Transport layer

Our goals:

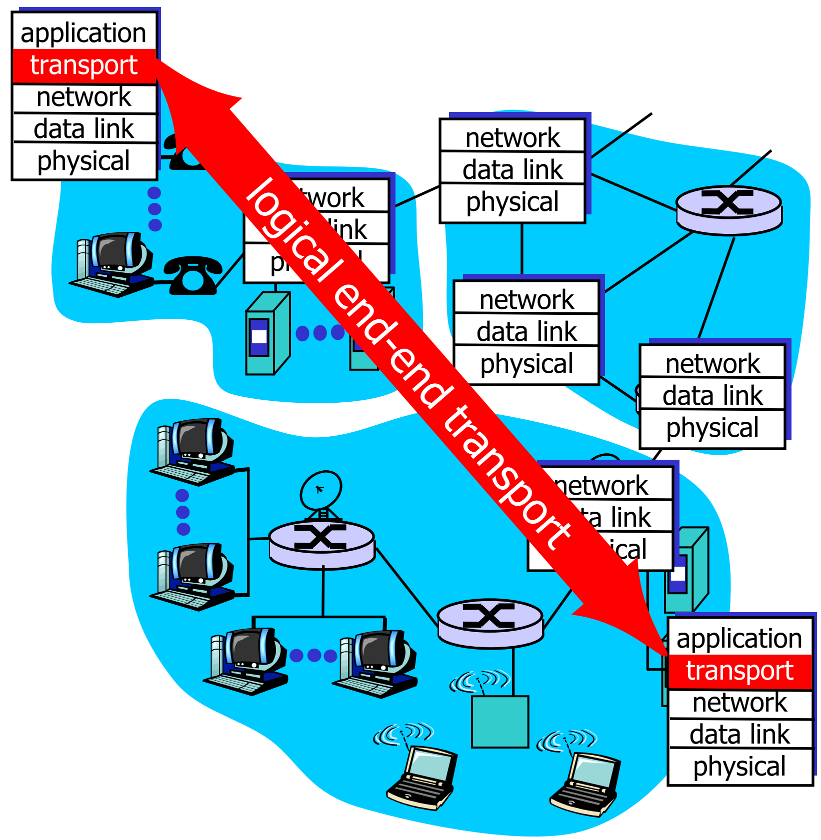
- Understand principles behind transport layer services:
  - Multiplexing/demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Transport layer: Outline

- 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

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

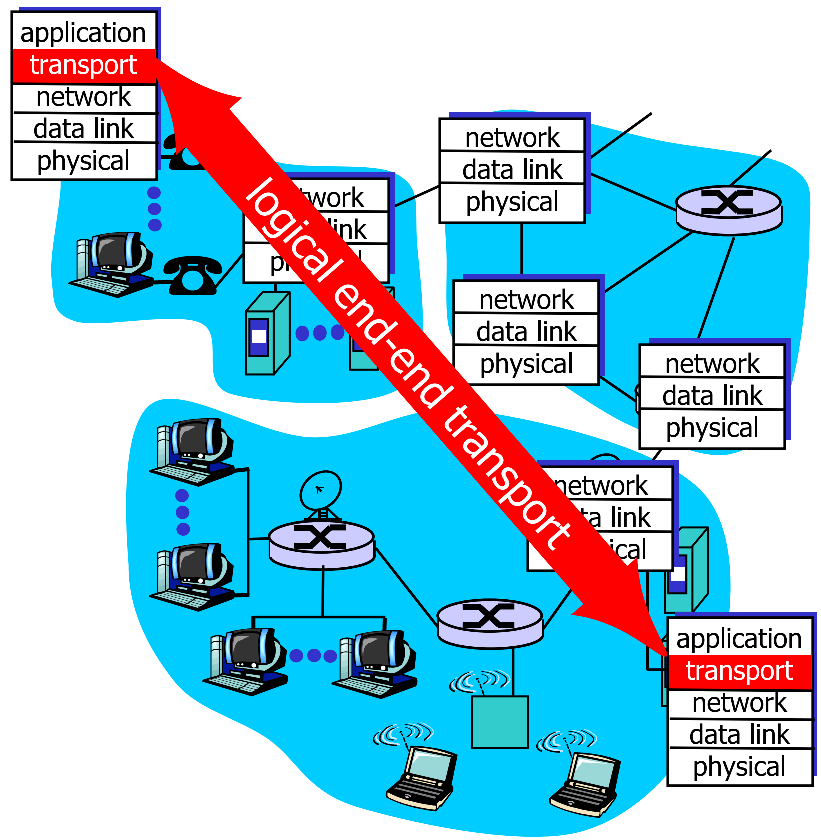
Household analogy:

12 kids sending letters to 12 kids

- Processes = kids
- App messages = letters in envelopes
- Hosts = houses
- Transport protocol = Ann and Bill
- 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



Transport layer: Outline

- 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

Multiplexing/demultiplexing

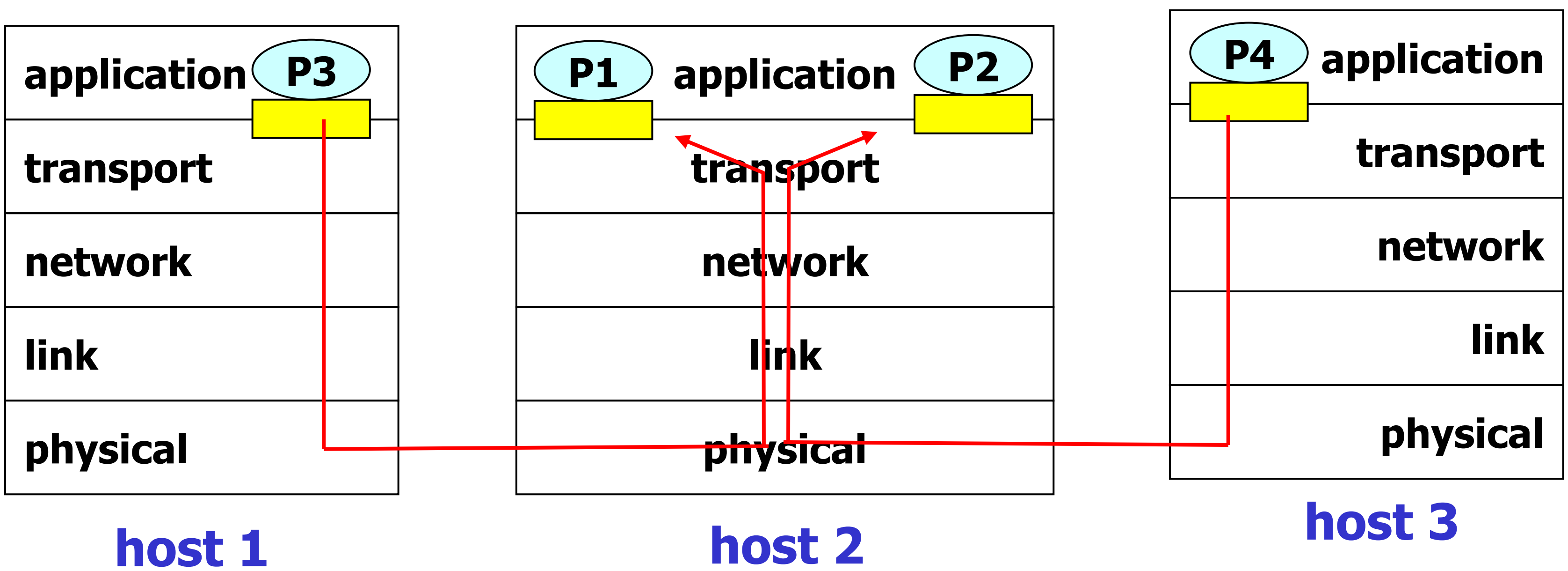
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket = process



Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

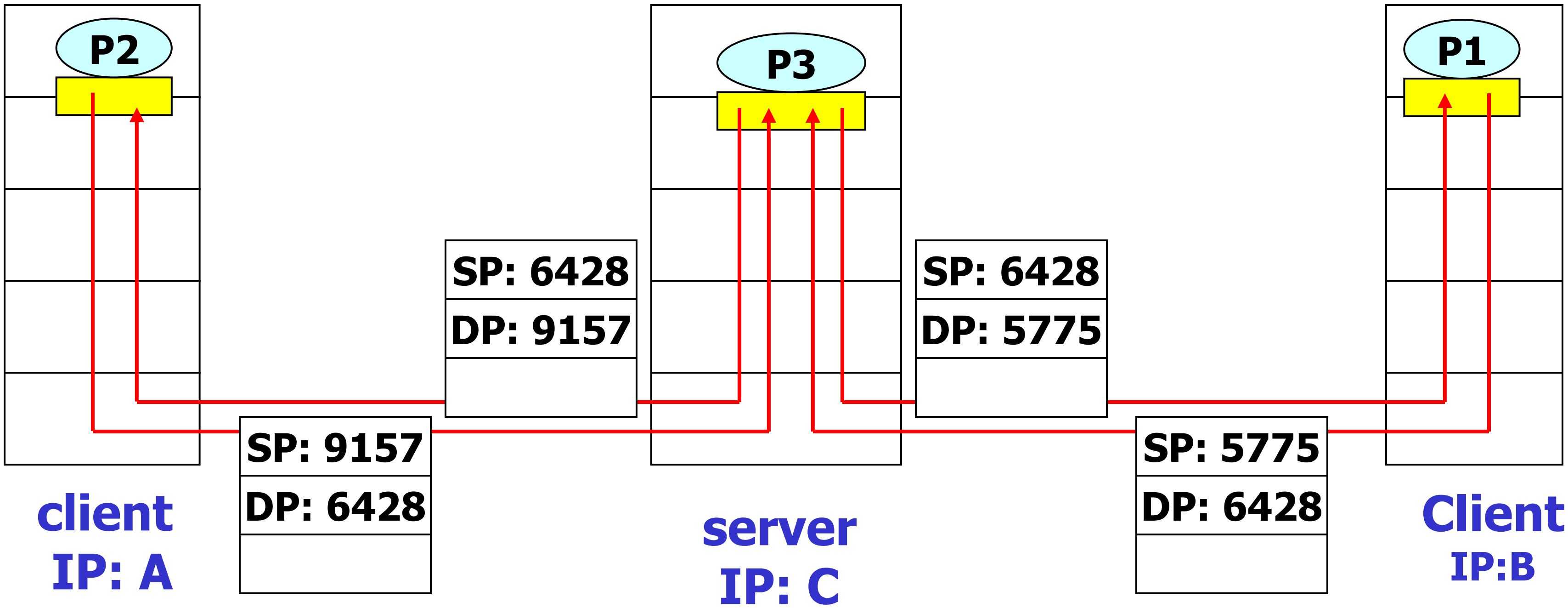
- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - Checks destination port number in segment
  - Directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont.)

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

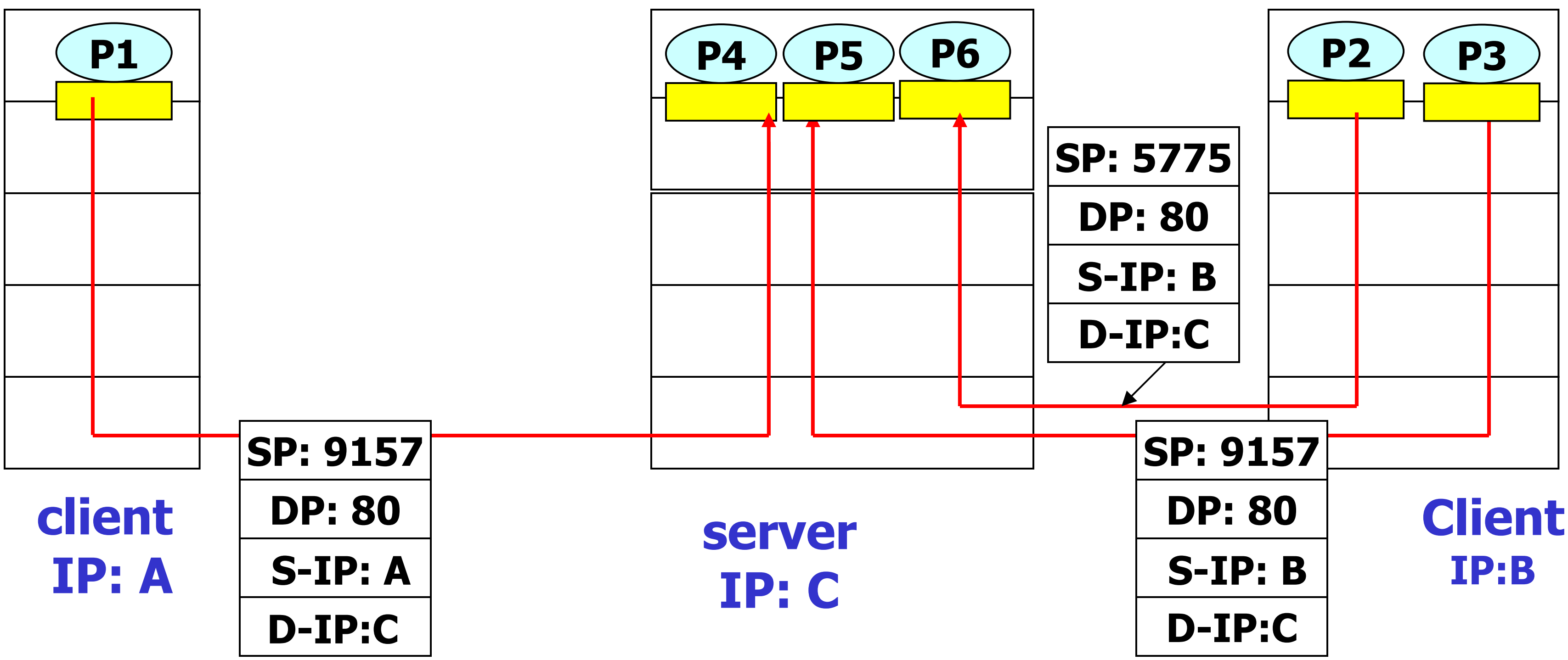


SP provides "return address"

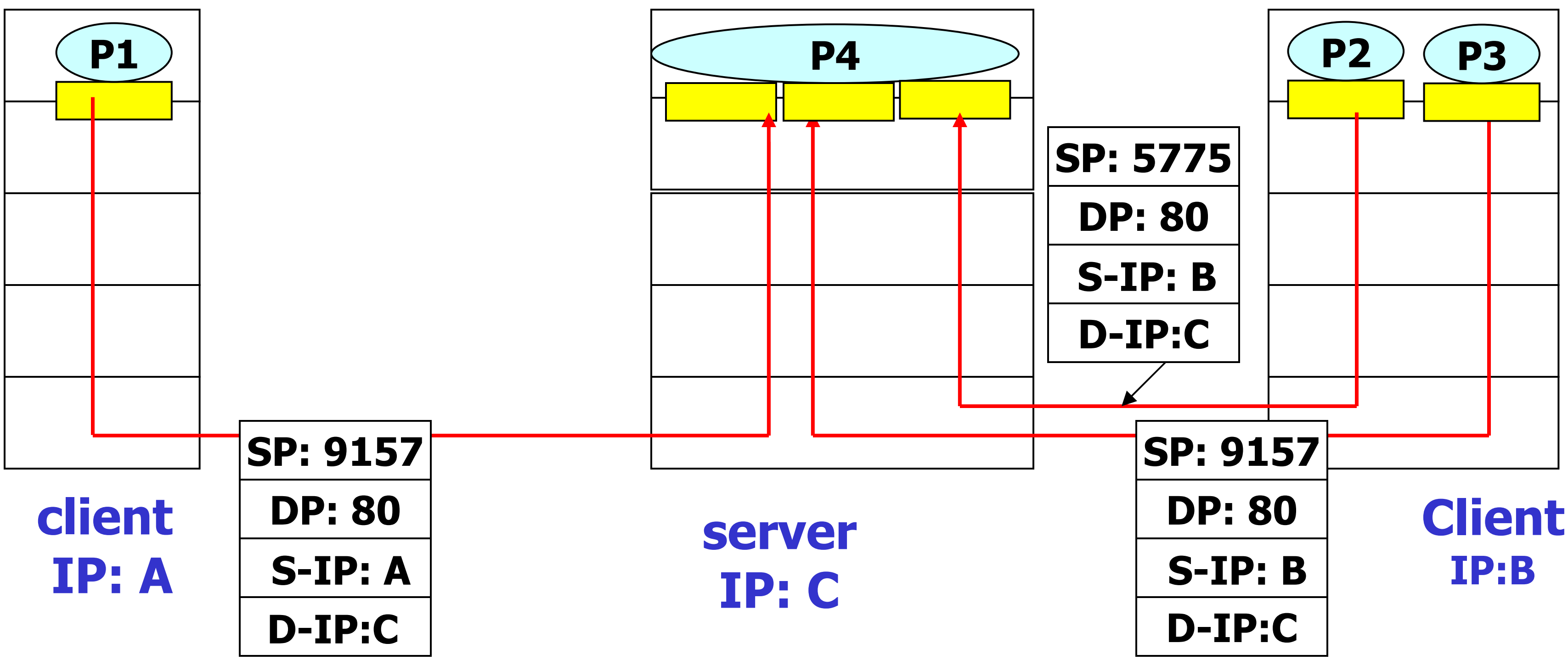
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Recv host 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 (cont.)



Connection-oriented demux: Threaded Web Server



Transport layer: Outline

- 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

UDP: User Datagram Protocol [RFC 768]

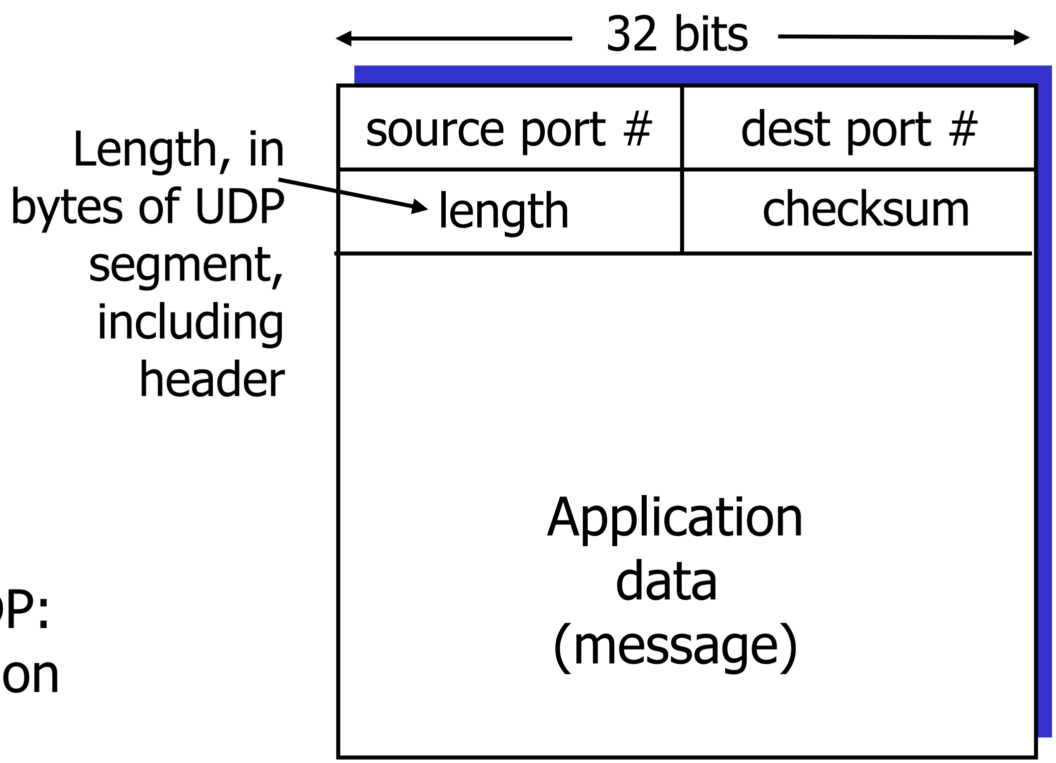
- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost
  - Delivered out of order to app
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

- No connection establishment (which can add delay)
- Simple: no connection state at sender, receiver
- Small segment header
- No congestion control: UDP can blast away as fast as desired

UDP: More

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses (why?):
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recover!



UDP segment format

UDP checksum

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

Sender:

- Treat segment contents as sequence of 16-bit integers
- Checksum: addition (1'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 nonethleess?* More later ….

Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

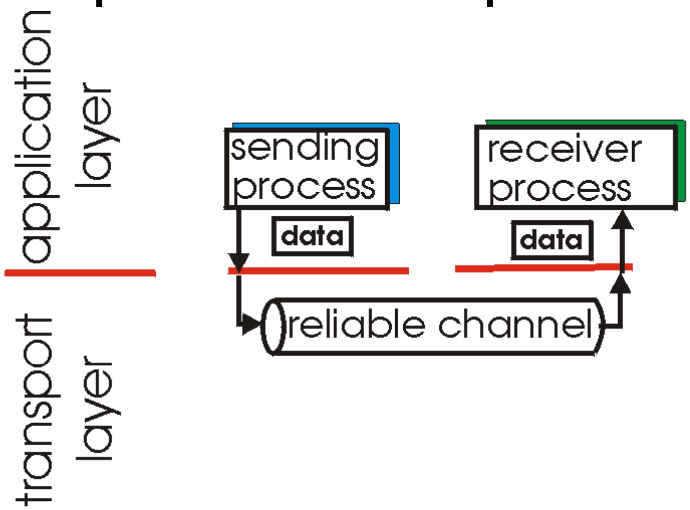
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Transport layer: Outline

- 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

Principles of reliable data transfer

- Important in app., transport, link layers
- Top-10 list of important networking topics!

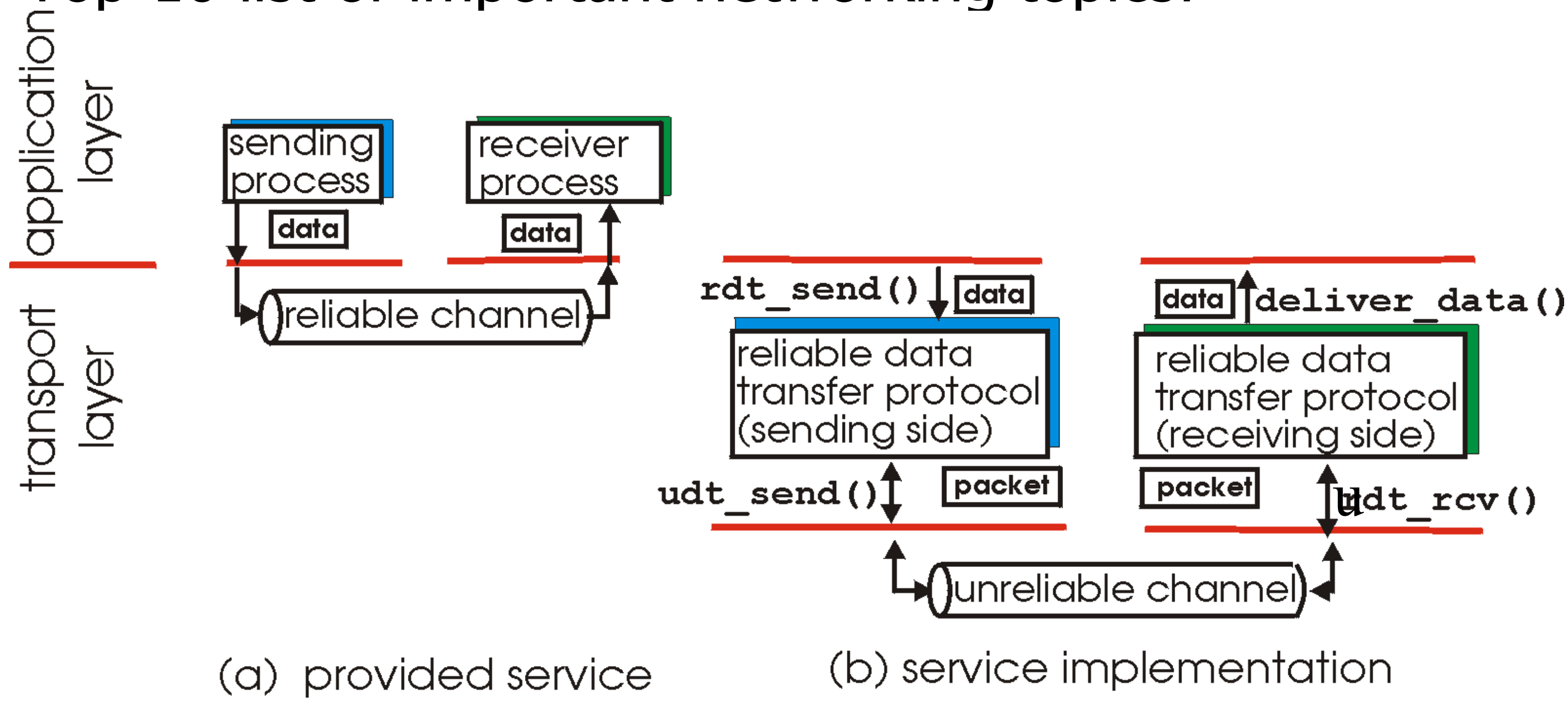


(a) provided service

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

Principles of reliable data transfer

- Important in app., transport, link layers
- Top-10 list of important networking topics!

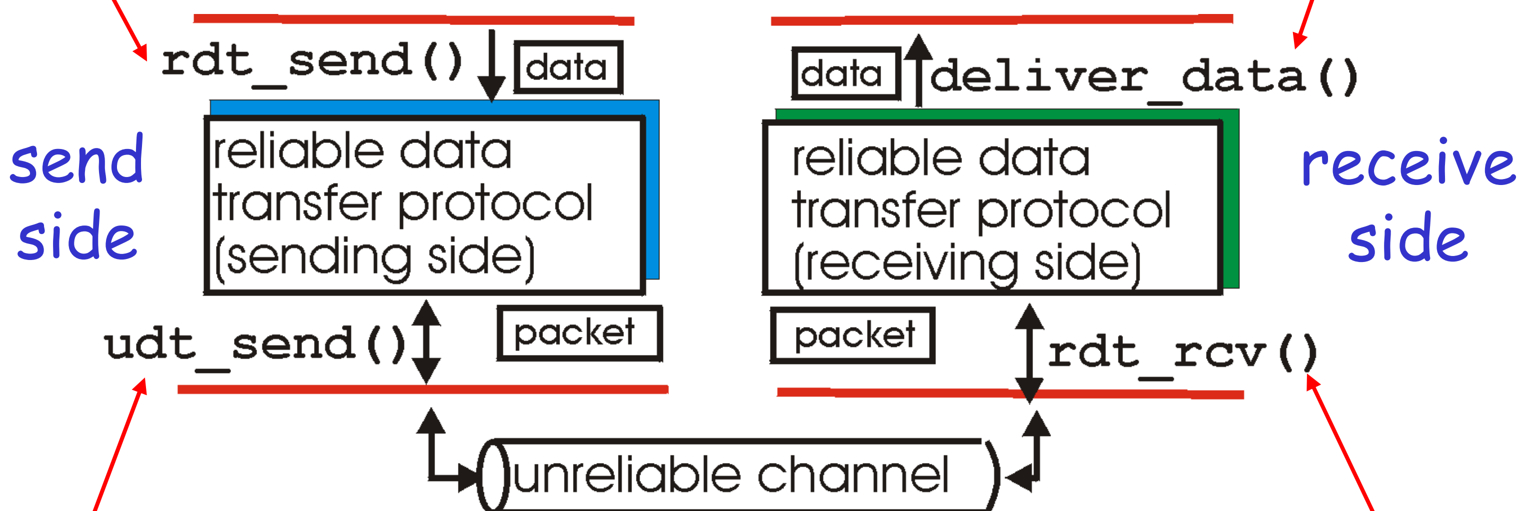


- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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

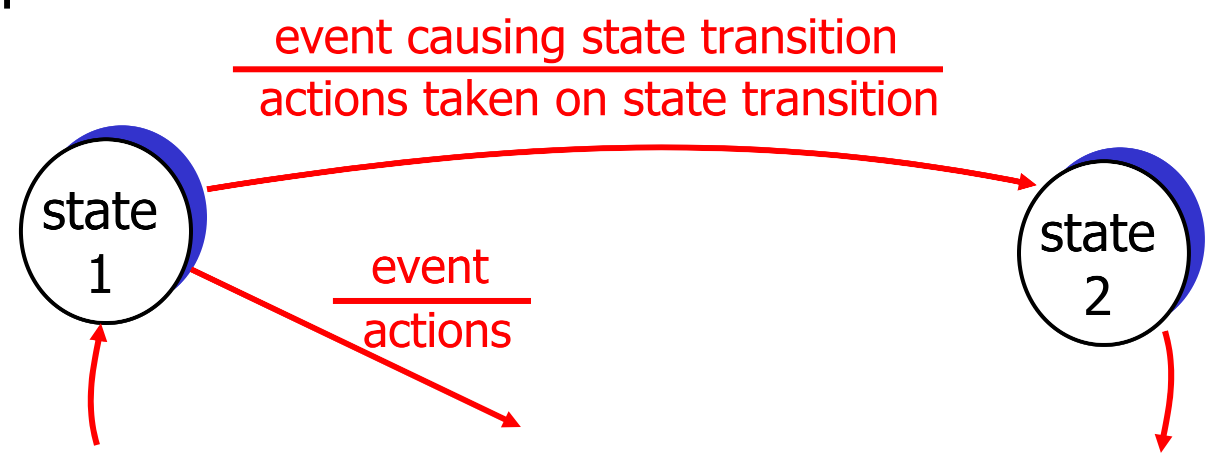
rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: Getting started

We'll:

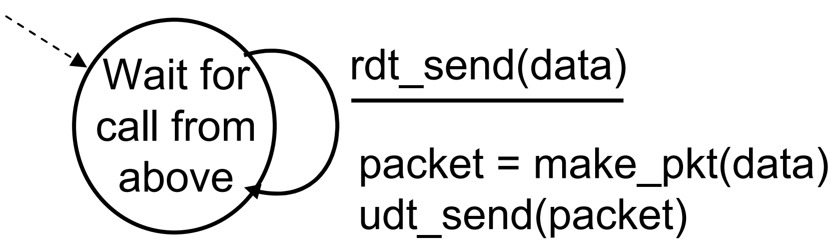
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - But control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver

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

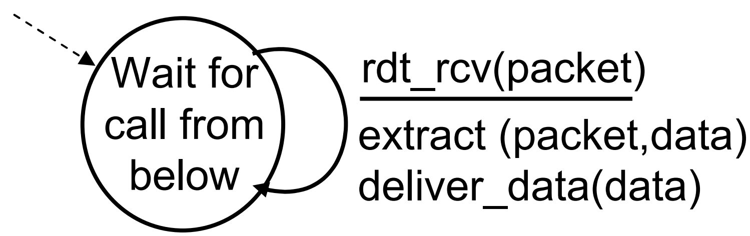


Rdt1.0: Reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - No bit erros
  - No loss of packets
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver read data from underlying channel



sender



receiver

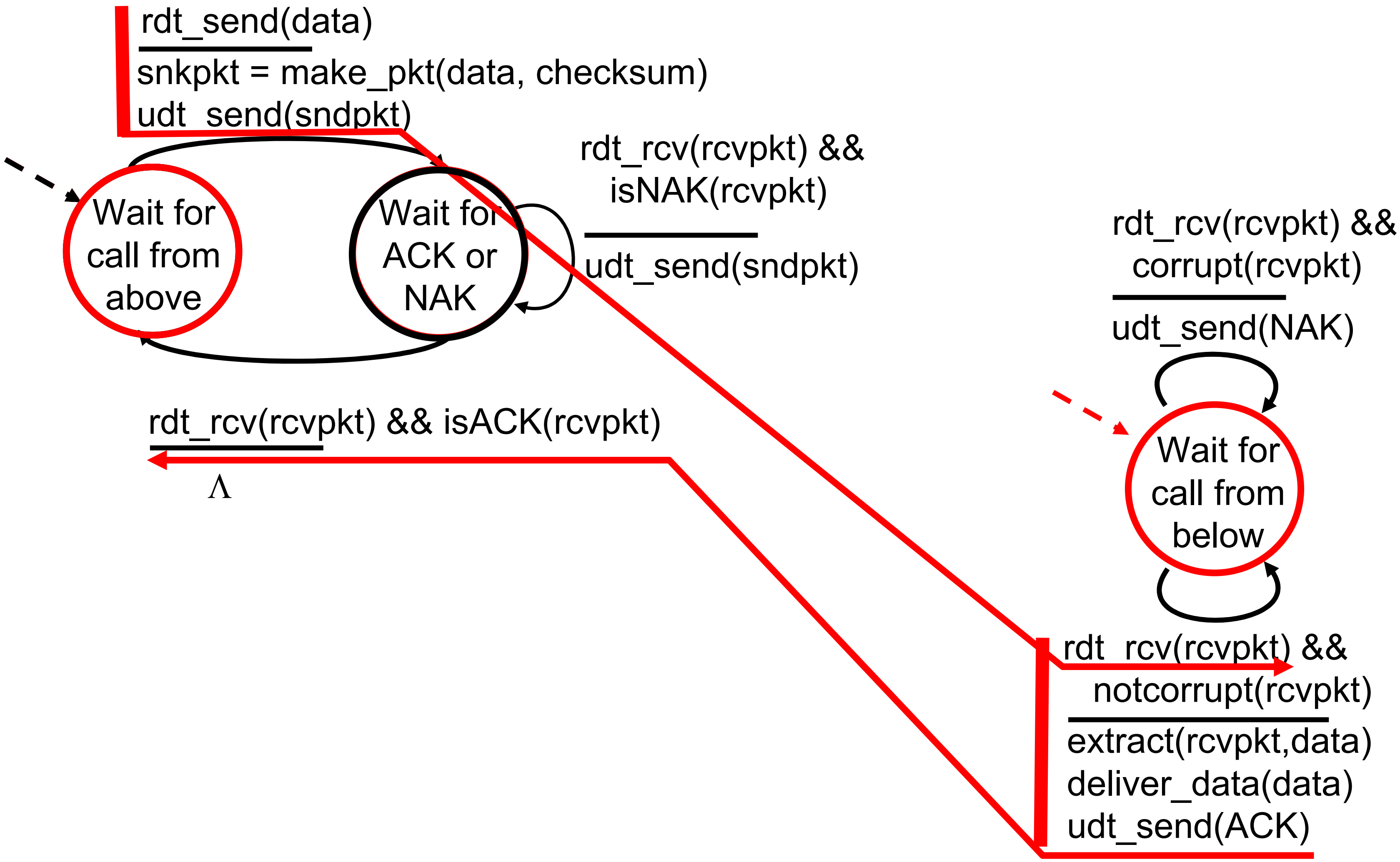
Rdt2.0: Channel with bit errors

- Underlying channel may flip bits in packet
  - Recall: UDP 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
  - Human scenarios using ACKs, NAKs?

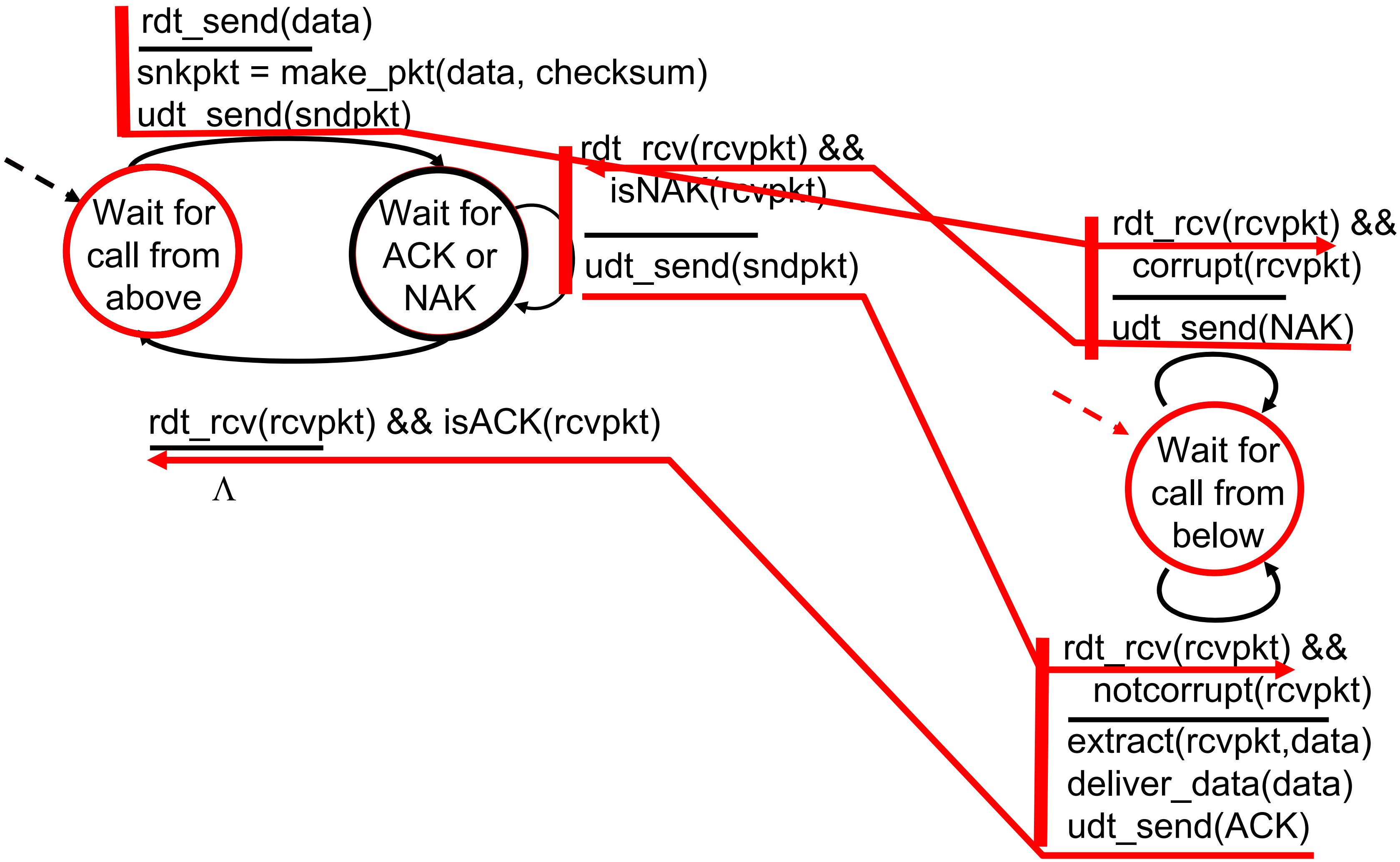
New mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):

  - Error detection
  - Receiver feedback: control msgs (ACK,NAK) rcvr->sender

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

What to do?

- Sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- Retransmit, but this might cause retransmission of correctly received pkt!

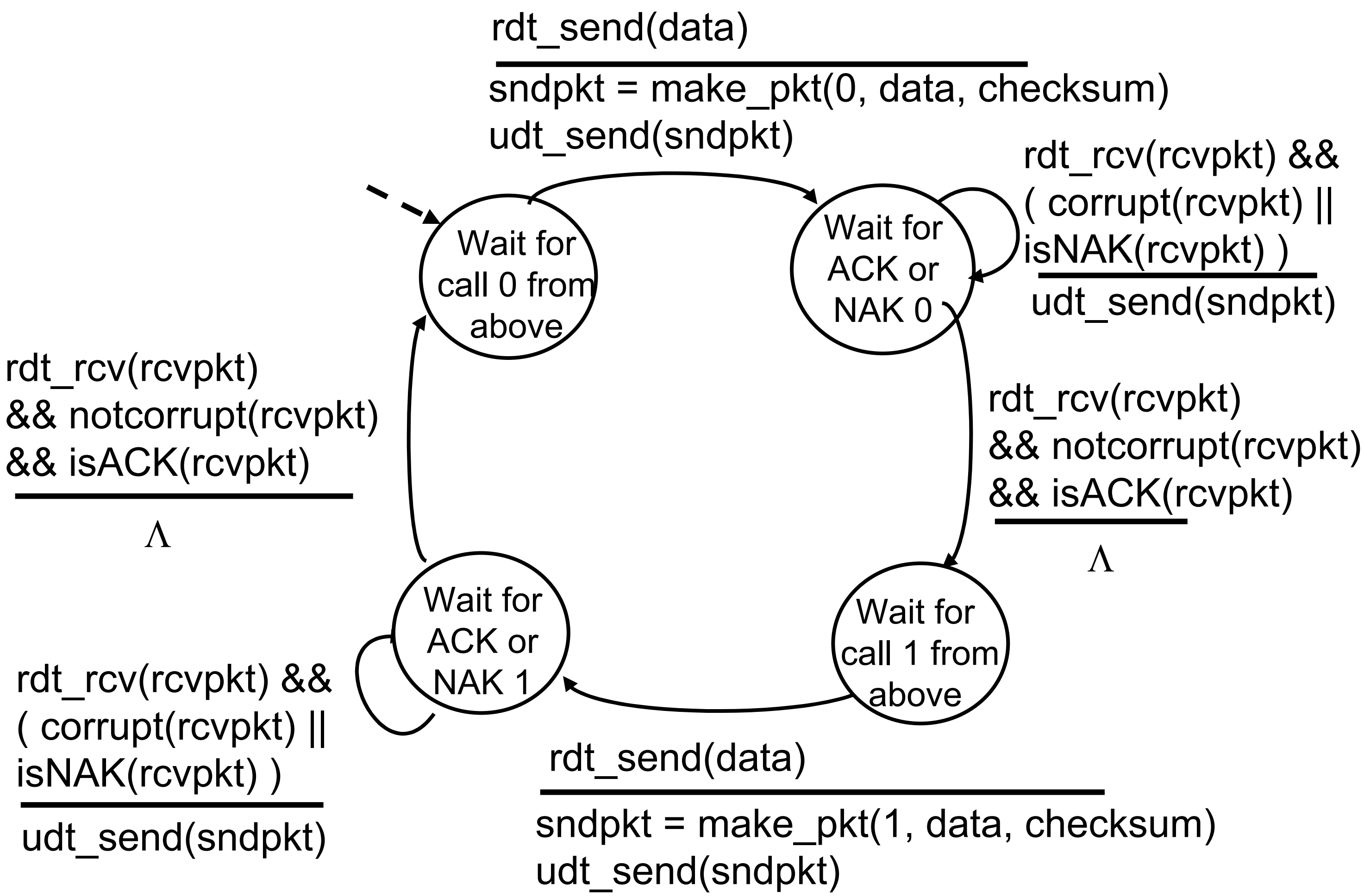
Handling duplicates:

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

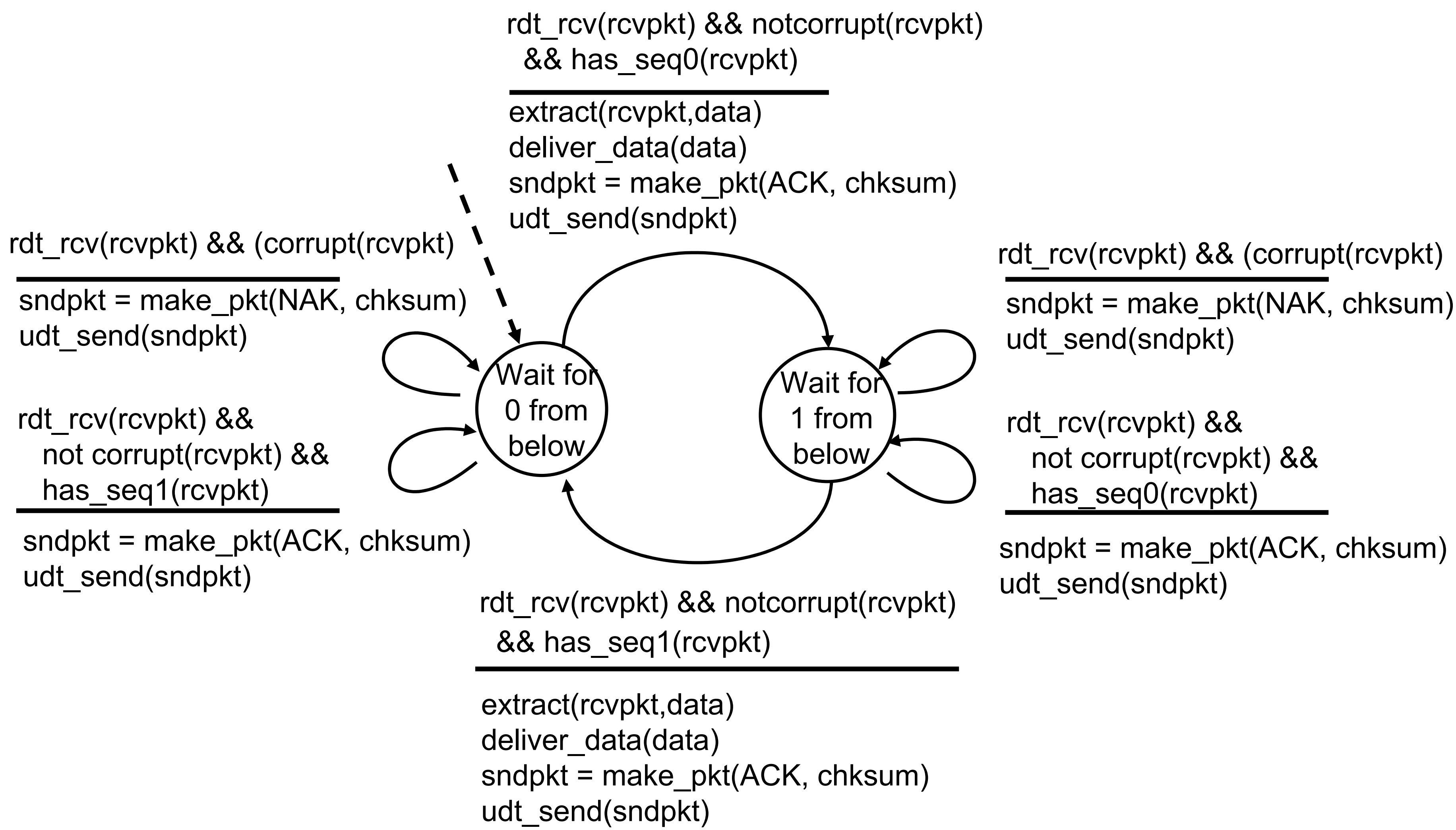
stop and wait

Sender sends one packet, then waits for receiver response

Rdt2.1: Sender with garbled ACK/NAKs



Rdt2.1: Receiver with garbled ACK/NAKs



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 "current" pkt has 0 or 1 seq. #

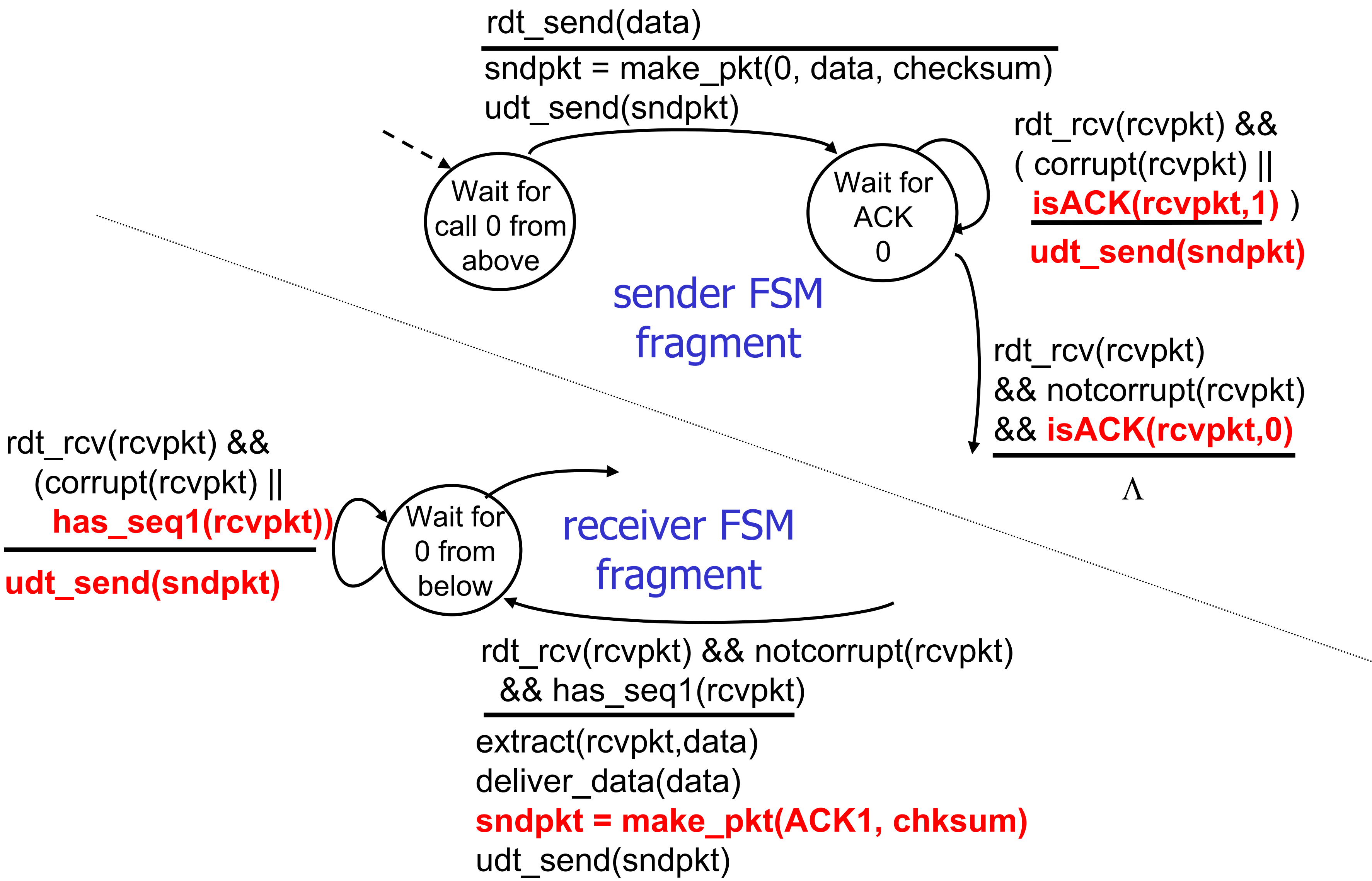
Receiver:

- 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 or ACKs)

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

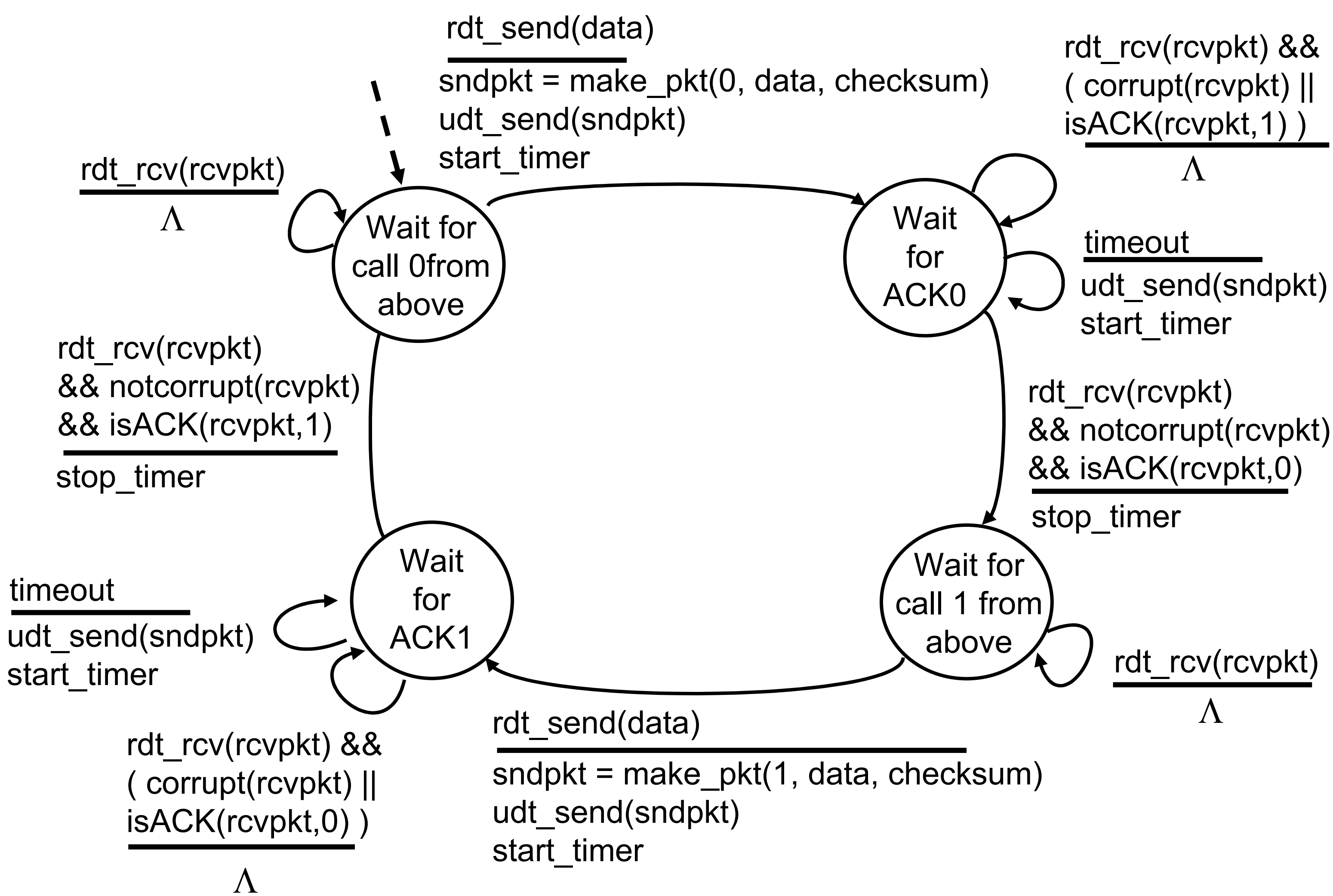
Q: How to deal with loss?

- Sender waits until certain data or ACK lost, then retransmits

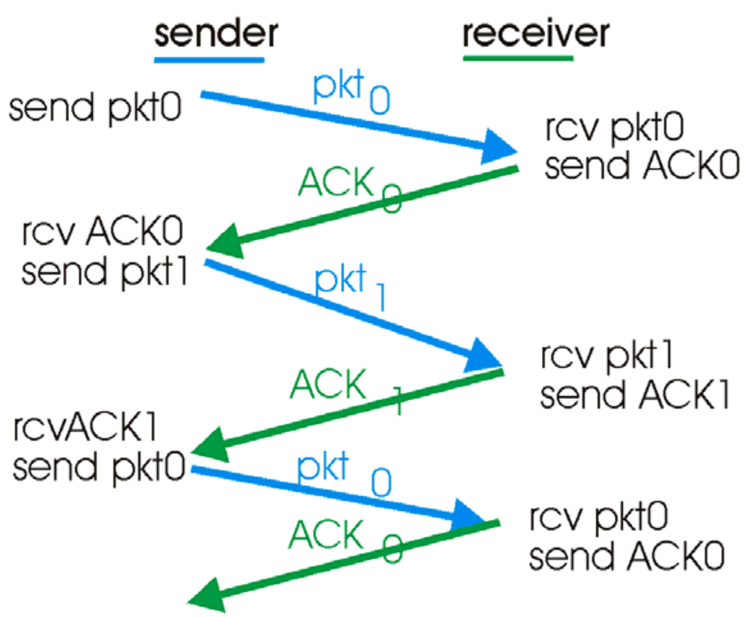
Approach: Sender waits "reasonable" amount of time for ACK

- Retransmits if no ACK received in this time
- If pkt (or ACK) just delayed (not lost):
  - Retransmission will be duplicate, but use of seq. #'s already handles this
  - Receiver must specify seq # of pkt being ACKed
- Requires countdown timer

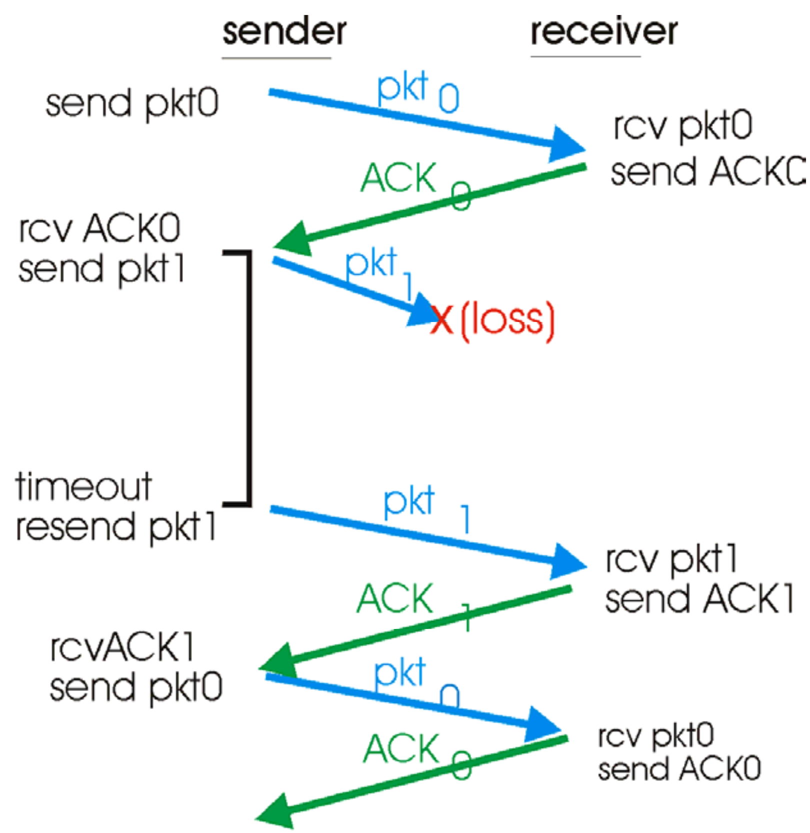
Rdt3.0 sender



Rdt3.0 in action

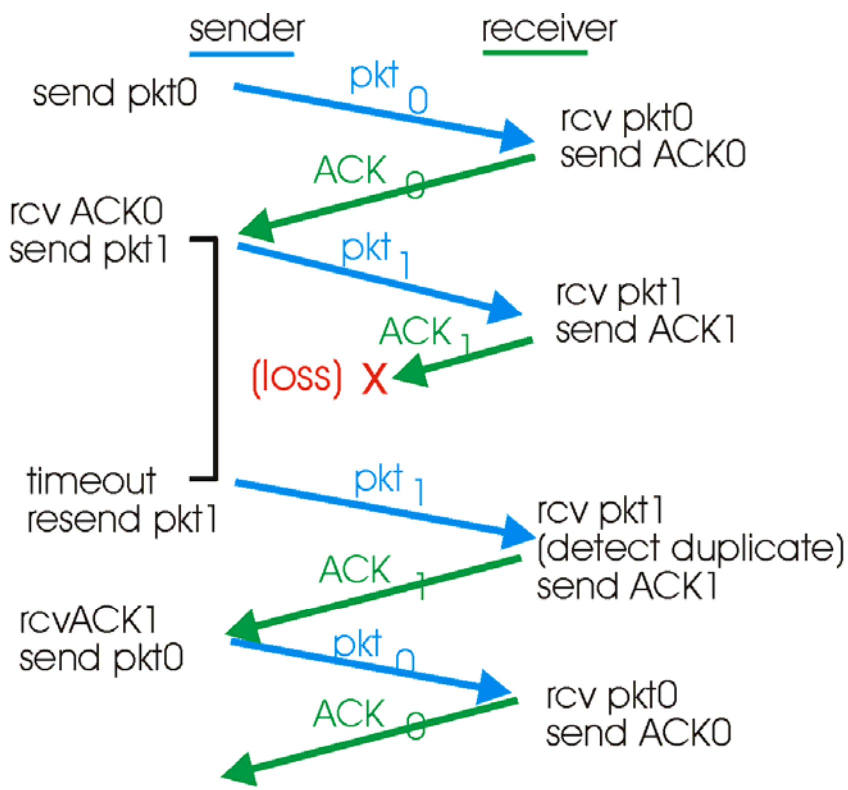


(a) operation with no loss

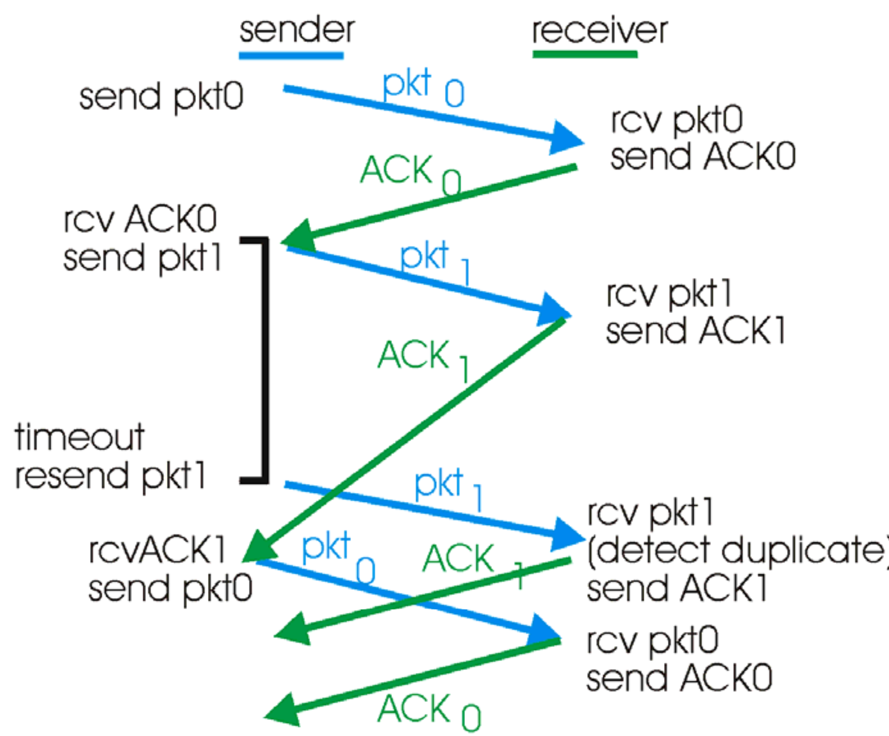


(b) lost packet

Rdt3.0 in action (cont.)



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

- Rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

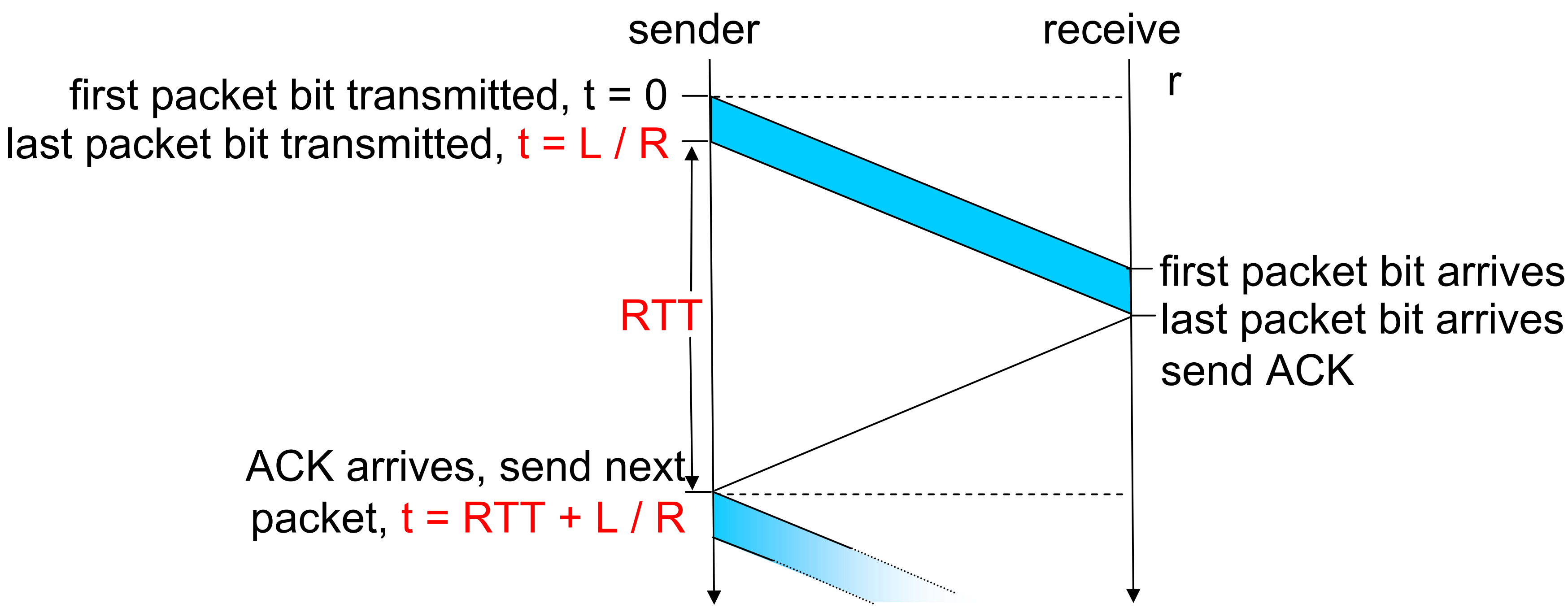
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\ {}^{kb}\!/_{pkt}}{10^9\ {}^{b}\!/_{s}} = 8 \mu s$$

  - U $_{sender}$: utilization – fraction of time sender busy sending

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008ms}{30.008ms} = 0.00027$$

  - 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
  - Network protocol limits use of physical resources

Rdt3.0: Stop-and-wait operation

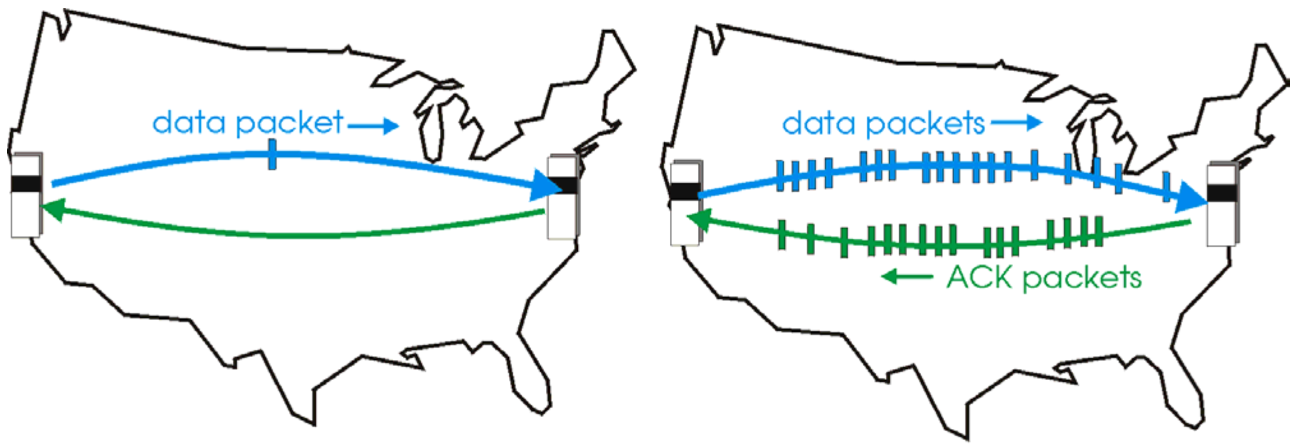


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008ms}{30.008ms} = 0.00027$$

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

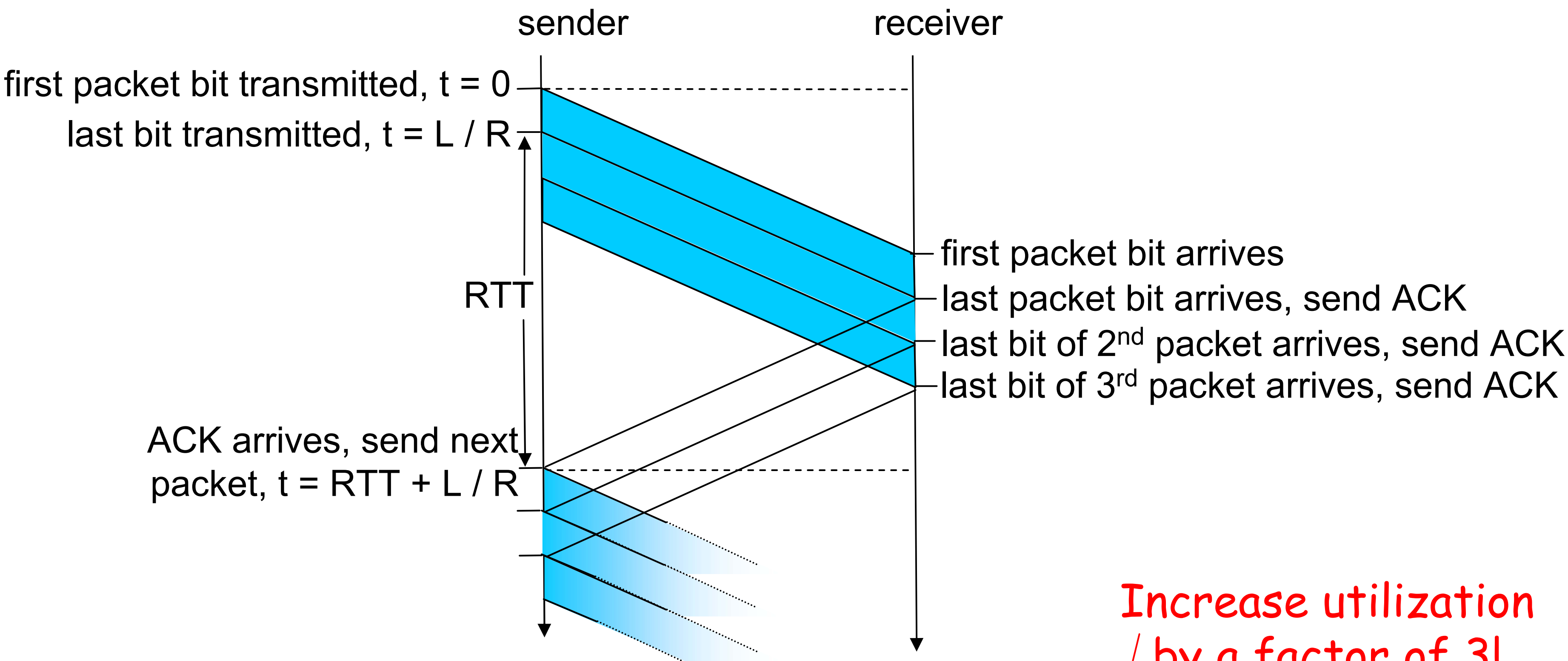
- Range of sequence numbers must be increased
- Buffering at sender and/or receiver



(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



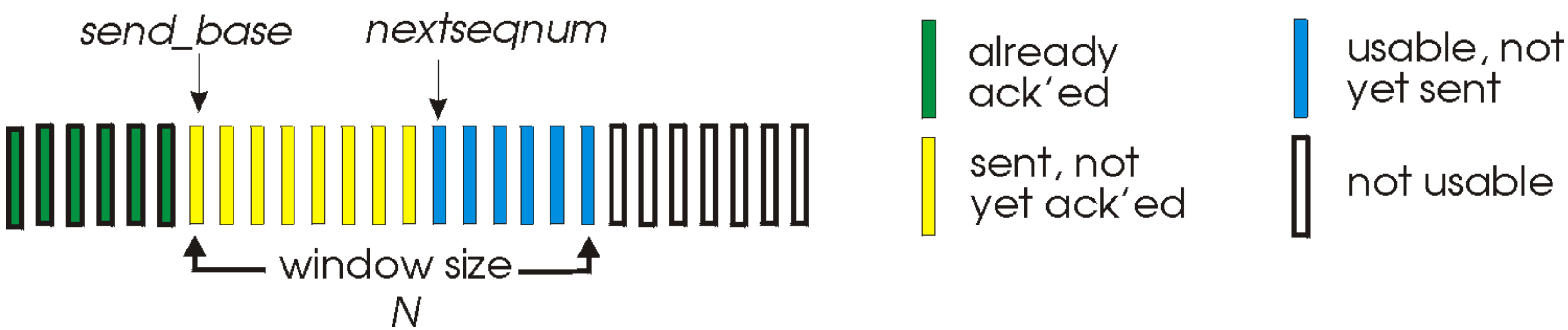
Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 \cdot L/R}{RTT + L/R} = \frac{0.024ms}{30.008ms} = 0.00079$$

Go-Back-N

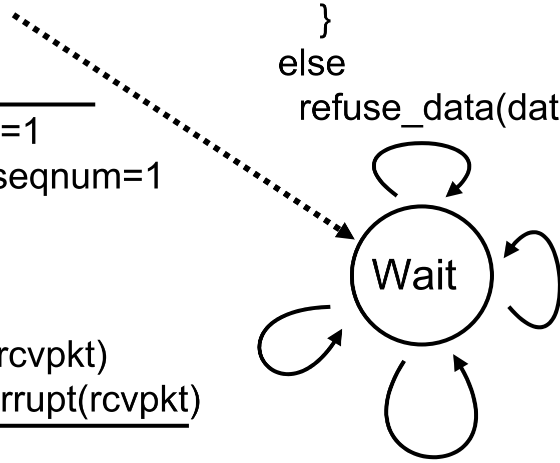
Sender:

- "Window" of up to N, consecutive unack'ed pkts allowed



- Why limit to N? → later (flow control, congestion control)
- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - May deceive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- *Timeout(n):* Retransmit pkt n and all higher seq # pkts in window
- K-bit seq # in pkt header (⇨ N < 2^K)

GBN: Sender extended FSM



Λ
___
base=1
nextseqnum=1

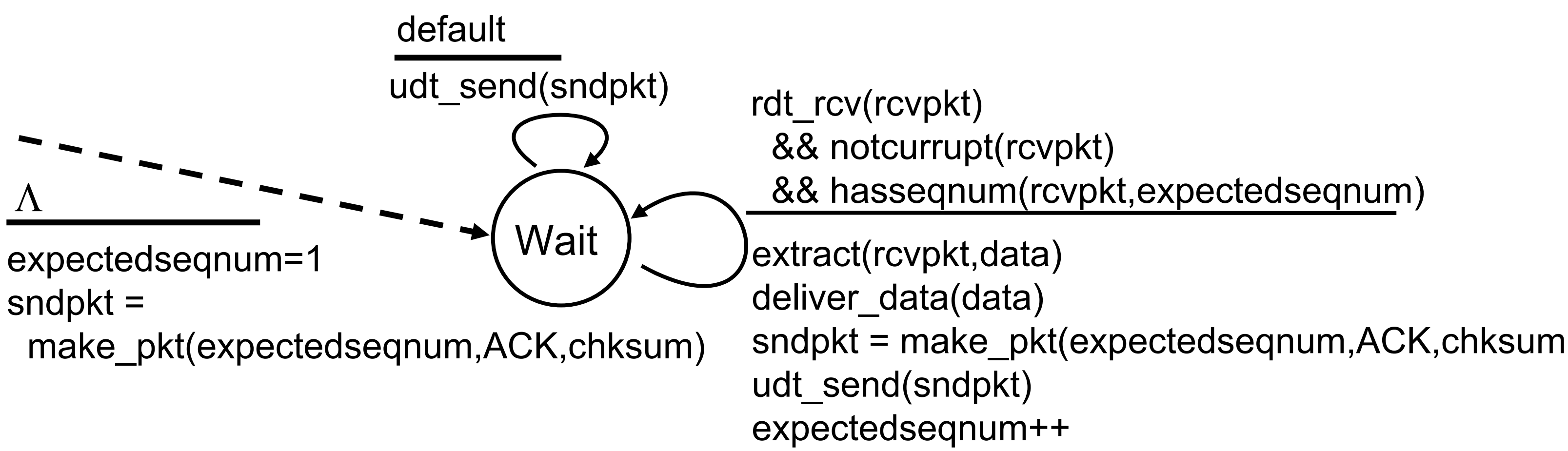
rdt_send(data)
___
```
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
       start_timer
    nextseqnum++
    }
else
  refuse_data(data)
```

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

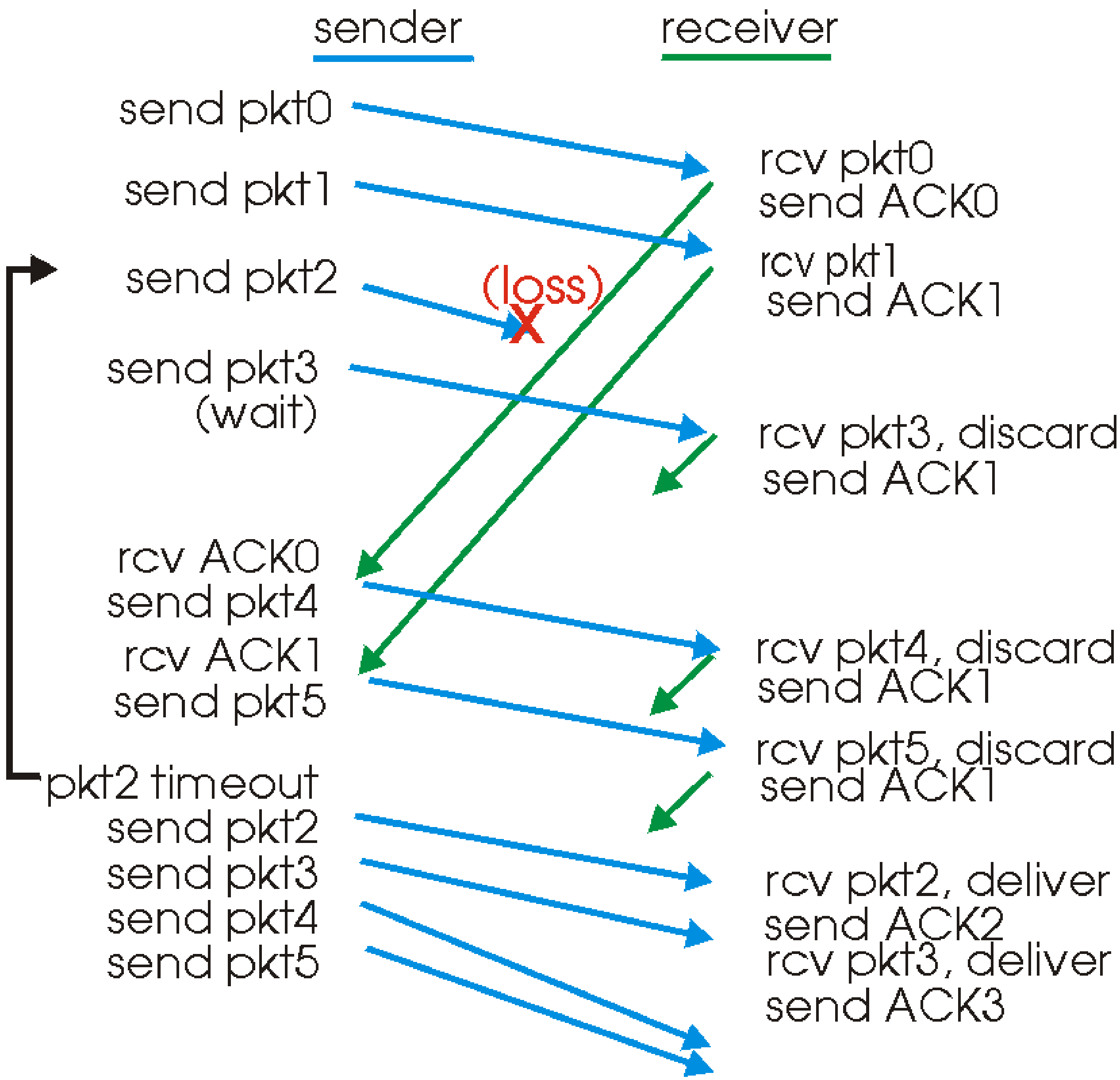


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #

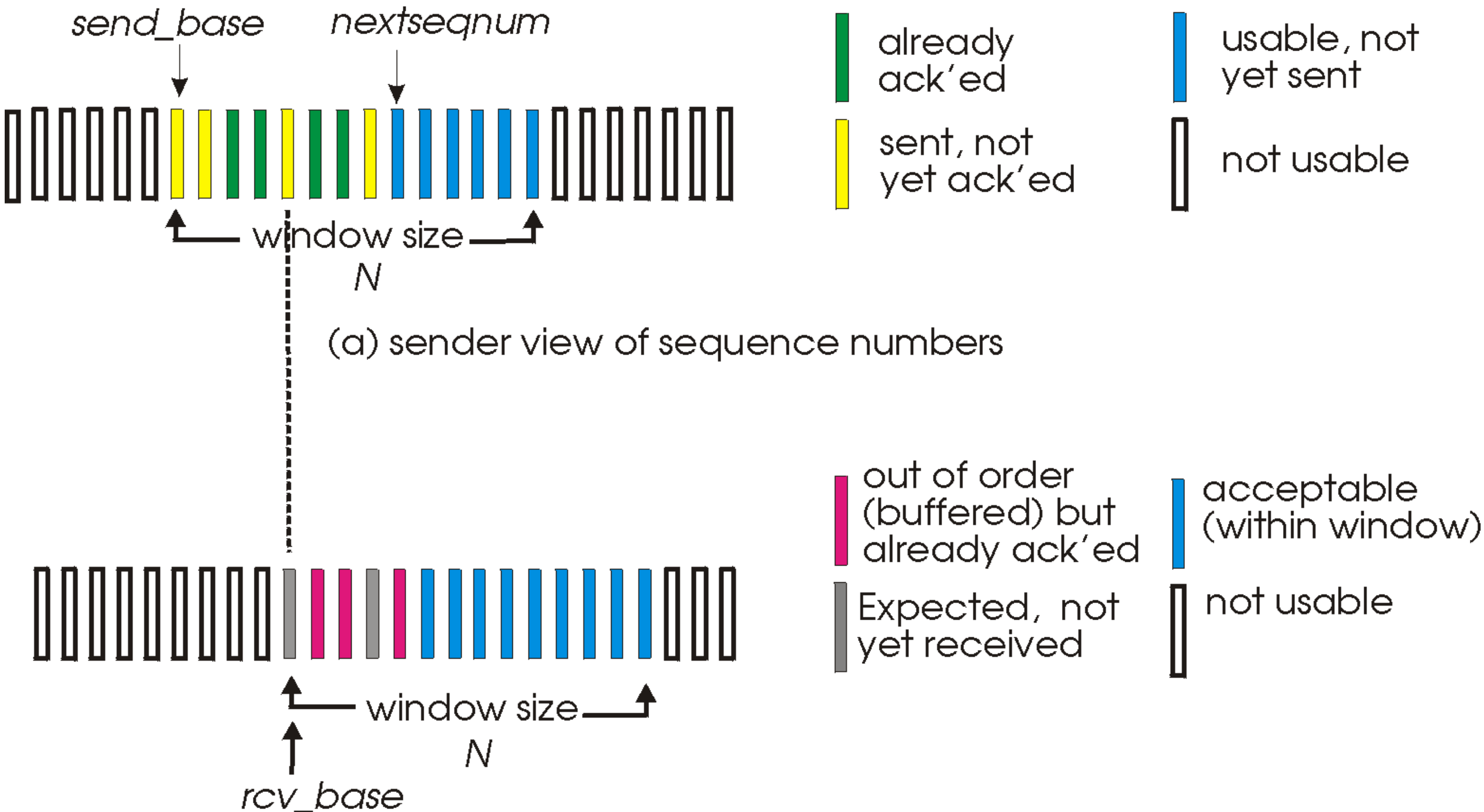
GBN in action



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
- ⇨ No need to retransmit correctly received, but out-of-order packets
- Sender window
  - N consecutive seq #'s
  - Again 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
- Cancel timer for n
- 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

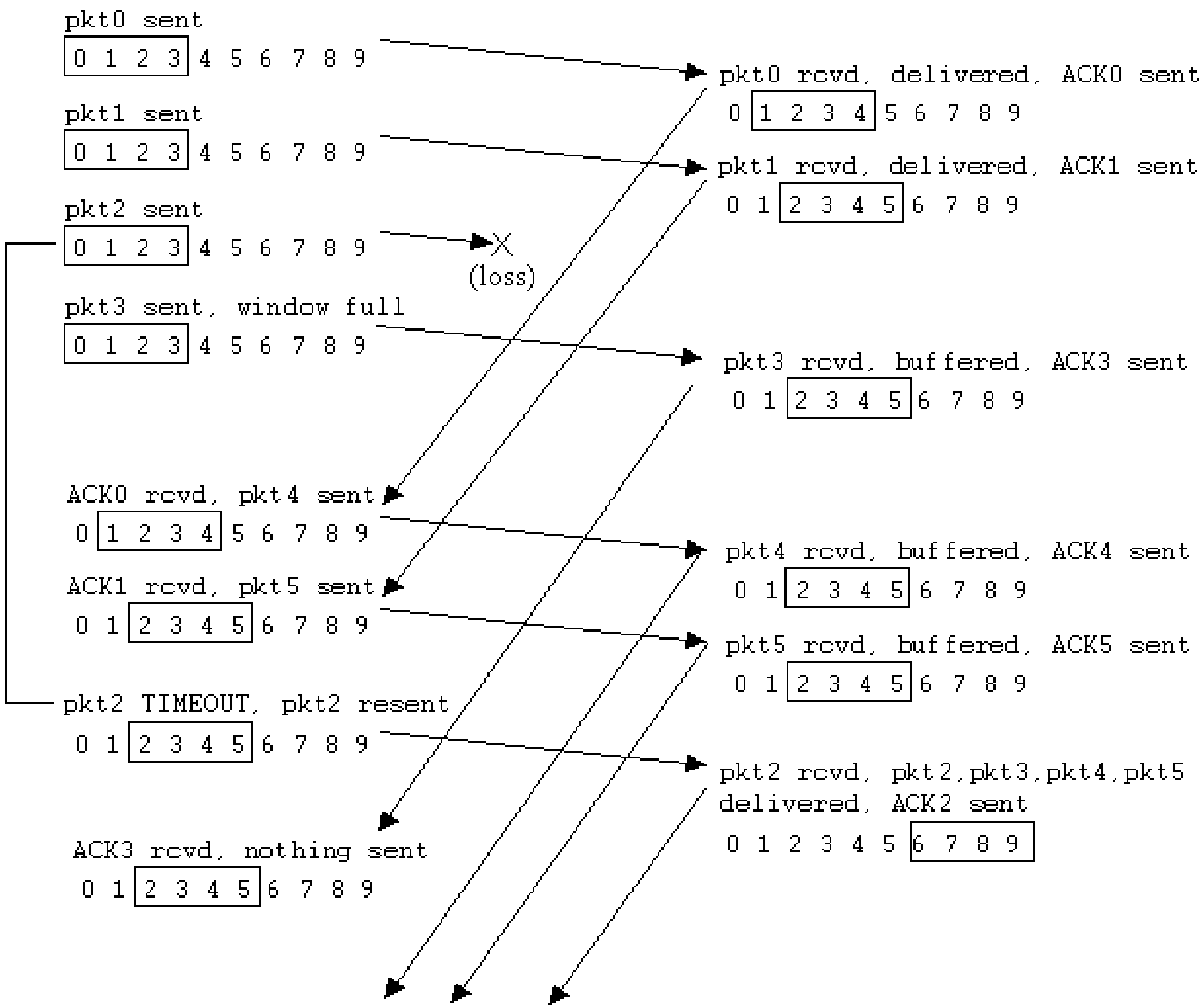
Pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

Otherwise:

- Ignore

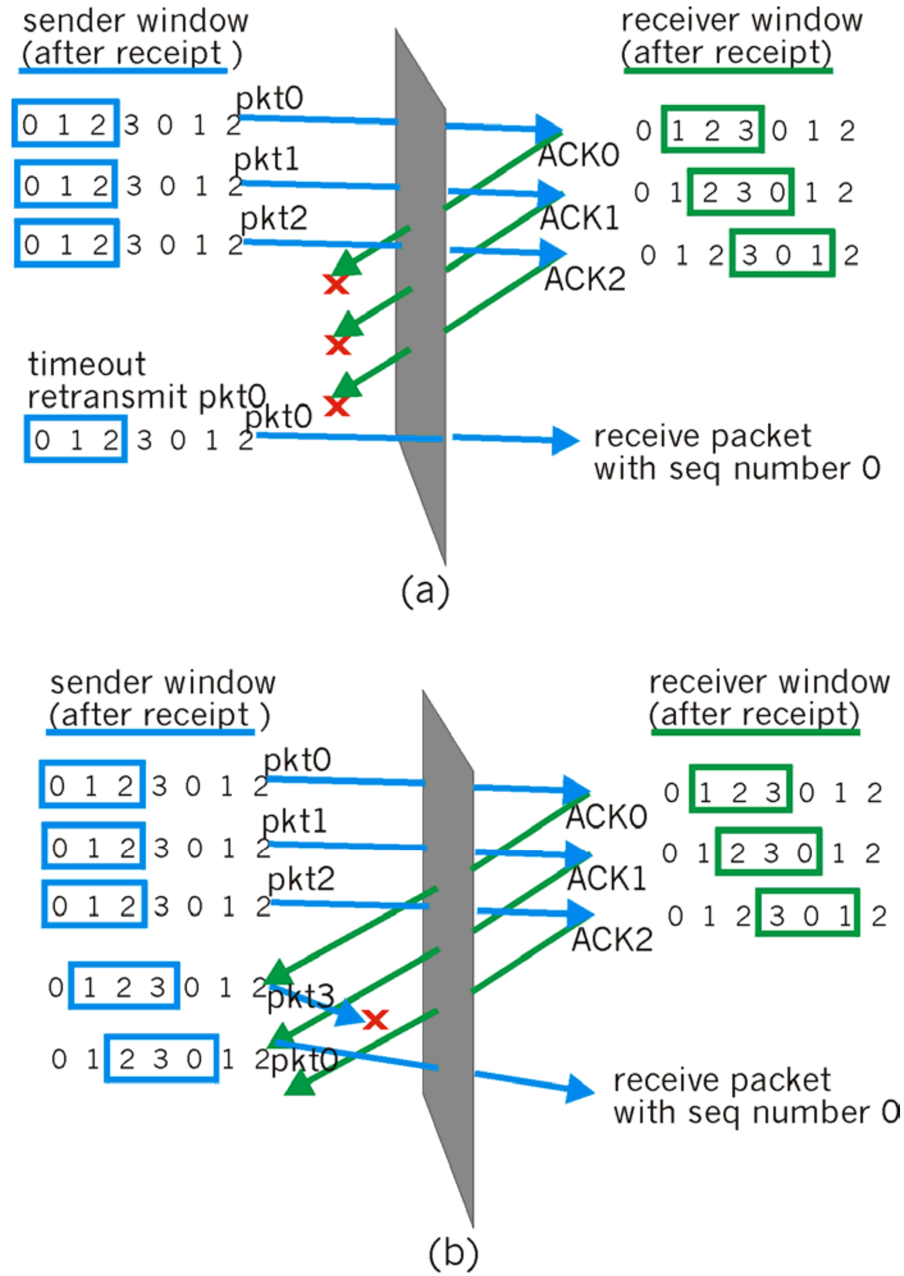
Selective repeat in action



Selective repeat: dilemma

Example:

- Seq #'s: 0, 1, 2, 3
- Window size=3



- Receiver sees no difference in two scenarios!
- Incorrectly passes duplicate data as new in (a)

Q: What relationship between seq # size and window size?