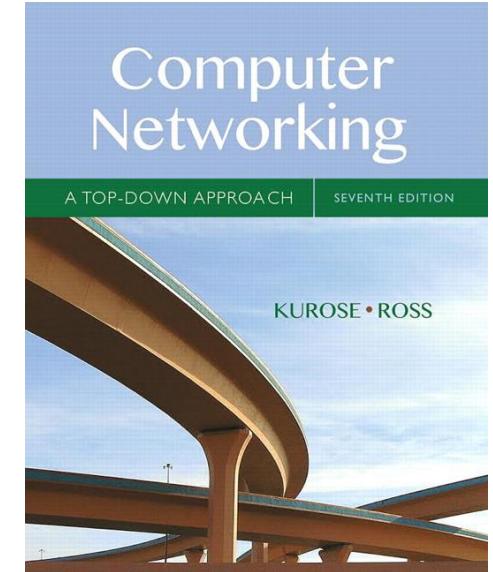


# Transport Layer

Computer Networks – COSC 4P14

# Course Outline



- **Introduction**
- **Application Layer**
- **Transport Layer 1**
- **Transport Layer 2**
- **Network Layer - Data Plane**
- **Network Layer - Control Plane**
- **Link Layer and LANs**
- **Wireless and Mobile Networks 1**
- **Wireless and Mobile Networks 2**
- **Secure Networks – Introduction**
- **Secure Networks – Attacks & Measures**
- **New technologies – IoT / NFV / SDN / ICN**

Slides based on Computer Networking's slides:  
© All material copyright 1996-2016  
J.F Kurose and K.W. Ross, All Rights Reserved

# Last Class

- Application layer
  - Principles of network applications
  - Web, HTTP, email, DNS
  - P2P
  - Streaming
  - APIs

# Transport Layer

our goals:

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

# 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

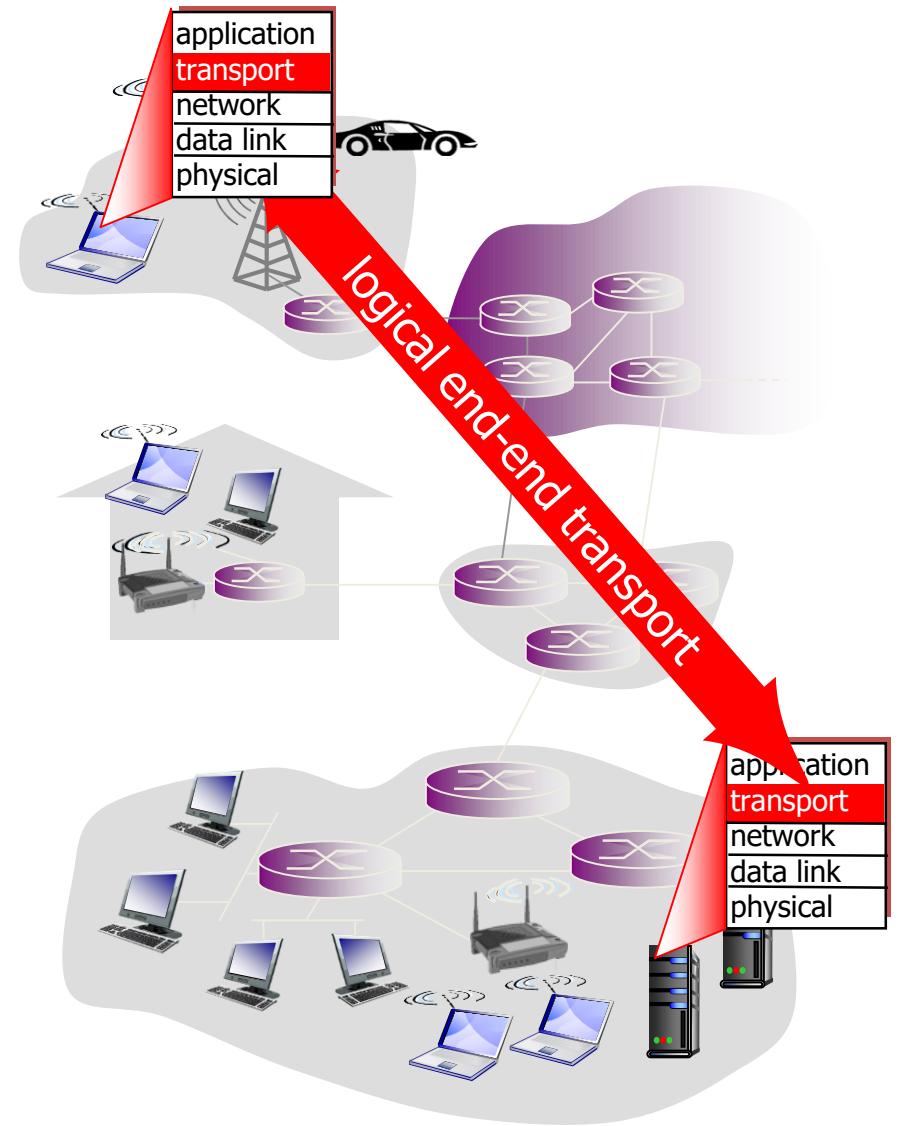
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# Transport services and protocols

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



# Transport vs. network layer

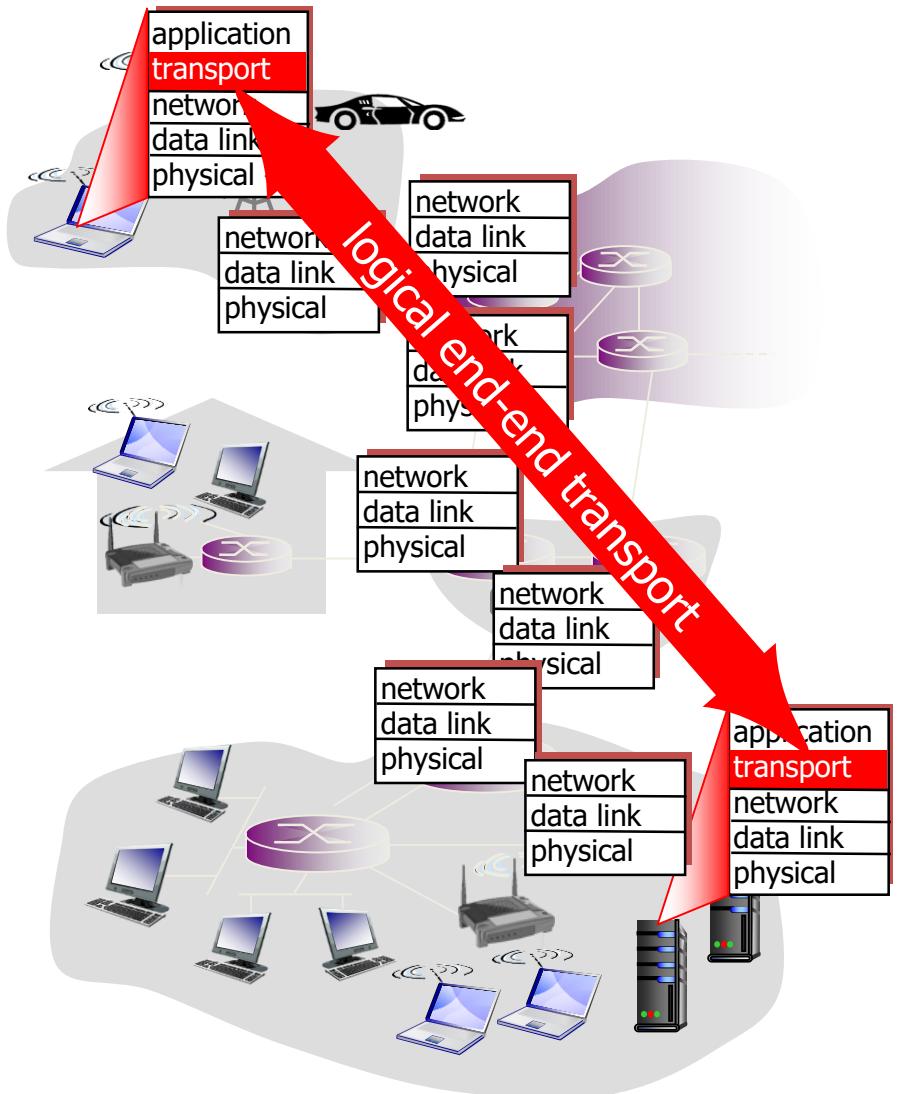
- *network layer*: logical communication between hosts
- *transport layer*: logical communication between processes
  - relies on, enhances, network layer services

## *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



# 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

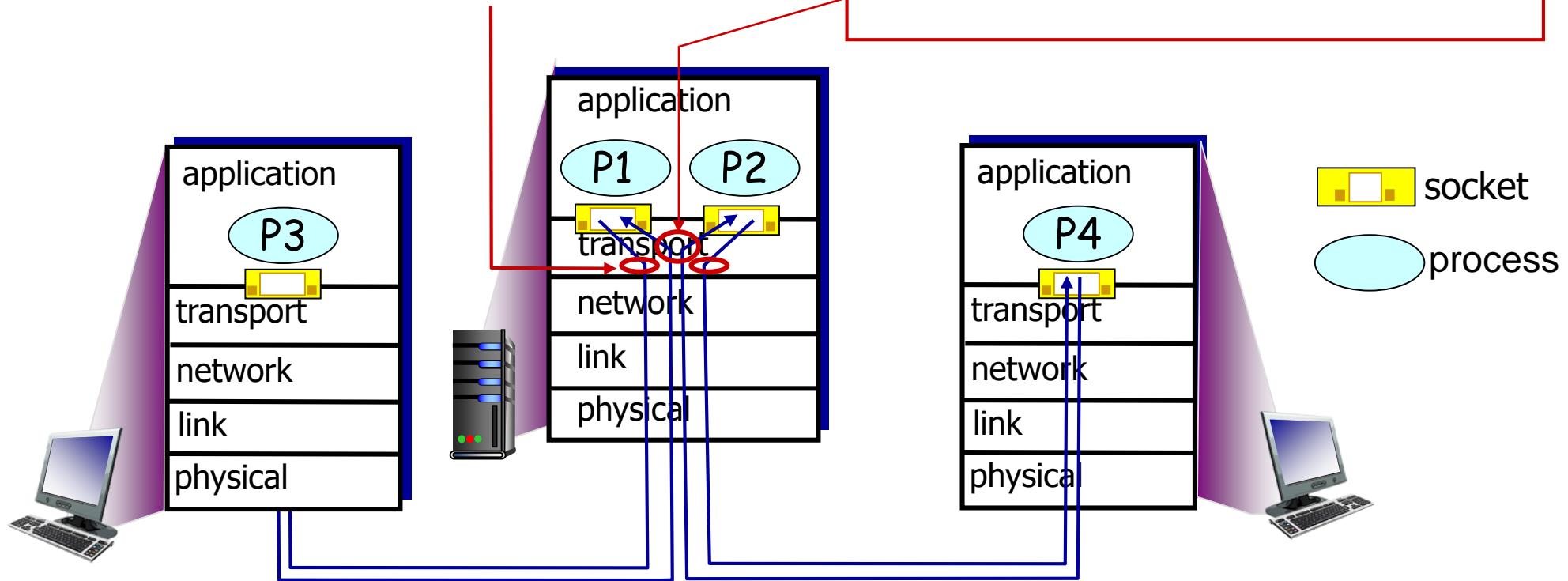
# Multiplexing/demultiplexing

*multiplexing at sender:*

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

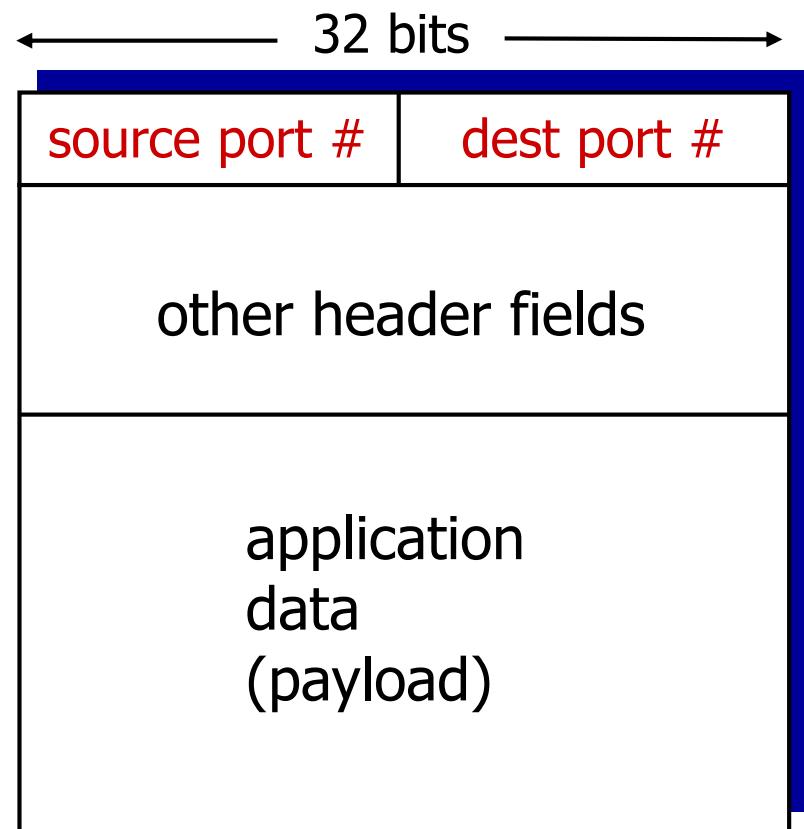
*demultiplexing at receiver:*

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:* created socket has host-local port #:

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

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

- when host receives UDP segment:

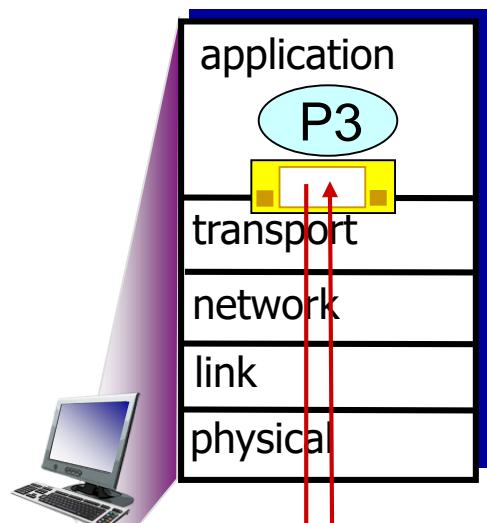
- checks destination port # in segment
- directs UDP segment to socket with that port #



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

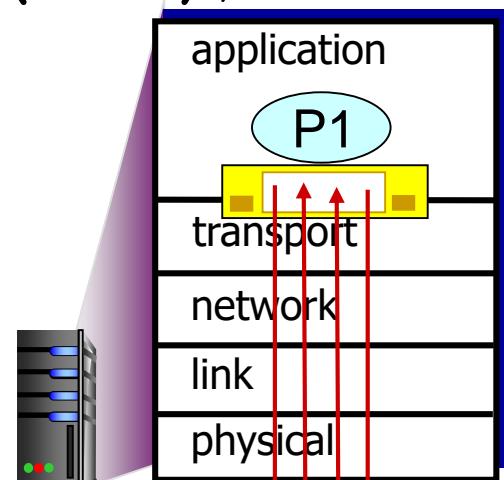
# Connectionless demux: example

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

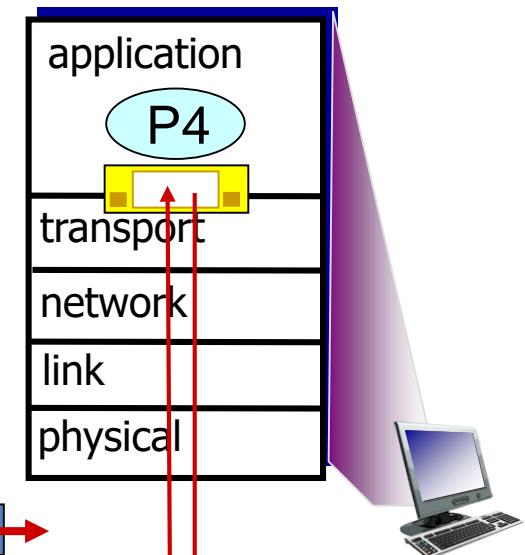


## DatagramSocket

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



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



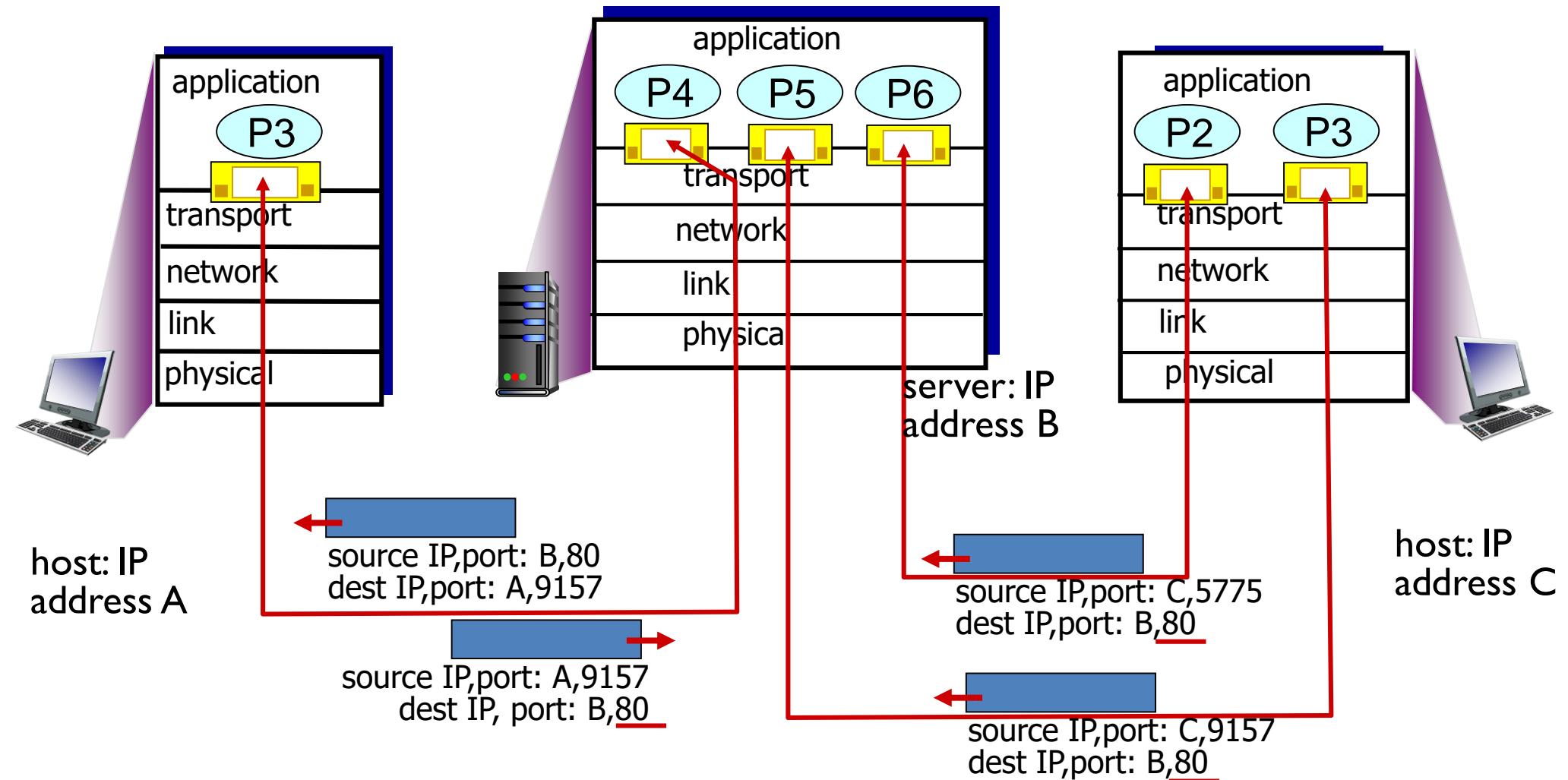
source port: 9157  
dest port: 6428

source port: ?  
dest port: ?

# Connection-oriented demux

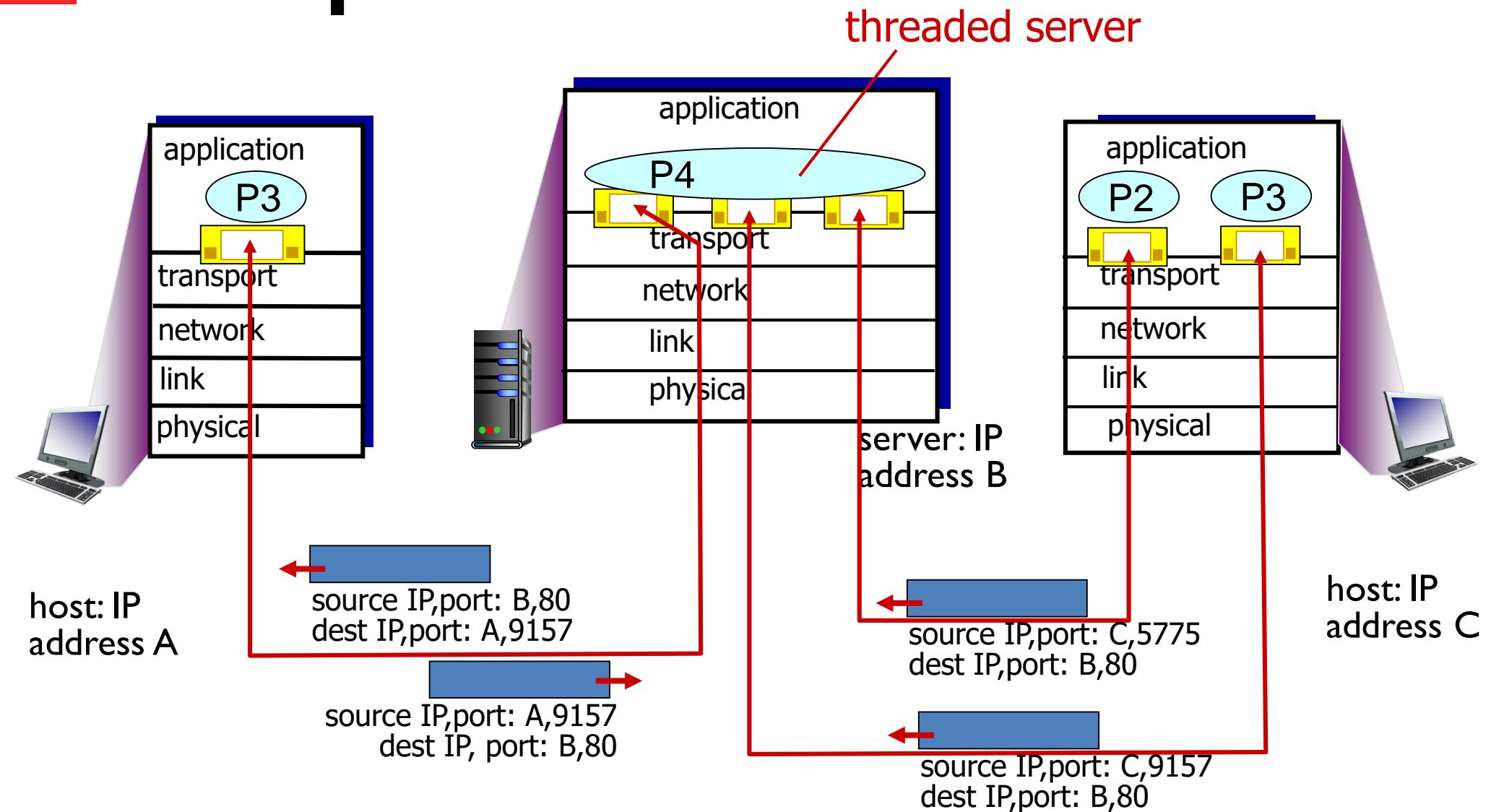
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- 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

# Connection-oriented demux: example



# 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

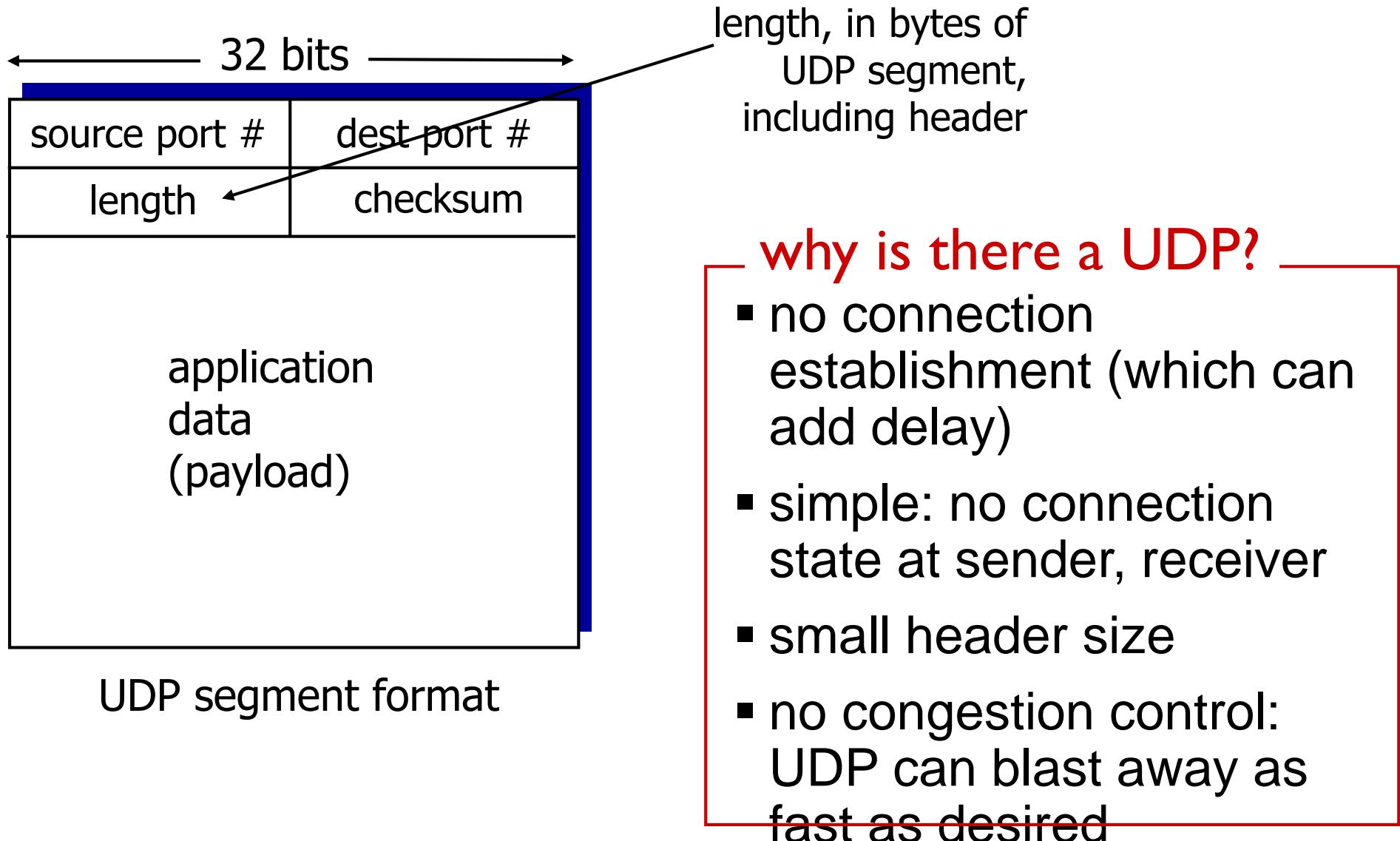
3.7 TCP congestion control

# 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 use:
  - 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 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
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
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/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# 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

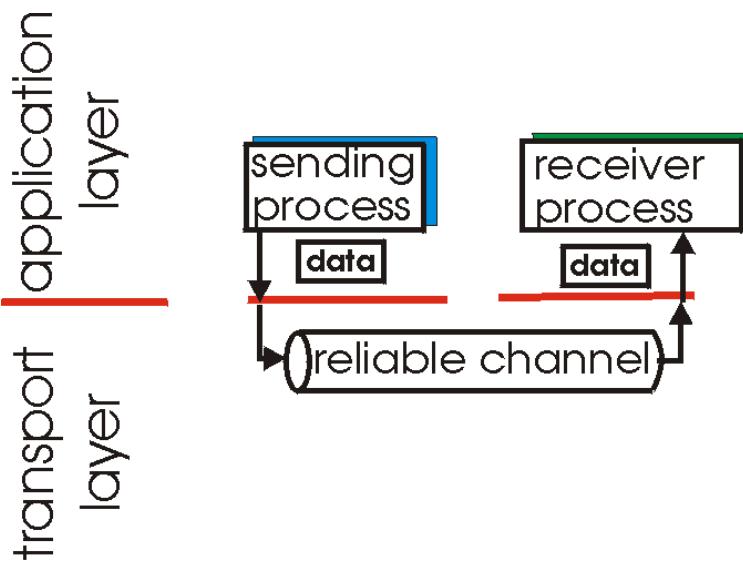
- segment structure
- reliable data transfer
- flow control
- connection management

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!

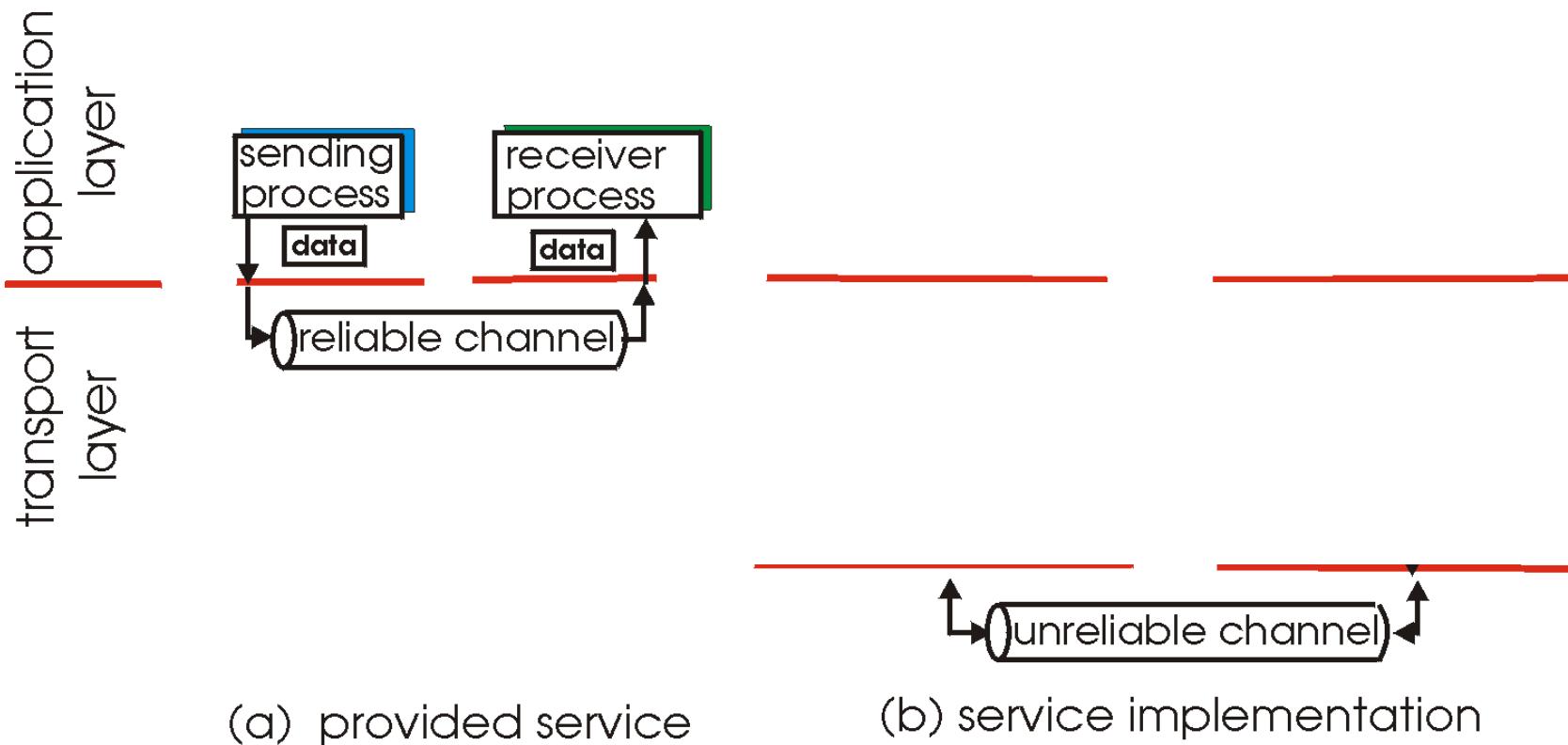


(a) provided service

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

# Principles of reliable data transfer

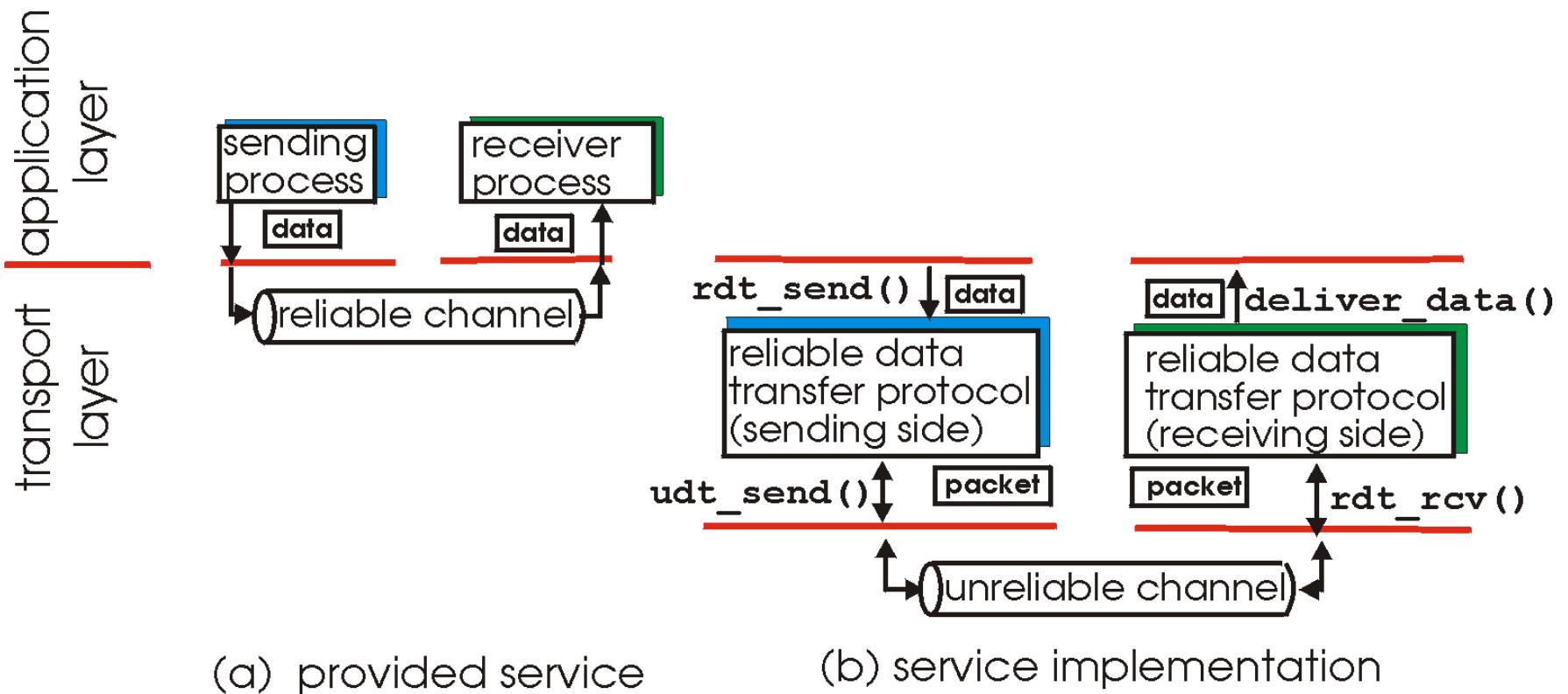
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

# Principles of reliable data transfer

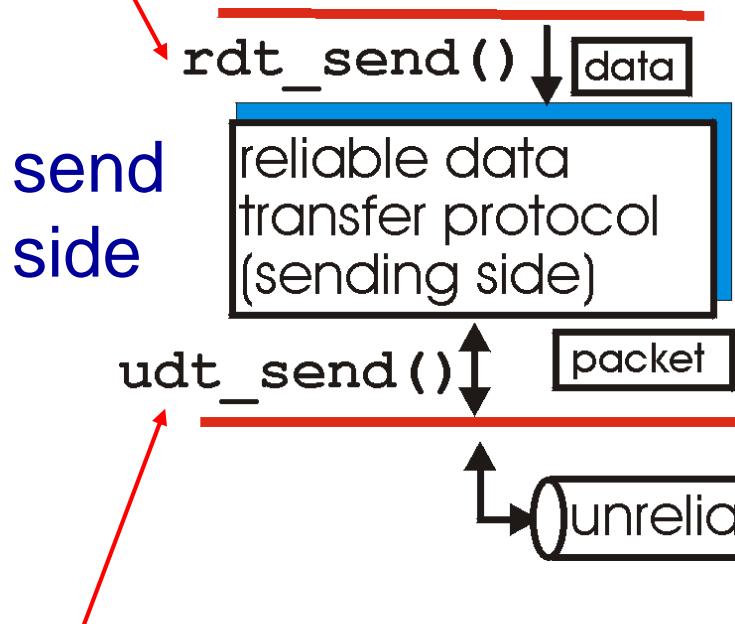
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

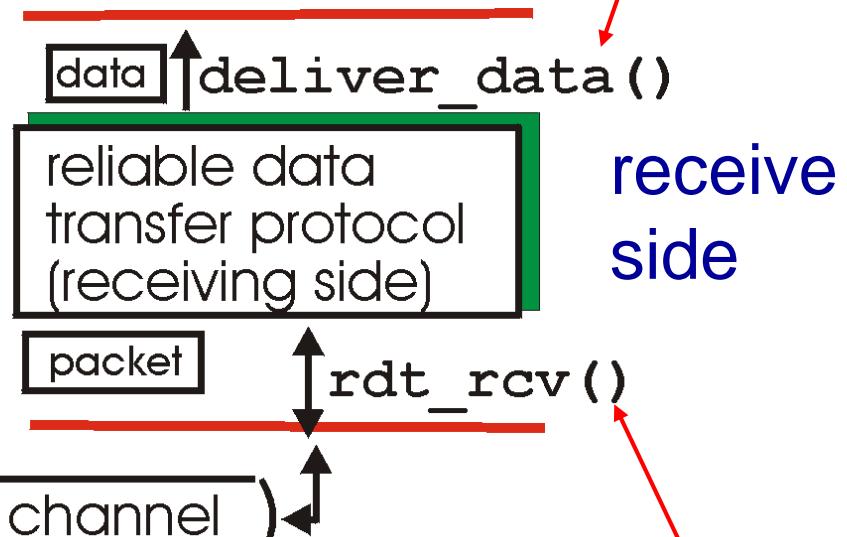
# Reliable data transfer: getting started

**rdt\_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



**udt\_send()**: called by rdt, to transfer packet over unreliable channel to receiver

**deliver\_data()**: called by **rdt** to deliver data to upper



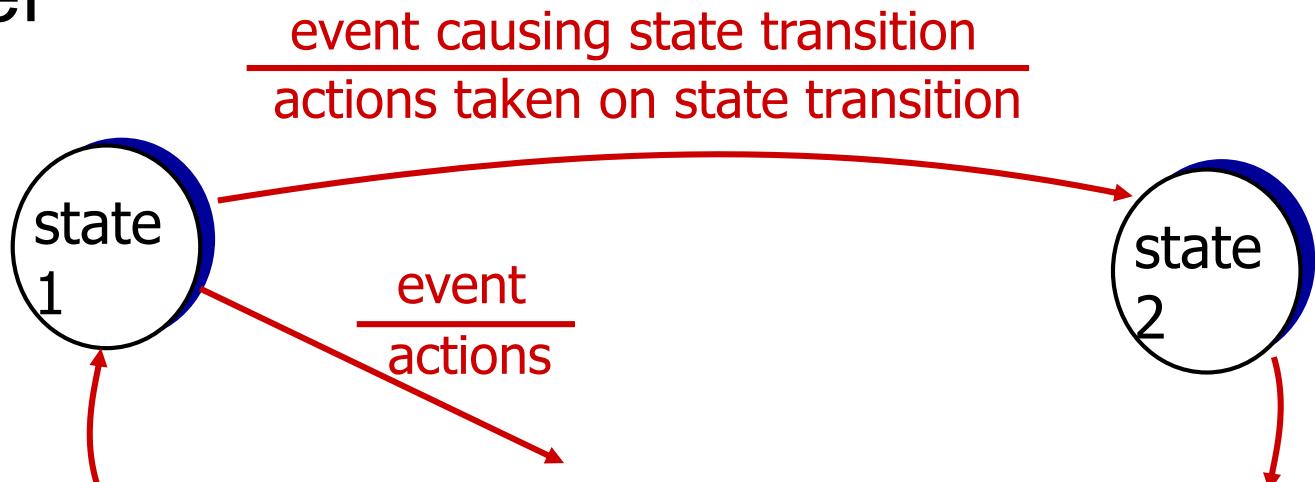
**rdt\_rcv()**: called when packet arrives on rcv-side of channel

# Reliable data transfer: getting started

we'll:

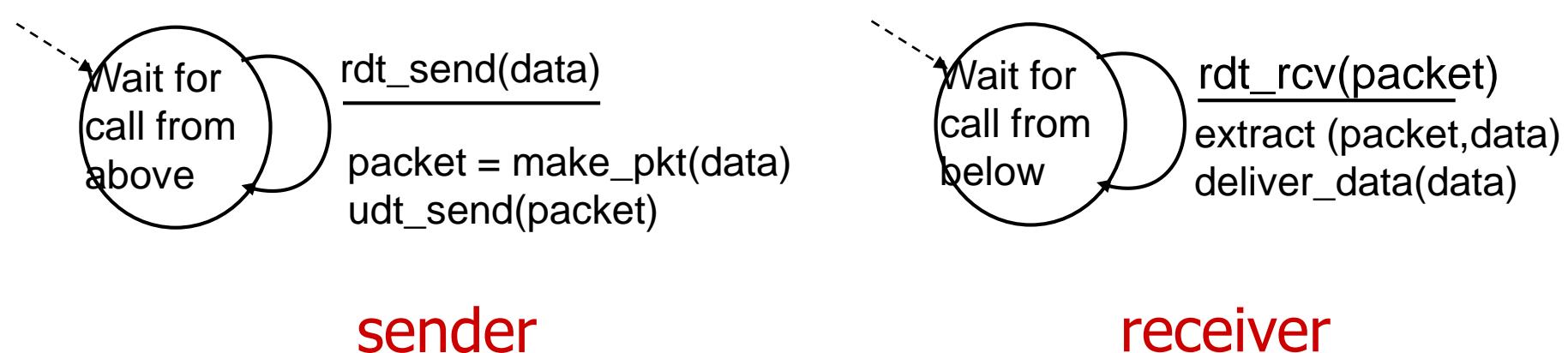
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

**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



sender

receiver

# rdt2.0: channel with bit errors

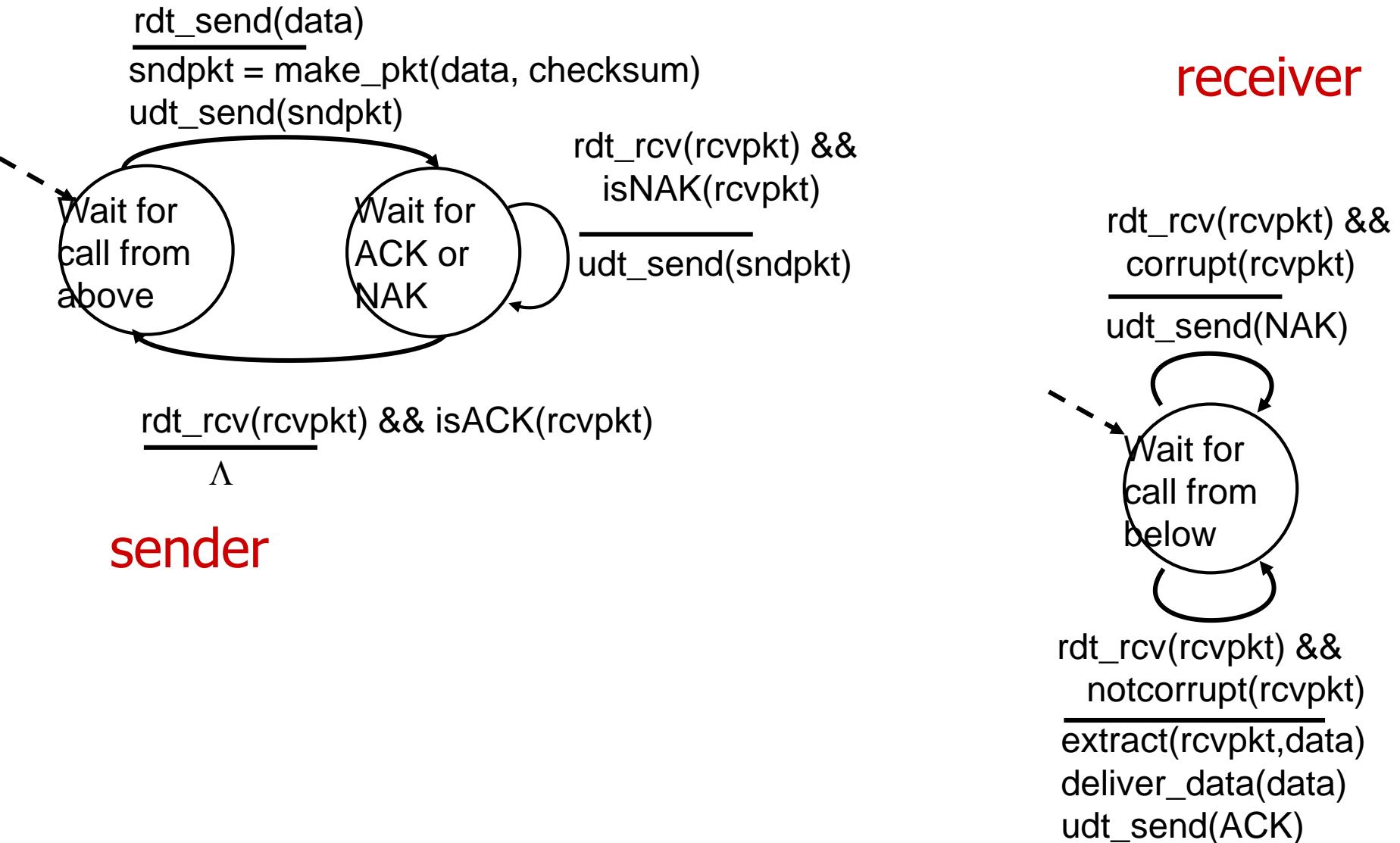
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the question: how to recover from errors:*

*How do humans recover from “errors” during conversation?*

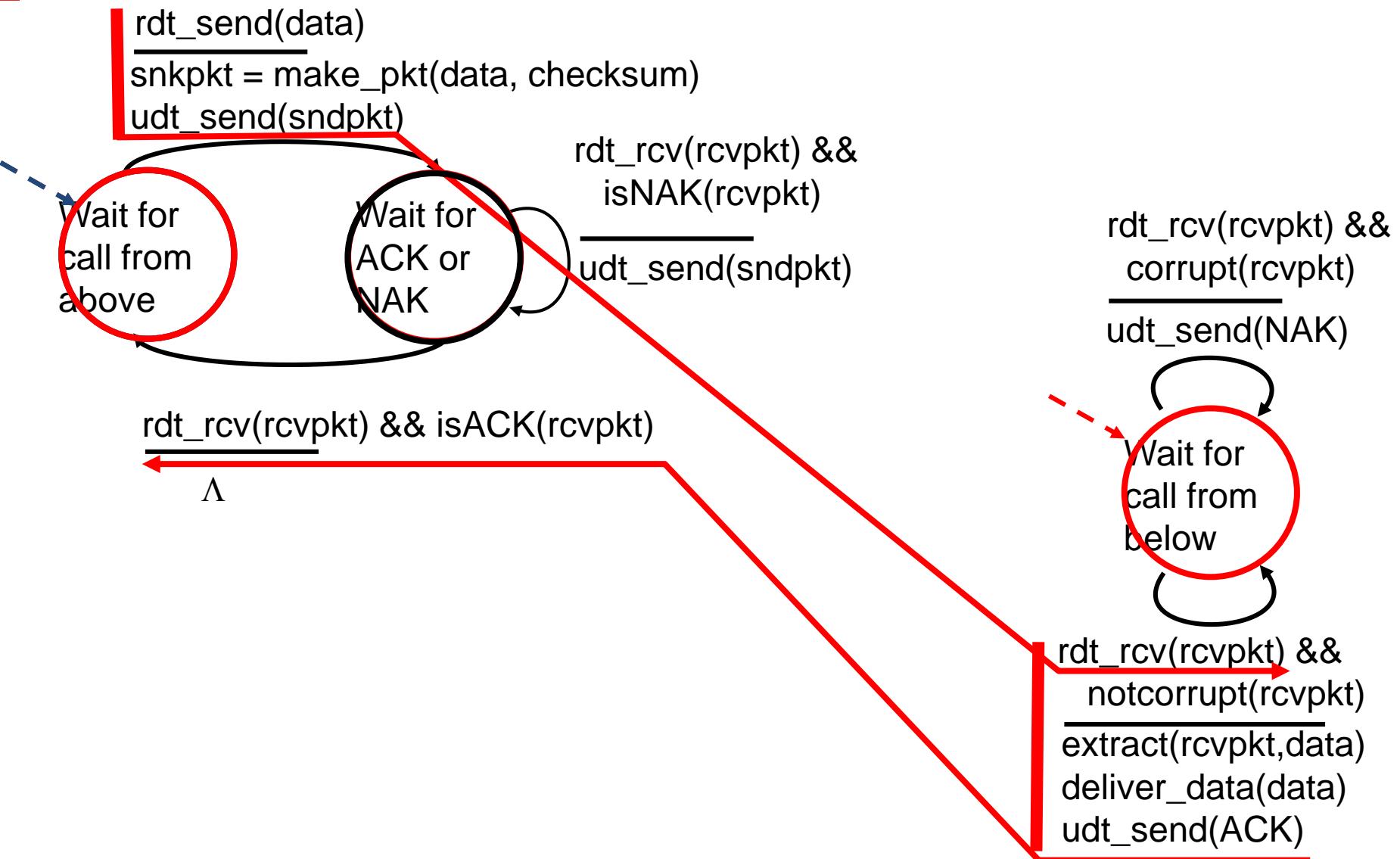
# rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the question:* how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

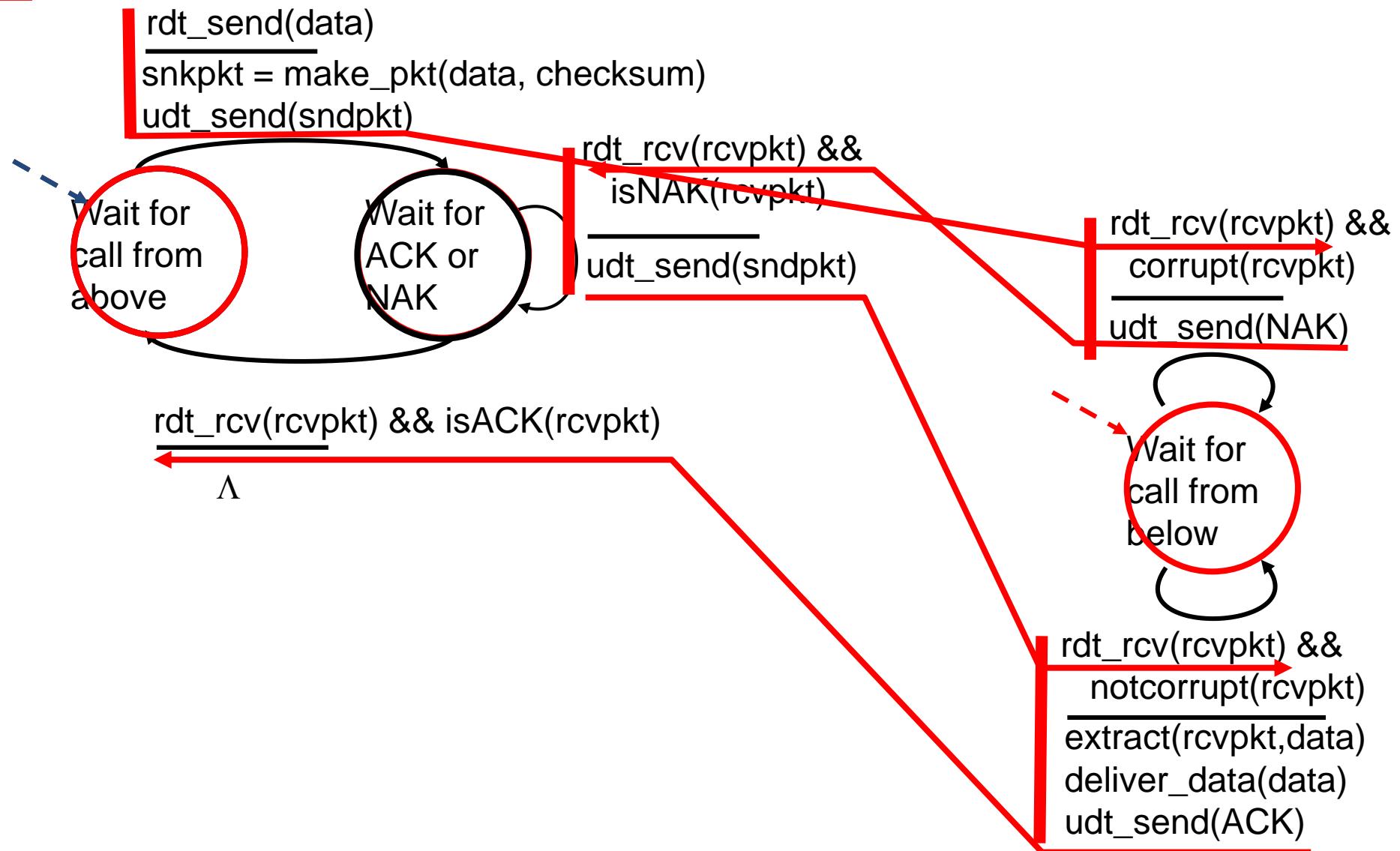
# rdt2.0: FSM specification



# rdt2.0: operation with no errors



# rdt2.0: error scenario



# rdt2.0 has a fatal flaw!

## what happens if ACK/NAK corrupted?

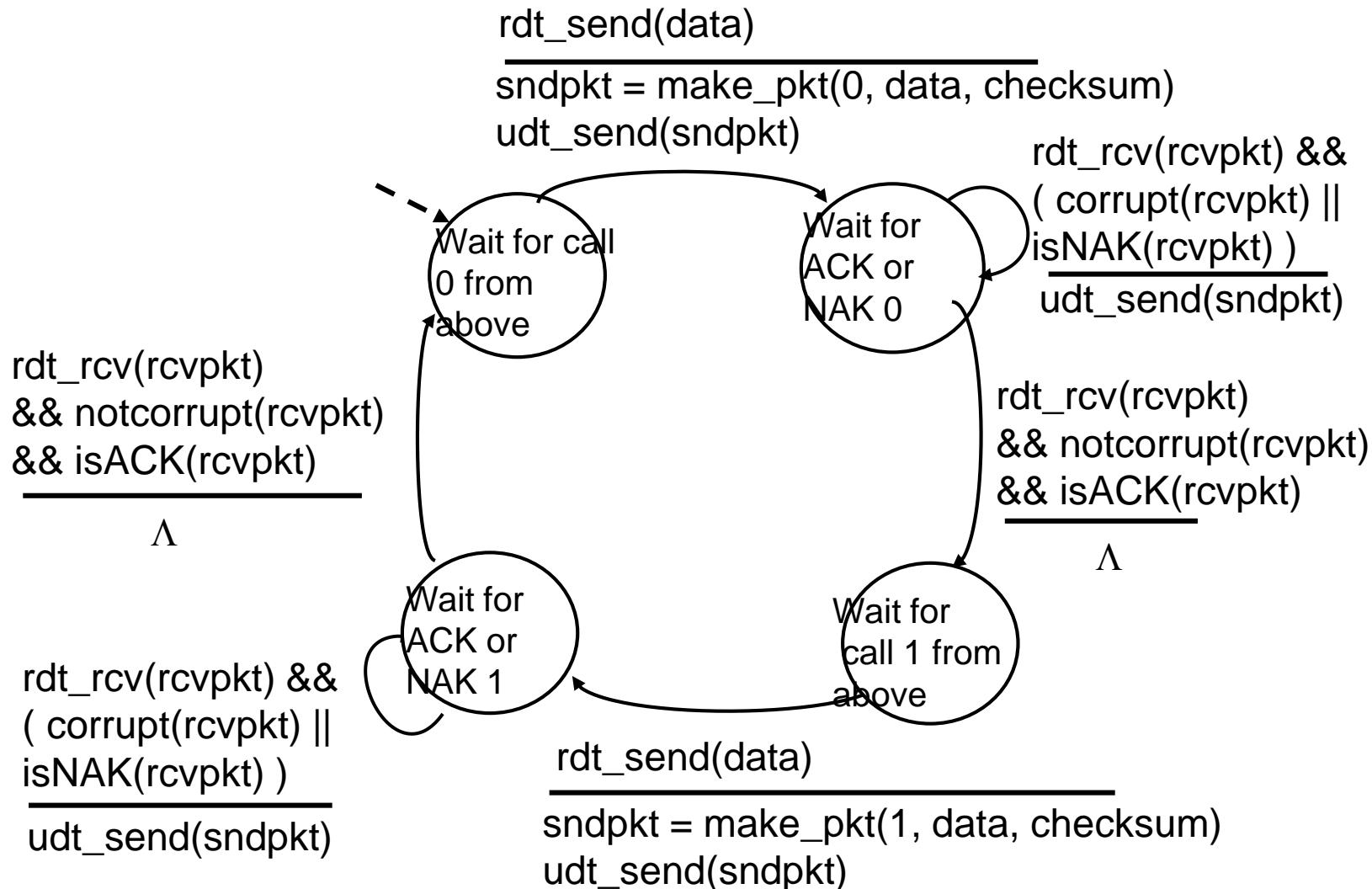
- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

## 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

# rdt2.1: sender, handles garbled ACK/NAKs



# rdt2.1: receiver, handles garbled ACK/NAKs

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) && notcorrupt(rcvpkt)  
    && has\_seq0(rcvpkt)

    extract(rcvpkt,data)  
    deliver\_data(data)  
    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)



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)

# rdt2.1: discussion

## 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

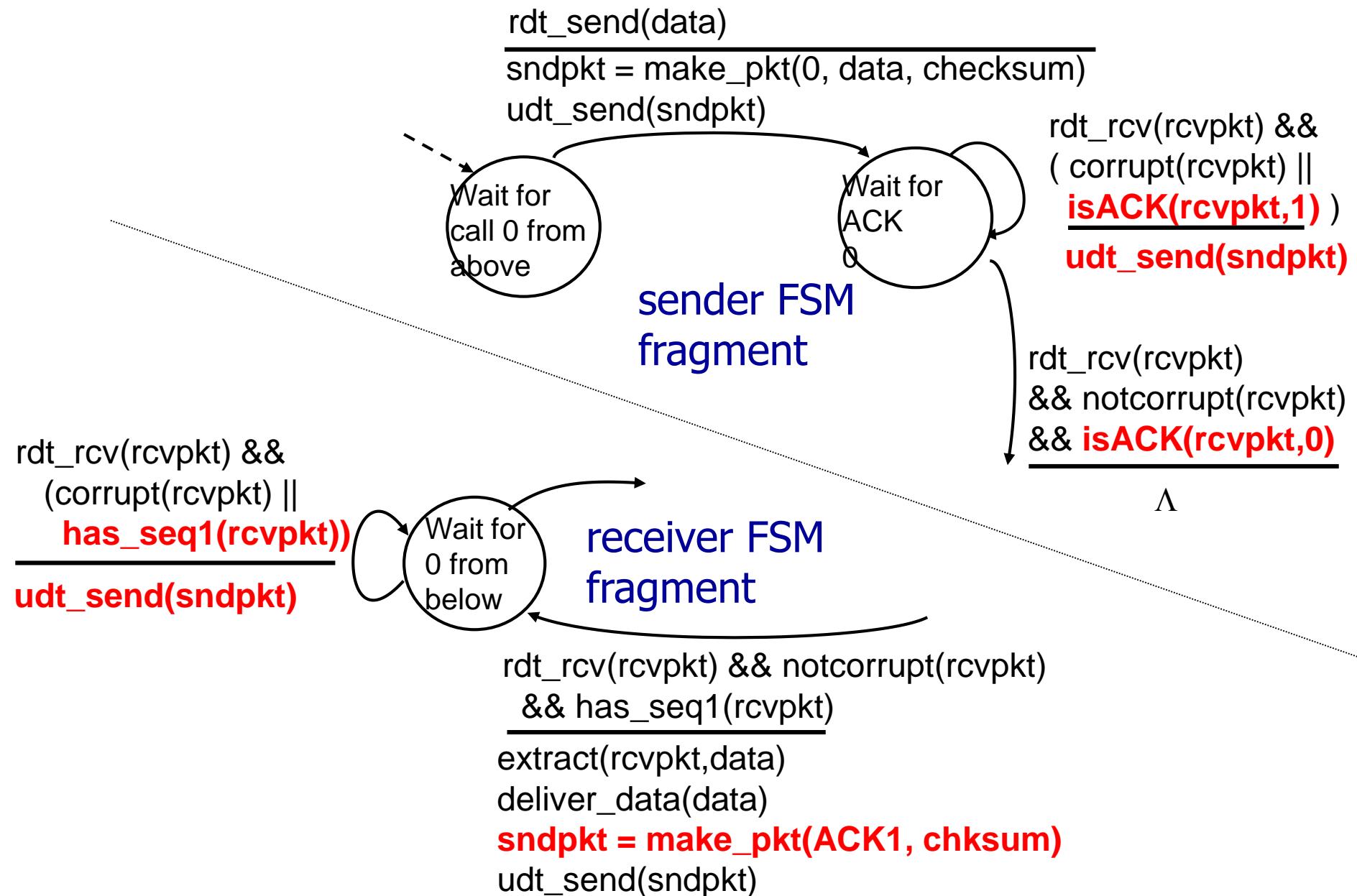
## receiver:

- 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

# rdt2.2: a NAK-free protocol

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

# rdt2.2: sender, receiver fragments



# rdt3.0: channels with errors *and* loss

## new assumption:

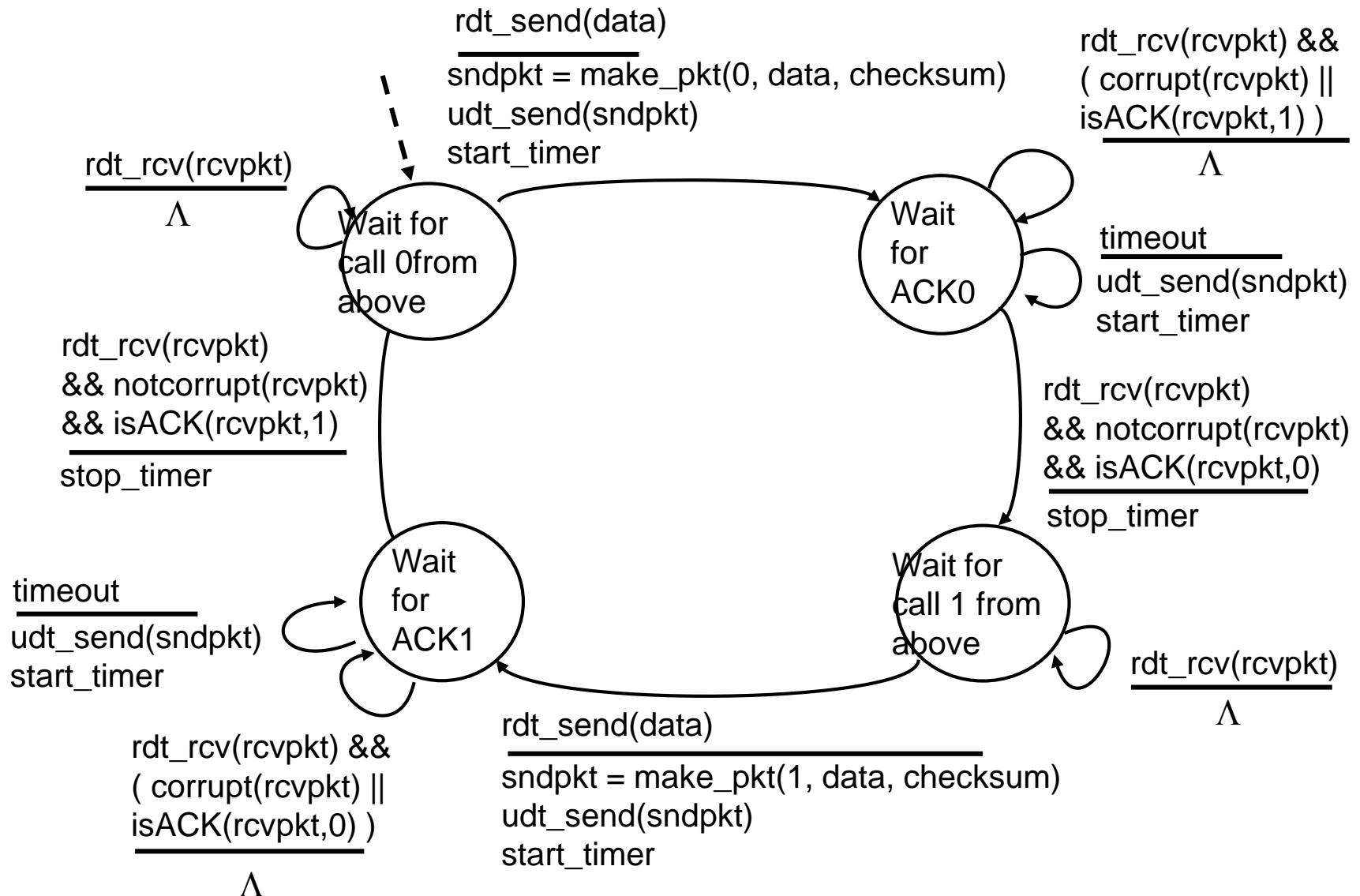
underlying channel  
can also lose packets  
(data, ACKs)

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

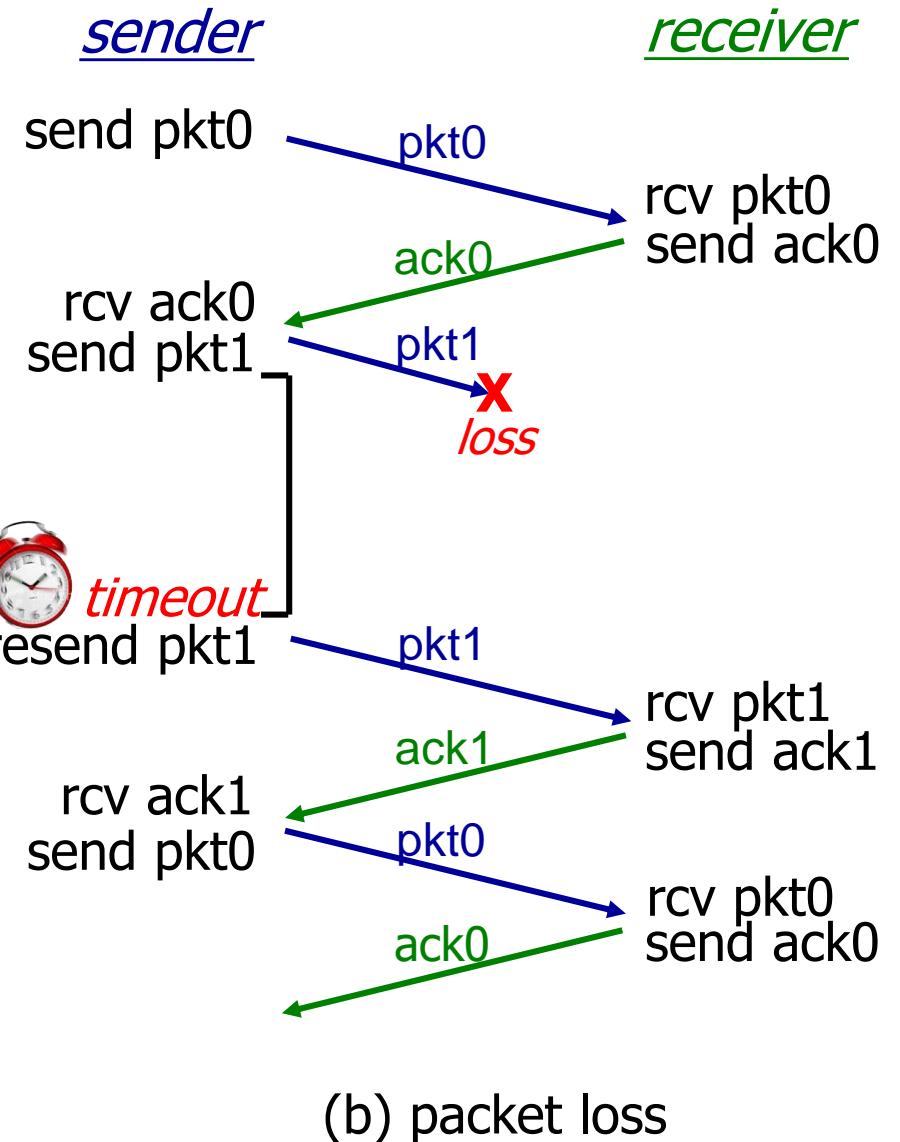
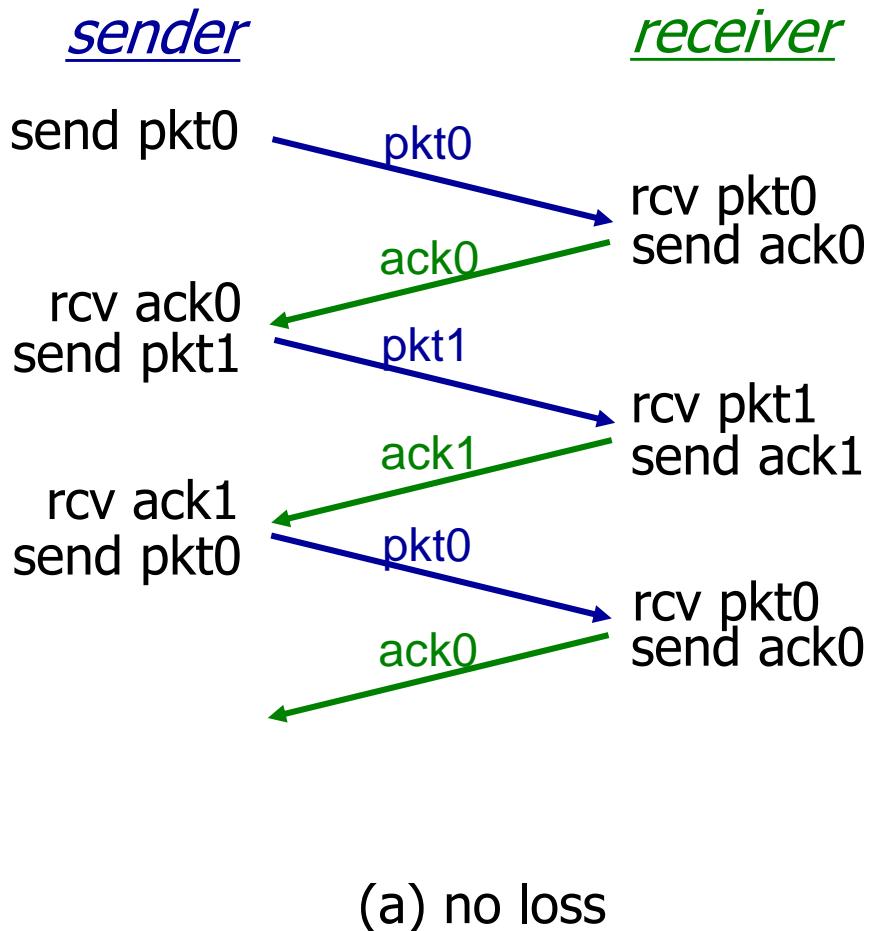
## approach: sender waits “reasonable” amount of time for ACK

- 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

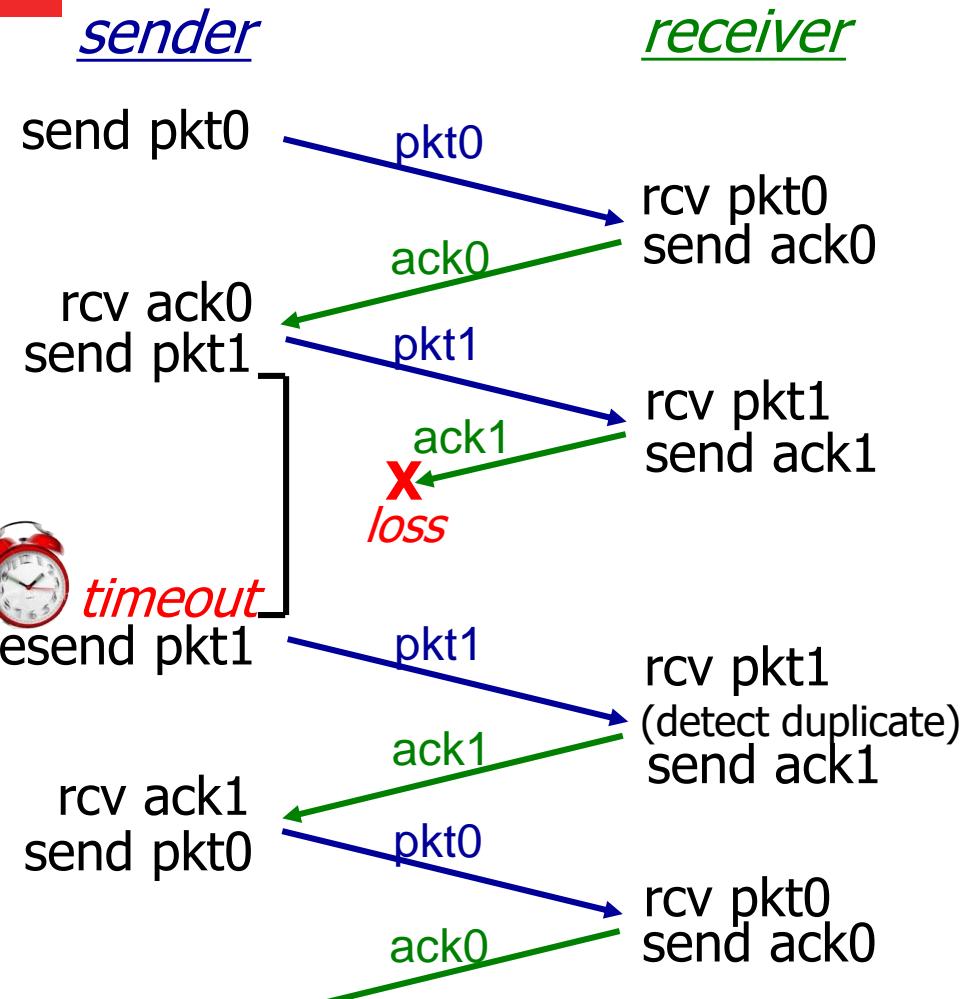


# rdt3.0 in action



# rdt3.0 in action

sender



(c) ACK loss

sender

send pkt0

rcv ack0  
send pkt1

timeout  
resend pkt1

rcv ack1  
send pkt0

rcv ack1  
send pkt0

rcv ack1  
send pkt0

rcv ack0  
send pkt0

rcv ack0  
send pkt0

pkt0

ack0

pkt1

ack1

pkt1

pkt0

ack1

ack0

pkt0

ack0

receiver

rcv pkt0  
send ack0

rcv pkt1  
send ack1

rcv pkt1  
(detect duplicate)  
send ack1

rcv pkt0  
send ack0

rcv pkt0  
(detect duplicate)  
send ack0



timeout

resend pkt1

rcv ack1  
send pkt0

rcv ack1  
send pkt0

rcv ack1  
send pkt0

rcv ack0  
send pkt0

rcv ack0  
send pkt0

# Performance of rdt3.0

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

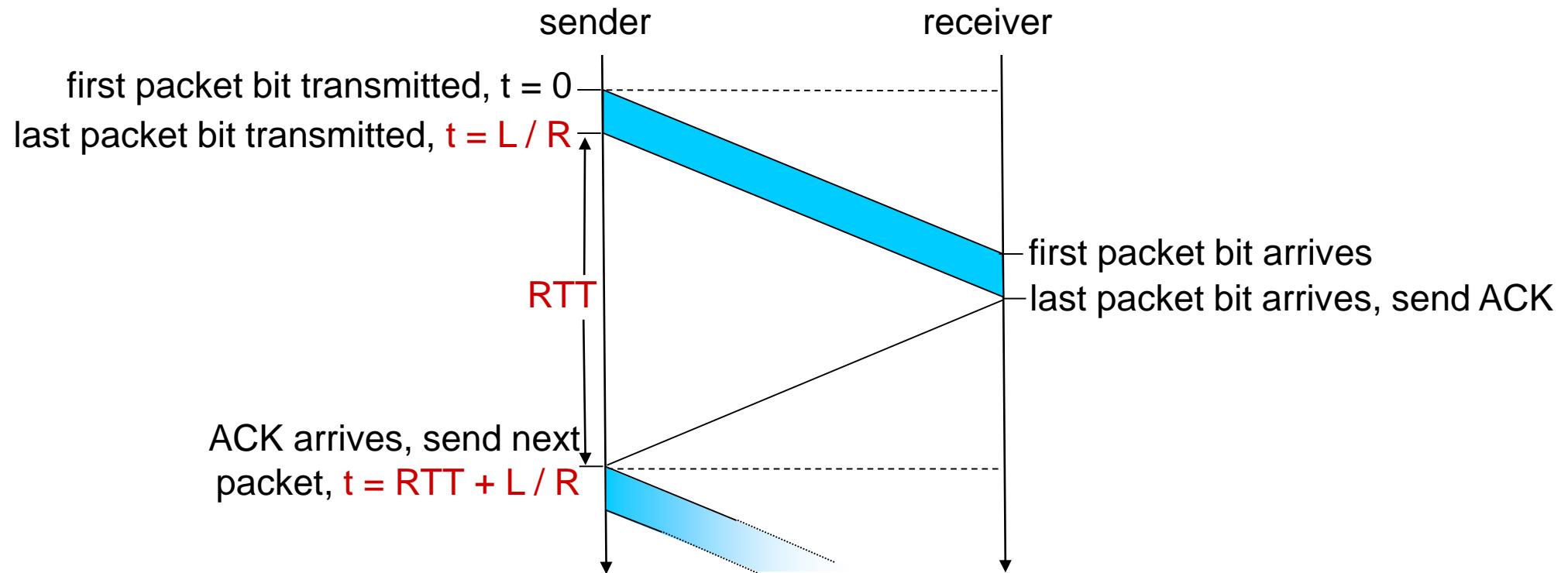
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

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

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

- 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

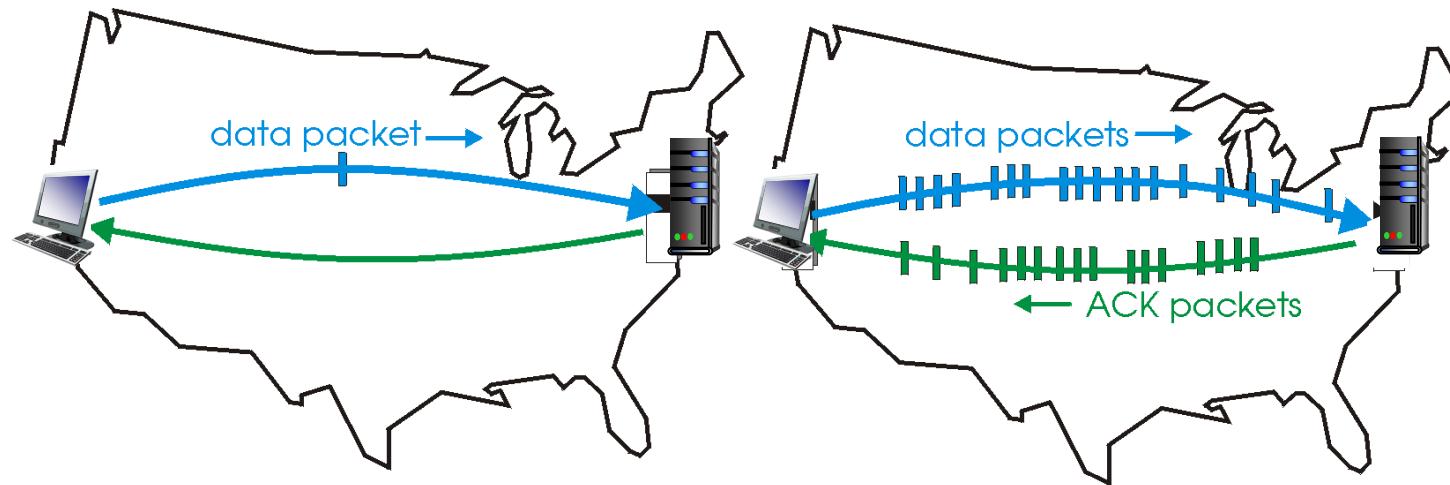


$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# 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

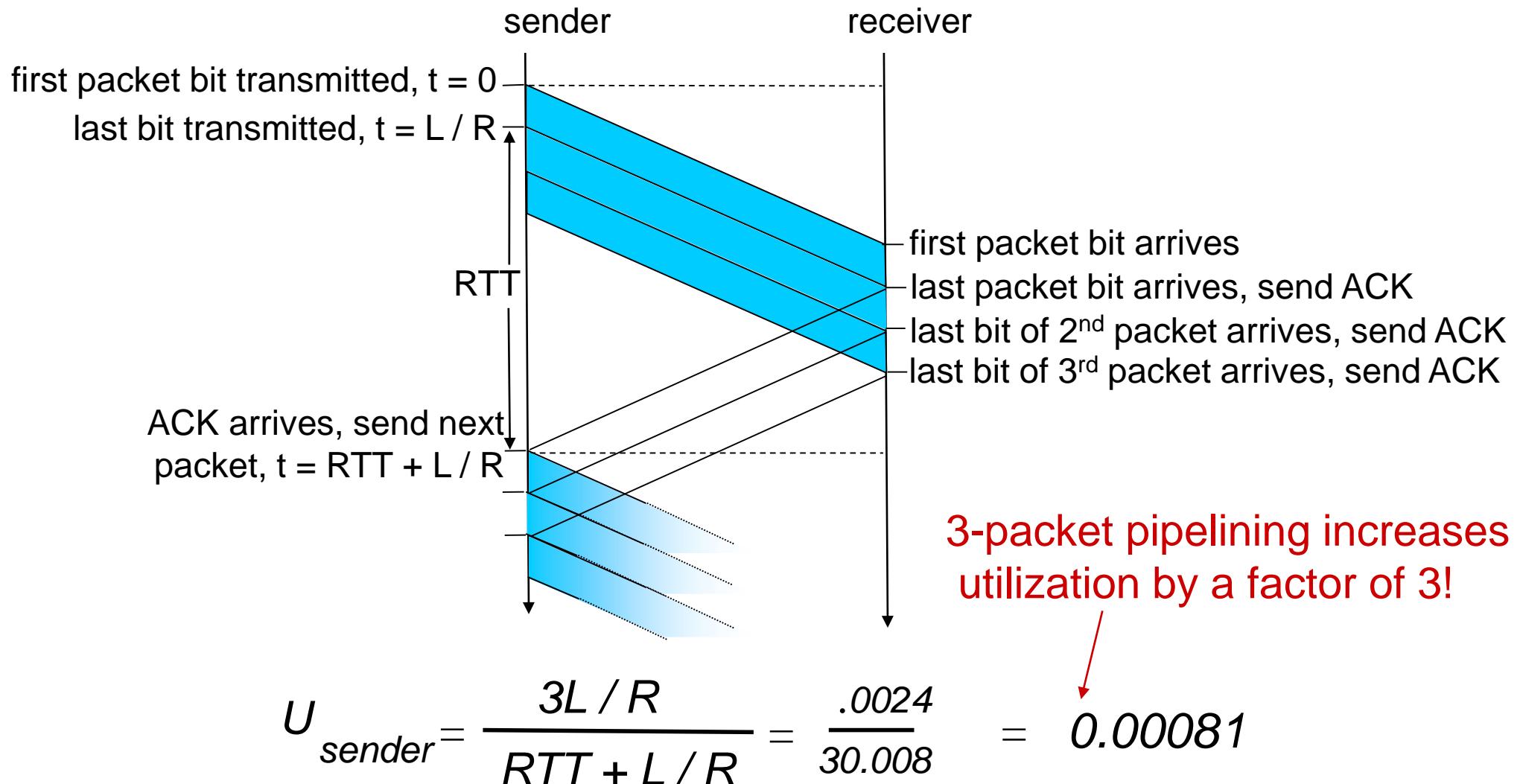


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

# Pipelining: increased utilization



# Pipelined protocols: overview

## Go-back-N:

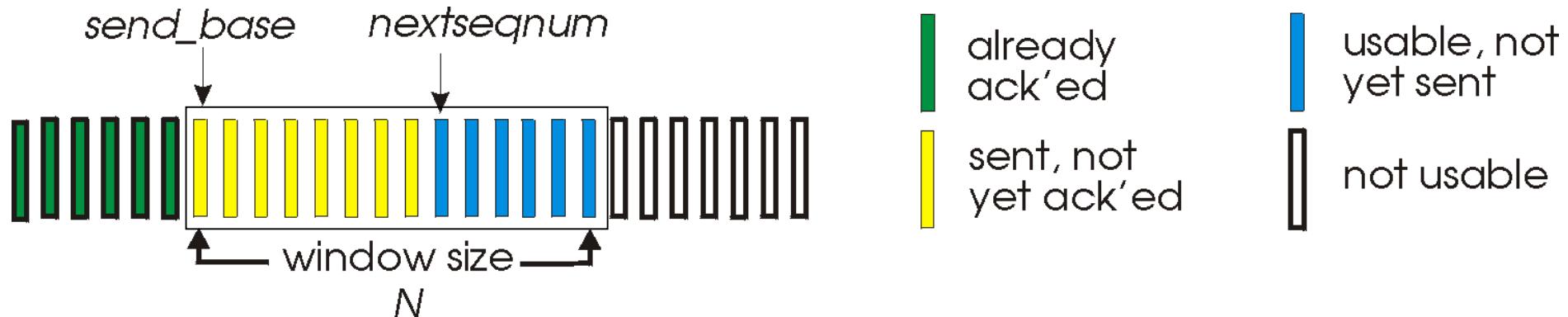
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

## Selective Repeat:

- sender can have up to N unack'd 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

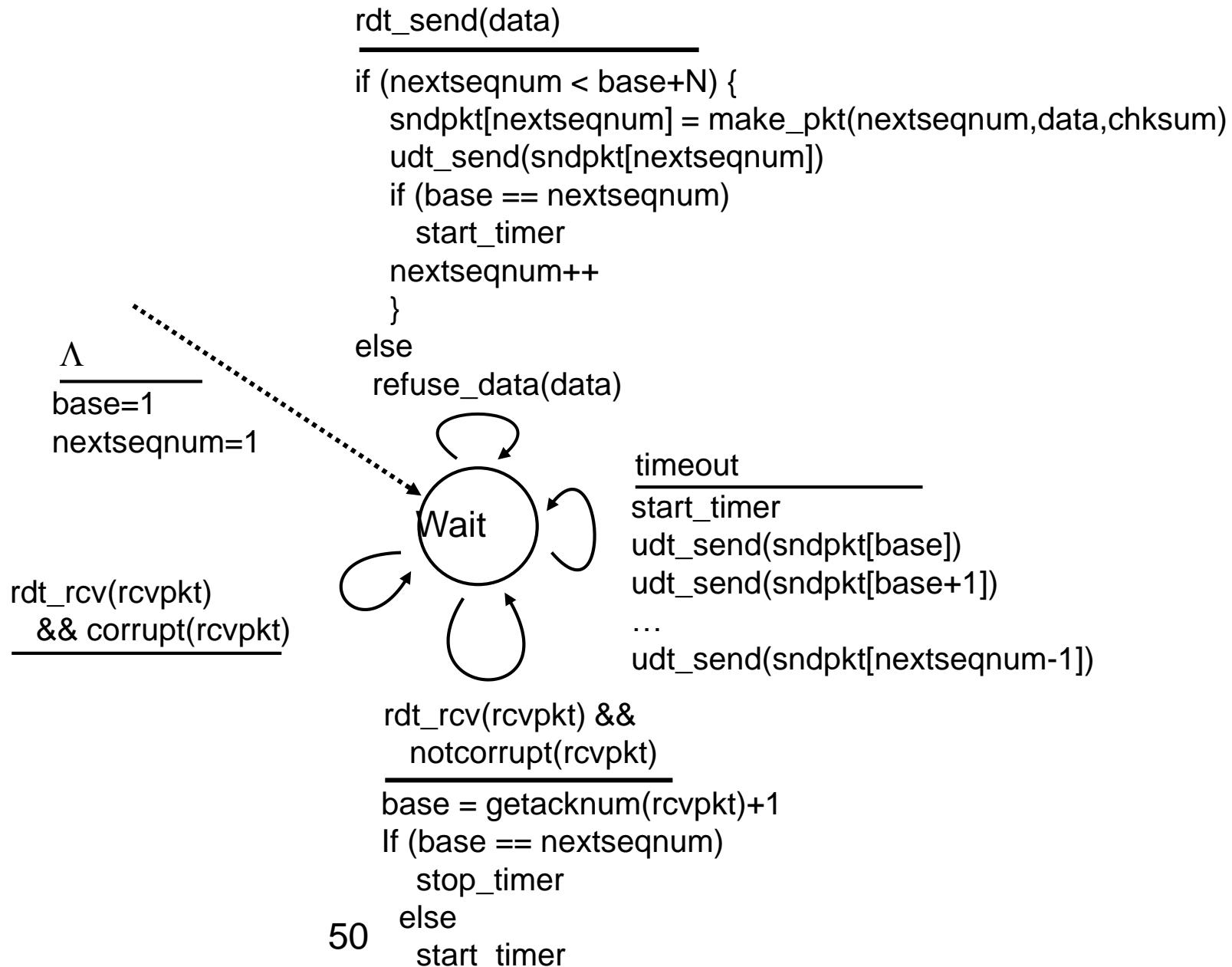
# Go-Back-N: sender

- k-bit seq # in pkt header
  - “window” of up to N, consecutive unack’ed pkts allowed

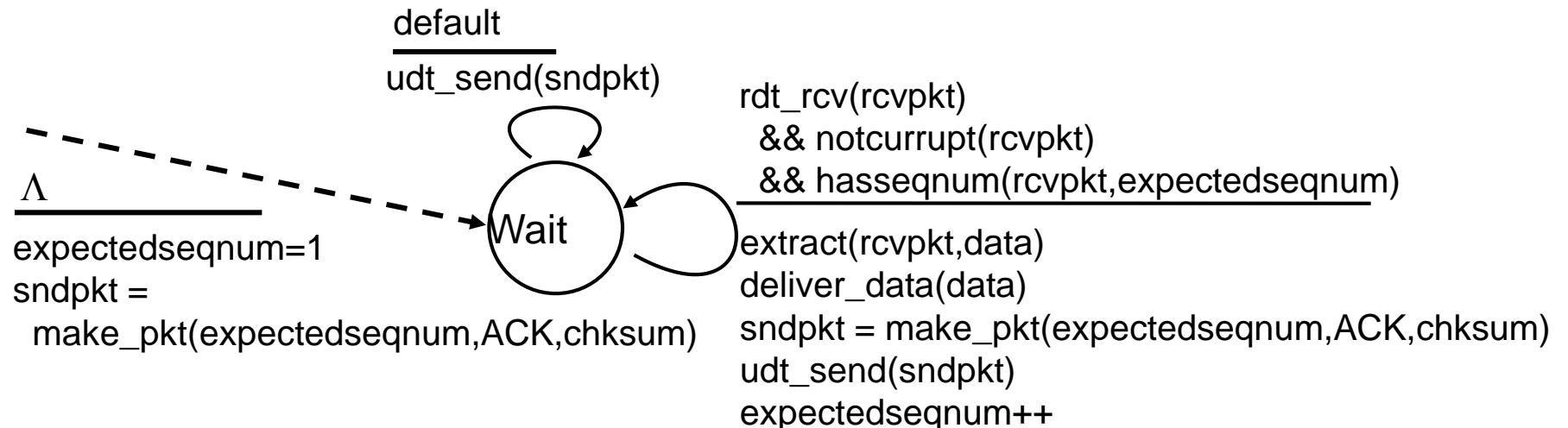


- ACK( $n$ ): ACKs all pkts up to, including seq #  $n$  - “*cumulative ACK*”
    - may receive duplicate ACKs (see receiver)
  - timer for oldest in-flight pkt
  - $\text{timeout}(n)$ : retransmit packet  $n$  and all higher seq # pkts in window

# GBN: sender extended FSM



# 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

sender

send pkt0  
send pkt1  
send pkt2  
send pkt3  
(wait)

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

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

rcv ack0, send pkt4  
rcv ack1, send pkt5

ignore duplicate ACK



*pkt 2 timeout*

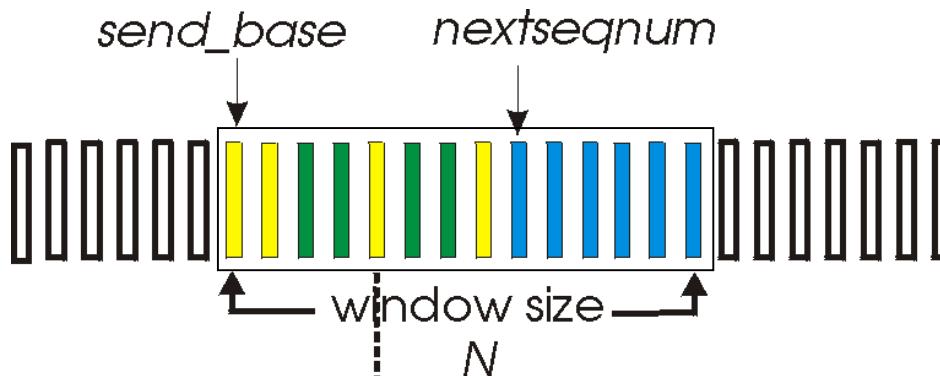
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

52

# Selective repeat

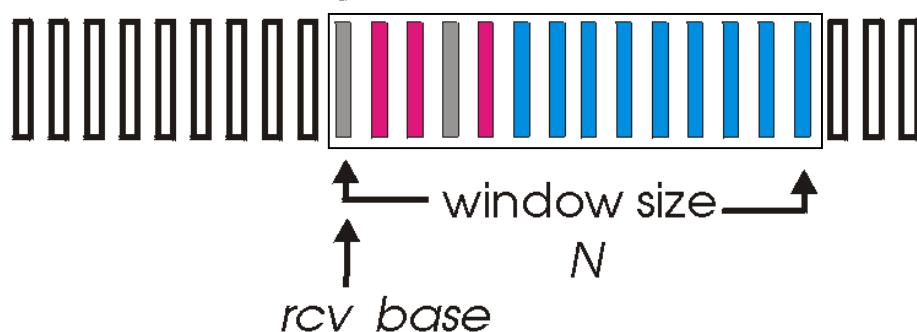
- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - $N$  consecutive seq #'s
  - limits seq #'s of sent, unACKed pkts

# Selective repeat: sender, receiver windows



already ack'ed  
sent, not yet ack'ed  
usable, not yet sent  
not usable

(a) sender view of sequence numbers



out of order (buffered) but already ack'ed  
Expected, not yet received  
acceptable (within window)  
not usable

(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 ack5 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?*

# Selective repeat: dilemma

example:

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

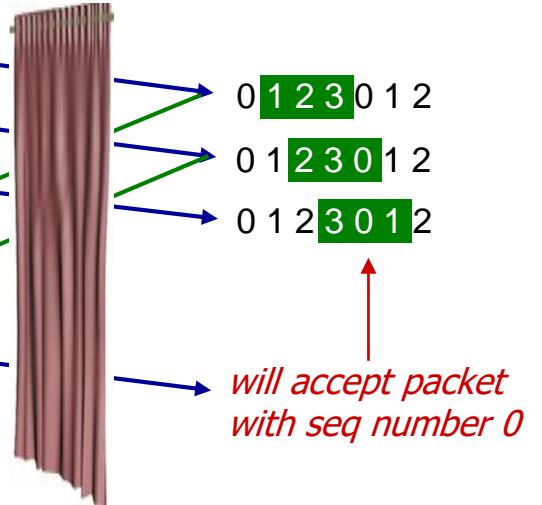
Q: what relationship between seq # size and window size to avoid problem in (b)?

sender window  
(after receipt)

0 1 2 3 0 1 2	pkt0
0 1 2 3 0 1 2	pkt1
0 1 2 3 0 1 2	pkt2
0 1 2 3 0 1 2	pkt3
0 1 2 3 0 1 2	

(a) no problem

receiver window  
(after receipt)



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

0 1 2 3 0 1 2	pkt0
0 1 2 3 0 1 2	pkt1
0 1 2 3 0 1 2	pkt2
	X
	X
	X
0 1 2 3 0 1 2	pkt0

(b) oops!

*will accept packet with seq number 0*

# Class Recap

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

# Next Class

- Connection-oriented transport protocol
  - TCP
  - Flow control
  - Congestion control