CS 455/555

Intro to Networks and Communications

The Transport Layer

Multiplexing, UDP, & Reliable Transport

Dr. Michele Weigle

Department of Computer Science

Old Dominion University

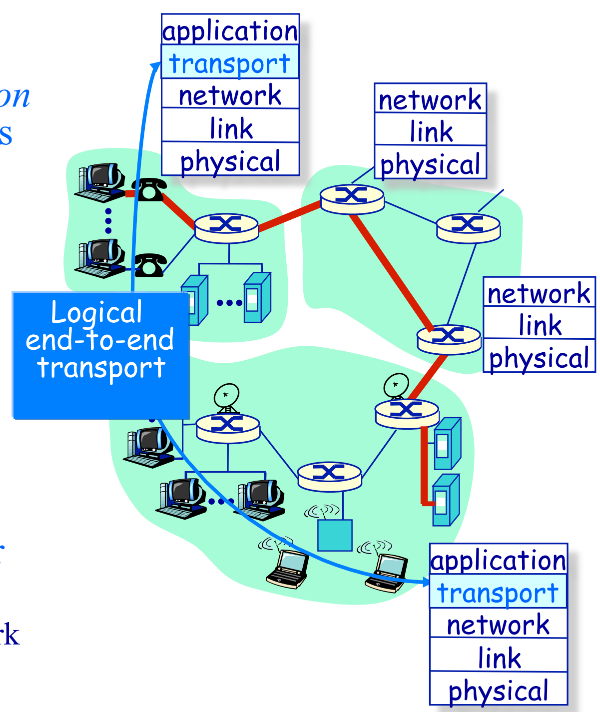
mweigle@cs.odu.edu

http://www.cs.odu.edu/~mweigle/CS455-S13

The Transport Layer

Transport services and protocols

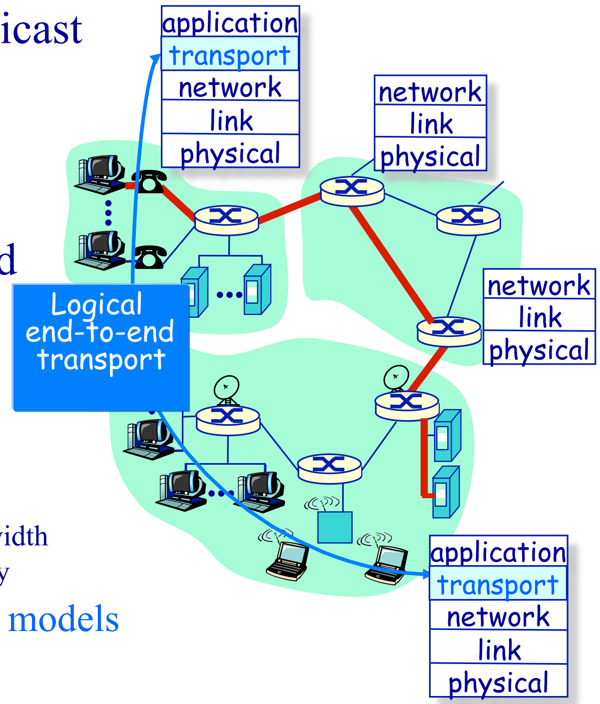
- Transport protocols:
  - » Provide *logical communication* between application processes running on different hosts
  - » Execute on the end systems (and *not* in the network)
- Transport *v.* network layer services:
  - » *Network layer:* data transfer between end systems
  - » *Transport layer:* data transfer between processes
    - Relies on, and enhances, network layer services



Transport Layer Protocols

Internet transport services

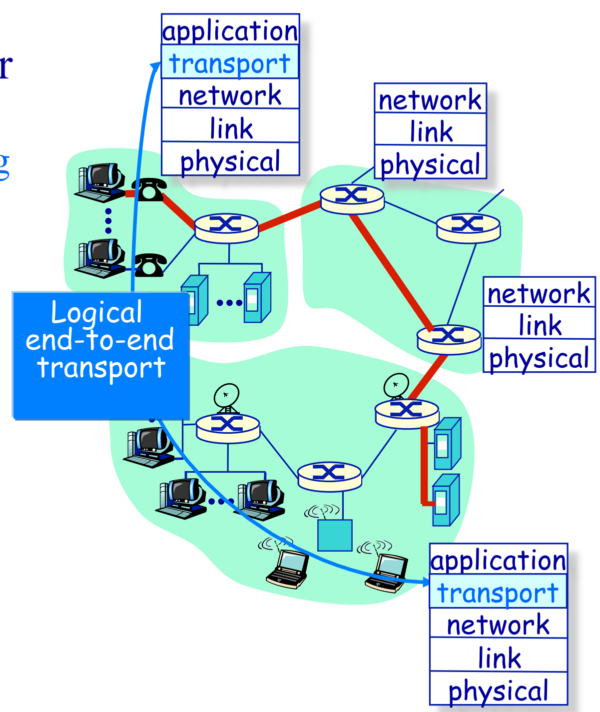
- TCP: Reliable, in-order, unicast delivery
  - » Congestion control
  - » Flow control
  - » Connection setup
- UDP: Unreliable, unordered ("best-effort"), unicast or multicast delivery
  - » (Minimal) error detection
- Services not available:
  - » Performance guarantees
    - ❖ No guarantees of available bandwidth
    - ❖ No guarantees of end-to-end delay
  - » Other (non-unicast) delivery models
    - ❖ Multicast (reliable *v.* unreliable)
    - ❖ Anycast



Transport Layer Protocols & Services

Outline

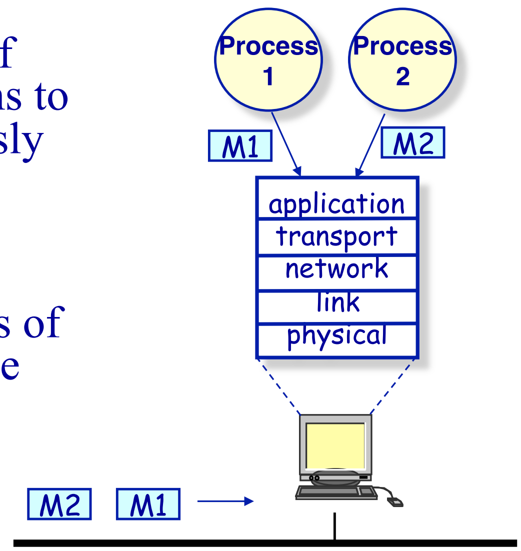
- Fundamental transport layer services
  - » Multiplexing/Demultiplexing
  - » Error detection
  - » Reliable data delivery
  - » Pipelining
  - » Flow control
  - » Congestion control
- Service implementation in Internet transport protocols
  - » UDP
  - » TCP



Fundamental Transport Layer Services

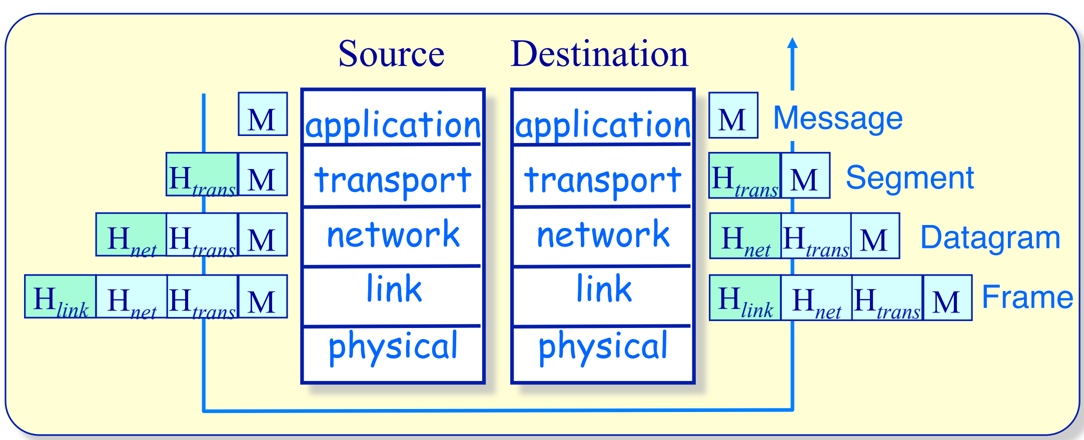
Multiplexing/Demultiplexing

- Each end-system has a single protocol "stack"
  - » The stack is shared between all applications using the network
- Multiplexing is the process of allowing multiple applications to use the network simultaneously
  - » (To send data into the network concurrently)
- Demultiplexing is the process of delivering received data to the appropriate application



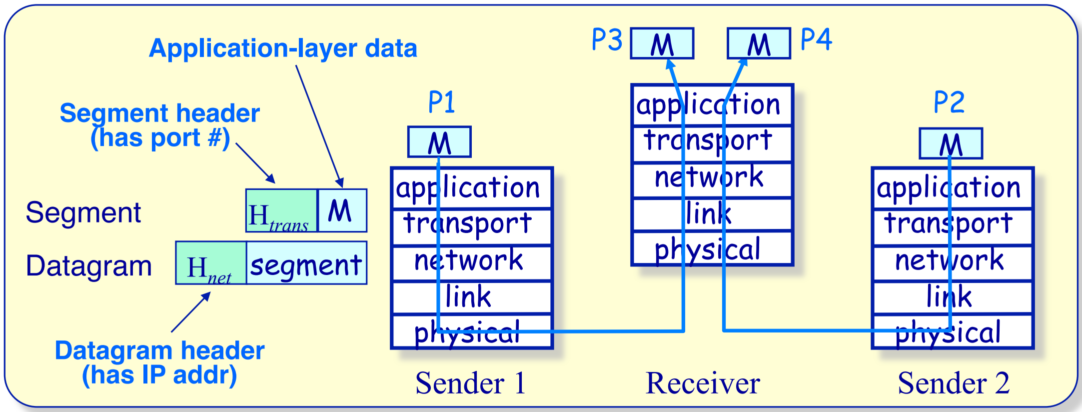
Multiplexing/Demultiplexing

Review: Protocol layering in the Internet



- At the sender, each layer takes data from above
  - » May subdivide into multiple data units at sending layer
  - » Adds header information to create new data unit
  - » Passes new data unit to layer below
- The process is reversed at the receiver

Multiplexing/Demultiplexing
Demultiplexing



- Demultiplexing is the process of delivering received segments to the correct application-layer process
  - » IP address (in network-layer datagram header) identifies the receiving machine
  - » Port number (in transport-layer segment header) identifies the receiving process

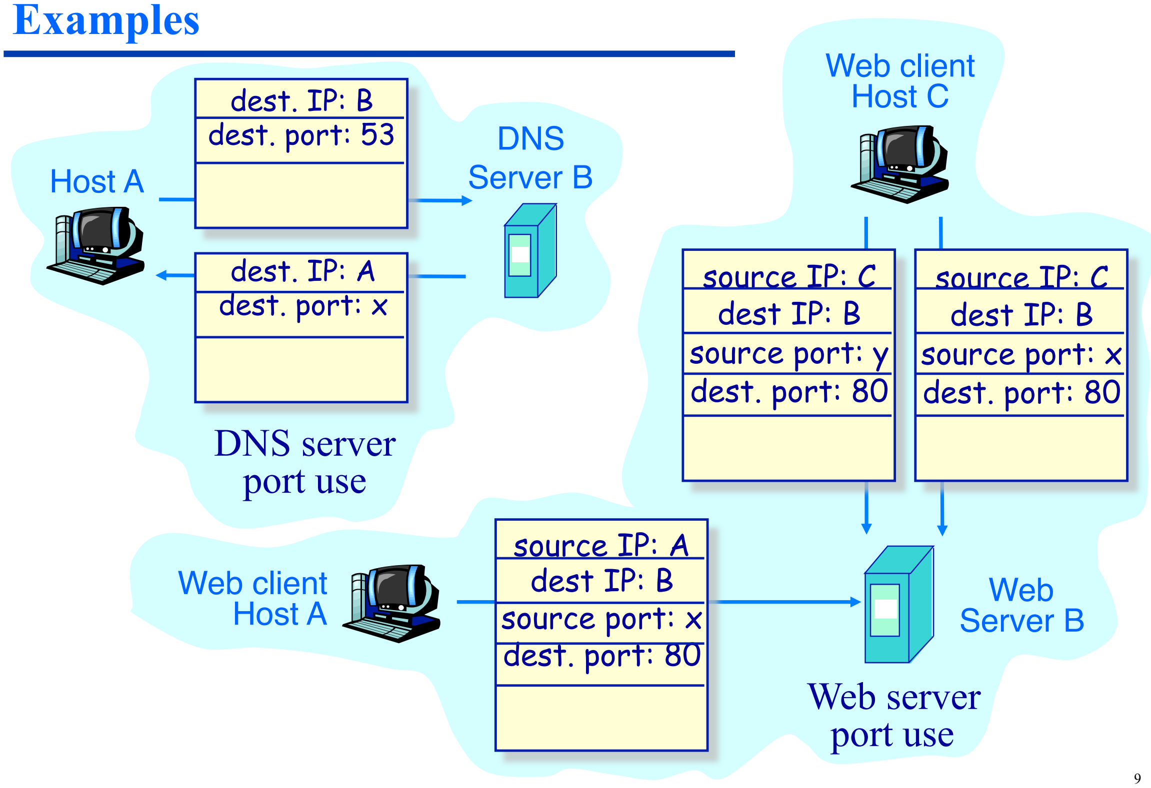
Multiplexing/Demultiplexing
Transport protocol specific demultiplexing

- Demultiplexing actions depend on whether the transport layer is connectionless (UDP) or connection-oriented (TCP)

- UDP demultiplexes segments to the *socket*
  - » UDP uses 2-tuple
    <destination IP address, destination port number>
    to identify the socket
  - » Socket is "owned" by some process (allocated by OS).

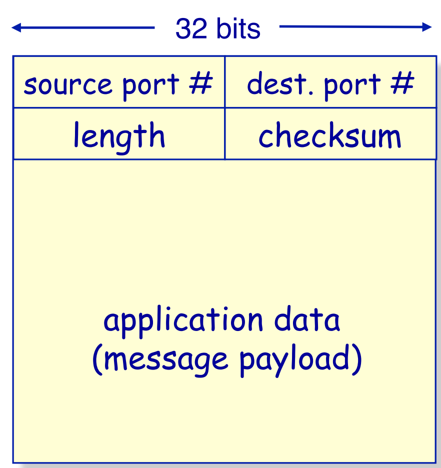
- TCP demultiplexes segments to the *connection*
  - » TCP uses 4-tuple
    <source IP addr, source port nbr, destination IP addr, destination port nbr>
    to the identify connection
  - » Connection (and its socket) is owned by some process

Multiplexing/Demultiplexing
Examples



Internet Transport Protocols
User Datagram Protocol (UDP) [RFC 768]

- No frills, "bare bones" Internet transport protocol
- Best effort service — UDP segments may be:
  - » Lost
  - » Delivered out of order to the application
  - » Delivered multiple times to the application
- "Connectionless"
  - » No handshaking between UDP sender, receiver
  - » Each UDP segment handled independently of others
- Error Detection
  - » Based on checksum
  - » Make sure received packets haven't been corrupted



UDP segment format

Length field is length in bytes, of UDP segment (including header)

User Datagram Protocol (UDP)

Is unreliable, unordered communications useful?

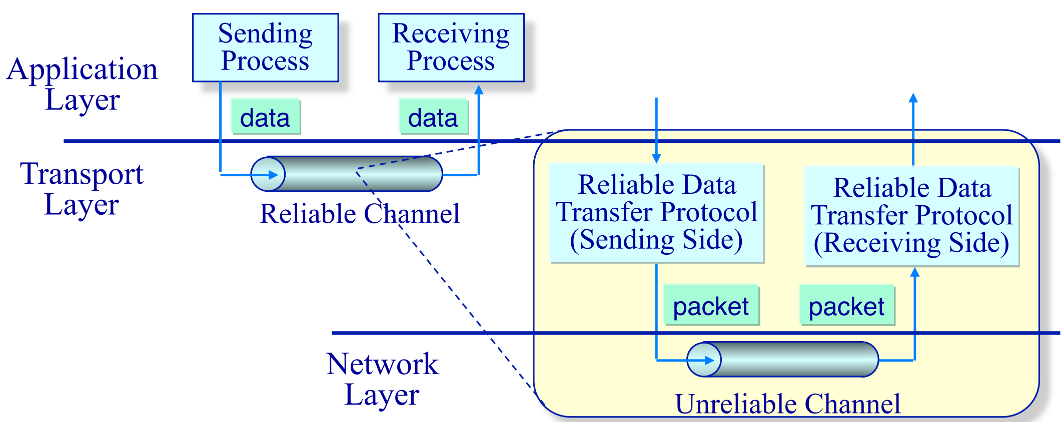
- Who uses UDP?
  - » Often used for streaming multimedia applications
  - » Loss tolerant
  - » Rate sensitive
- Other UDP uses (why?):
  - » DNS
  - » SNMP
  - » Routing protocols

Why use 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

- Reliable transfer over UDP still possible
  - » Reliability can always be added at the application layer
  - » (Application-specific error recovery)

Fundamental Transport Layer Services

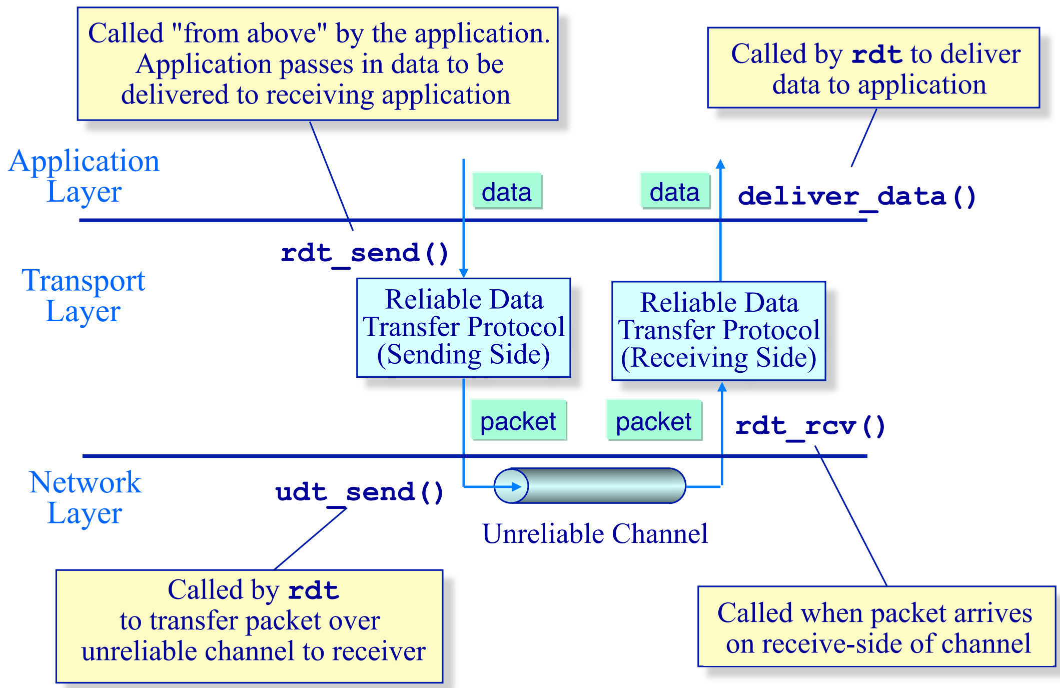
Principles of reliable data transfer



- Goal: Provide a reliable channel abstraction
  - » The characteristics of the underlying channel will determine the complexity of providing reliable communications
- Issues: State required at sender and receiver and number of control messages exchanged

Reliable Data Transfer

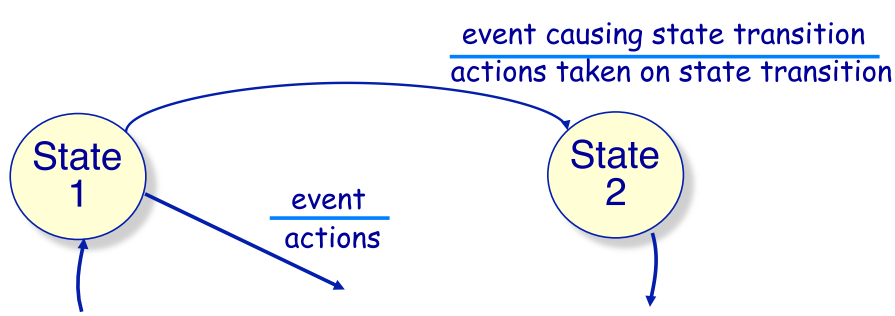
Programming interfaces



Reliable Data Transfer

Protocol specification method

- Use finite state machines to specify sender and receiver algorithms
  - » When in a given state, the next state (and actions) are uniquely determined by the next event

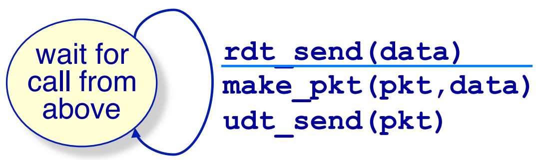


Reliable Data Transfer Protocol 1.0

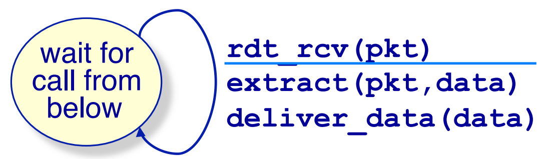
Reliable transfer over a reliable channel

- The underlying channel is assumed to be perfectly reliable
  - » No bit errors
  - » No loss of packets

- Sender state machine

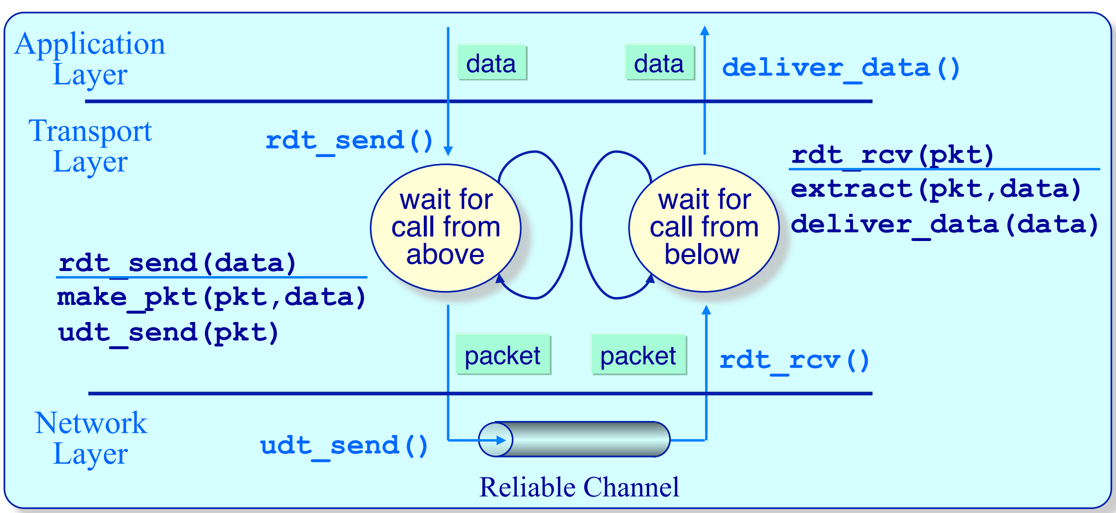


- Receiver state machine



Reliable Data Transfer Protocol 1.0

Programming interfaces



- This is the complete protocol under the assumption of a reliable network channel

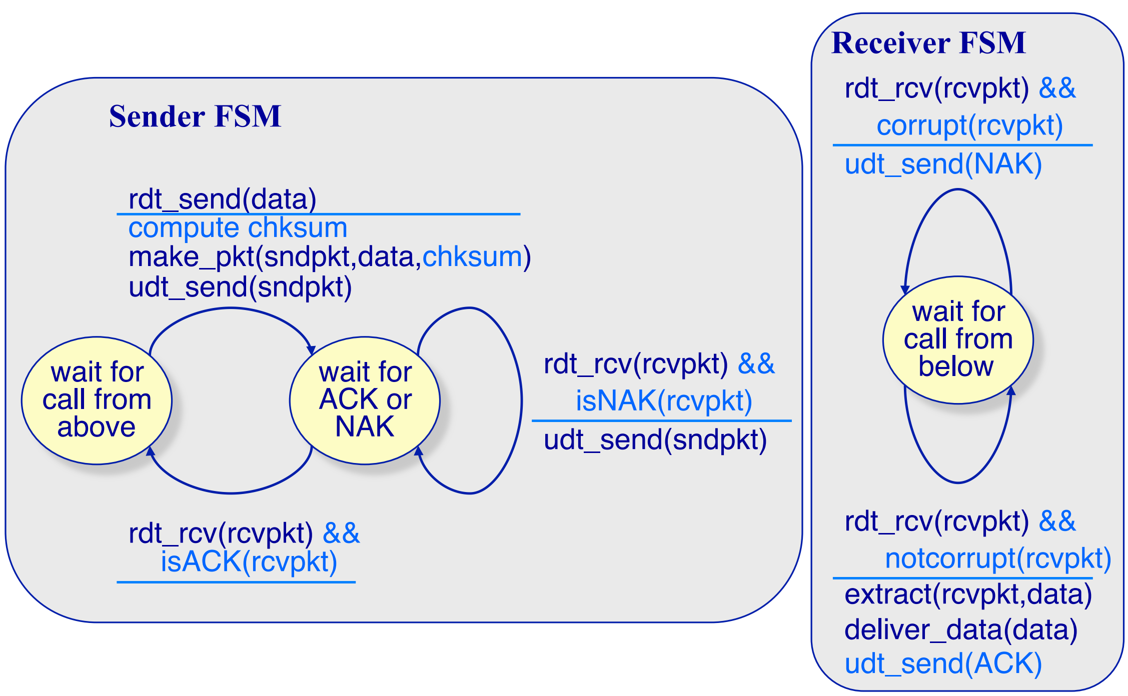
Reliable Data Transfer Protocol 2.0

Reliable transfer over a channel with bit errors

- Now assume the underlying channel may "flip" random bits in a packet
- How to detect errors?
- How to recover from errors:
  - » *acknowledgements* (*ACKs*) — the receiver explicitly tells the sender that a packet was received OK
  - » *negative acknowledgements* (*NAKs*) — the receiver explicitly tells the sender that a packet had errors
  - » Sender retransmits packet on receipt of NAK
- New mechanisms to deal with bit errors:
  - » Error detection
  - » Control messages (ACK, NAK) from a receiver to the sender
  - » Retransmission

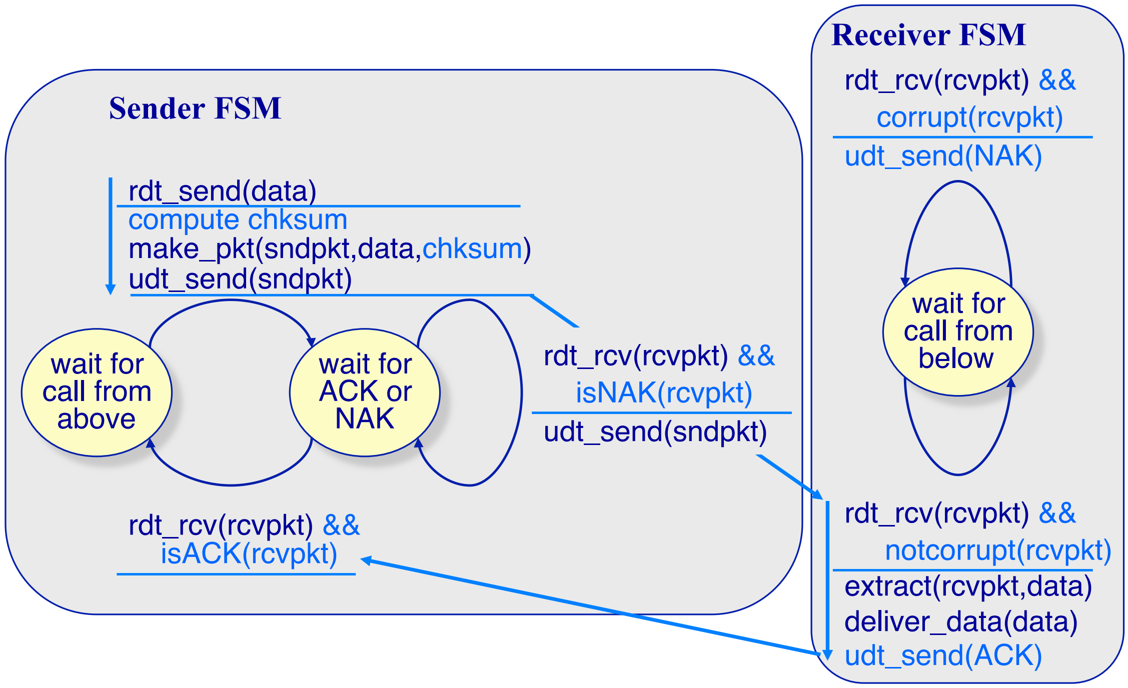
Reliable Data Transfer Protocol 2.0

Reliable transfer over a channel with bit errors only



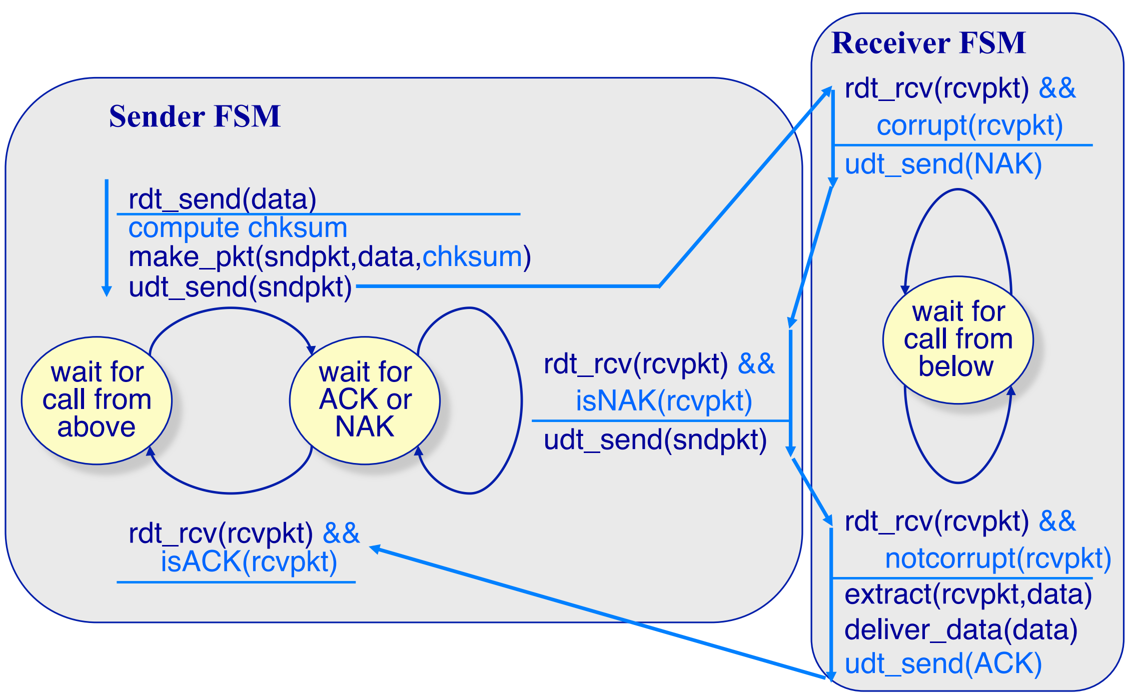
Reliable Data Transfer Protocol 2.0

Example 1: No Errors Occur



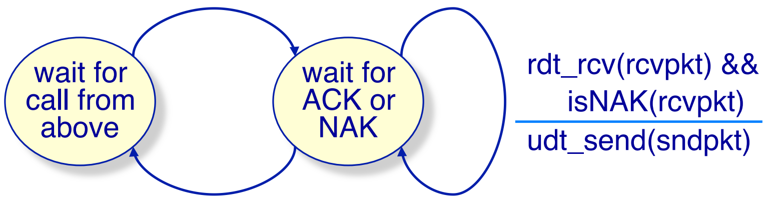
Reliable Data Transfer Protocol 2.0

Example 2: A corrupted packet arrives at the receiver



Reliable Data Transfer Protocol 2.0

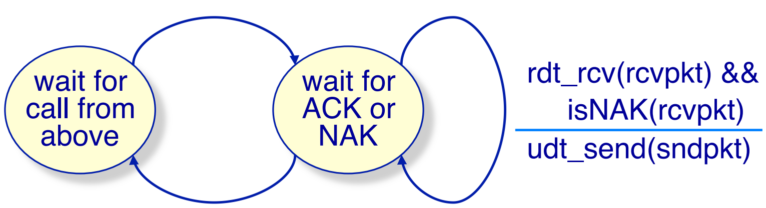
Simple… but wrong!

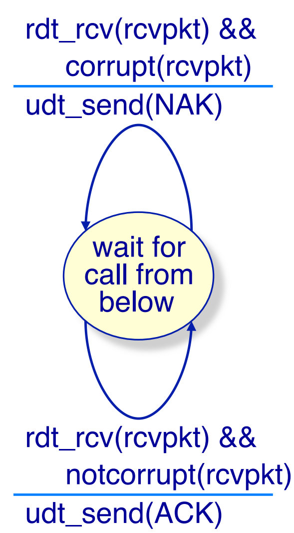


- What happens if an ACK/NAK is corrupted?
  - » Sender doesn't know what happened at the receiver!
- What to do?
  - » Sender ACKs/NAKs the receiver's ACK/NAK?
  - » Retransmit last data packet?

Reliable Data Transfer Protocol 2.0

Simple… but wrong!

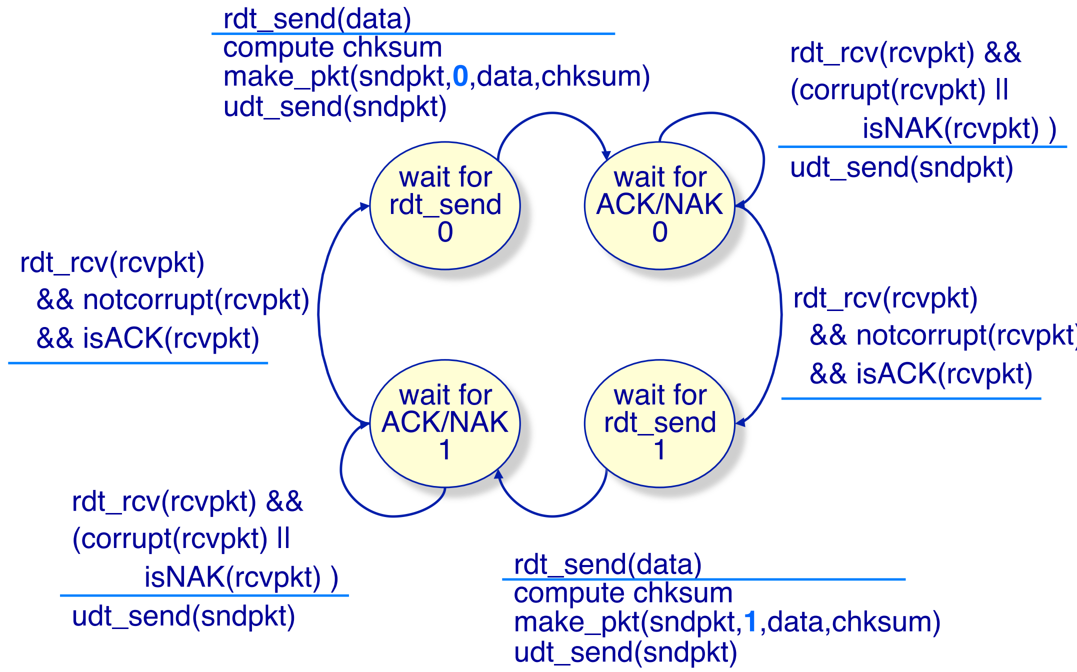




- Deal with corrupted ACKs/NAKs by retransmission of data packets
- Sender will add a sequence number to each packet to allow the receiver to detect duplicate packets
  - » Receiver's transport layer discards duplicate packets
- How much space to reserve in a header field for sequence numbers?

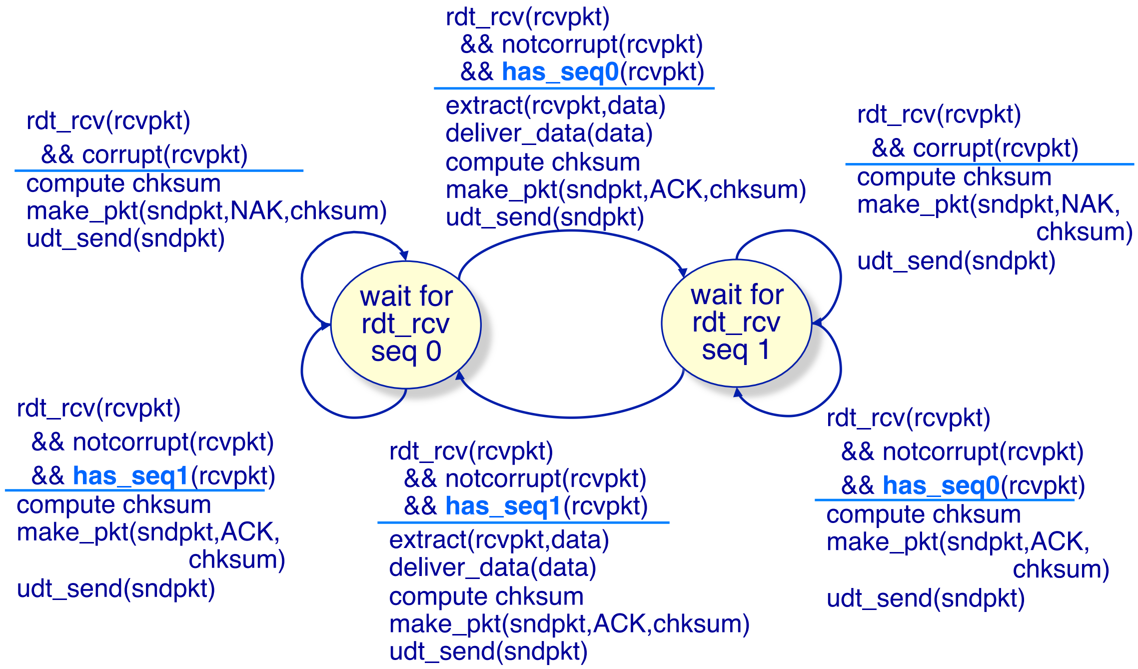
Reliable Data Transfer Protocol 2.1

Sender state machine to handle garbled ACKs/NAKs



Reliable Data Transfer Protocol 2.1

Receiver state machine to handle garbled ACKs/NAKs



Reliable Data Transfer Protocol 2.1

Discussion (Handling garbled ACKs/NAKs)

- Sender issues

Sequence number added to header

» Two sequence numbers suffice

Must check if received ACK/NAK is corrupted

Number of states doubles

» State encodes whether current packet has sequence number 0 or 1

- Receiver issues

Must check if received packet is duplicate

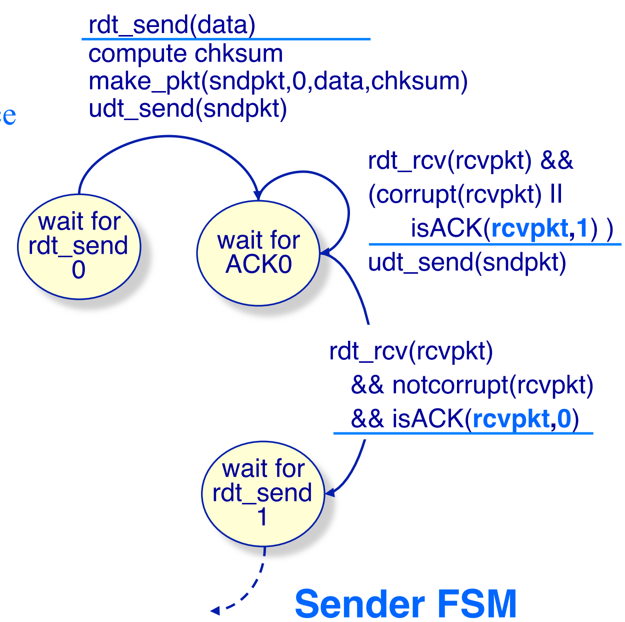
» State encodes whether expected packet sequence number is 0 or 1

Note: receiver can *not* know if its last ACK/NAK received OK at sender

Reliable Data Transfer Protocol 2.2

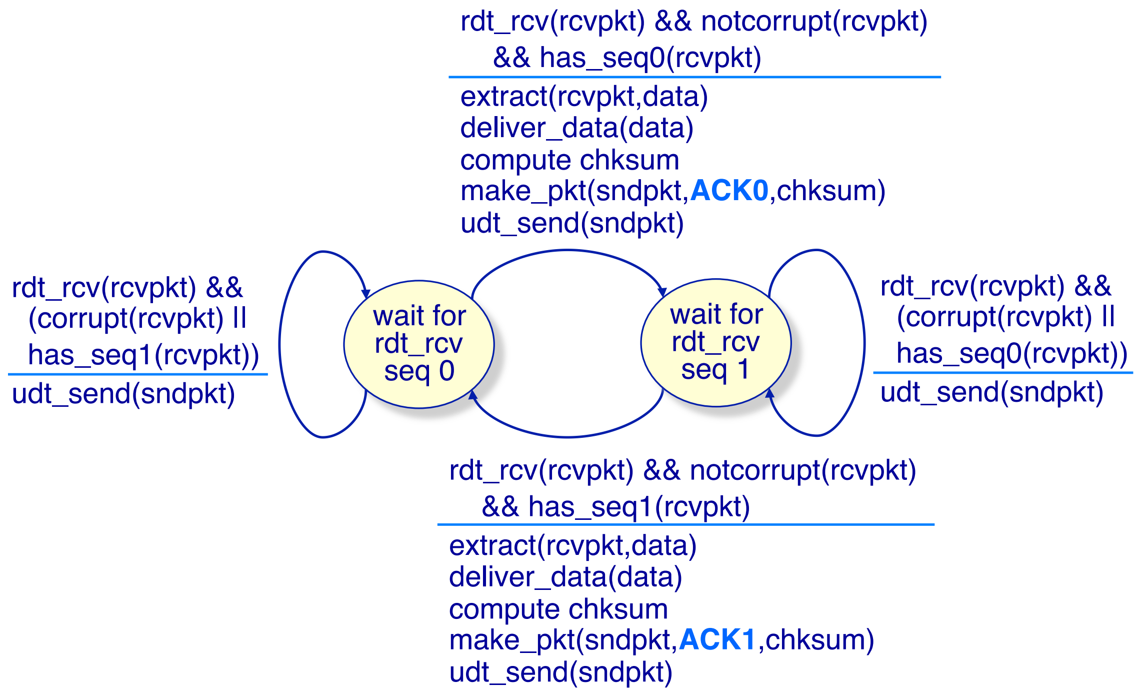
A NAK-free protocol

- Instead of NAKing, receiver sends ACK for last packet received OK
  - » Receiver must include the sequence number of packet being ACKed in ACK
- Receipt of duplicate ACKs at sender is equivalent to a NAK
  - » Sender retransmits current packet



Reliable Data Transfer Protocol 2.2

Receiver state machine to eliminate NAKs



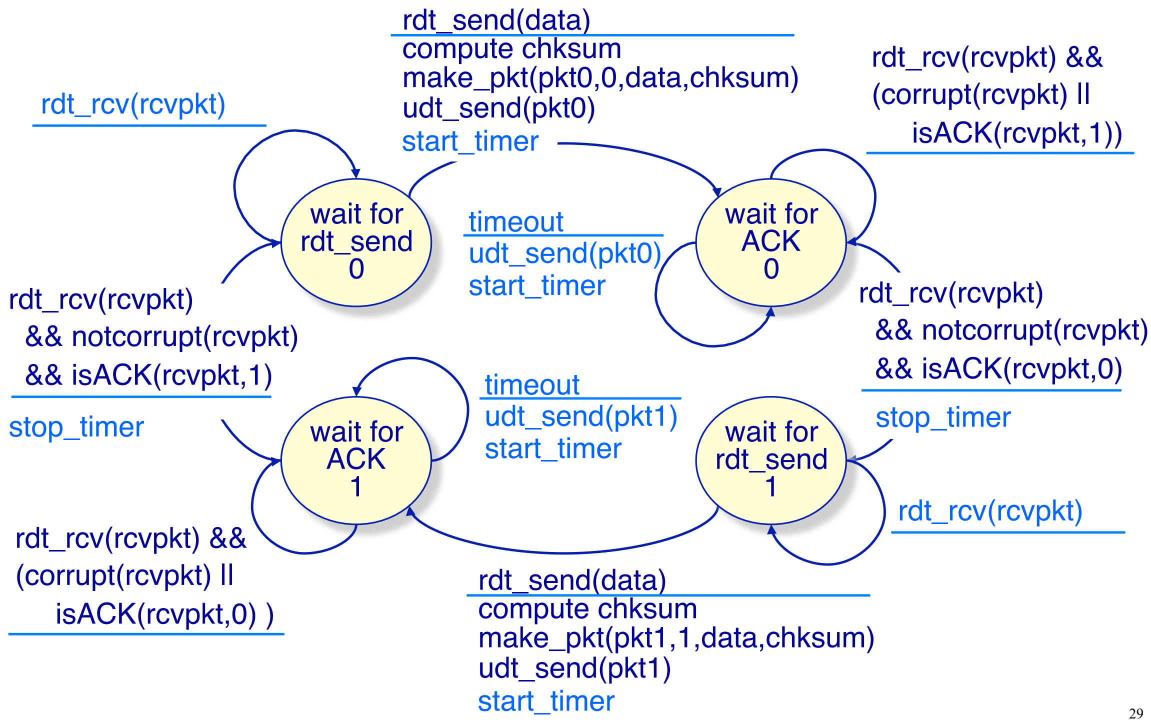
Reliable Data Transfer Protocol 3.0

Dealing with channels with errors *and* loss

- Now assume the underlying channel can also lose packets
- New problem: How to detect loss?
  - » Are checksums, ACKs, sequence numbers, retransmissions enough?
- Approach: sender waits "reasonable" amount of time and retransmits if no ACK received in this time
  - » Requires the use of a countdown timer
- What if packet (or ACK) just delayed beyond its timer?
  - » Retransmission will be duplicate…
  - » But use of sequence numbers already handles this!

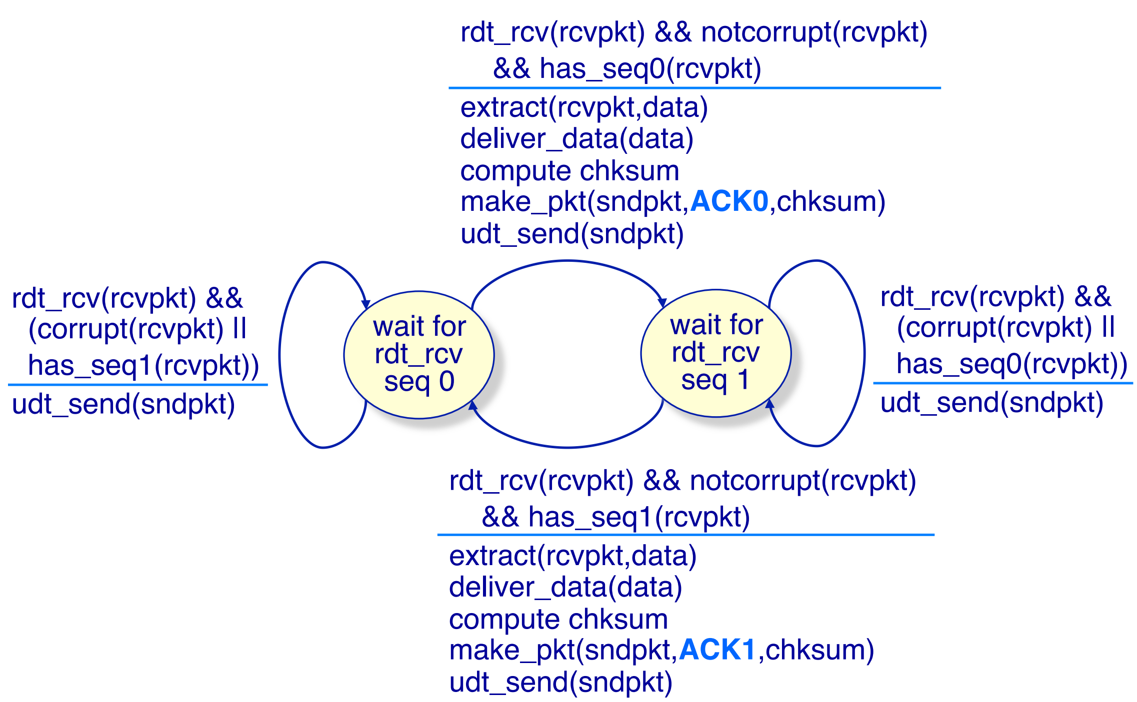
Reliable Data Transfer Protocol 3.0

Sender state machine to handle lost/garbled packets



Receiver State Machine for RDT 2.2

What changes are needed to handle lost/garbled packets?



Fundamental Transport Layer Services

Principles of reliable data transfer

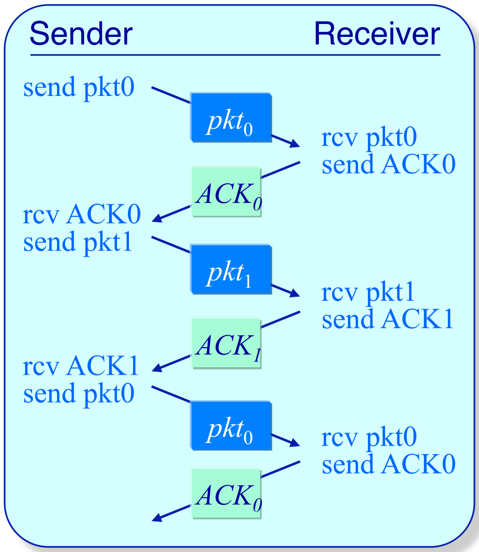
- Use acknowledgements (ACKs) to indicate that a packet has been received
- Simple protocol:
  - » *stop-and-wait* - can't send a new packet until the previous packet has been acknowledged
  - » packet loss - sender sets a timer and re-sends the packet if no ACK received when timer expires
  - » ACK loss - ACKs are not retransmitted

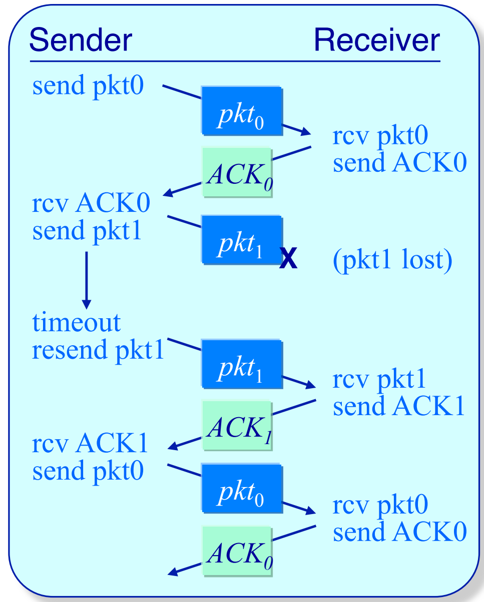
RDT 3.0

Overview

- Sender
  - » put a sequence number (0 or 1) on each packet
  - » when receive an non-duplicate ACK
    - advance seqno
    - reset the timer
    - send the next packet
  - » when receive a duplicate ACK
    - wait for a non-duplicate ACK
  - » if timer expires before ACK received
    - re-send the previous packet
- Receiver
  - » keep track of which seqno expected next (0 or 1)
  - » when receive the next seqno expected
    - send an ACK for this seqno
    - advance next seqno expected
  - » when receive a duplicate packet (packet isn't the next expected)
    - re-send last ACK (for last seqno)

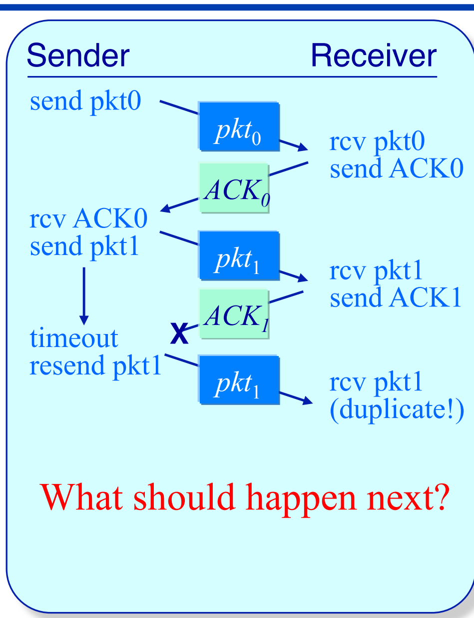
Reliable Data Transfer
Simple Protocol Examples





- Protocol operation with no loss
- Protocol operation with a lost packet

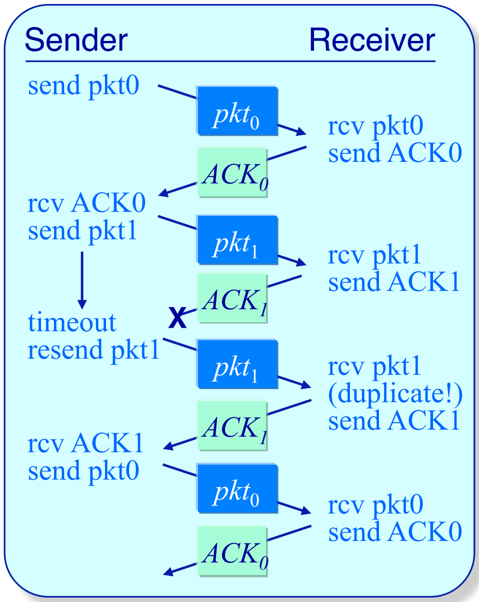
Reliable Data Transfer
Simple Protocol Examples

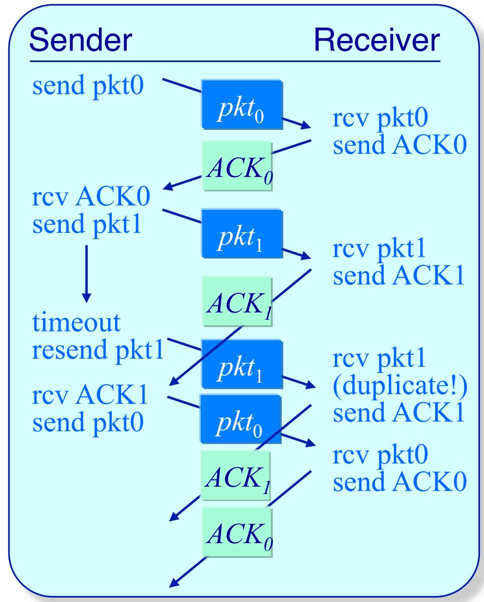


- Protocol operation with a lost ACK

Reliable Data Transfer

Simple Protocol Examples

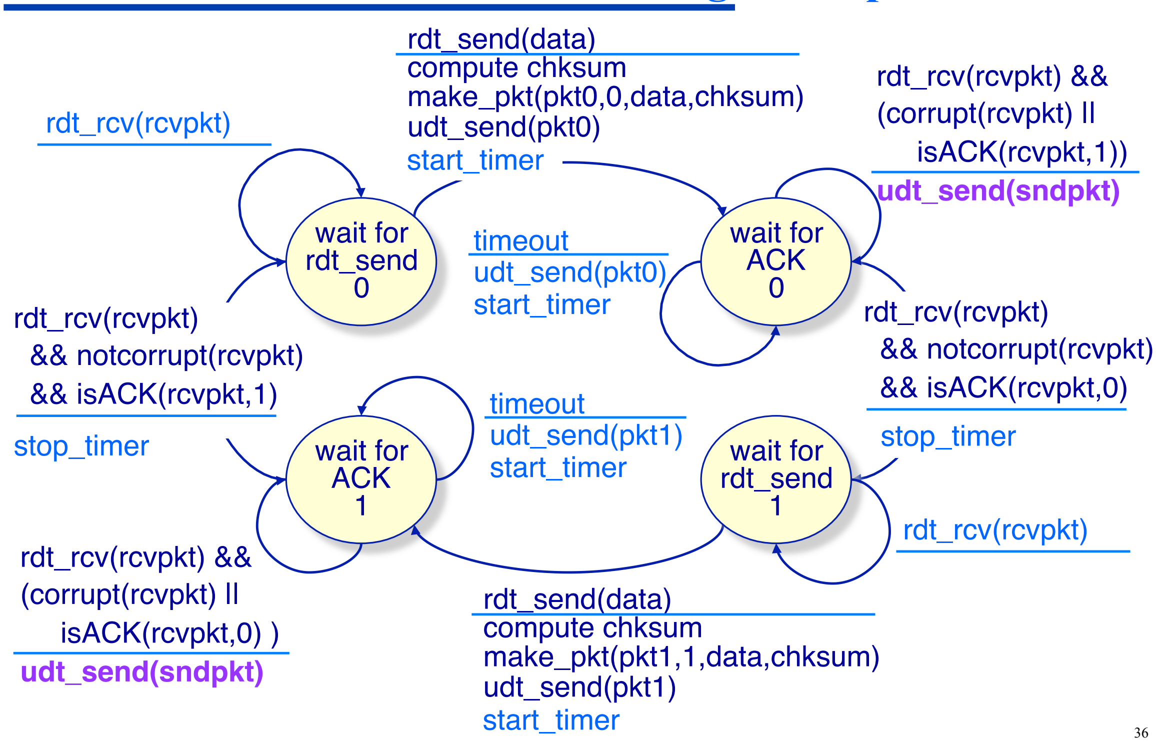




- Protocol operation with a lost ACK
- Protocol operation with a poor timeout value

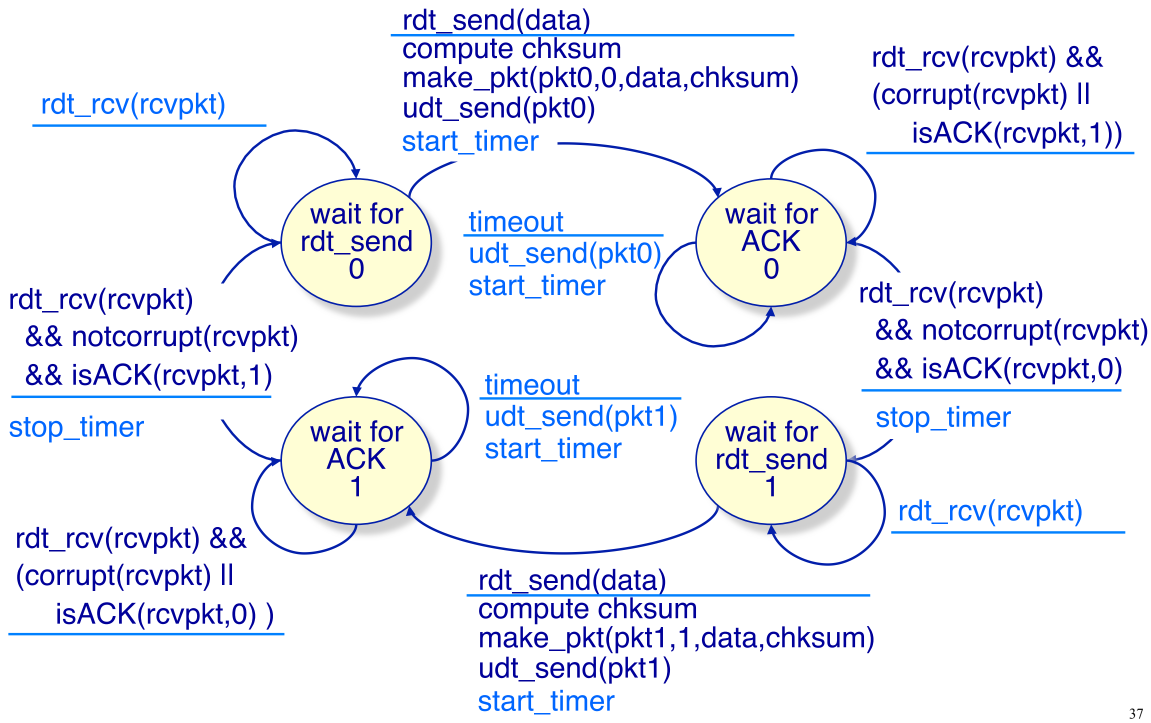
Reliable Data Transfer Protocol 3.0

Sender state machine to handle lost/garbled packets



Reliable Data Transfer Protocol 3.0

Sender state machine to handle lost/garbled packets



Transport Protocol Performance

Performance of RDT 3.0

- Can an end-system make efficient use of a network under this protocol?
- Consider a 1 Gbps link with 15 *ms* end-to-end propagation delay
- How busy is the network?

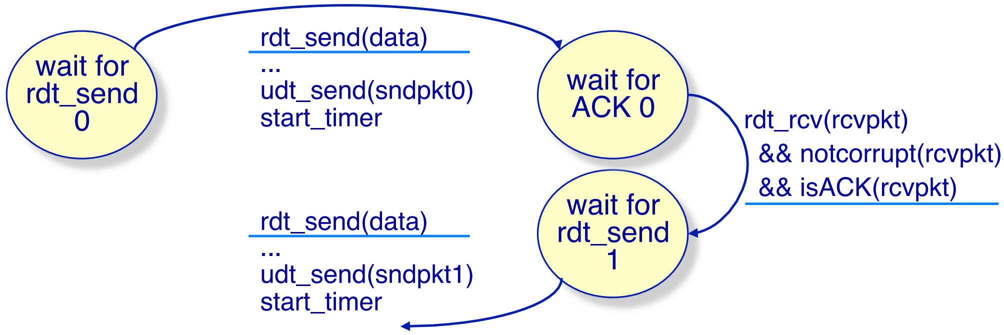
$$utilization = \frac{time\ network\ busy}{observation\ interval} = \frac{time\ to\ transmit\ a\ packet}{packet\ generation\ time}$$

- How long does it take to transmit a 1,000 byte packet?

$$\text{transmission time} = \frac{1\ kB\ packet \times 8\ bits/B}{10^9\ bps} = 8\ \mu s$$

- How fast can an end-system transmit packets?

Transport Protocol Performance
Performance of RDT 3.0



- How fast can an end-system transmit packets?
  - Packet generation/transmission time = 8 μs (0.008 ms)
  - Propagation delay to receiver = 15 ms
  - ACK generation/transmission time ≈ 8 μs (0.008 ms)
  - Propagation time for ACK to return to sender = 15 ms
- 1 packet every 30.016 ms

Reliable Data Transfer
Performance

- How busy is the network?

$$utilization = \frac{time\ network\ busy}{observation\ interval} = \frac{time\ to\ transmit\ a\ packet}{packet\ generation\ time}$$

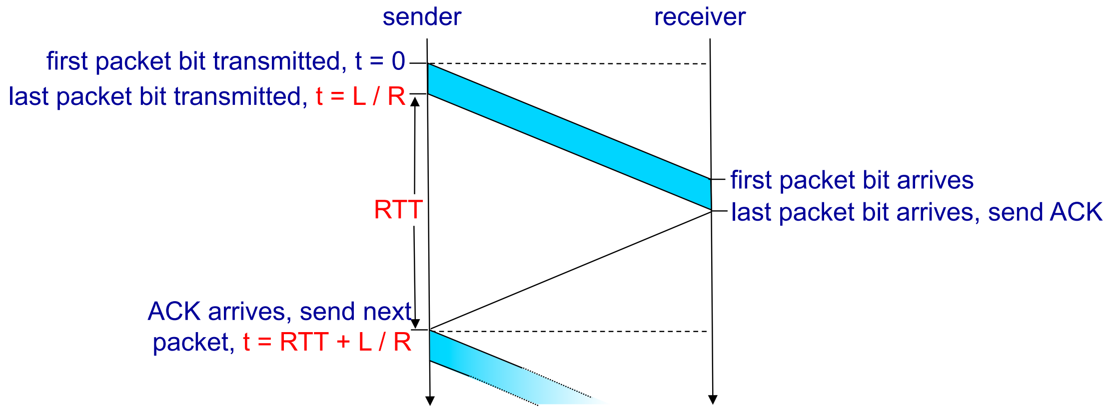
$$= \frac{8\ \mu s}{30.016\ ms} = 0.027\%$$

- Is this good?
  - 1,000 byte packet every 30 *ms* results in (maximum) throughput of 266 *kbps* over a 1 Gbps link!
    (266,000 *bps* over a 1,000,000,000 *bps* link)

Network protocols limit the use of physical resources!

Reliable Data Transfer 3.0

Stop and Wait

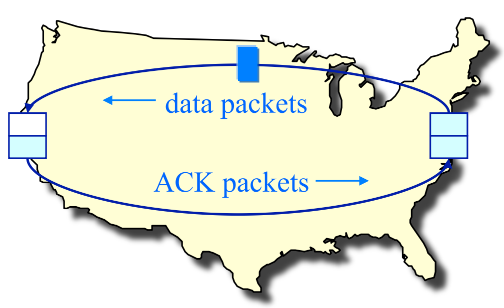


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

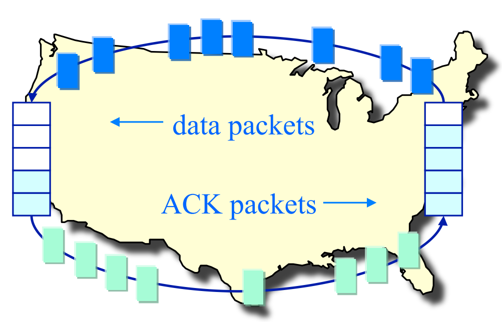
Improving Transport Protocol Performance

Pipelining data transmissions

- Performance can be improved by allowing the sender to have multiple unacknowledged packets "in flight"



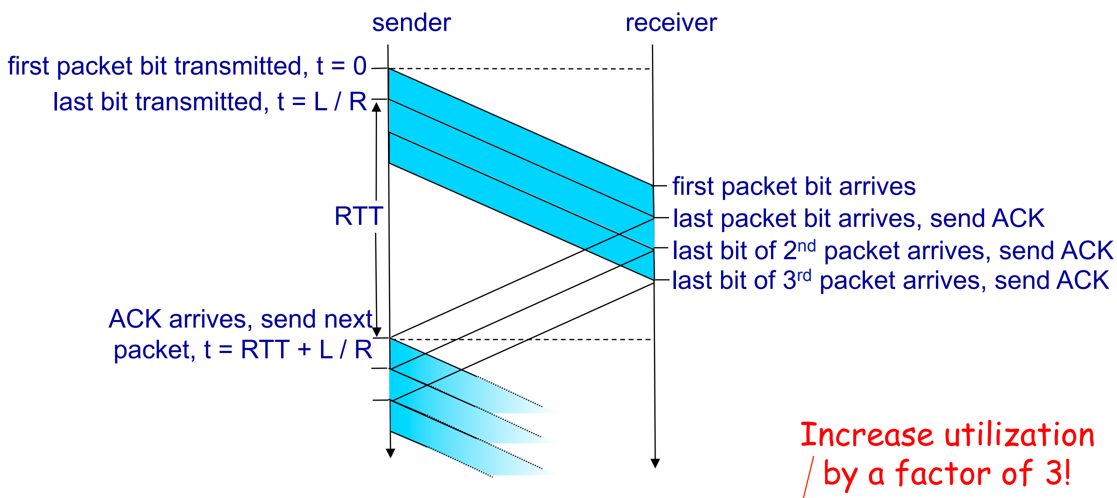
Stop-and-Wait protocol



Pipelined protocol

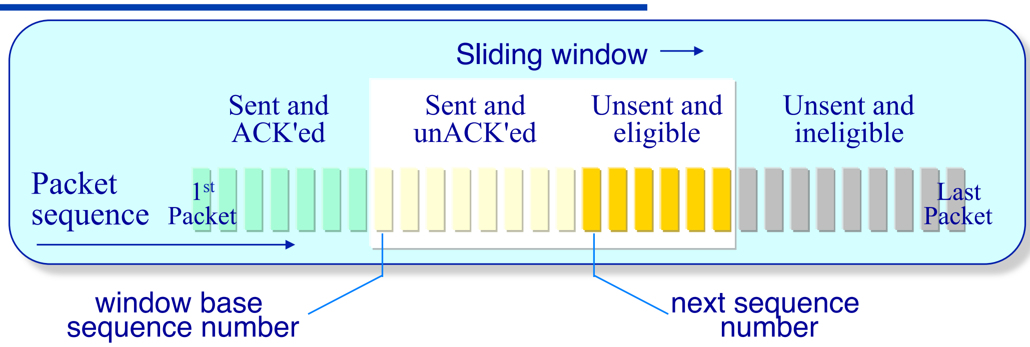
- Issues
  - » The range of sequence numbers must be increased
  - » ACKs need sequence numbers (what packet is being ACKed?)
  - » More packets must be buffered at sender and receiver

Reliable Data Transfer
Pipelining



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

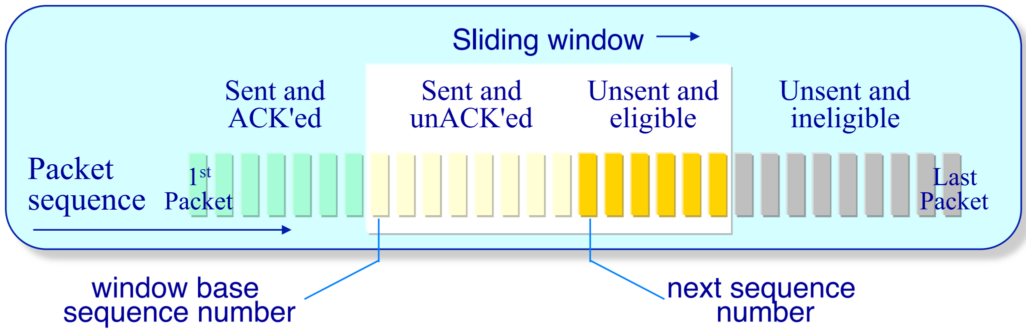
Pipelined Protocols
"Go-Back-*n*" protocols



- Packet header contains a k-bit sequence number
- A "window" of up to $N \leq 2^k$ consecutive, unacknowledged packets allowed to be in-flight
  - » Up to N packets may be buffered at the sender
  - » Window advances as ACKs are received
- Receiver generates "cumulative ACKs"
  - » ACKs contain the sequence number of the last in-order packet received

Pipelined Protocols

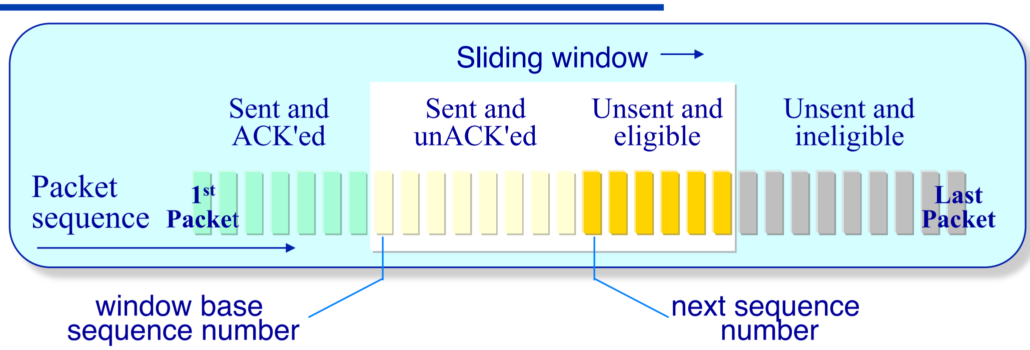
"Go-Back-*n*" protocols



- Receiver protocol
  - Use cumulative ACKs — ACK packet *n* only if all packets numbered less than *n* have been received
  - If losses occur, sender may receive duplicate ACKs
- Sender protocol
  - A timer is set for the each (or just the oldest) in-flight packet
  - On timeout for packet *n*, retransmit packet *n* and all higher number packets in the current window

Go-Back-n Protocol

Sender



- Sender waits for an event:
  - application has data to send
  - timer goes off
  - ACK is received

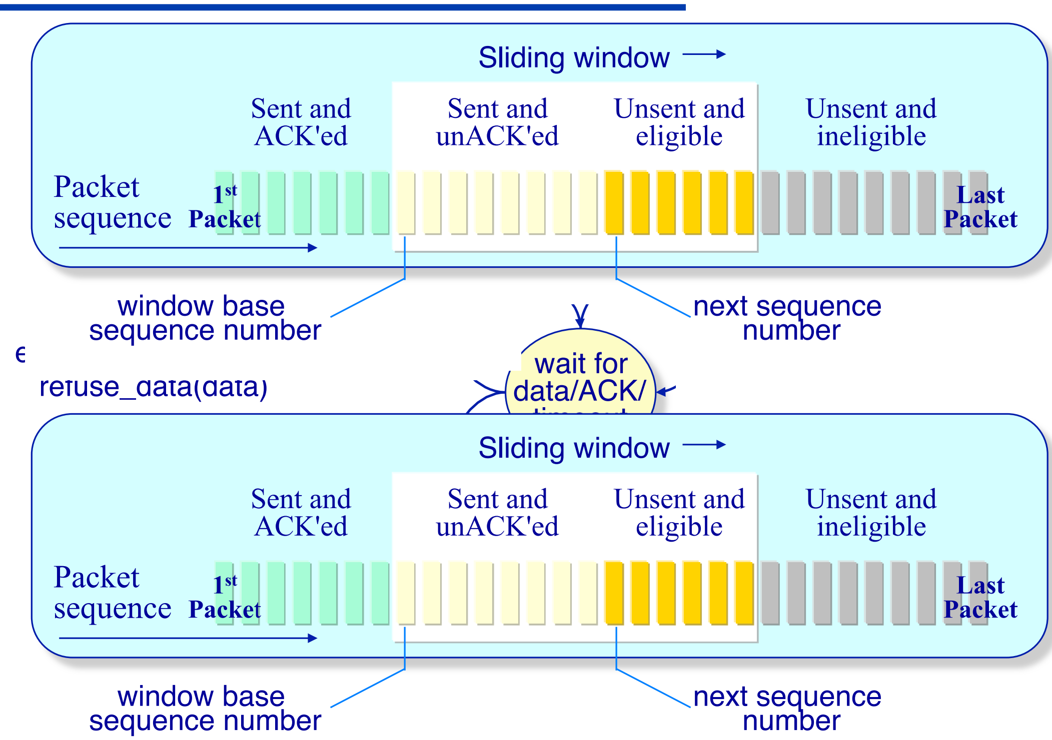
Go-Back-*n* Protocol

Sender extended FSM

- THIS SLIDE INTENTIONALLY LEFT BLANK!

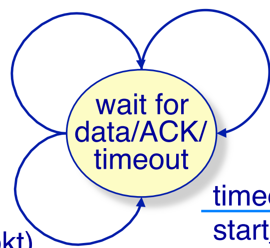
Go-Back-n Protocol

Sender extended FSM



Go-Back-n Protocol
Sender extended FSM

rdt_send(data)
______
```
if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt[nextseqnum],nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum) start_timer
   nextseqnum += 1
  }
else
   refuse_data(data)
```



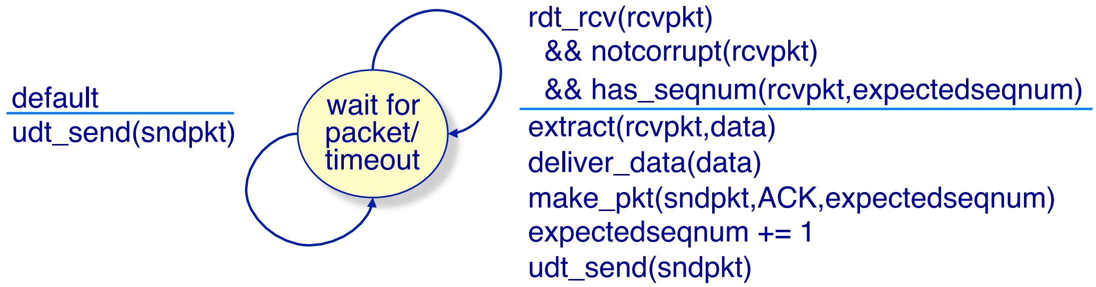
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
______
```
base := getacknum(rcvpkt) + 1
if (base == nextseqnum)
   stop_timer
else
   start_timer
```

timeout
______
```
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
...
udt_send(sndpkt[nextseqnum–1])
```


Go-Back-*n* Protocol
Sender extended FSM

- THIS SLIDE INTENTIONALLY LEFT BLANK!

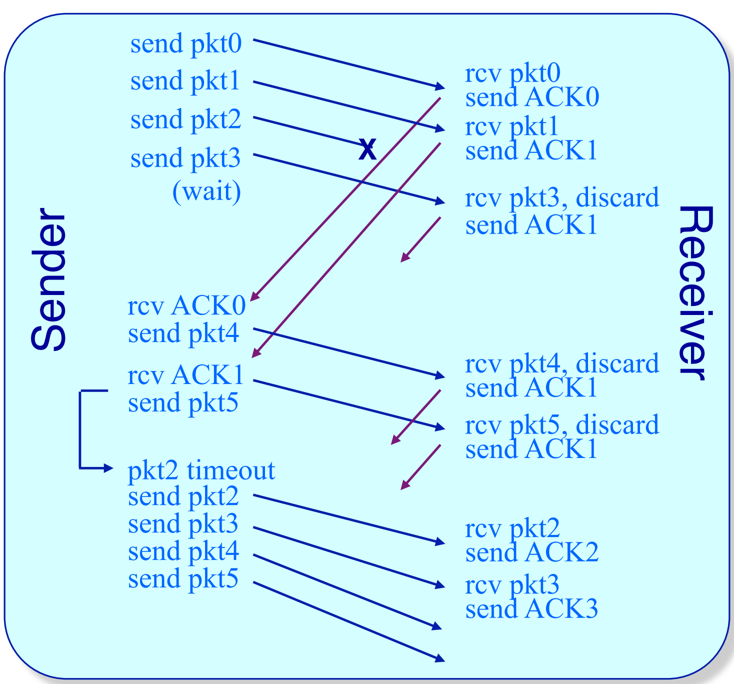
Go-Back-n Protocol
Receiver extended FSM



- In-order packets processed, out-of-order packets discarded
  - » Sender will eventually timeout and retransmit out-of-order packets
  - » Thus the receiver need not buffer any packets

- Always send ACK for correctly-received packet with highest in-order sequence number
  - » May generate duplicate ACKs
  - » But minimal state — need only remember *expectedseqnum*

Go-Back-*n* Protocol
Execution example



- Assume a window size of 4 packets
- Receiver ignores out-of-order packets
- Sender retransmits only on timeout

Transport Protocol Performance
Performance of Go-Back-*n* protocols

- Can an end-system make more efficient use of a network under a Go-Back-*n* protocol?
- Consider again transmitting 1,000 byte packets on a 1 Gbps link with 15 *ms* end-to-end propagation delay

$$utilization = \frac{\textit{time to transmit a packet}}{\textit{packet generation time}}$$

$$\textit{transmission time} = \frac{1\ kB\ packet \times 8\ bits/B}{10^9\ bps} = 8\ \mu s$$

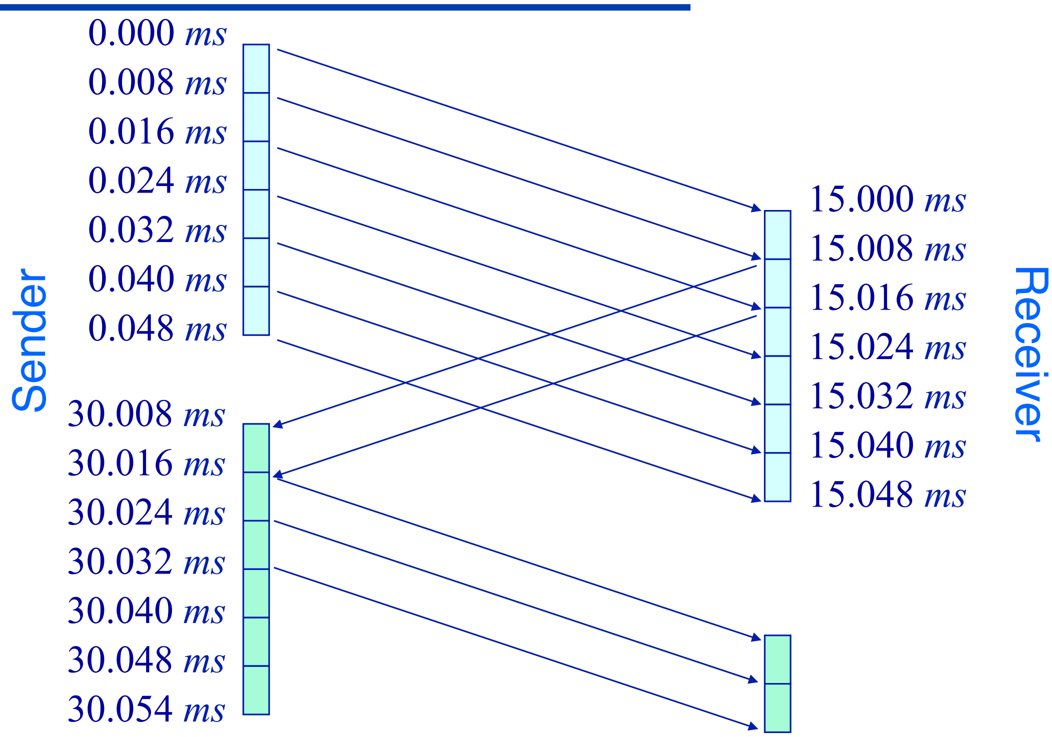
- How fast can an end-system transmit packets?
  - » Depends on the window size!

Transport Protocol Performance
Performance of Go-Back-n protocols

- How fast can an end-system transmit packets?
  - » N packets can be sent before the sender must wait for an ACK
- N packets sent every 30.016 ms
  - » Packet generation/transmission time = 8 μs
  - » Round-trip-time to receiver = 30 ms
  - » ACK generation/transmission time ≈ 8 μs

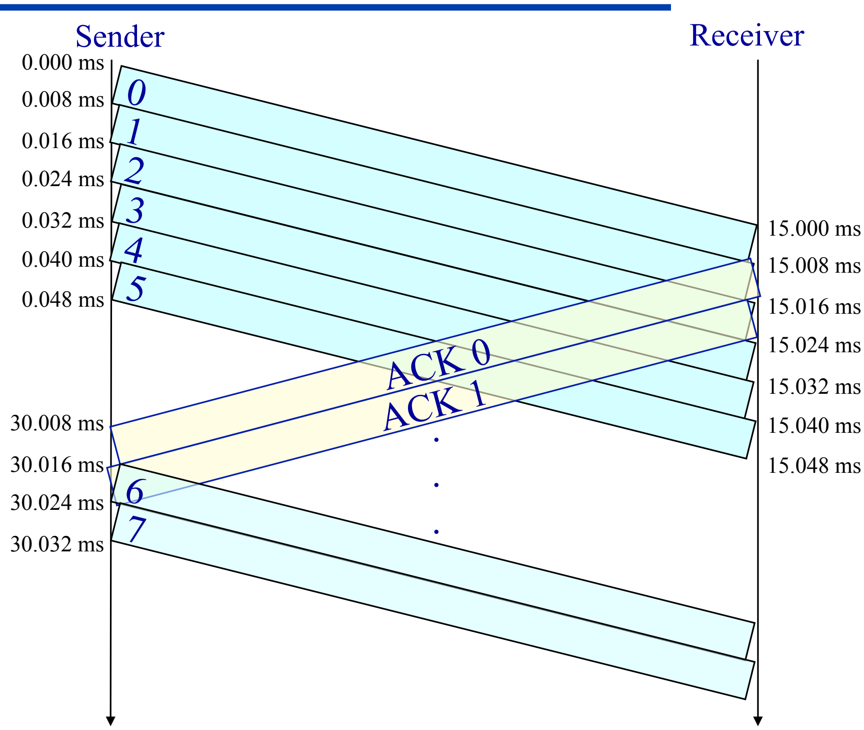
Transport Protocol Performance

Performance of Go-Back-*n* protocols



Transport Protocol Performance

Performance of Go-Back-*n* protocols



Transport Protocol Performance

Performance of Go-Back-*n* protocols

- Performance with a window size of $N = 64$ packets:

$$utilization = \frac{\textit{time to transmit N packets}}{\textit{time to receipt of first ACK}}$$

$$= \frac{512\ \mu s}{30.016\ ms} = 1.7\%$$

A 64x improvement!

- Is this good?
  - » 64 1,000 byte packets every 30 *ms* results in (maximum) throughput of 17 *Mbps* over a 1 Gbps link!

Pipelined Protocols

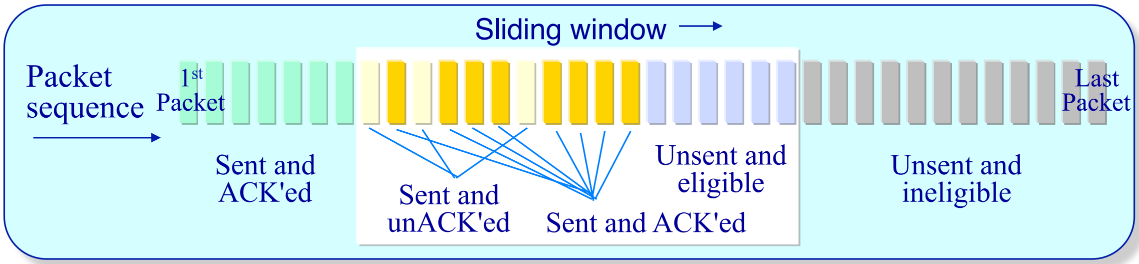
"Selective Repeat" protocols

- Receiver *individually* acknowledges all correctly received packets
  - » Buffers packets as needed for eventual in-order delivery to upper layer
- Sender only resends packets for which an ACK has not been received
  - » Sender maintains a timer for each unACK'ed packet
- Sender window is the same as before
  - » *N* consecutive sequence numbers (Limits the sequence numbers of sent, unACK'ed packets)

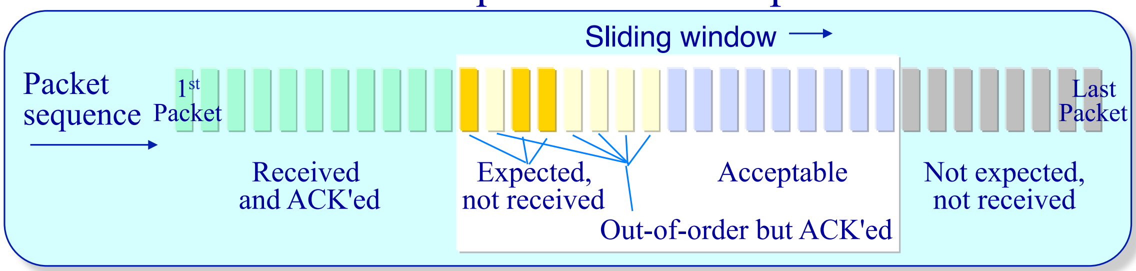
Selective Repeat Protocols

Sender and receiver windows

- Sender's view of sequence number space

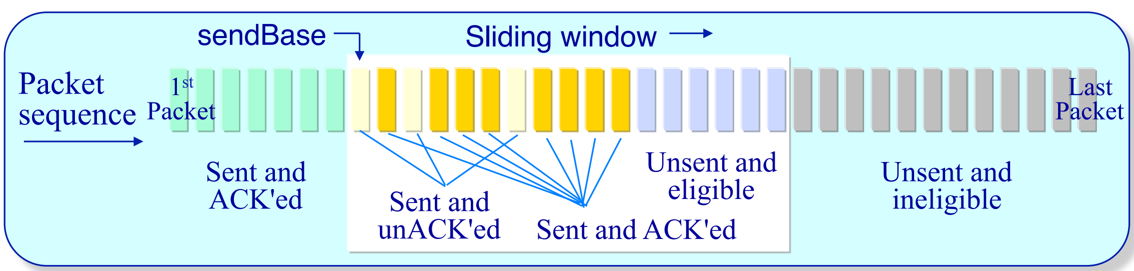


- Receiver's view of sequence number space



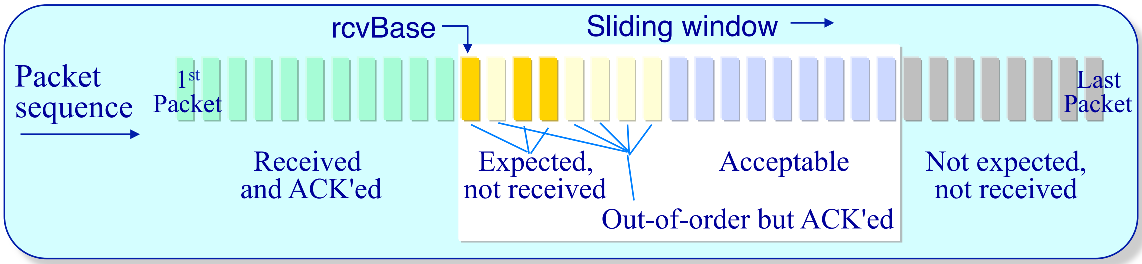
Selective Repeat Protocols

Sender state machine



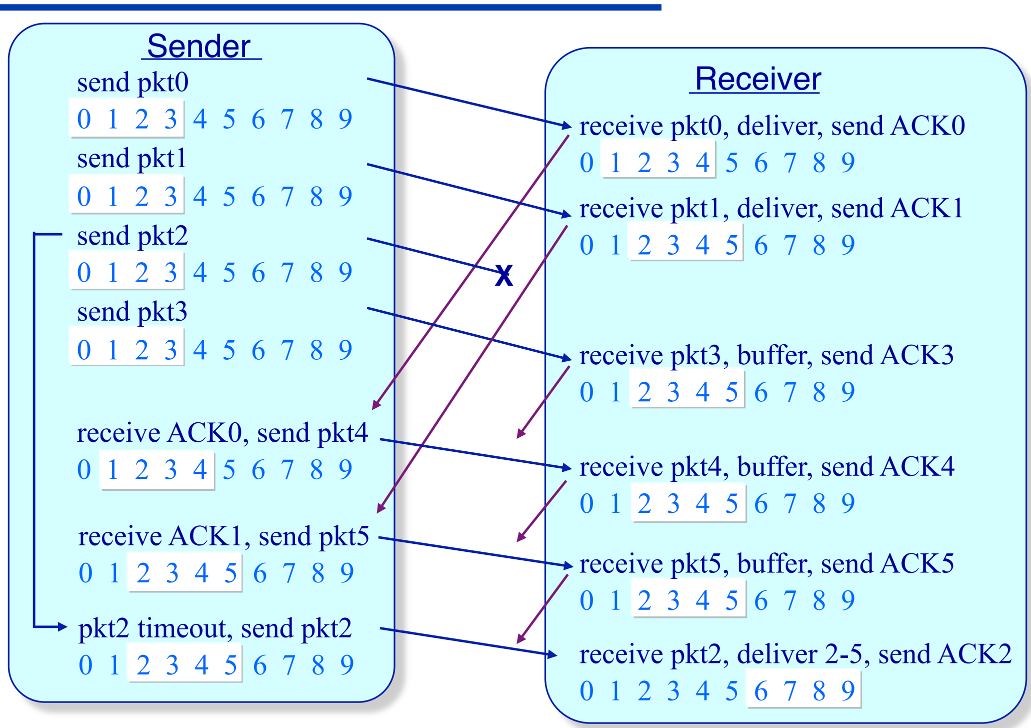
- Call from above:
  - » If next available sequence number is within window, send the packet and start a timer for it
- Timeout for packet *n*:
  - » Resend packet *n*, restart timer for packet *n*
- ACK received for packet with sequence number *n*:
  - » If *n* in [*sendBase*, *sendBase*+*N*–1] then mark packet *n* as received
  - » If *n* == *sendBase*, advance *sendBase* to next highest unACKed sequence number and move the window forward by that amount

Selective Repeat Protocols
Receiver state machine



- Packet *n* in [*rcvbase*, *rcvbase*+*N*–1] correctly received:
  - » Send an ACK for packet *n*
  - » If packet *n* is out-of-order then buffer
  - » If *n* == *rcvBase*, deliver packet *n*, and all other buffered consecutive in-order packets, to application, and advance the window by the number of delivered packets
- Packet *n* in [*rcvbase*–*N*, *rcvbase*–1] received:
  - » Send an ACK for packet *n*
- Otherwise discard packet (without ACK'ing)

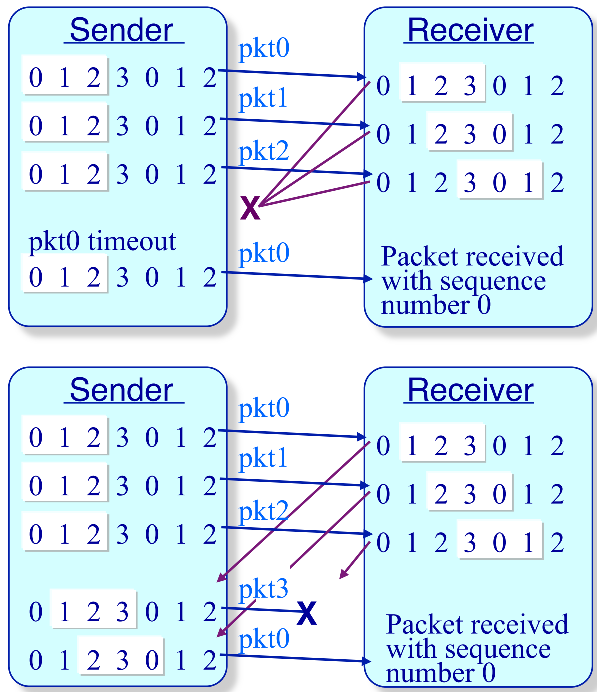
Selective Repeat Protocols
Execution example



Selective Repeat Protocols

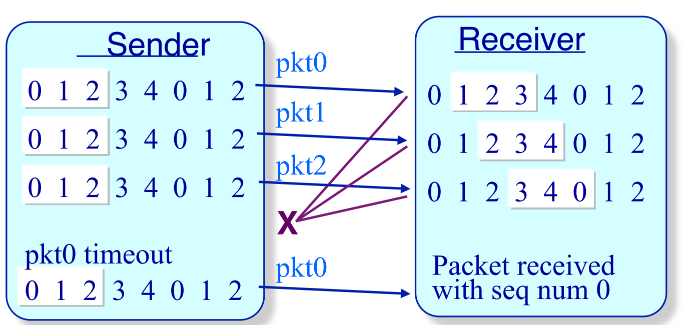
Window state ambiguity

- How many sequence numbers do we need?
  - » As many as the largest number of packets that can be in flight?
- If the sequence number space is close to the window size then the receiver can get confused

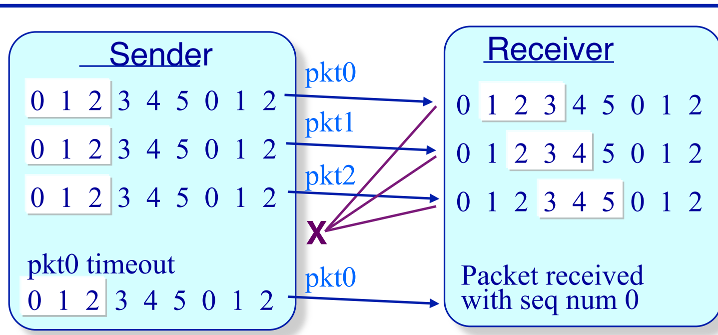


Selective Repeat Protocols

Window state ambiguity



5 seq nums



6 seq nums