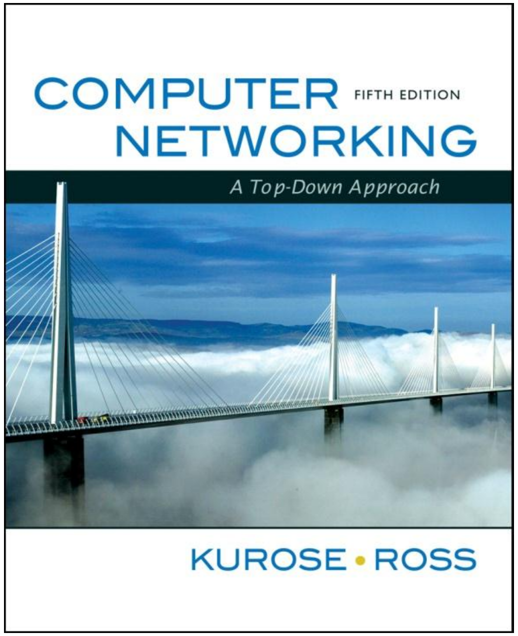
Chapter 3
Transport Layer



A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

Chapter 3: Transport Layer

Goals:

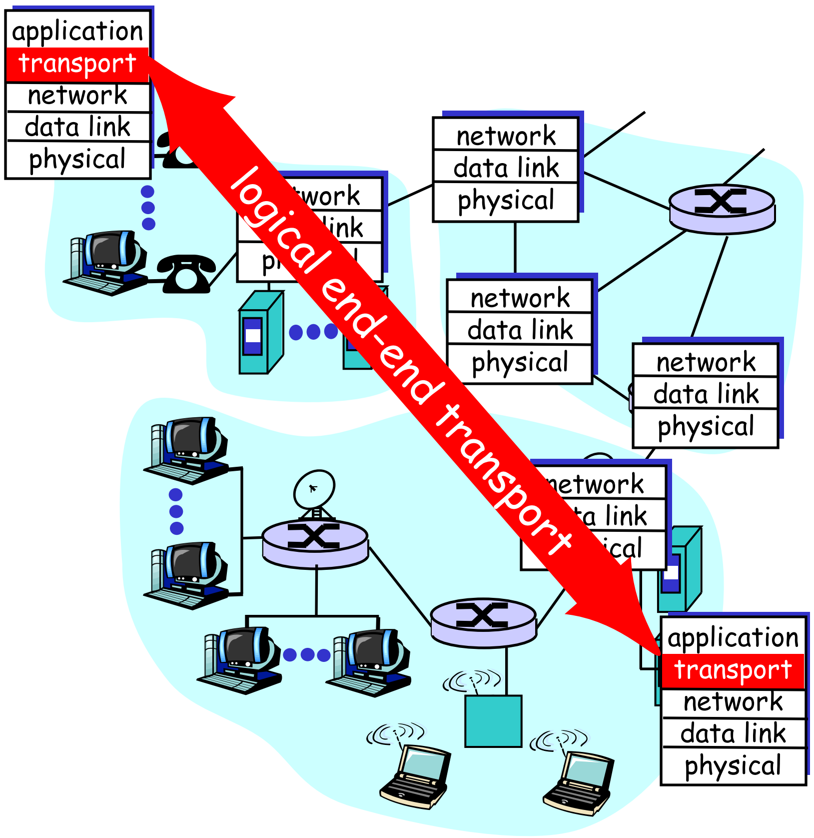
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion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

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

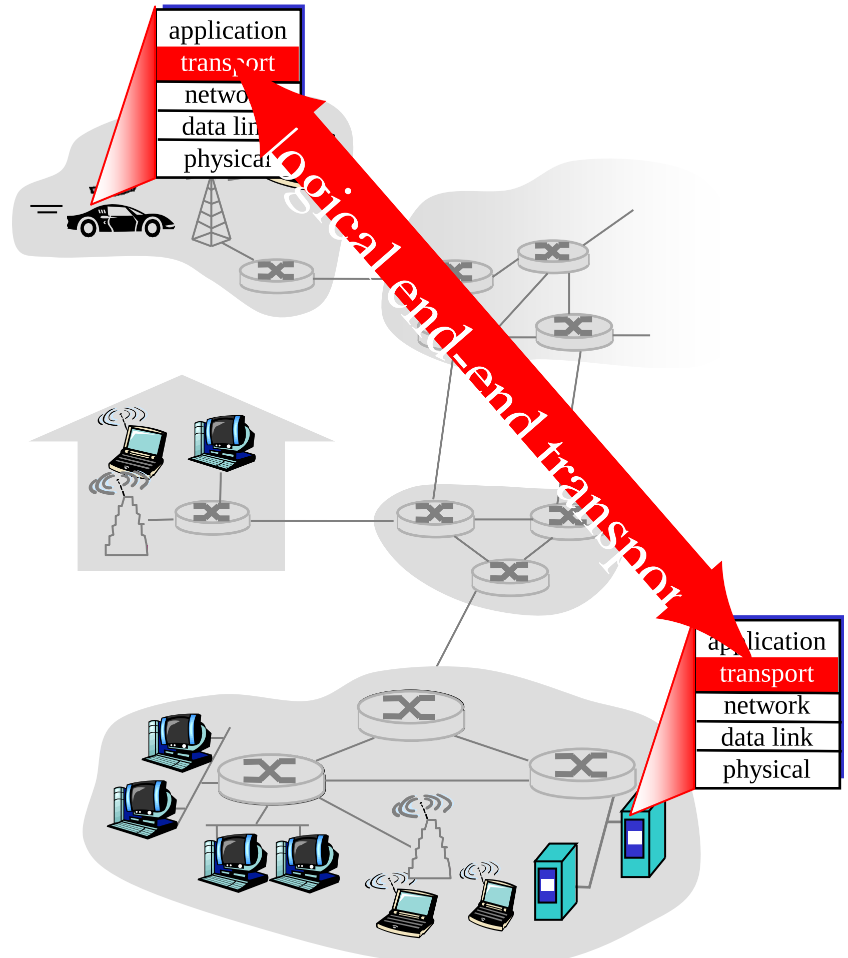
Transport Services and Protocols

- *Network layer service:* data transfer between end systems
- *Transport layer:* data transfer between processes running on different hosts
- Transport layer relies on, enhances, network layer services



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



Internet Transport-layer Protocols

- Reliable, in-order unicast delivery (TCP)
  - connection setup
  - flow control
  - congestion control
- Unreliable ("best-effort"), unordered unicast or multicast delivery (UDP)
- Services NOT available:
  - for real-time applications - delay bound requirement
  - bandwidth guarantees
  - reliable multicast

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

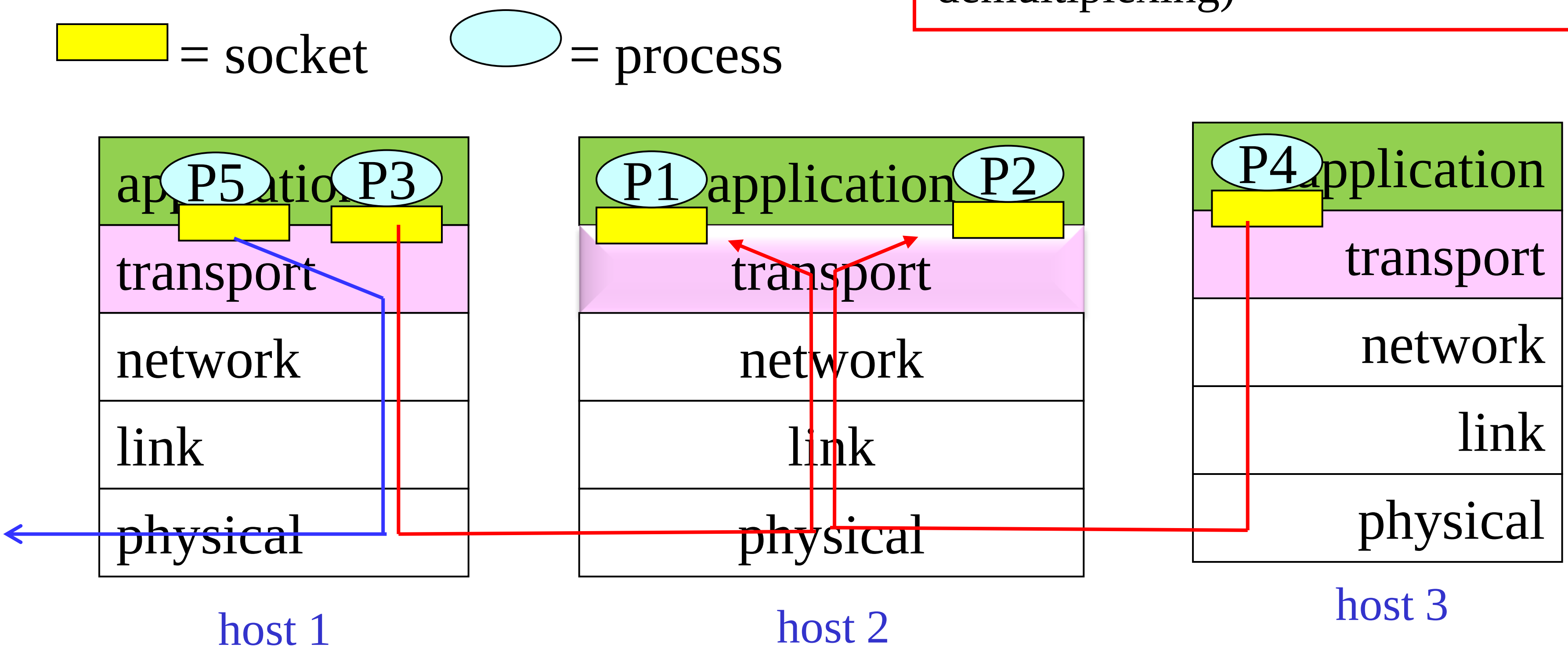
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

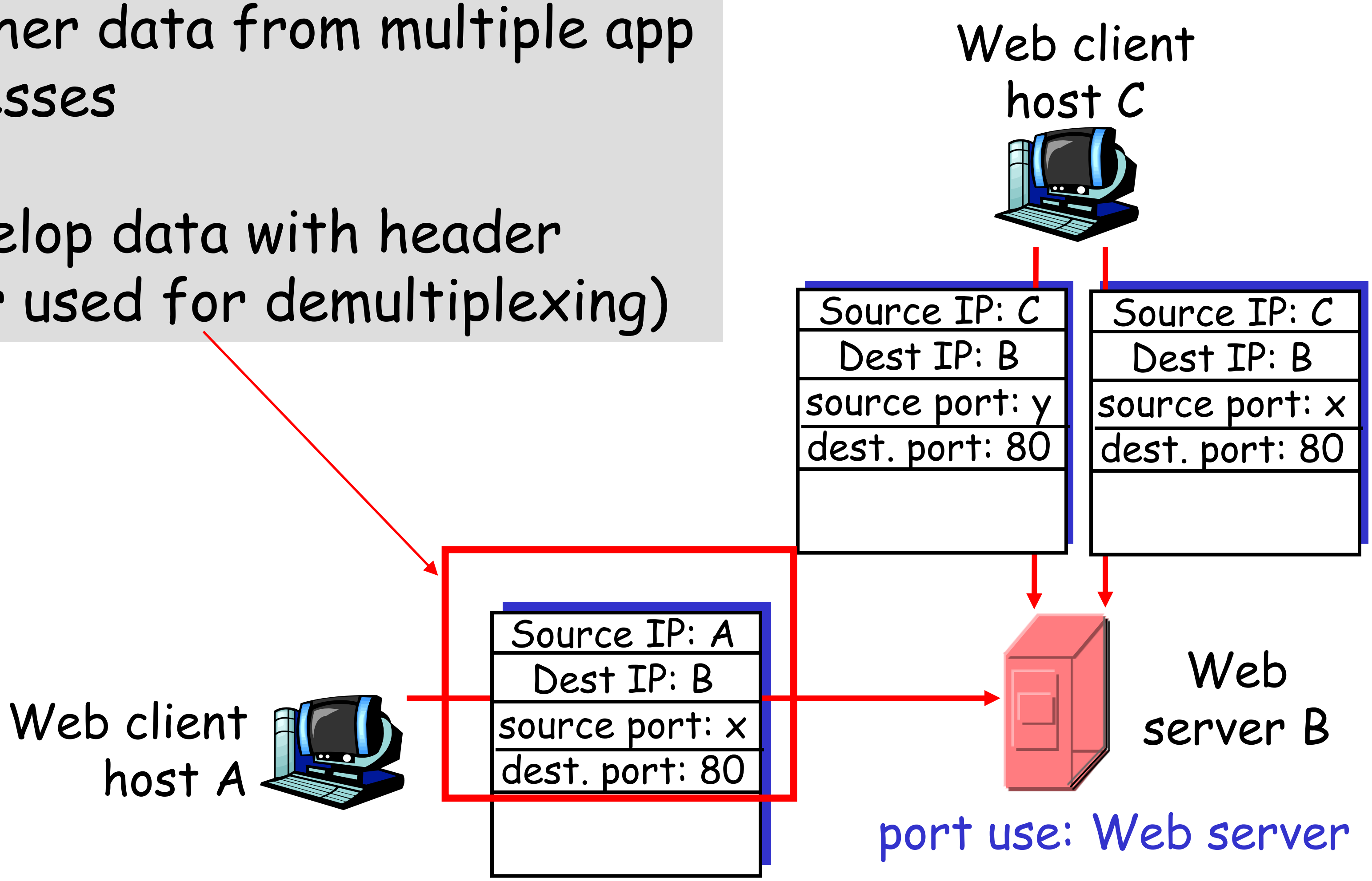
Multiplexing at send host:

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



Multiplexing

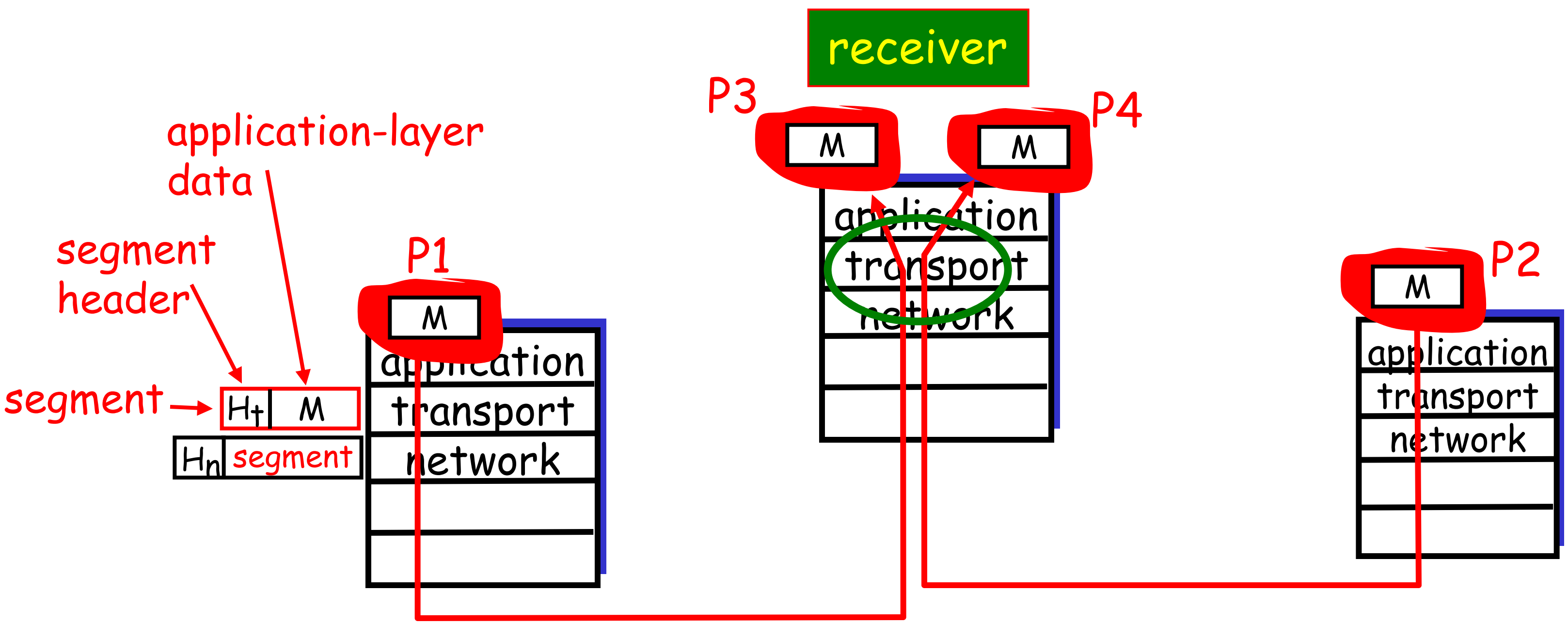
- Gather data from multiple app processes
- Envelop data with header (later used for demultiplexing)



Demultiplexing

- TPDU: transport protocol data unit

Demultiplexing: delivering received segments to correct app layer processes



UDP Datagram Dispatching

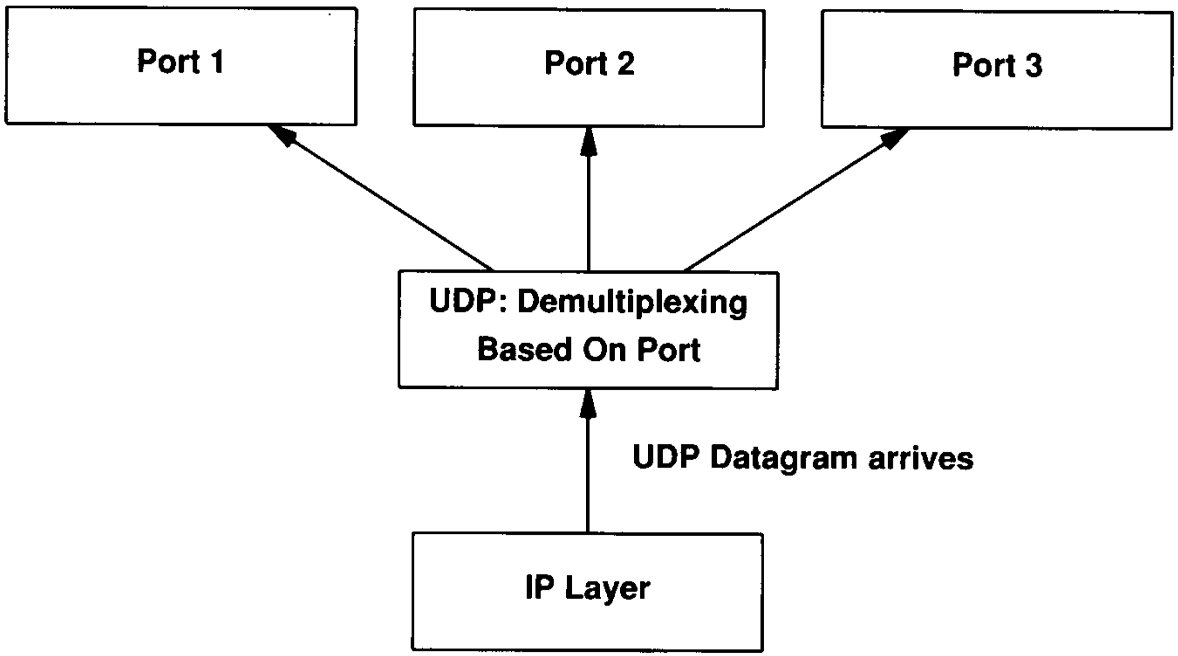
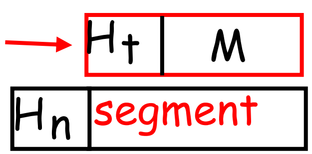
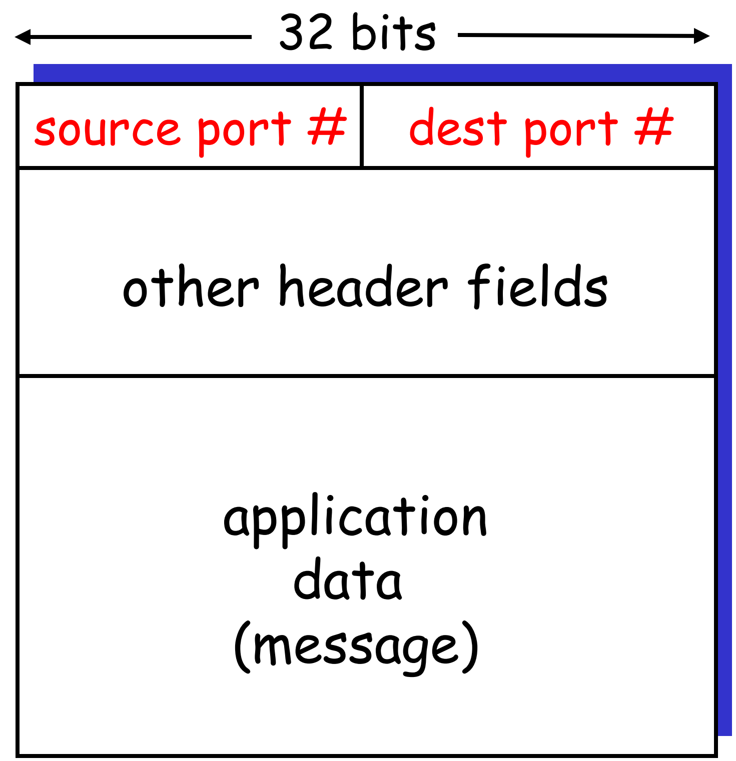


Figure 12.5 Example of demultiplexing one layer above IP. UDP uses the UDP destination port number to select an appropriate destination port for incoming datagrams.

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
- 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(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

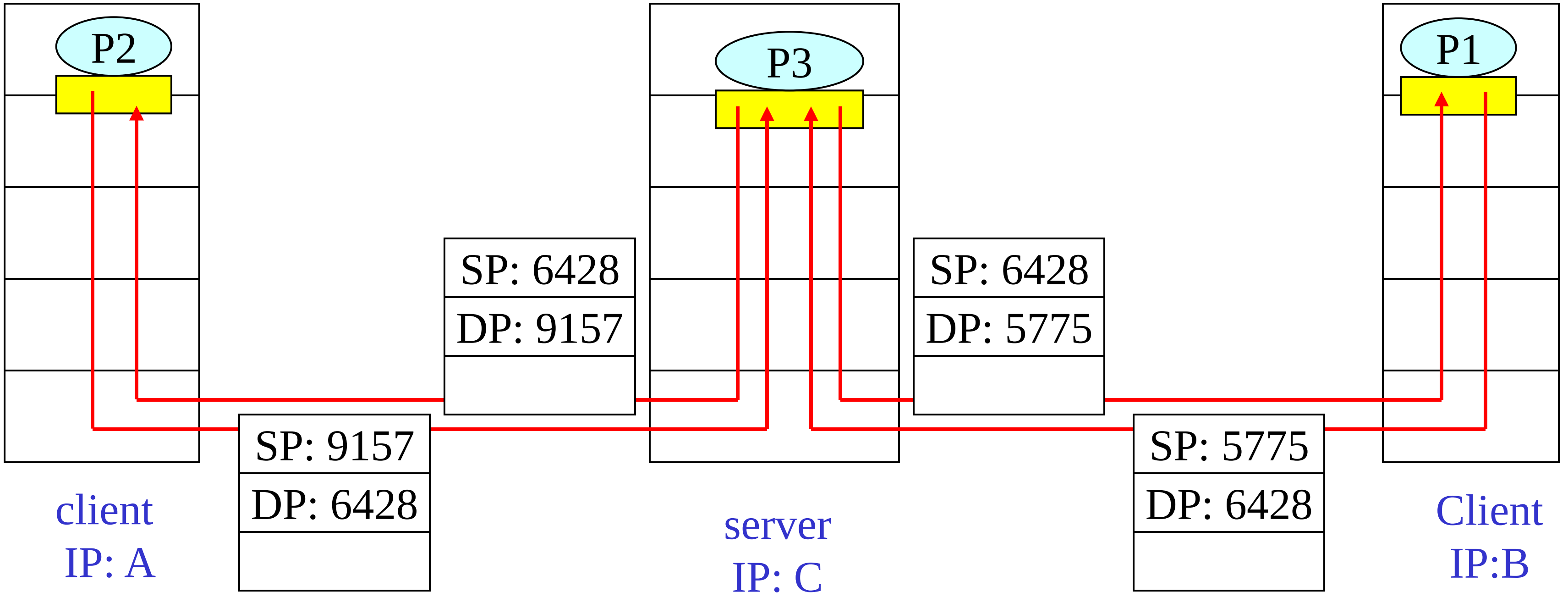
- UDP socket identified by two-tuple:

(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);
```

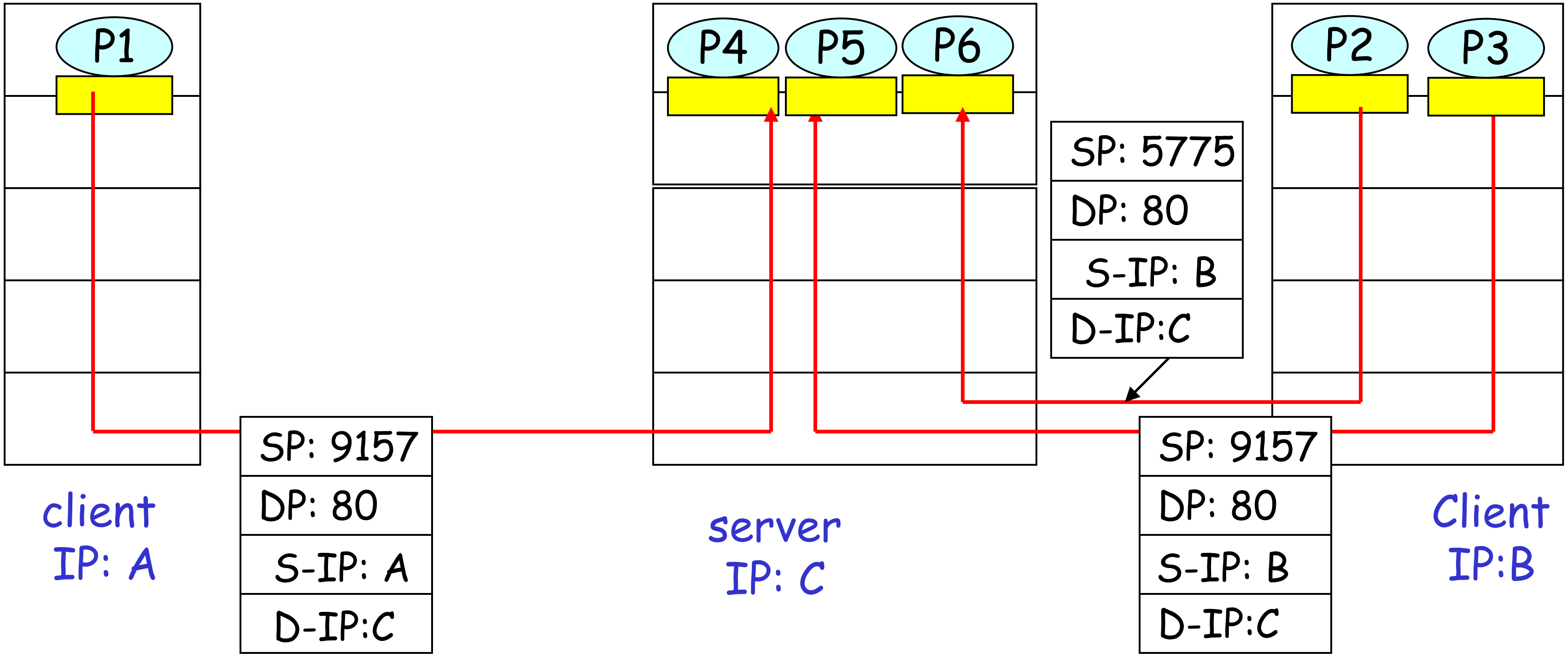


SP provides “return address”

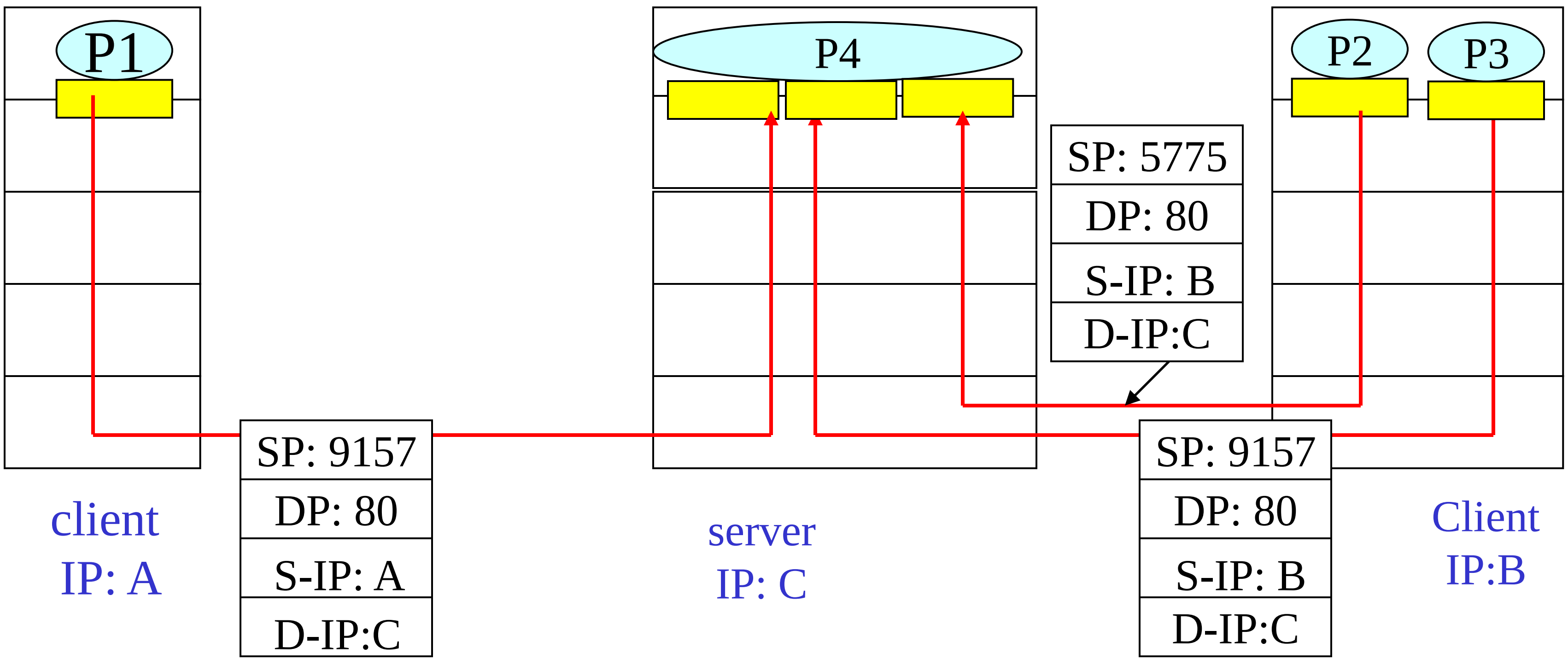
Connection-oriented demux

- 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



Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be
  - *Errored*
  - *Lost*
  - *Delayed*
  - *duplicated* or
  - *delivered out of order*
- Error detection, handling and recovery by the *upper layer applications*

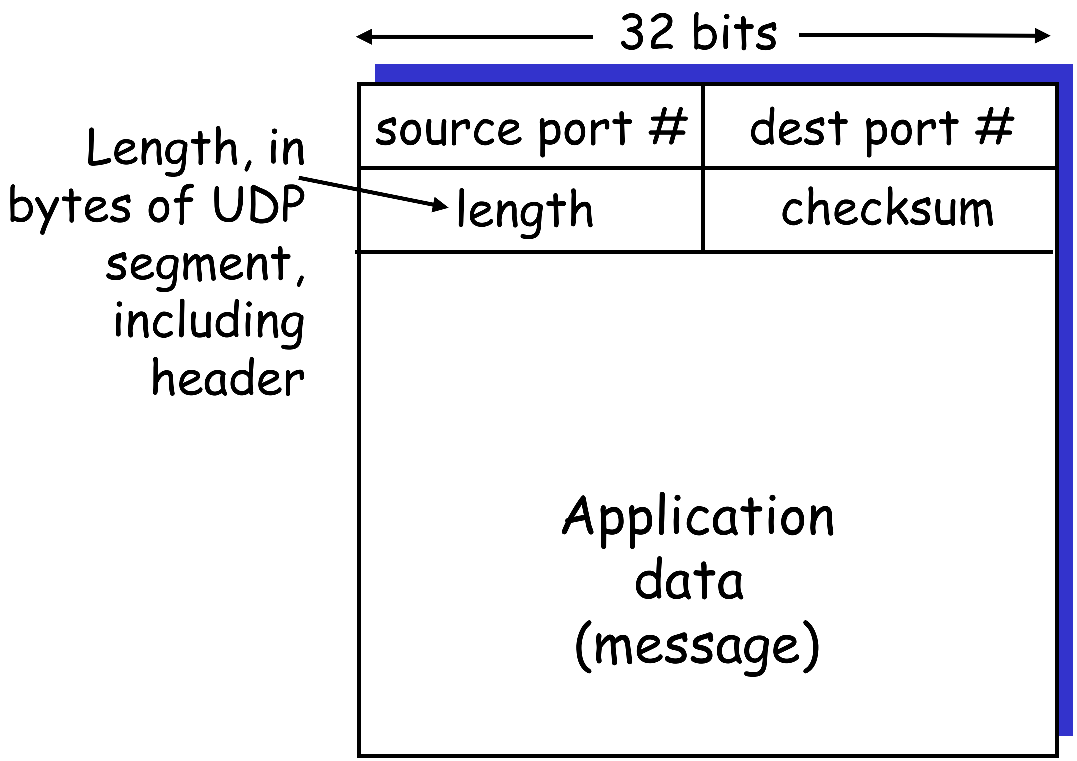
- *Connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

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

- Header - 8 bytes
- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses (why?):
  - DNS (53)
  - SNMP (161, 162)



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 nonetheless?* 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**

UDP Datagram Dispatching

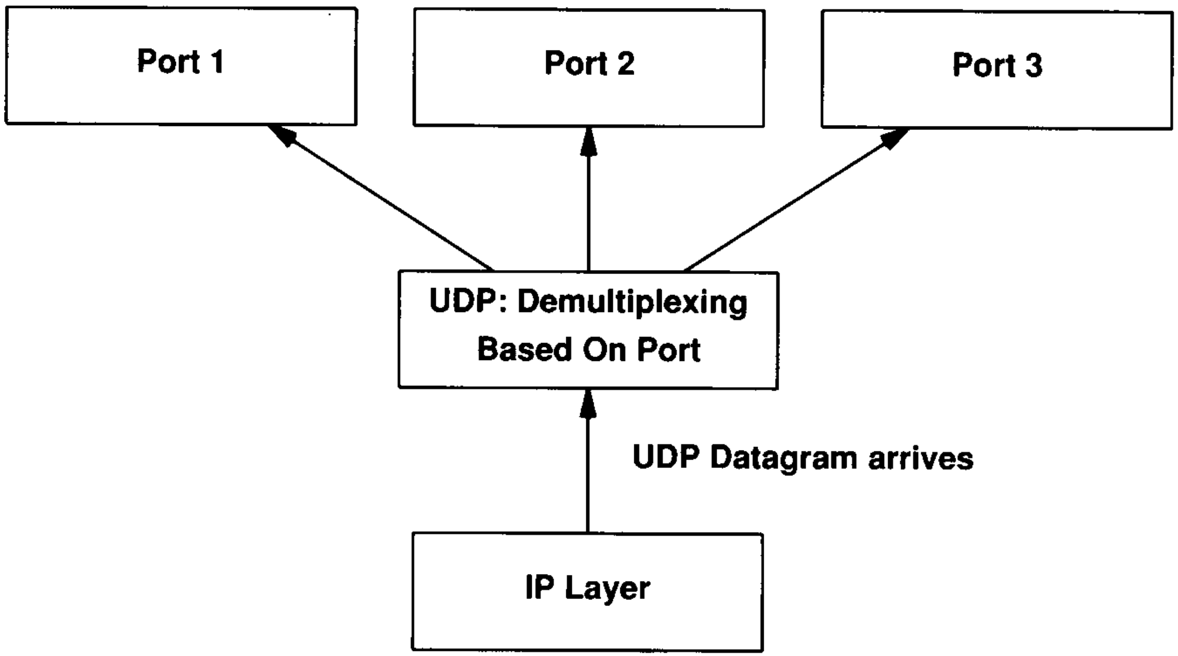


Figure 12.5 Example of demultiplexing one layer above IP. UDP uses the UDP destination port number to select an appropriate destination port for incoming datagrams.

UDP Ports Assigned

Decimal	Keyword	UNIX Keyword	Description
0	-	-	Reserved
7	ECHO	echo	Echo
9	DISCARD	discard	Discard
11	USERS	systat	Active Users
13	DAYTIME	daytime	Daytime
15	-	netstat	Who is up or NETSTAT
17	QUOTE	qotd	Quote of the Day
19	CHARGEN	chargen	Character Generator
37	TIME	time	Time
42	NAMESERVER	name	Host Name Server
43	NICNAME	whois	Who Is
53	DOMAIN	nameserver	Domain Name Server
67	BOOTPS	bootps	Bootstrap Protocol Serve
68	BOOTPC	bootpc	Bootstrap Protocol Client
69	TFTP	tftp	Trivial File Transfer
111	SUNRPC	sunrpc	Sun Microsystems RPC
123	NTP	ntp	Network Time Protocol
161	-	snmp	SNMP net monitor
162	-	snmp-trap	SNMP traps
512	-	biff	UNIX comsat
513	-	who	UNIX rwho daemon
514	-	syslog	system log
525	-	timed	Time daemon

Figure 12.6 An illustrative sample of currently assigned UDP ports showing the standard keyword and the UNIX equivalent; the list is not exhaustive. To the extent possible, other transport protocols that offer identical services use the same port numbers as UDP.

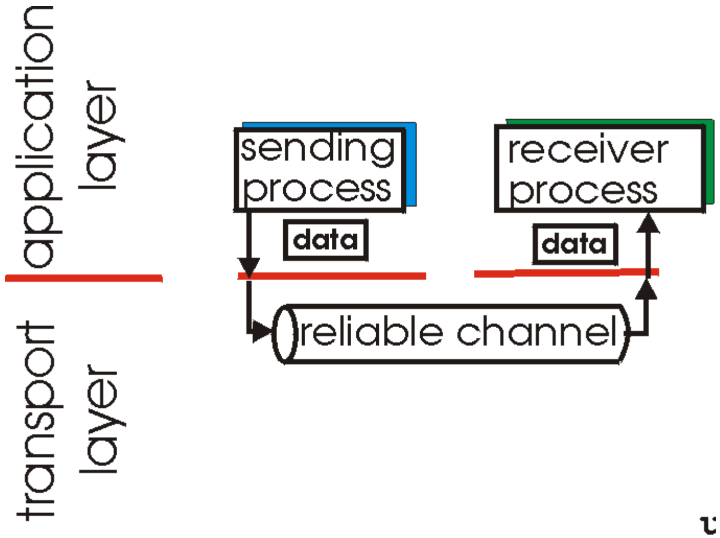
7 Echo
53 DNS
69 TFTP
123 NTP
161 SNMP
162 SNMP-trap

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Principles of Reliable data transfer

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

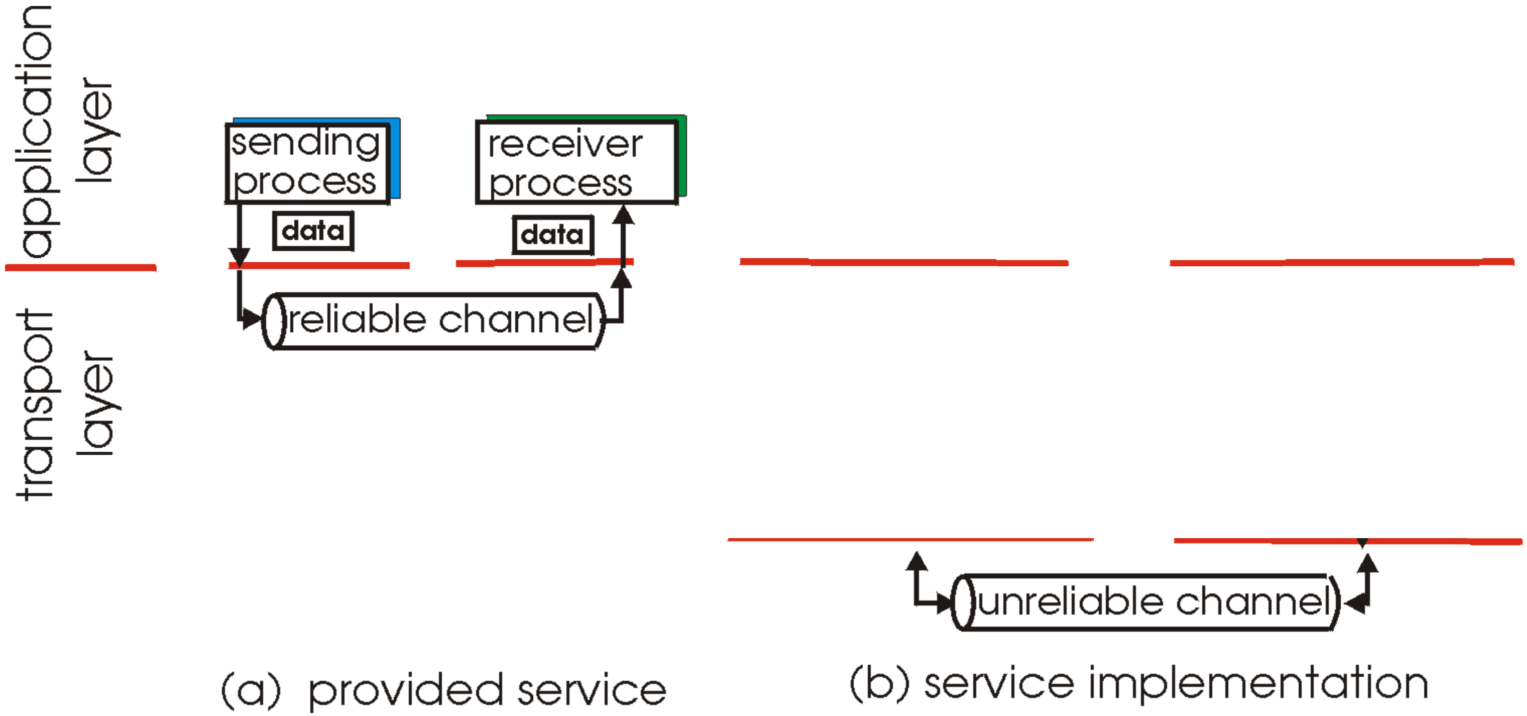


(a) provided service

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

Principles of Reliable data transfer

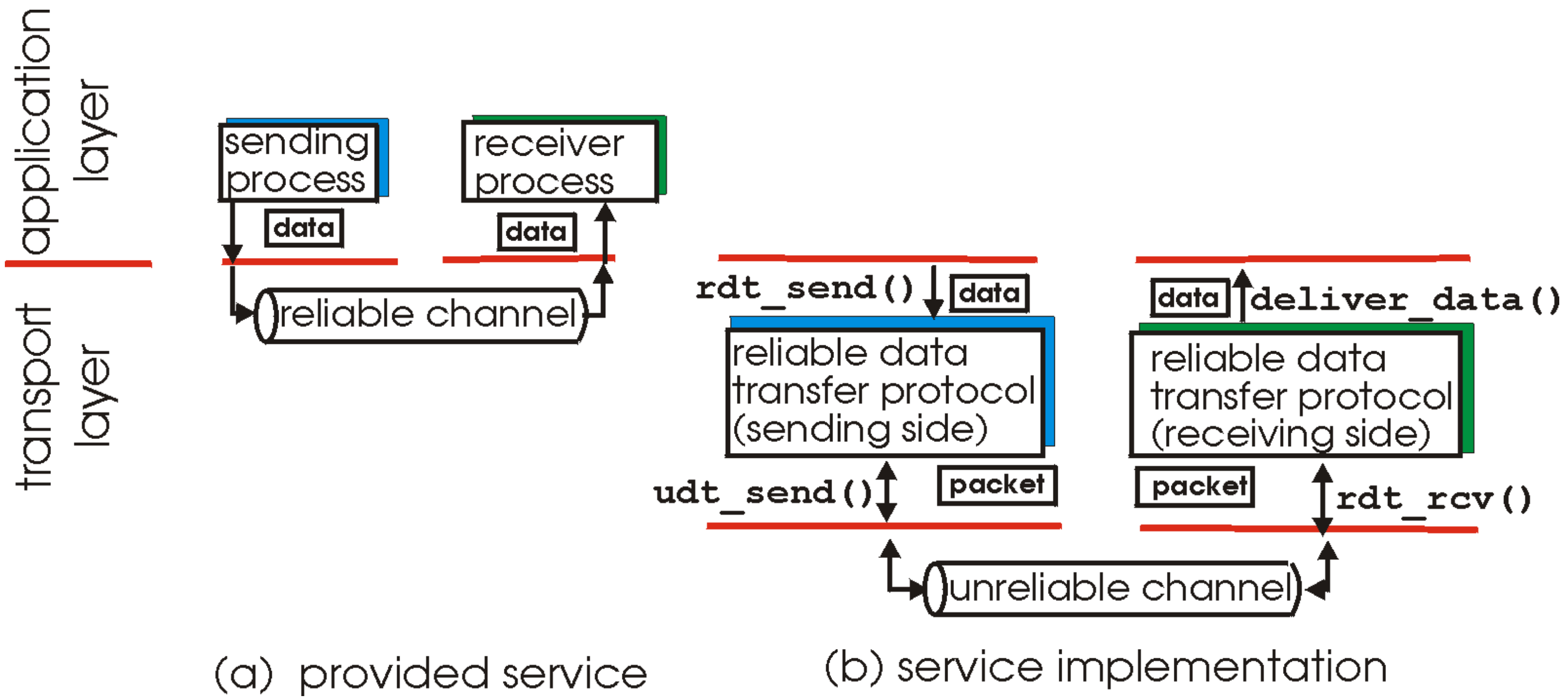
- important in app., transport, link layers
- top-10 list of important networking topics!



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

Principles of Reliable data transfer

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



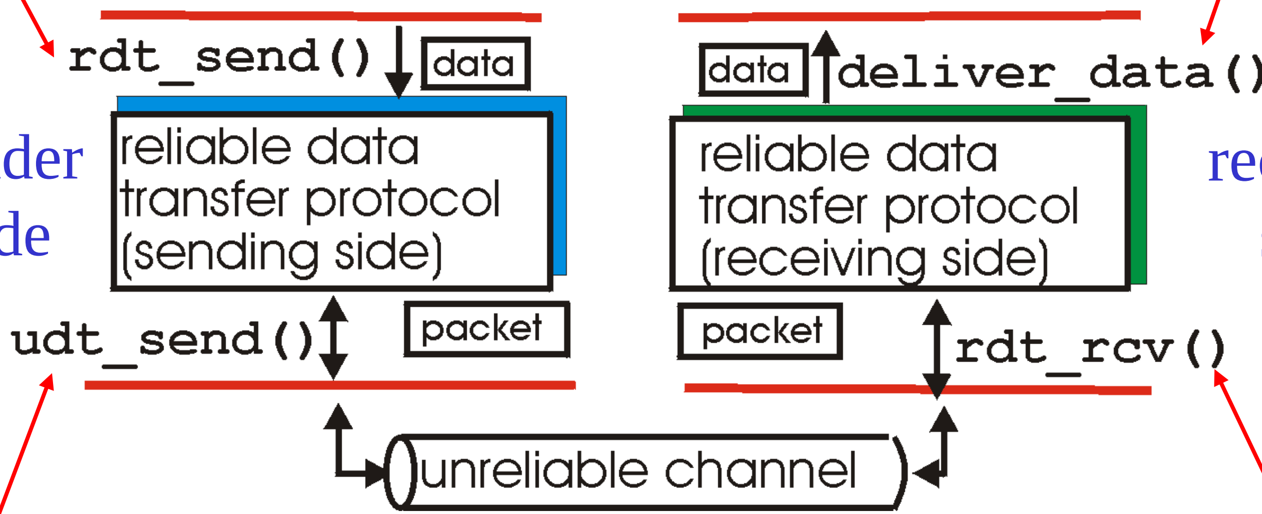
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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

sender side



receiver side

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

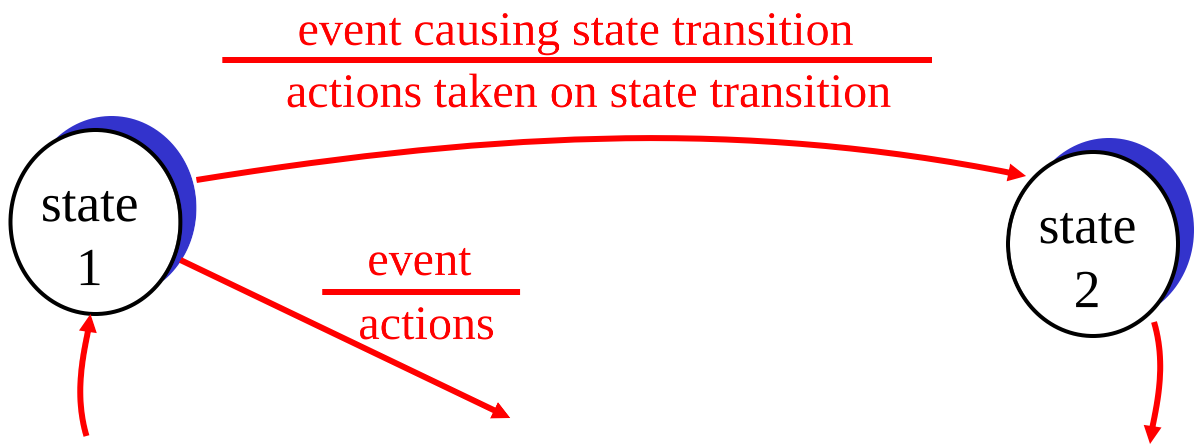
rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

We'll:

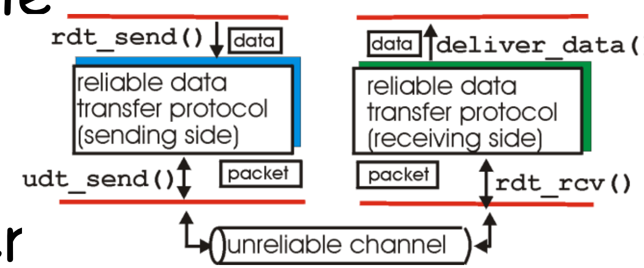
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

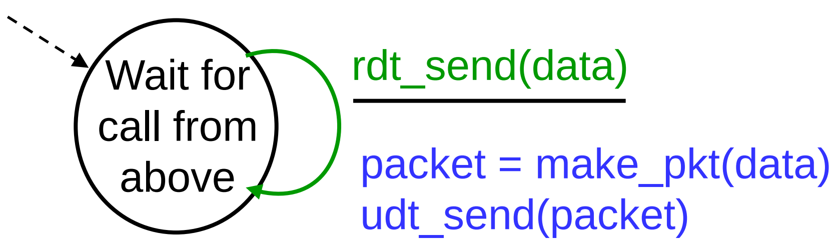
state: when in this "state" next state uniquely determined by next event



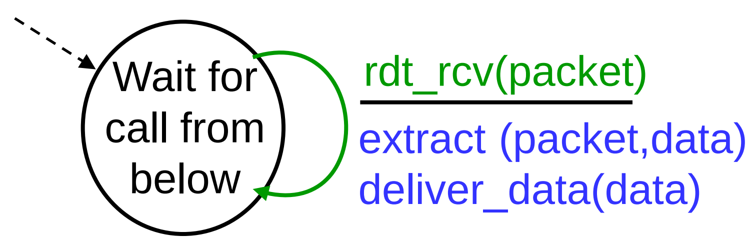
Rdt1.0: reliable transfer over a reliable channel

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





sender

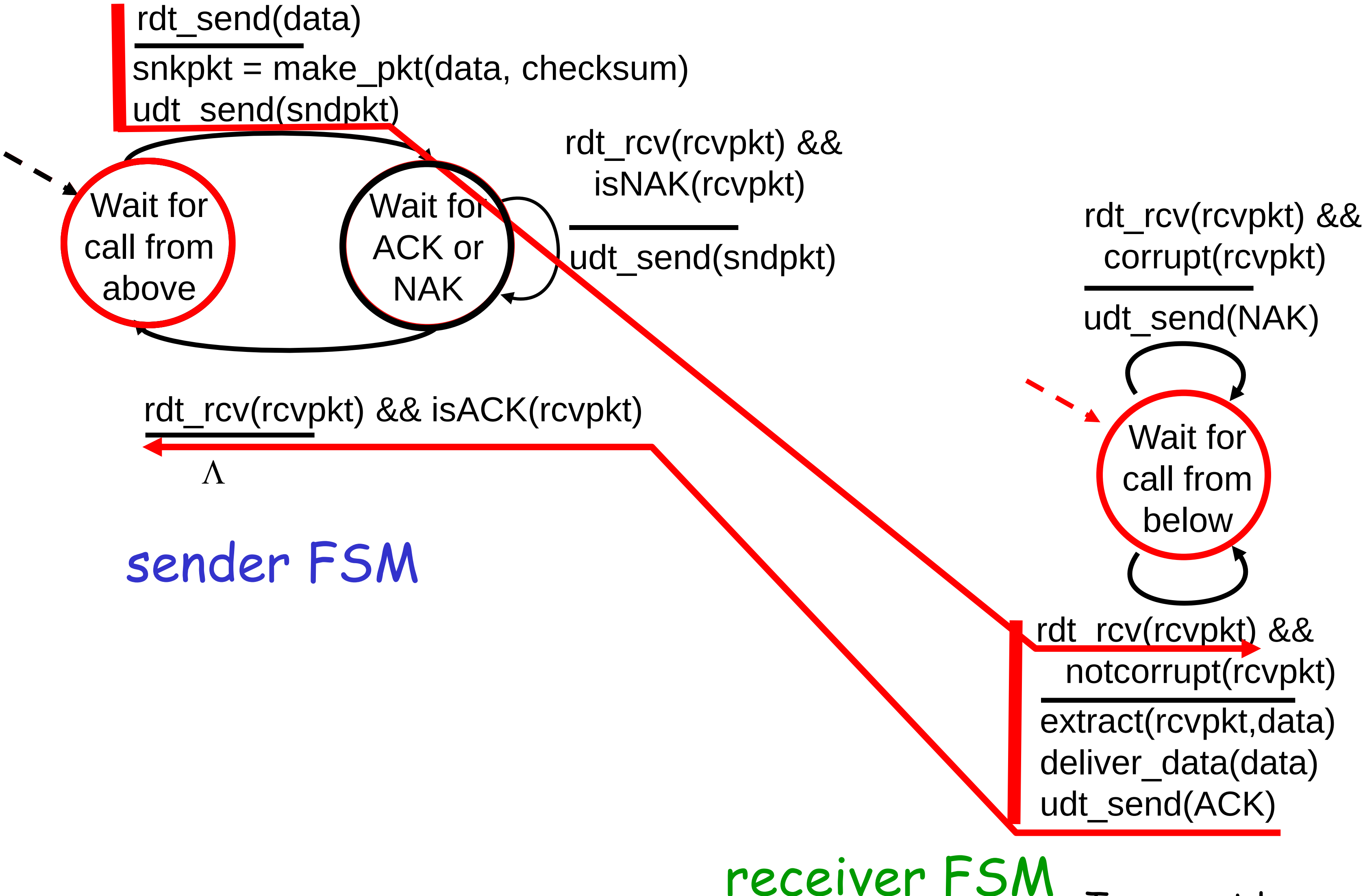


receiver

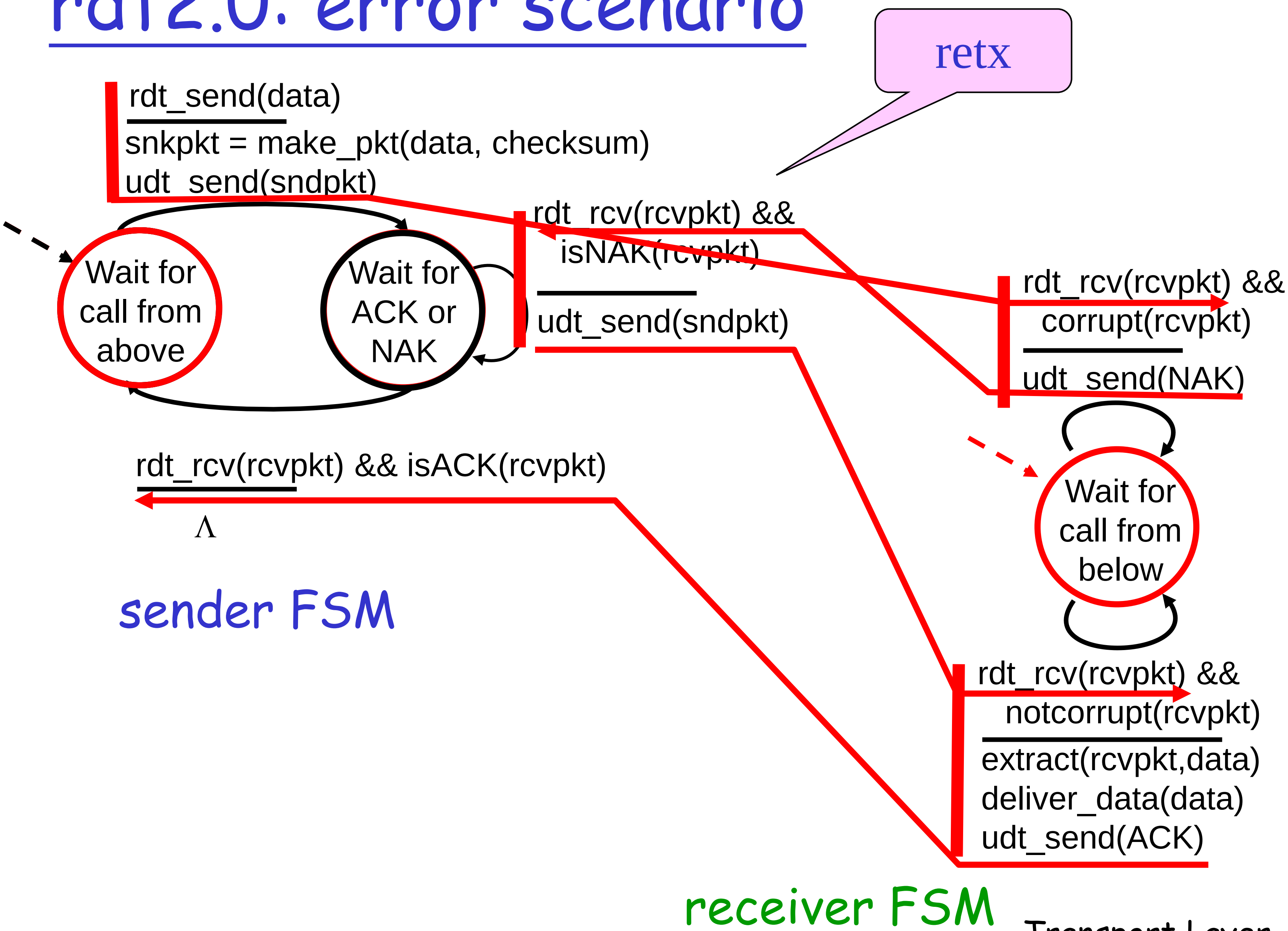
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - 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
  - receiver feedback: control msgs (ACK,NAK), rcvr->sender

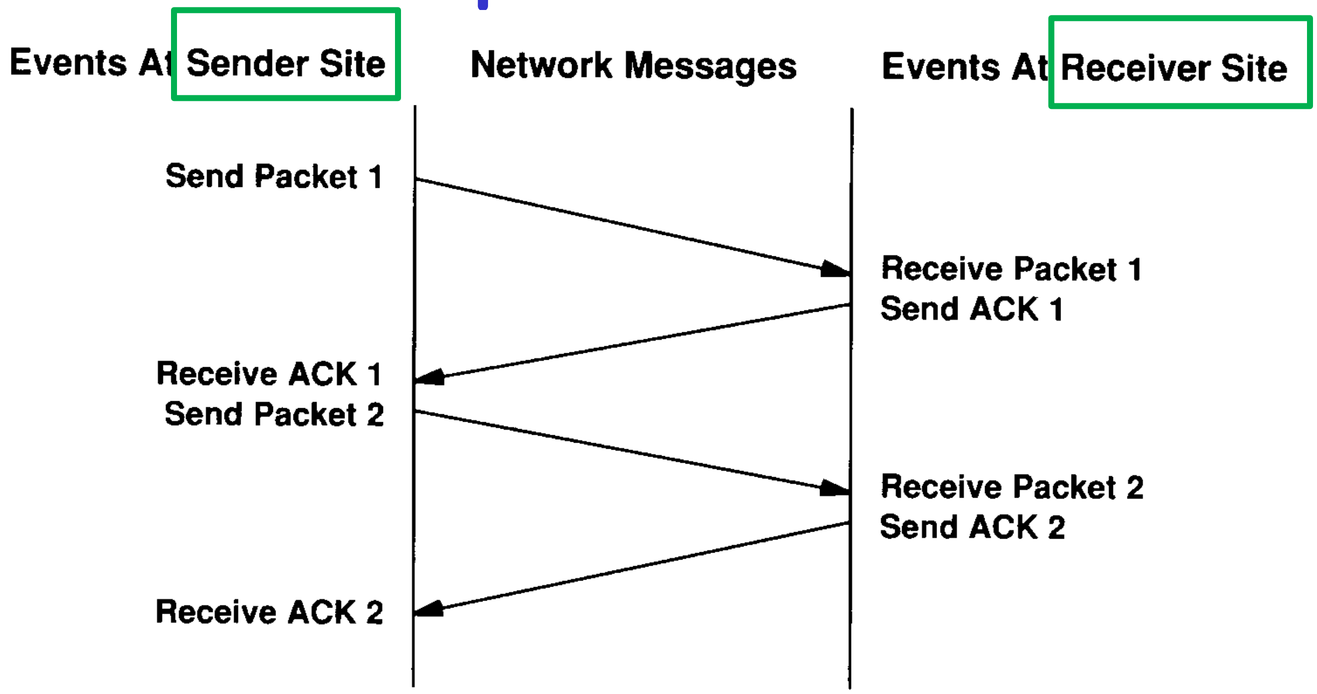
rdt2.0: operation with no errors



rdt2.0: error scenario

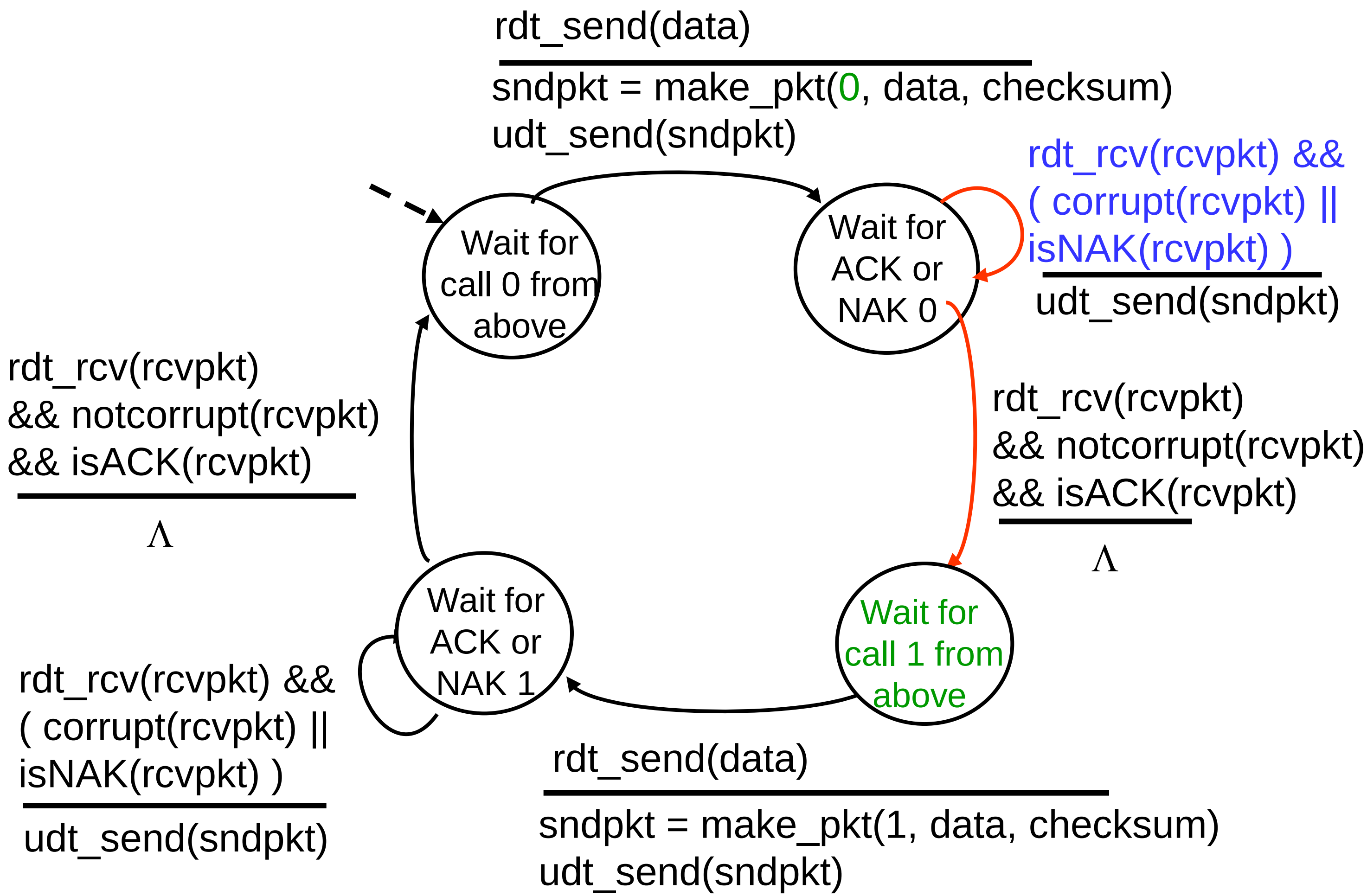


Stop-and-Wait

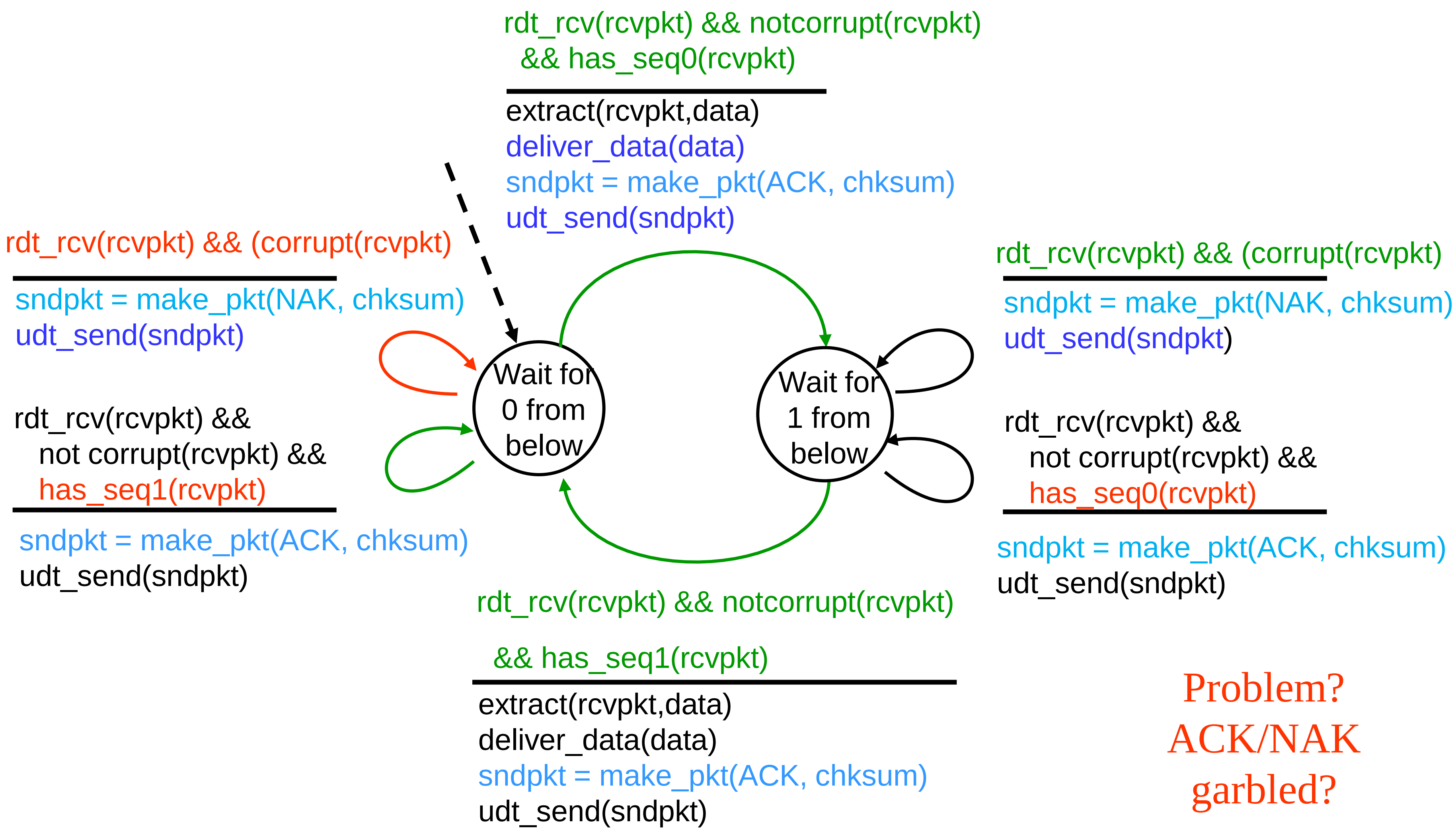


Sender sends a packet and waits for its ack before sending the next one

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



Problem?
ACK/NAK
garbled?

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. #

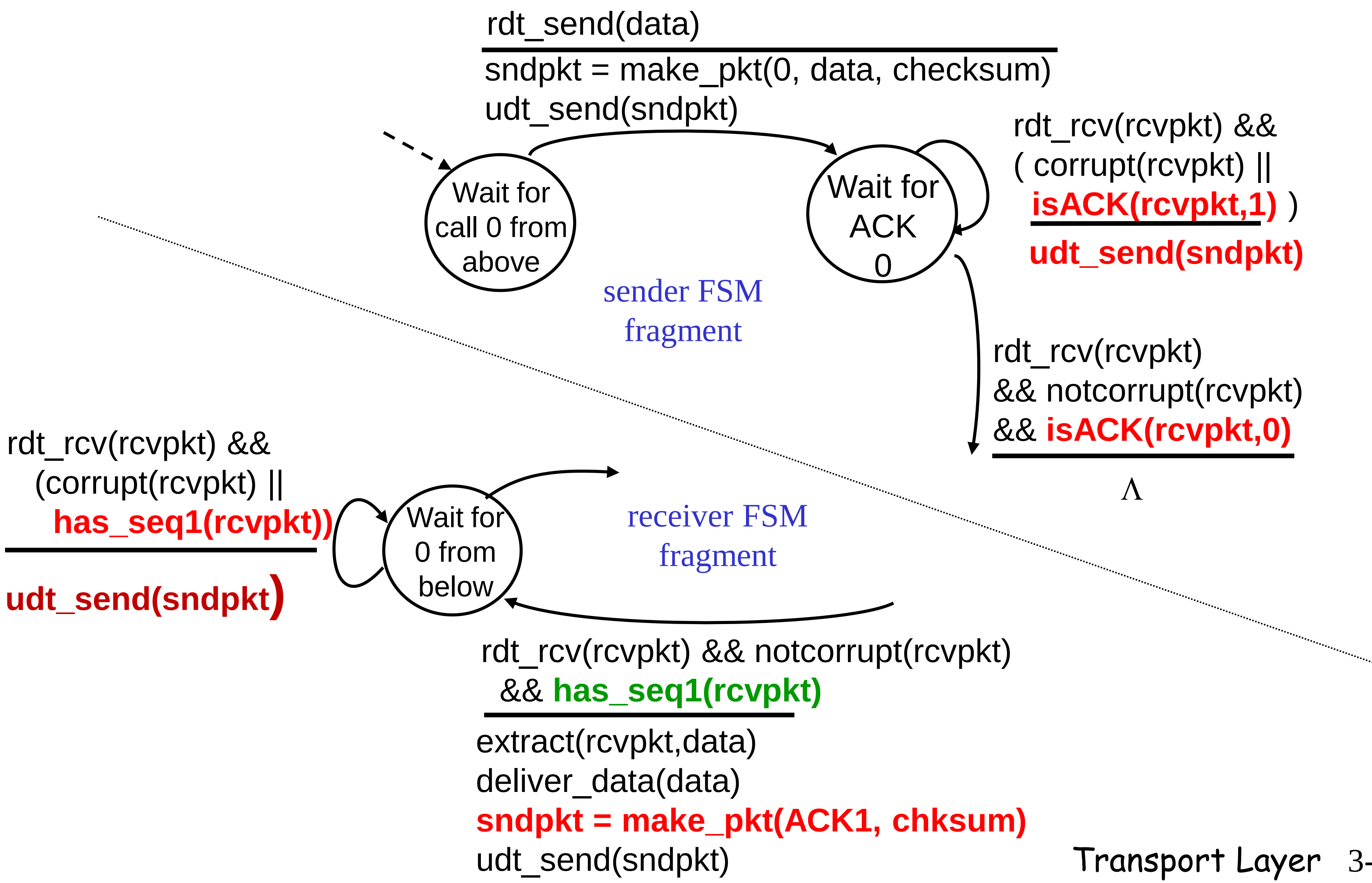
Receiver:

- 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

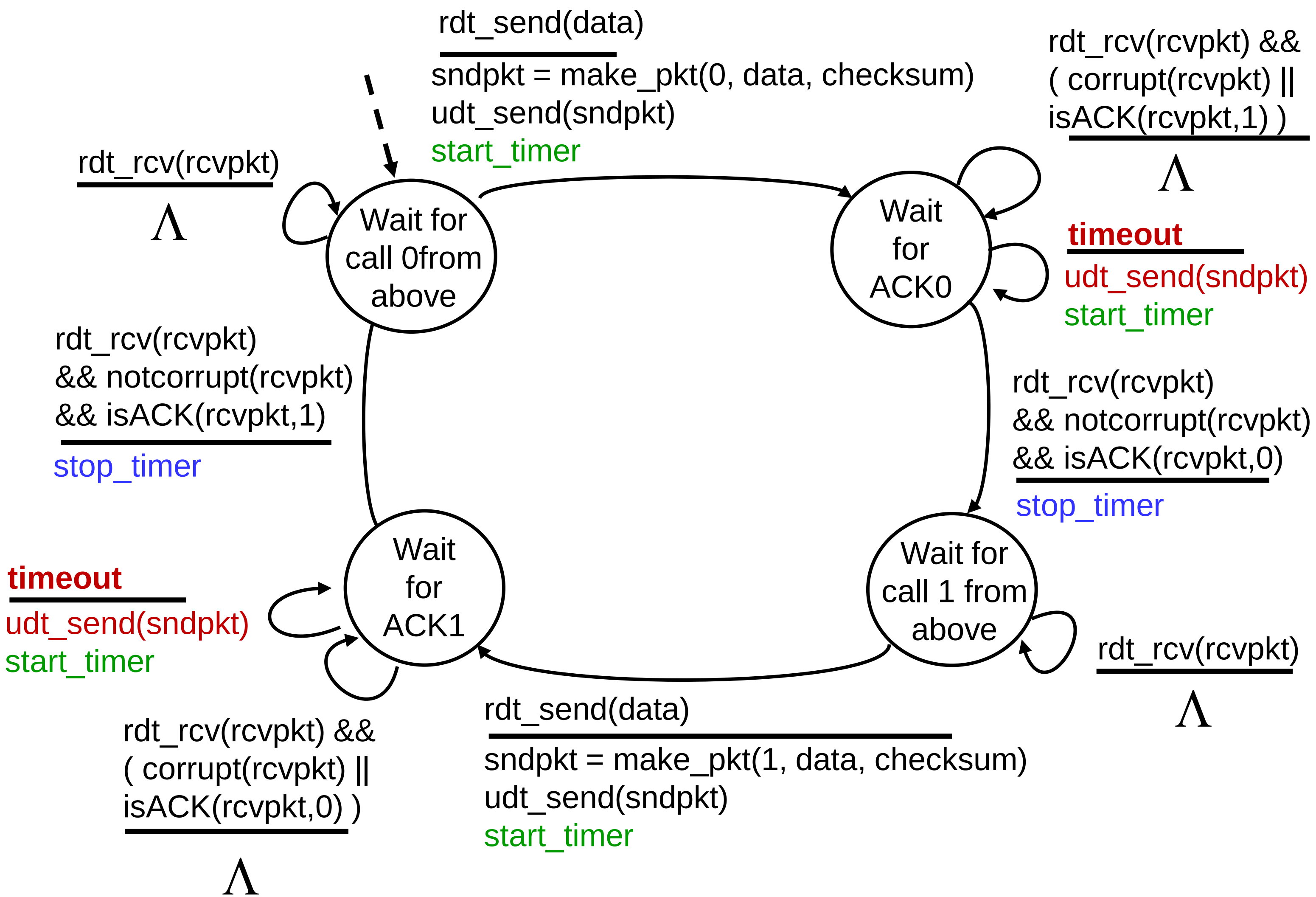
New assumption: underlying channel can also lose packets (data or ACKs)

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

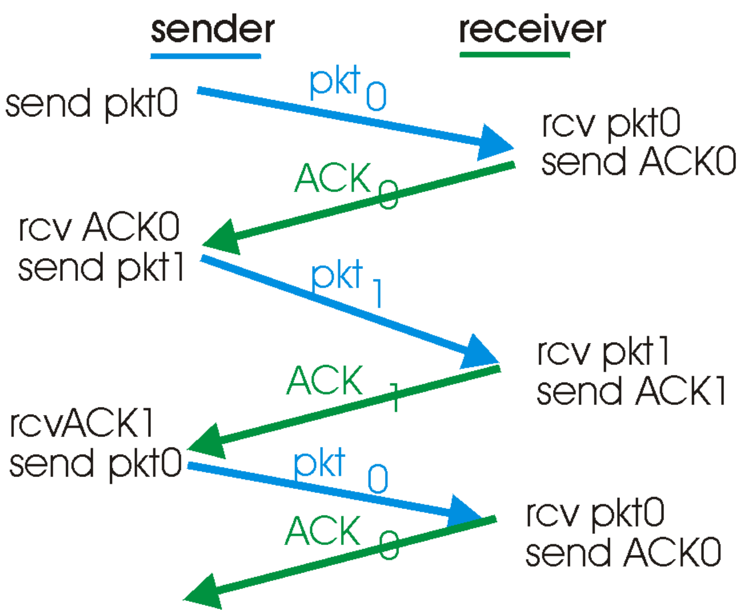
Approach: sender waits "reasonable" amount of time for ACK

- 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

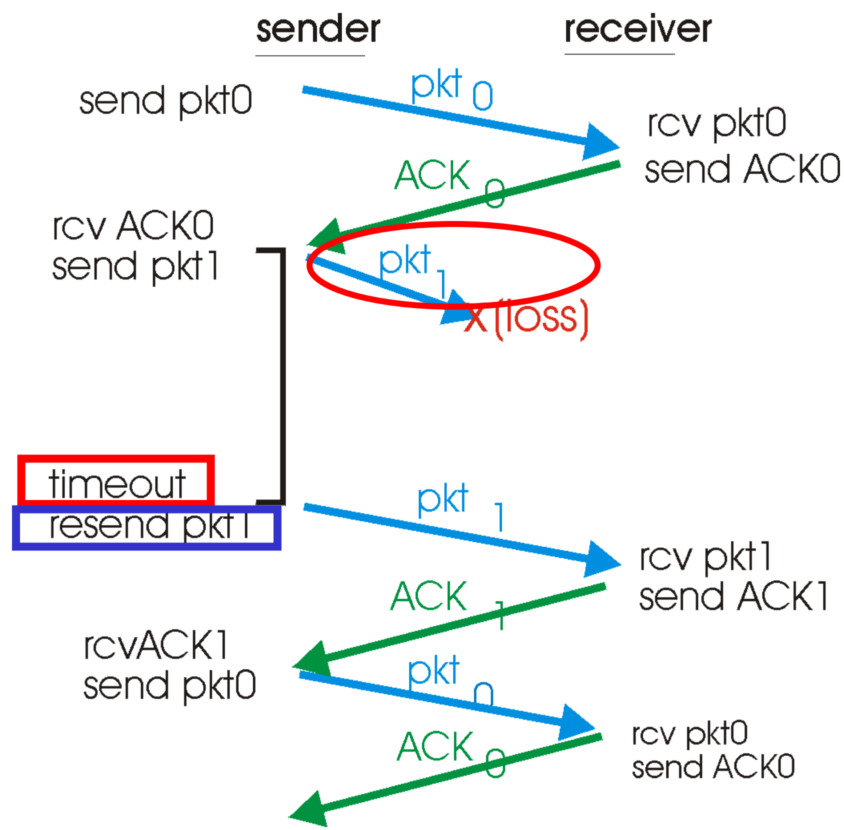
rdt3.0 sender



rdt3.0: Timer-based Retransmission

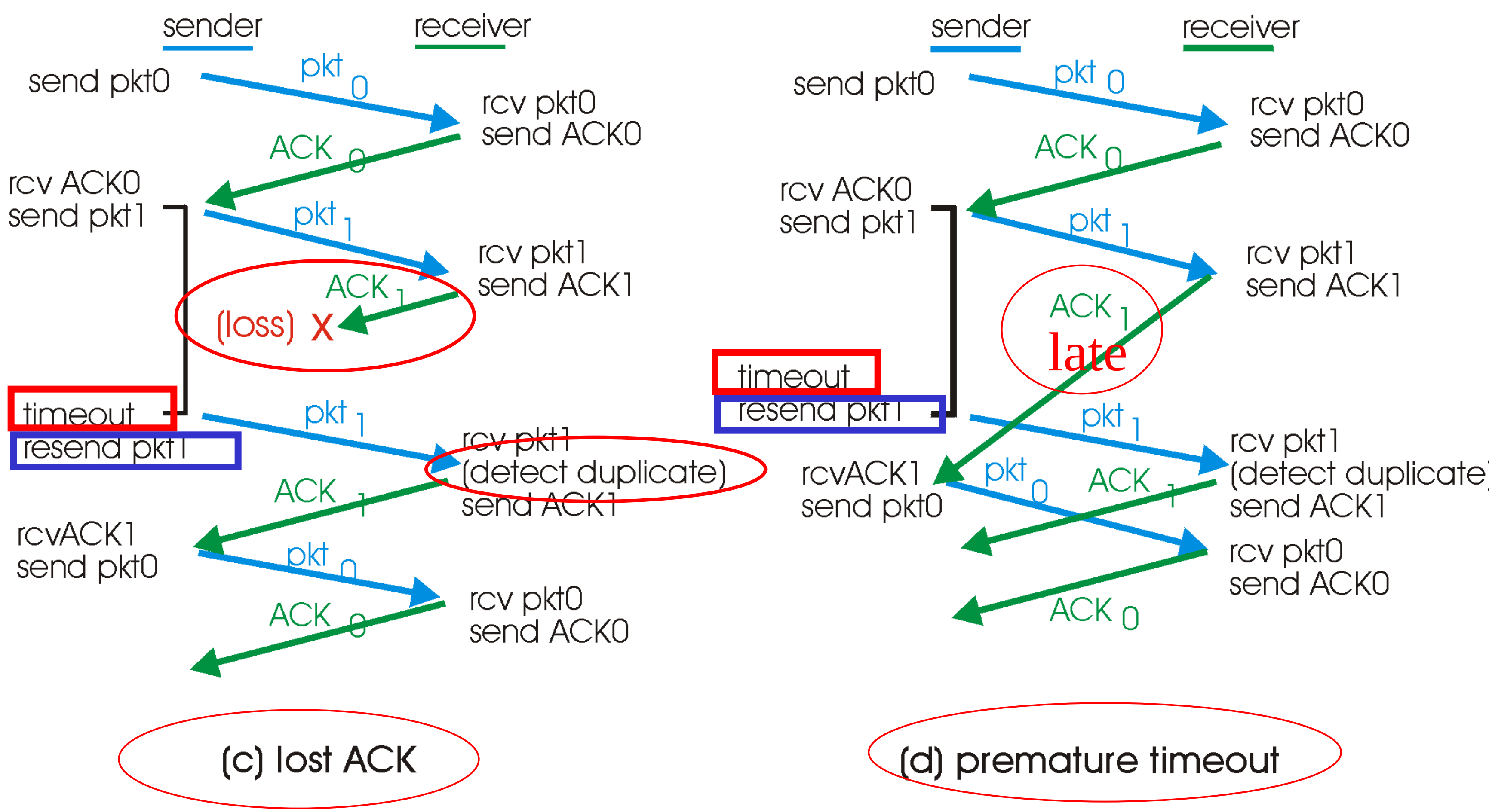


(a) operation with no loss



(b) lost packet

rdt3.0: Timer-based Retransmission



Error Control and Recovery: summary (1/7)

- For service interfaces that provide "reliable" delivery
- Problem 1:
  - "how to let sender know receiver correctly receives the data that sender sends?"

  => Need feedback from the receiver
  - Solution 1.1: Use Positive or Negative Acknowledgement (ACK)
- Problem 2:
  - "How to distinguish received packets, i.e. packet retransmission and duplicate packet?"
  - Solution 3: Packets are numbered
    - Sequence number assignment

Error Control and Recovery: summary (2/7)

- Problem 3:
  - "What happens if packet, ACK/NAK corrupted or lost?"
    - sender doesn't know what happened at receiver!
  - Solution 3: Use timer in the presence of error or hardware malfunction.
  - Sender starts a timer when transmits a frame out.
    - Timeout interval must be properly set.
    - At least a round trip time from sender to receiver
    - Sum of transmission time from sender to receiver, processing time delay at receiver, and ack transmission time from receiver to sender.

Error Control and Recovery: summary - two ways (3/7)

- Positive ACK + timer at sender
  - P-ACK(n) by receiver
  - if P-ACK(n) lost
    - Timer (n) goes off at sender
    - Sender retransmits packet(n)
- Negative ACK + timer at receiver
  - if P(n) is not received
    - Timer (n) goes off
    - N-ACK(n) by receiver
    - Sender retransmit packet(n)

Error Control and Recovery: summary - discussions (4/7)

- Congestion at receiver
  - if Ack(n) is on the way & Timer(n) goes off
    - Sender retransmits packet(n)
    - Receiver receives duplicate packet(n)
    - Receiver discards retransmitted packet(n)

Error Control and Recovery: summary (5/7)

Acknowledgement packets can be transmitted either via

- separate packets (e.g. use "type" field in the frame header to distinguish them)

or

- Piggybacking
  - Attach acknowledgement information to the outgoing data packets, i.e. include an "ack" field in the packet header
  - Problem
    - May result in variable delays for ack transmission

Error Control and Recovery: summary - Acknowledgement Packet (6/7)

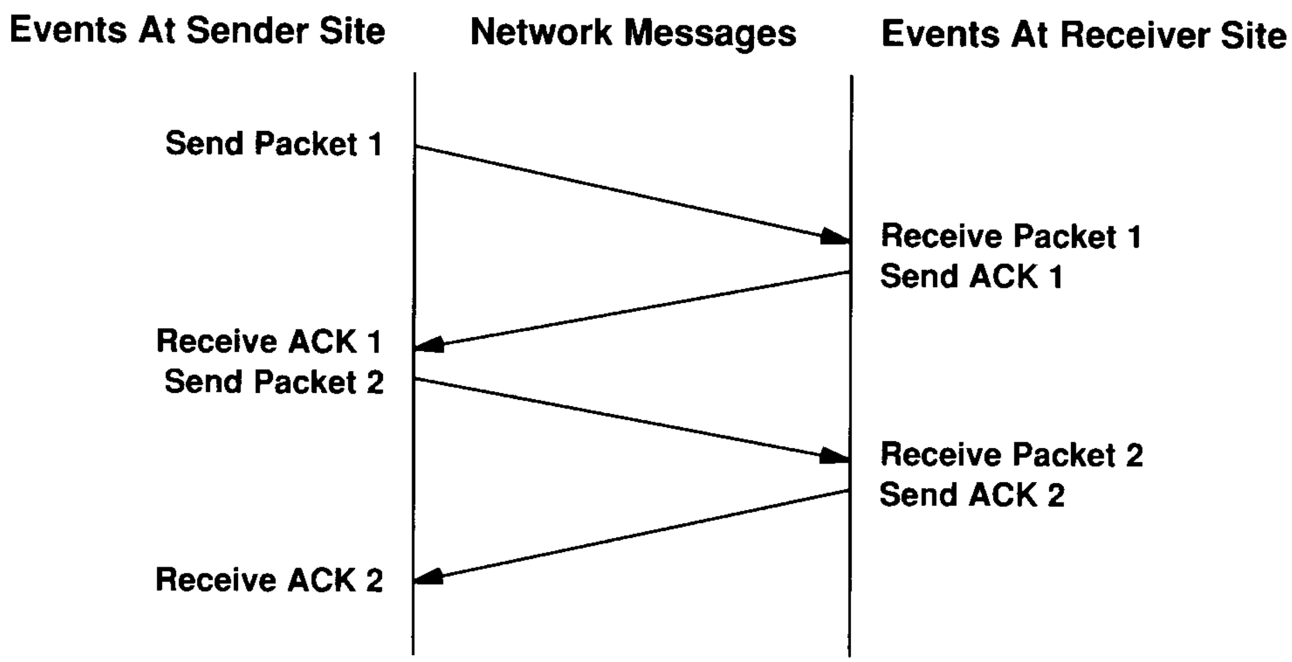
- Advantages
  - Use less resources (e.g., bandwidth)
  - Less interrupts to local processing unit
- "How long should the receiver wait for a packet onto which to piggyback the ACK?"
  - Solution:
    - Wait for a fixed amount of time T
    - If a new frame to transmit, piggyback the ack onto it
    - Otherwise, send a separate ack packet
    - Note T should be determined based on the traffic characteristics, e.g., RTT.

Error Control and Recovery: summary - Robustness (7/7)

- We say a protocol is robust if it works under all circumstances, such as errored packets, lost packets, and premature timeouts or their combinations).

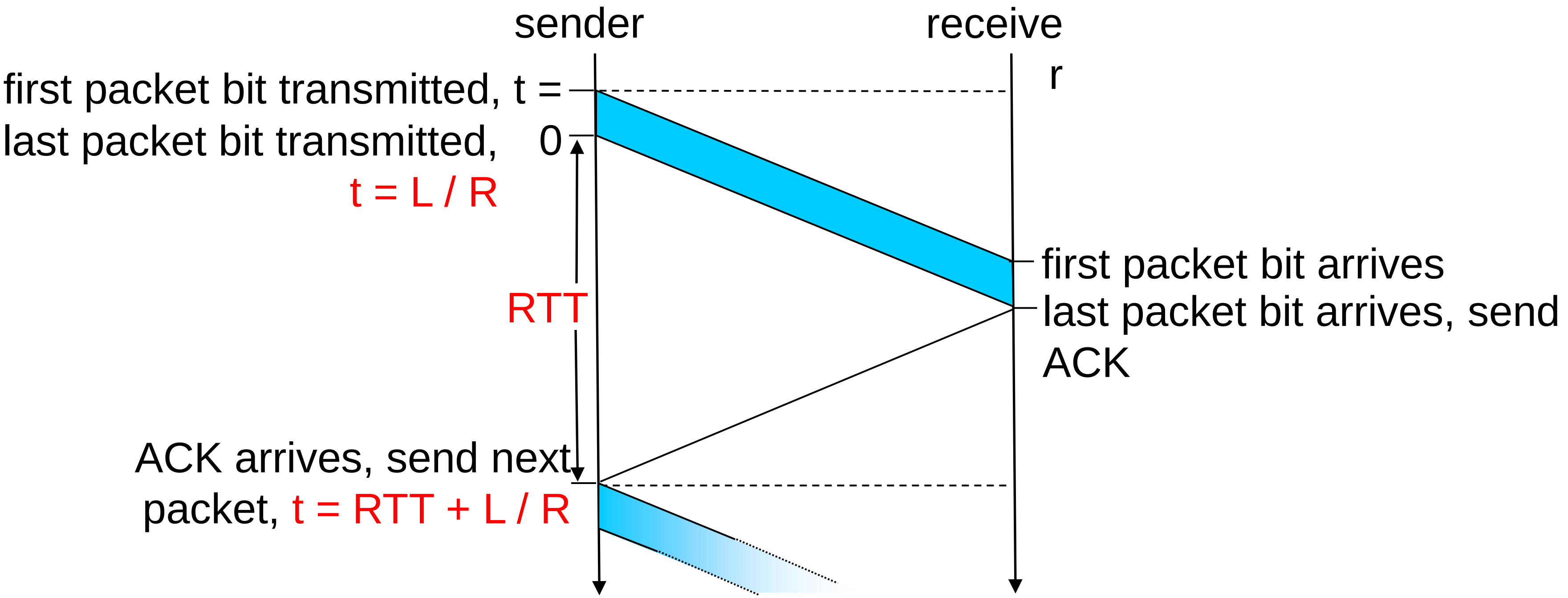
Flow Control

Stop-and-Wait



- Maximum window size is one
- Sequence number – one bit
- Sender sends a packet and waits for its ack before sending the next one

rdt3.0: stop-and-wait operation



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

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

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

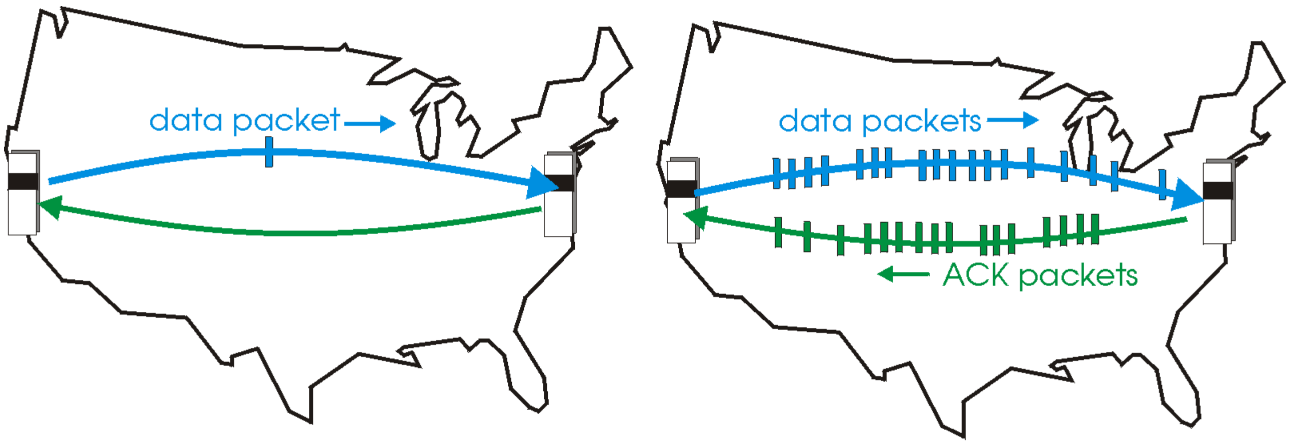
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

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

Pipelined protocols

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

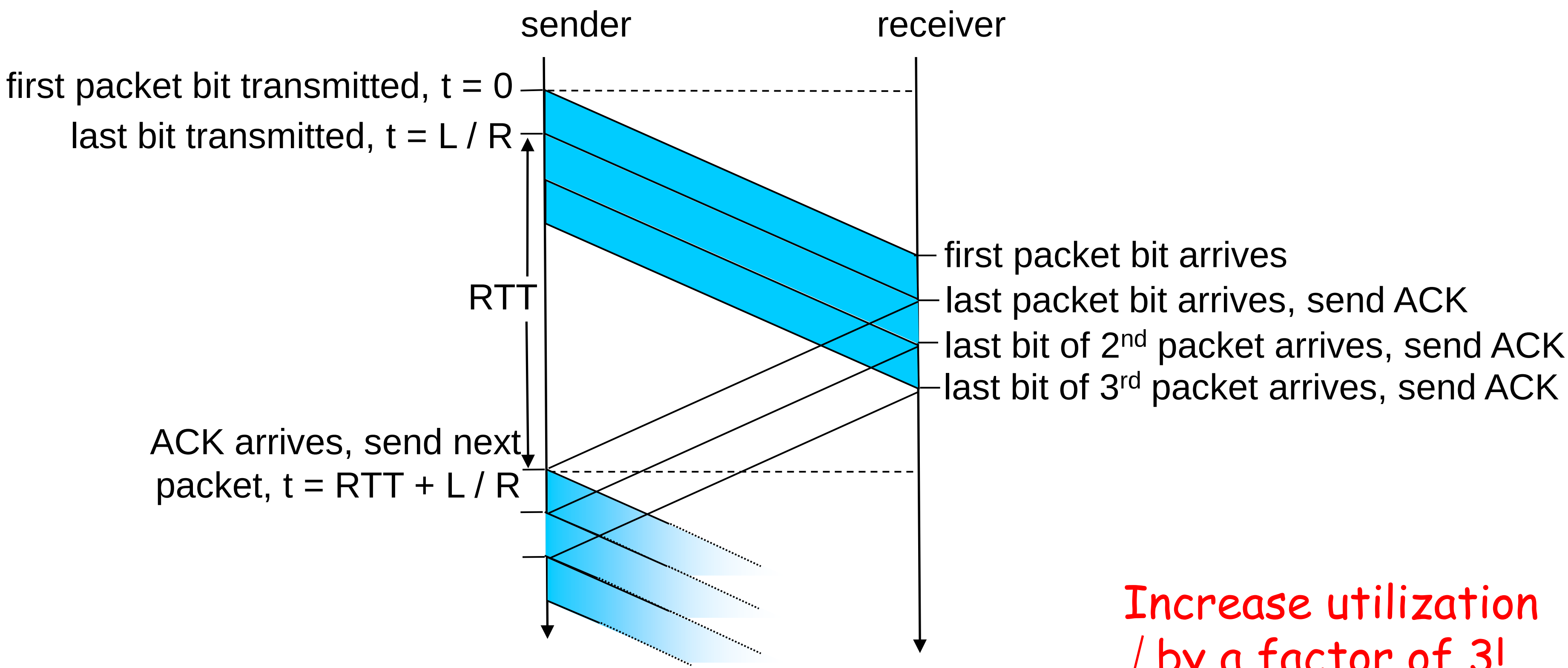
- range of sequence numbers must be increased
- buffering at sender and/or receiver



(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

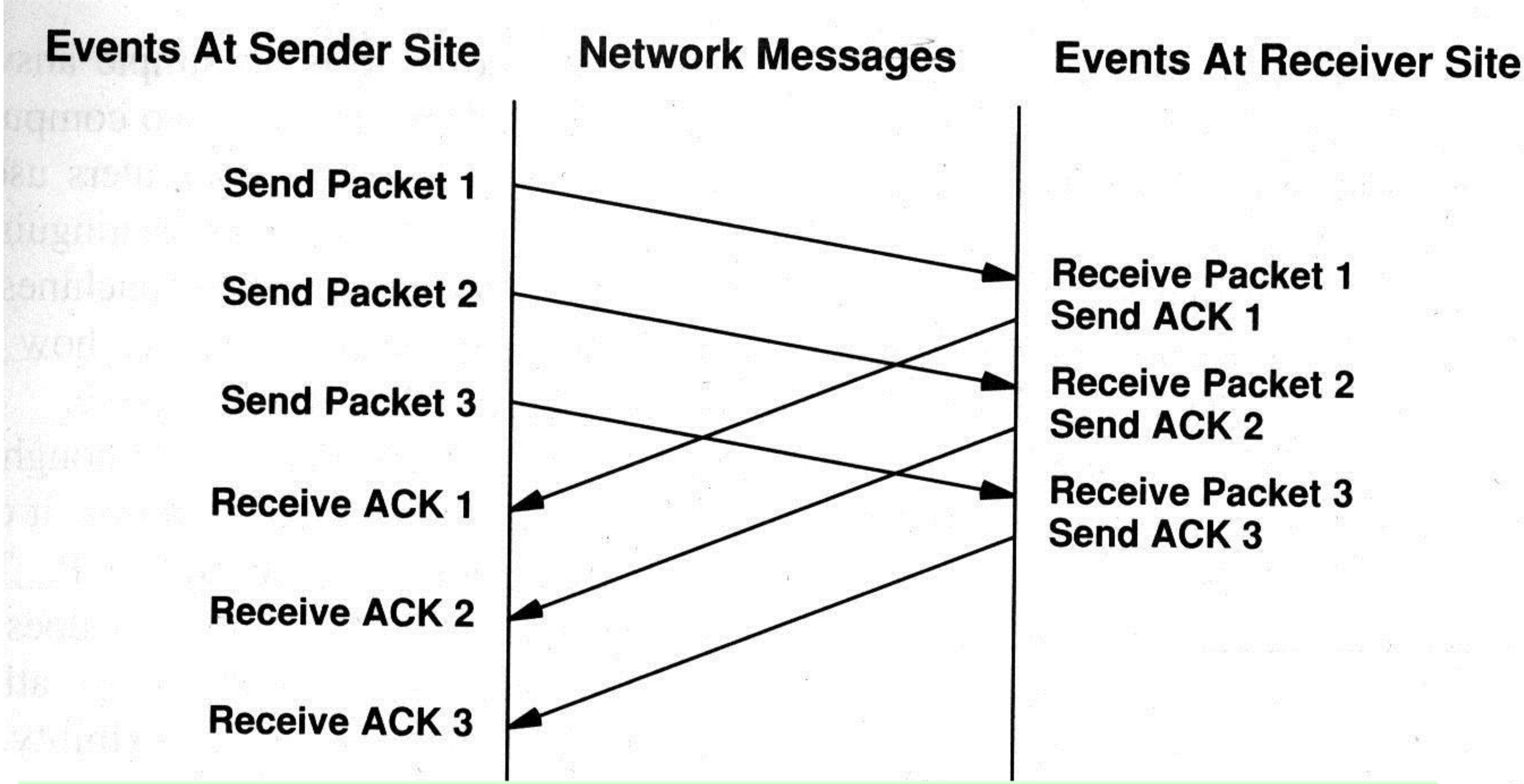


Increase utilization by a factor of 3!

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

Pipelining: summary

- To achieve better efficiency
  - Allows the sender to transmit up to w packets before blocked
  - Multiple outstanding packets
- Issues: determine w?
  - e.g., consider the previous example
  - 500/20=25 -> w=25
- w is the maximum number of outstanding unacked packets
  - Wrong!
- Issue
  - What happens if a packet (data or ack) in the middle of the long stream is damaged or lost?
- Two approaches: go-back-n & Selective repeat



Back

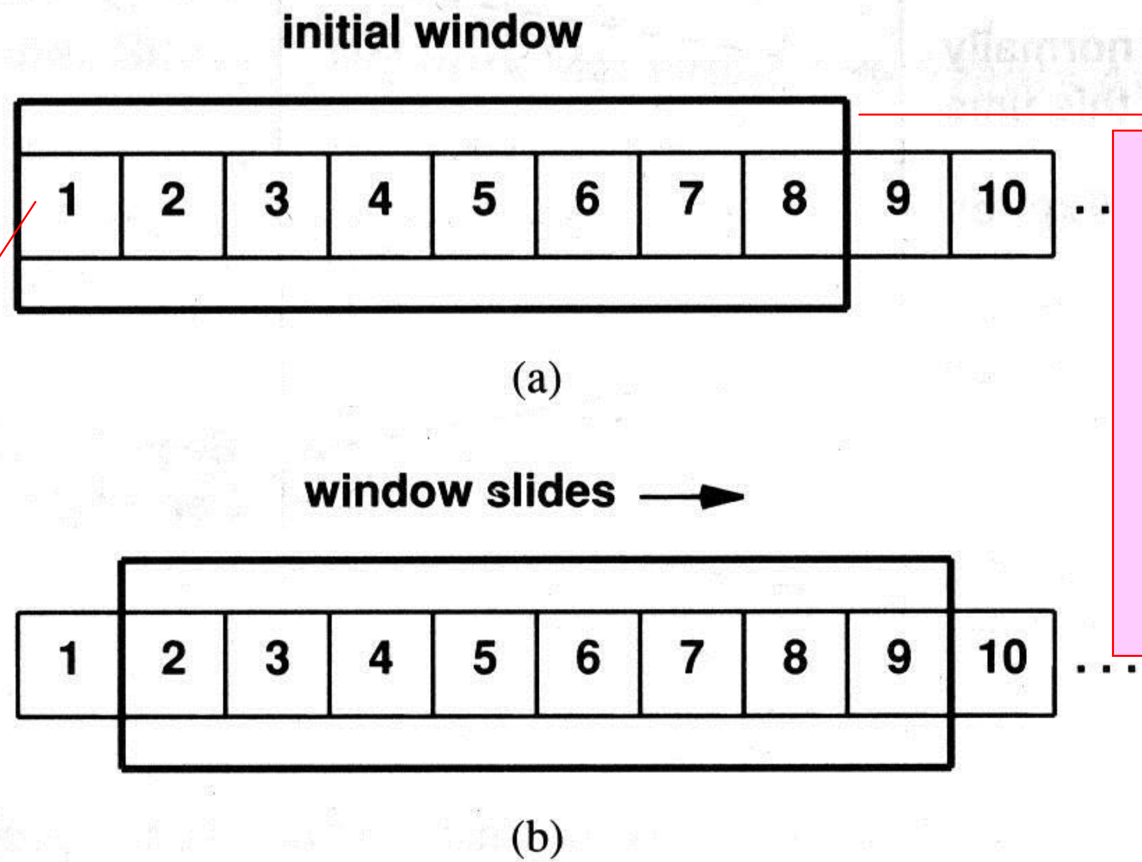
$$U_{pipe} = \frac{w \cdot (L/R)}{RTT + (L/R)} \qquad U_{pipe} = w \cdot U_{stop-and-wait}$$

Basics on Flow Control: Sliding Window Protocols

- Stop-and-wait (one bit) sliding window
- Go-back-n
- Selective repeat

- Note: these methods differ in efficiency, complexity and buffer requirements.

Packet arrivals



Sequence number (range)

Maximum number of packets allowed To send (quota)

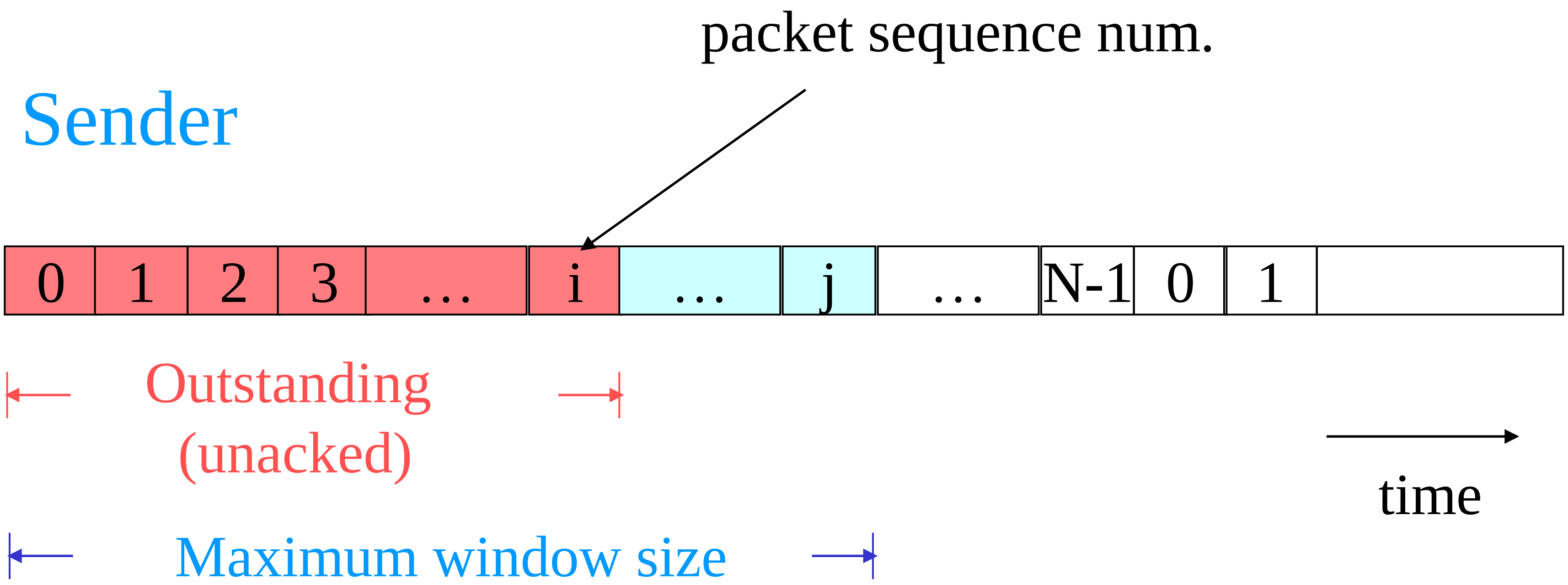
Sliding window

Next

Sliding Window Protocols – basic idea

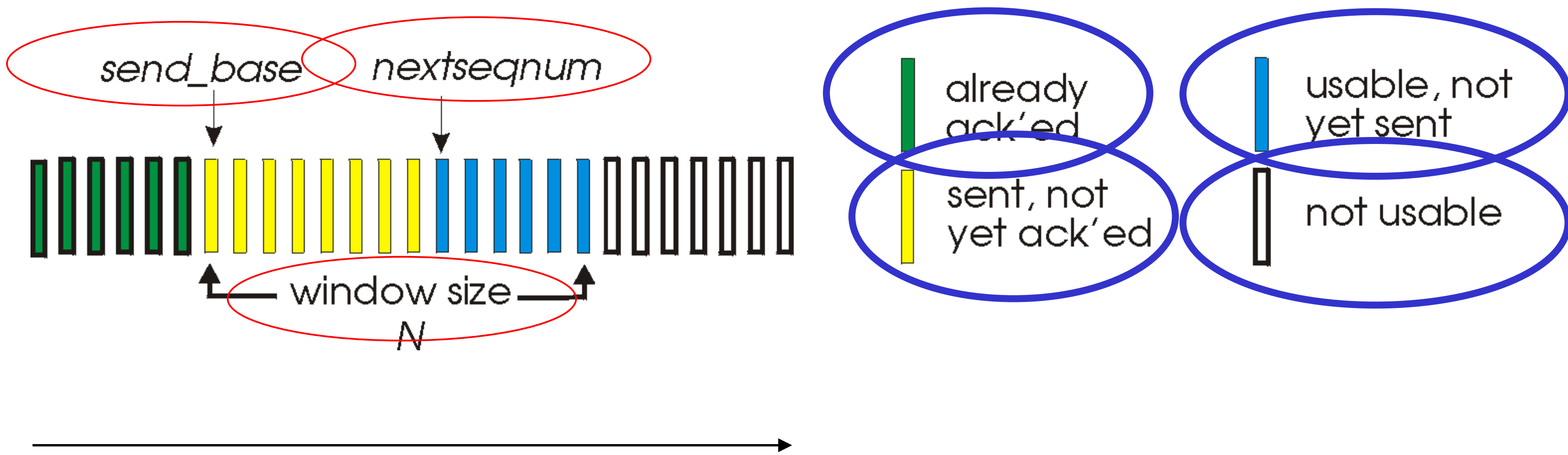
- Each outbound packet contains a sequence number, ranging from 0 to some maximum number (usually 0~ 2^n-1 using n-bit field)
- Sender maintains a list of consecutive sequence numbers, corresponding to packets it is *permitted* to send that is called sending window.
- Receiver also maintains a list of consecutive sequence numbers, corresponding to packets it is *permitted* to accept that is called receiving window.

Sliding Window Scheme



- Packets marked that have been sent and are waiting for acknowledgment
- The sequence numbers that can be assigned for any new outbound packets

Sender's Sliding window



front

rear

Pipelining Protocols

Go-back-N: big picture:

- Sender can have up to N unacked packets in pipeline
- Rcvr only sends cumulative acks
  - Doesn't ack packet if there's a gap
- Sender has timer for oldest unacked packet
  - If timer expires, retransmit all unacked packets

Selective Repeat: big pic

- Sender can have up to N unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only unack packet

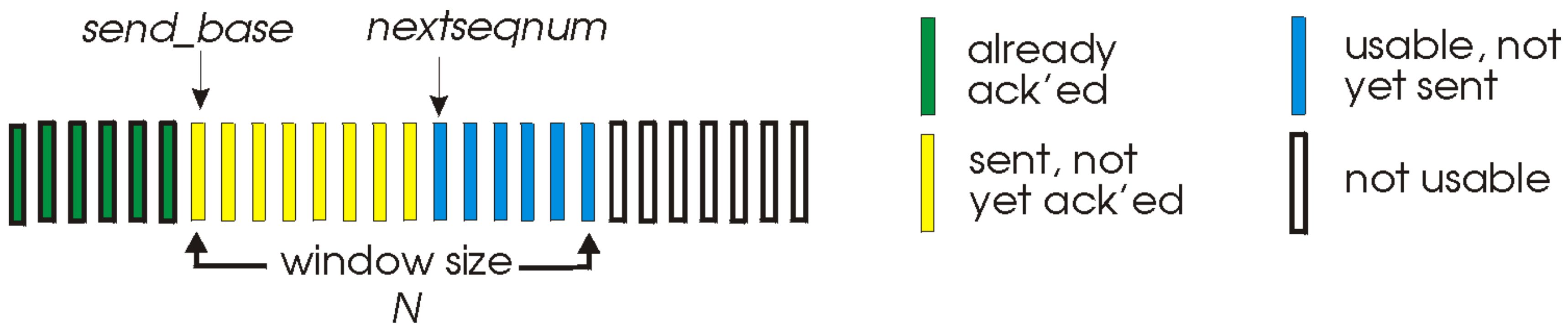
Go-Back-N Sliding Window Protocol

- When receiver receives an *error* packet, it discards all subsequent packets, i.e. drop all out-of-sequence packets.
- Drawback
  - Waste bandwidth in high error rate channel
- Advantage
  - Simpler operational complexity for receivers

Go-Back-N Sliding Window Protocol

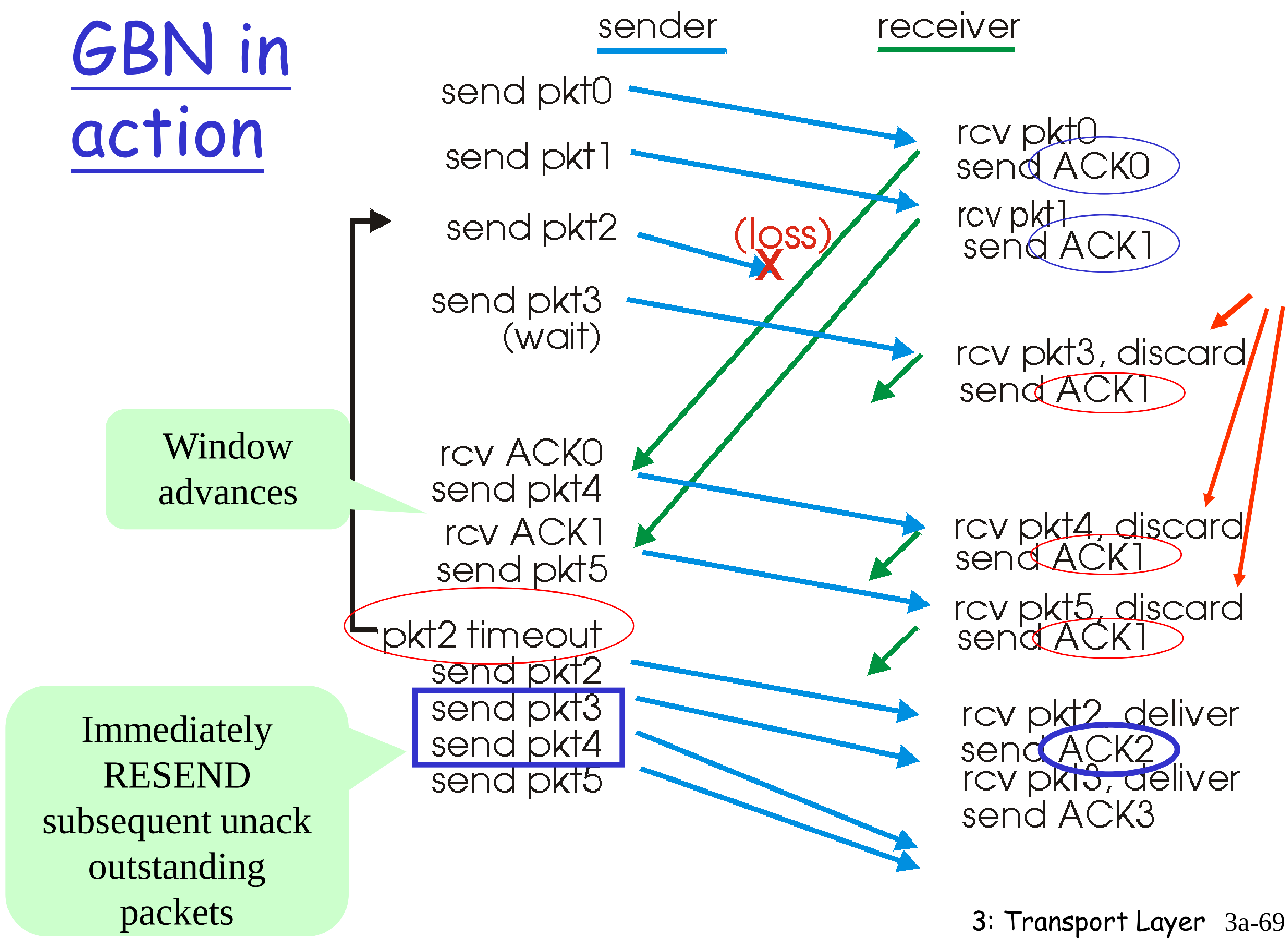
Sender:

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

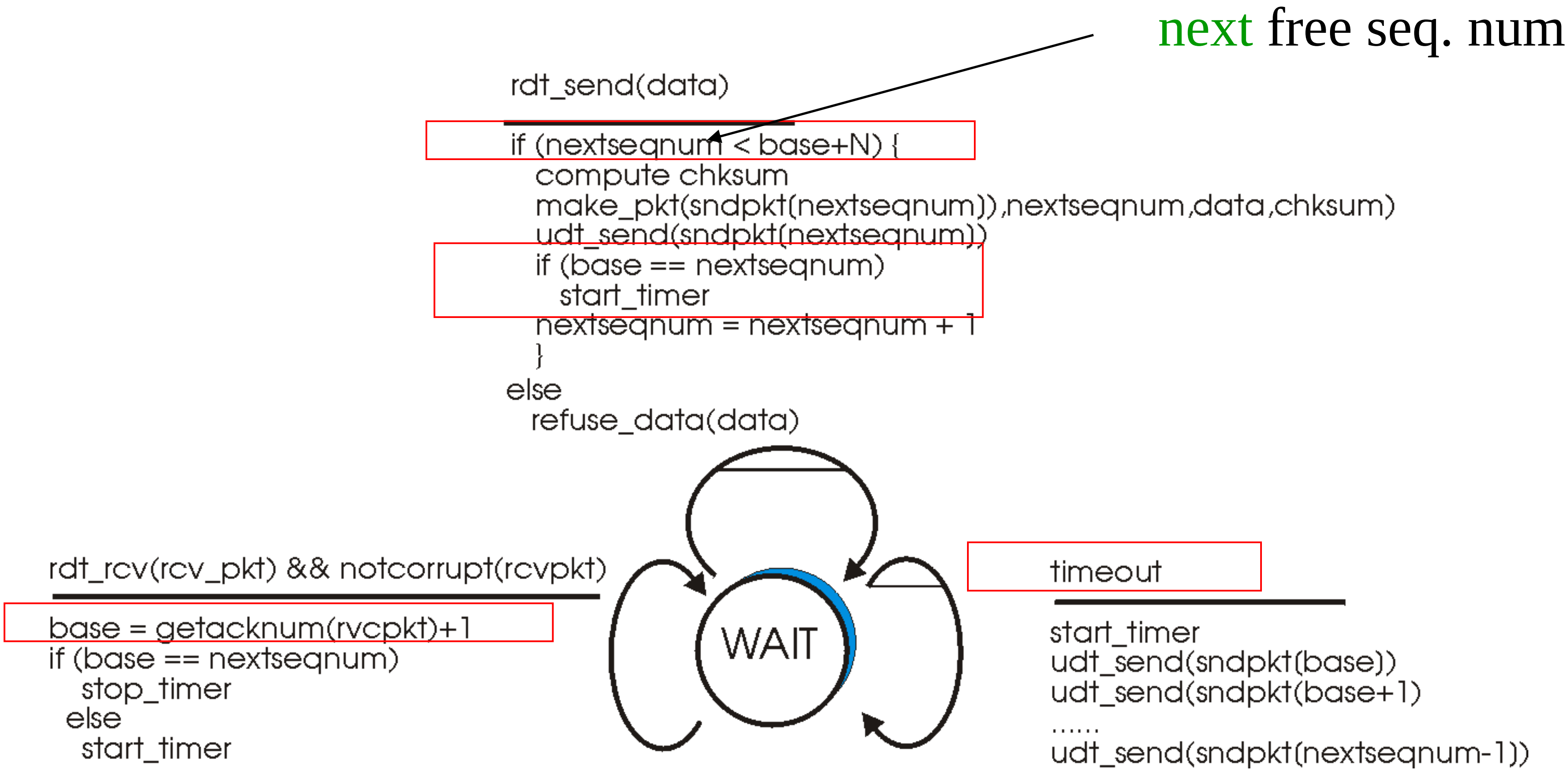


- ACK(n): ACKs all pkts up to, including seq # n - "**cumulative ACK**".
  - may deceive duplicate ACKs (see receiver).
- timer for each in-flight pkt.
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window.

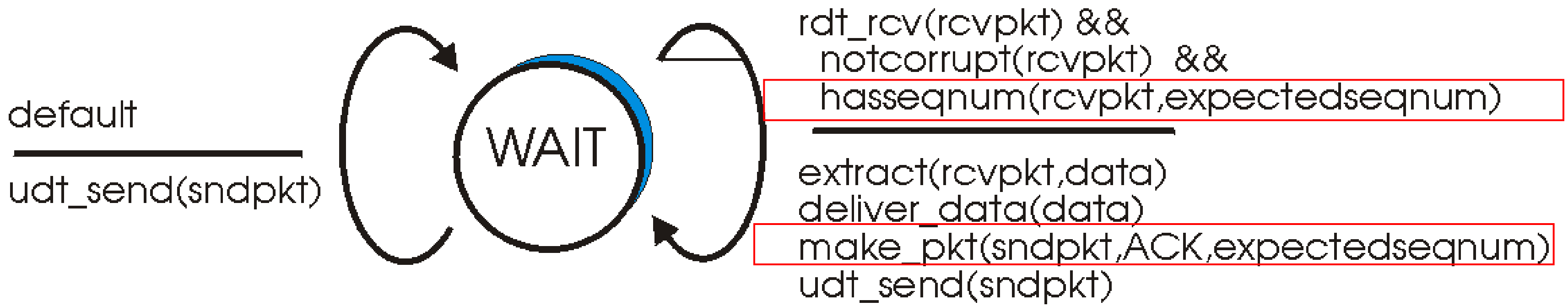
GBN in action



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 #

Go-Back-N Sliding Window Protocol

- Advantage
  - Simpler operational complexity for receivers
  - Sender
    - One timer (vs. one for each outstanding packet)
  - Receiver
    - No need to buffer out-of-order packets (less buffer requirement, simple operation)

- Drawback
  - Waste bandwidth in high error rate channel

Sliding Window Protocol using "Selective Repeat"

- Receiver is able to accept and buffer all correctly received, out-of-sequence packets.
- Receiver individually acknowledges all correctly received pkts.
- Eventual in-order delivery to upper layer

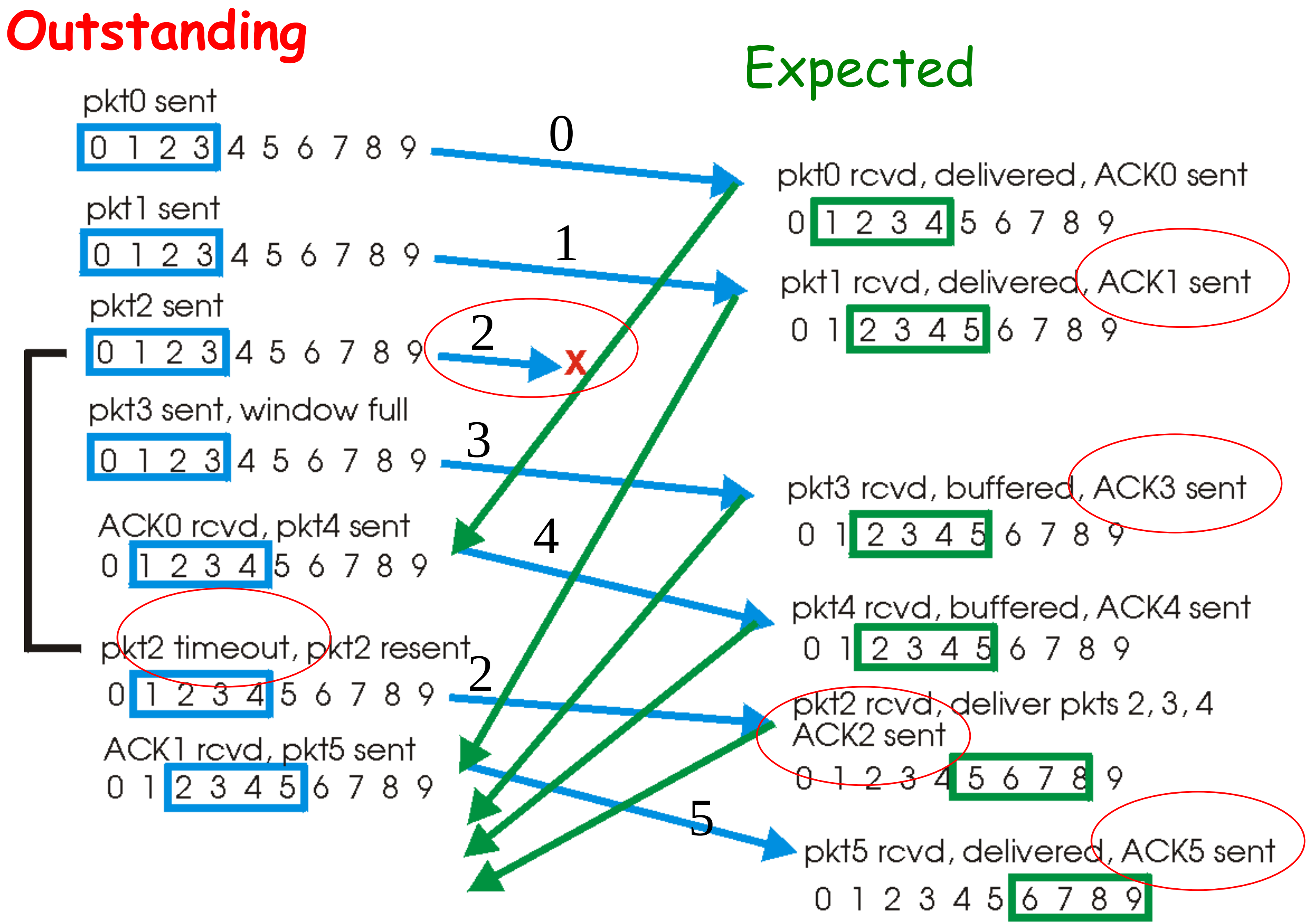
- Algorithm at the receiver
  - For an out-of-sequence packet, check if falls within the receiving window.
  - Check if it is not a duplicate
  - If both are ok, store the packet in the buffer

Sliding Window Protocol using "Selective Repeat" (cont'd)

Algorithm at the sender

- 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" in action



Selective repeat: dilemma

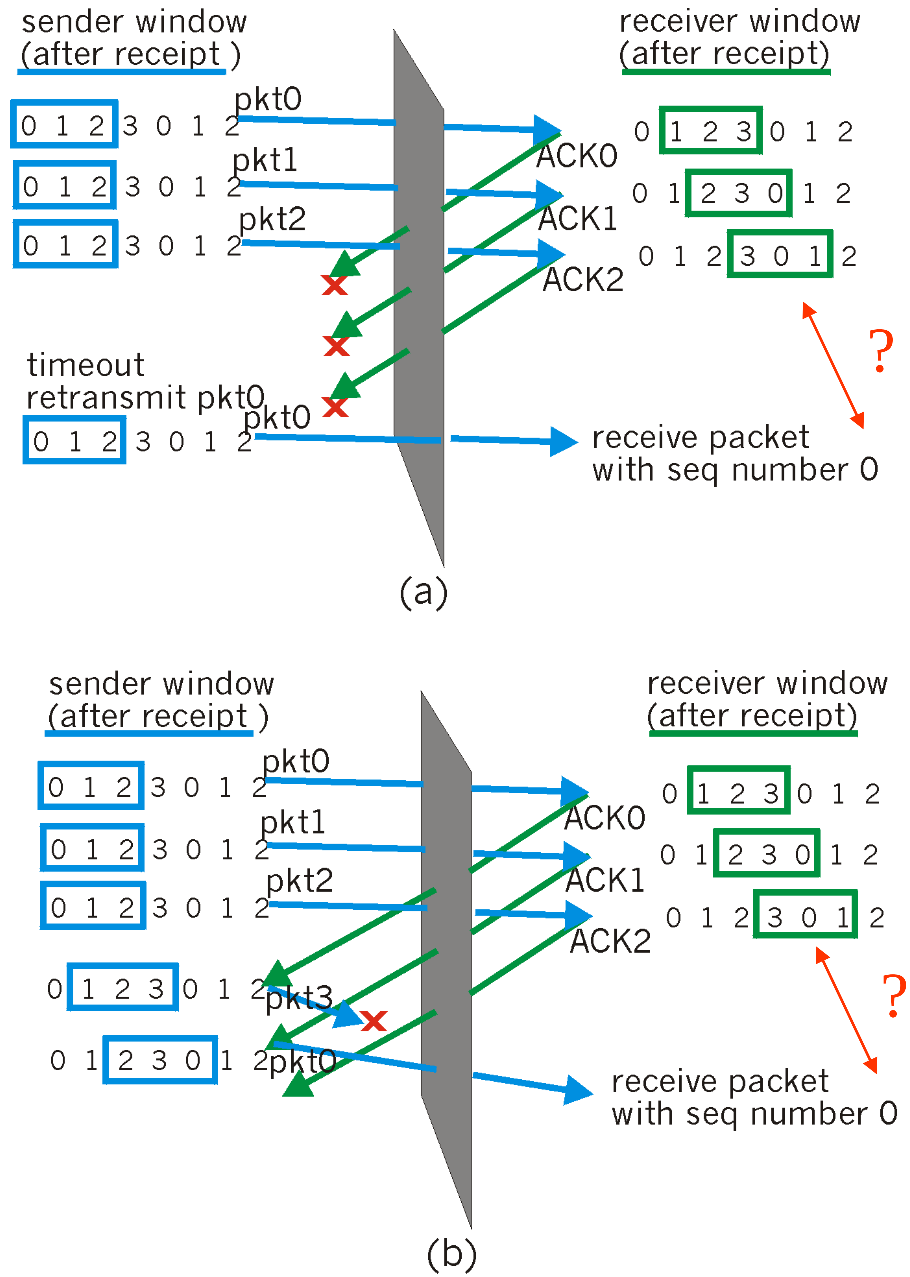
Example:

- 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?

windowSize <= sequneceNum/2



Selective repeat

sender

data from above :

- if next available seq # in window, send pkt
- set a timer for pkt n

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

To be continued … ☺