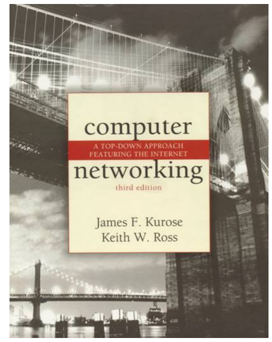
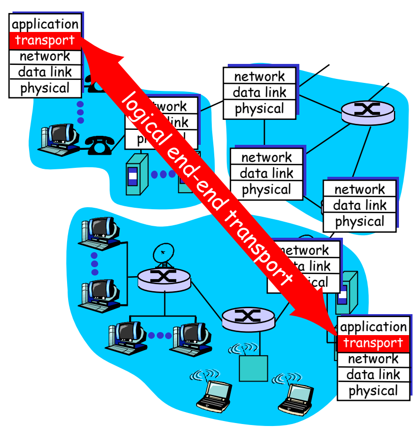
Chapter 3
Transport Layer



Computer Networking: A Top Down Approach Featuring the Internet, 3rd edition.
Jim Kurose, Keith Ross
Addison-Wesley, July 2004.

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

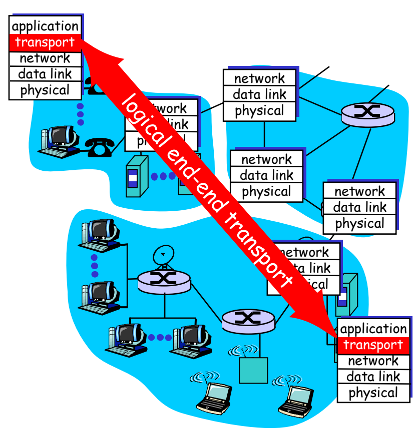


Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

Internet transport protocols: TCP and UDP

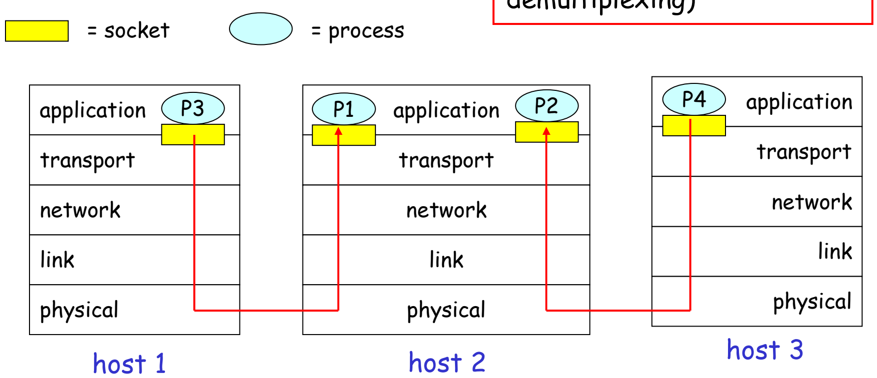
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - extension of "best-effort" IP from host to process
- services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing/demultiplexing

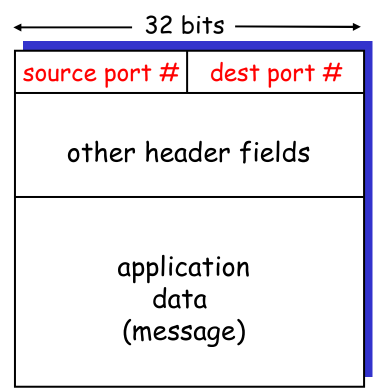
Demultiplexing at rcv host:
delivering received segments to correct socket

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

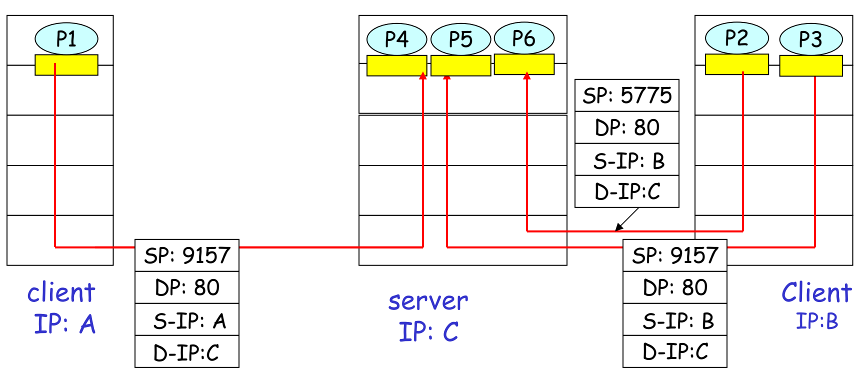


TCP/UDP segment format

Connection-oriented demux

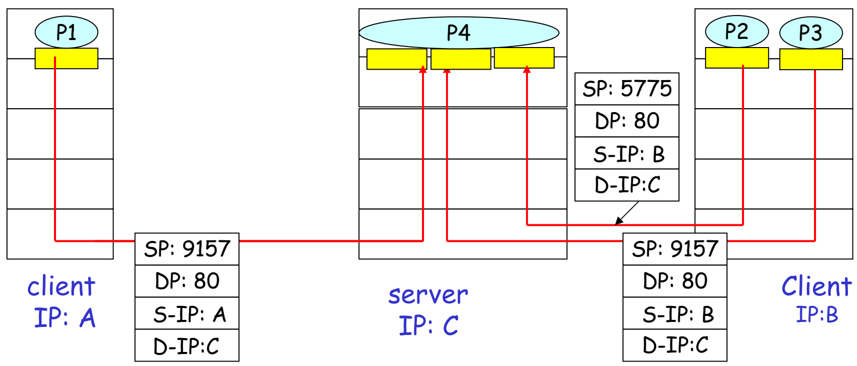
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



(SP=source port, DP=destinaiton port)

Connection-oriented demux: Threaded Web Server



Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(33111);
DatagramSocket mySocket2 = new
   DatagramSocket(33222);
```

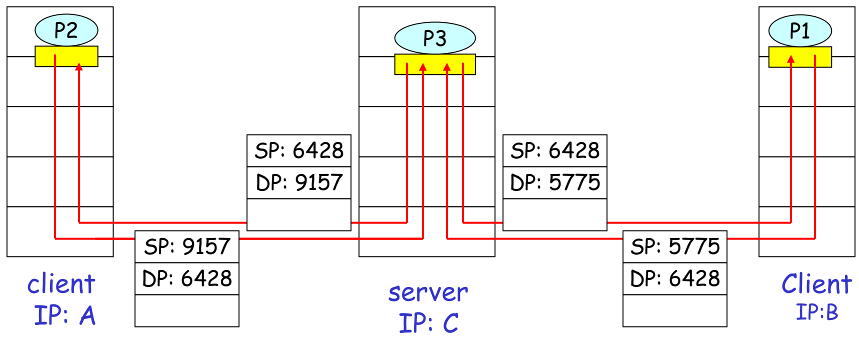
- UDP socket identified by two-tuple:

(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)

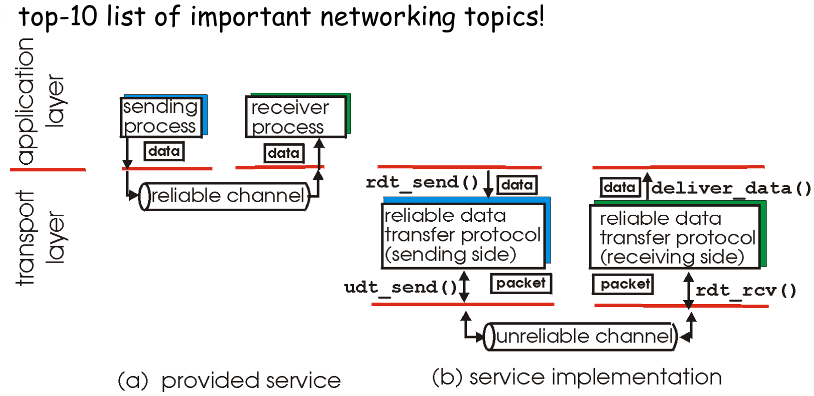
```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



SP provides "return address"

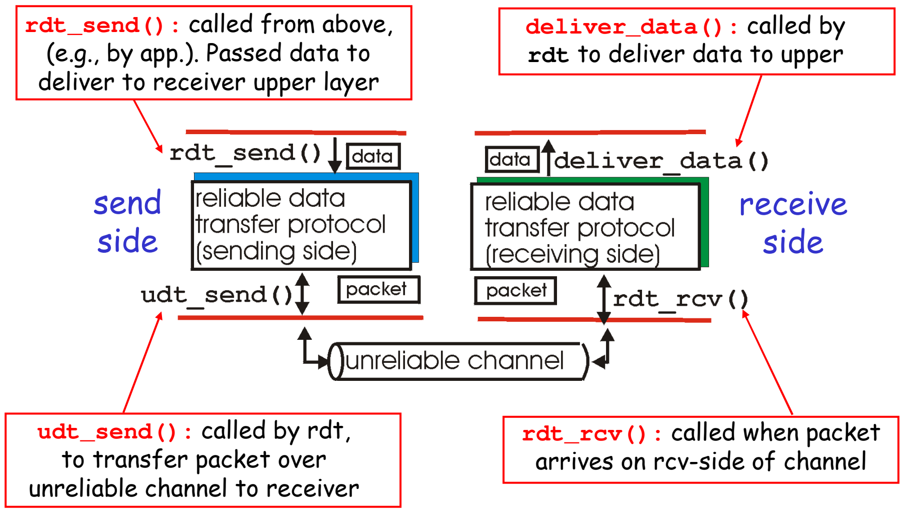
Principles of Reliable data transfer

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



- (rdt = reliable data transfer, udt=underlying data transfer)

Reliable data transfer: getting started



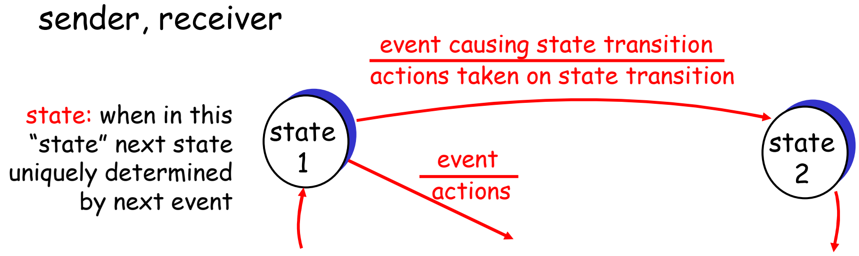
Step by Step development of a **reliable data transfer protocol** based on an **unreliable channel**

From stop-and-wait to TCP

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



Step by Step development of a **reliable data transfer protocol**

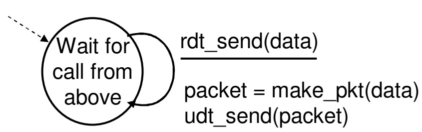
Step 1

...let's suppose the underlying channel is reliable... ☺

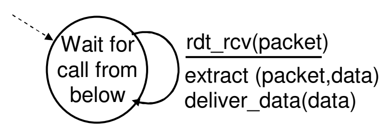
Protocol Rdt 1.0:

reliable transfer over a reliable channel

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



sender



receiver

Step by Step development of a **reliable data transfer protocol**

Step 2

...let's suppose the underlying channel has bit errors but no packet loss ☺

(approach: stop&wait ACK/NACK protocol)

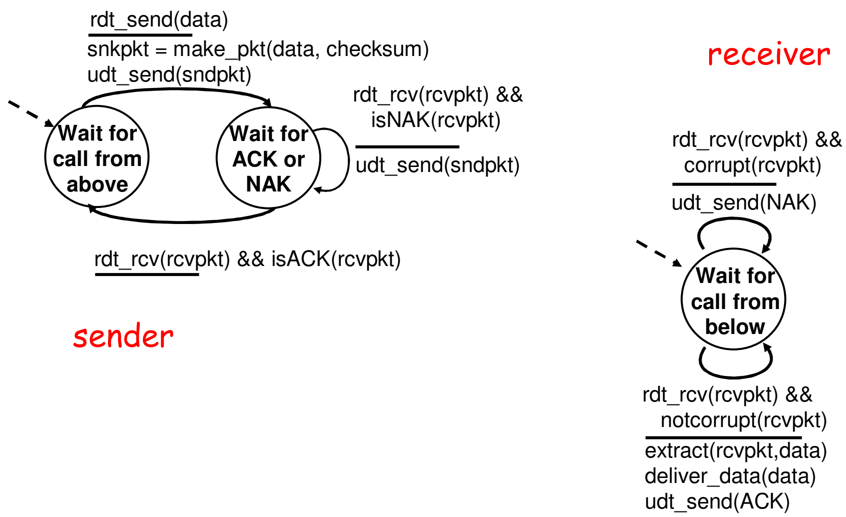
Rdt 2.0: channel with bit errors

- 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

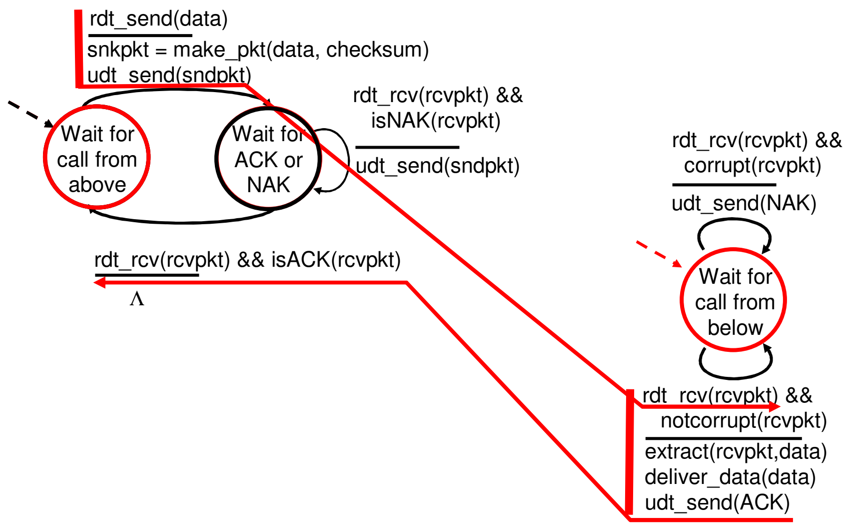
(this makes it an **ARQ** (automatic repeat request) protocol)

- New mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback:
    - control msgs (ACK,NAK) rcvr->sender

rdt2.0: FSM specification

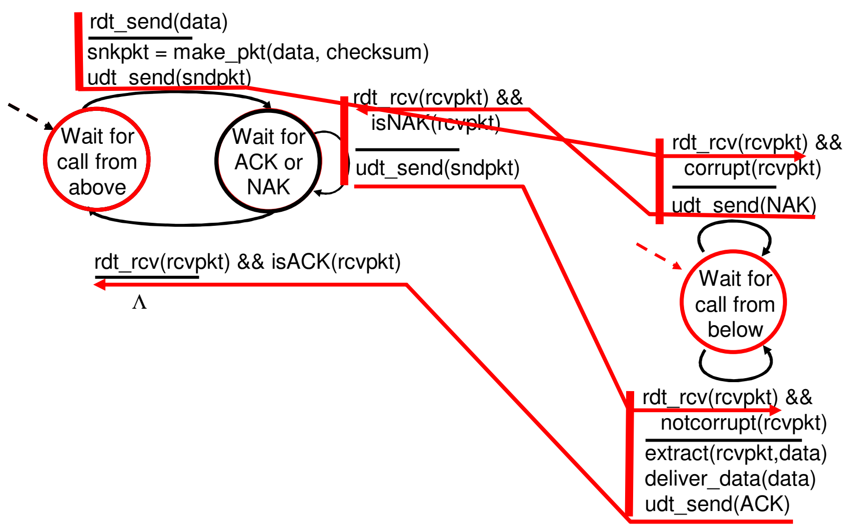


rdt2.0: operation with no errors



Transport Layer 3-21

rdt2.0: error scenario (no loss!)



Transport Layer 3-22

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

- IMPROVE 2.0 to 2.1 - New mechanisms :
  - duplicate detection (packet numbering and seq. checking)

rdt2.1: discussion

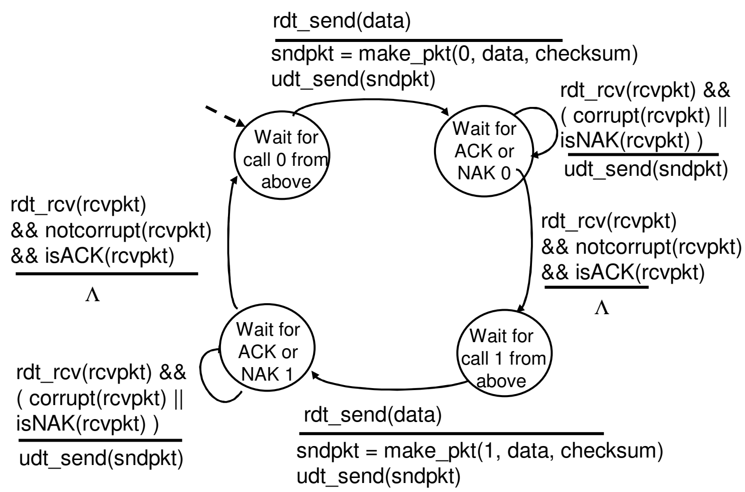
Sender:

- 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. #

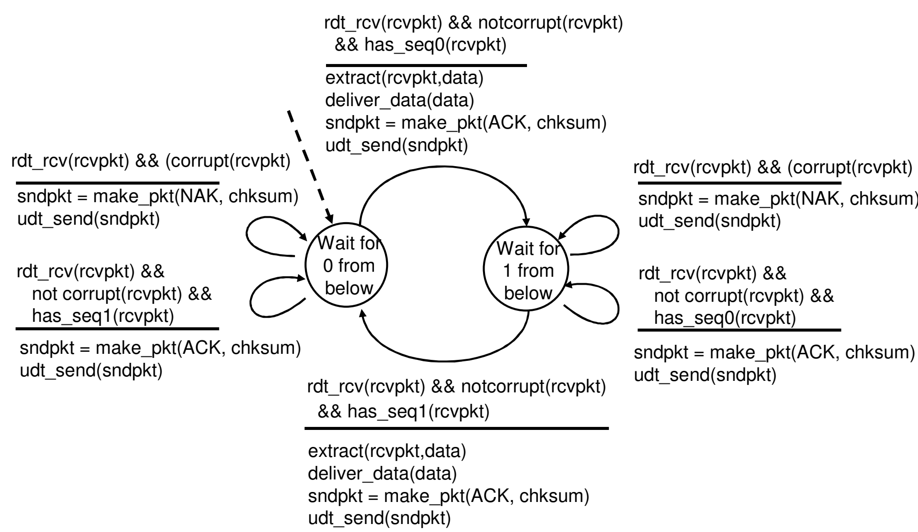
Receiver:

- 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

rdt2.1: sender, handles garbled ACK/NAKs



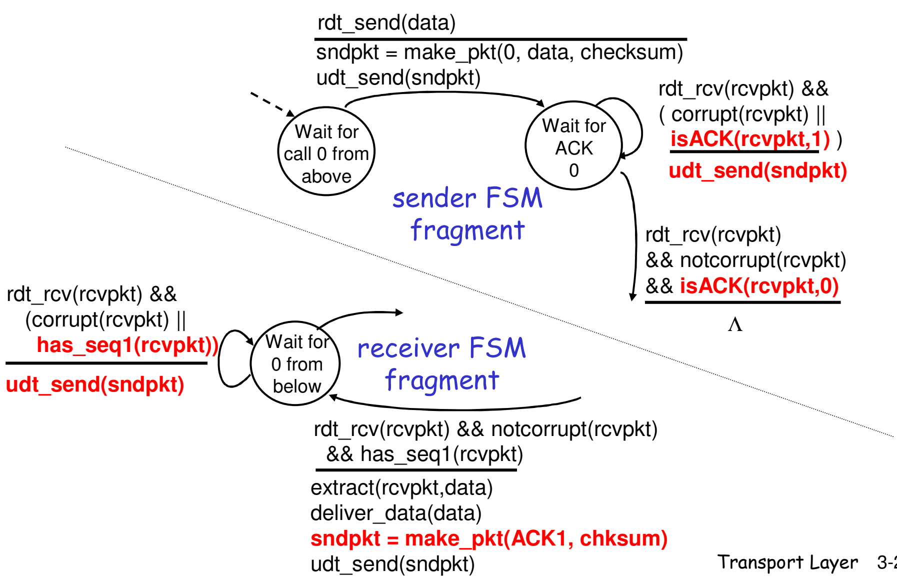
rdt2.1: receiver, handles garbled ACK/NAKs



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



Step by Step development of a **reliable data transfer protocol**

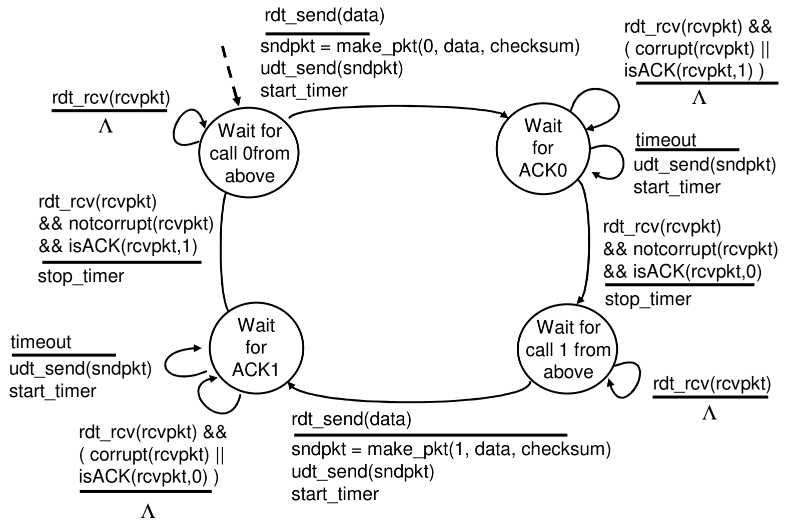
Step 3
...let's suppose the underlying channel non only has bit errors but also packet loss ☹
(approach: a stop and wait protocol –
(that waits 2much))

rdt3.0: channels with errors *and* loss

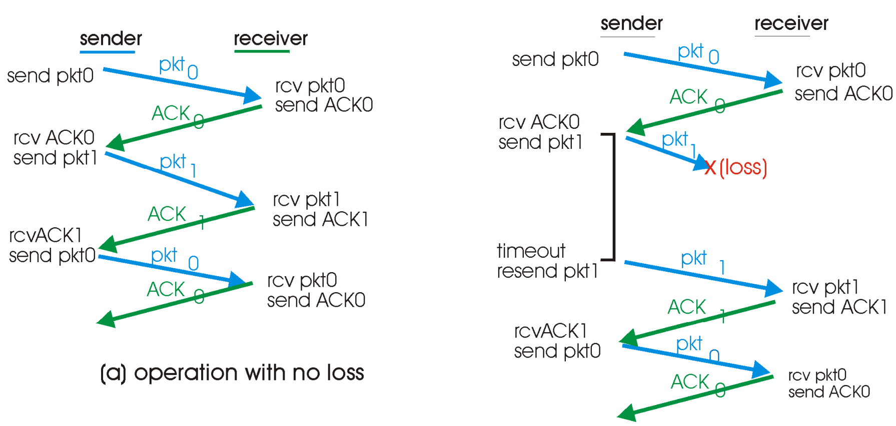
New assumption: underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

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



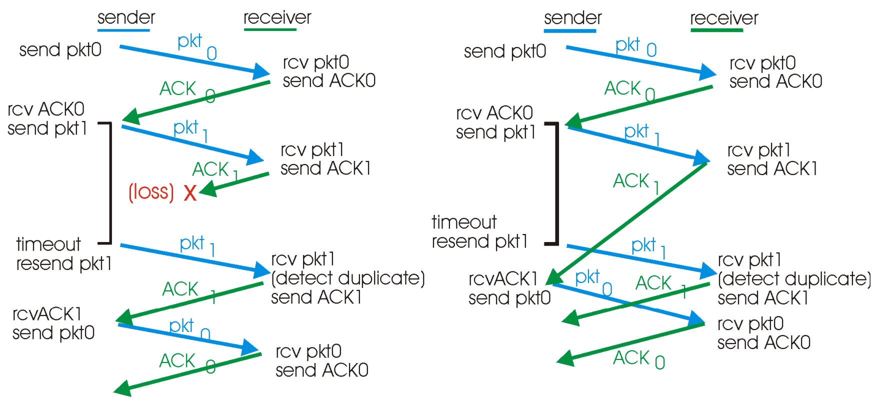
rdt3.0 in action



(a) operation with no loss

(b) lost packet

rdt3.0 in action



(c) lost ACK

(d) premature timeout

Performance of rdt3.0

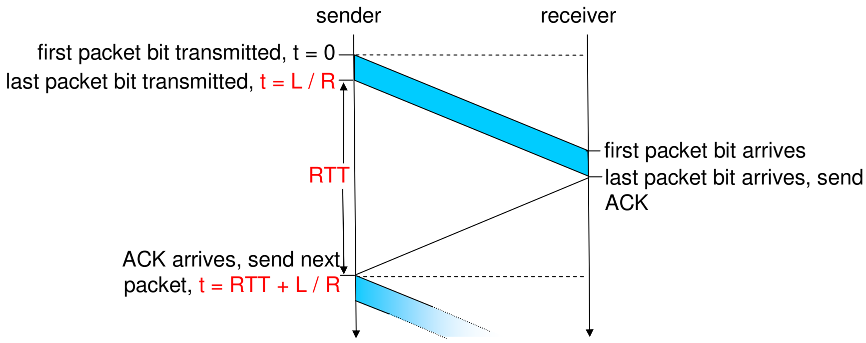
- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

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

  - U_{sender}: utilization – fraction of time sender busy sending
  - 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link!!!
  - the network protocol limits the use of physical resources

rdt3.0: stop-and-wait operation



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

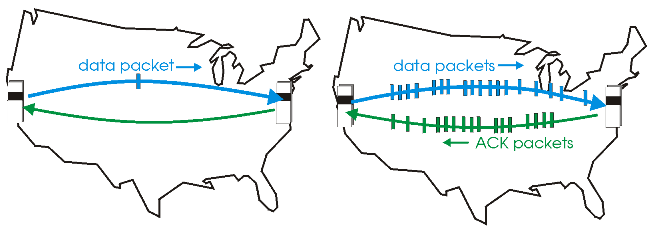
Step by Step development of a reliable data transfer protocol

Step 4
...stop wasting time in waiting...
(approach: a pipelined, windowed protocol (or two))

Pipelined protocols

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

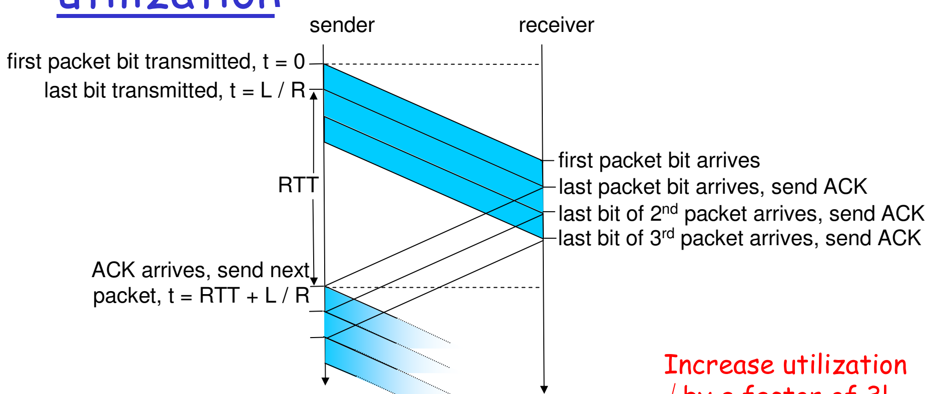
- range of sequence numbers must be increased
- buffering at sender and/or receiver



(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, e.g 3 pkts: increased utilization



Increase utilization by a factor of 3!

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

the TCP case (I)

- Data is sent by TCP in segments that are typically 1460 bytes in length.
  (1460 + 20bytes for TCP header + 20 bytes for IP header makes a 1500 bytes payload for Ethernet frames)
- If the sender is permitted a window size of only 1 segment, the sender transmits a single segment, and waits for an acknowledgement from the receiver.
  - If the transmission delay between sender and receiver is long, this means very low throughput (very few segments transferred per unit time) since both sender and receiver spend most of their time waiting for messages to be transmitted from one end of the connection to the other.

the TCP case (II)

- In order to improve throughput, multiple segments are transmitted by the sender without waiting for the next acknowledgement from the receiver.
- The TCP window is the amount of unacknowledged data in flight between the sender and the receiver.
- The TCP window is an estimate of the upper bound on the number of segments that will fit in the length of the pipe between sender and receiver.

the TCP case (III)

- If the pipe is pretty big, and the round-trip delay is long, a lot of segments will fit in the network between the sender and receiver, so the window size needs to be pretty big. How big?

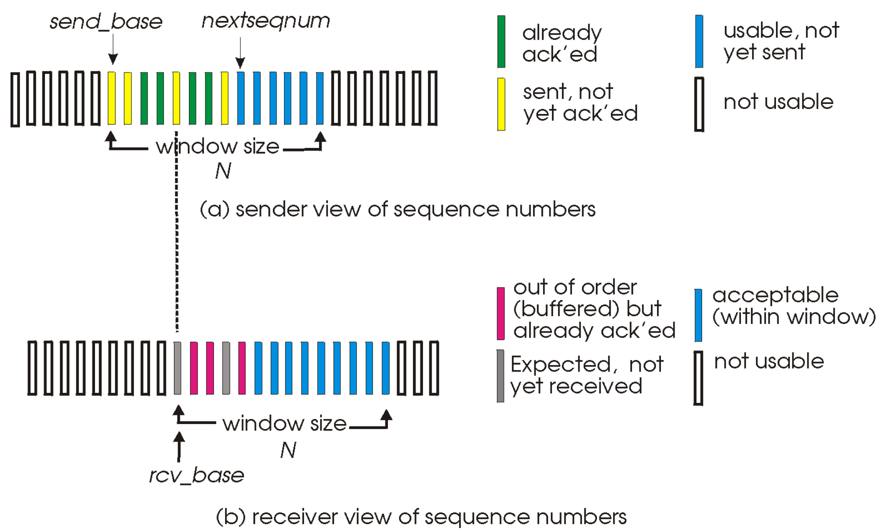
  window size = bandwidth * delay

- For a 10 Mbit/s bandwidth and a round-trip delay of 0.010 sec, that gives a window size of about 12 KB
  (= 9 (nine) 1460-byte segments)

Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - rcv has to buffer pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sdr must keep a 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 is the 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

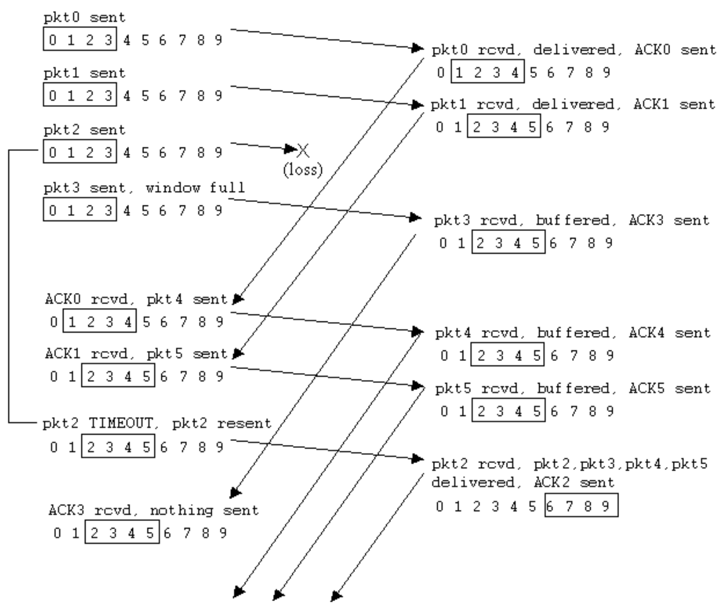
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action

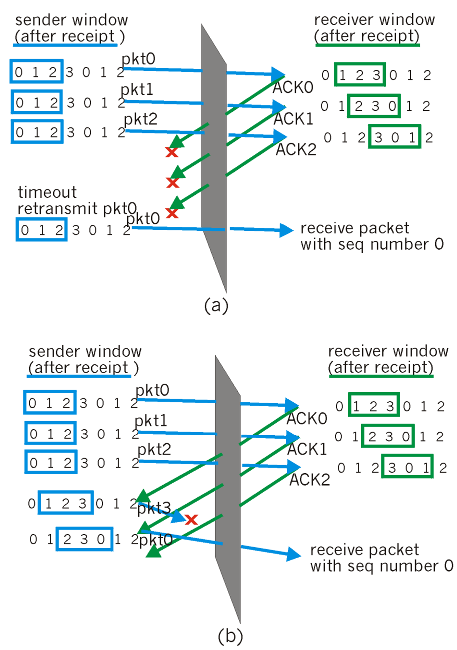


Selective repeat: dilemma

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)

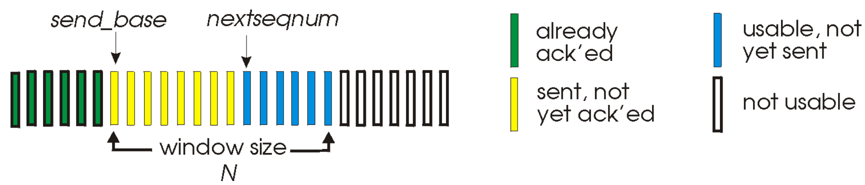
Q: what relationship between seq # size and window size?



Go-Back-N

Sender:

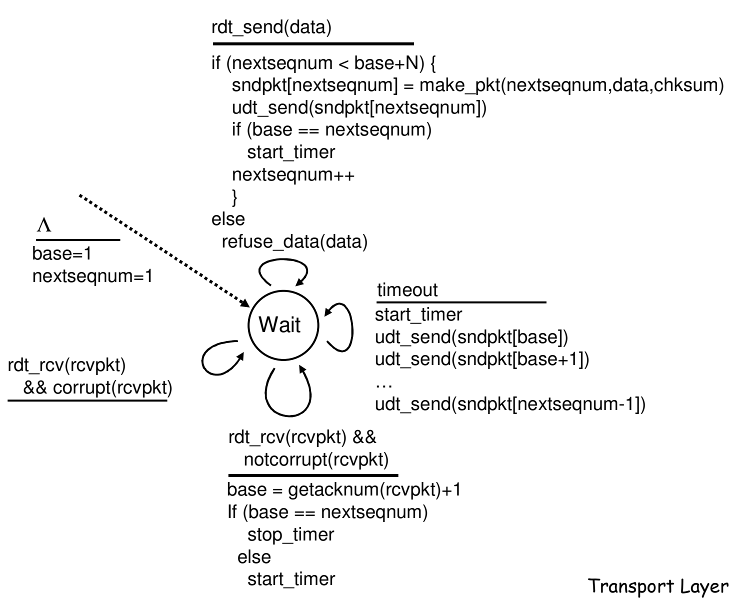
- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



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

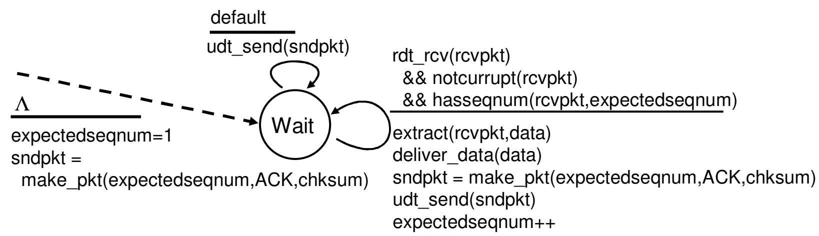
Transport Layer 3-47

GBN: sender extended FSM



Transport Layer 3-48

GBN: receiver extended FSM

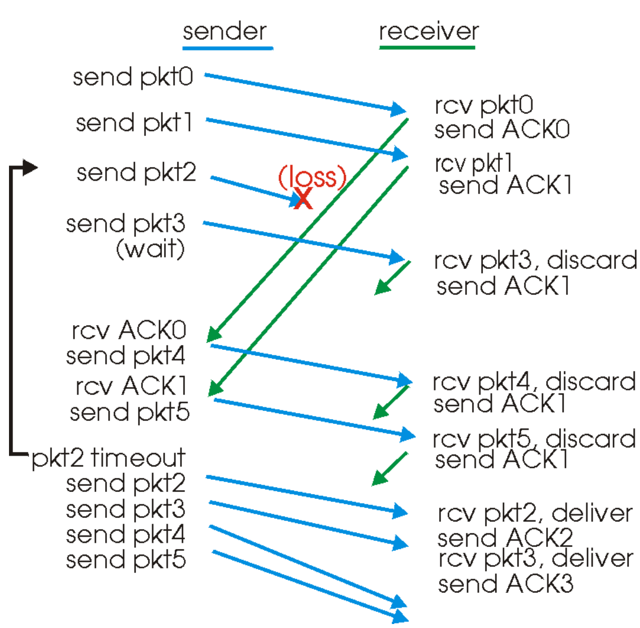


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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



Step by Step development of a **reliable data transfer protocol: the real TCP**

Step 5
TCP
a variable-size window
connection oriented, flow controlled
(full-duplex) protocol etc...

The TCP service

- Connection oriented service
- Streaming service
- Full-duplex service
- Reliable service
- End-to-end semantics
- Flow control and congestion avoidance

The TCP service

- Connection oriented service
  - Before two applications can start sending data to each other they must establish a TCP connection between them, which is terminated upon completion of the communication session.

The TCP service

- Connection oriented service
- Streaming service
  - Once a TCP connection is established between two application processes the sender writes a stream of bytes into the connection and the receiver reads them out of the connection. TCP hides its packet mode of operation to applications, and hide possible message boundaries to the network.

The TCP service

- Connection oriented service
- Streaming service
- Full-duplex service
  - Data flow in both ways on the same connection (ACKs and data use biggybacking)

The TCP service

- Connection oriented service
- Streaming service
- Full-duplex service
- Reliable service
  - Delivery of every single byte, in order and with no duplication (however no timing guaranties) through
    - acknowledgments
    - and retransmissions after timeouts or duplicted acks

The TCP service

- Connection oriented service
- Streaming service
- Full-duplex service
- Reliable service
- End-to-end semantic
  - When a TCP sender receives an ACK, it is guaranteed that the data have reached the receiver safely. The semantic would be violated if any intermediate node generates an ACK in behalf of the destination

The TCP service

- Connection oriented service
- Streaming service
- Full-duplex service
- Reliable service
- End-to-end semantic
- Flow control and congestion avoidance
  - flow control mechanism (for end-to-end flow)
    - sender should not exceed receiver's capacity
  - congestion avoidance mechanisms
    - multiple senders should not exceed underlying network's capacity