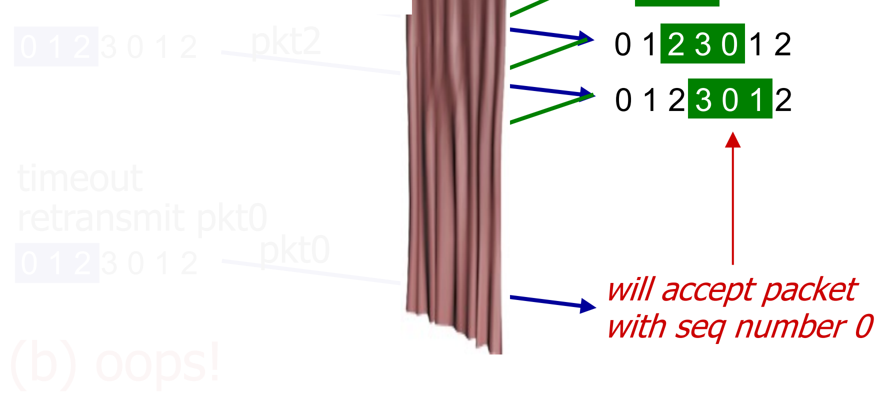
Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



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



Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

Selective repeat: dilemma (N+1)

example:

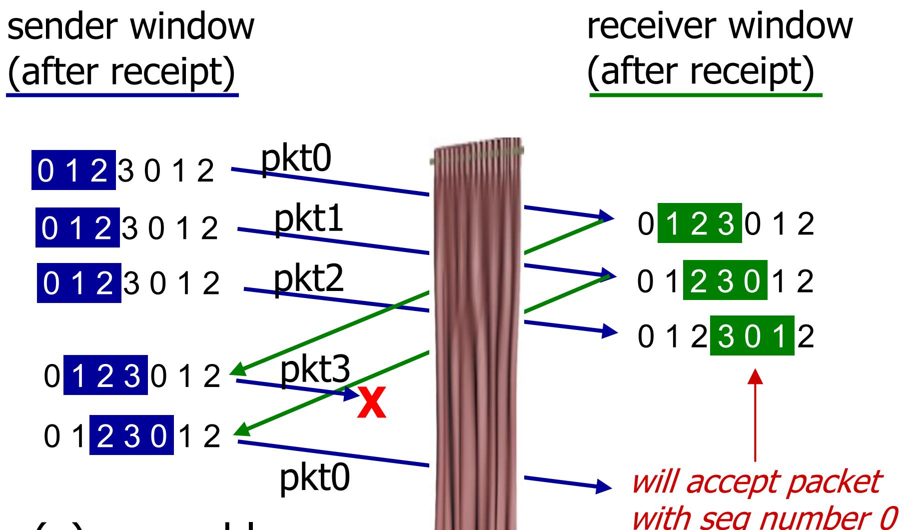
- window size=3
- seq #' s: 0, 1, 2, 3

❖ receiver sees no difference in two scenarios!

❖ duplicate data accepted as new in (b)

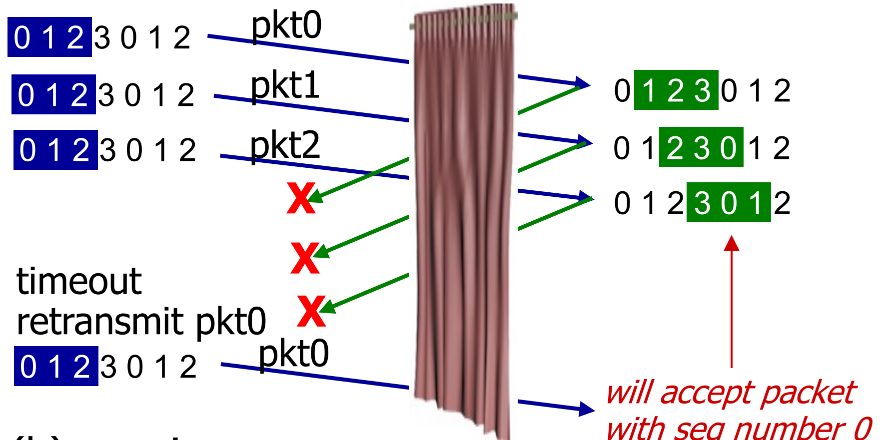
Q: what relationship between seq # size and window size to avoid problem in (b)?

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 upon NAK
rdt2.1	Same as 2.0	(4)seq# (1 bit) for each pkt
rdt2.2	Same as 2.0	(no NAK): Unexpected ACK = NAK
Rdt3.0	errors + loss	(5)Retransmission upon timeout; ACK-only

Performance issue: low utilization

Goback-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: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

ACK: seq # of next expected byte; A bit: this is an ACK

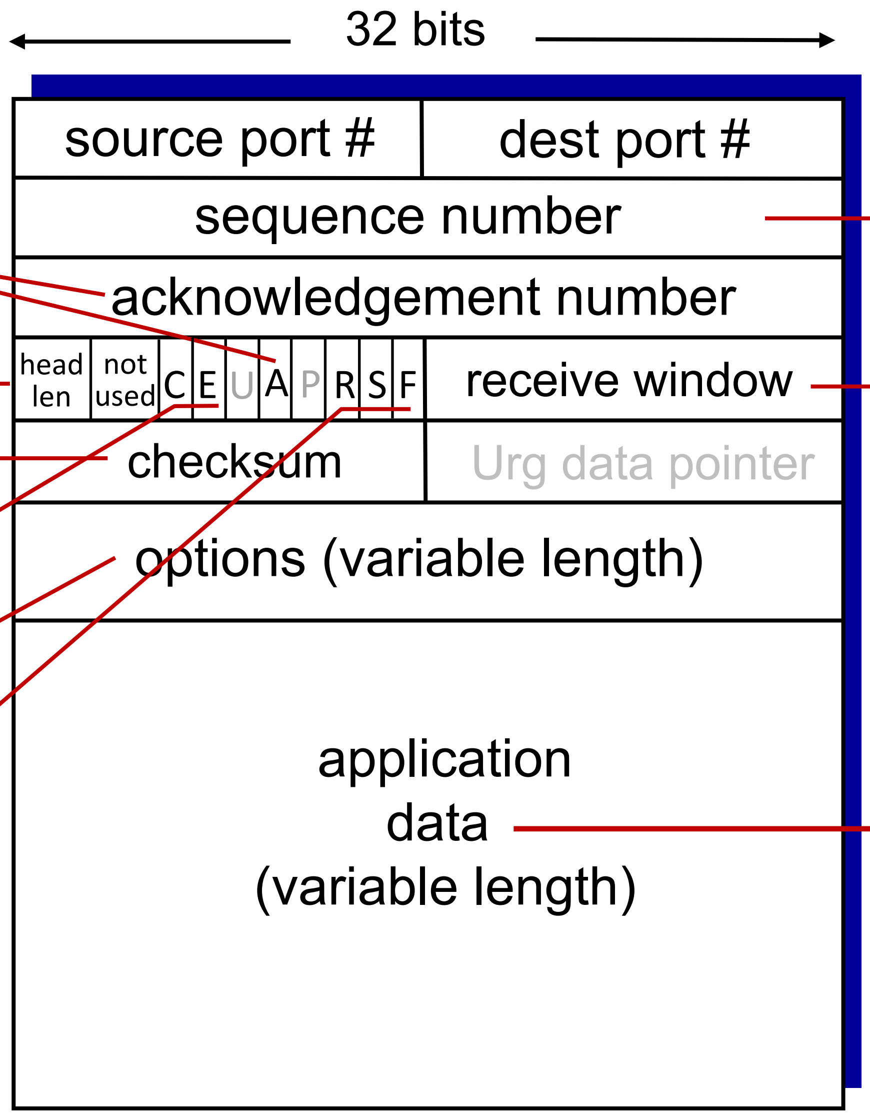
length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management



segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

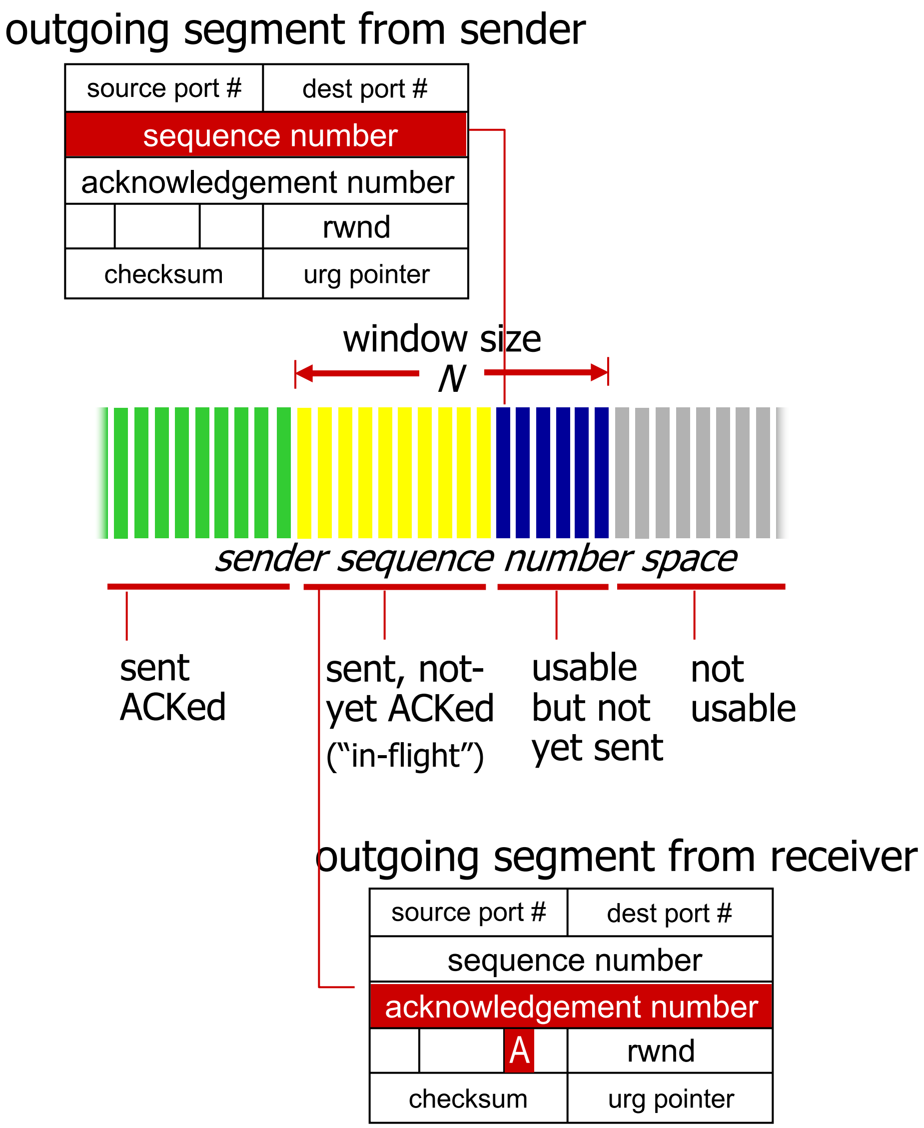
- byte stream "number" of first byte in segment's data

Acknowledgements:

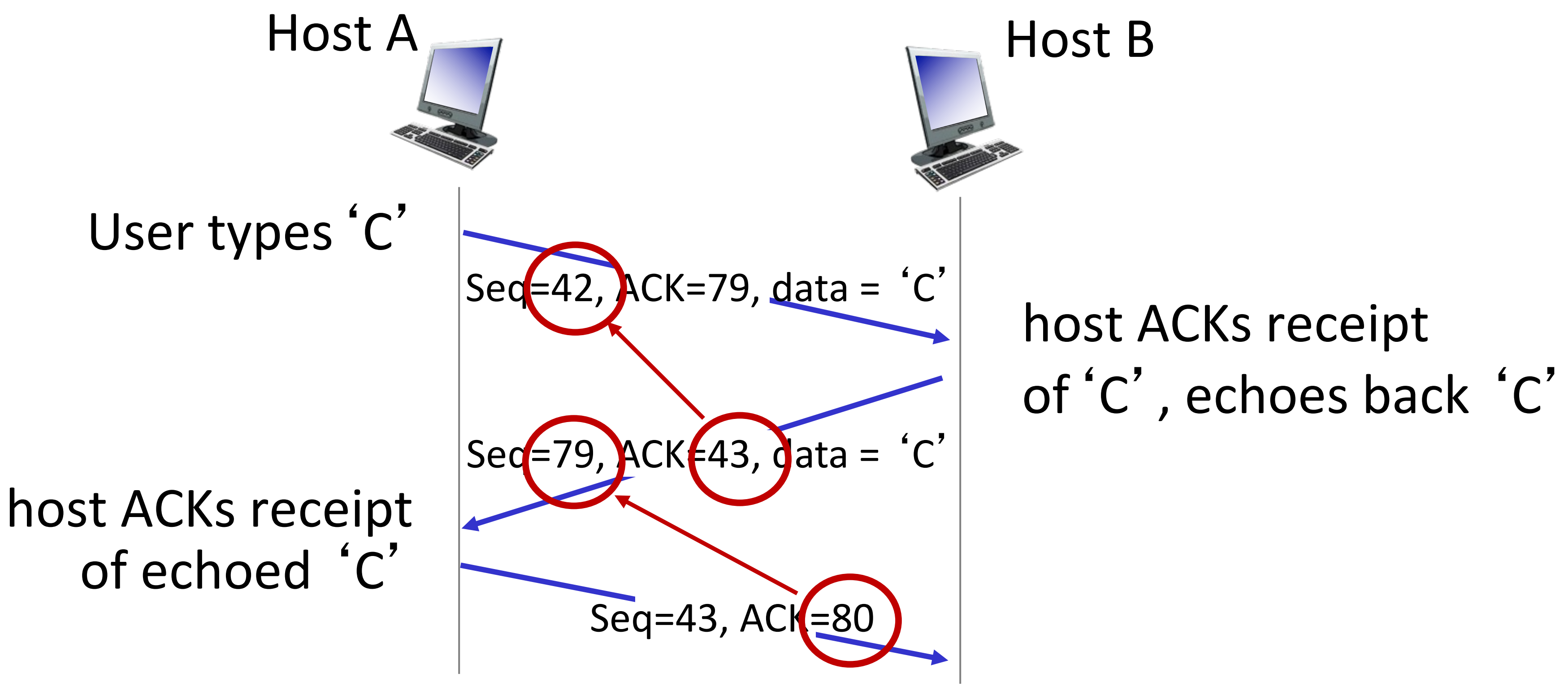
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP sequence numbers, ACKs



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

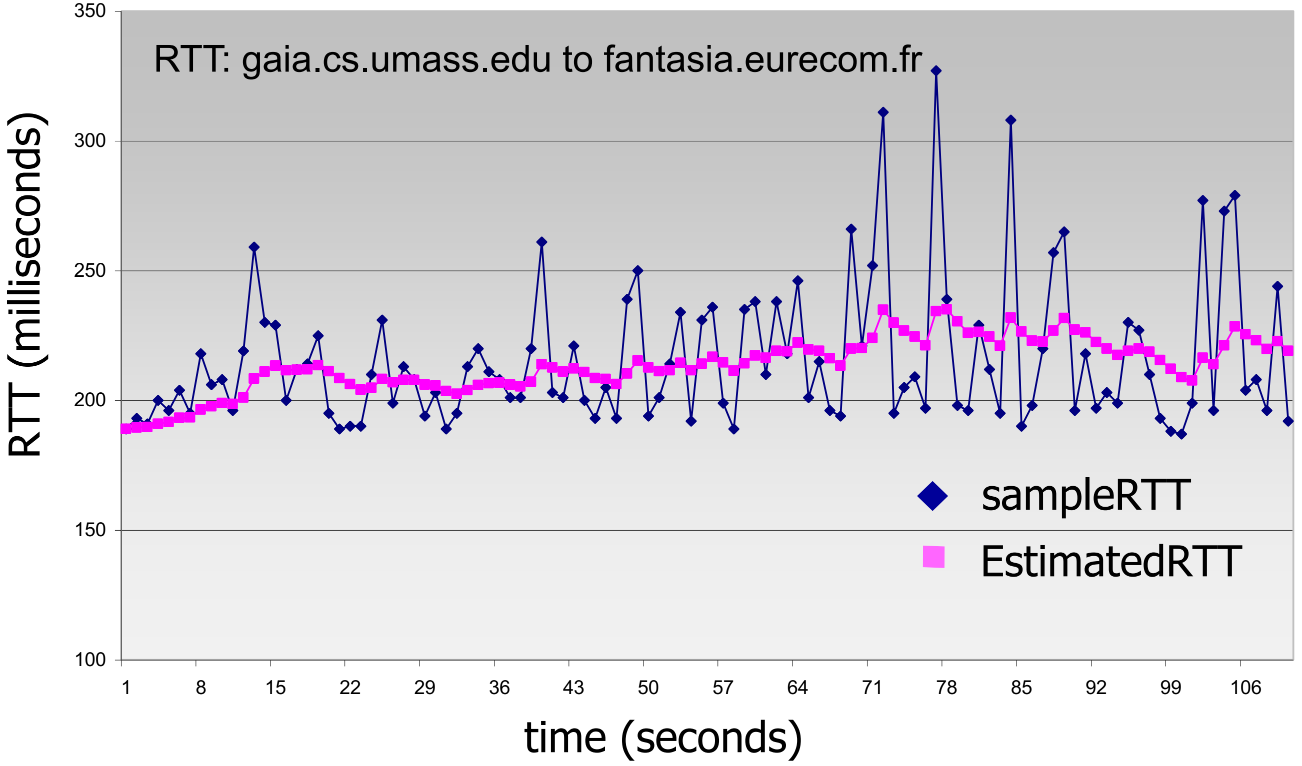
Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions (why?)
- `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 = 1/8



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT:`** want a larger safety margin

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

estimated RTT

"safety margin"

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

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

(typically, β = 1/4)

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

TCP Sender (simplified)

event: data received from application

- 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`**

event: timeout

- retransmit segment that caused timeout
- restart timer

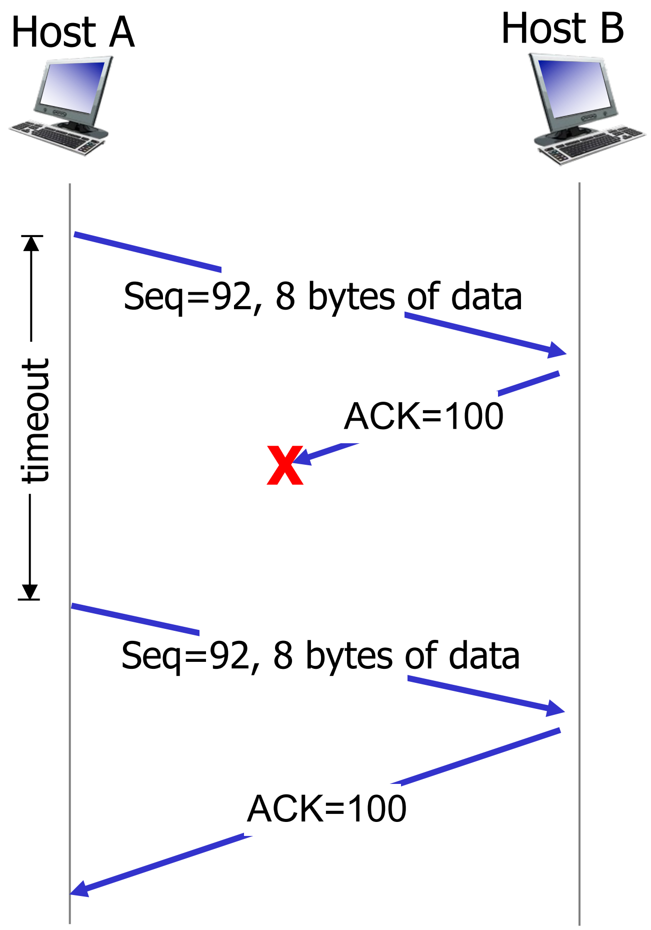
event: ACK received

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

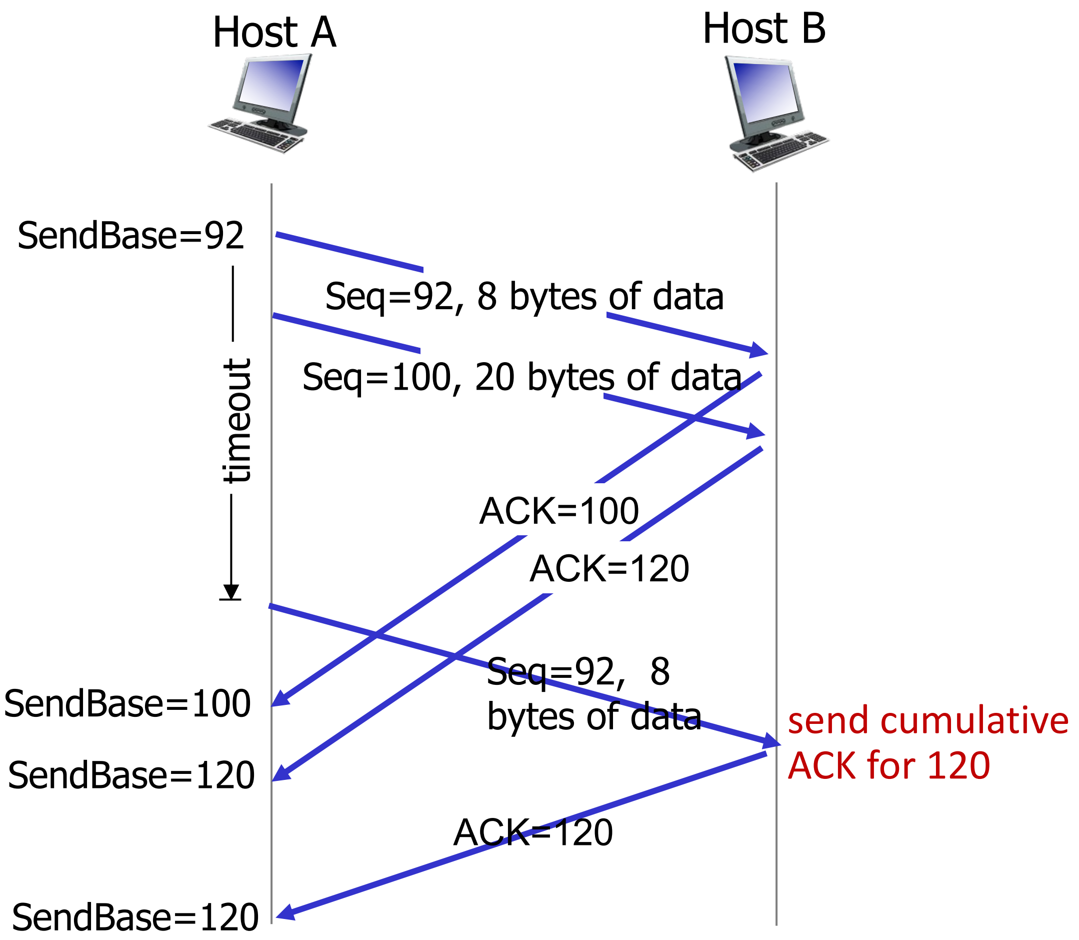
TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*

TCP: retransmission scenarios

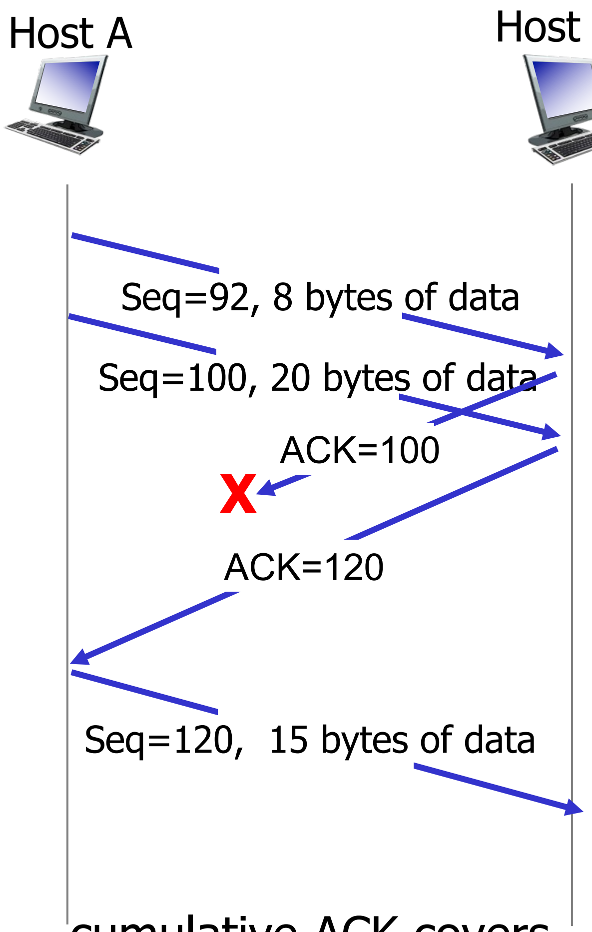


lost ACK scenario



premature timeout

TCP: retransmission scenarios



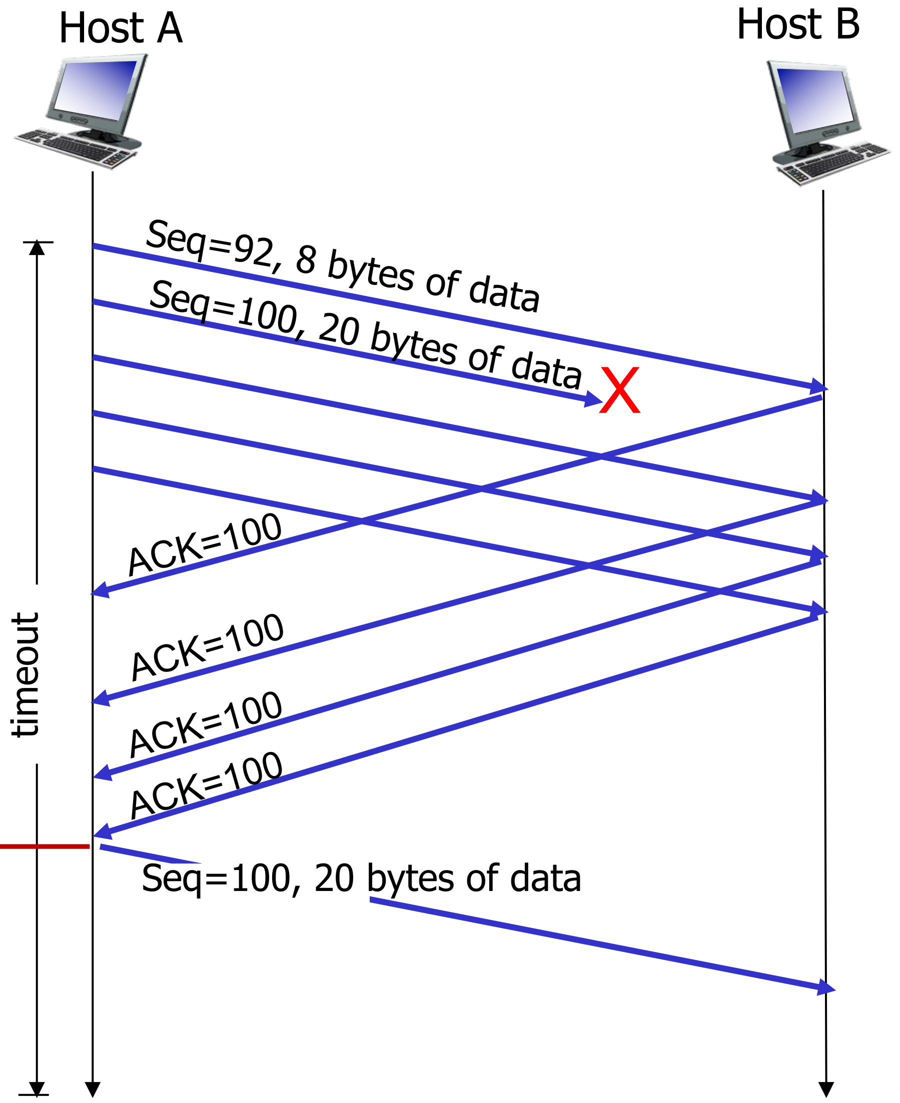
cumulative ACK covers for earlier lost ACK

TCP fast retransmit

TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout

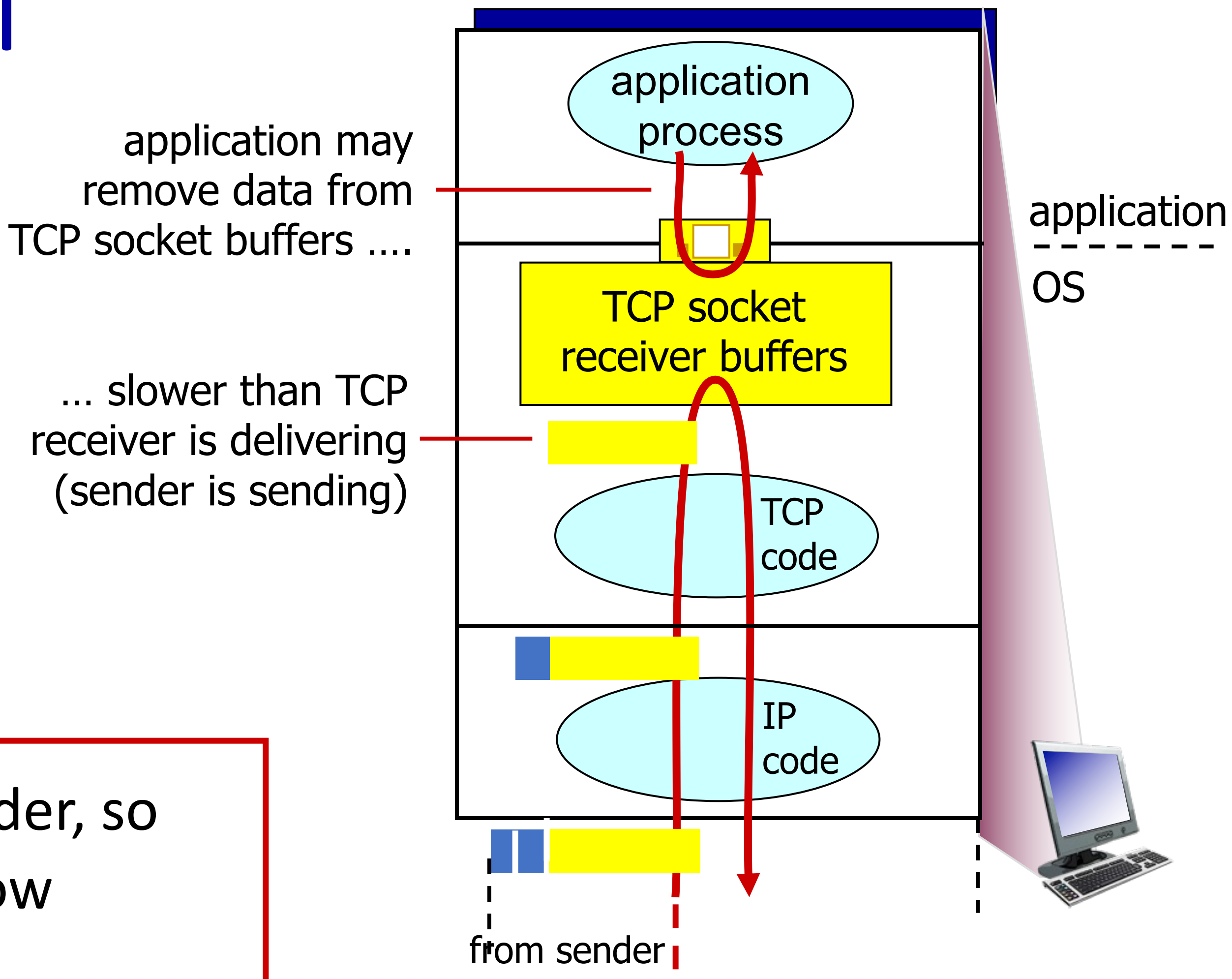


Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP flow control



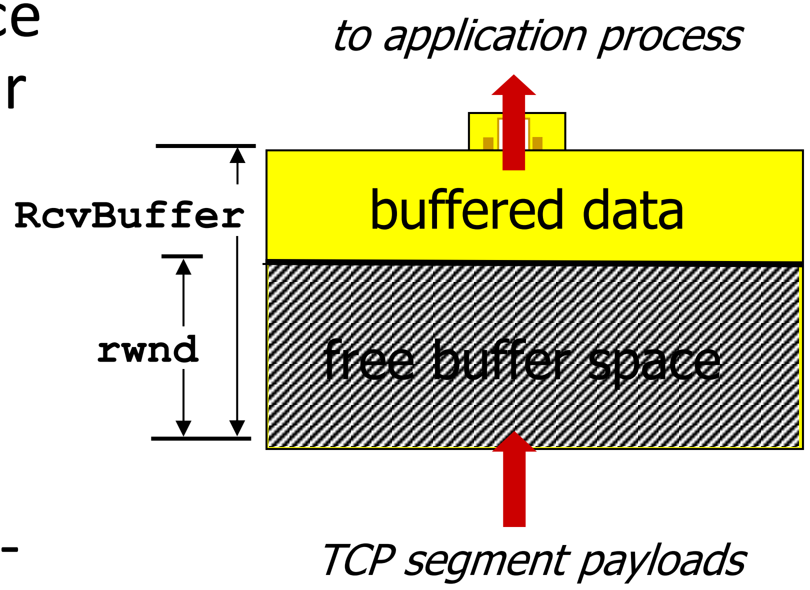
receiver protocol stack

flow control

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

TCP flow control

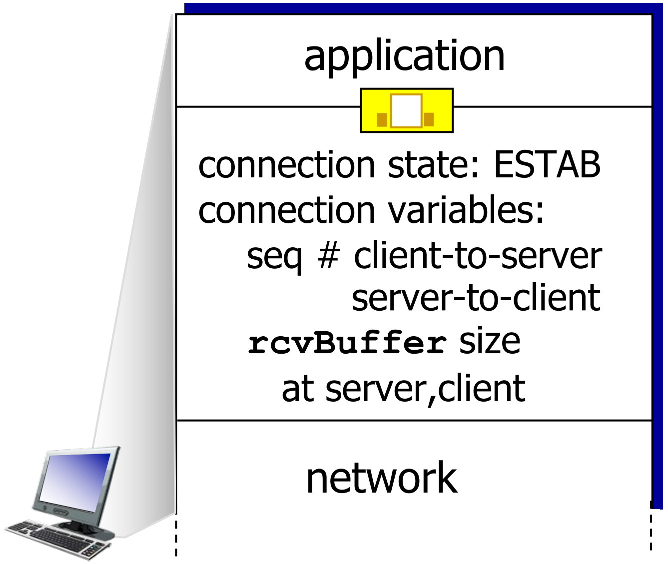
- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to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



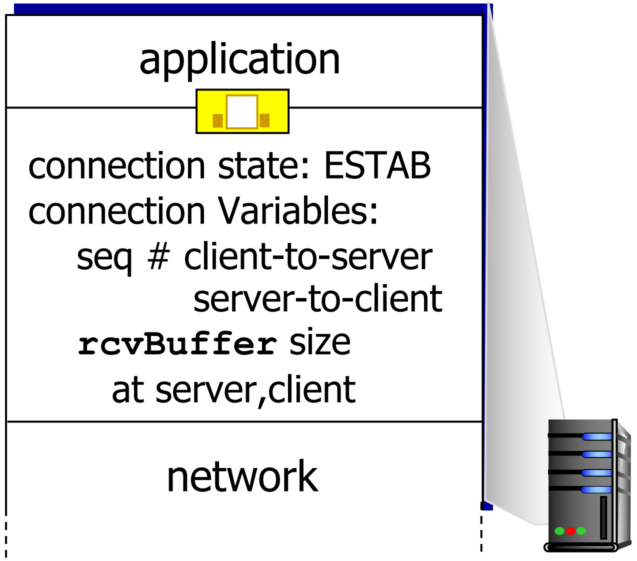
receiver-side buffering

TCP connection management

before exchanging data, sender/receiver “handshake”:
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)



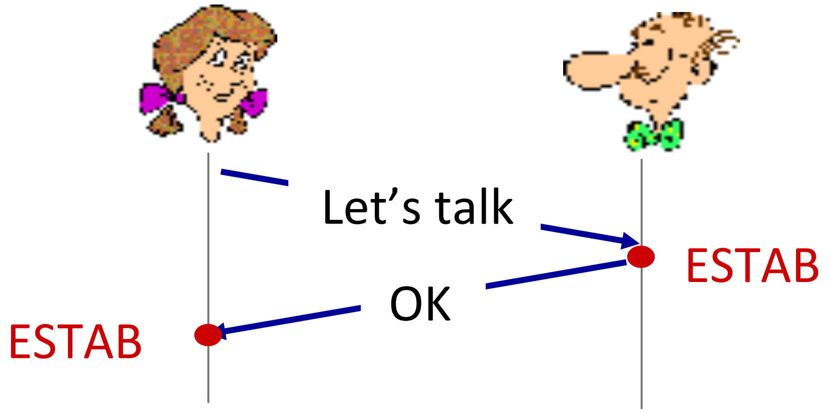

```
Socket clientSocket =
  newSocket("hostname","port number");
```

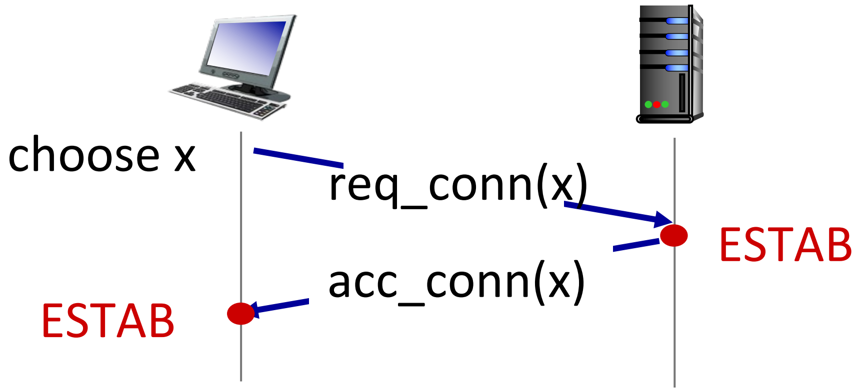



```
Socket connectionSocket =
  welcomeSocket.accept();
```

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

TCP 3-way handshake

Server state

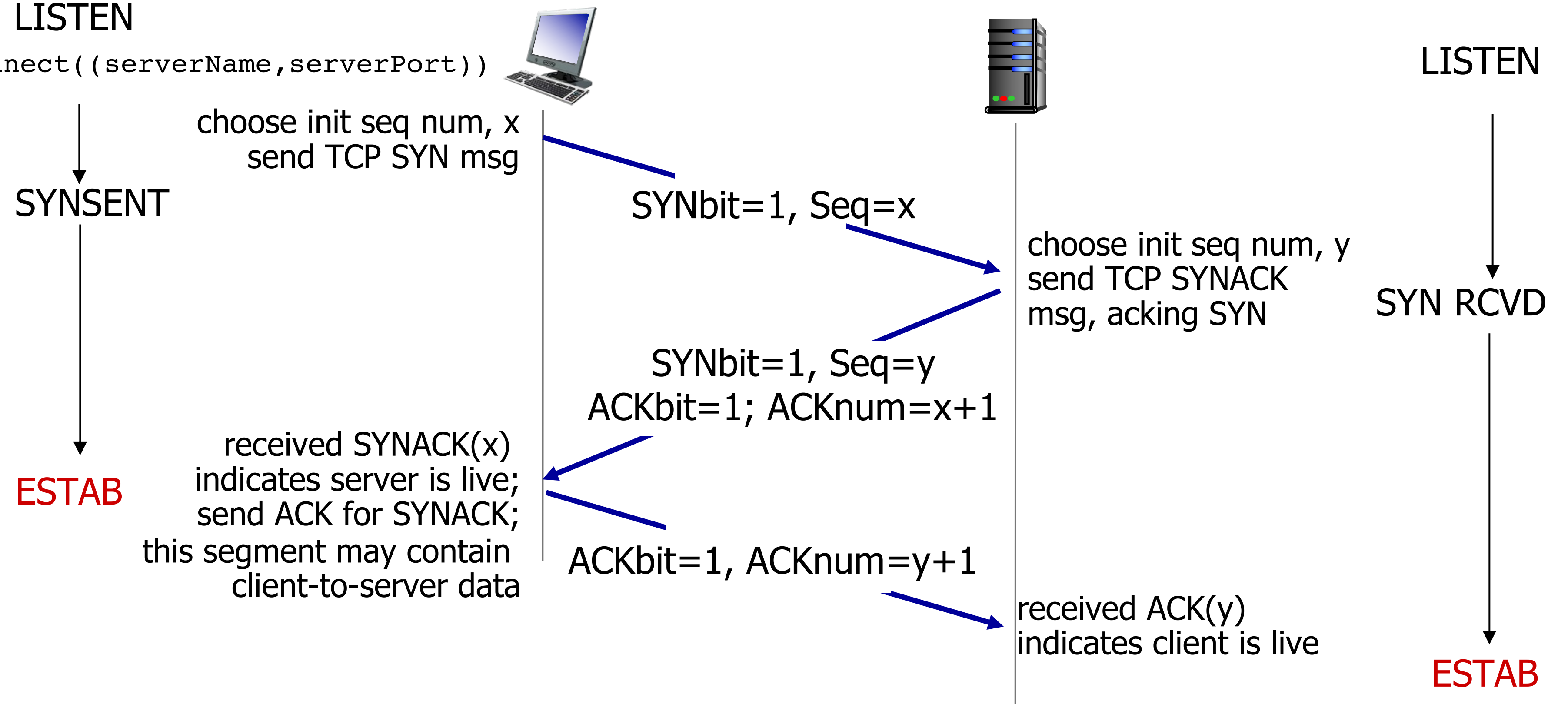
```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

Client state

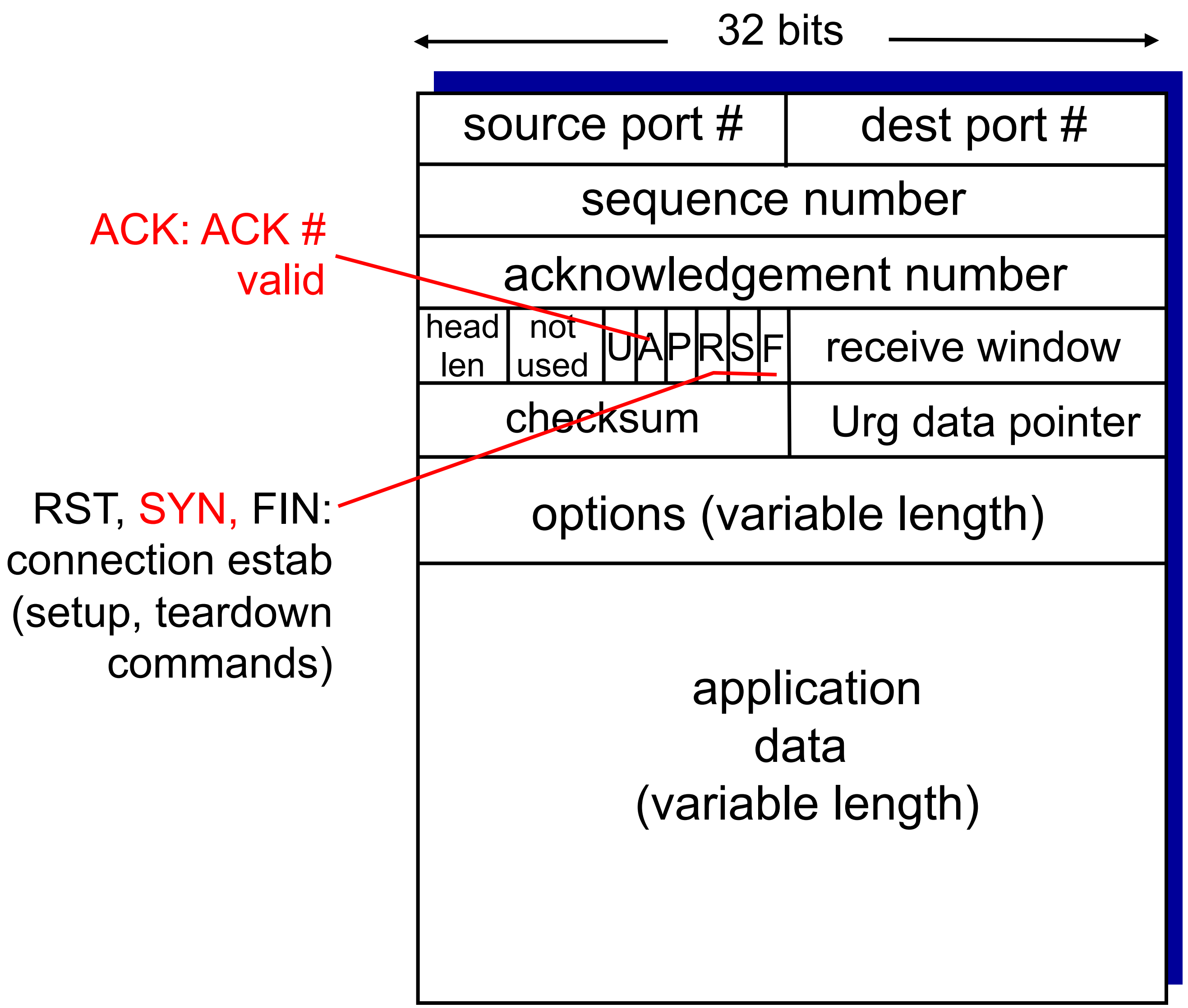
```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName,serverPort))
```



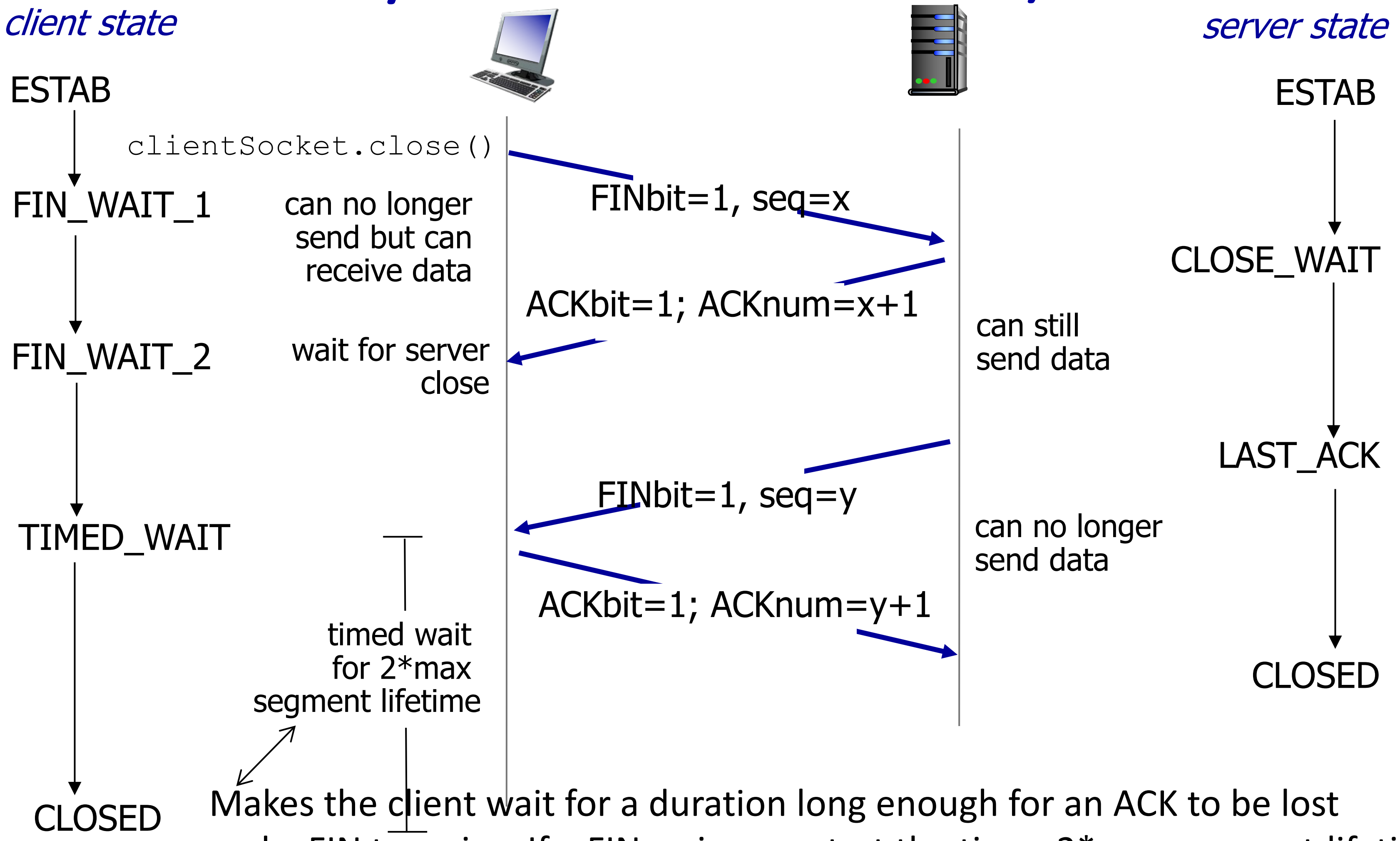
How to set SYNC, ACK bit?



Closing a TCP 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

Closing TCP connection (i.e., two 1-way subconnections)



Makes the client wait for a duration long enough for an ACK to be lost and a FIN to arrive. If a FIN arrives, restart the timer 2*max-segment-lifetime

Drop any delayed segments during timer=2*max-segment-time (2min default)

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Principles of congestion control

Congestion:

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

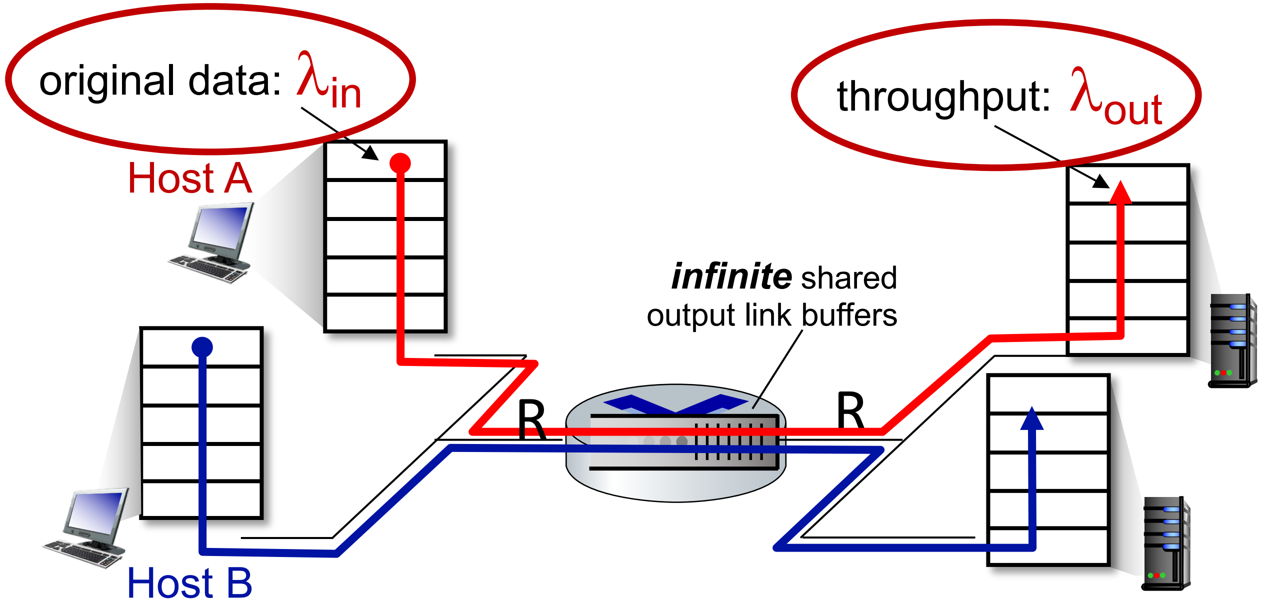
congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

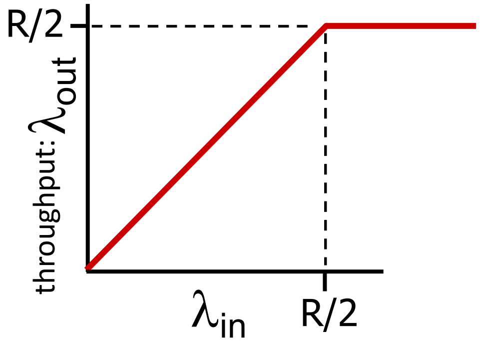
Causes/costs of congestion: scenario 1

Simplest scenario:

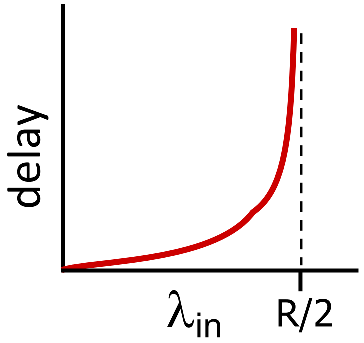
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed



Q: What happens as arrival rate λ_{in} approaches R/2?



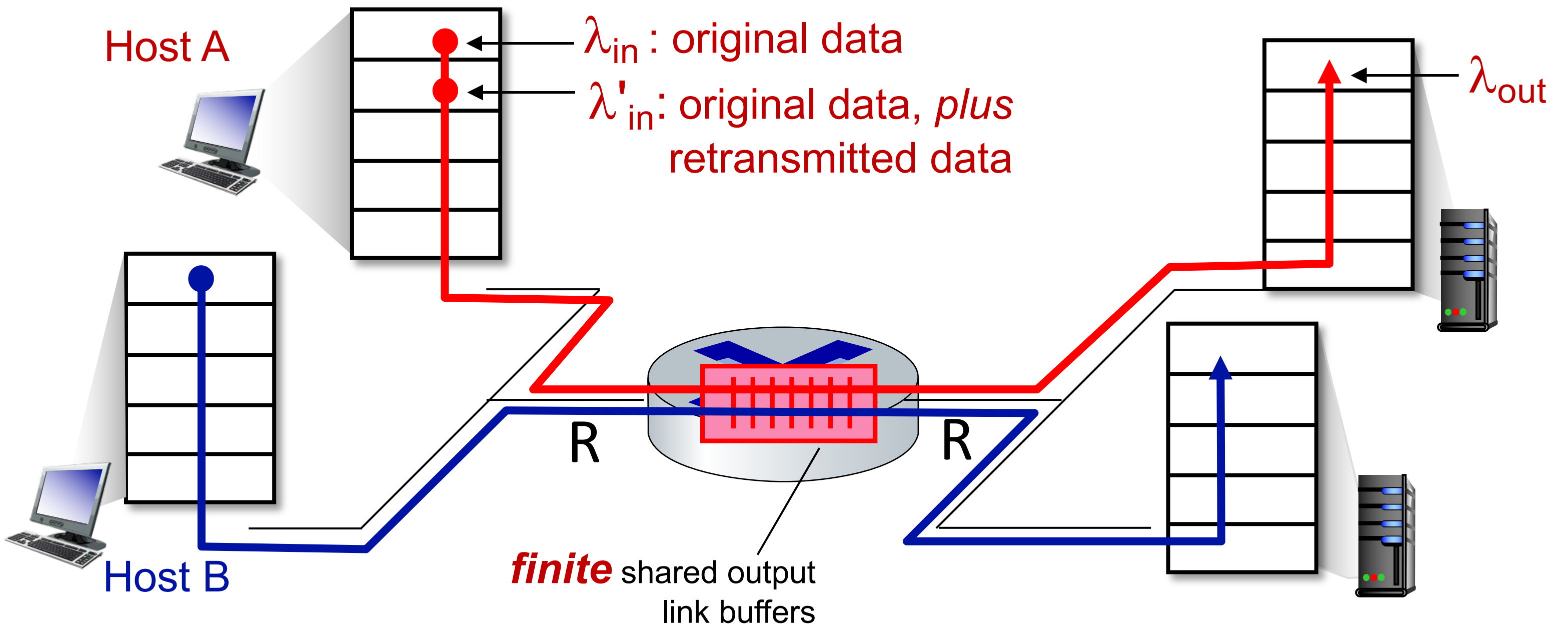
maximum per-connection throughput: R/2



large delays as arrival rate λ_{in} approaches capacity

Causes/costs of congestion: scenario 2

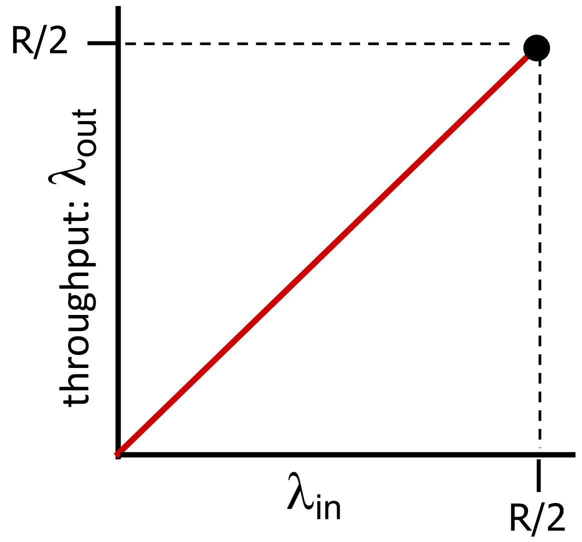
- one router, *finite* buffers
- sender retransmits lost, timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

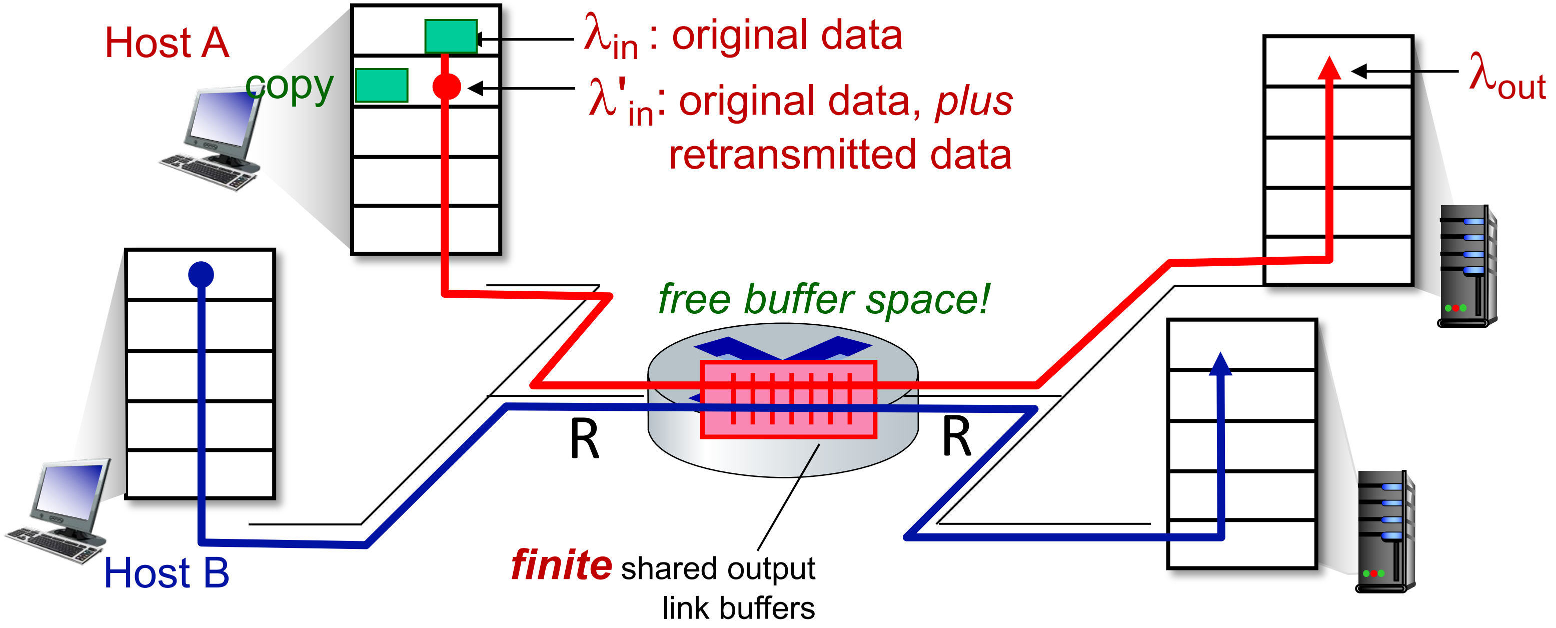


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available

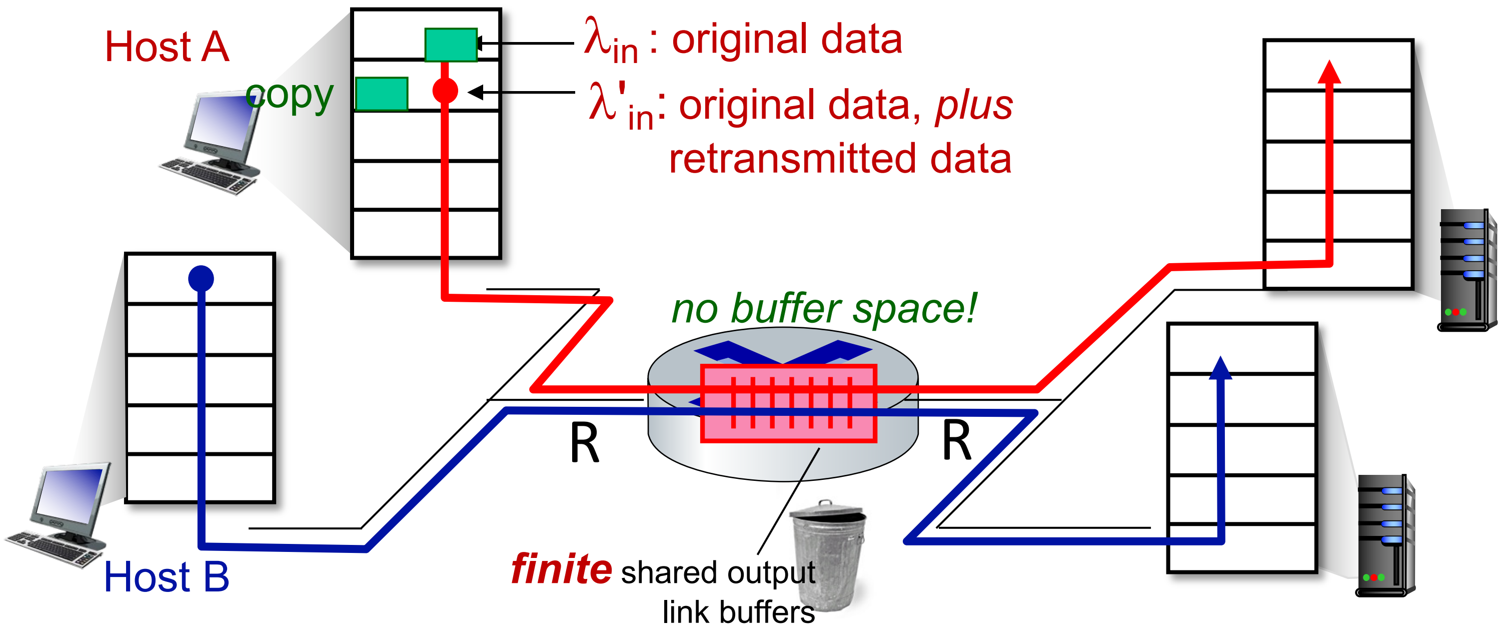




Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

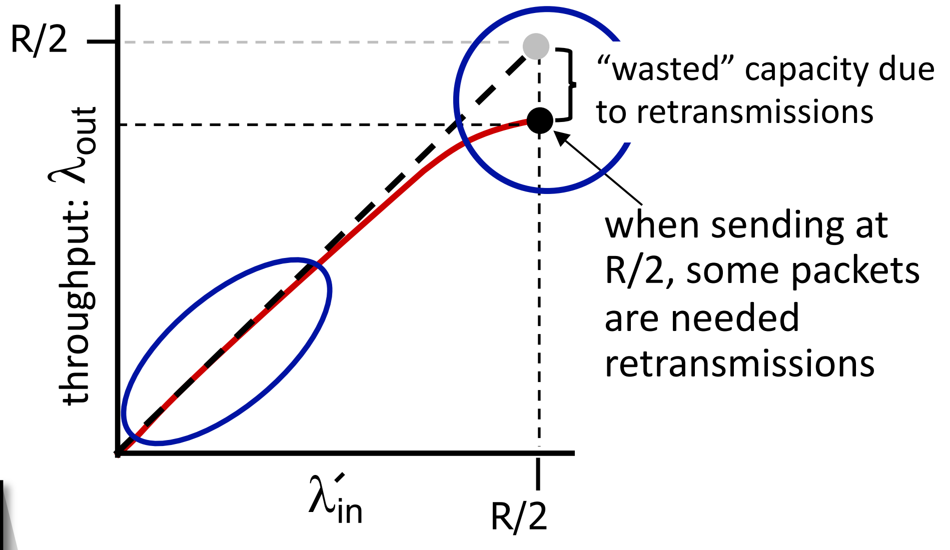
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

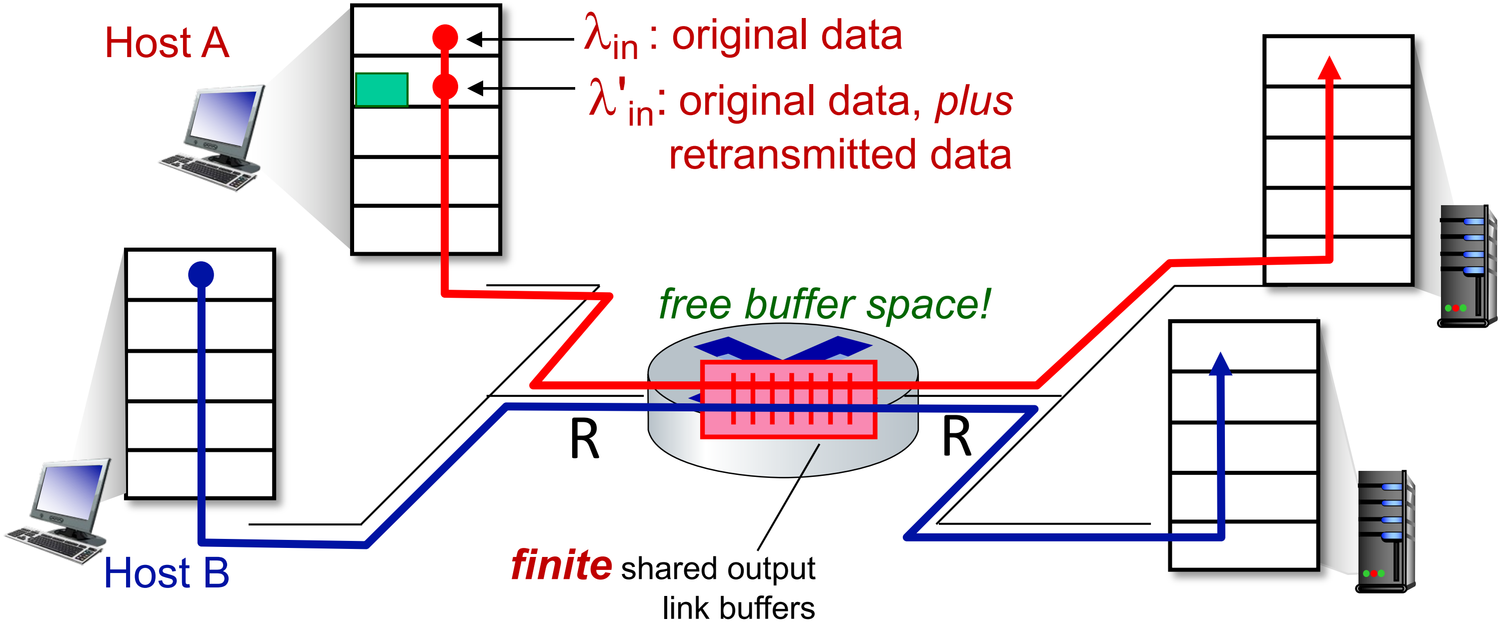


Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

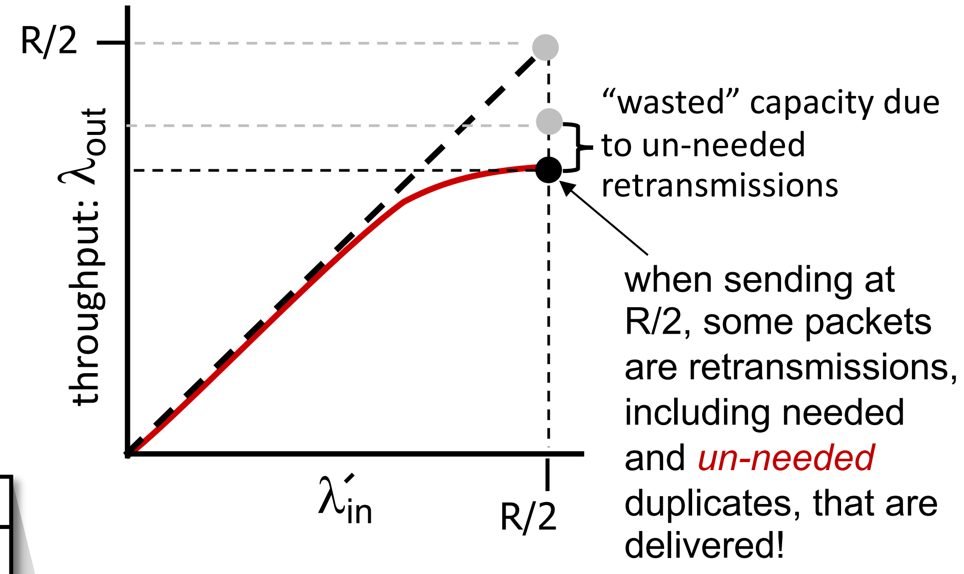


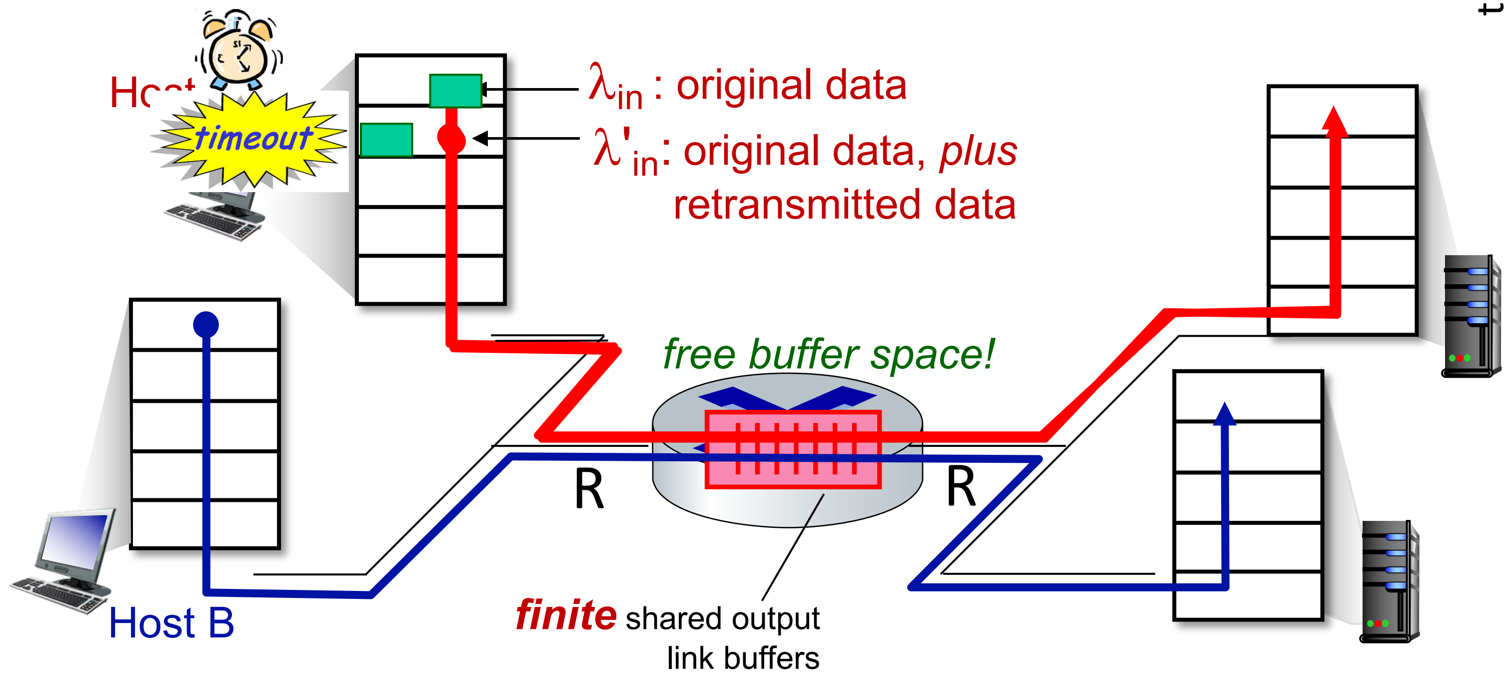


Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered

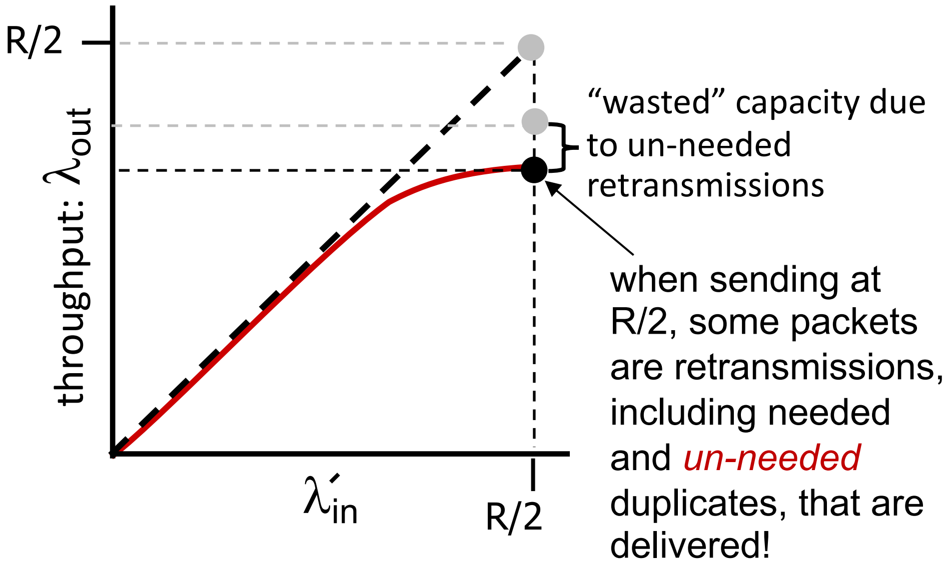




Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



"costs" of congestion:

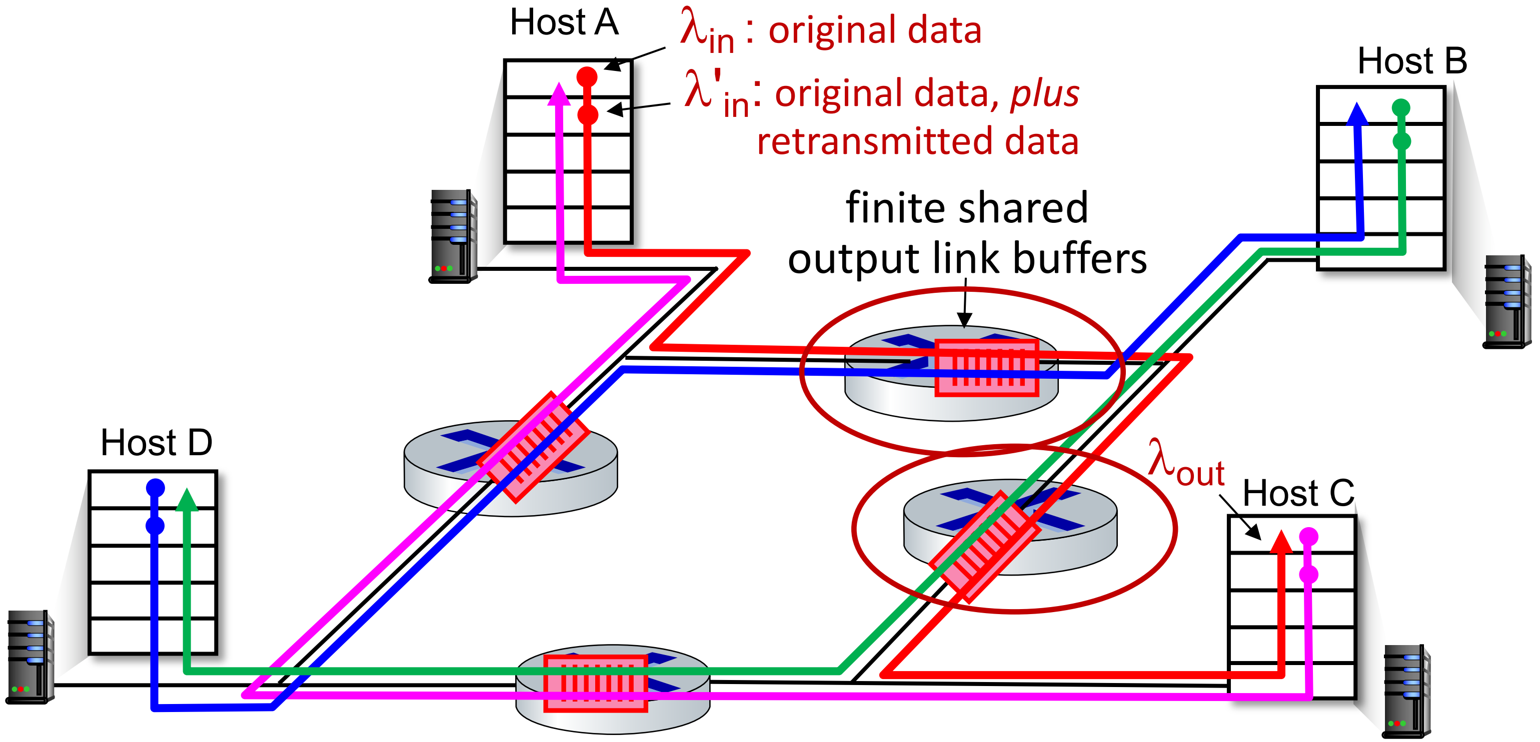
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3

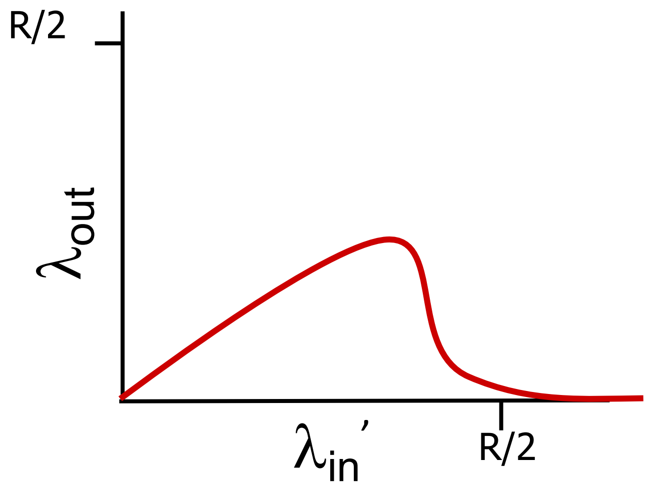
- *four* senders
- *multi-hop* paths
- timeout/retransmit

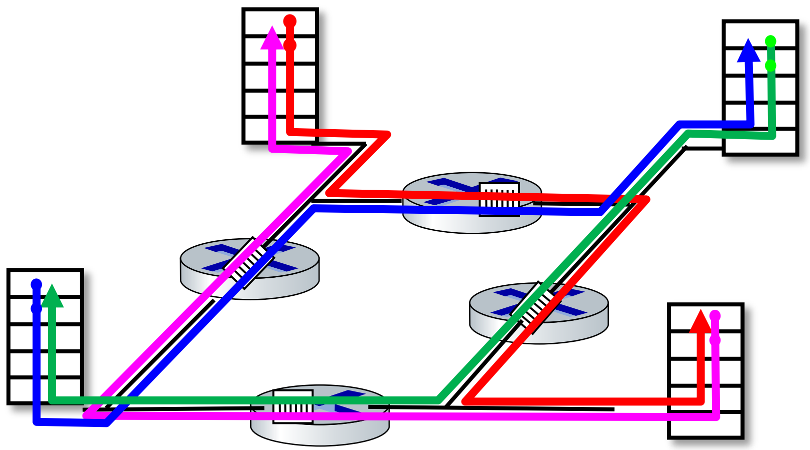
Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



Causes/costs of congestion: scenario 3



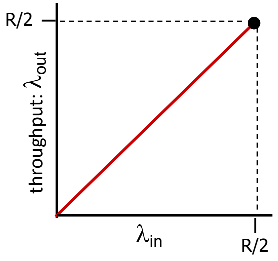


another "cost" of congestion:

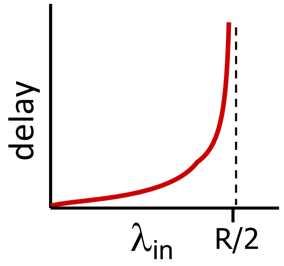
- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

Causes/costs of congestion: insights

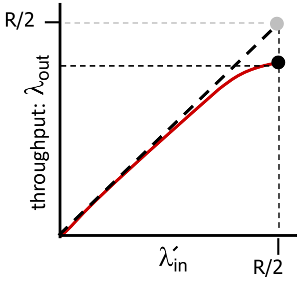
- throughput can never exceed capacity



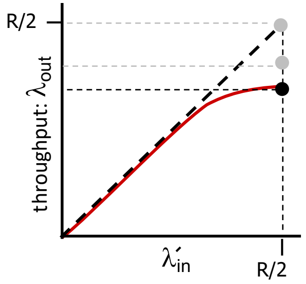
- delay increases as capacity approached



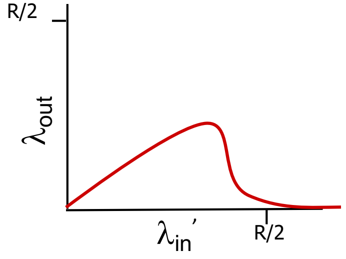
- loss/retransmission decreases effective throughput



- un-needed duplicates further decreases effective throughput



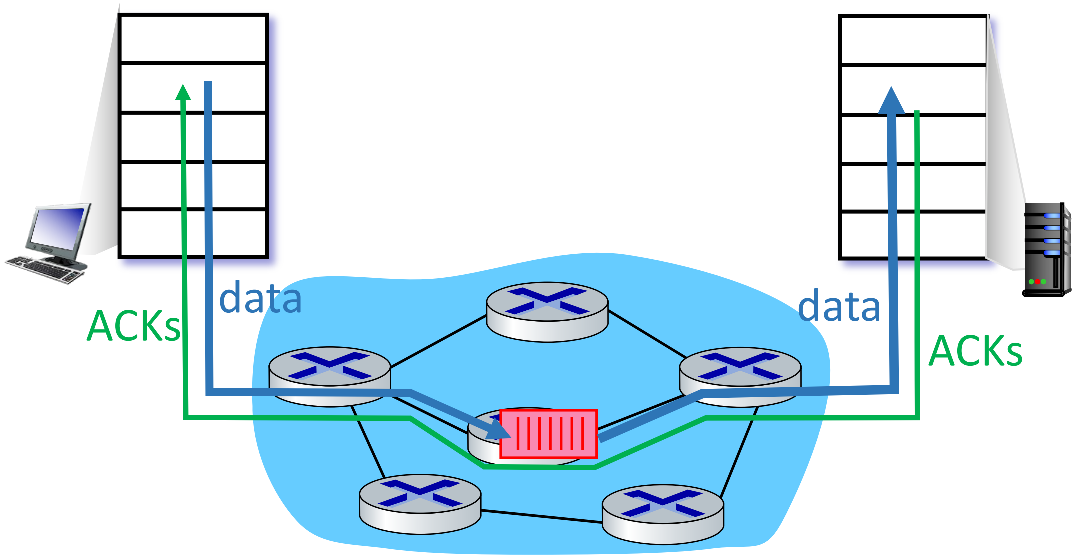
- upstream transmission capacity / buffering wasted for packets lost downstream



Approaches towards congestion control

End-end congestion control:

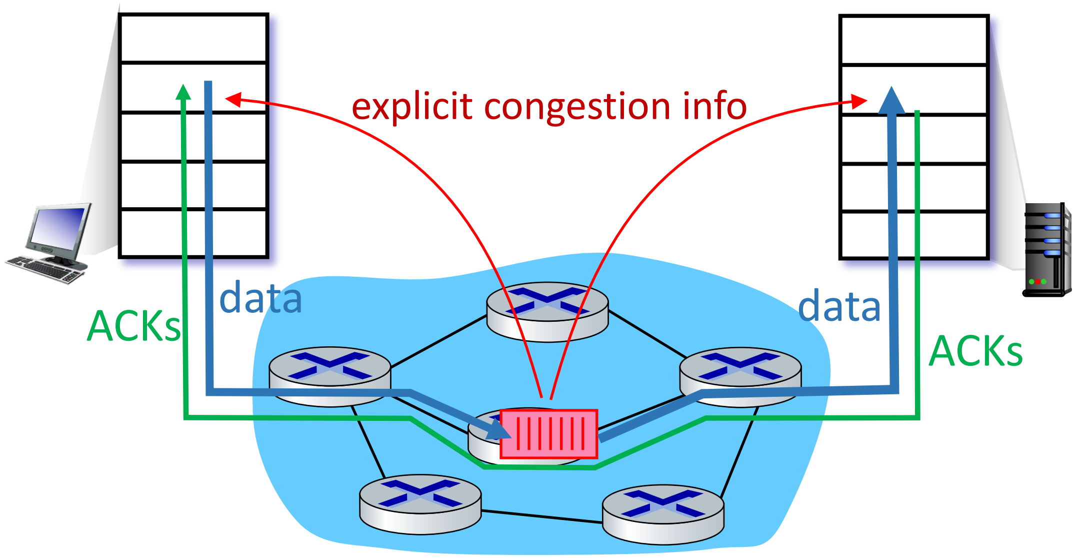
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

TCP Congestion Control

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

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

TCP Congestion Control

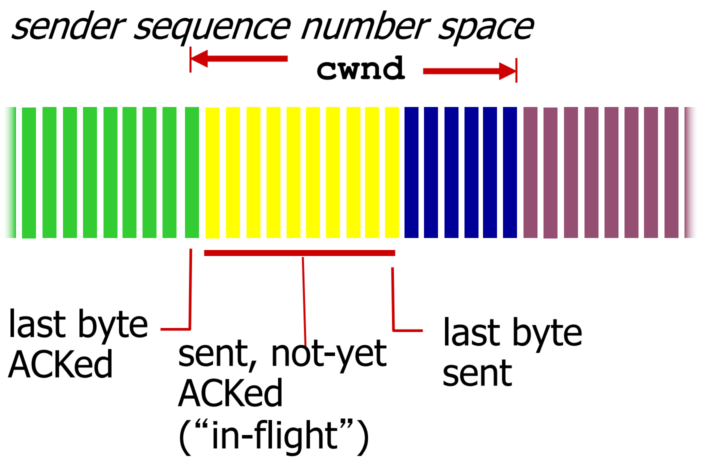
- Assumptions for congestion control
  - TCP pipelined reliable data transfer (SR in the common cases)
  - Works with TCP flow control
  - ***<u>All losses of TCP segments</u>*** are due to Internet ***congestion***
    - Ignore the transmission errors (since link quality is good in general)
- Mechanism: Window-based congestion control
  - Adjust the window size for SR to change the TCP sending rate
- Changes in congestion window size (cwnd)
    - Slow increases to absorb new bandwidth
    - Quick decreases to eliminate congestion

TCP Congestion Control

- sender limits transmission:

LastByteSent-LastByteAcked

≤ cwnd



- **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

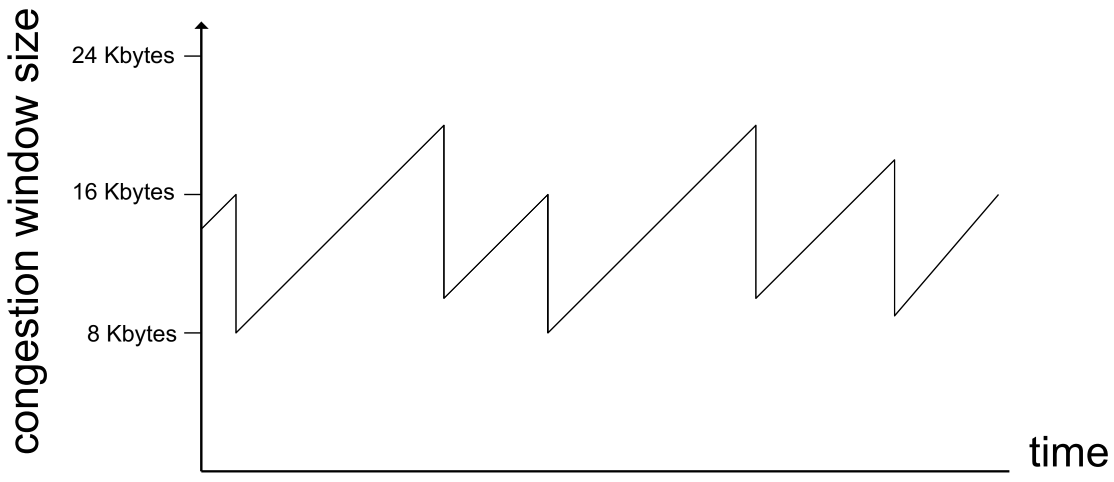
three mechanisms:

- 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

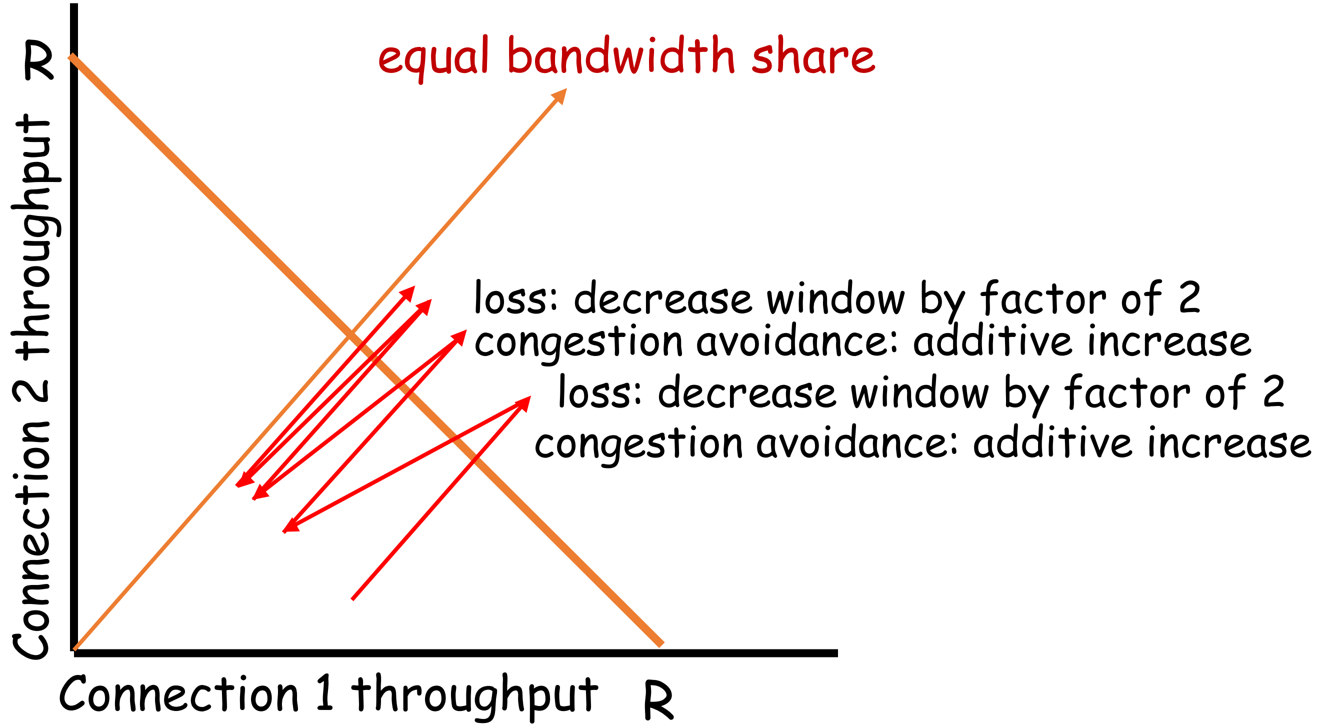
Saw tooth behavior: probing for bandwidth



What AIMD? TCP Fairness

Two competing sessions:

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



TCP Congestion Control (RFC 5681)

How to implement TCP Congestion Control?

Multiple algorithms work together:

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

TCP Congestion Control Summary

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+=(MSS/cwnd) * MSS per ACK
fast retransmit	3 duplicate ACK	reduce the cwnd by half (multicative decreasing)	ssthresh = max(cwnd/2,2MSS) 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/2,2MSS) cwnd = 1MSS; retx the lost packet

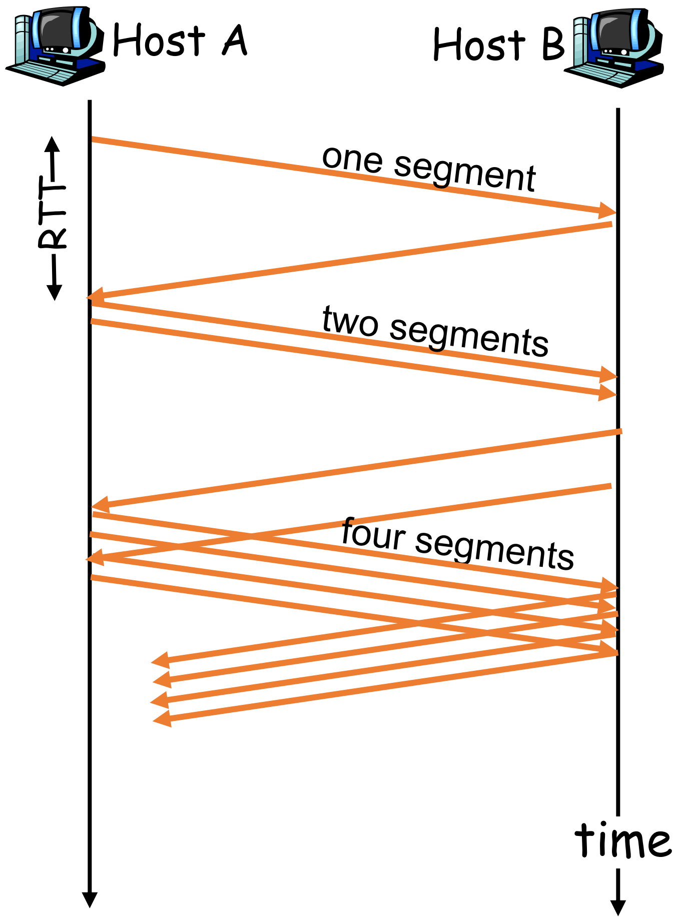
TCP Slow Start

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

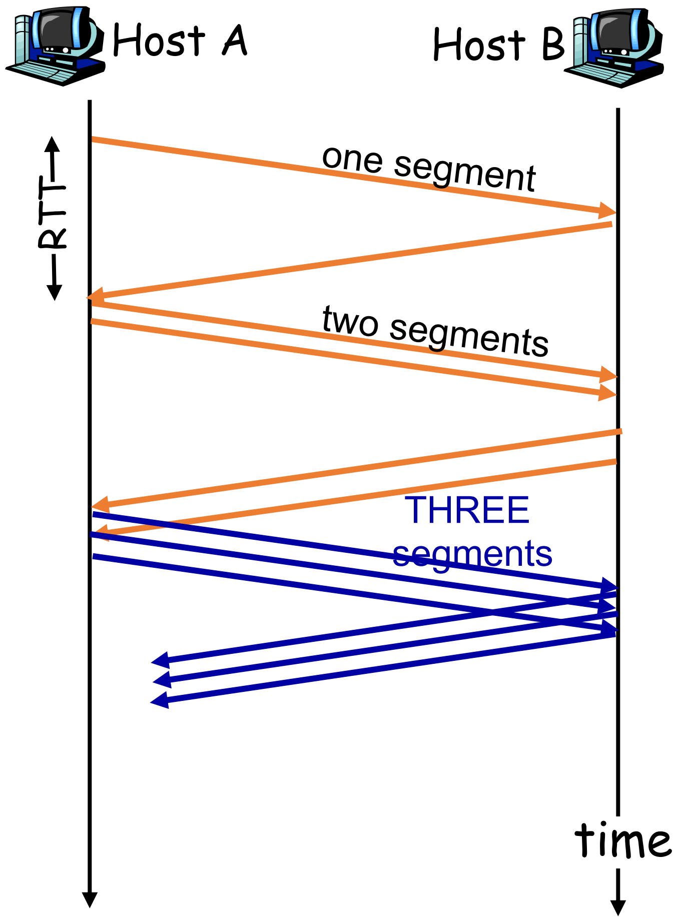


Congestion Avoidance

- 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+=(MSS/cwnd) * MSS per ACK
fast retransmit	3 duplicate ACK	reduce the cwnd by half (multicative decreasing)	ssthresh = max(cwnd/2,2MSS) 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/2,2MSS) cwnd = 1MSS; retx the lost packet

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

Algorithm for fast rexmit/fast recovery

- 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

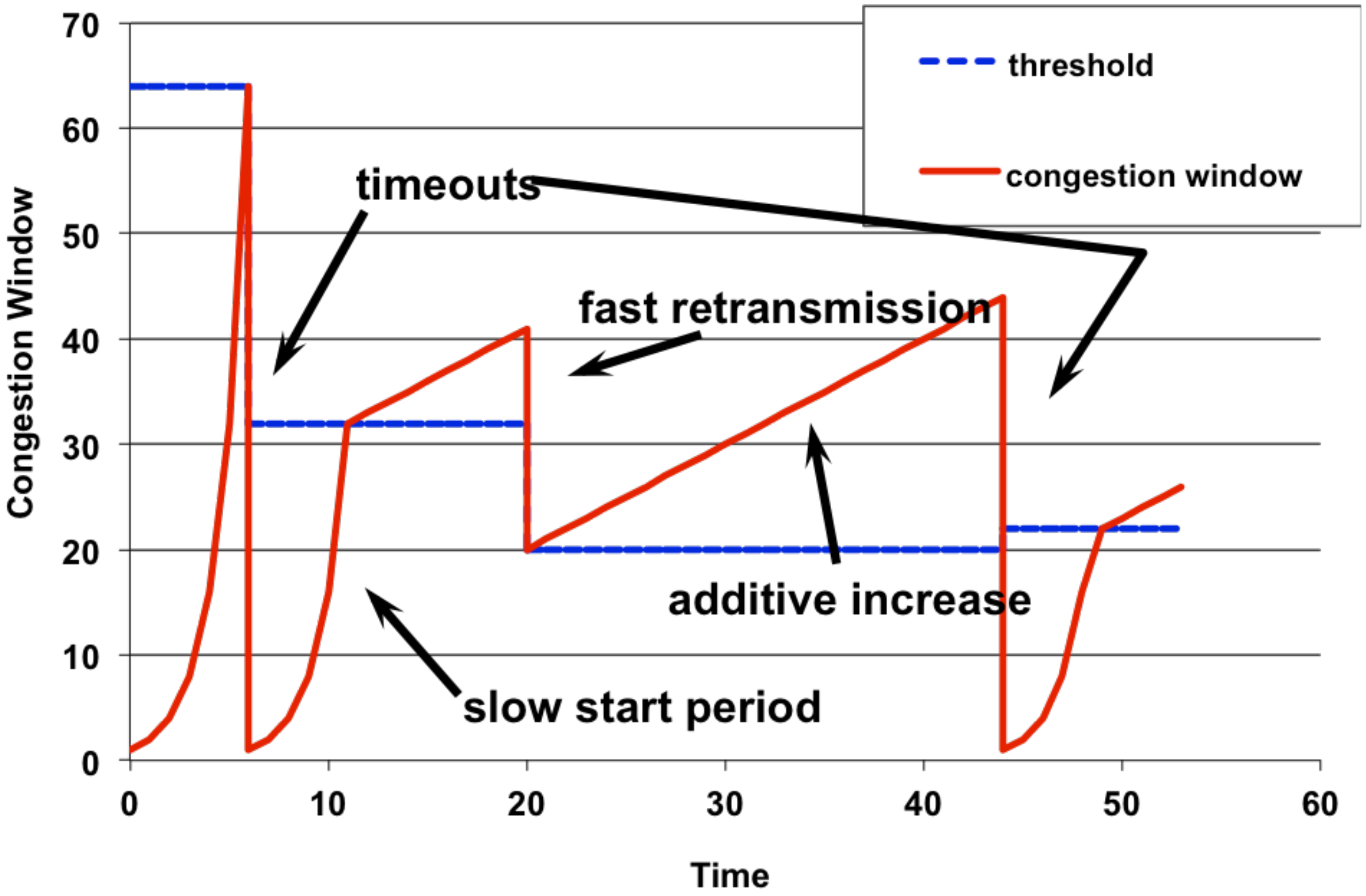
when retransmission timer expires

- 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 Summary

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+=(MSS/cwnd) * MSS per ACK
fast retransmit	3 duplicate ACK	reduce the cwnd by half (multicative decreasing)	ssthresh = max(cwnd/2,2MSS) 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/2,2MSS) cwnd = 1MSS; retx the lost packet

Putting Things Together in TCP

How Selective repeat, congestion control, flow control work together:

- 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

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 send window size min(rwnd,cwnd) =cwnd
- Assume 1 acknowledgement per packet (i.e., no delayed ACK is used), 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
- Assume that the receiver will buffer out of order packets (like selective repeat)

We will how TCP congestion control algorithms work together

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

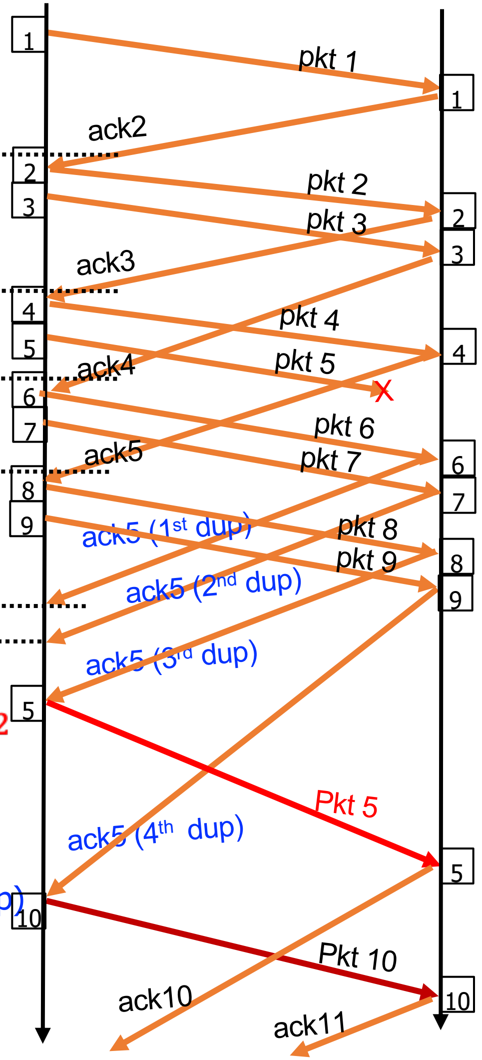
[5][6][7][8][9][ ]

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 | cwnd 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

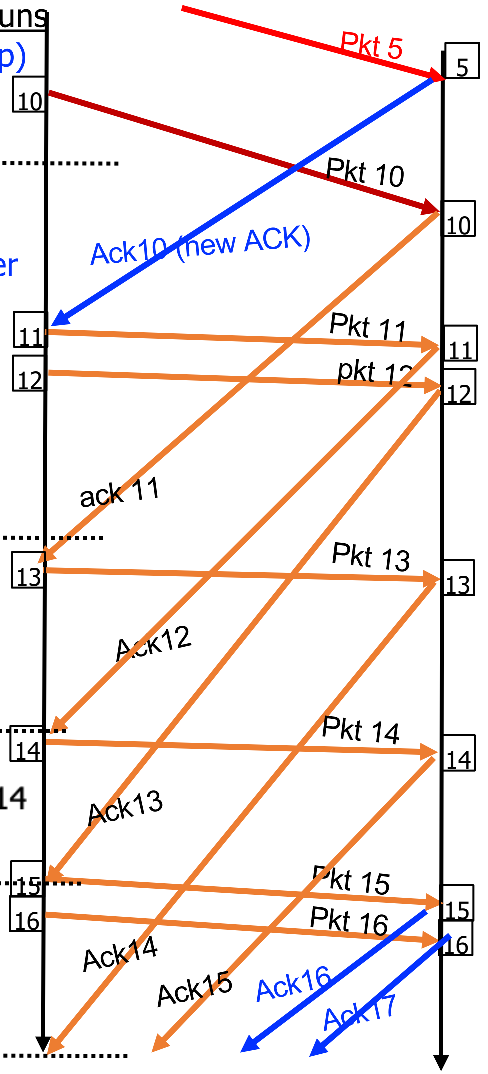
12	13	14			

Congestion avoidance upon ack 13

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

Send packets 15, 16

13	14	15	16		



Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Evolving transport-layer functionality

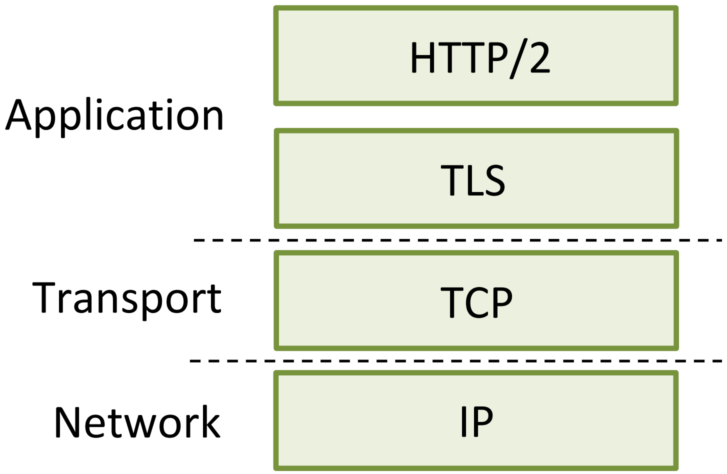
- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)



HTTP/2 over TCP

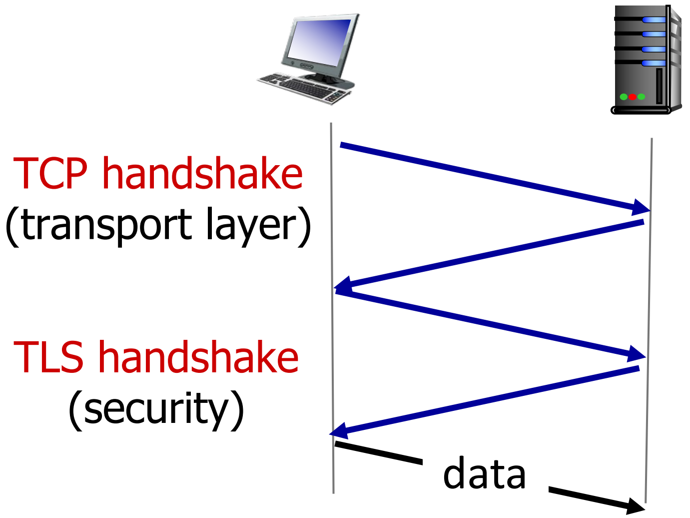
QUIC: Quick UDP Internet Connections

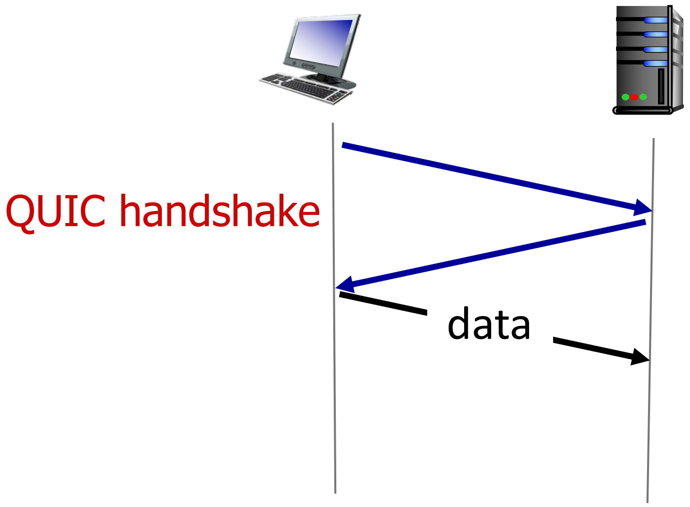
adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Connection establishment





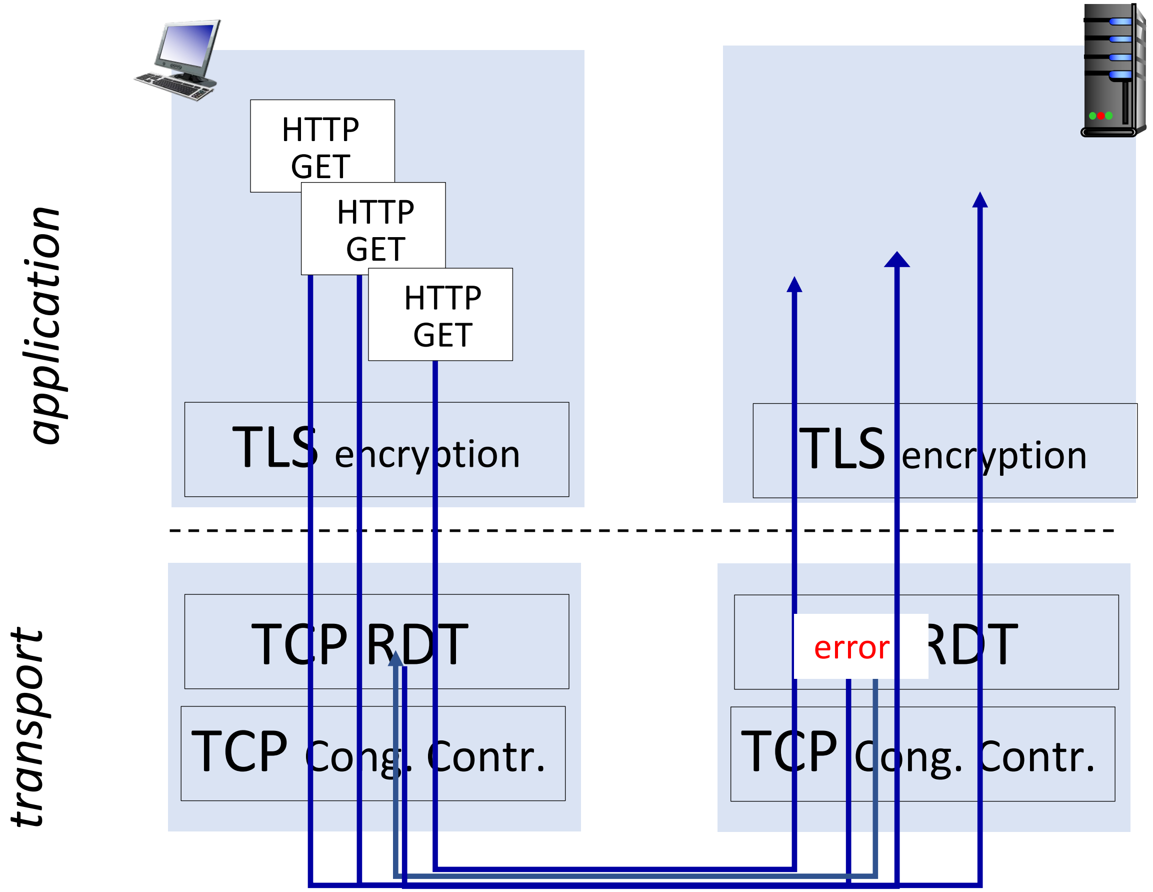
TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane