Chapter 3: Transport Layer
Part A

Course on Computer Communication and Networks, CTH/GU

The slides are adaptation of the slides made available by the authors of the course's main textbook

Chapter 3: Transport Layer

Chapter goals:

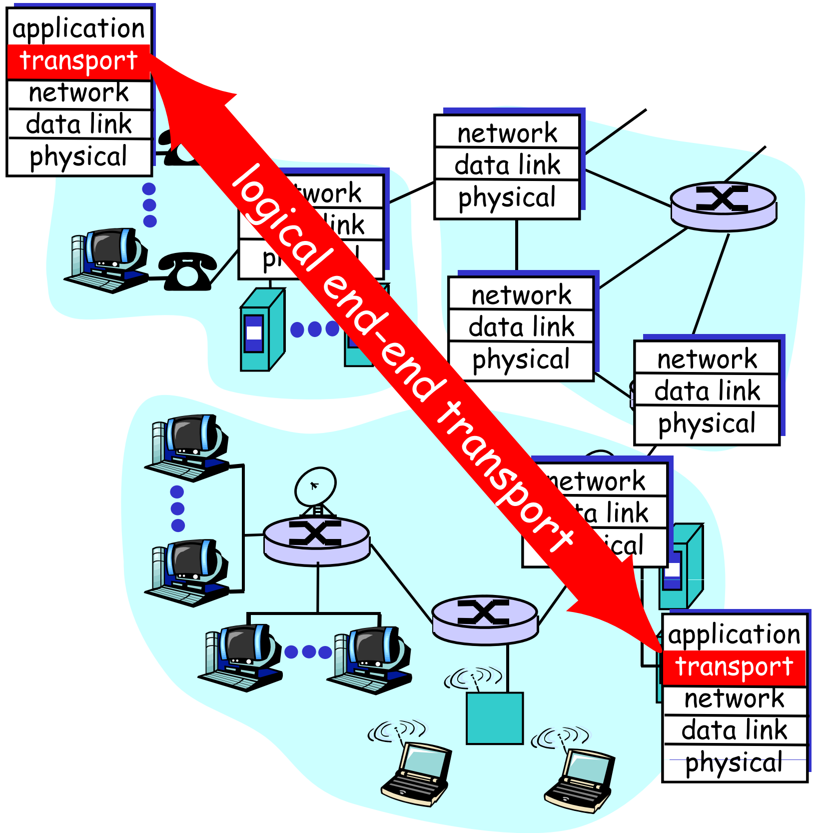
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control (some now; more in connection with RT applications)
- instantiation and implementation in the Internet

Chapter Overview:

- transport layer services
- multiplexing/demultiplexing
- connectionless transport: UDP
- principles of reliable data transfer
- connection-oriented transport: TCP
  - reliable transfer
  - flow control
  - connection management
  - TCP congestion control

Transport services and protocols

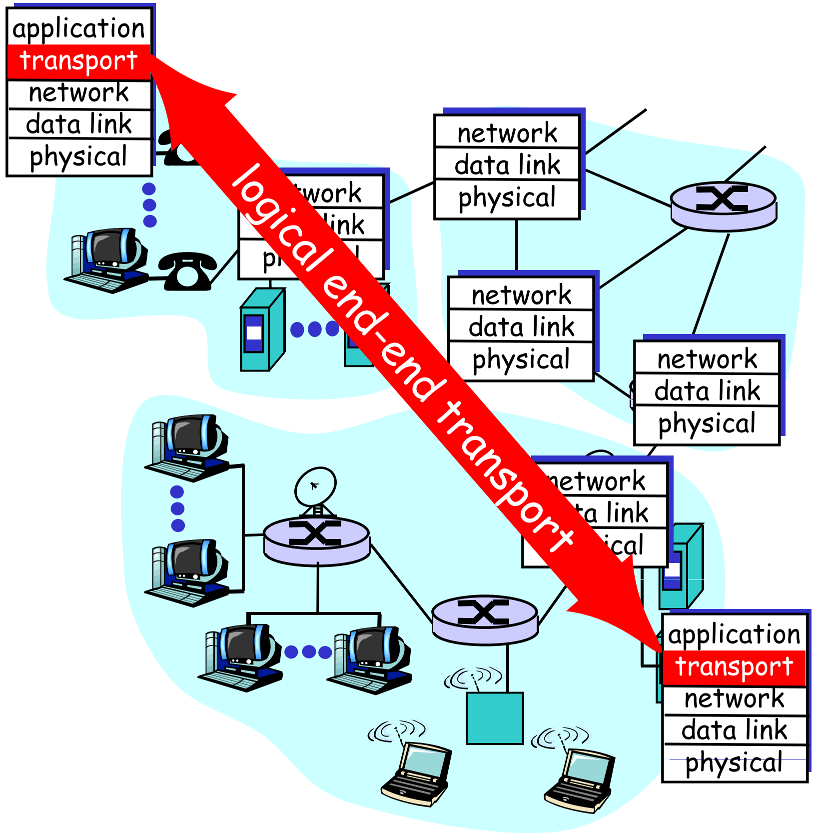
- provide *logical communication* between app' processes running on different hosts
- transport protocols run in end systems
- transport vs network layer services:
  - *network layer:* data transfer between end systems
  - *transport layer:* data transfer between processes
    - uses and enhances, network layer services



Transport-layer protocols

Internet transport services:

- reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup
- unreliable ("best-effort"), unordered unicast or multicast delivery: UDP
- services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast



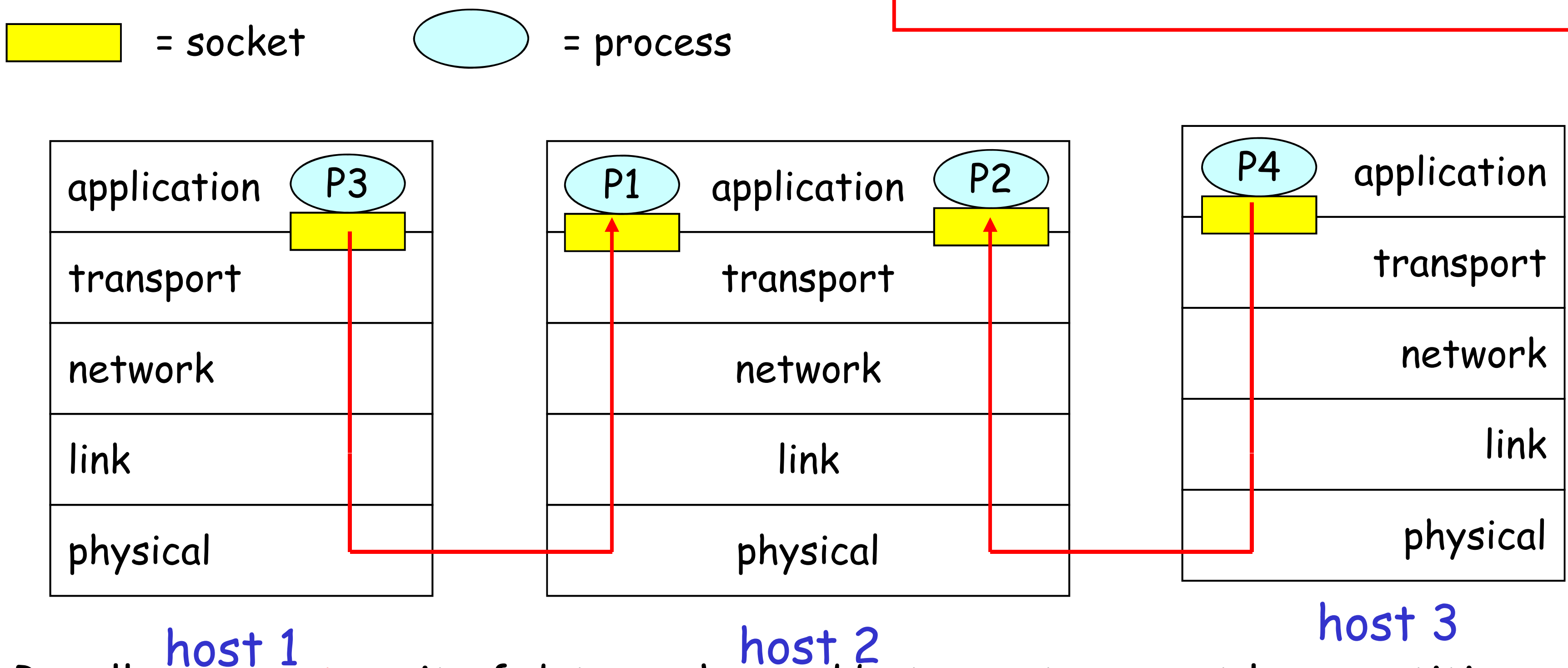
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

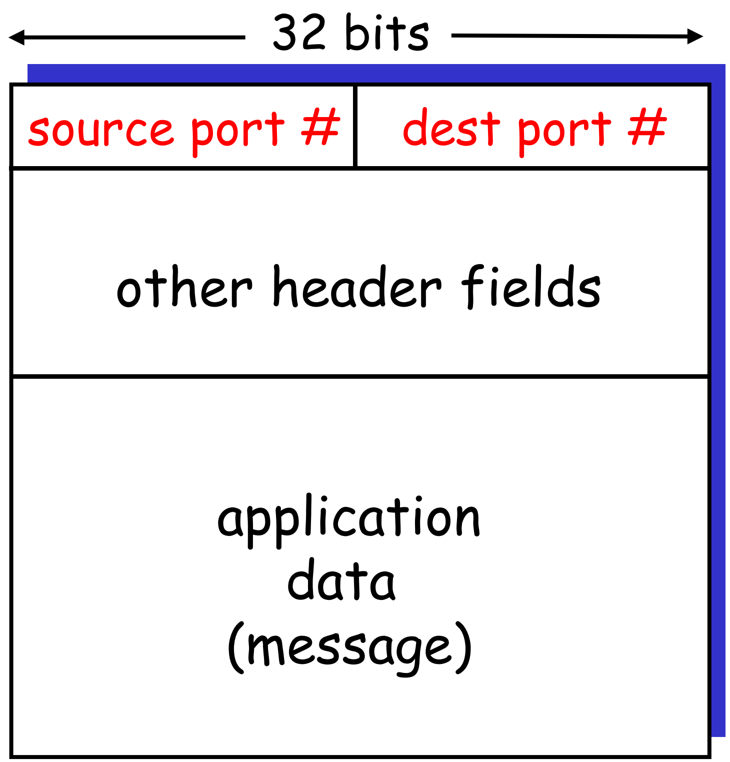
gathering data, enveloping data with header (later used for demultiplexing)



Recall: *segment* - unit of data exchanged between transport layer entities
aka TPDU: transport protocol data unit

How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

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

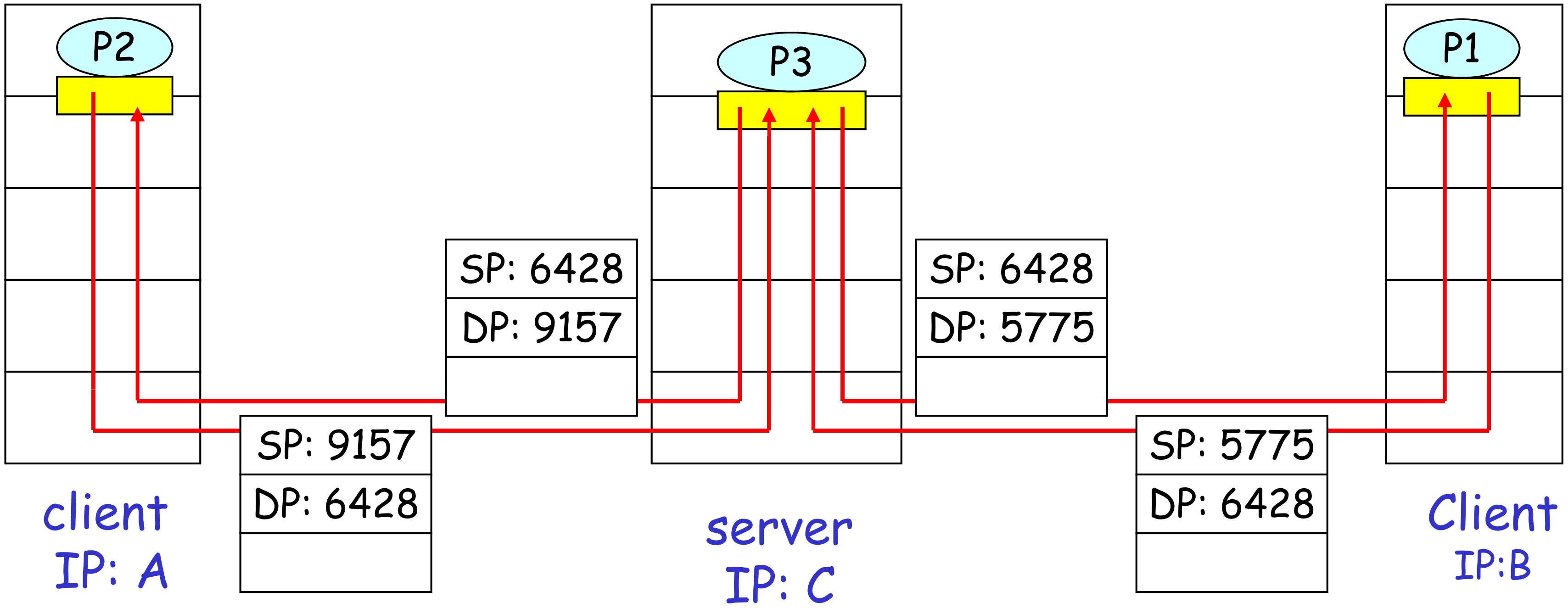
- UDP socket identified by two-tuple:

(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);
```

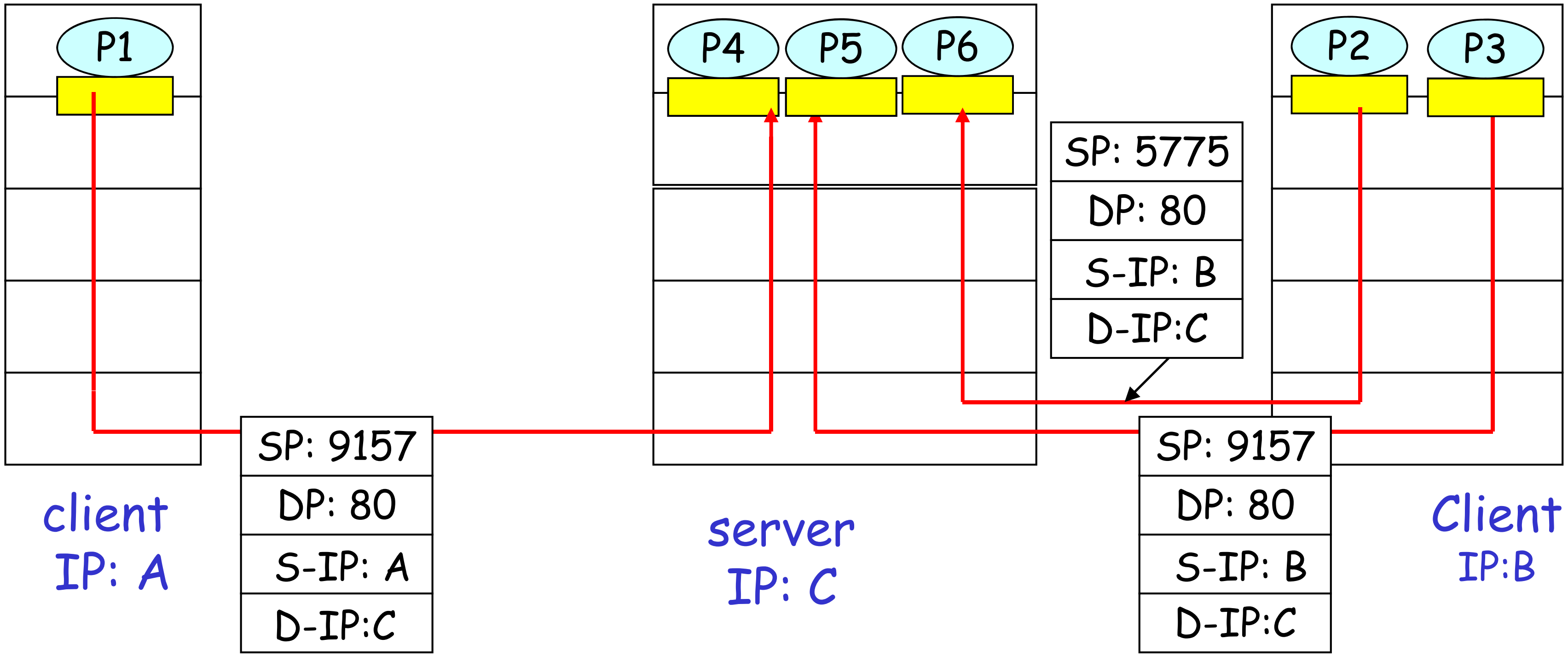


SP provides "return address"

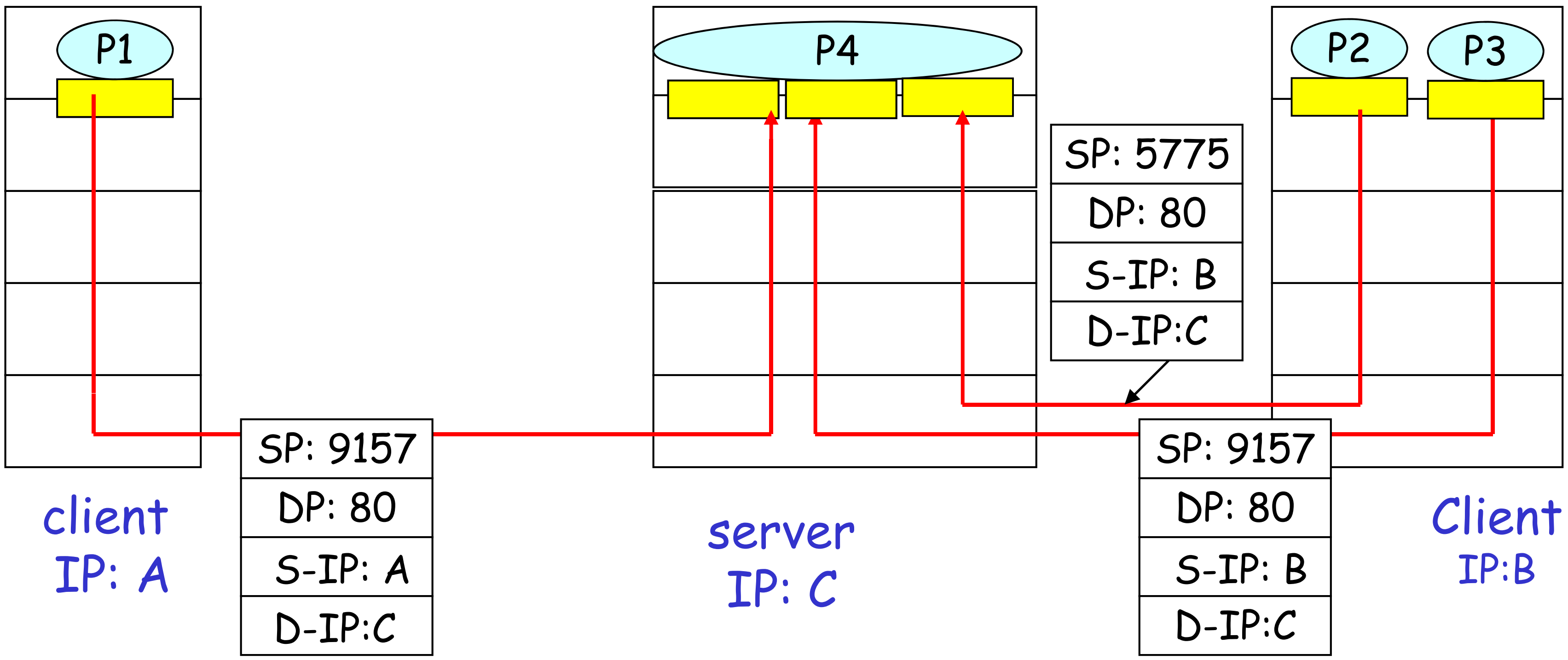
Connection-oriented demux

- 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



UDP: User Datagram Protocol [RFC 768]

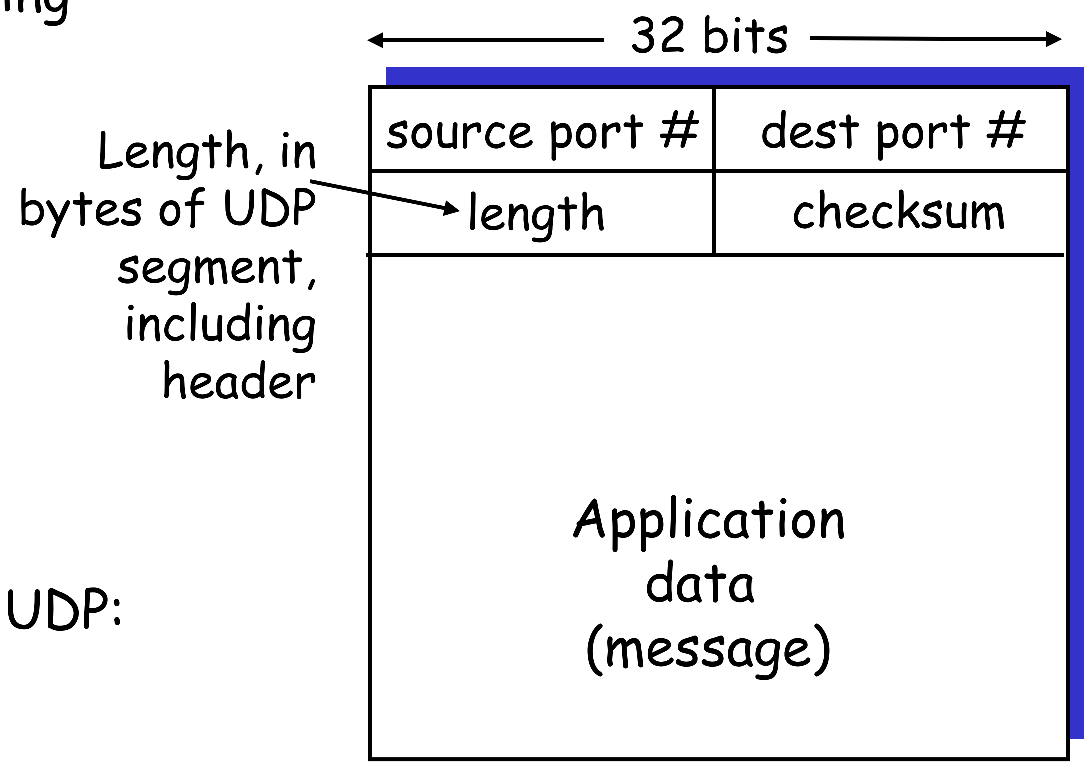
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others; subsequent UDP segments can arrive in wrong order

Is UDP any good?

- 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 users (why?):
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!



UDP segment format

UDP Checksum: check bit flips

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

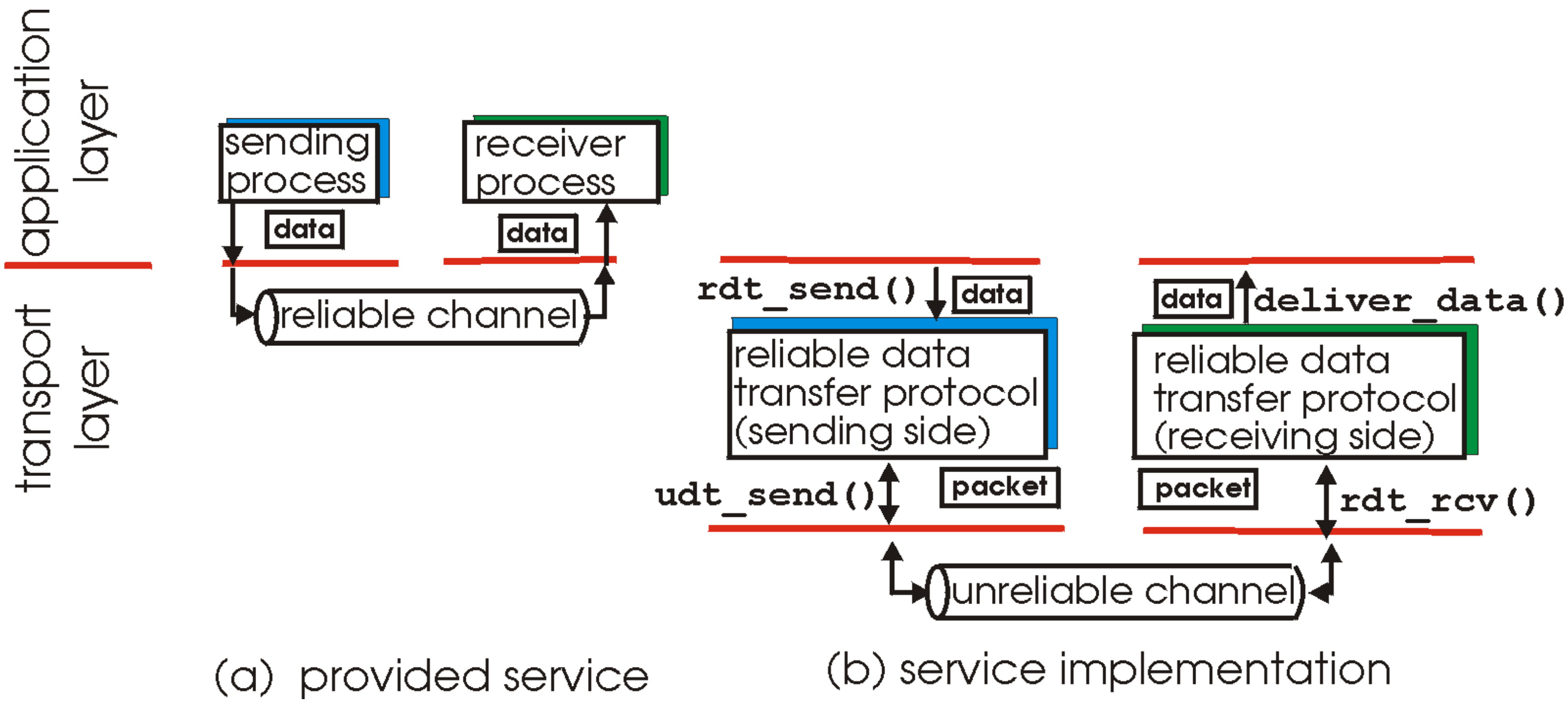
Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected (*report error to app or discard*)
  - YES - no error detected.
    - *But maybe (very rarely) errors nonethless?* More later ….

	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: Add to final	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

Principles of Reliable data transfer

- important in (app.,) transport, link layers
- in top-10 list of important networking topics!

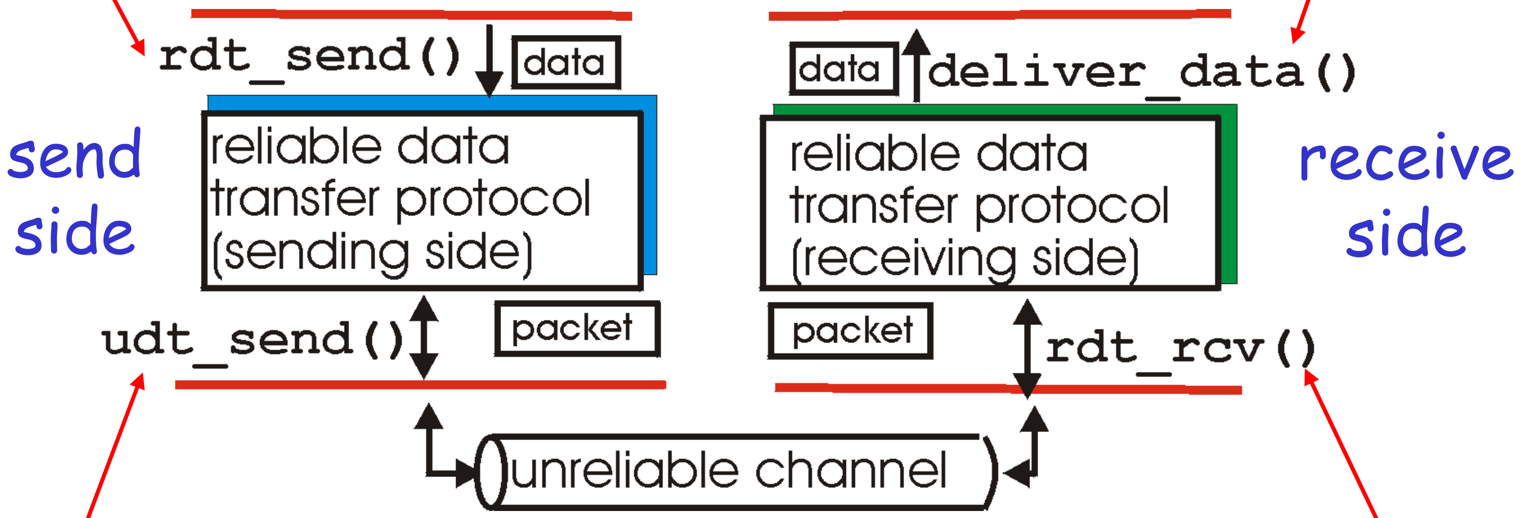


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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



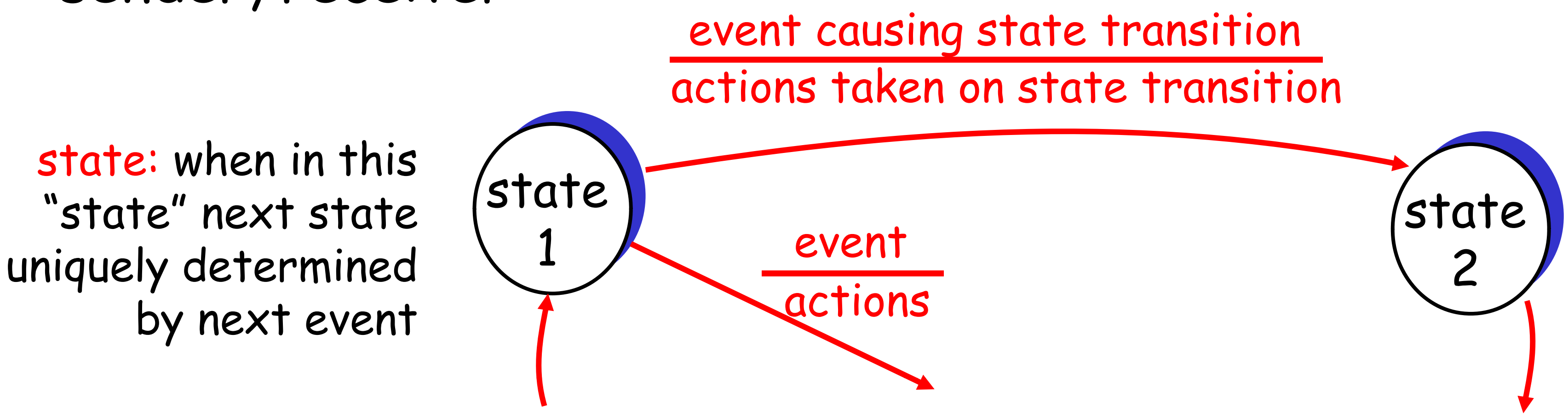
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

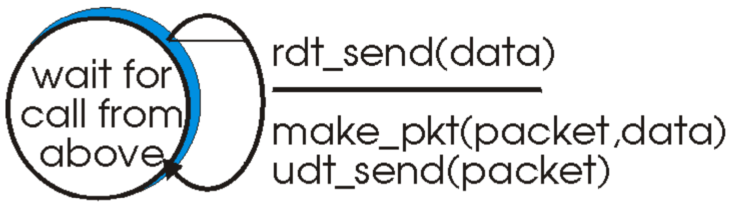
We'll:

- 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

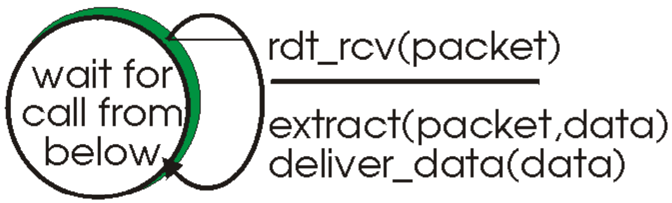


Rdt1.0: reliable transfer over a reliable channel

- 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



(a) rdt1.0: sending side

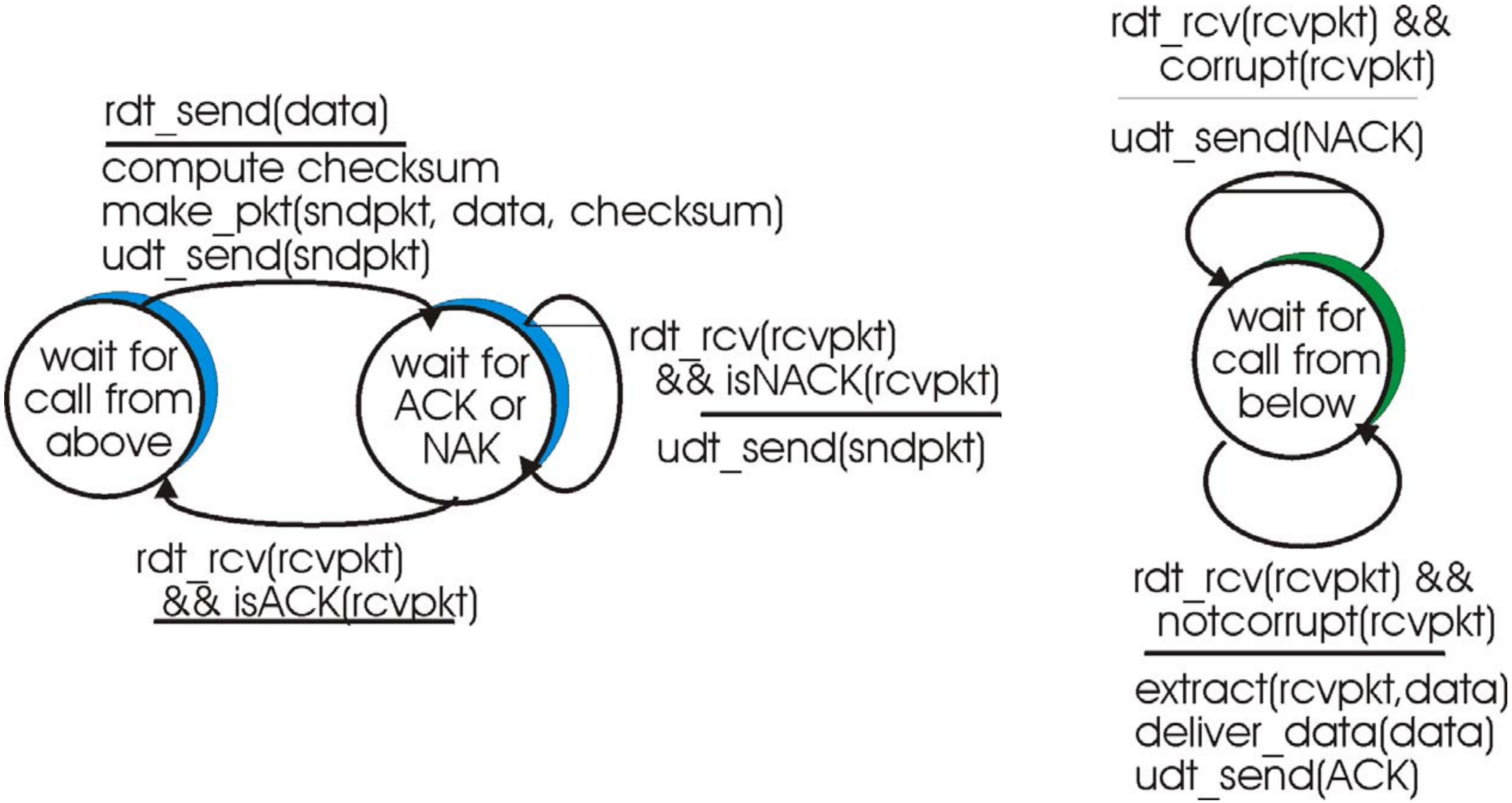


(b) rdt1.0: receiving side

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

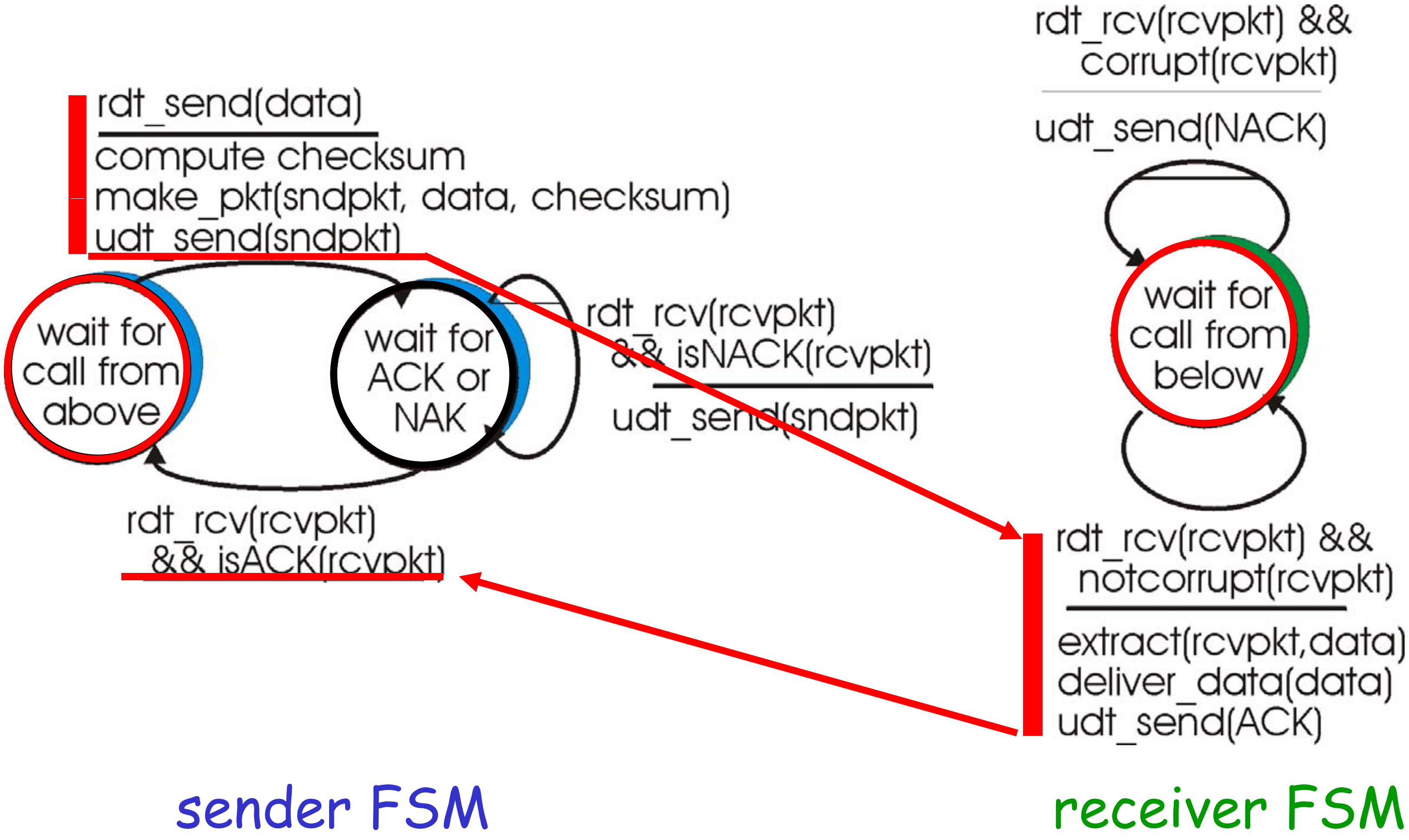
rdt2.0: FSM specification



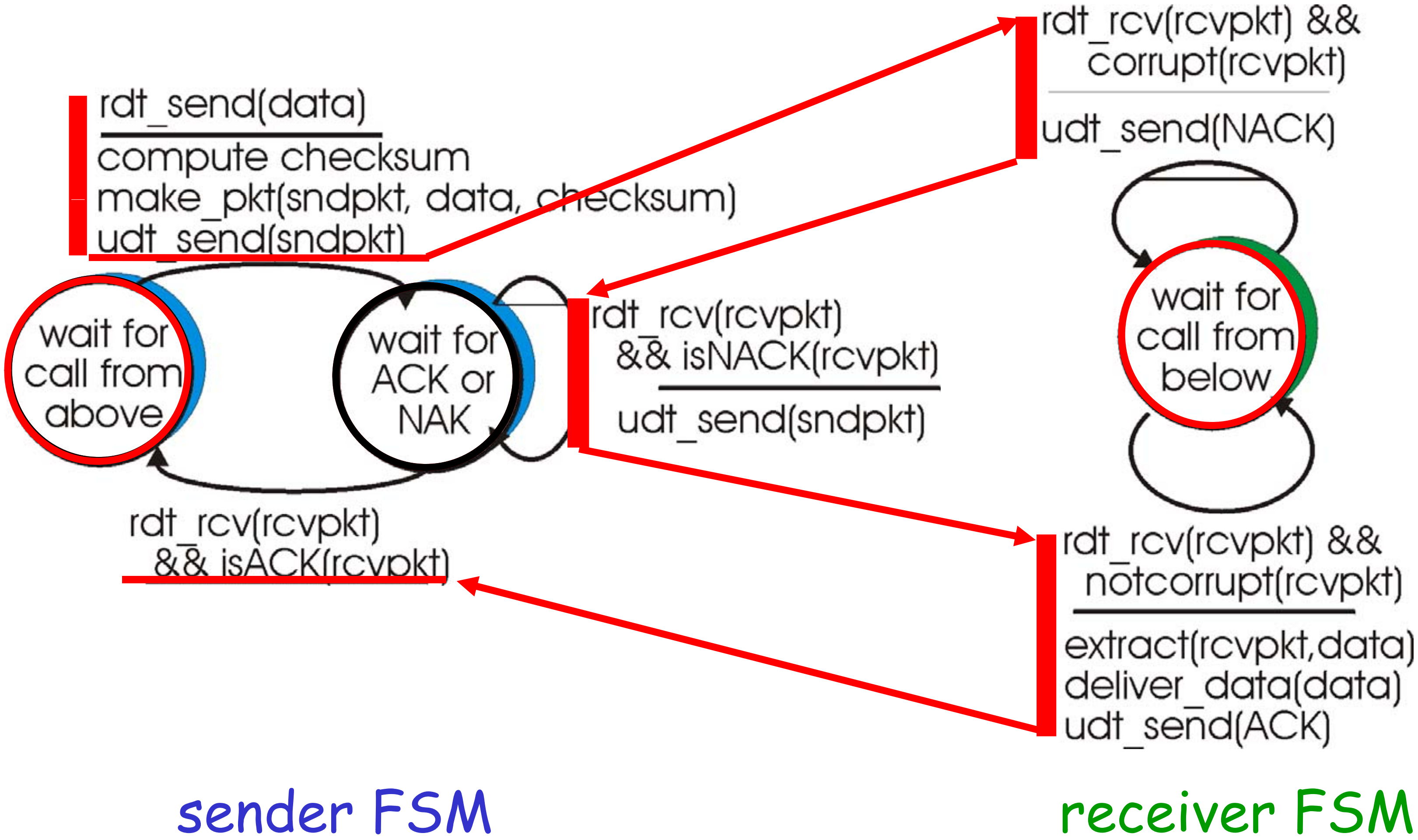
sender FSM

receiver FSM

rdt2.0: in action (no errors)



rdt2.0: in action (error scenario)



rdt2.0 has an issue:

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!

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!

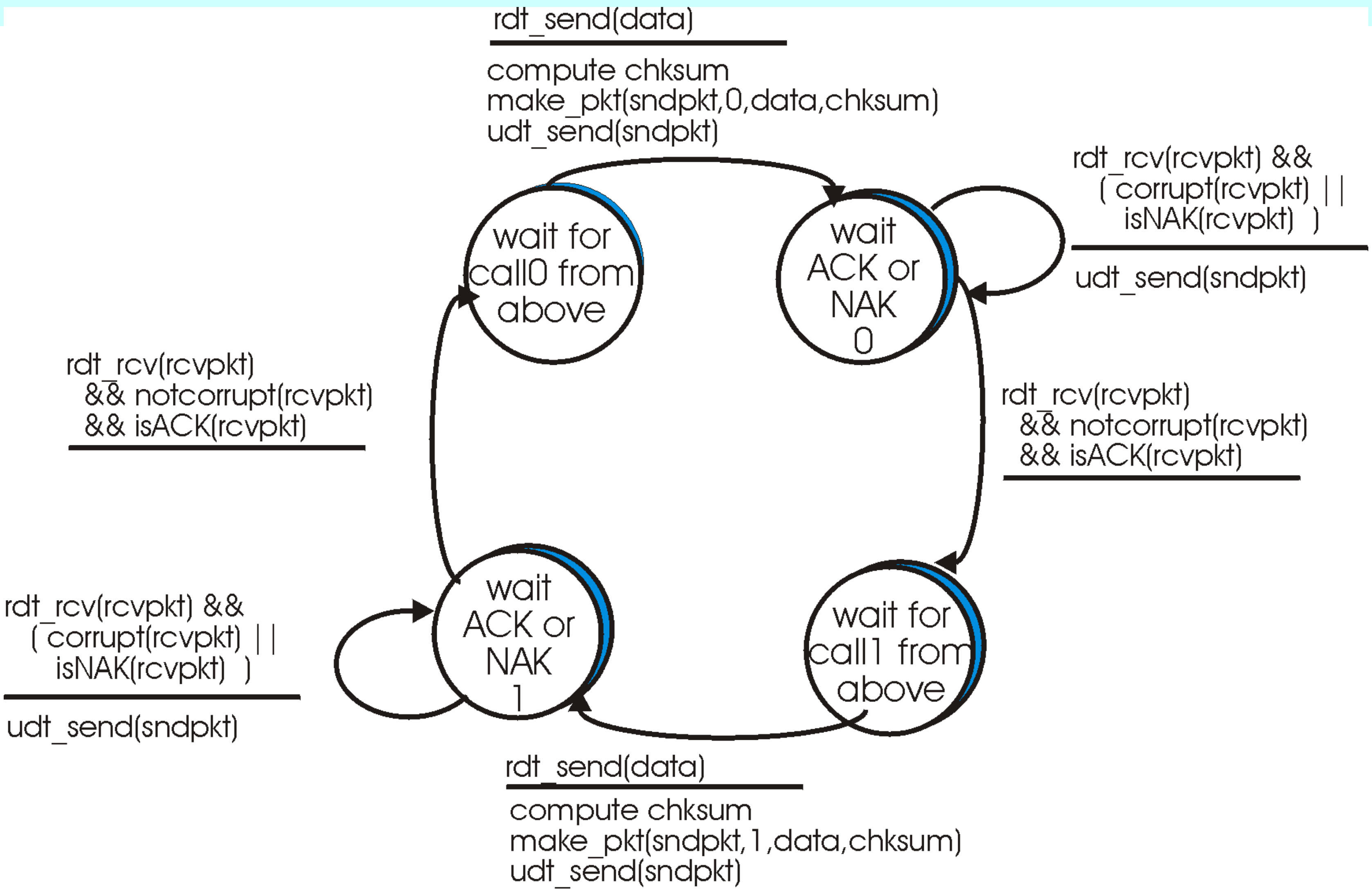
Handling duplicates:

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

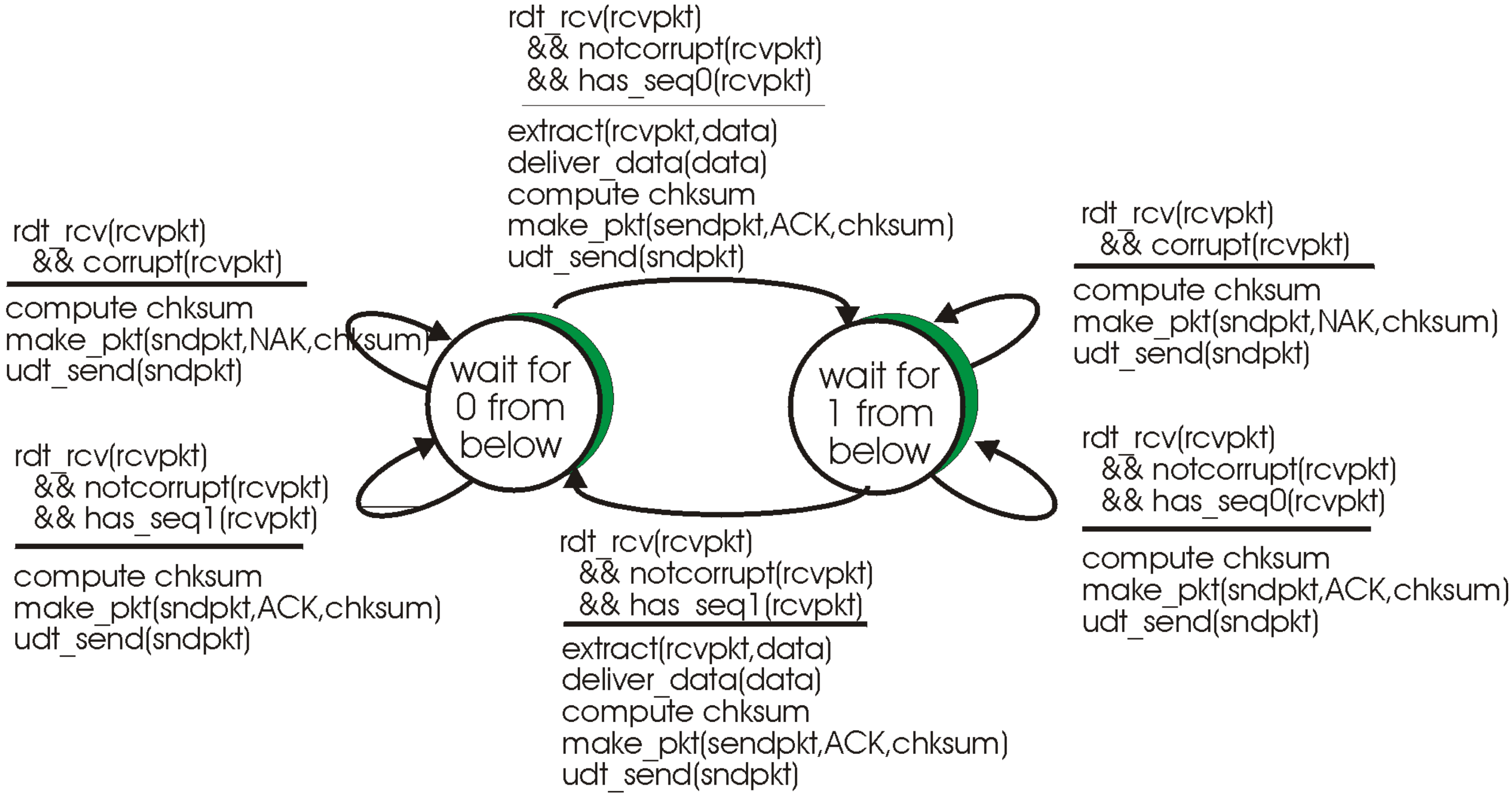
stop and wait

Sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles 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

Bounding sequence numbers...

... s.t. no wraparound, i.e. we do not run out of numbers: binary value suffices for stop-and-wait:

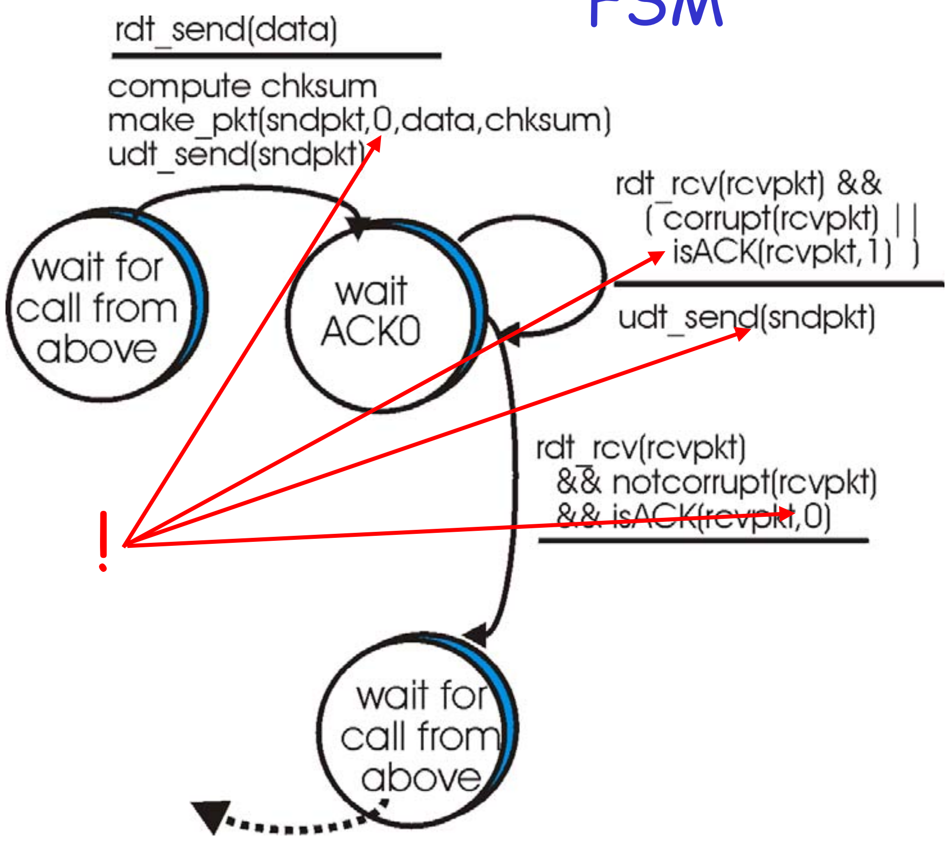
Prf: assume towards a contradiction that there is wraparound when we use binary seq. nums.

- R expects segment #f, receives segment #(f+2):
  R rec. f+2 => S sent f+2 => S rec. ack for f+1
  => R ack f+1=> R ack f => contradiction
- R expects f+2, receives f:
  R exp. f+2 => R ack f+1 => S sent f+1
  => S rec. ack for f => contradiction

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*

sender FSM



rdt3.0: channels with errors *and* loss

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

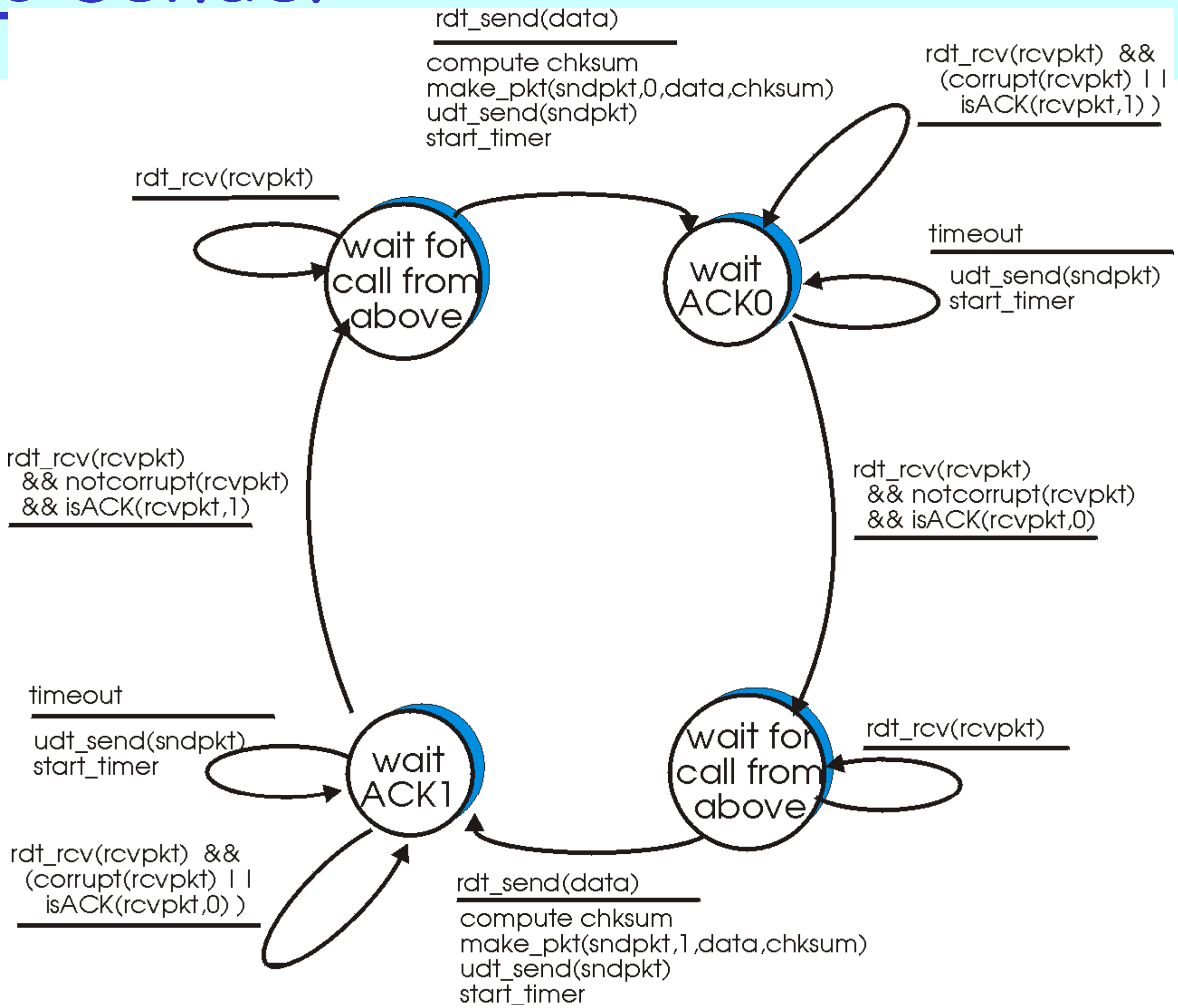
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: how to deal with loss?

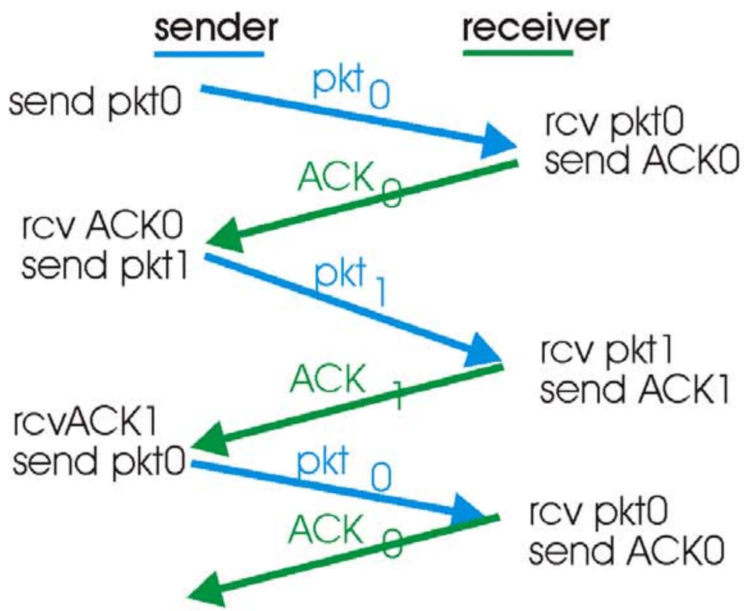
Approach: sender waits "reasonable" amount of time for ACK

- 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

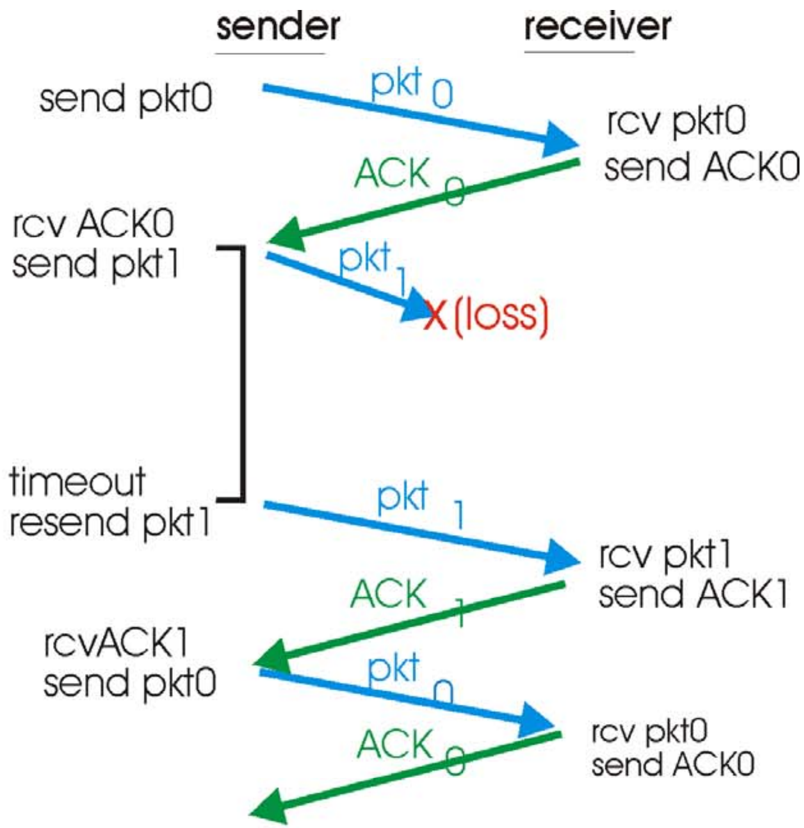
rdt3.0 sender



rdt3.0 in action

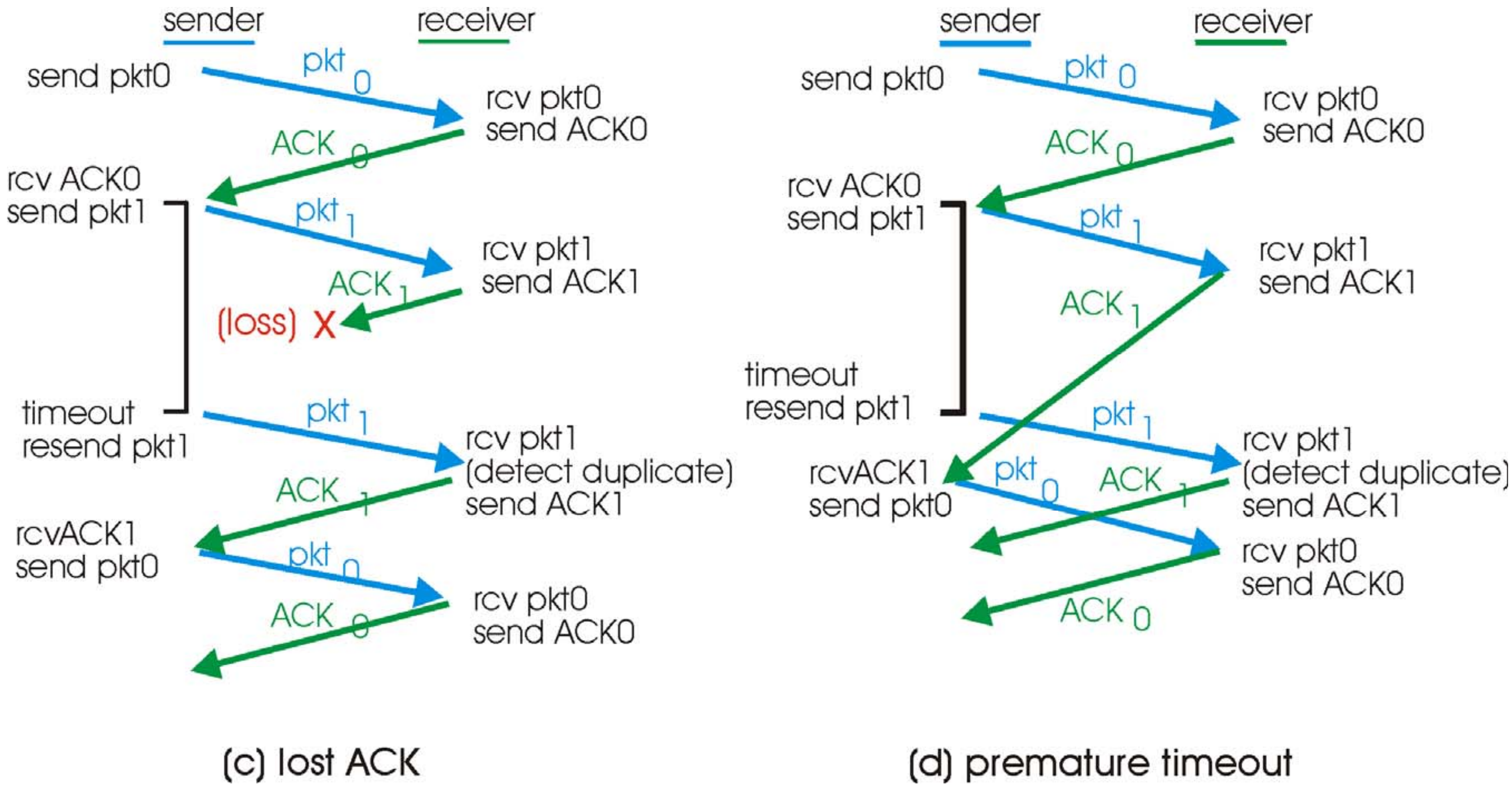


(a) operation with no loss

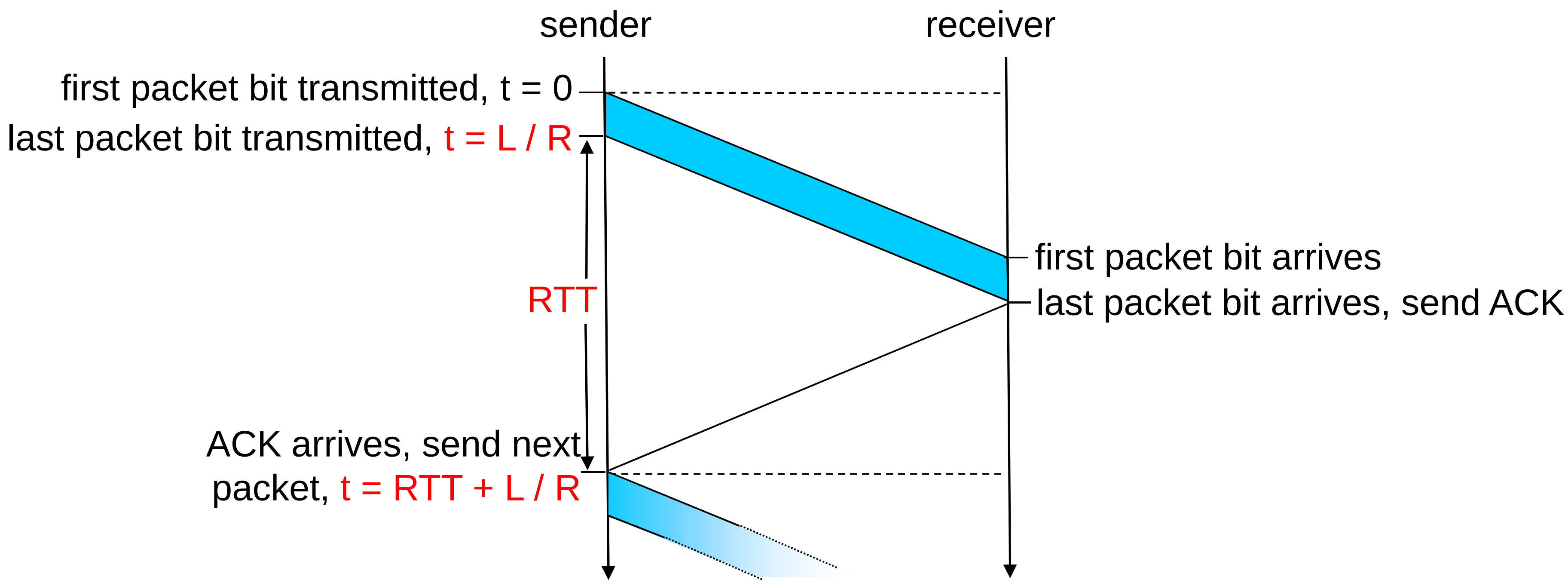


(b) lost packet

rdt3.0 in action



rdt3.0: stop-and-wait operation



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

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- Example: 50 Kbps, 500-msec round-trip propagation delay (satellite connection), transmit 1000-bit segments

$$T_{\text{transmit}} = \frac{1000\text{b}}{50\ \text{Kb/sec}} = 20\ \text{msec}$$

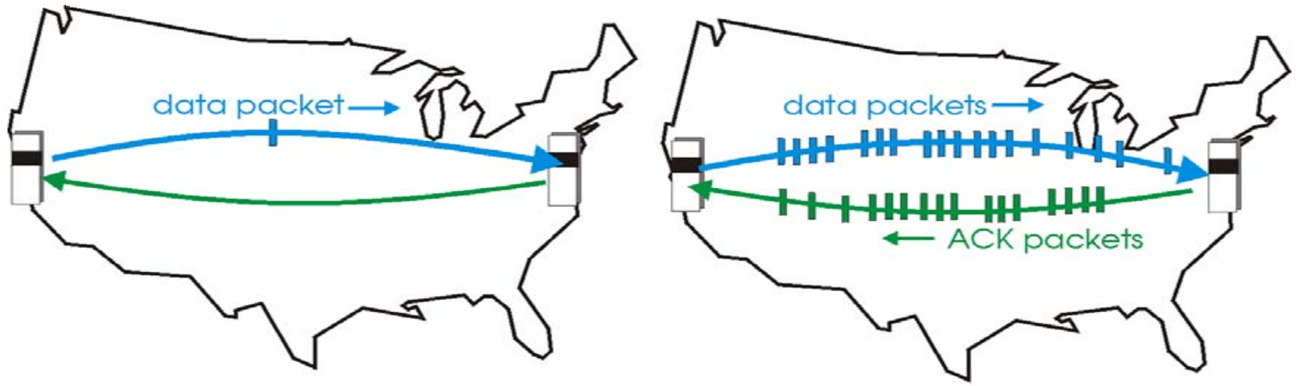
Utilization = U = $\dfrac{\textit{fraction of time}}{\textit{sender busy sending}} = \dfrac{20\ \text{msec}}{520\ \text{msec}} = 0.04$

- 1 segment every 520 msec -> 2 Kbps thruput (**effective bit-rate)** over 50 Kbps link
- network protocol limits use of physical resources!

Pipelined protocols

Pipelining: **Solution** to the problem of low utilization of stop-and-wait: sender allows multiple, up to N, "in-flight", yet-to-be-acknowledged pkts.

- Choice of N: optimally, it should allow the sender to continously transmit during the round-trip transit time
- range of sequence numbers must be increased
- buffering at sender and/or receiver

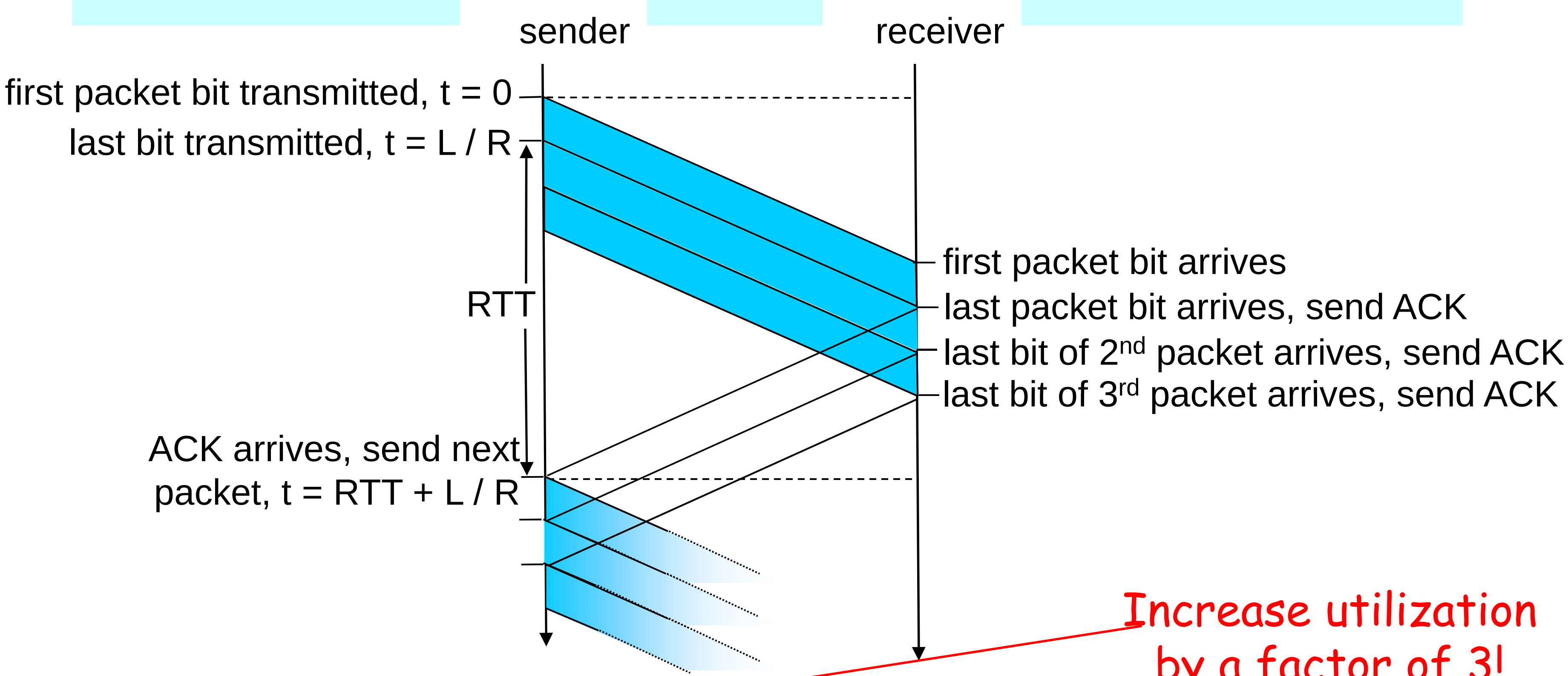


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat (check also corresponding on-line material in book's site)*

Pipelining: increased utilization

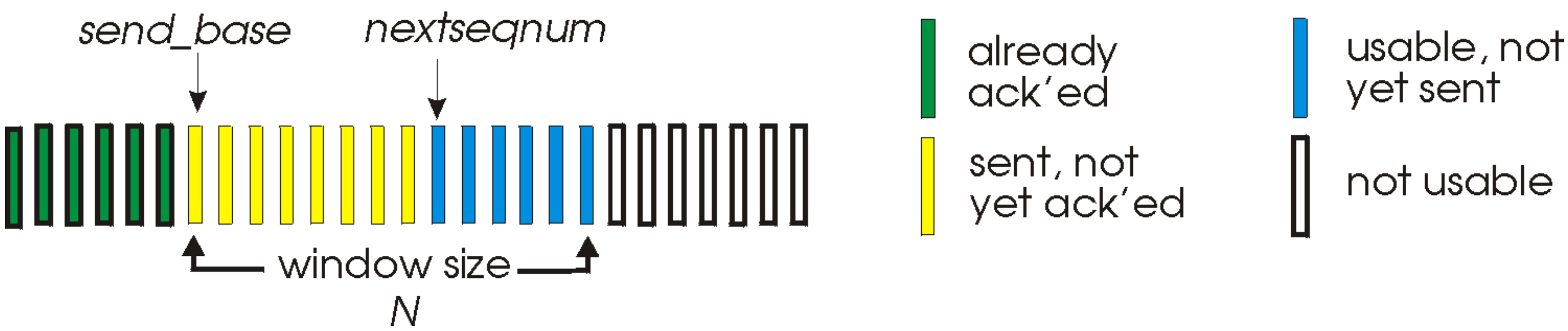


$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

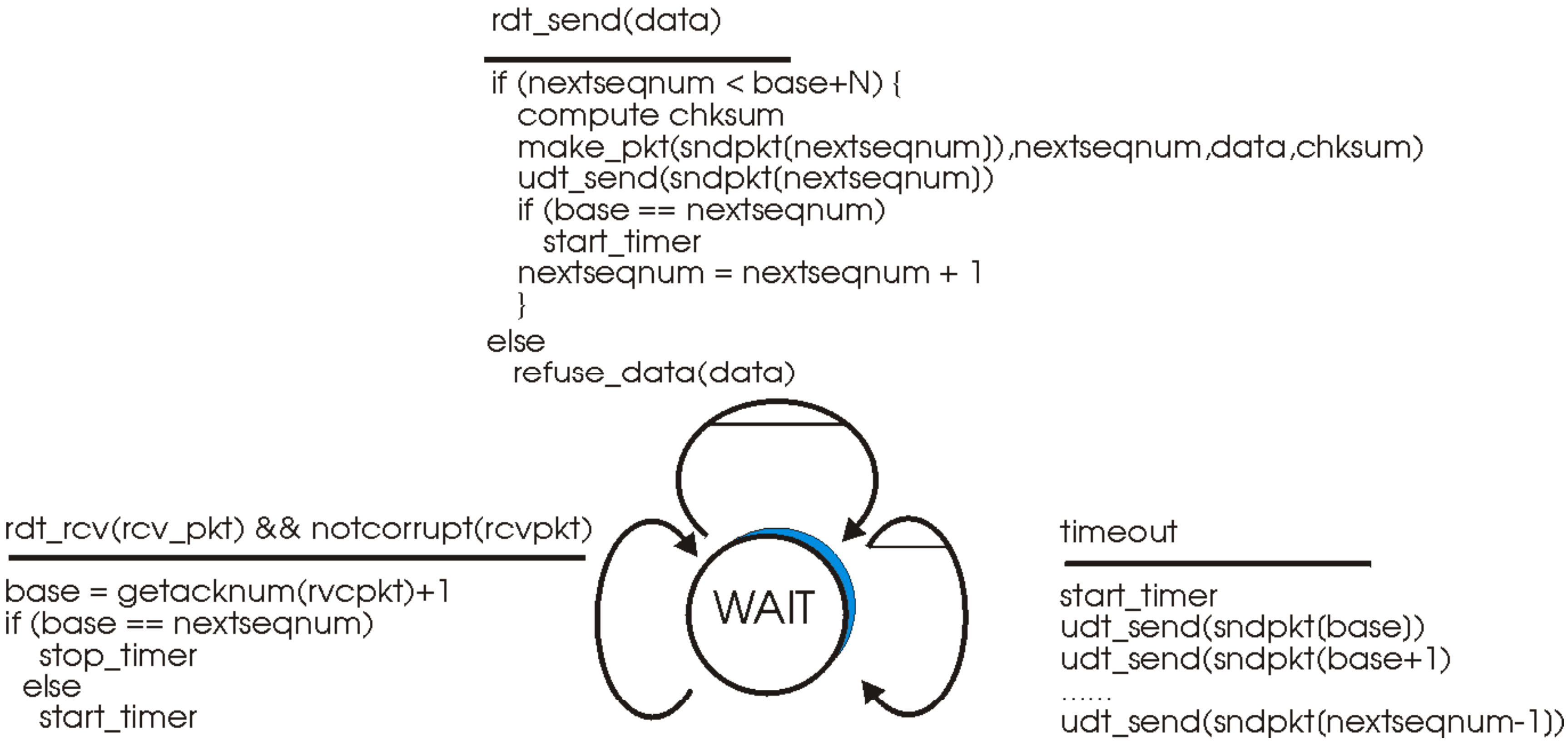
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

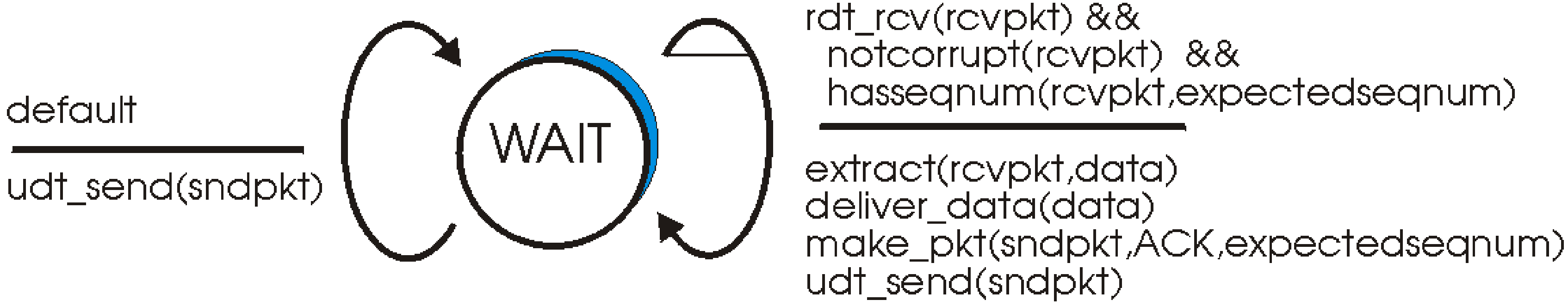


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



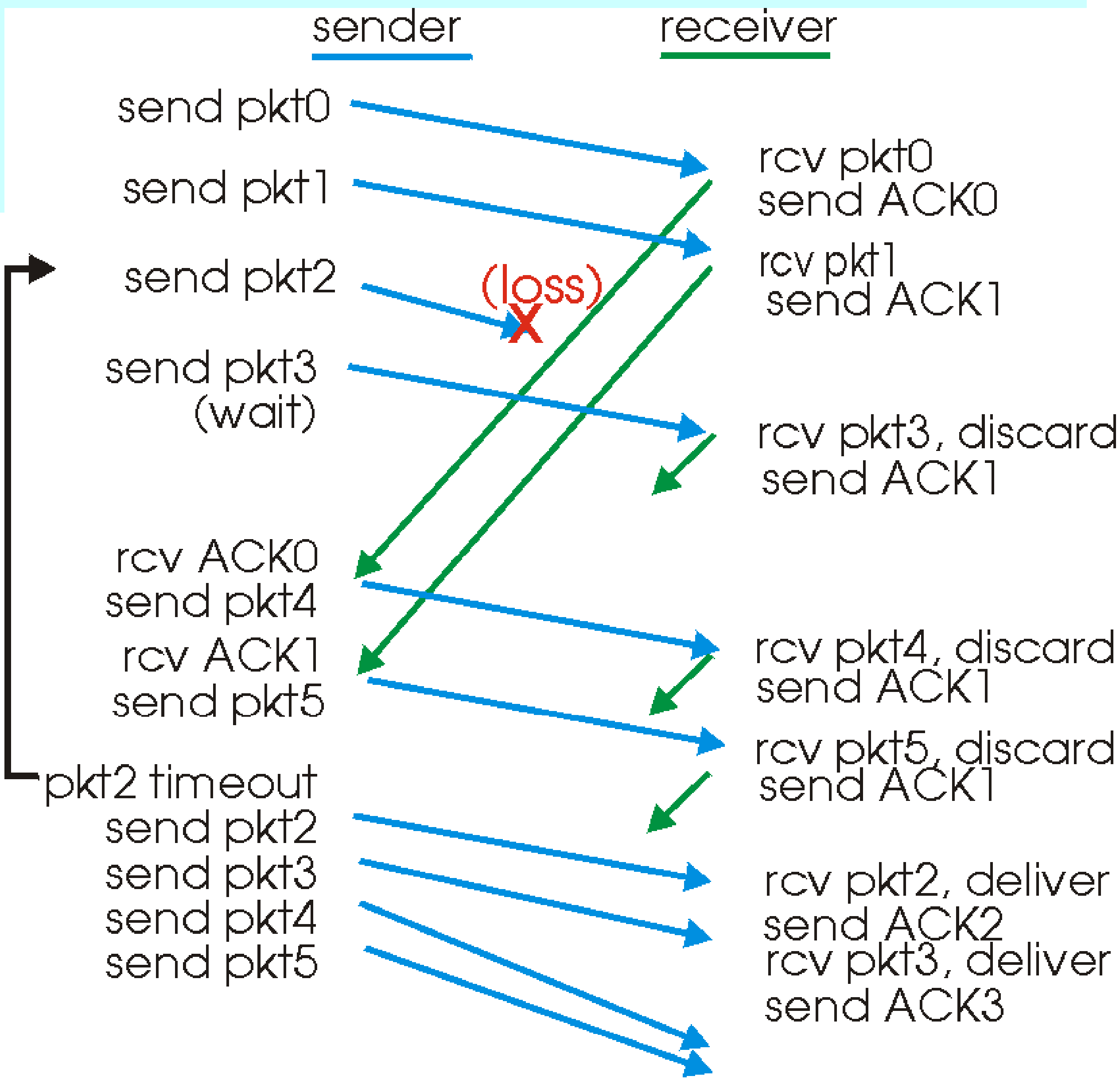
GBN: receiver extended FSM



receiver simple:

- 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!
  - ACK pkt with highest in-order seq #

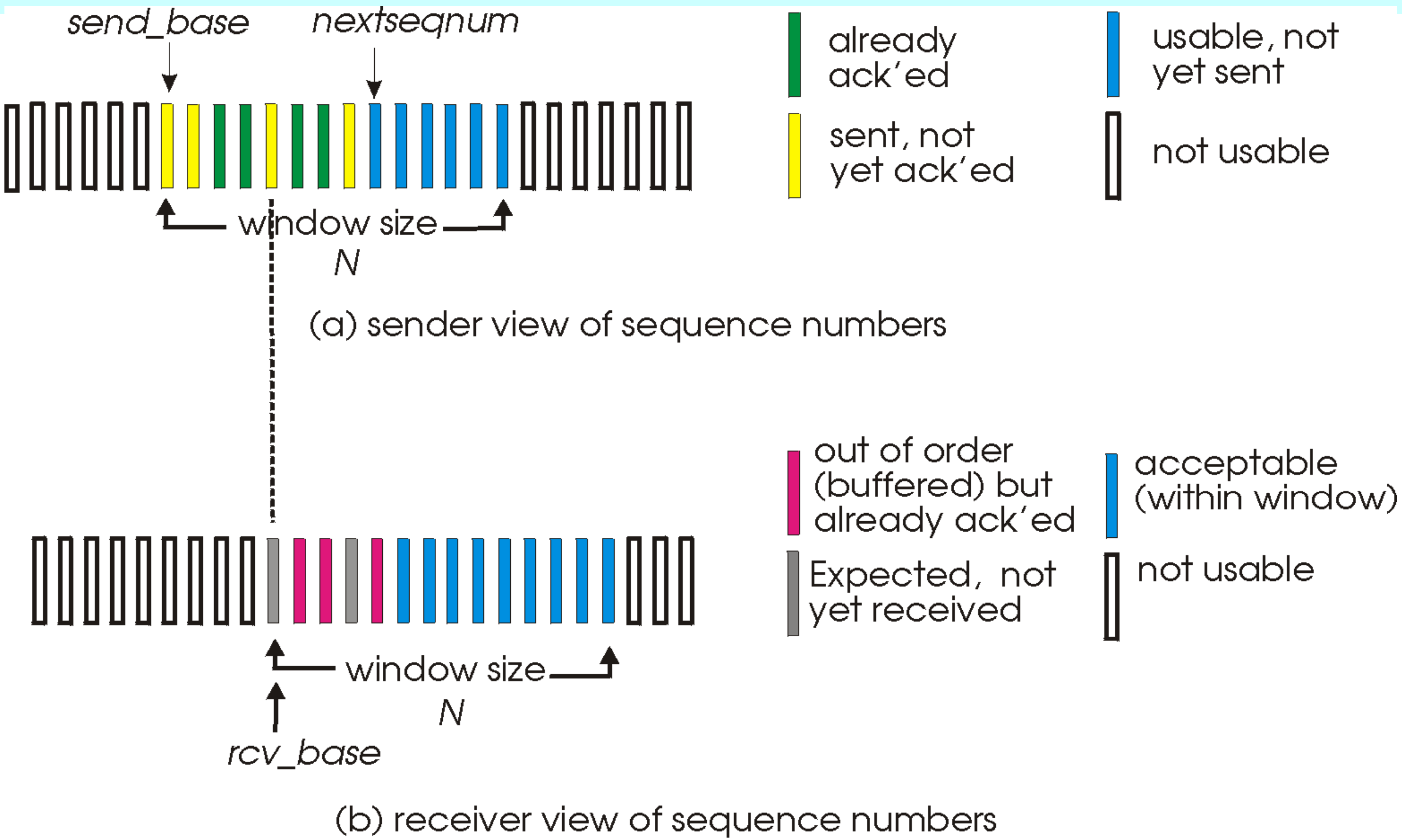
GBN in action



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
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

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

- 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

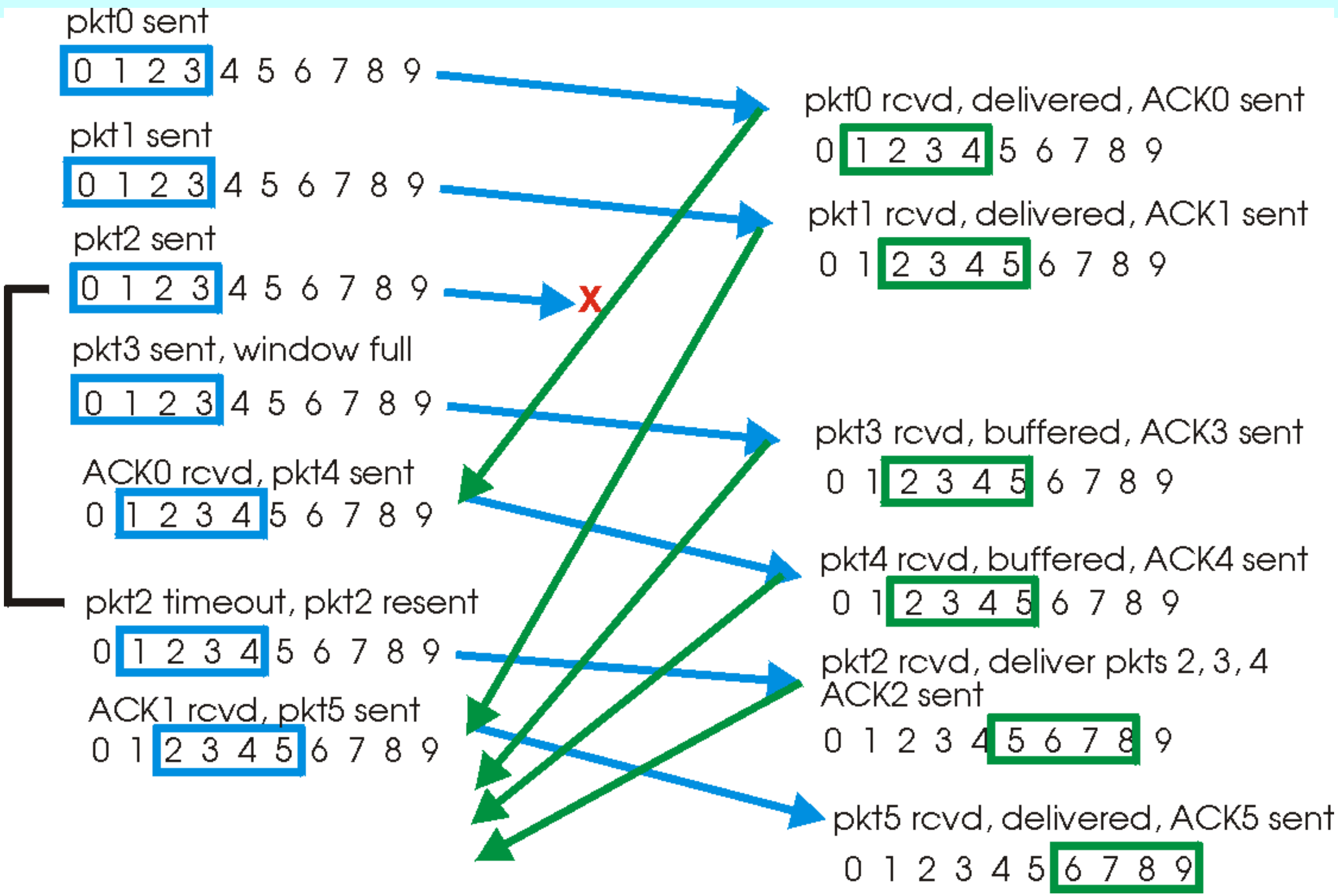
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



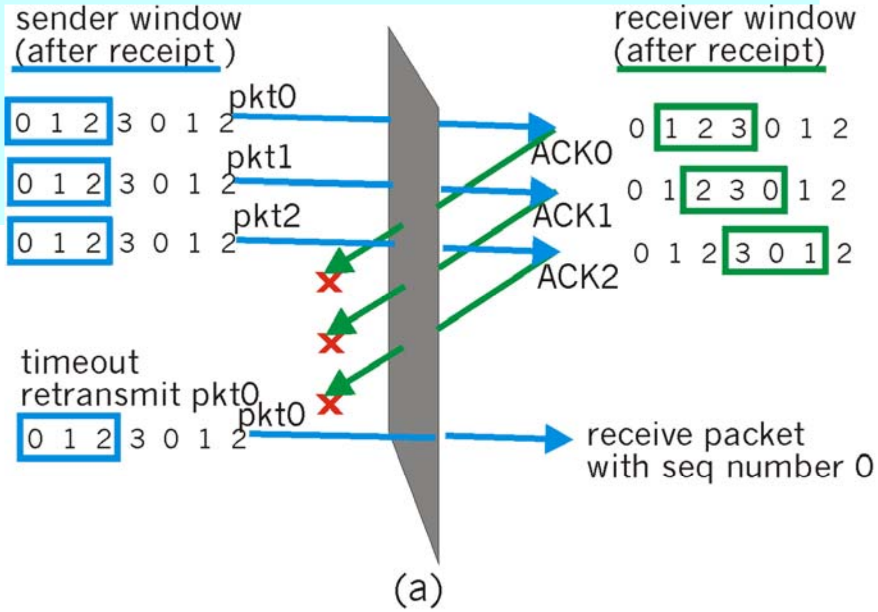
Selective repeat: dilemma

Example:

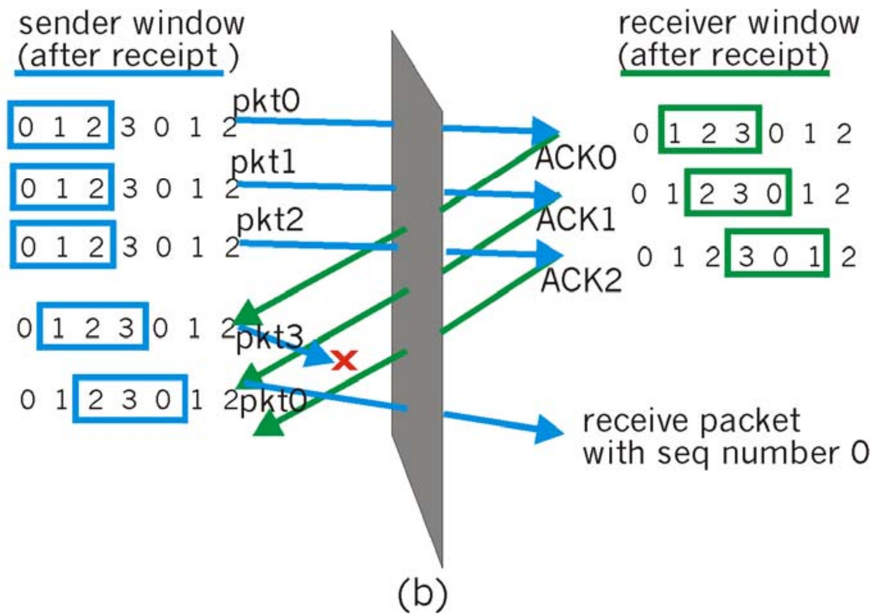
- seq #'s: 0, 1, 2, 3
- window size=3

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

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



(a)



(b)