# Chapter 3: Transport Layer

## Our goals:

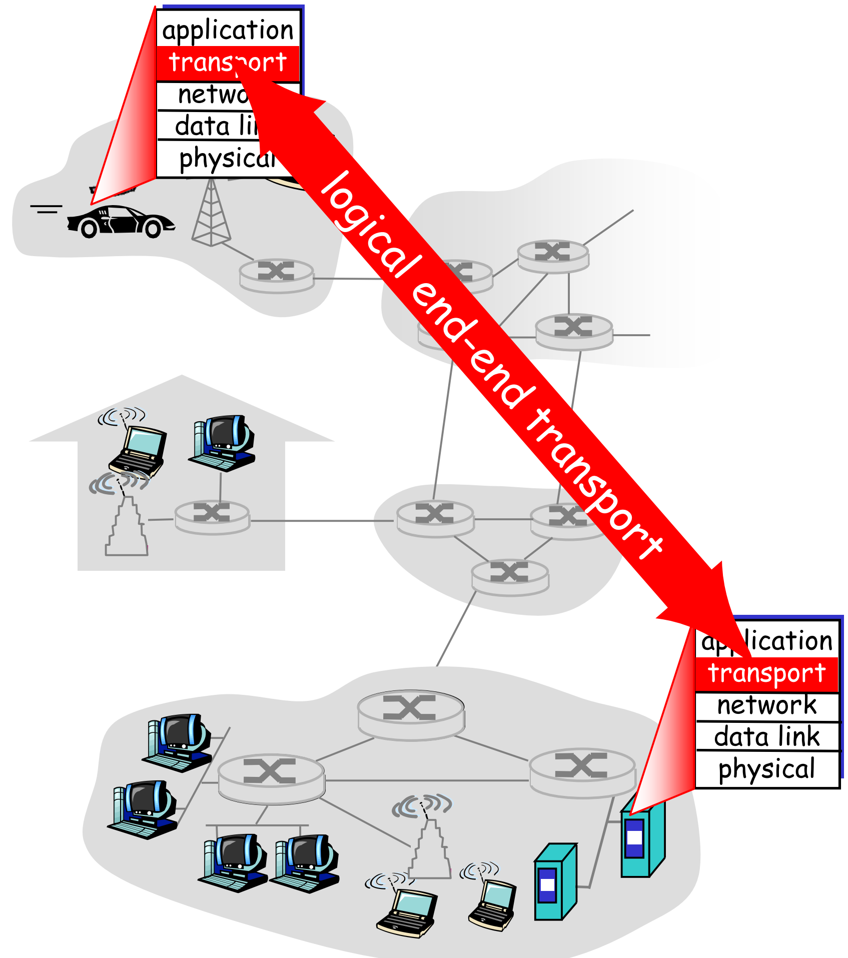
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion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

# 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

- 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

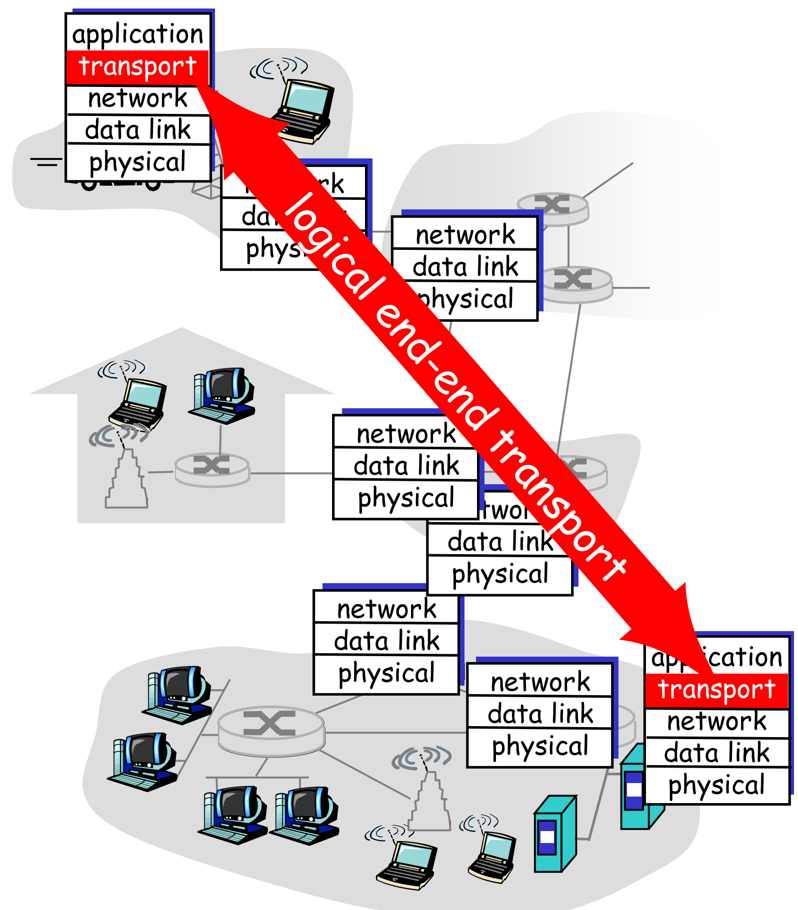
**Household analogy:**

*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



# 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

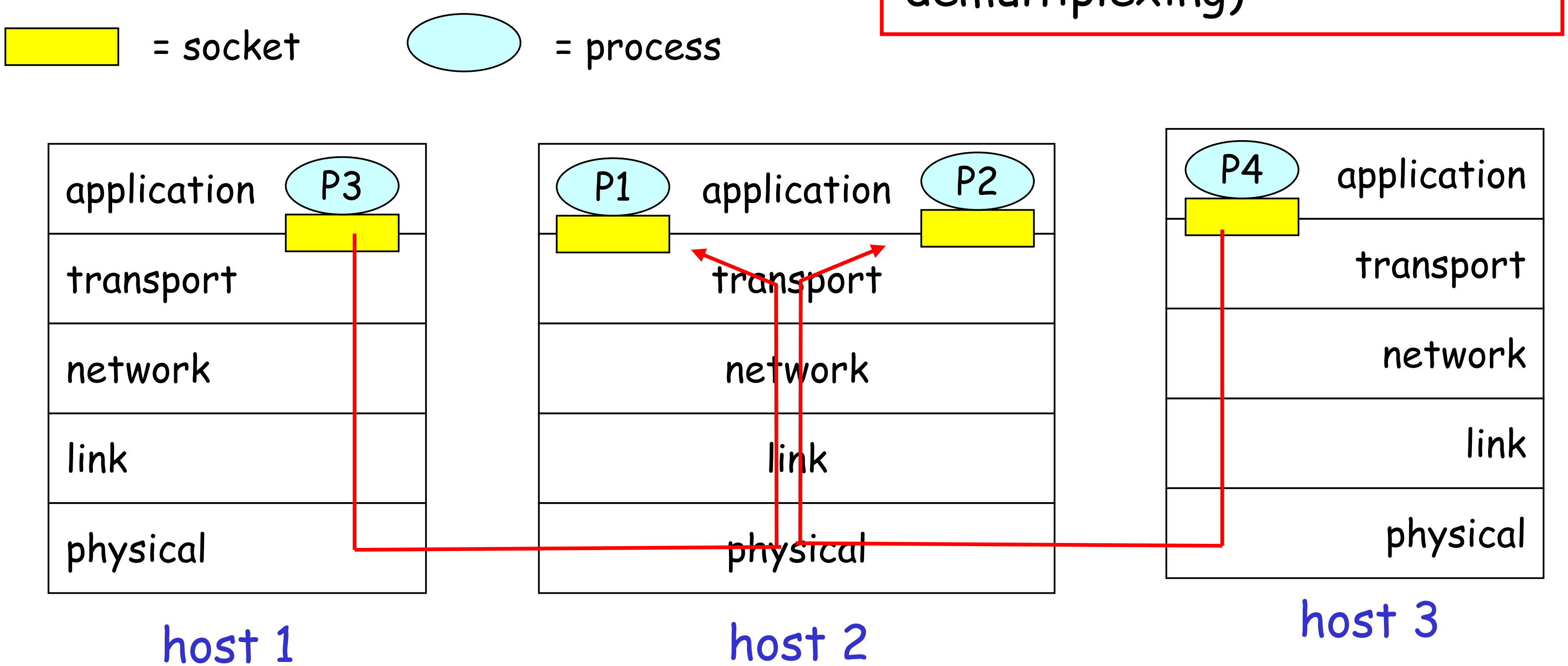
# Multiplexing/demultiplexing

**Demultiplexing at rcv host:**

delivering received segments to correct socket

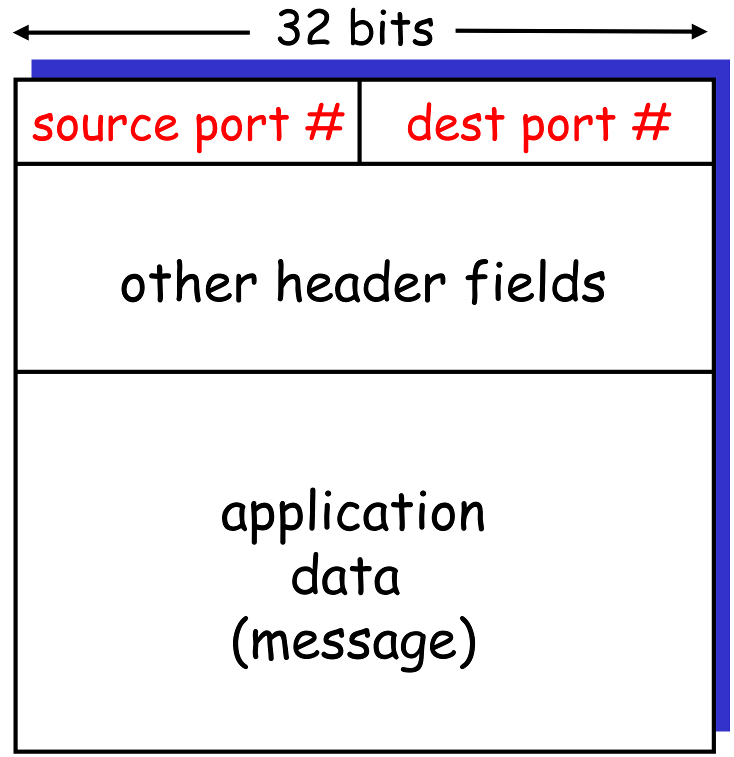
**Multiplexing at send host:**

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



# 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);
```

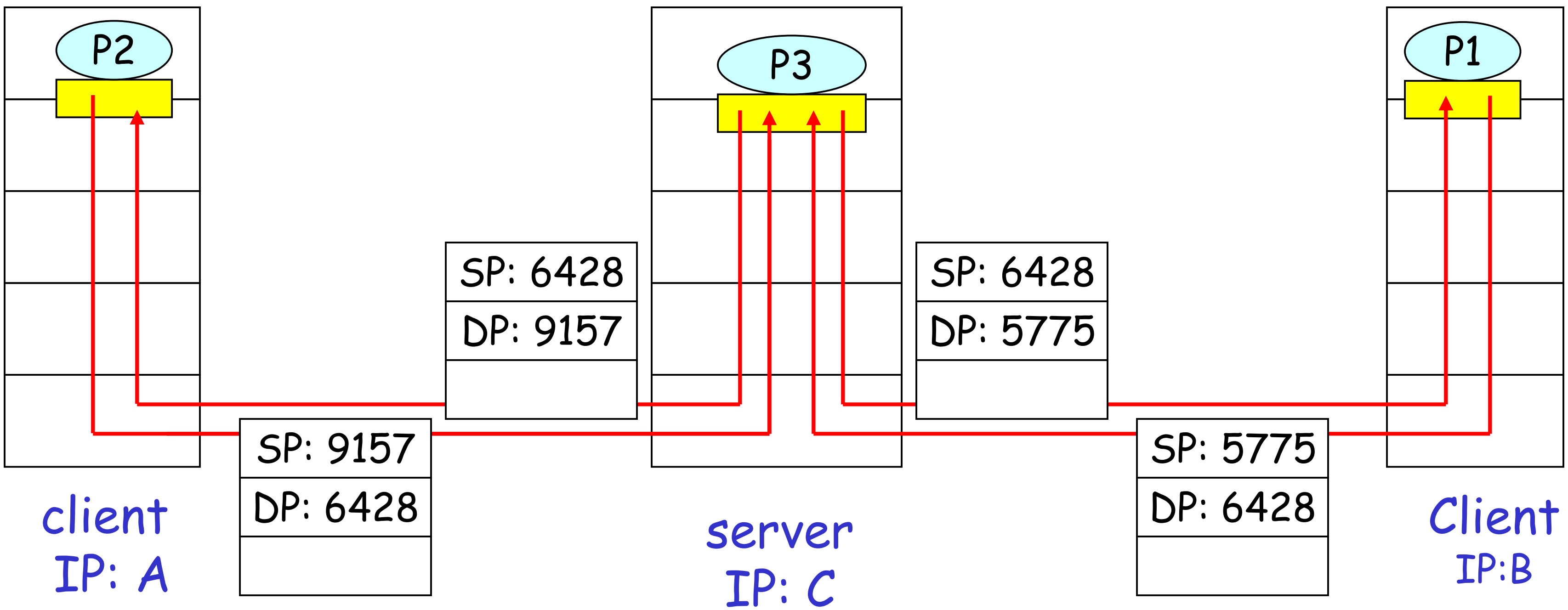
- UDP socket identified by two-tuple:

(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);
```



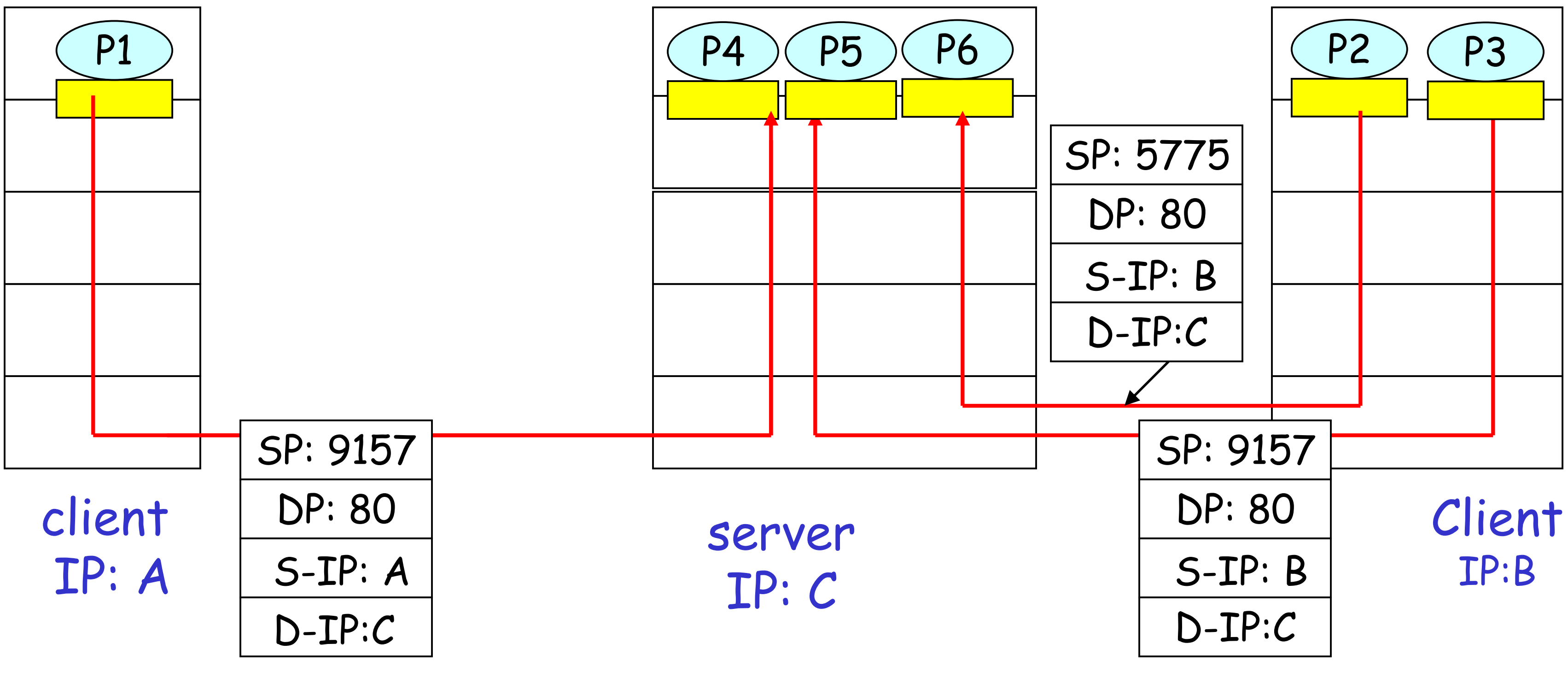

SP provides "return address"

# Connection-oriented demux

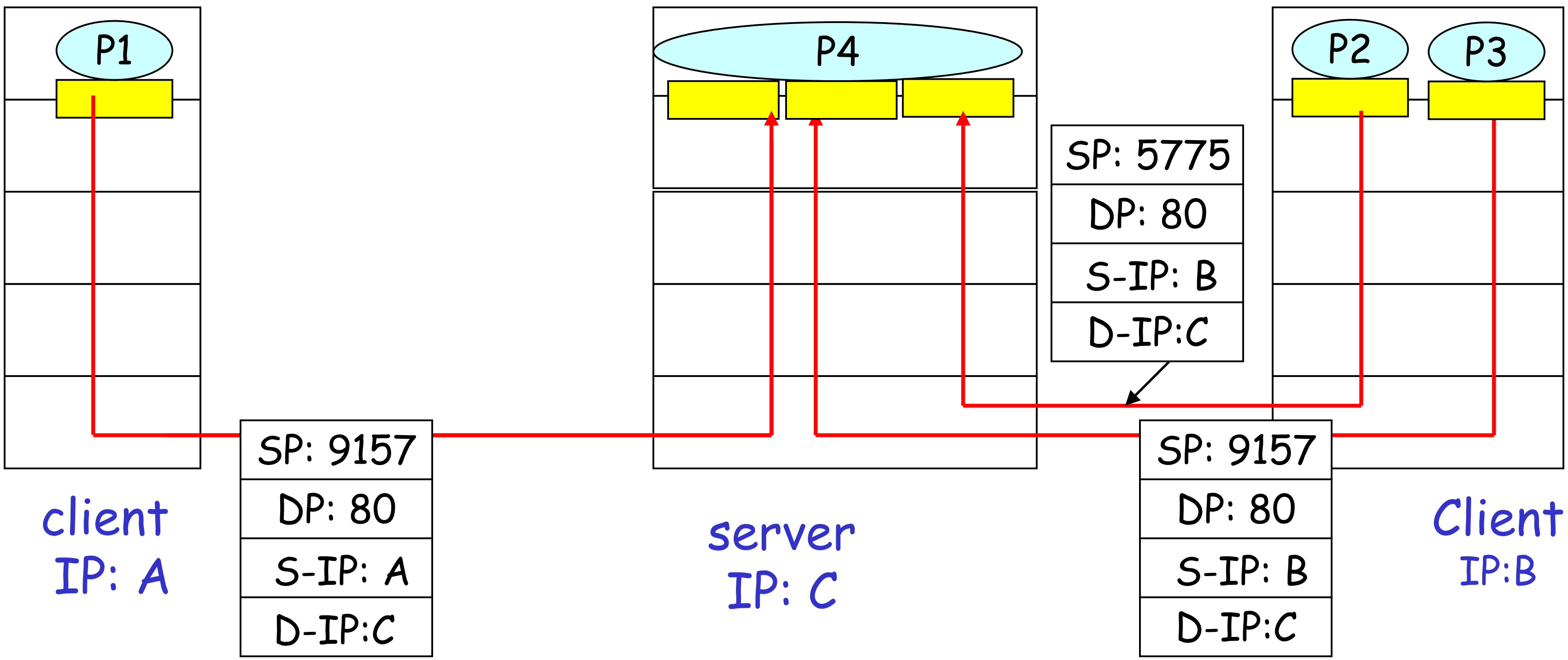
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket

- 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]

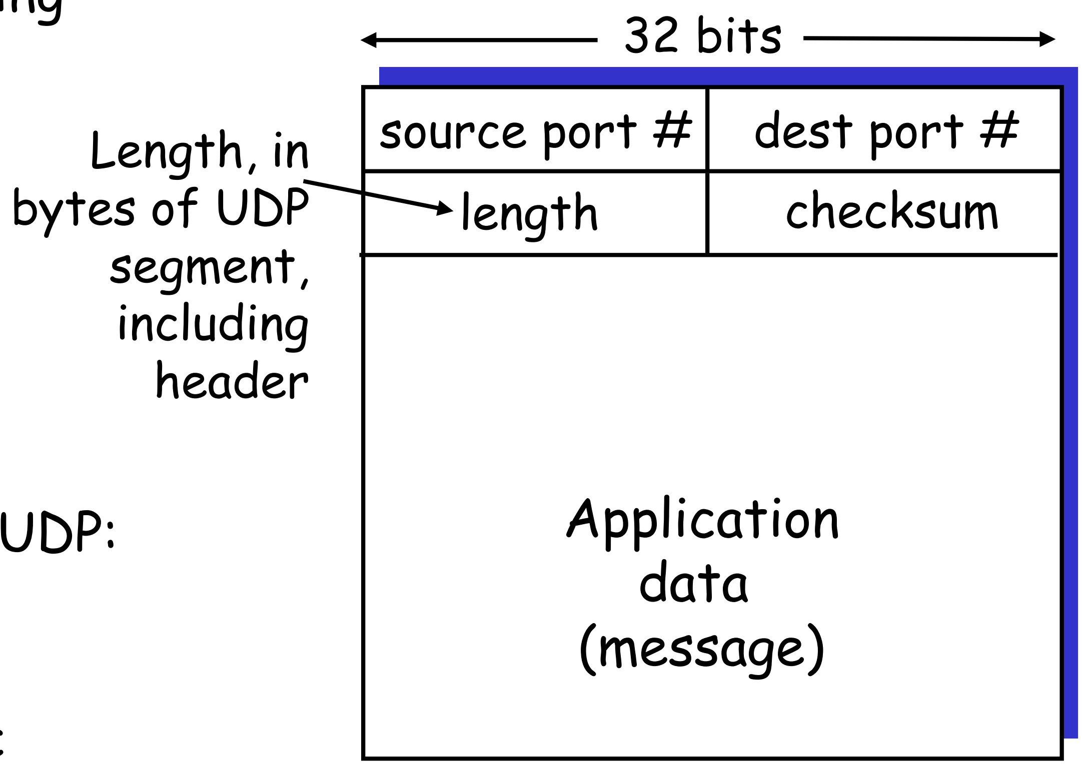
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

**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
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!




UDP segment format

# 关于UDP的思考

UDP 提供不可靠数据传输服务

- 为什么DNS使用UDP？
- 为什么SNMP使用UDP？

# 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

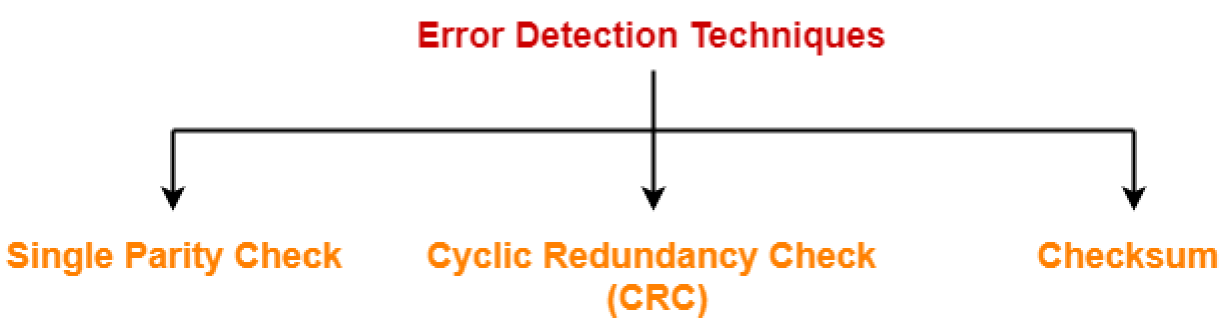
- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 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 |

# 关于Checksum的思考

Checksum is an error detection method.

- How many bit errors Checksum can detect at most?
- Checksum is used in IP header, TCP header, and UDP header.

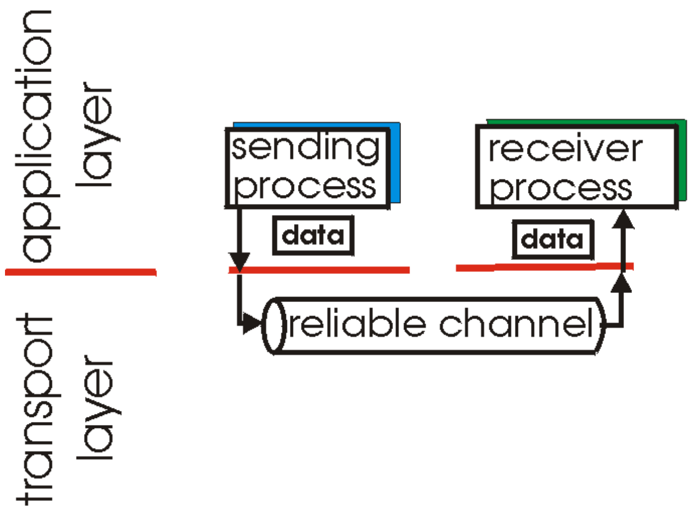


# Chapter 3 outline

- 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!

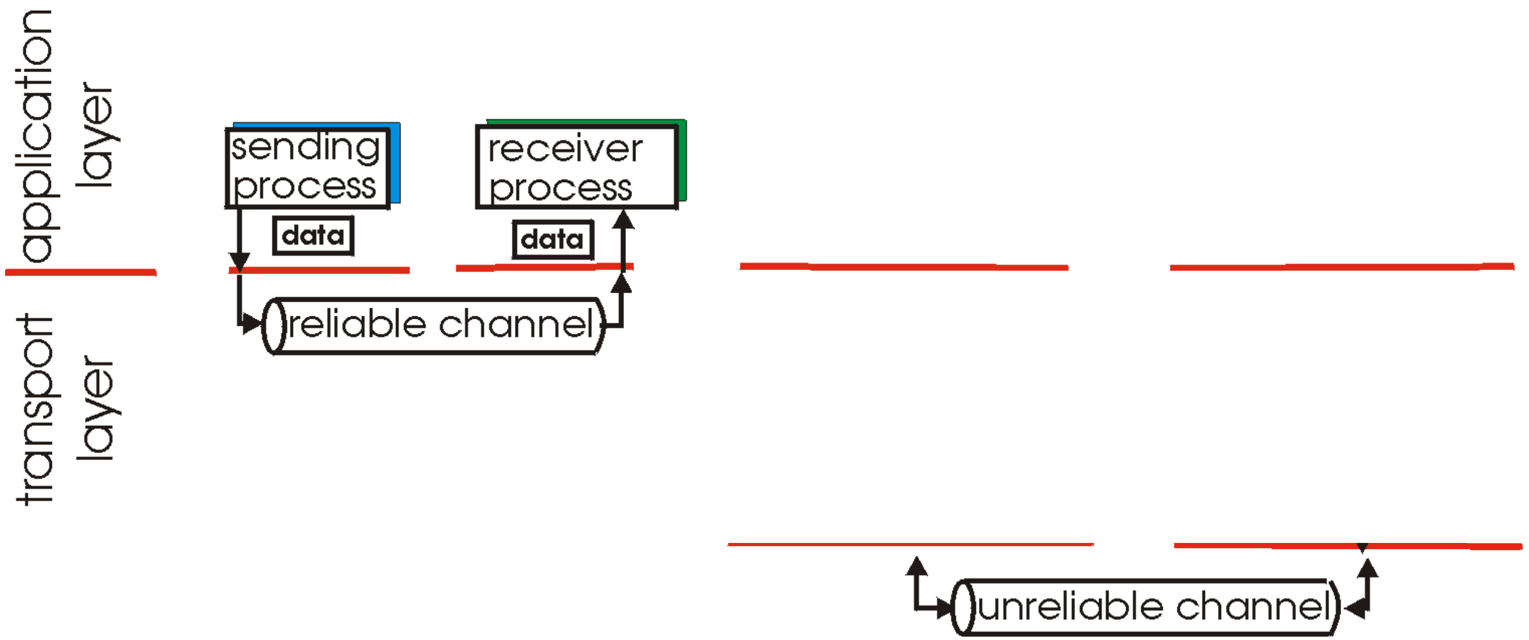



(a) provided service

- 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!

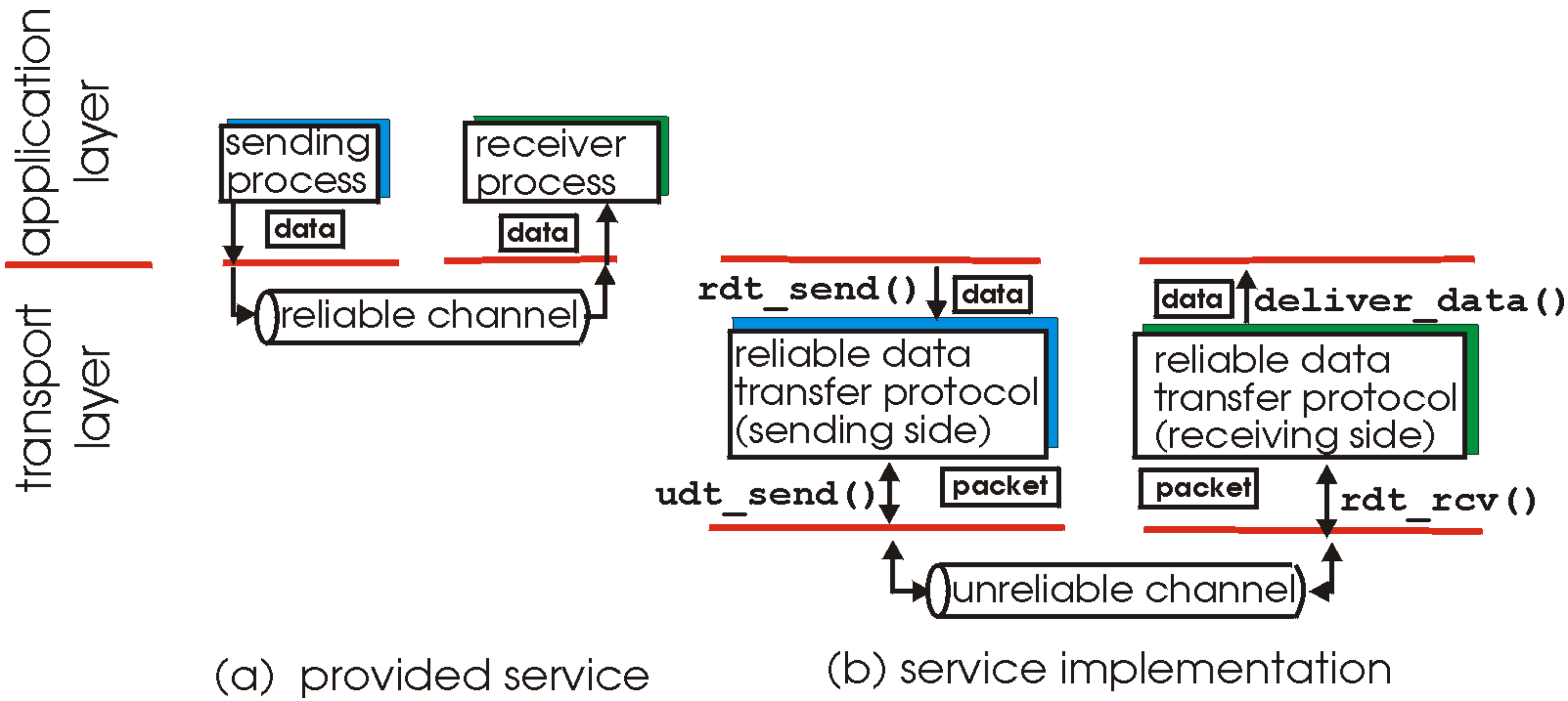



(a) provided service (b) service implementation

- 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!

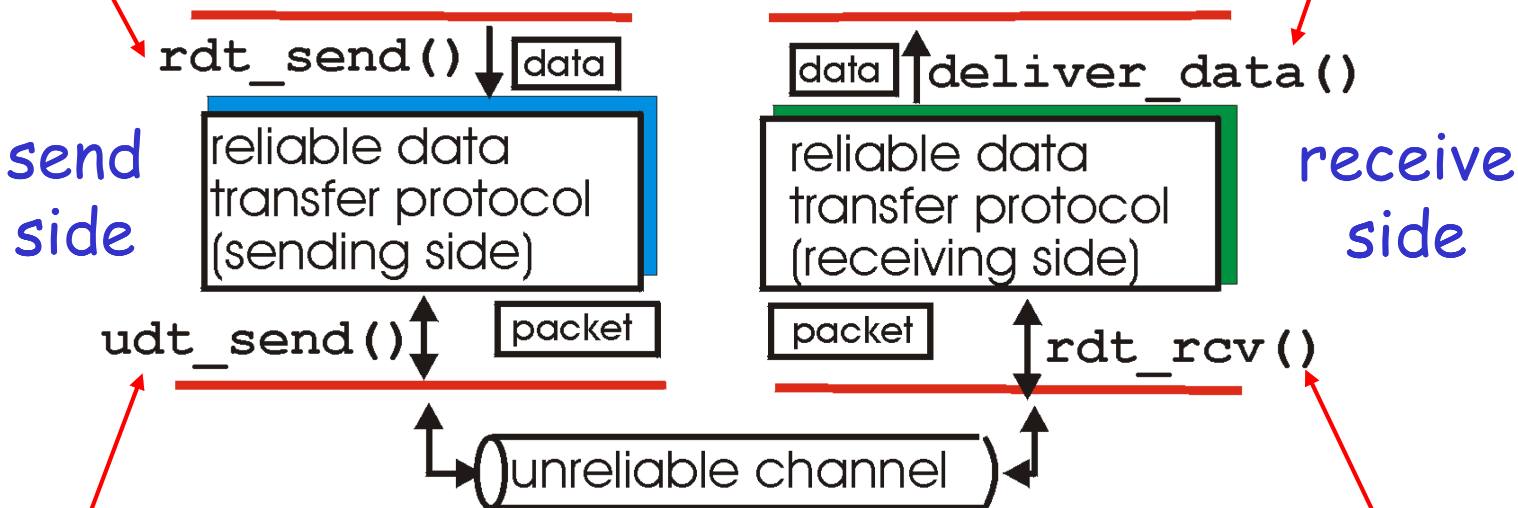



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# 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



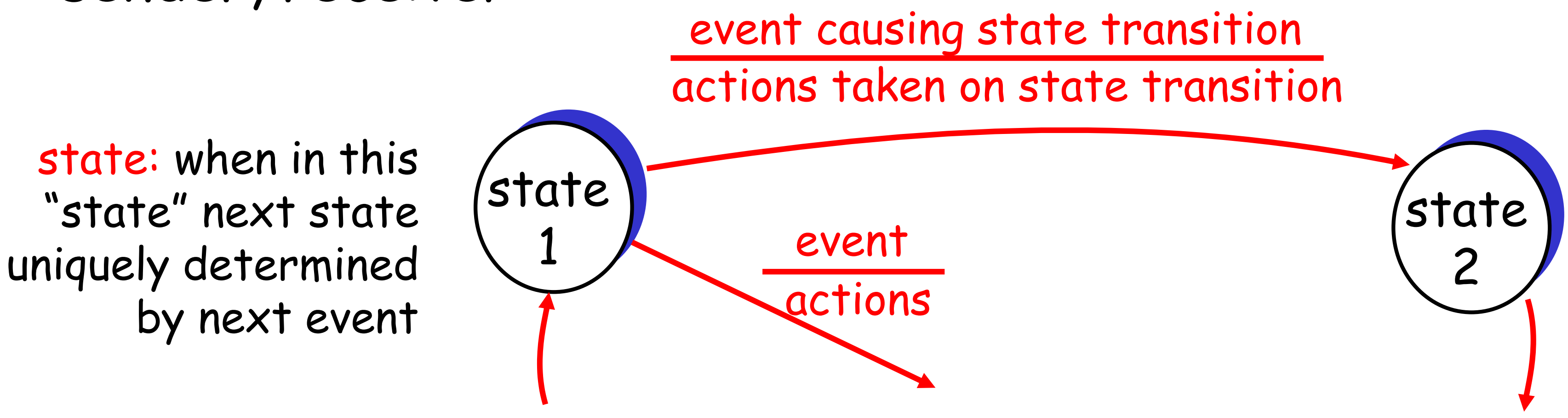

**udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver

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

# Reliable data transfer: getting started

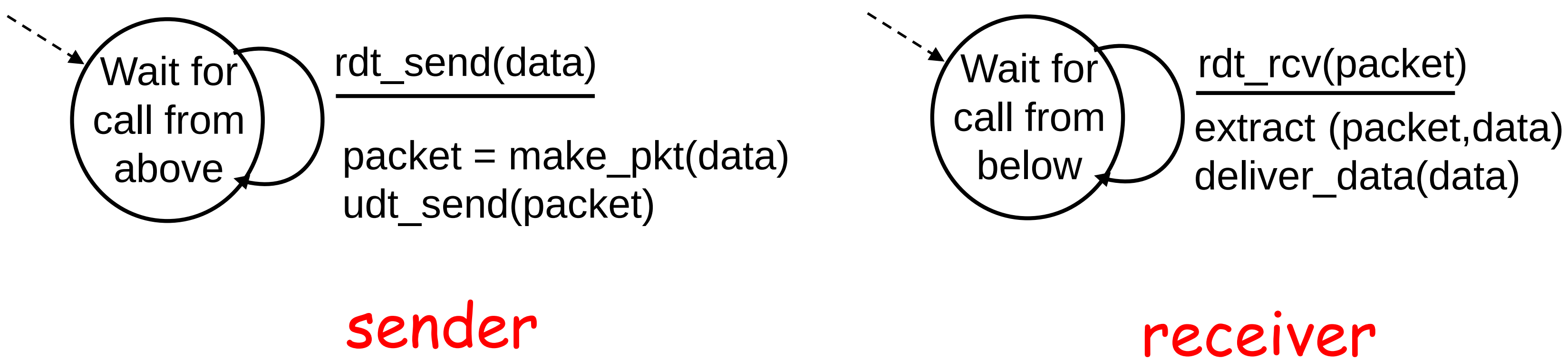
We'll:

- 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



# Rdt1.0: reliable transfer over a reliable channel

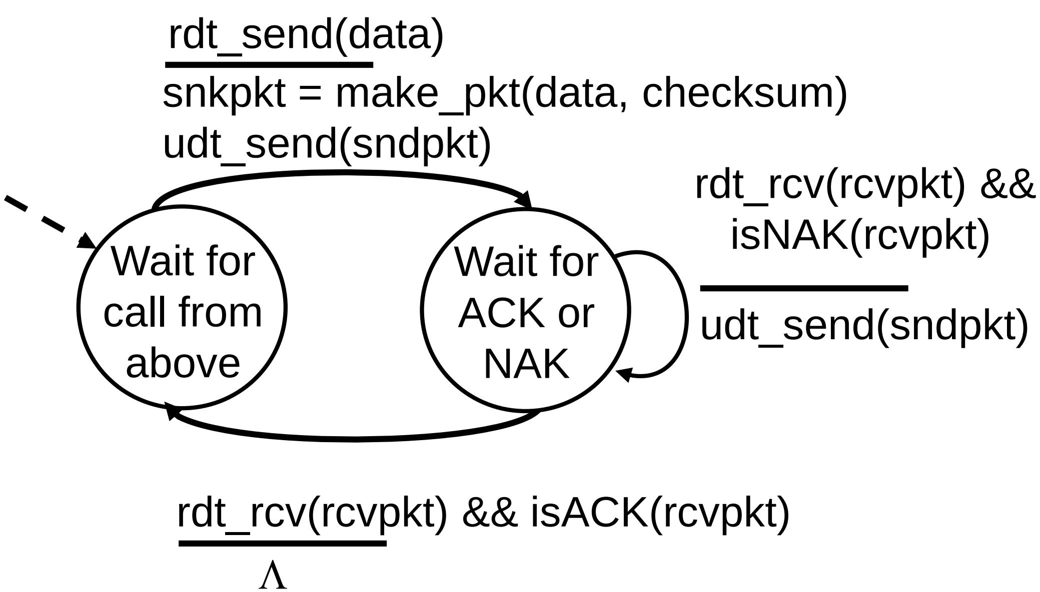
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



# Rdt2.0: channel with bit errors

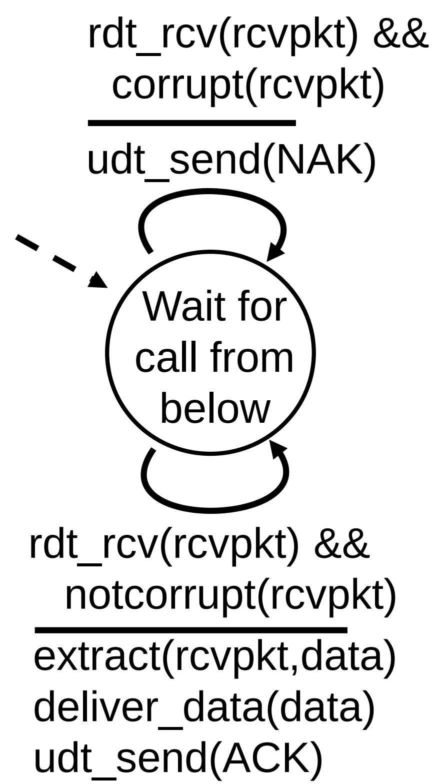
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - 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: FSM specification

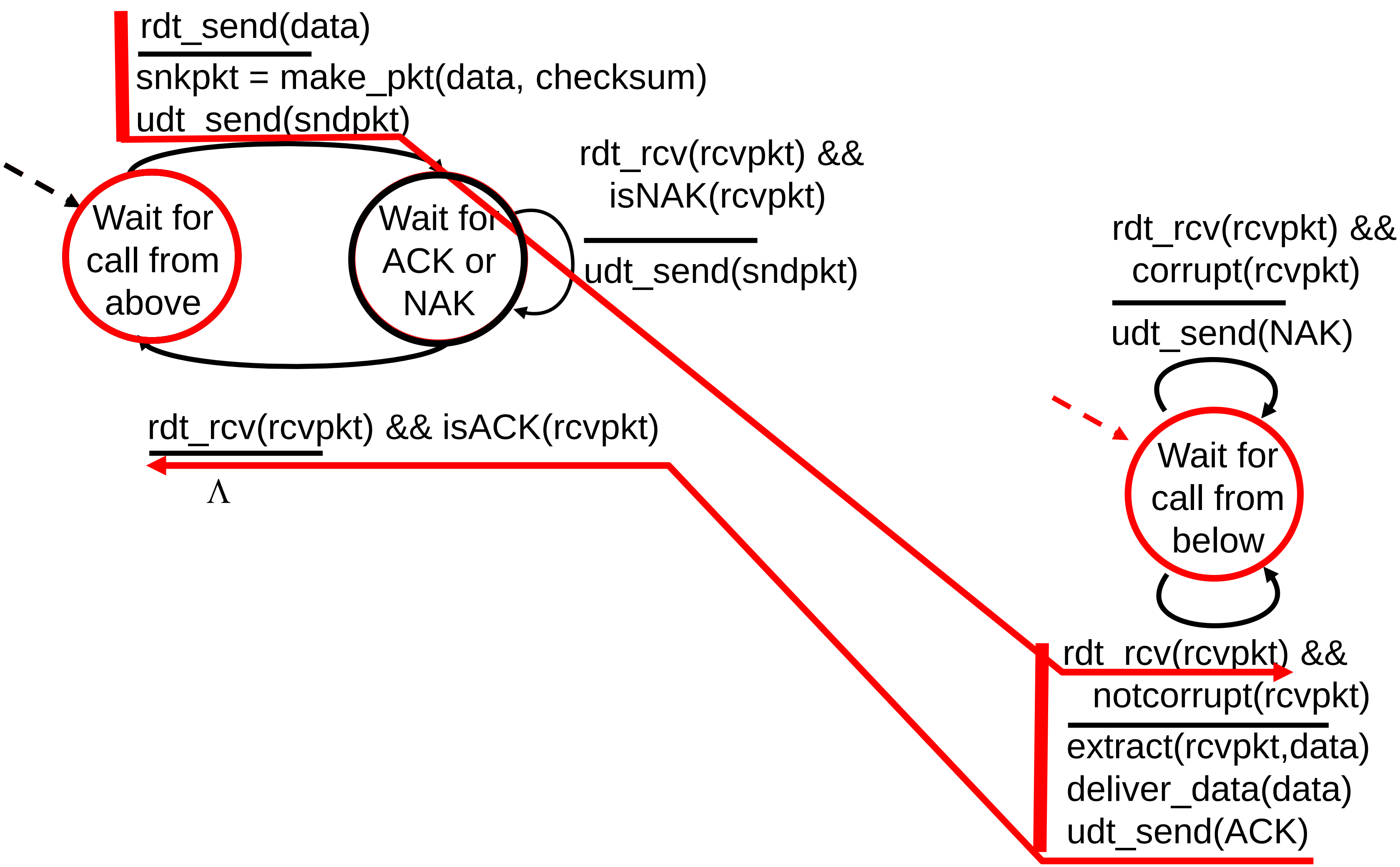



sender

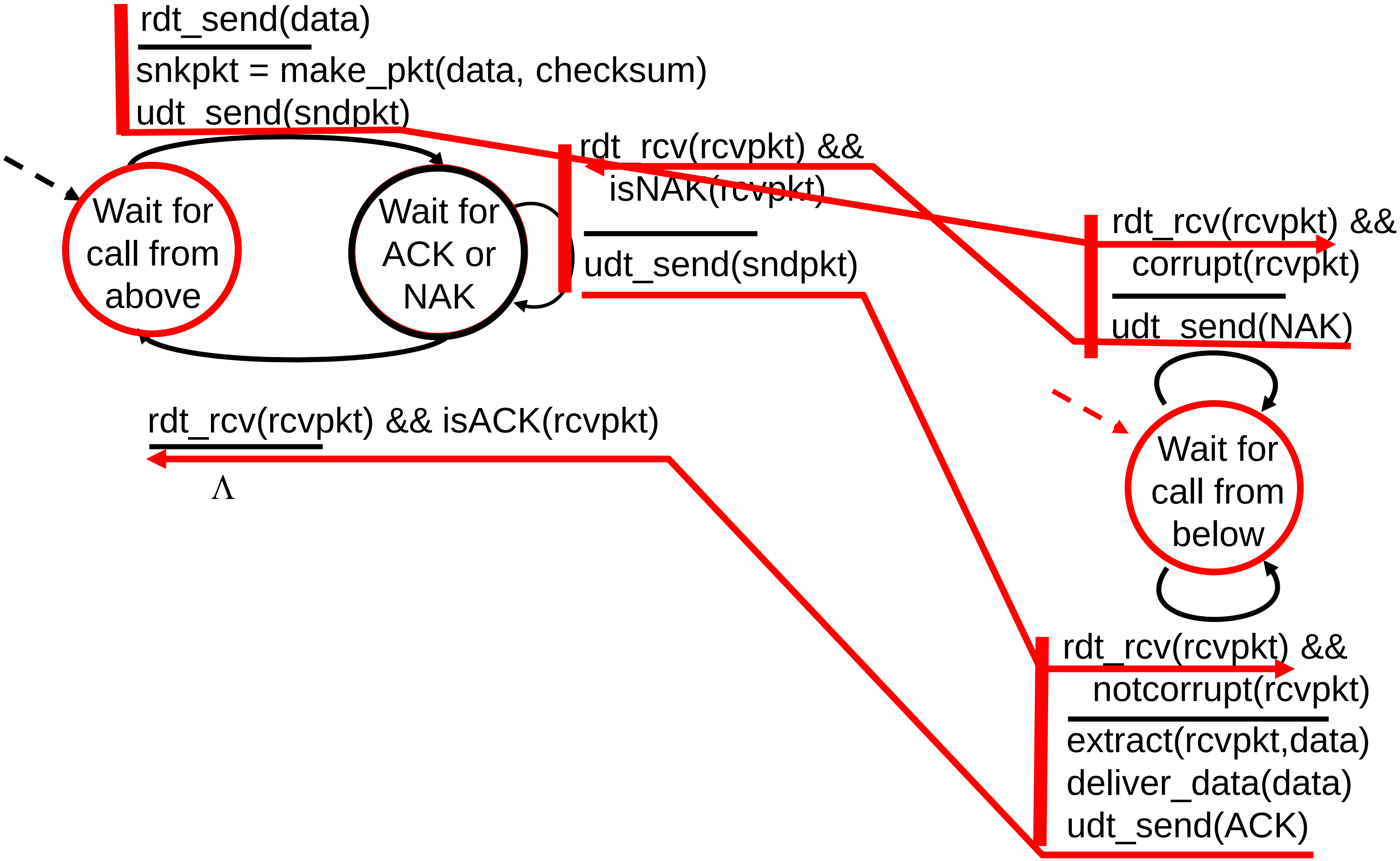
receiver



# 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

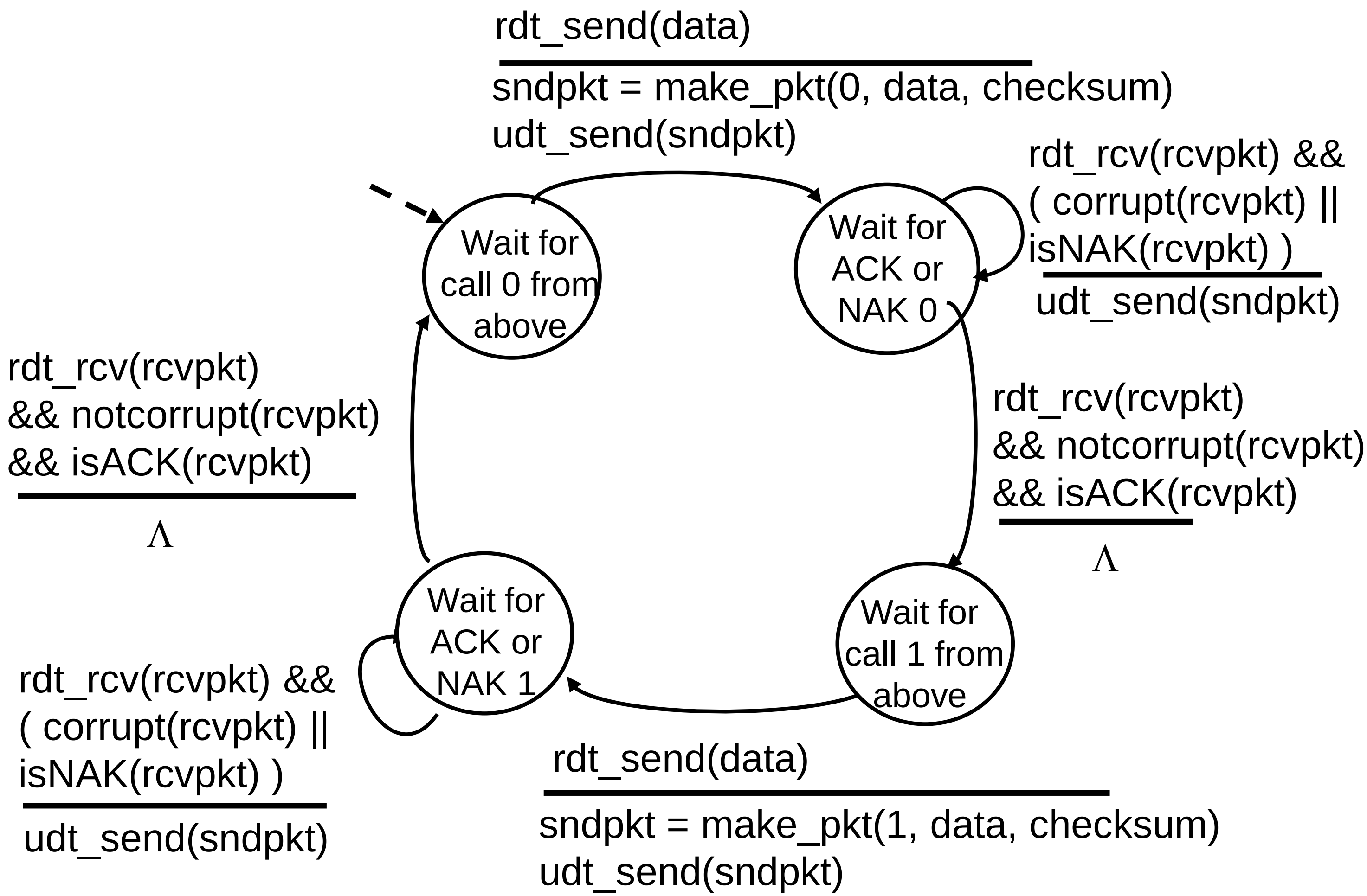
Handling duplicates:

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

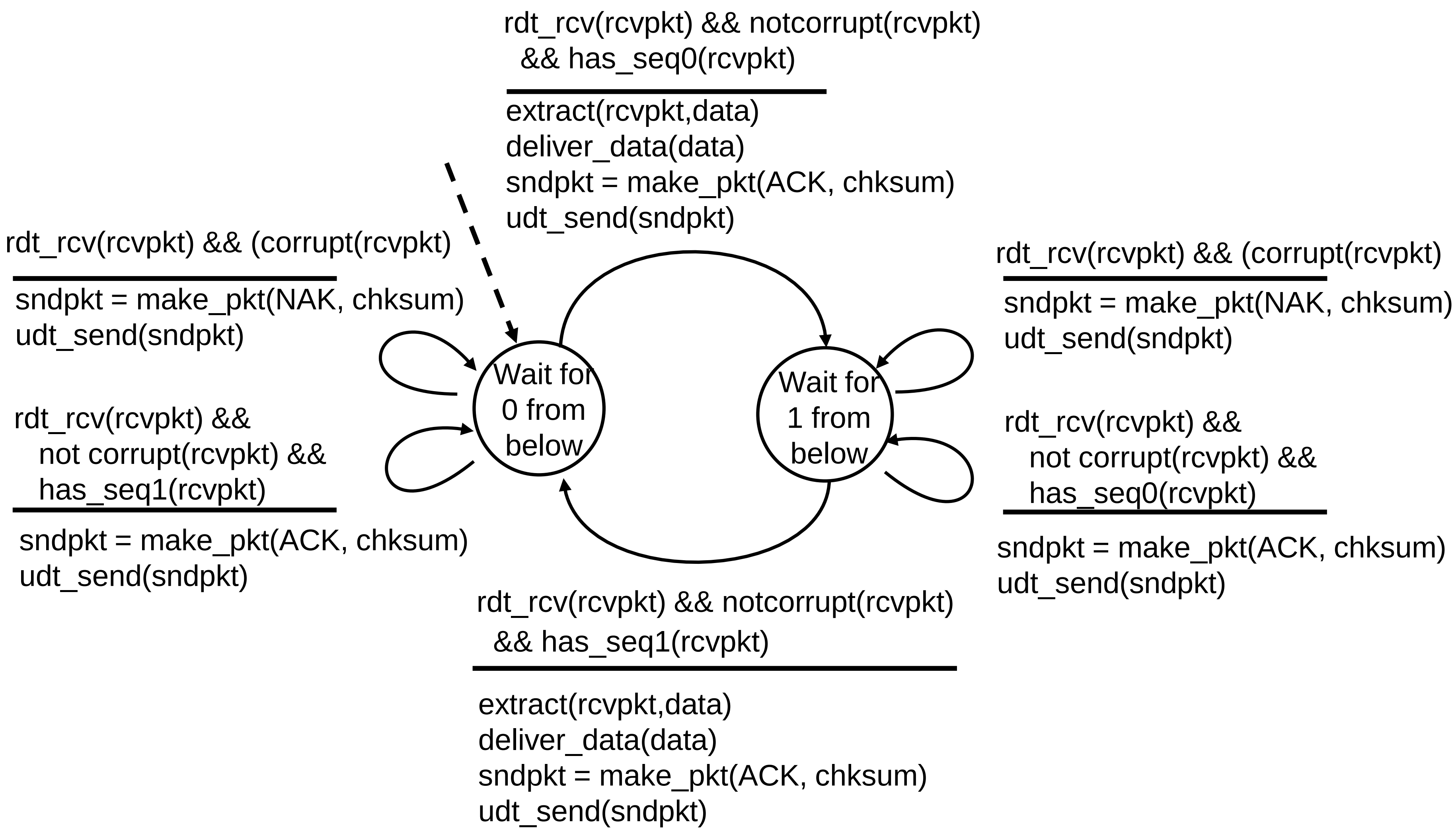
**stop and wait**

Sender sends one packet, then waits for receiver response

# rdt2.1: sender, 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. #

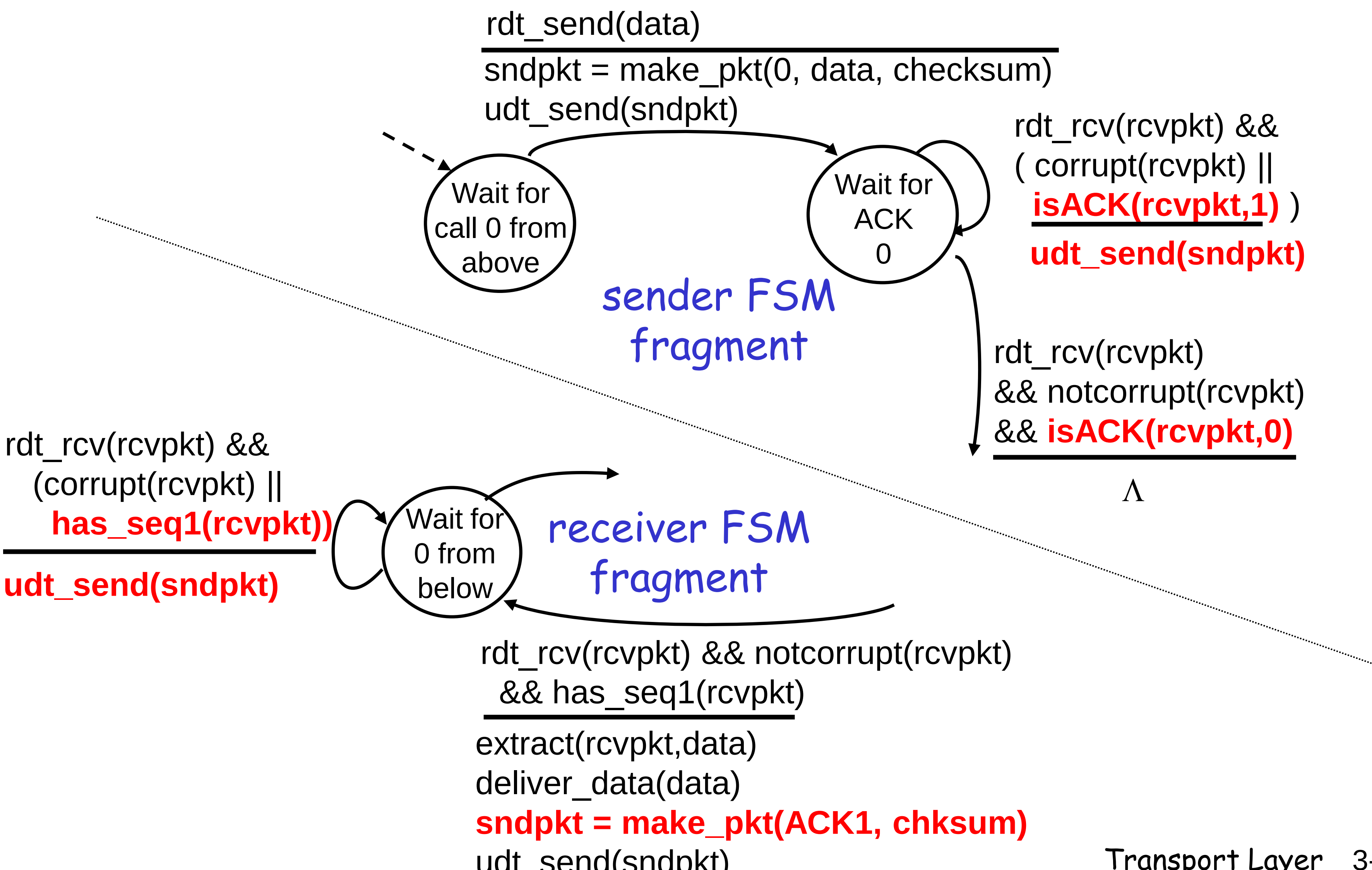
Receiver:

- 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

# 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

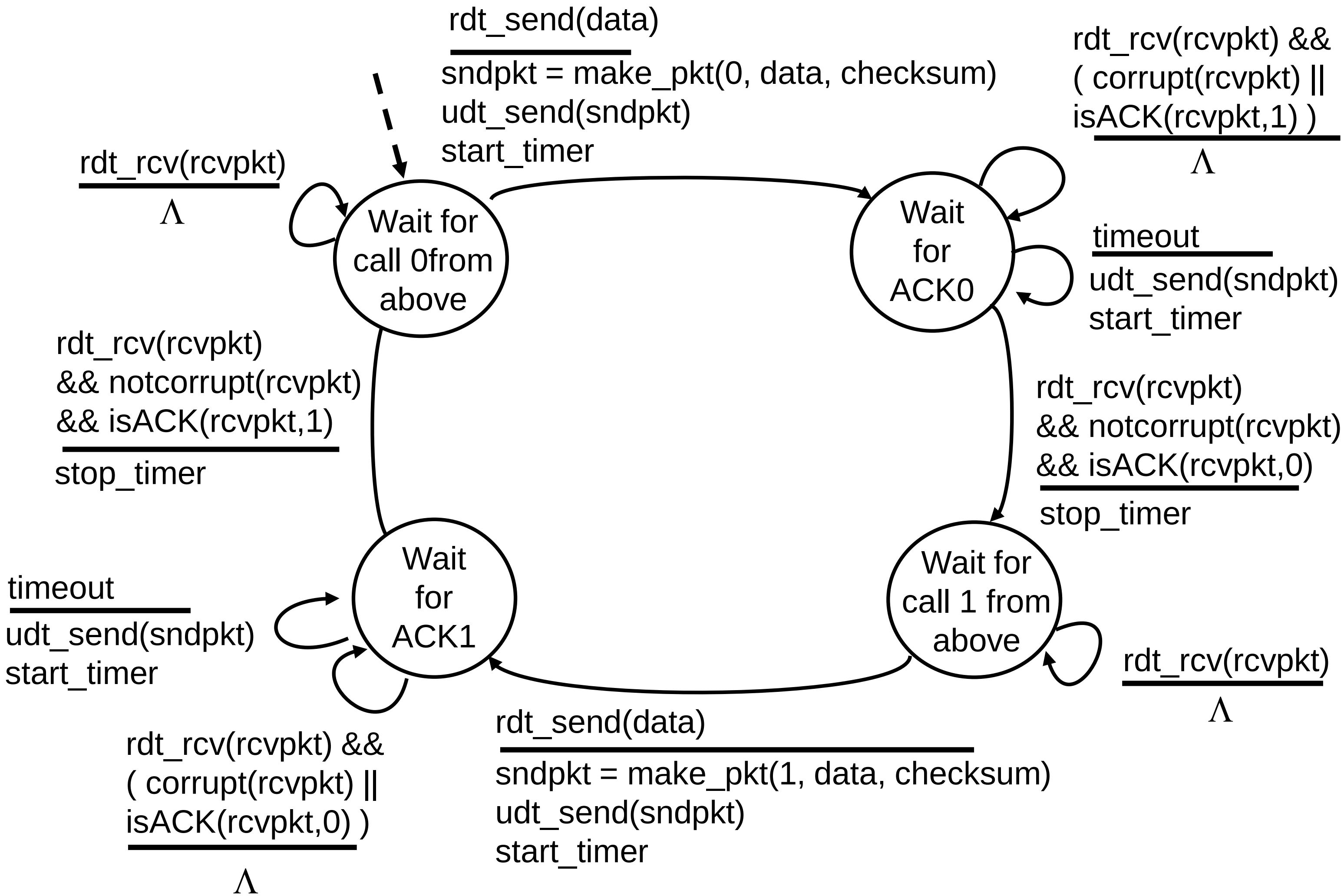
**New assumption:** underlying channel can also lose packets (data or ACKs)

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

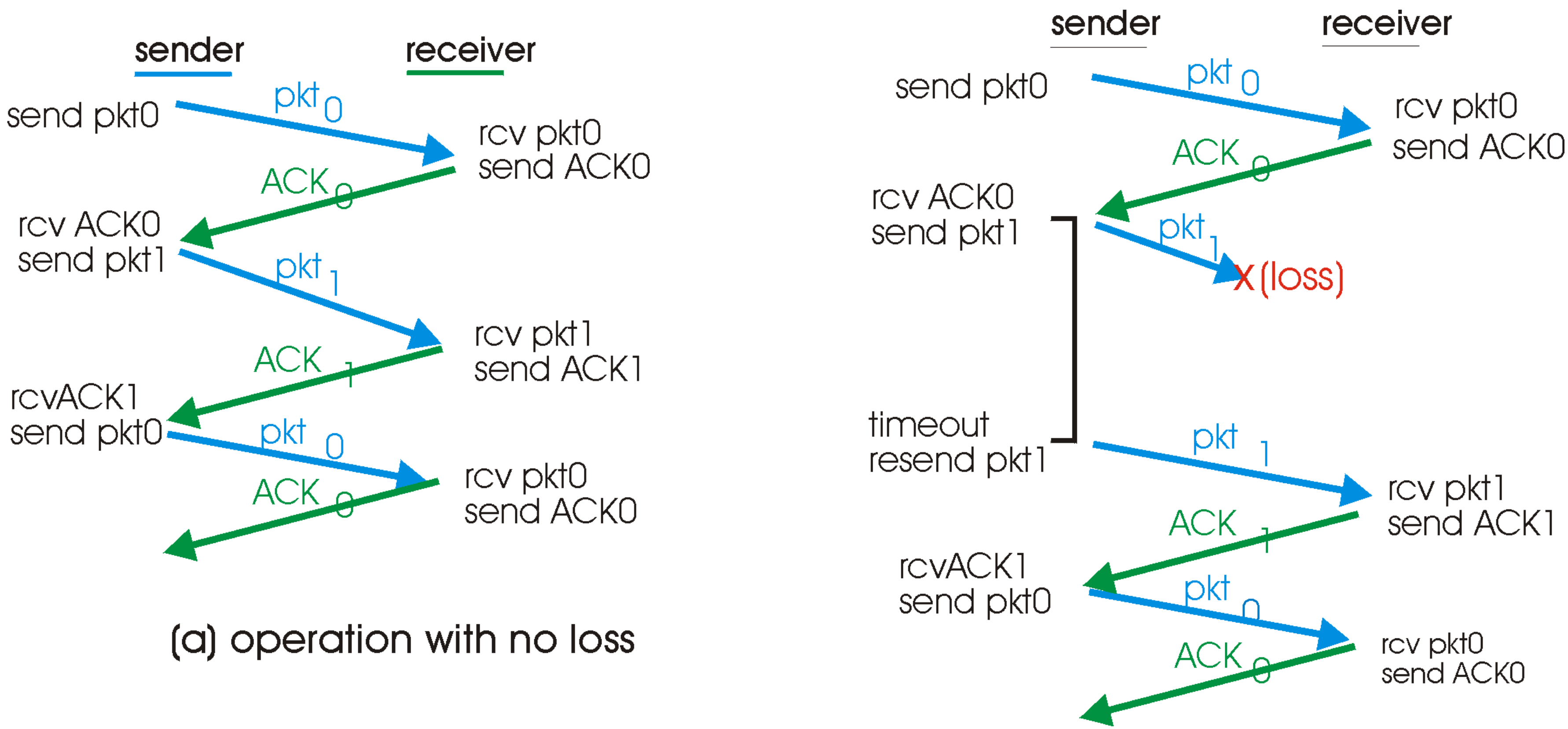
**Approach:** sender waits "reasonable" amount of time for ACK

- 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



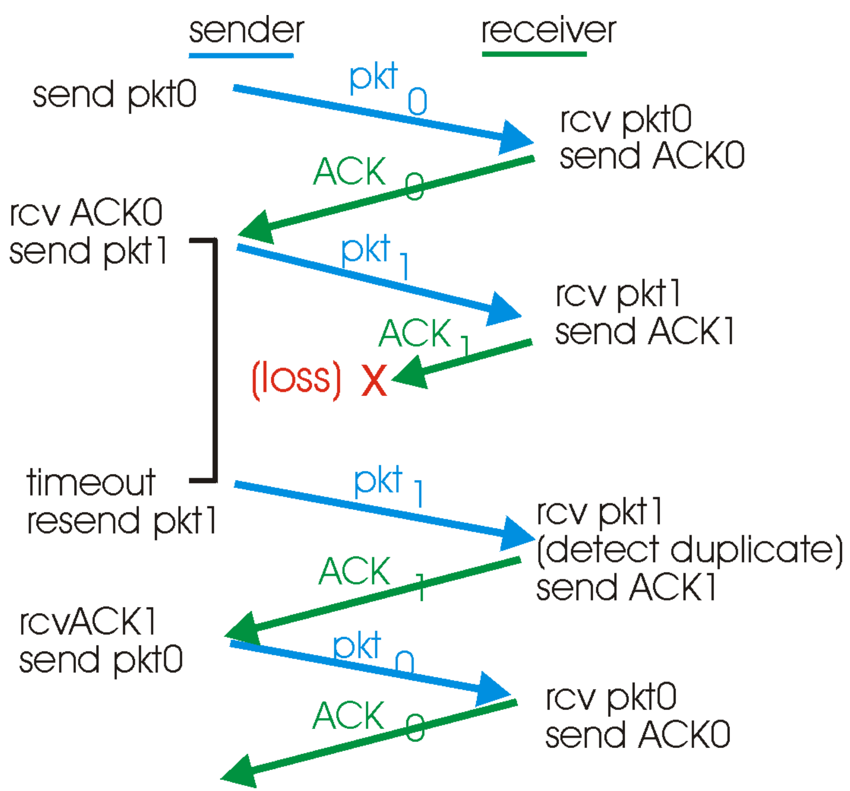
# rdt3.0 in action



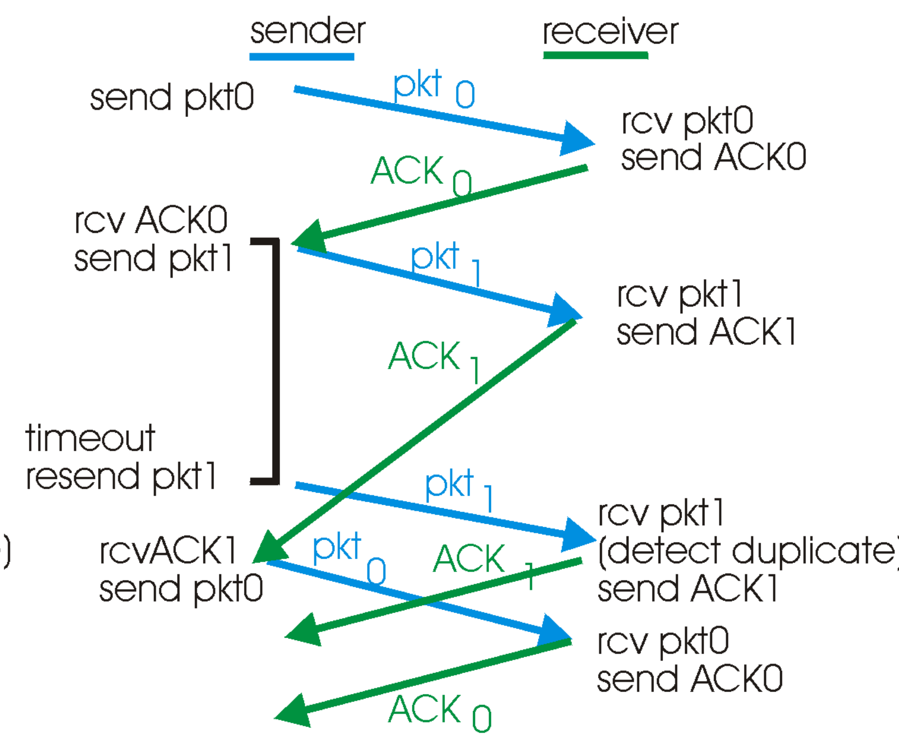

(a) operation with no loss

(b) lost packet

# rdt3.0 in action




(c) lost ACK




(d) premature timeout

# RDT小结

目标：在不可靠的信道上实现可靠数据传输

- 情况1——考虑数据可能出现bit错误。
  - 要求：1，接收端具有检查错误、ACK的能力，从而发现错误并告知发送端；2，发送端具有重传的能力。
  - 但是，发送端一旦具有重传的能力，就会带来新的问题——发送端重传的数据被接收端当做新的数据。因此，要引入序列号。
  - 最终的解决方案要整合三项机制:错误检查（Checksum）、ACK、序列号。

# RDT小结

目标：在不可靠的信道上实现可靠数据传输

- 情况2——在情况1的基础上，考虑丢包。
  - 要求：1，发送端要有检查丢包的能力：设置定时器，一旦定时器超时，就认为数据包丢失，并重传。
  - 但是，定时器超时并不一定意味着数据包丢失，有可能是因为数据包在网络中延迟太大。这样会造成接收端接收到重复的数据，好在序列号已经解决了这一问题。
  - 最终的解决方案要整合四项机制:错误检查（Checksum）、ACK、序列号、定时器

# RDT小结

目标：在不可靠的信道上实现可靠数据传输

- 是否还有第三种情况？类比一下网购快递的过程
- 是否考虑完全？四种情况的组合：发送端发现数据损坏、接收端发现数据损坏、发送端丢包、接收端丢包。
- 定时器应如何设定？

# Performance of rdt3.0

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

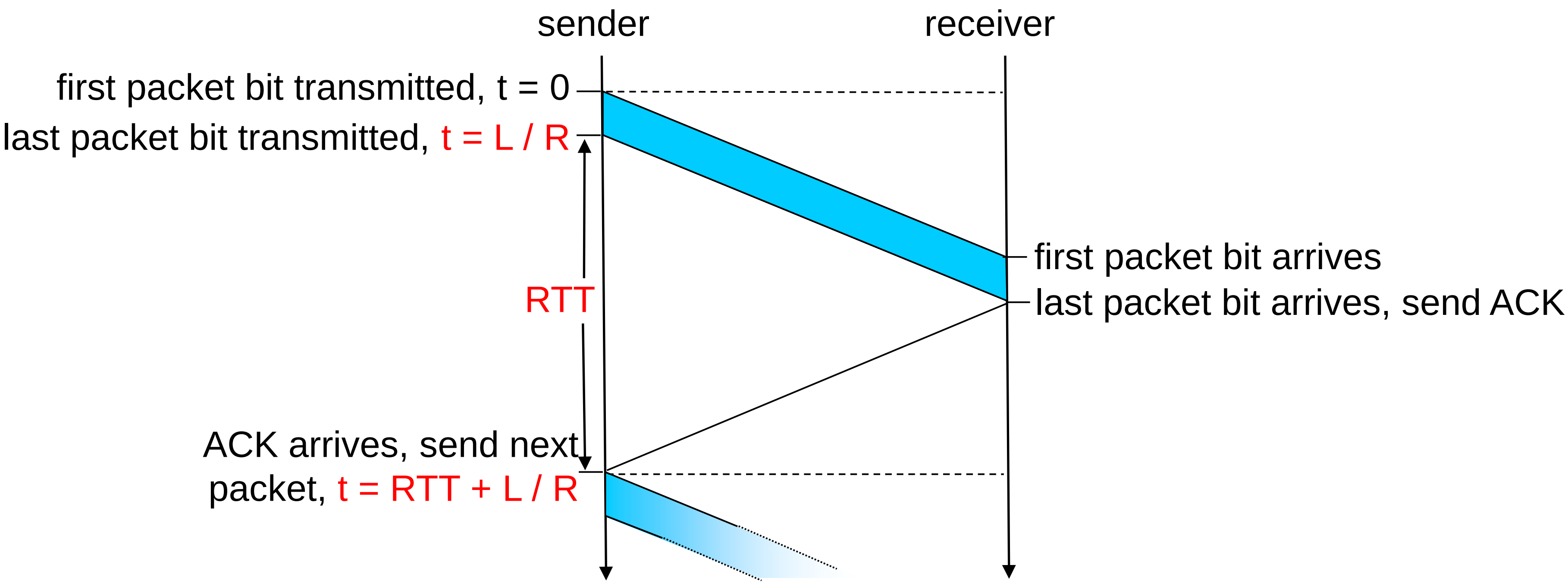
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\,\text{microseconds}$$

  - $U_{sender}$: 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!

# rdt3.0: stop-and-wait operation

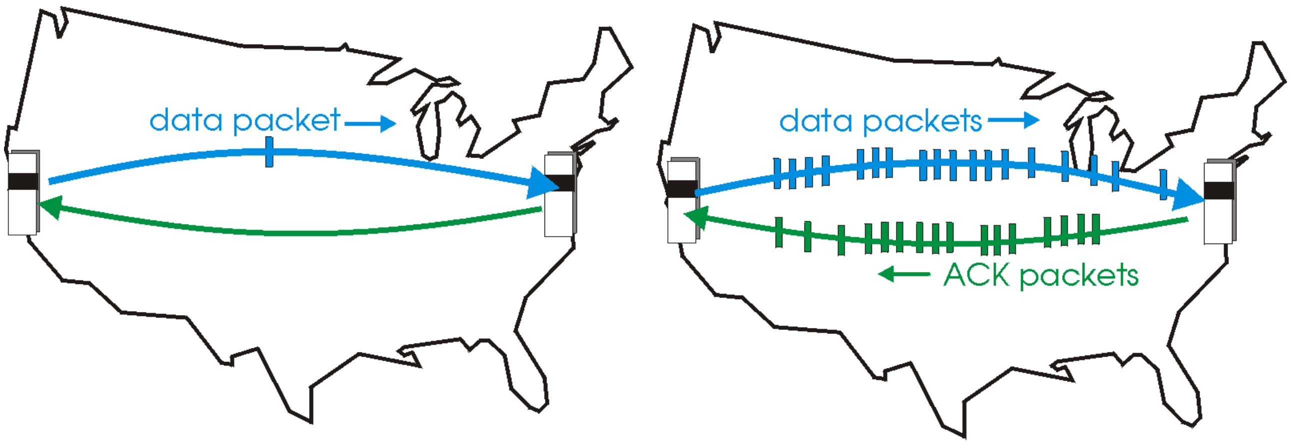



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# Pipelined protocols

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

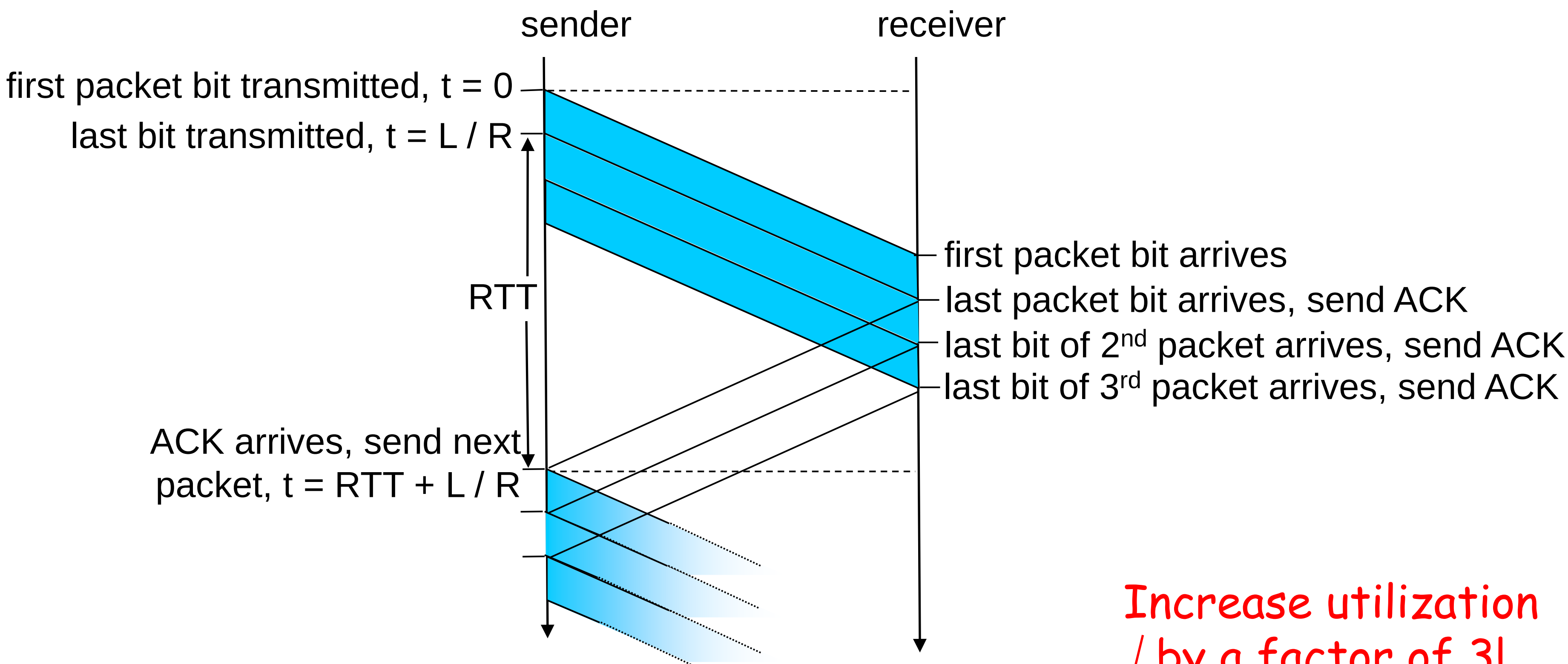
- range of sequence numbers must be increased
- buffering at sender and/or receiver




(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




Increase utilization by a factor of 3!

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

# 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

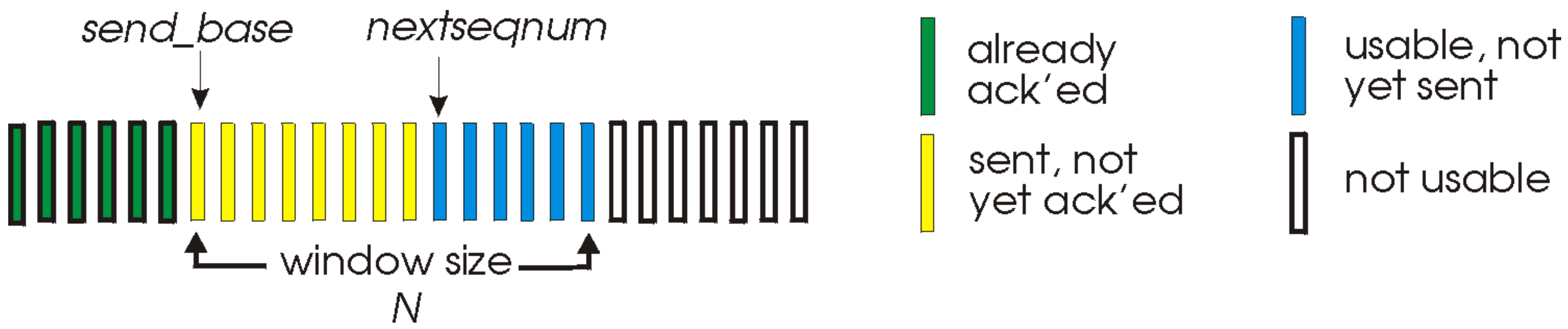
# Selective repeat: big picture

- 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

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

rdt_send(data)

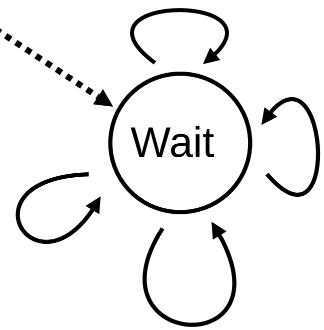
---

```
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)
```

$\Lambda$

---

base=1
nextseqnum=1




timeout

---

```
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

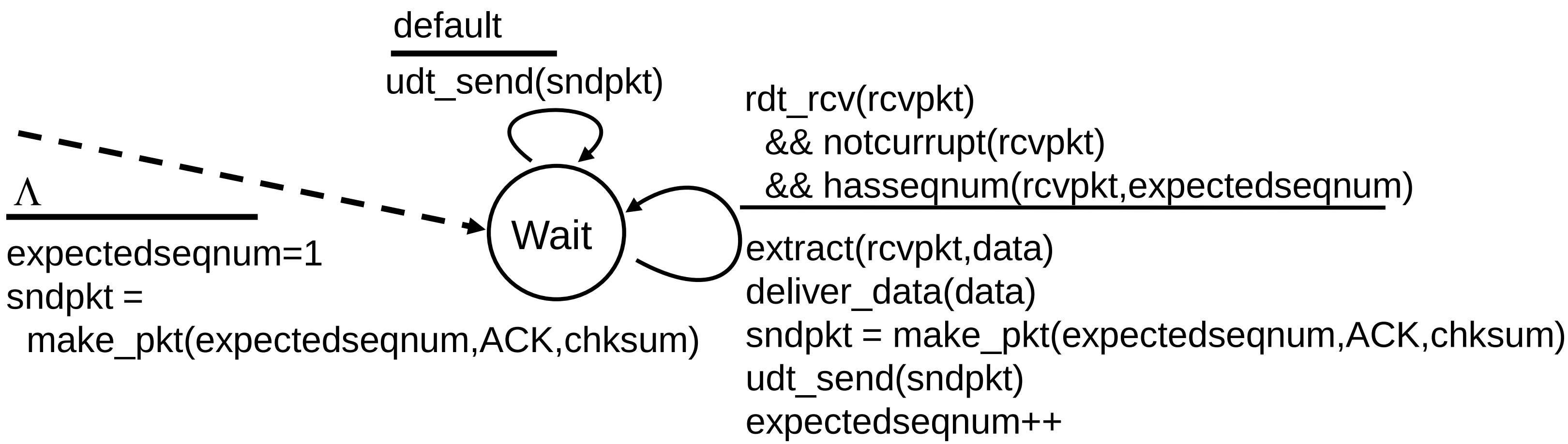
---

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

---

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

# GBN: receiver extended FSM

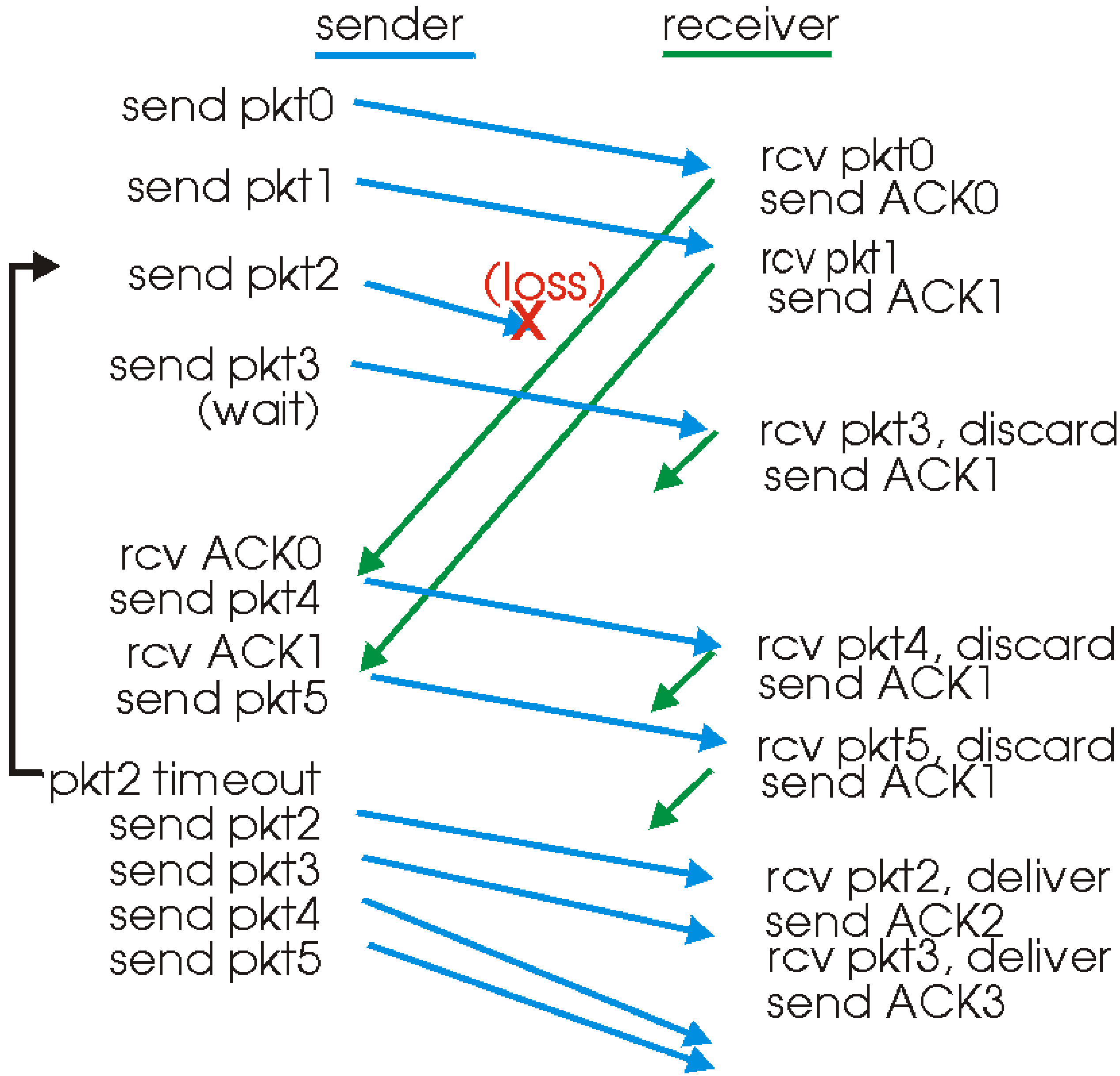



ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #

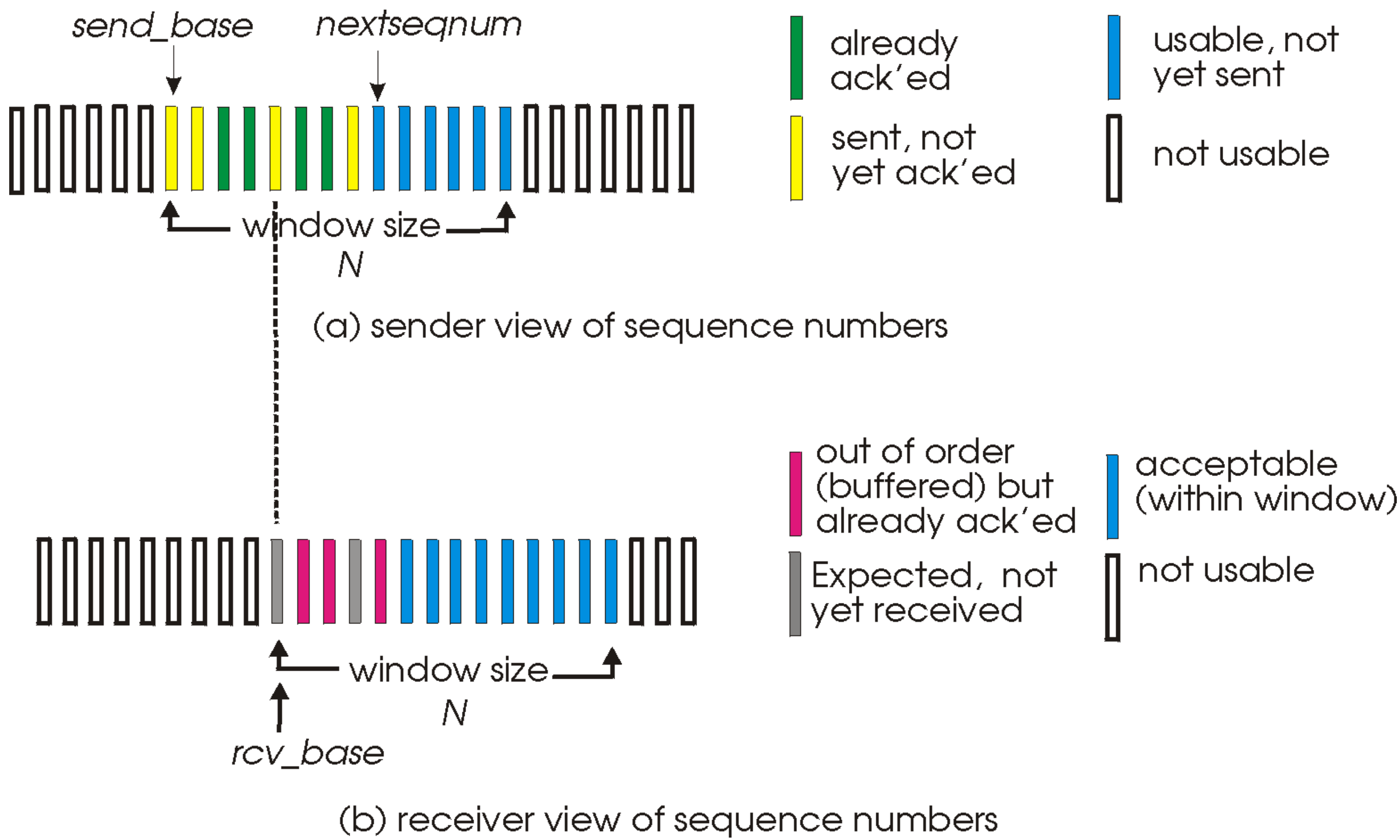
# GBN in action



# 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
- 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

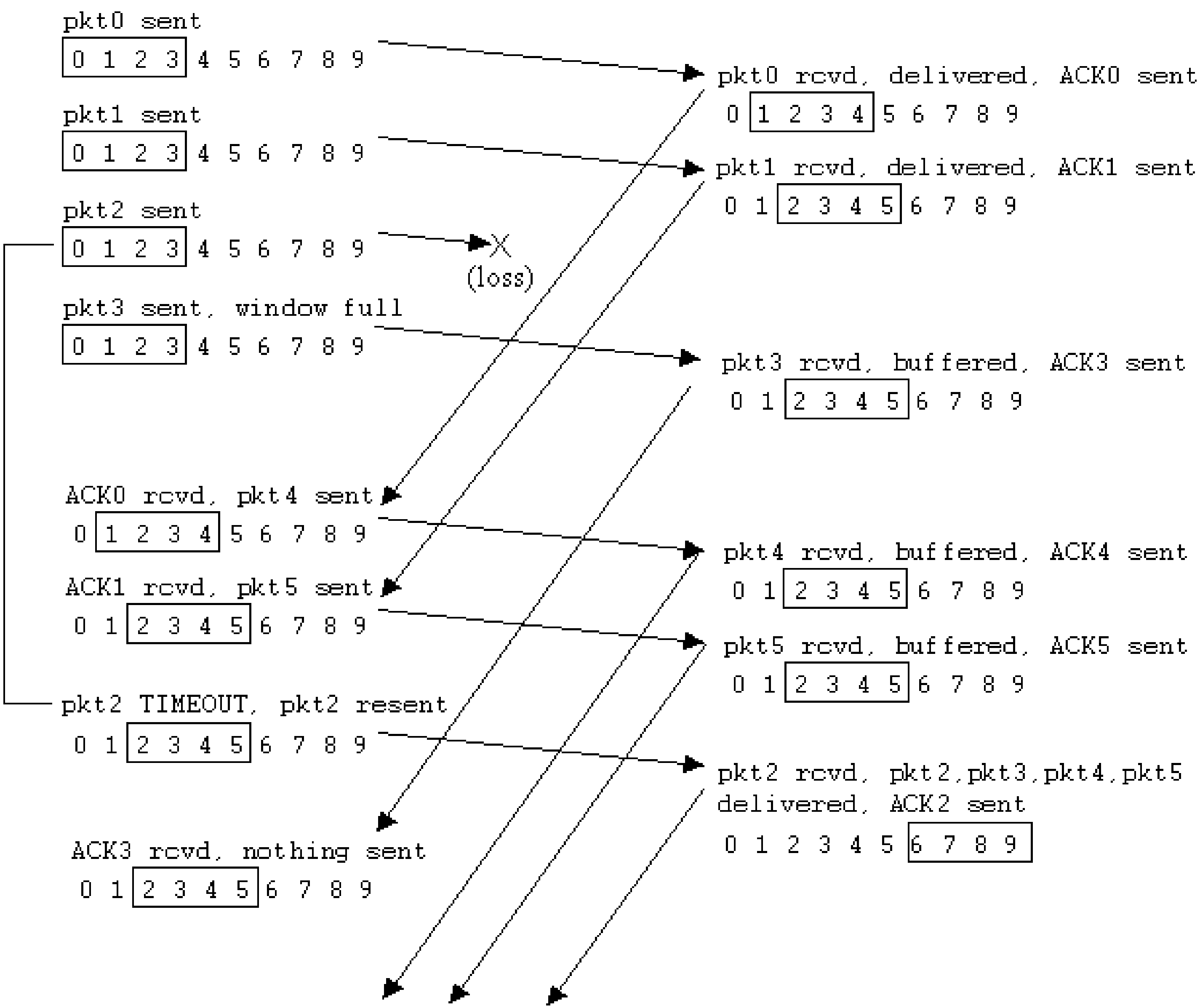
**pkt n in** [rcvbase-N,rcvbase-1]

- ACK(n)

**otherwise:**

- ignore

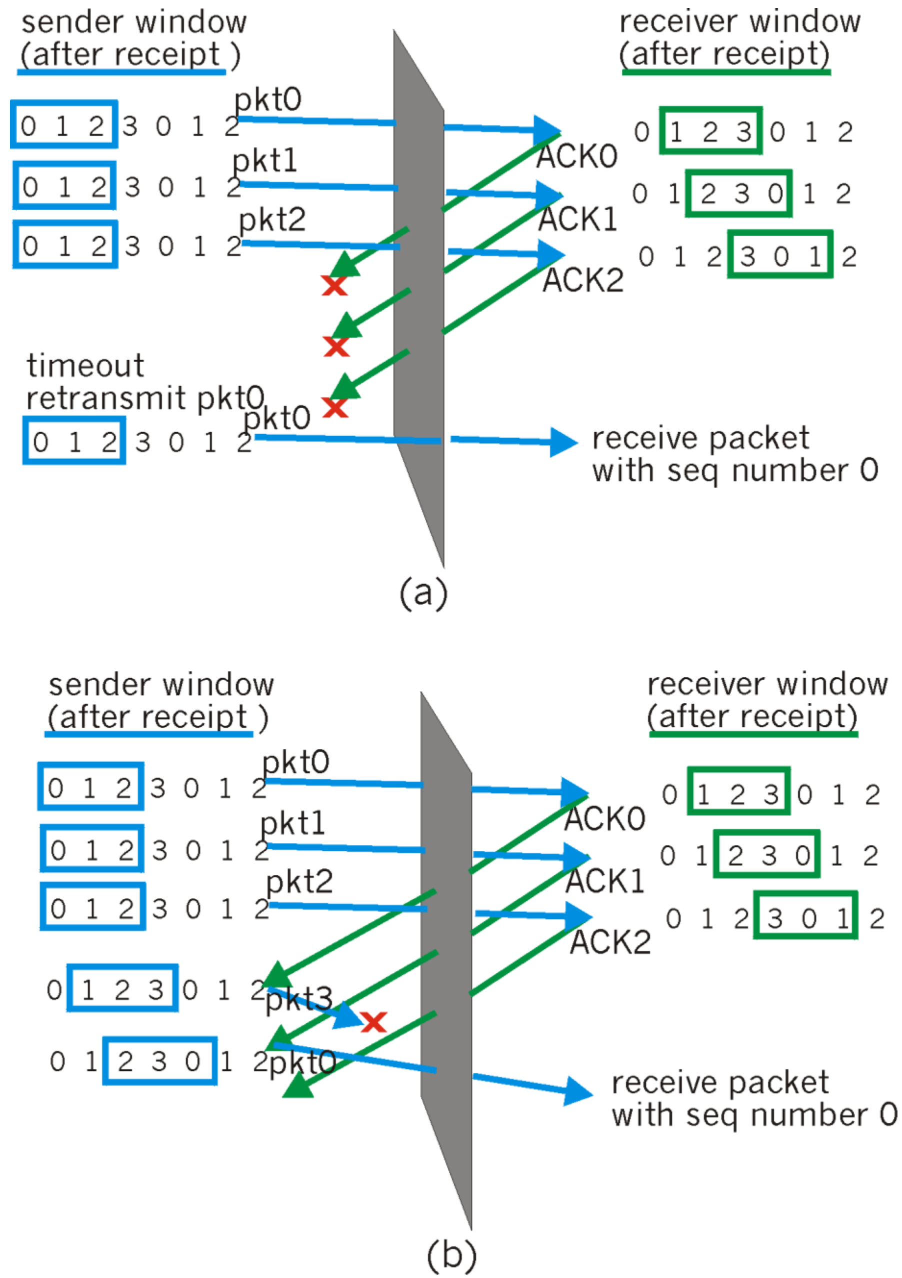
# Selective repeat in action



# 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?

# GBN、SR小结

| BASIS FOR COMPARISON | GO-BACK-N | SELECTIVE REPEAT |
| --- | --- | --- |
| Basic | Retransmits all the frames that sent after the frame which suspects to be damaged or lost. | Retransmits only those frames that are suspected to lost or damaged. |
| Bandwidth Utilization | If error rate is high, it wastes a lot of bandwidth. | Comparatively less bandwidth is wasted in retransmitting. |
| Complexity | Less complicated. | More complex as it require to apply extra logic and sorting and storage, at sender and receiver. |
| Window size | N-1 | <= (N+1)/2 |

# GBN、SR小结

| | | |
|---|---|---|
| Sorting | Sorting is neither required at sender side nor at receiver side. | Receiver must be able to sort as it has to maintain the sequence of the frames. |
| Storing | Receiver do not store the frames received after the damaged frame until the damaged frame is retransmitted. | Receiver stores the frames received after the damaged frame in the buffer until the damaged frame is replaced. |
| Searching | No searching of frame is required neither on sender side nor on receiver | The sender must be able to search and select only the requested frame. |
| ACK Numbers | NAK number refer to the next expected frame number. | NAK number refer to the frame lost. |
| Use | It more often used. | It is less in practice because of its complexity. |

# GBN、SR小结

目标：提高效率，都属于滑动窗口方法

- 与之相对，RDT可以认为是一种Stop-And-Wait方法，窗口大小为1。
- GBN与SR的通信效率一致，GBN适用于网络条件较好的情况，SR适用于网络条件较差的情况。
- GBN与SR的动画演示：http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/