Last Lecture

- Peer-to-Peer (P2P) Applications

This Lecture

- Overview of the transport layer
- Principles of Reliable Data Transfers

The Transport Layer

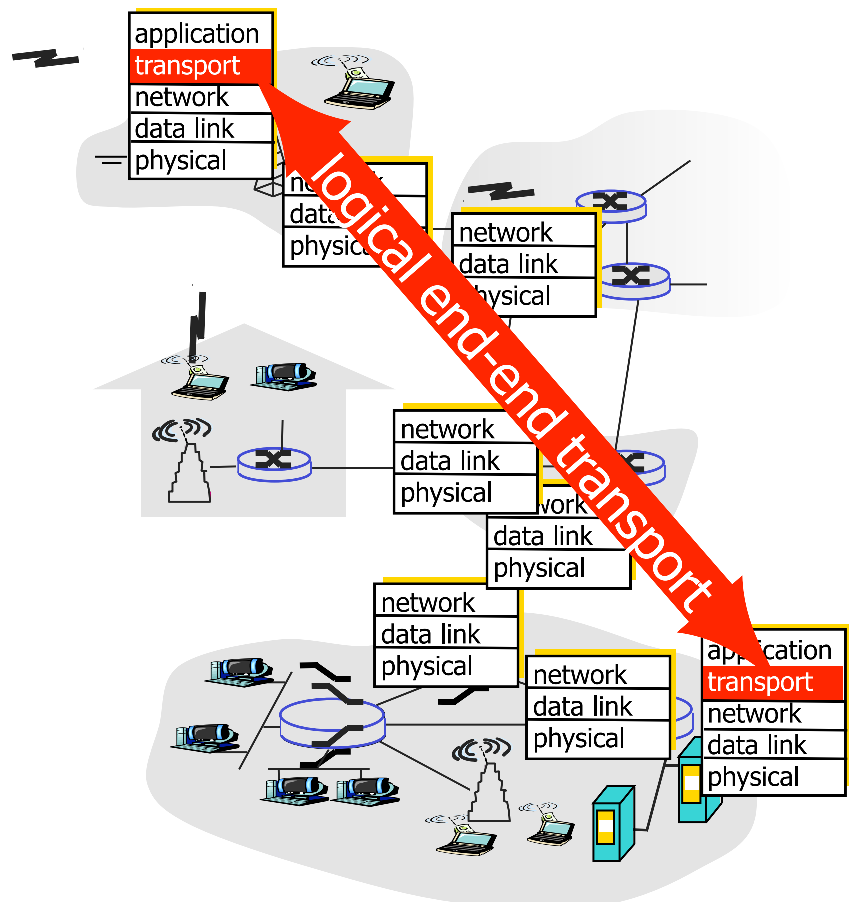
- Provide services to applications
  - What kind of services?
  - How to implement them?
- Make use of services provided by the network layer
  - Network gives *best-effort* packet delivery service
- Help networks out too
  - Don't pump too much data in if networks can't handle, i.e. *congestion control*

What Services to Provide to Applications?

- Difficult to decide, because
  - Can't envision all future applications
  - Even current applications are too diverse in requirements
  - Can't provide services which can't be implemented
- Currently, two main services are
  - TCP: reliable, connection-oriented
  - UDP: unreliable, connectionless
- There are *many* other proposals & implementations, but not widespread
  - RTP, RSTP for real-time streaming
  - SCTP, DCCP: somewhere between UDP and TCP
- Many applications have transport functionalities built-in

Services Provided by the Network Layer

- Depend on the network
  - *Datagram network*: service sucks! Just best-effort
  - *ATM network*: connection-oriented, virtual-circuit, some QoS guarantee
  - …
- A general transport protocol can only assume that network service is best effort



What Is Best-Effort Again?

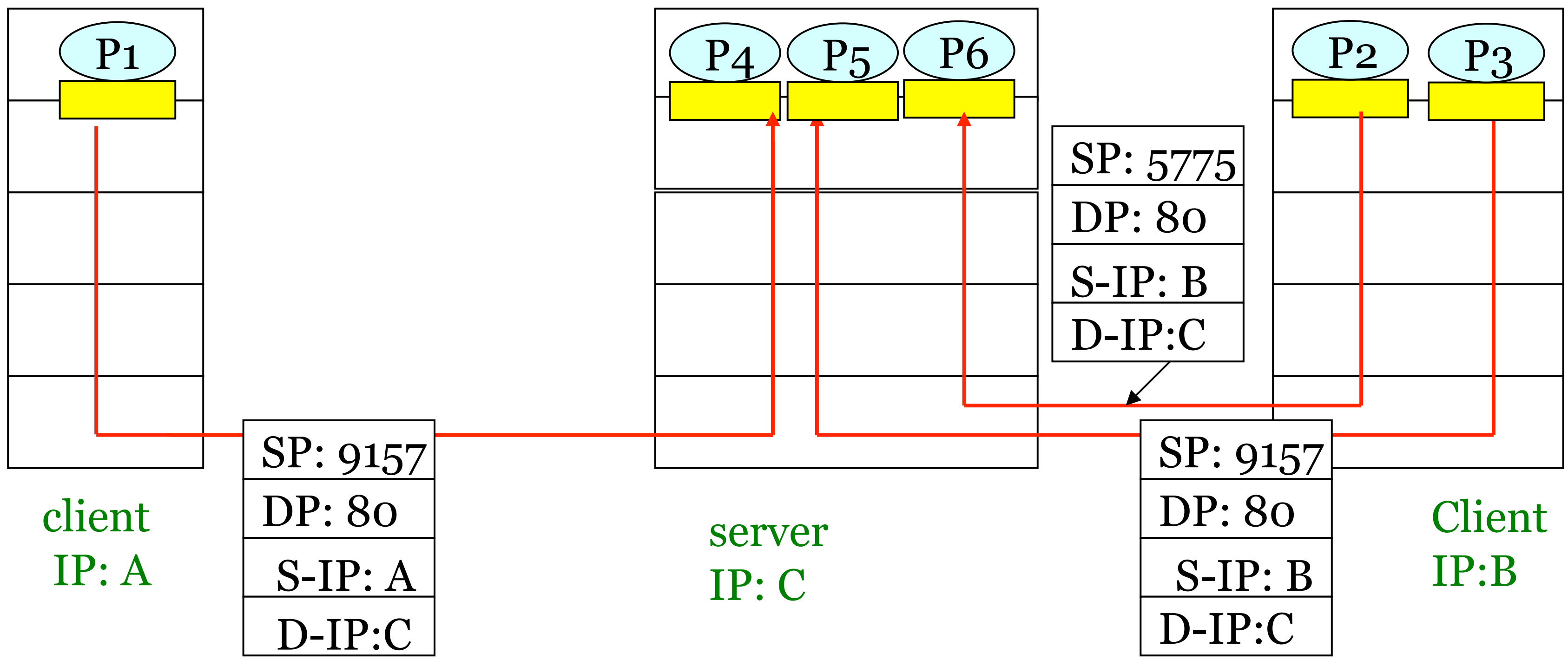
- Packets may be *corrupted*
- Packets may be *lost*
- Packets may be *duplicated*
- Packets may be delivered *out of order*
- Inter-arrival times can vary wildly
- End-to-end delay may vary wildly

We Will Focus on TCP Alone

- *Multiplexing & de-multiplexing*
- *Reliable data transfer (& try to be efficient too)*
- *Connection-oriented*
- *Flow control*
- Help network with *congestion control & avoidance*
- **No** guarantee on timing (delay, jitter, bandwidth)

- TCP is sufficiently complex to illustrate fundamental ideas
- Services suitable for media streaming like RTP, RSTP, etc. are still active research topics!

Multiplexing & De-multiplexing



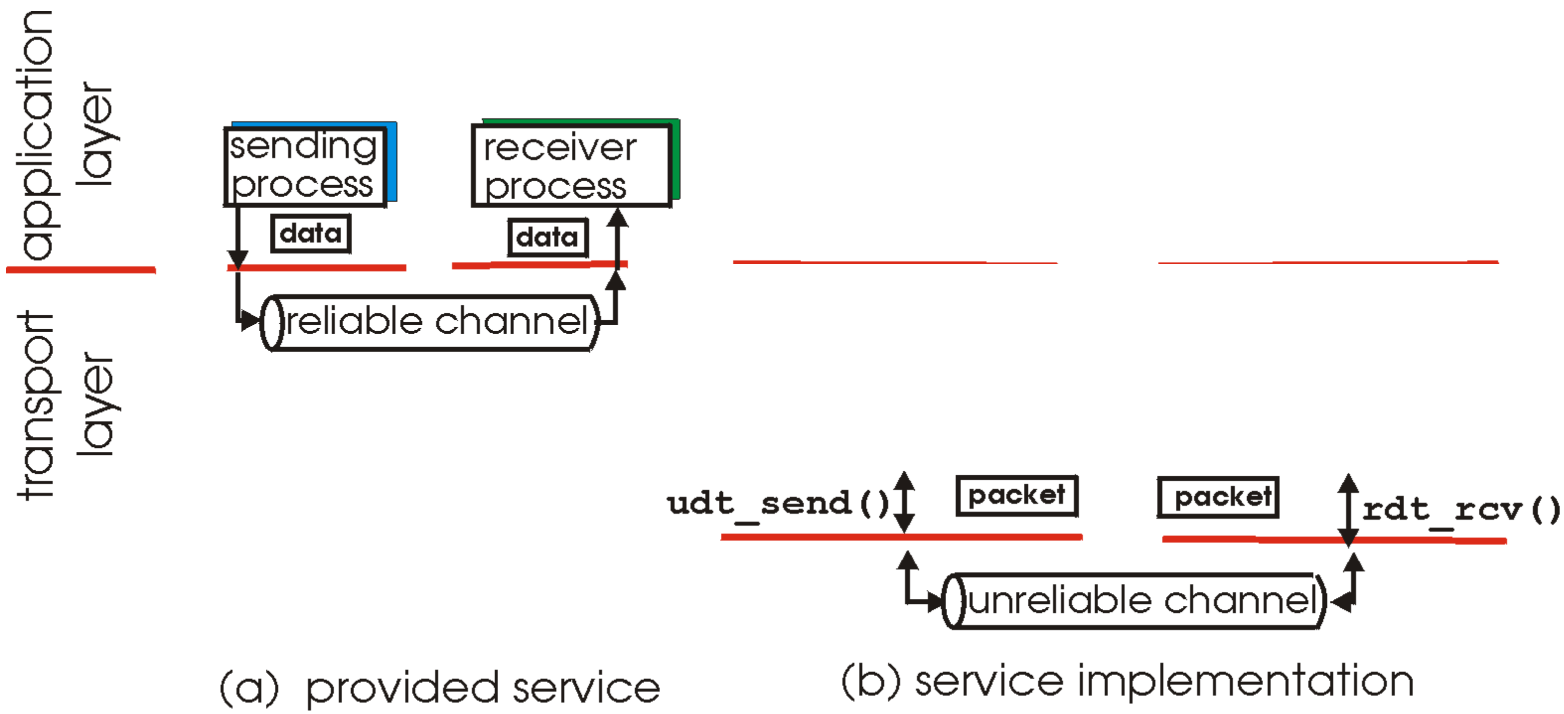
Principles of Reliable Data Transfer

- Before looking at how TCP does it, let's try to design a *reliable data transfer protocol* (RDT) ourselves

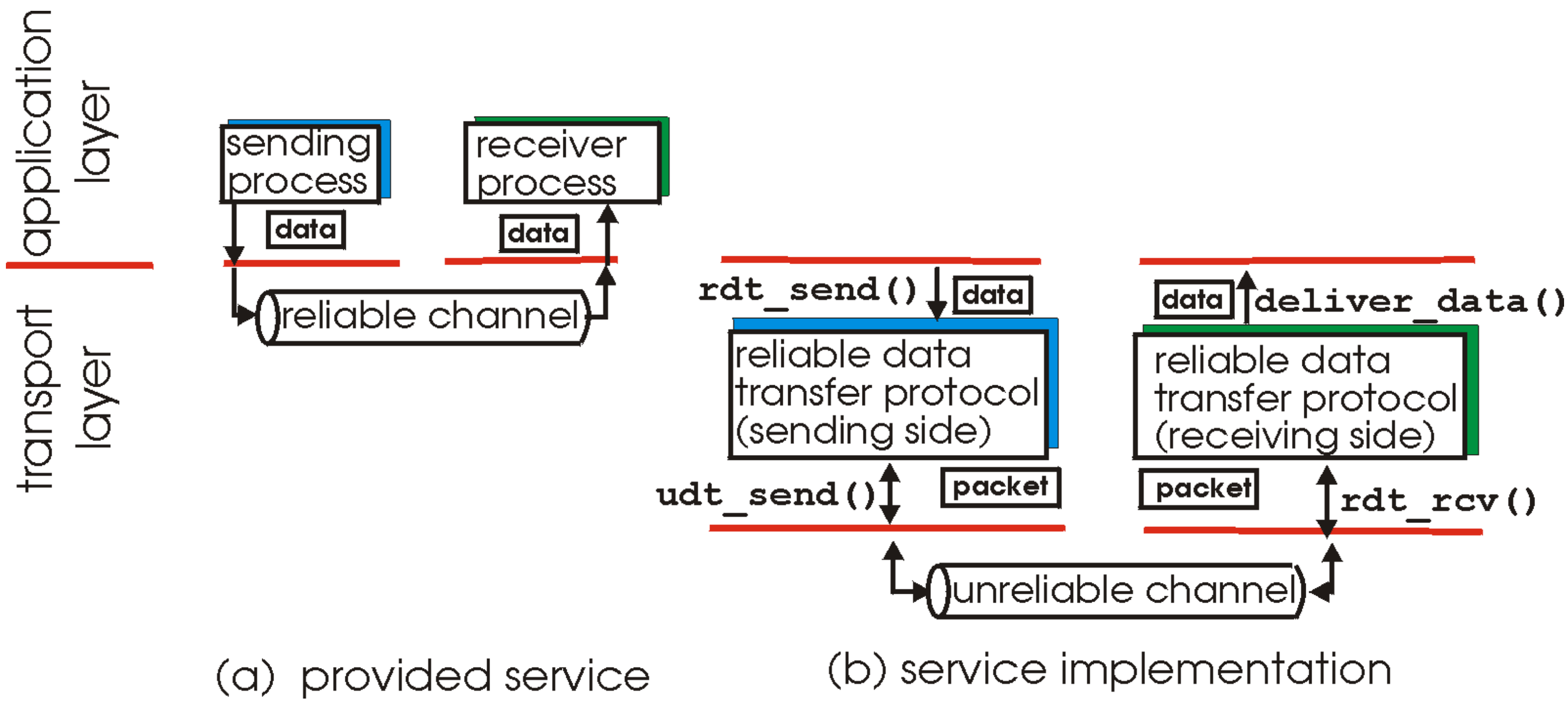
- *Main question*: *how to reliably transfer data* when
  1. Network can *corrupt* packets (*bit error*)
  2. Network can *lose* packets
  3. Network can deliver *duplicates*
  4. Network can deliver packets *out of order*

- We address the 4 problems one by one, in that order

The Bird-Eye View



The Bird-Eye View

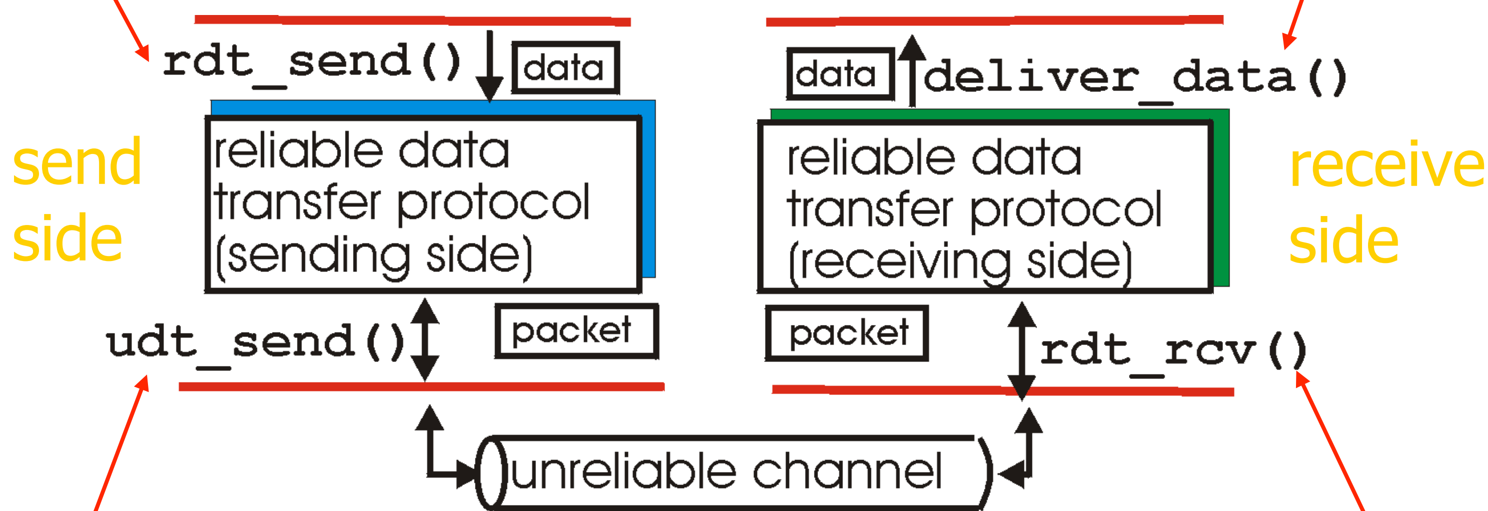


- Characteristics of unreliable channel will determine complexity of the RDT protocol

RDT: 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 layer



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

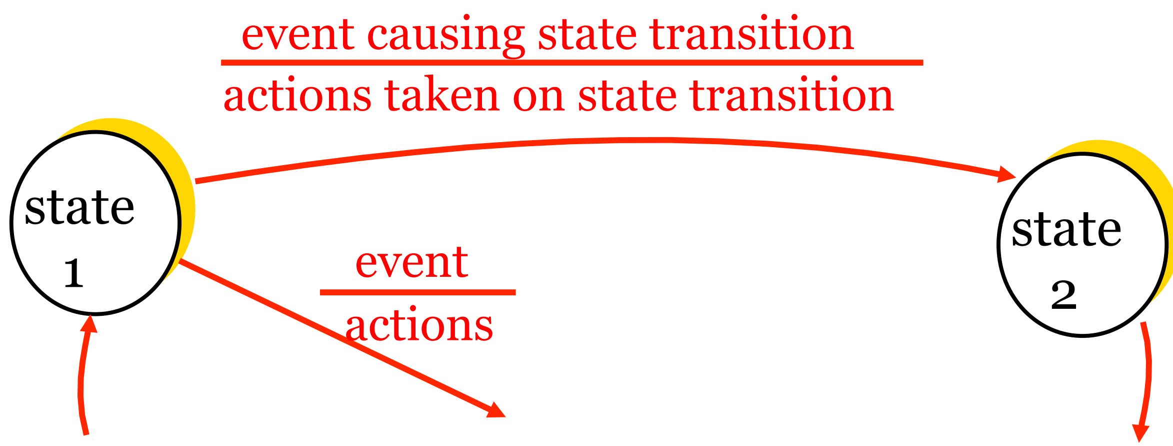
rdt_rcv(): called when packet arrives on rcv-side of channel

Finite State Machines

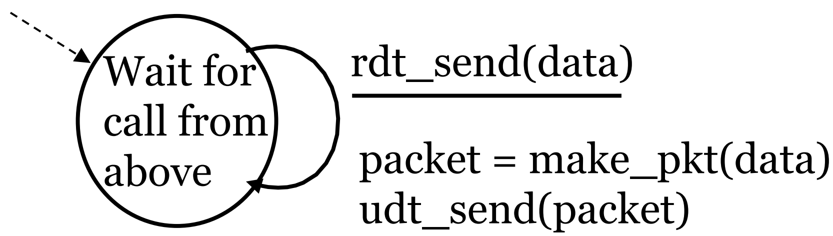
- FSMs are convenient for specifying protocol's behaviors

- FSM notations:

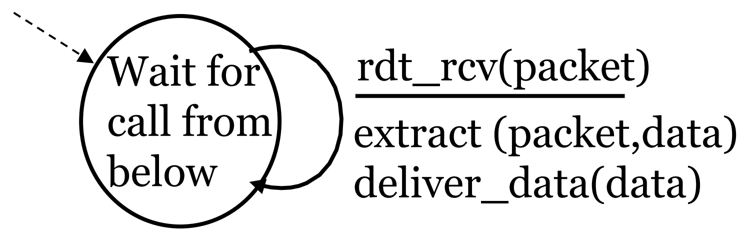
state: when in this "state" next state uniquely determined by next event



RDT 1.0: Perfectly Reliable Channel



Sender



Receiver

- *Next*: suppose network can corrupt packets (*i.e.*, *bit errors* may occur)
- *But still*: no packet loss, no out of order packets, no duplicate packets

RDT 2.0: Dealing with Bit-Errors

1. How to detect that a packet has been corrupted?
   - *Error-detecting code* (e.g. checksuming)
2. What to do when corrupted packet received?
   - *Error-correcting code*
     - May not always work, depend on how much error
     - Too much (time/space) overhead if error is rare
     - Decoding time might be too long
   - Tell sender to *retransmit*
     - ACK: acknowledgement of a good packet
     - NACK: acknowledgement of a bad packet

- New mechanisms in RDT 2.0:
  - Error detection
  - ARQ – *automatic repeat request*

Error Detection

- *Problem*: detect bit errors in packets (frames)
- *Solution*: add *extra* bits to each packet
- *Goals*:
  - Reduce overhead (number of redundancy bits)
  - Increase the number and the type of bit error patterns that can be detected
- *Examples*:
  - Two-dimensional parity
  - (Internet) Checksum
  - Later (when we discuss data link layer)
    - Cyclic Redundancy Check (CRC)
    - Hamming Codes

Parity

- Even parity
  - Add a parity bit to 7 bits of data to make an even number of 1's

0110100	**1**
1011010	**0**

- How many bits of error can be detected by a parity bit?
- What's the overhead?

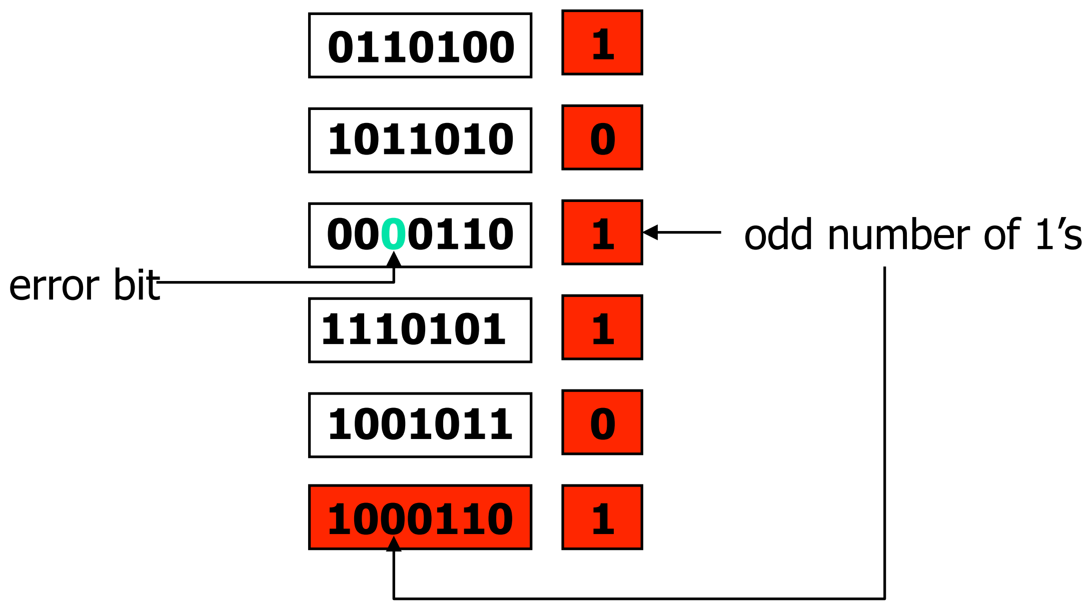
Two-dimensional parity

- Add one extra bit to a 7-bit code such that the number of 1’s in the resulting 8 bits is even (for even parity, and odd for odd parity)
- Add a parity byte for the packet
- Example: five 7-bit character packet, even parity

0110100	1
1011010	0
0010110	1
1110101	1
1001011	0
1000110	1

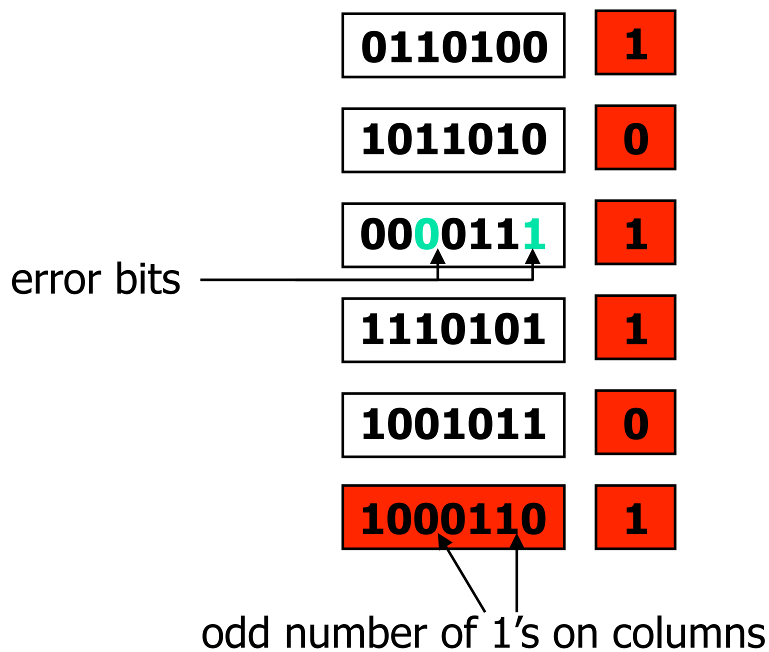
Two-dimensional parity detection capability

- All 1-bit errors
- Example:



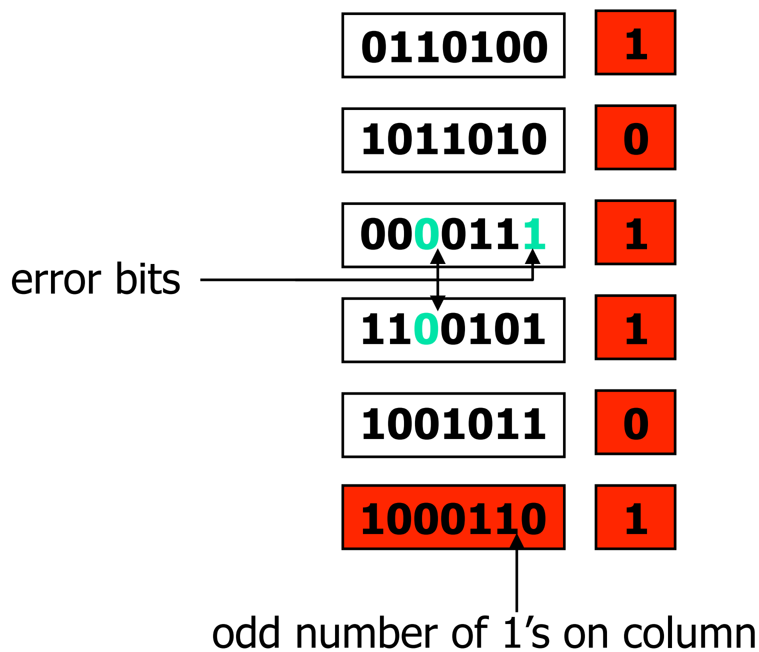
Two-dimensional parity detection capability

- All 2-bit errors
- Example:



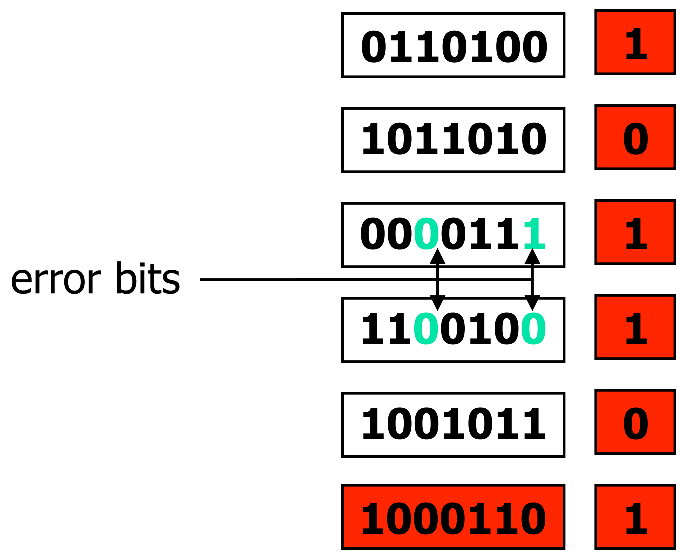
Two-dimensional parity detection capability

- All 3-bit errors
- Example:



Two-dimensional parity detection capability

- Most 4-bit errors
- Example of 4-bit error that is not detected:



How many errors can this code correct?

Internet Checksum [RFC1071]

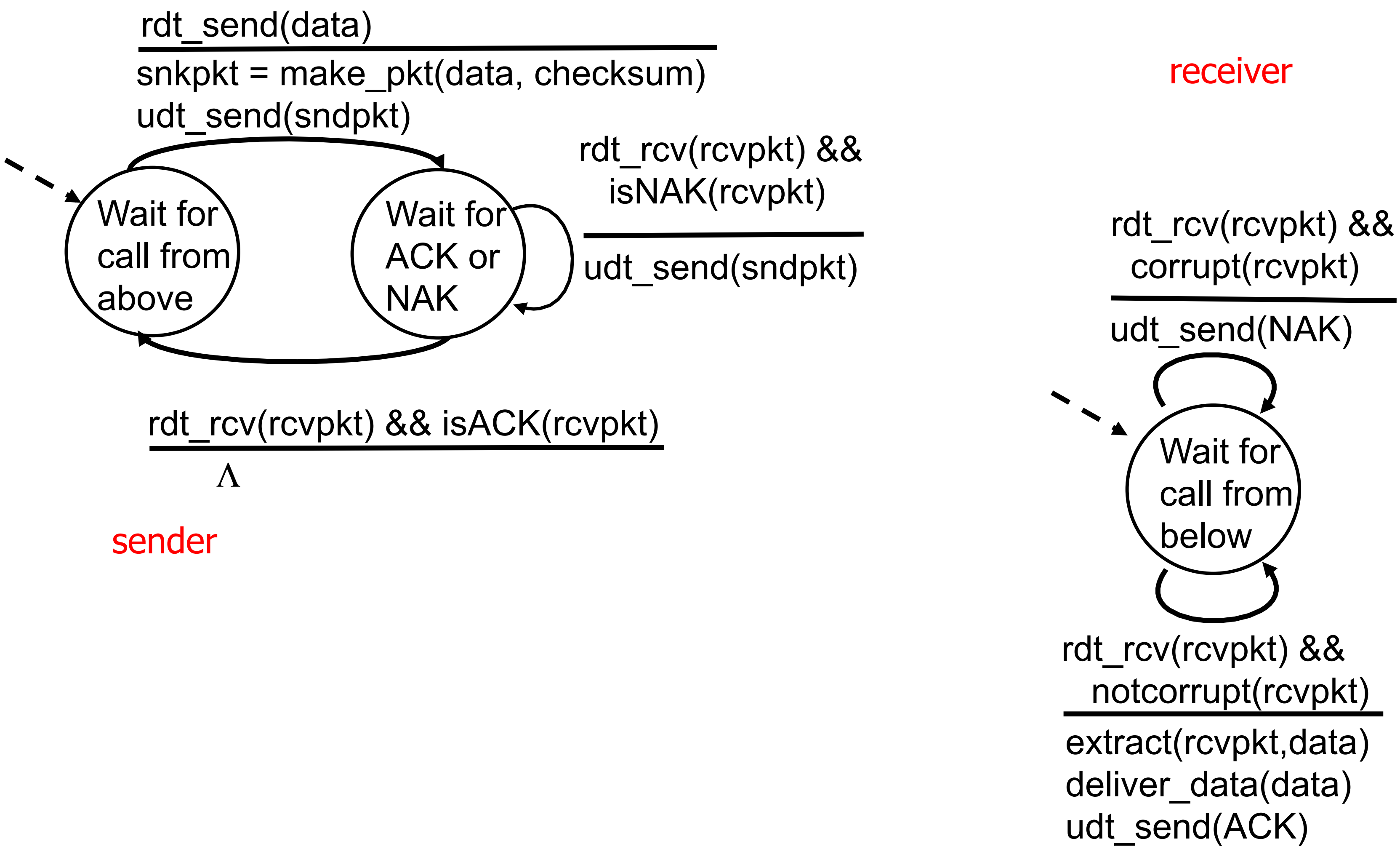
- Used in TCP, UDP, IP
- *The Internet checksum algorithm:*
  - Adjacent octets to be checksummed are paired to form 16-bit integers, and the *1's complement sum* of these 16-bit integers is formed.
  - To generate a checksum, the checksum field itself is cleared, the 16-bit 1's complement sum is computed over the octets concerned, and the 1's complement of this sum is placed in the checksum field.
  - To check a checksum, the 1's complement sum is computed over the same set of octets, including the checksum field. If the result is all 1 bits (-0 in 1's complement arithmetic), the check succeeds.

Example of Internet Checksum

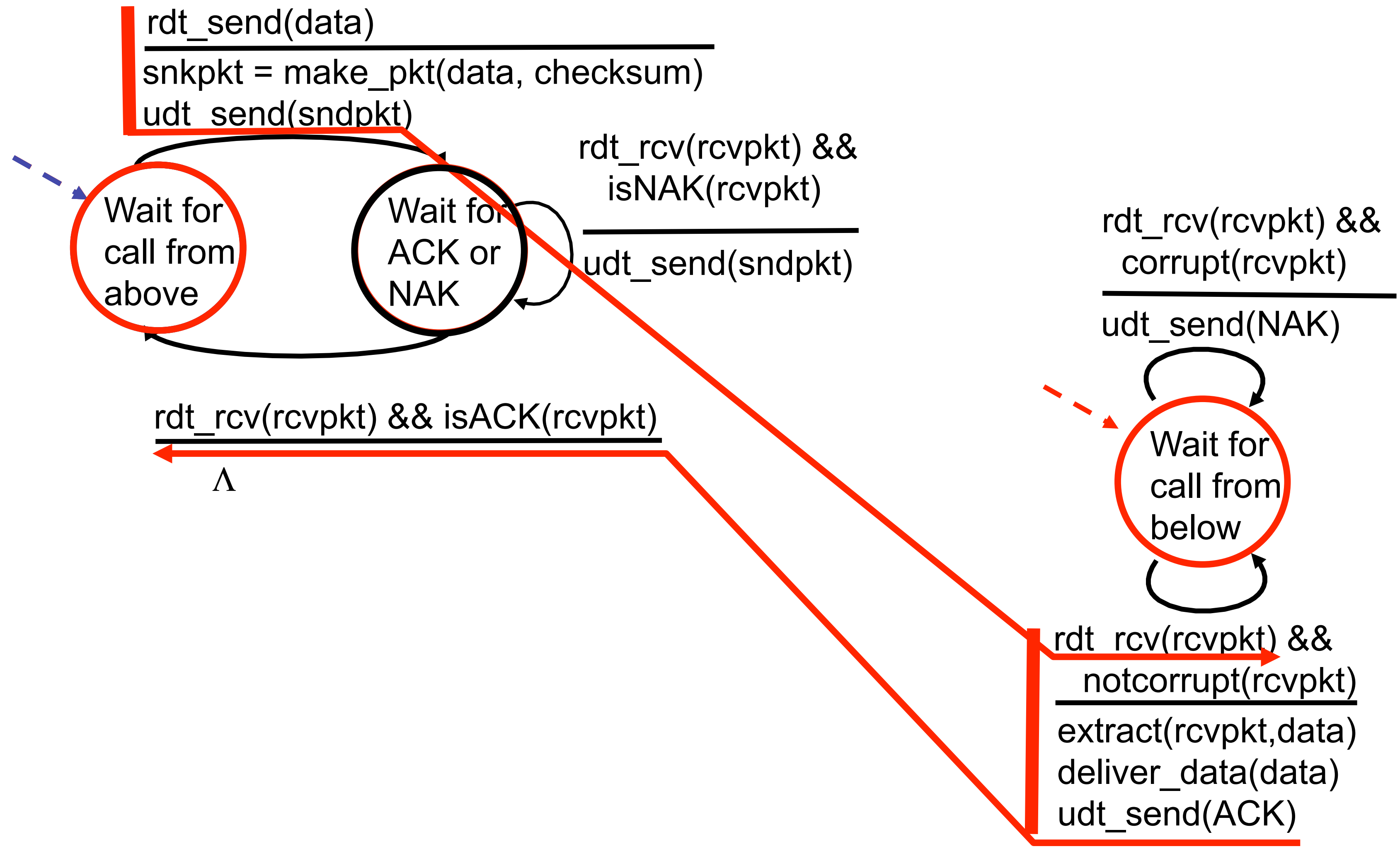
```
Computing the Checksum
              1000 0110 0101 1110   First 16-bit value
         +    1010 1100 0110 0000   Second 16-bit value
            ---------------------
            1 0011 0010 1011 1110   Carry-out “loops”
         +   \----------------> 1   back into LBb
            ---------------------
              0011 0010 1011 1111
         +    0111 0001 0010 1010   Third 16-bit value
            ---------------------
            0 1010 0011 1110 1001   No carry to swing around
         +    1000 0001 1011 0101   Fourth 16-bit value
            ---------------------
            1 0010 0101 1001 1110   Carry-out “loops”
         +   \----------------> 1   back into LBb
            ---------------------
              0010 0101 1001 1111   ”One's complement sum”
            ---------------------
              1101 1010 0110 0000   Take 1’s complement
                                    again, that’s the
                                    checksum of the data
```

```
Input data:
1000 0110 0101 1110
1010 1100 0110 0000
0111 0001 0010 1010
1000 0001 1011 0101
```

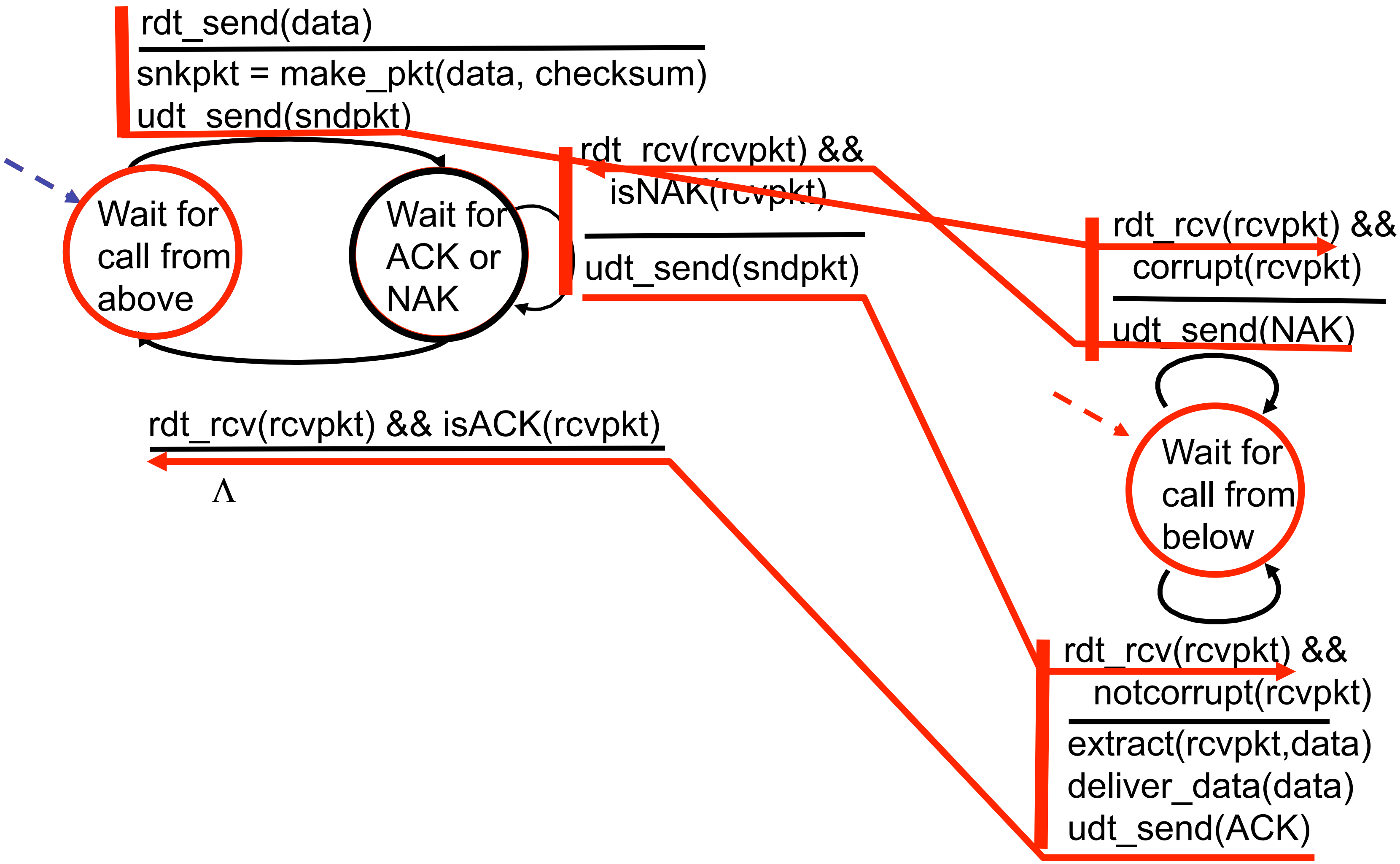
RDT2.0: the FSM Specification



RDT2.0: Operation with No Errors



RDT2.0: Error Scenario



RDT2.0: A Fatal Flaw

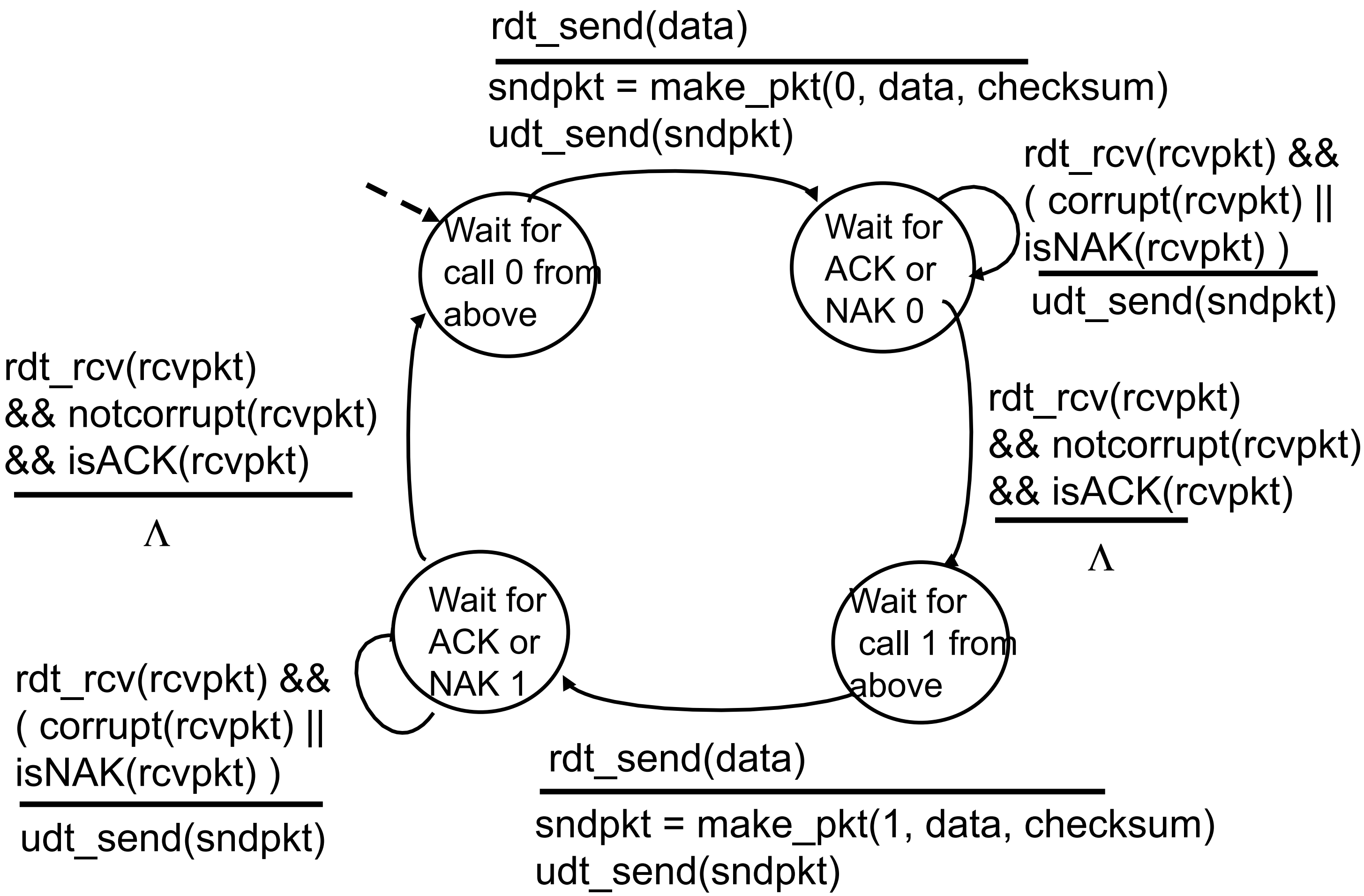
- *What if the ACK/NACK is corrupted?*
- Let's try a few options
  1. Ask receiver to resend
     - Lead to a sort of infinite loop + too much state information ("*I'm sending the ACK of the ACK of the NACK of the ACK that I sent an hour ago*")
  2. Just retransmit the packet (i.e. assume the worst)
     - Potentially create duplicate packets
- We'll pick option # 2
  - How to solve the duplication problem?
    - Use *sequence number* (i.e. packet ID)
- New mechanism in RDT2.1:
  - Sequence number

Stop-and-Wait Protocol's Sequence Numbers

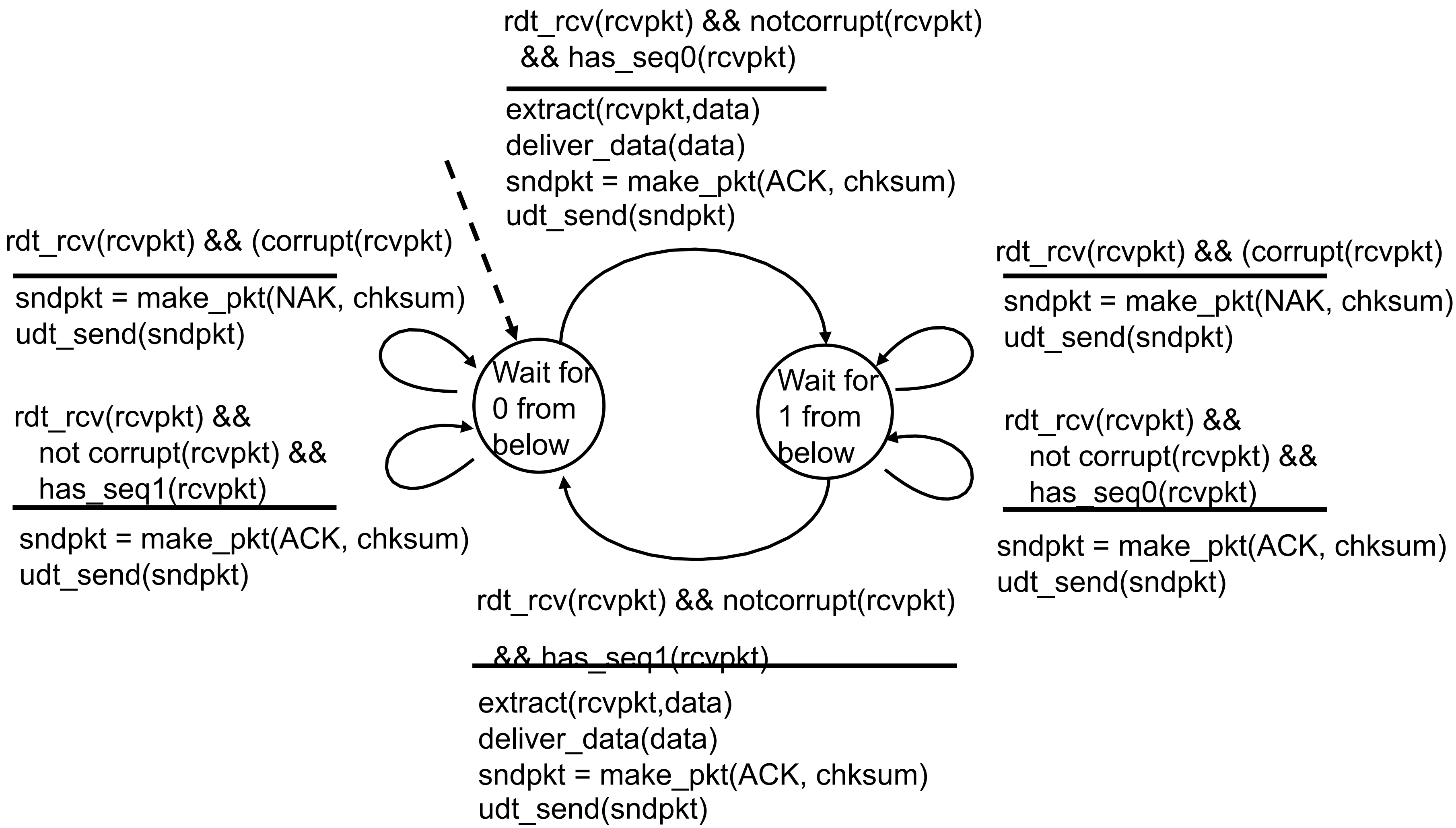
- RDT so far is a *stop and wait* protocol:
  - Sends a packet
  - Wait for ACK/NACK
  - Resend if ACK/NACK corrupted or NACK received
  - Only then, next packet can be sent

- We only need two sequence numbers: 0 and 1
  - 1-bit sequence number space

RDT2.1: Sender Side



RDT2.1: Receiver Side

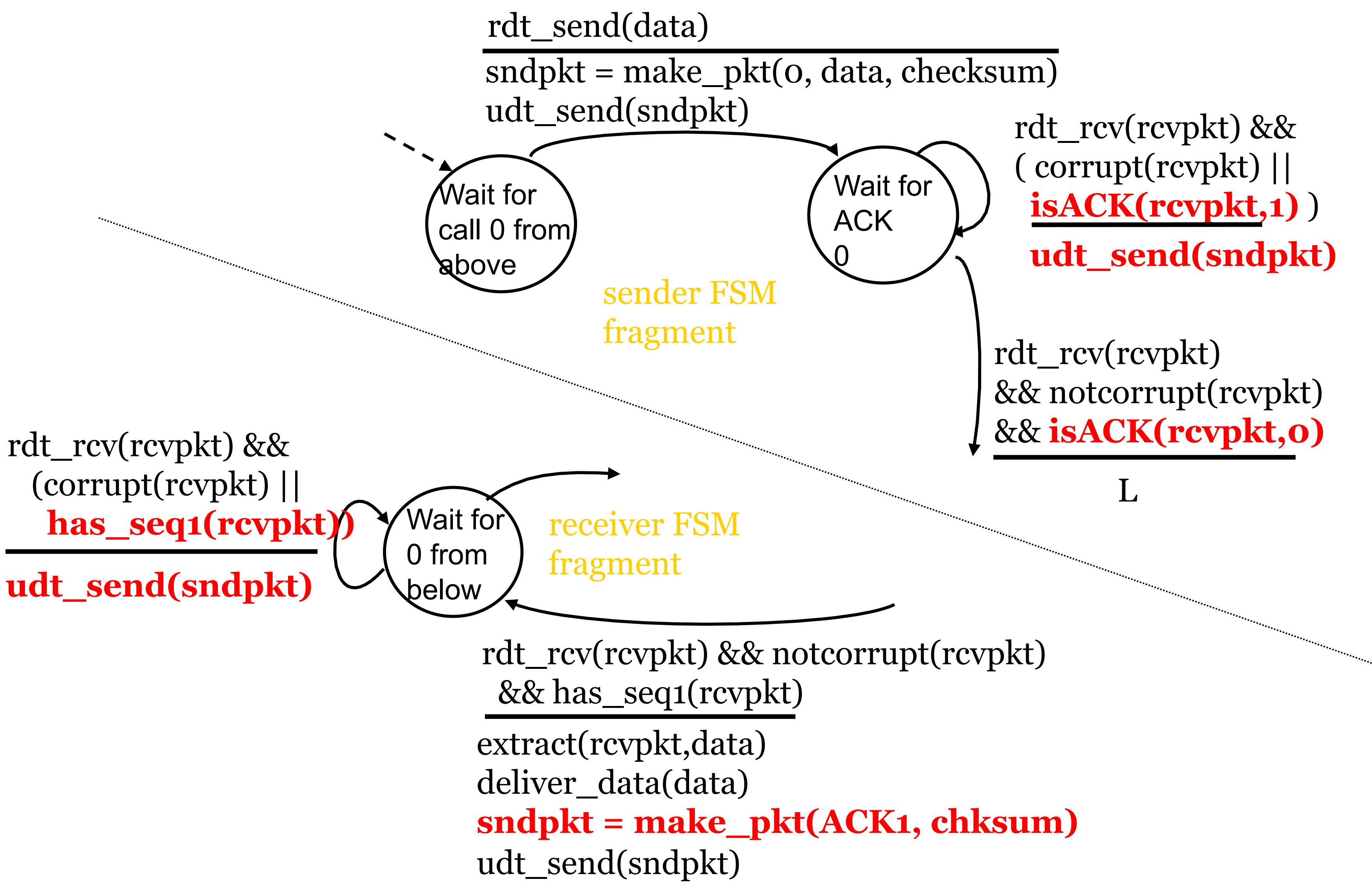


RDT2.1: Observations

- Senders and receivers need twice as many states!
- Receiver must check if a packet is a duplicate

- Can make it *NACK-free*
  - Specify in the ACK the sequence # it is acknowledging
  - If a packet is corrupted, send ACK of the previous sequence # again
  - This idea seems to appear "out of no where", will have utility later

RDT2.2: the NACK-free Version of 2.1



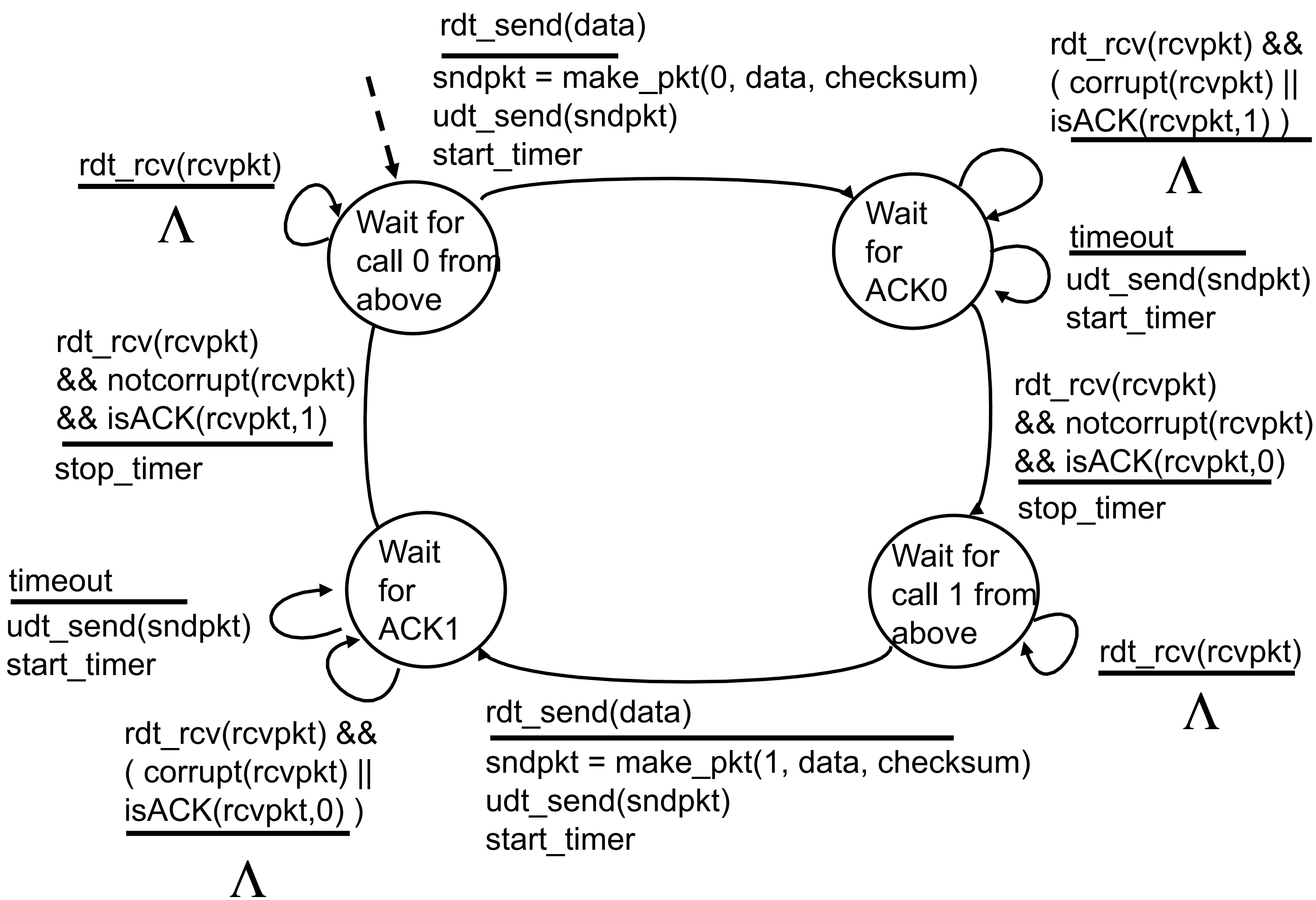
RDT3.0: Dealing with Packet Losses Too

- Both data packets and ACK can be corrupted or lost

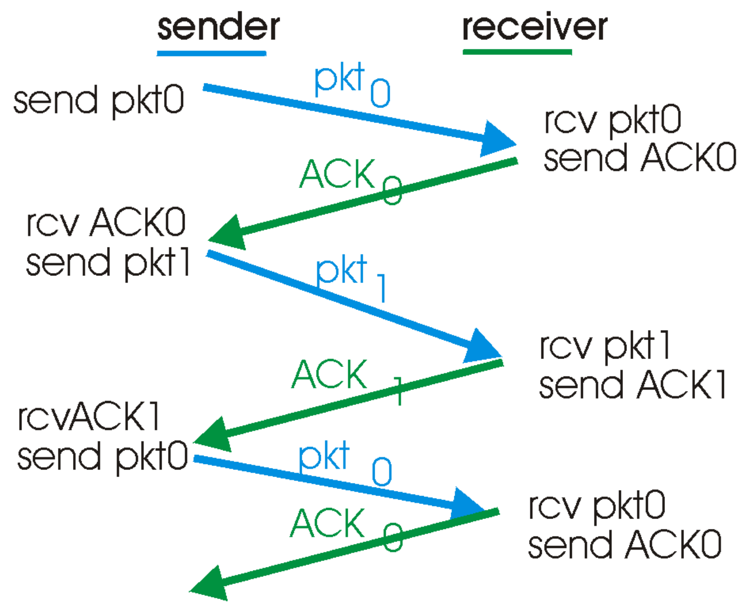
- How to detect that there's a lost packet?
  - Use a timer
  - *But how long?*
    - Too long → inefficiency
    - Too short → duplicate packet (seq. no. took care of this!)
    - We'll get back to this issue later

- What to do when a packet is lost? Well, retransmit!

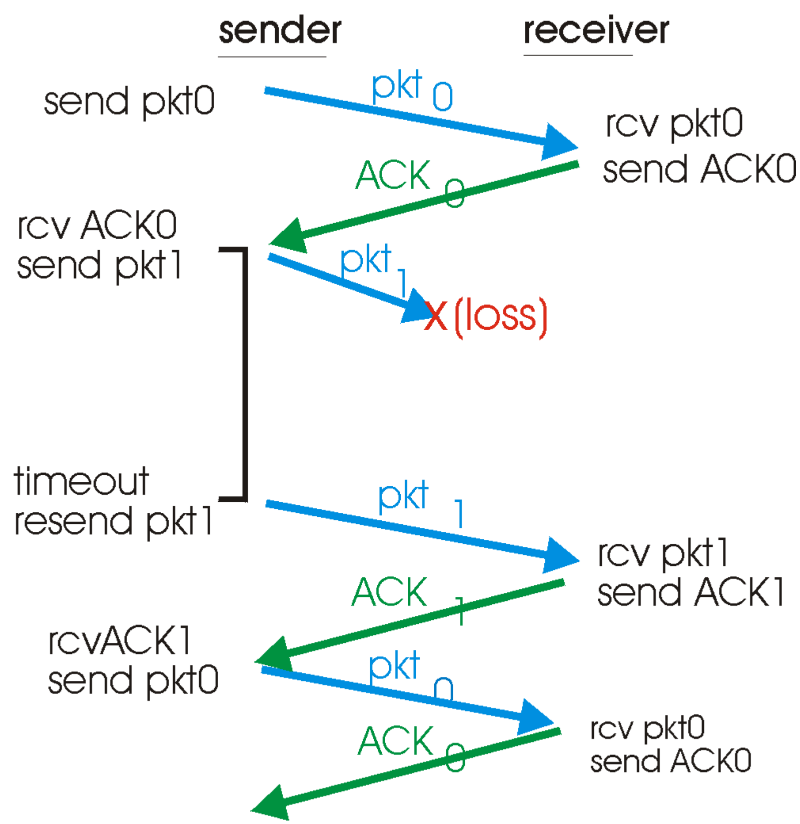
RDT3.0: Sender Side



RDT3.0 In Action

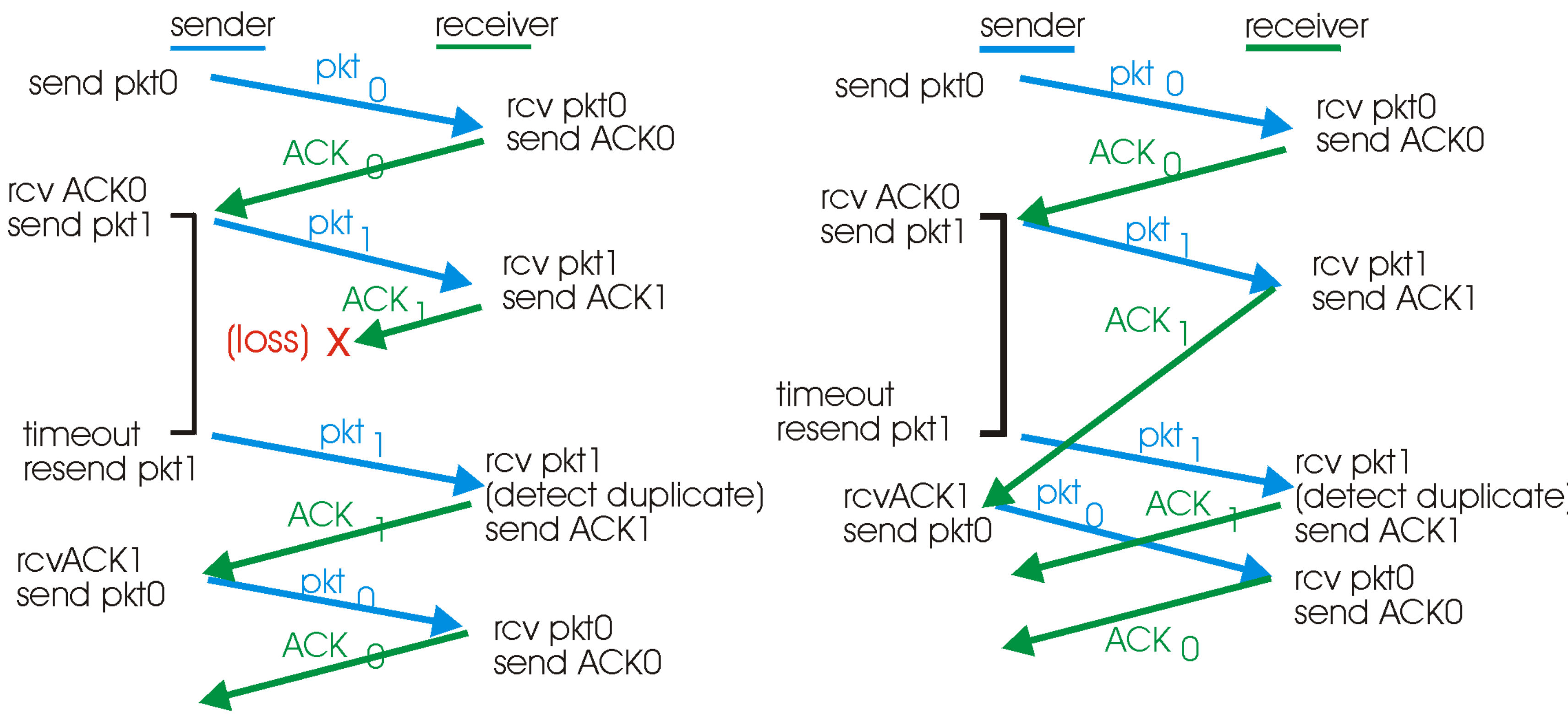


(a) operation with no loss



(b) lost packet

RDT3.0 In Action



(c) lost ACK (d) premature timeout

Performance of RDT3.0

- RDT3.0 works, but performance stinks!
- Example: 1 Gbps link, 15 ms propagation delay, 8000 bit packet:

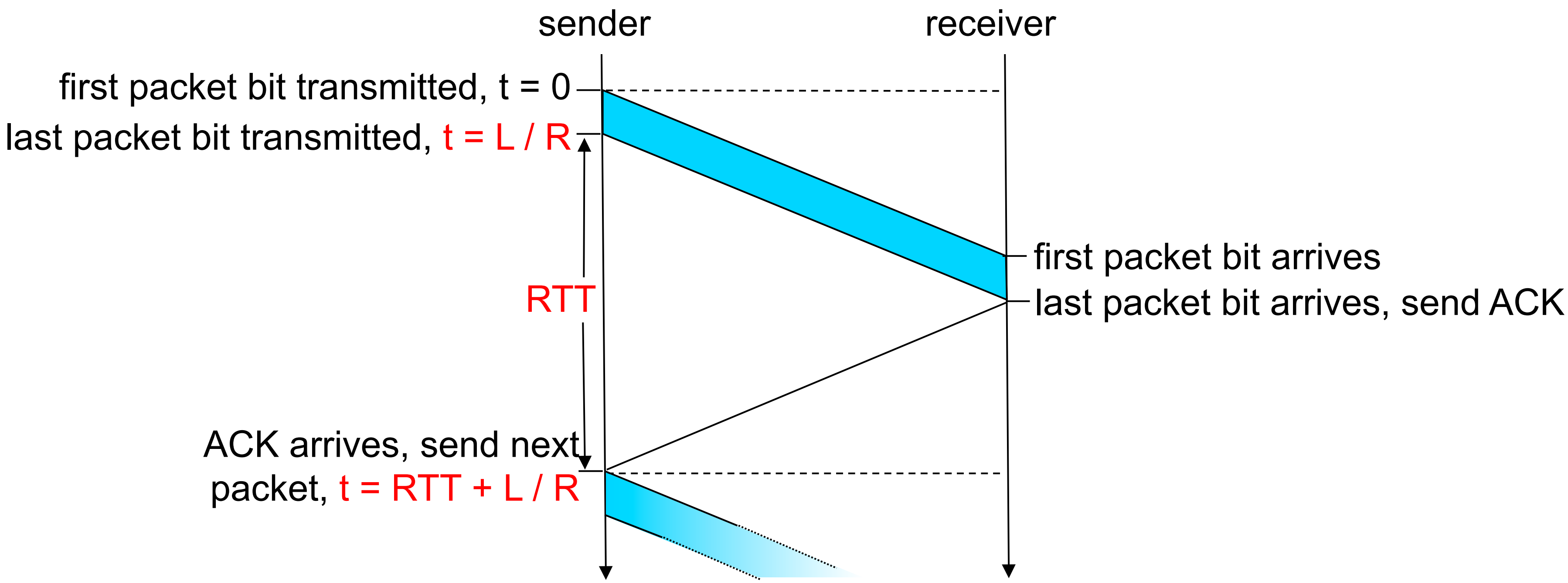
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\,\text{microseconds}$$

  - *Channel utilization* – fraction of time sender busy sending

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

  - 1KB packet every 30 msec, i.e. 33kB/sec *throughput* over 1 Gbps link
  - *Lesson: network protocol limits use of physical resources!*

RDT3.0: Stop-and-Wait → Bad Utilization

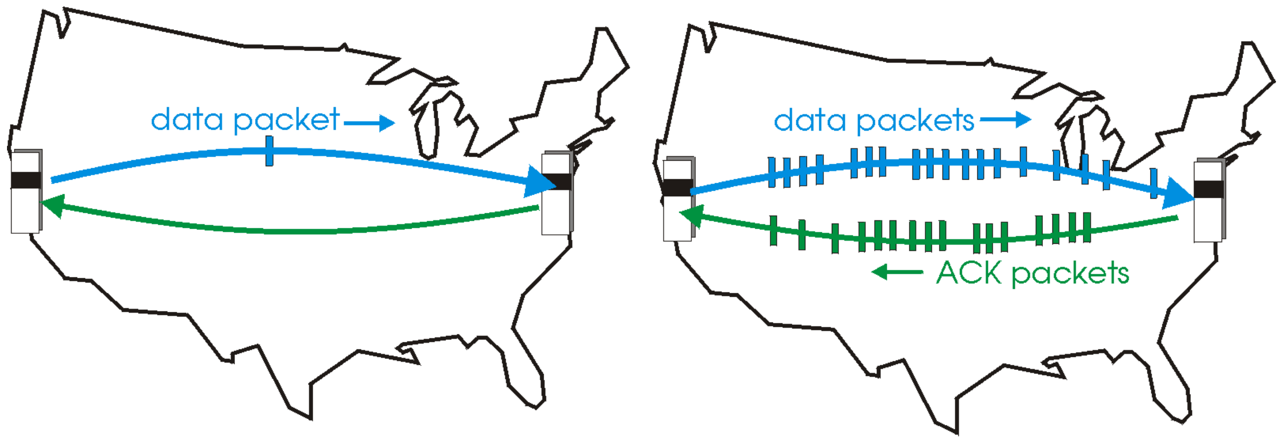


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Question: *how to utilize the channel better?*

Pipelined Protocols

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

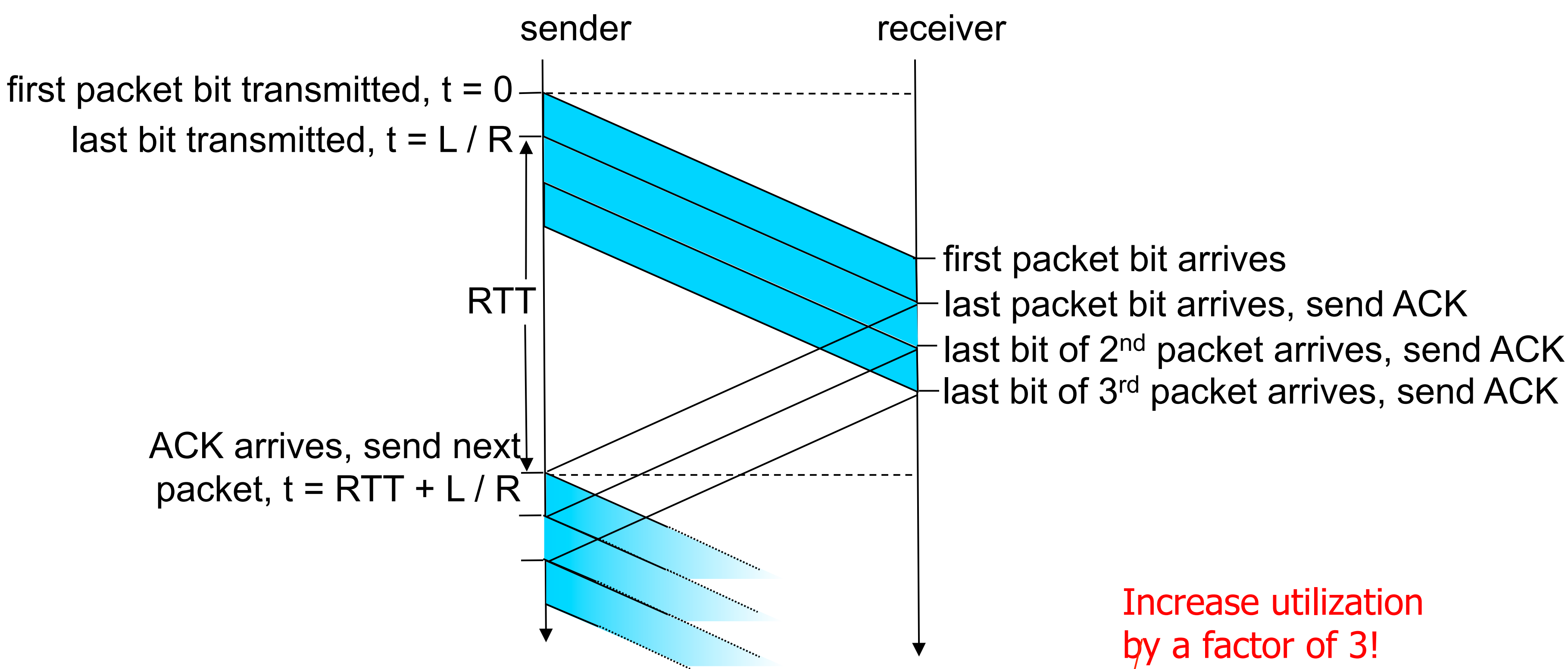


(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

Pipelining introduces new problems:

- 1-bit seq. # no longer works → enlarge seq. # space
- We have to deal with out-of-order packets now
- Larger buffers at senders and receivers

Pipelining Helps Increase 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* unack'ed packets in pipeline
  - i.e. keep a *window* of size N
- Receiver only sends cumulative ACKs
  - *Drop any out-of-order packet*
  - Re-ACK oldest received pkt
- Sender has a timer for *oldest* unacked packet
  - When timer expires, retransmit all unack'ed packets in the window

Selective Repeat: big picture

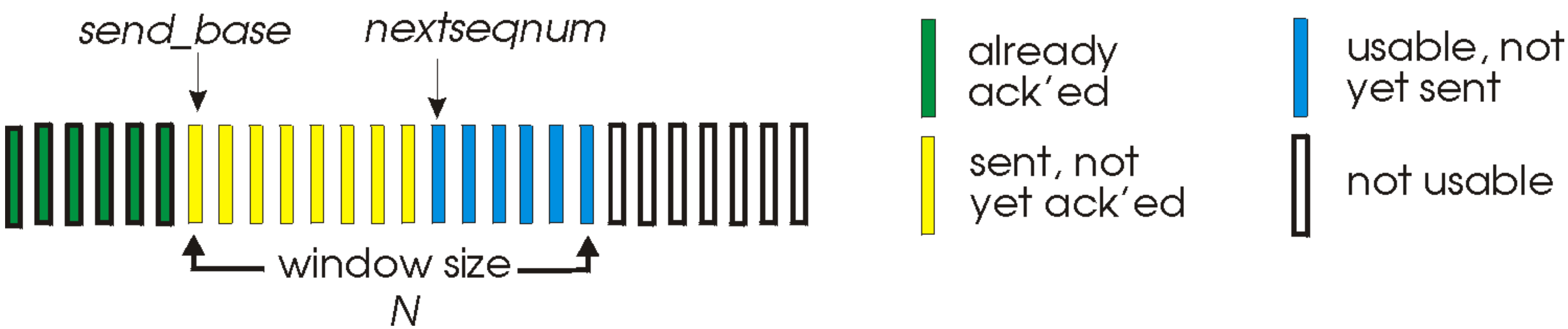
- Sender can have up to *N* unack'ed packets in pipeline
  - i.e. keep a *window* of size N
- Receiver acknowledges individual packets
  - Including out-of-order packets within window
- Sender maintains a timer for *each* unacked packet
  - When any timer expires, retransmit that unacked packet

Question: why limit the number of unack'ed packets to N?

Go-Back-N

Sender:

- Uses a *k*-bit sequence number in packet header
- Maintains a *window* of up to *N*, consecutive unack'ed packets allowed
- Also called *sliding window protocol*



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

Go-Back-N: Sender Extended FSM

rdt_send(data)

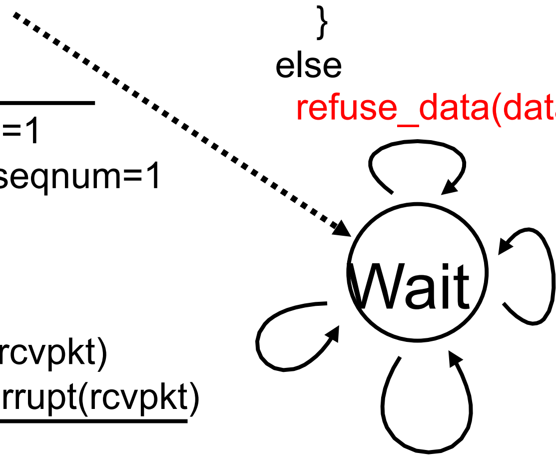
---

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)

Λ

---

base=1
nextseqnum=1



timeout

---

start_timer // and resend all unack'ed packets
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

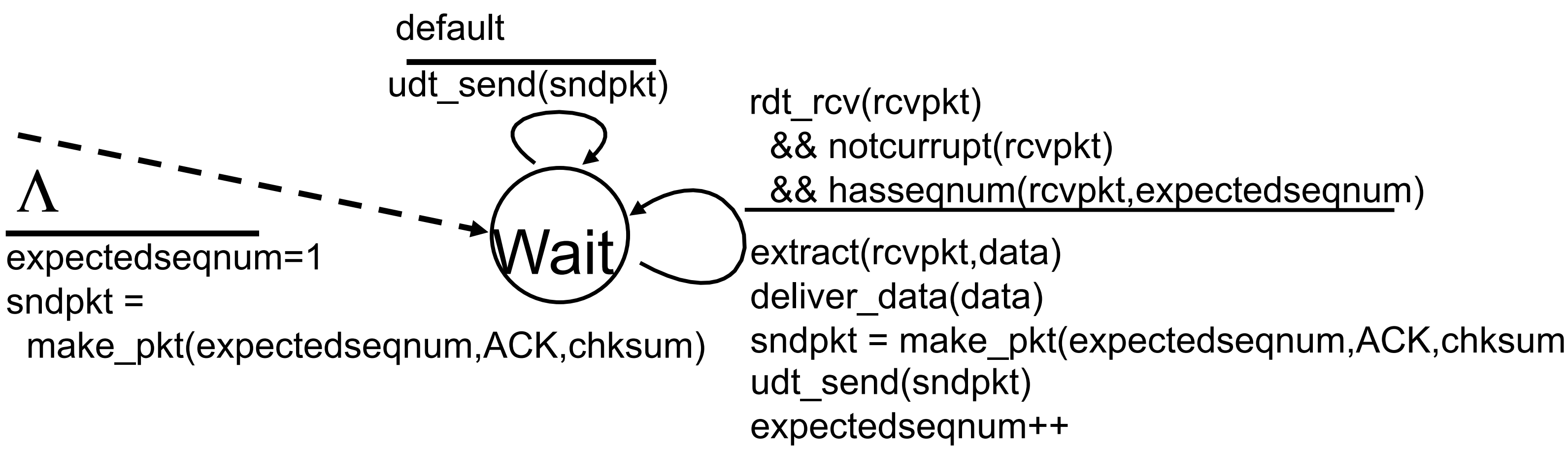
---

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

---

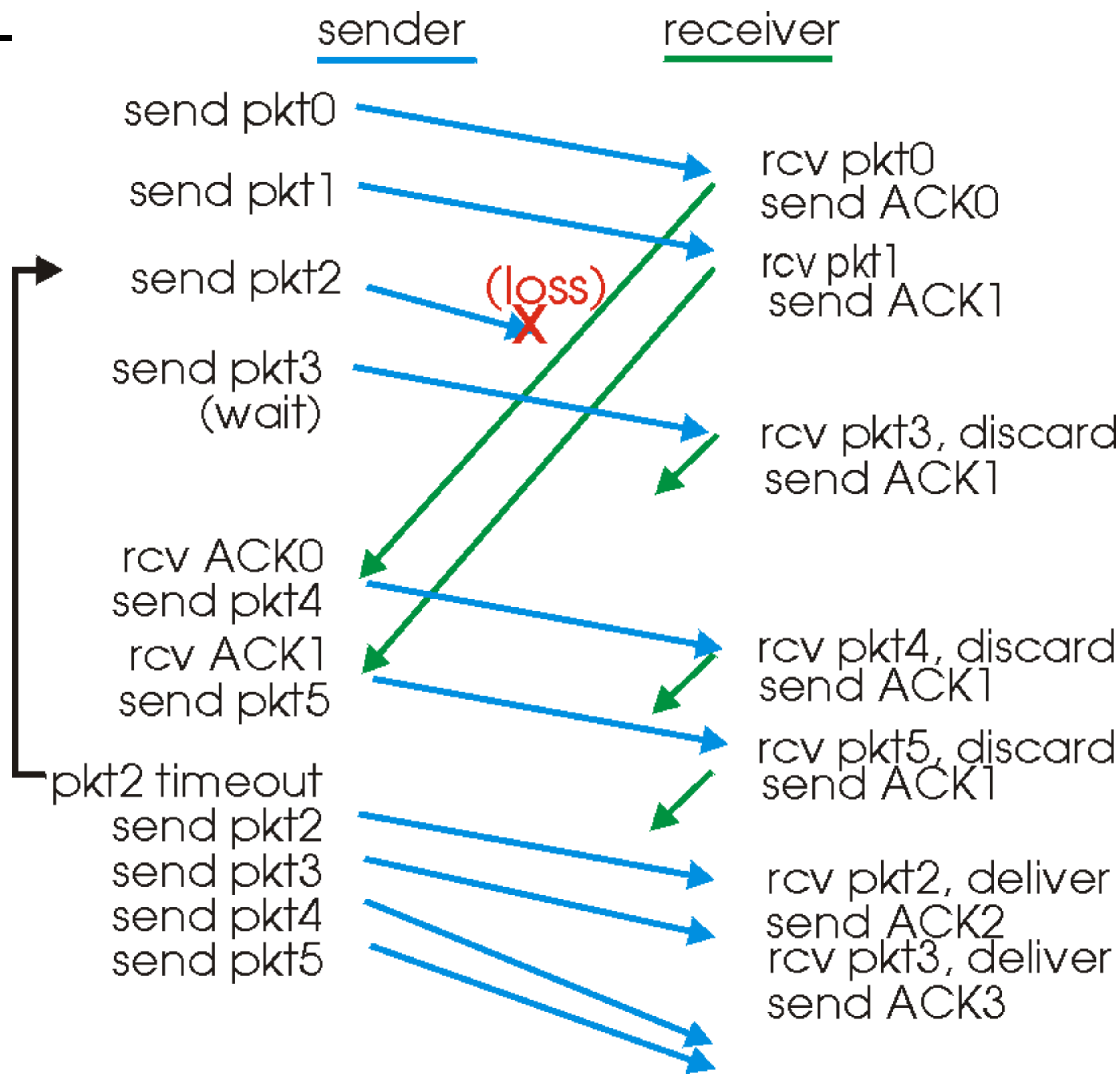
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer // restart timer for new base

Go-Back-N: Receiver Extended FSM

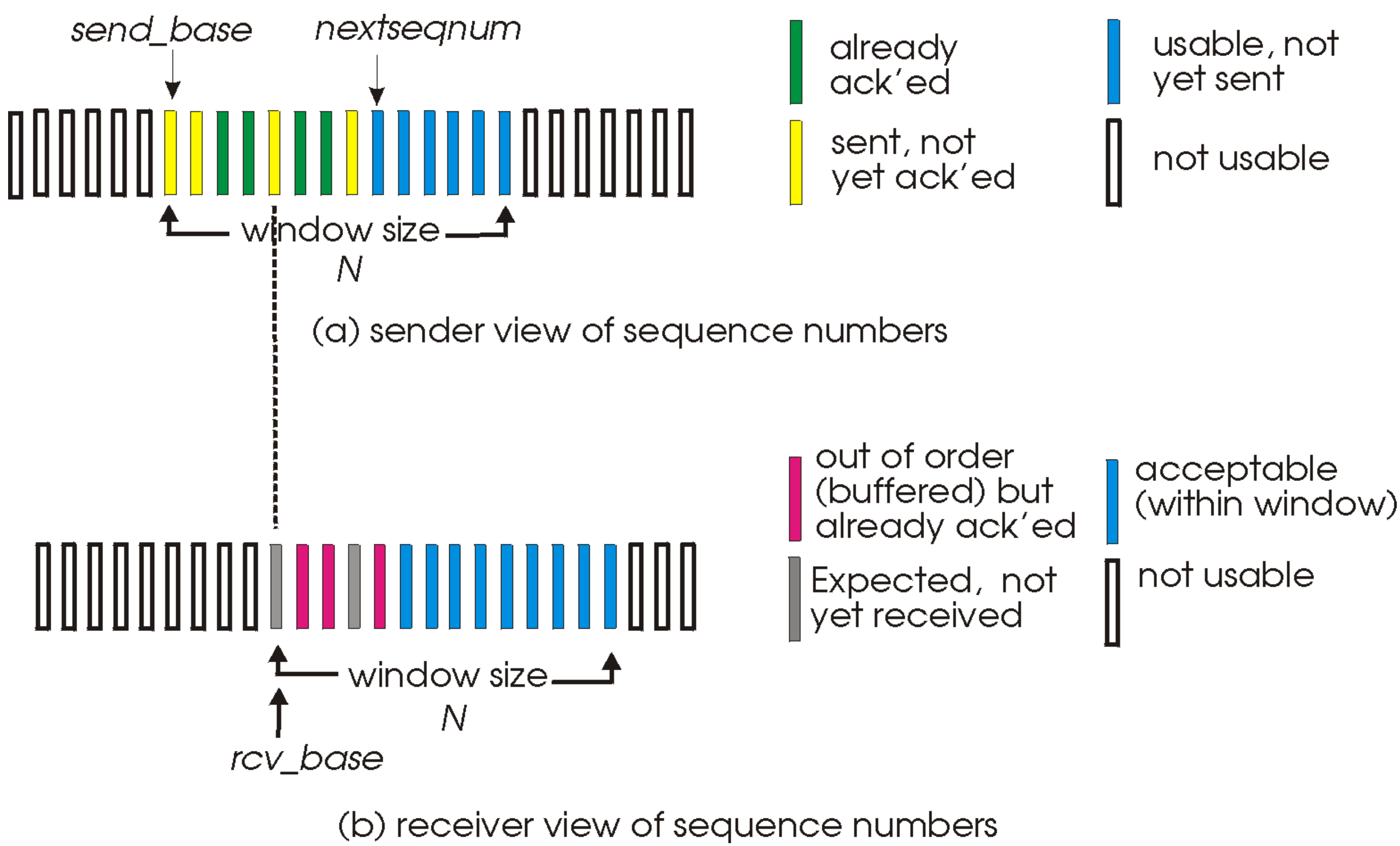


- ACK-only: always send ACK for correctly-received packet with highest *in-order* sequence number
  - may generate duplicate ACKs
  - need only remember **`expectedseqnum`**
- Out-of-order pkt:
  - discard (don't buffer) -> *no receiver buffering*!
  - Re-ACK packet with highest in-order sequence number

Go-Back-N in Action



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

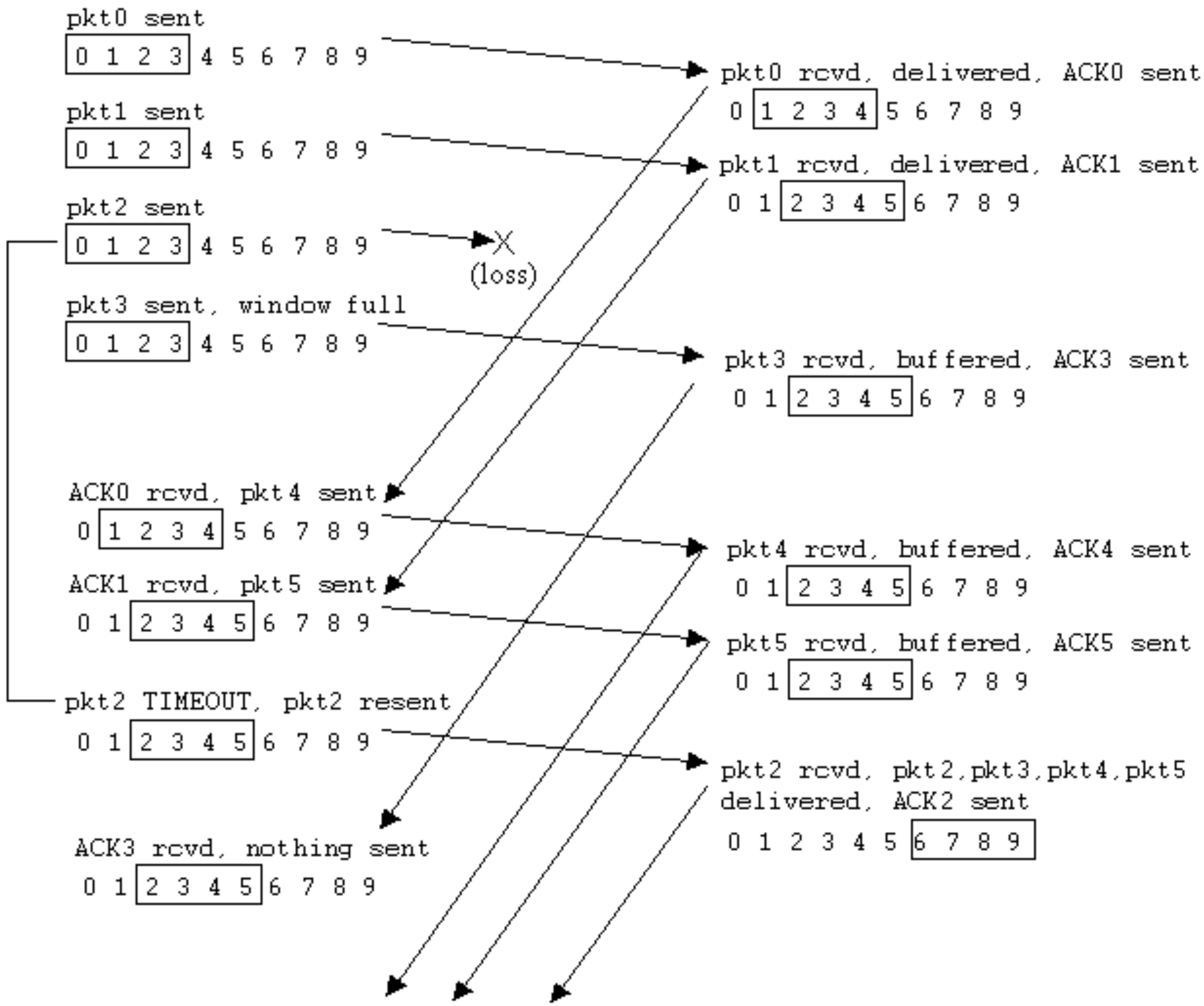
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

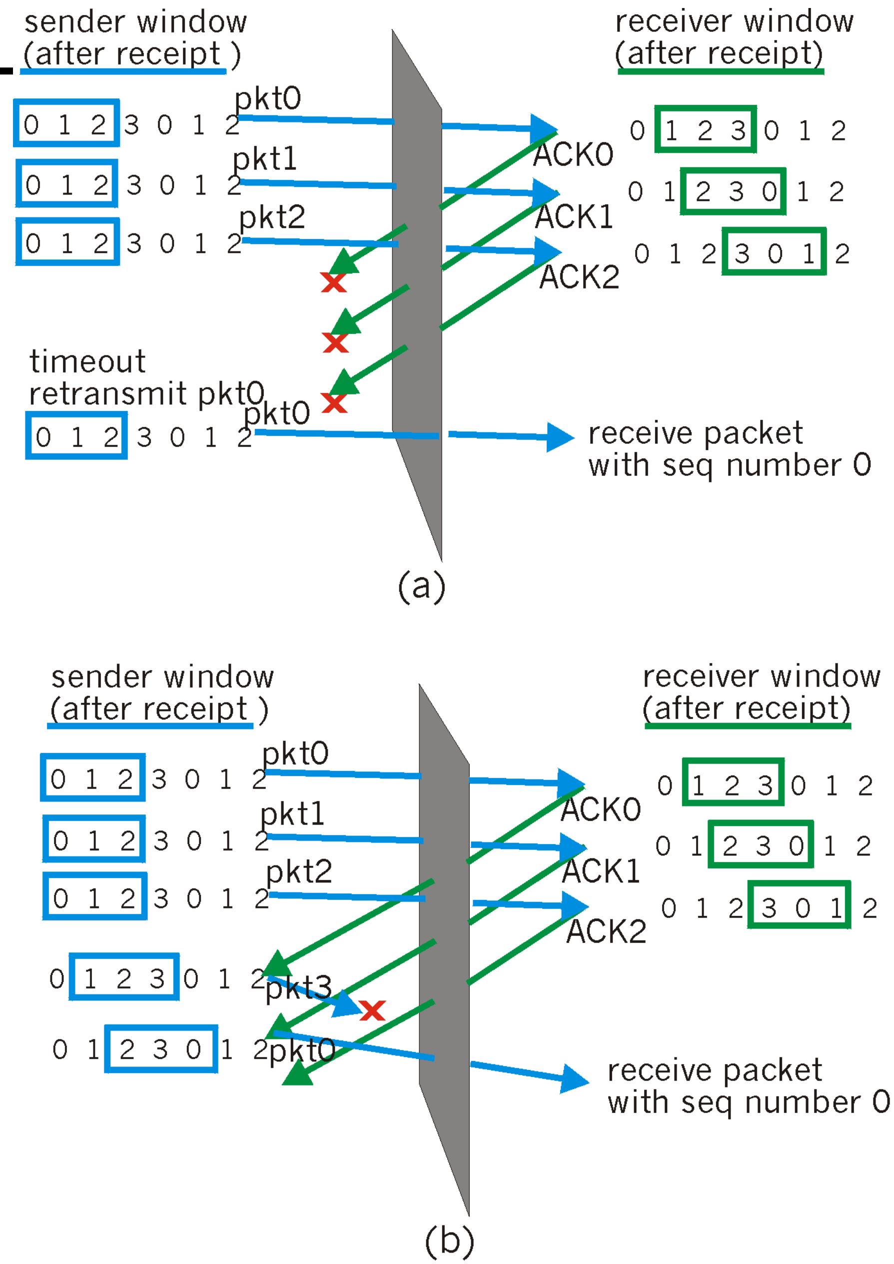
Selective Repeat In Action



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)

Question: what relationship between seq # size and window size must hold?

Summary of Ideas We Have Learned

- Channel *bit errors* require
  - Error detecting codes
  - Receiver feedback (ACK/NAK)
  - Retransmissions (when NAK received or ACK/NAK corrupted)
- Retransmissions introduce *duplicates*
  - Need sequence numbers
- *Packet loss* requires
  - Timeout + retransmission (again introduce duplicates)
  - Estimating the “right” timeout is a fundamental problem!
- *Pipelining* improves utilization + throughput
  - Needs to enlarge sequence number space
  - Needs more buffer space at both sender & receiver
  - ACK + retransmission strategies: Go-Back-N & Selective Repeat
  - Window size & sequence number range strongly related
- *We have not discussed out-of-order,* ***long****-delayed packets*
- *And how long the timeout should be before retransmission*