

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

1

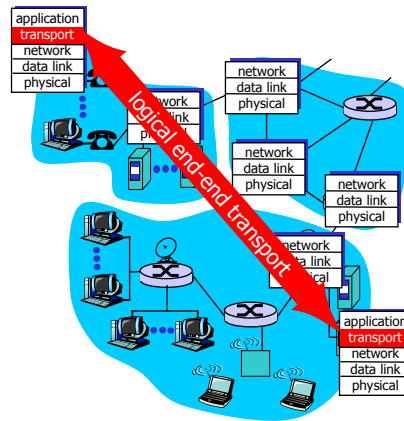
Transport layer: Outline

- **Transport-layer services**
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
 - Segment structure
 - Reliable data transfer
 - Flow control
 - Connection management
- Principles of congestion control
- TCP congestion control

2

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



3

Transport vs. network layer

- ❑ *Network layer*: logical communication between hosts
- ❑ *Transport layer*: logical communication between processes
 - Relies on, enhances, network layer services

Household analogy:

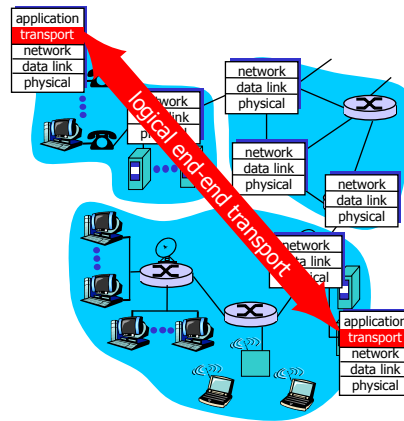
*12 kids sending letters
to 12 kids*

- ❑ Processes = kids
- ❑ App messages = letters in envelopes
- ❑ Hosts = houses
- ❑ Transport protocol = Ann and Bill
- ❑ Network-layer protocol = postal service

4

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



5

Transport layer: Outline

- ❑ Transport-layer services
- ❑ Multiplexing and demultiplexing
- ❑ Connectionless transport: UDP
- ❑ Principles of reliable data transfer
- ❑ Connection-oriented transport: TCP
 - Segment structure
 - Reliable data transfer
 - Flow control
 - Connection management
- ❑ Principles of congestion control
- ❑ TCP congestion control

6

Multiplexing/demultiplexing

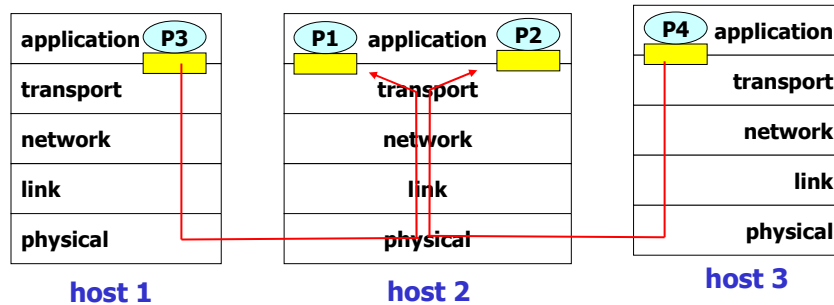
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

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

■ = socket ○ = process



7

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
  DatagramSocket(99111);
```

```
DatagramSocket mySocket2 = new
  DatagramSocket(99222);
```

- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:

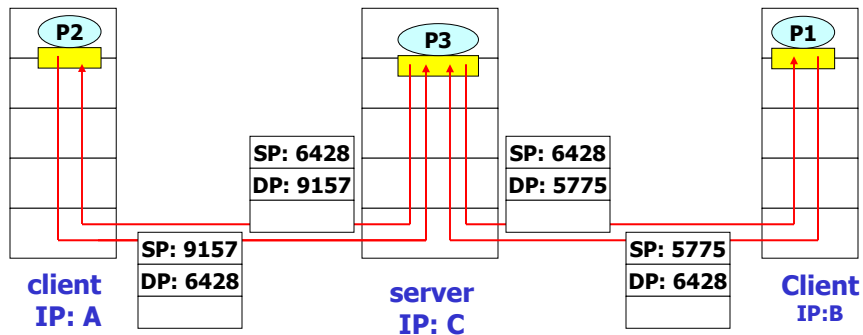
- Checks destination port number in segment
- Directs UDP segment to socket with that port number

- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

8

Connectionless demux (cont.)

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



SP provides "return address"

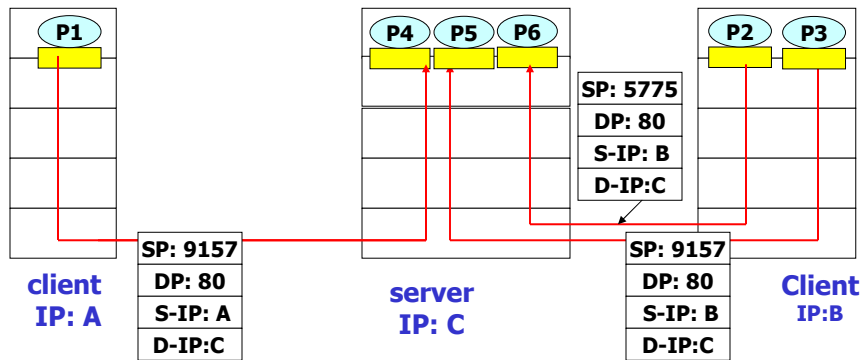
9

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

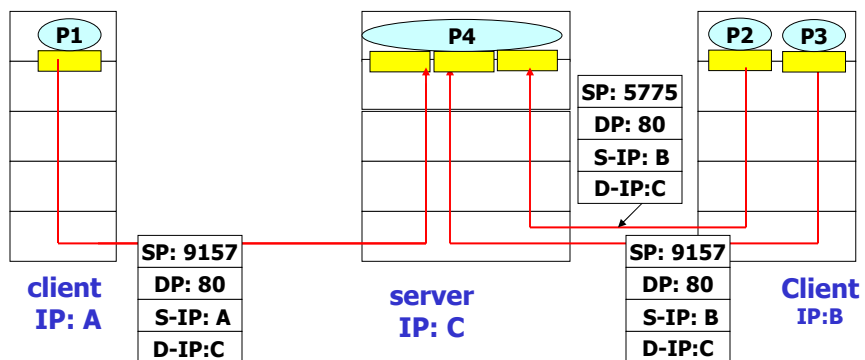
10

Connection-oriented demux (cont.)



11

Connection-oriented demux: Threaded Web Server



12

Transport layer: Outline

- ❑ Transport-layer services
- ❑ Multiplexing and demultiplexing
- ❑ **Connectionless transport: UDP**
- ❑ Principles of reliable data transfer
- ❑ Connection-oriented transport: TCP
 - Segment structure
 - Reliable data transfer
 - Flow control
 - Connection management
- ❑ Principles of congestion control
- ❑ TCP congestion control

13

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

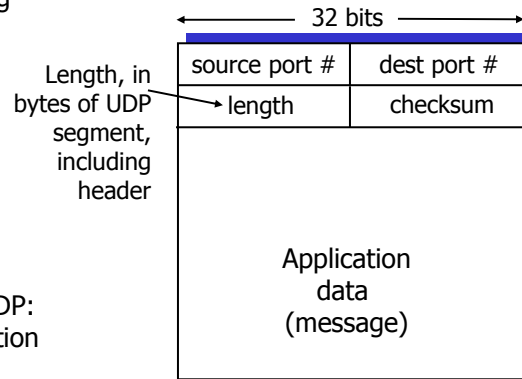
Why is there a UDP?

- ❑ 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

14

UDP: More

- ❑ Often used for streaming multimedia apps
 - Loss tolerant
 - Rate sensitive
- ❑ Other UDP uses (why?):
 - DNS
 - SNMP
- ❑ Reliable transfer over UDP: add reliability at application layer
 - Application-specific error recover!



UDP segment format

15

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

16

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
		<hr/>															
wraparound	①	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
		<hr/>															
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

17

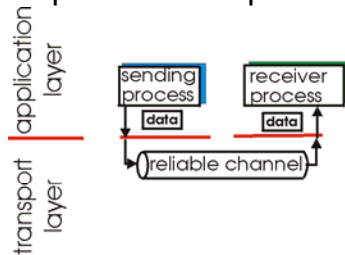
Transport layer: Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
 - Segment structure
 - Reliable data transfer
 - Flow control
 - Connection management
- Principles of congestion control
- TCP congestion control

18

Principles of reliable data transfer

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



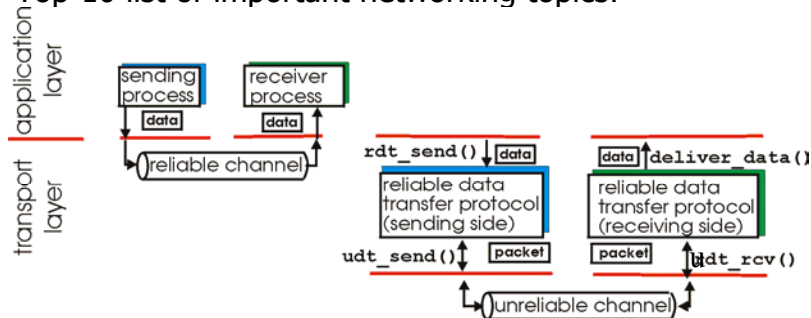
(a) provided service

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

19

Principles of reliable data transfer

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



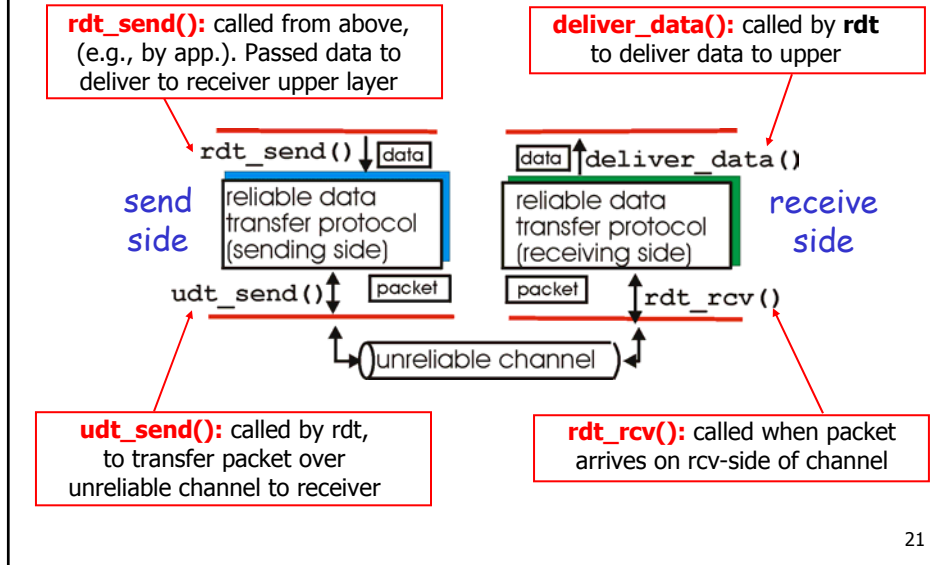
(a) provided service

(b) service implementation

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

20

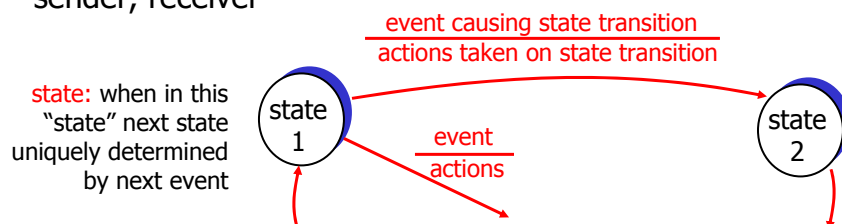
Reliable data transfer: Getting started



Reliable data transfer: Getting started

We'll:

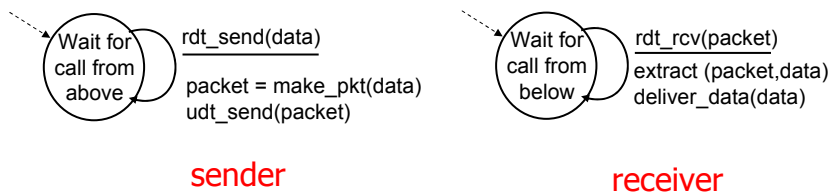
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
 - But control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver



22

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



23

Rdt2.0: Channel with bit errors

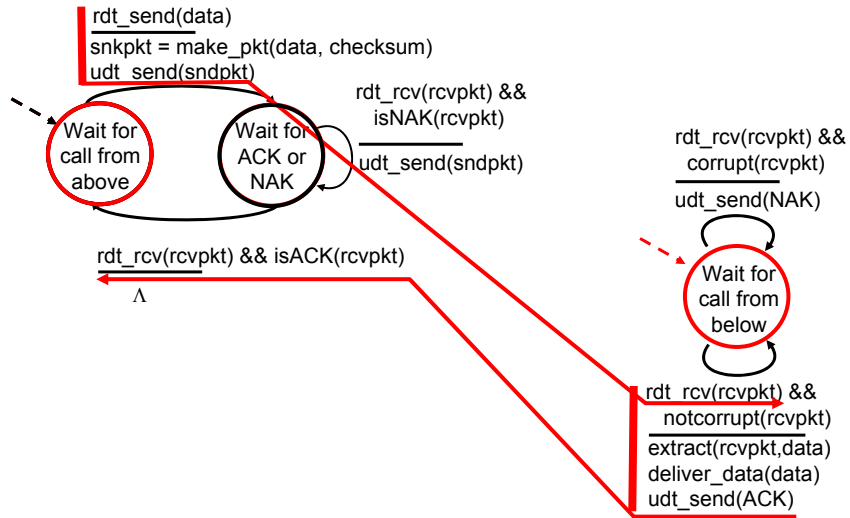
- Underlying channel may flip bits in packet
 - Recall: UDP checksum to detect bit errors
- *The question: how to recover from errors:*
 - *Acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
 - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
 - Sender retransmits pkt on receipt of NAK
 - Human scenarios using ACKs, NAKs?

New mechanisms in **rdt2.0** (beyond **rdt1.0**):

- Error detection
- Receiver feedback: control msgs (ACK,NAK) rcvr->sender

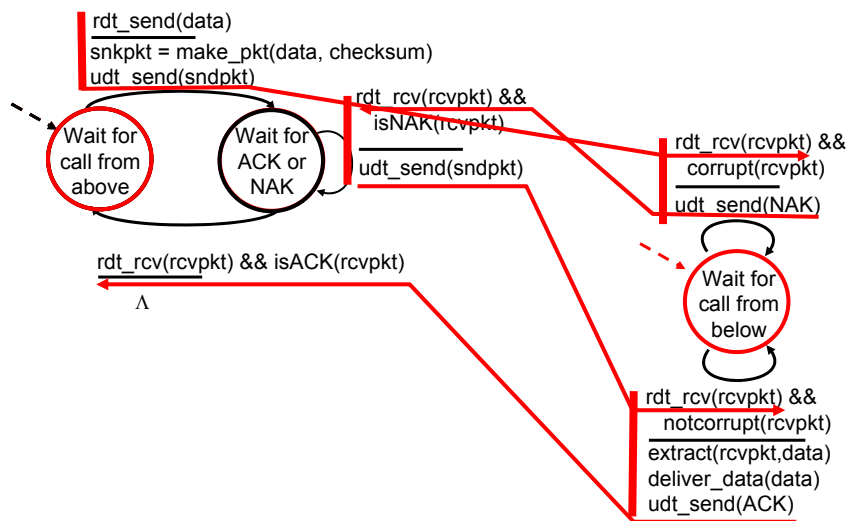
24

Rdt2.0: Operation with no errors



25

Rdt2.0: Error scenario



26

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

What to do?

- Sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- Retransmit, but this might cause retransmission of correctly received pkt!

Handling duplicates:

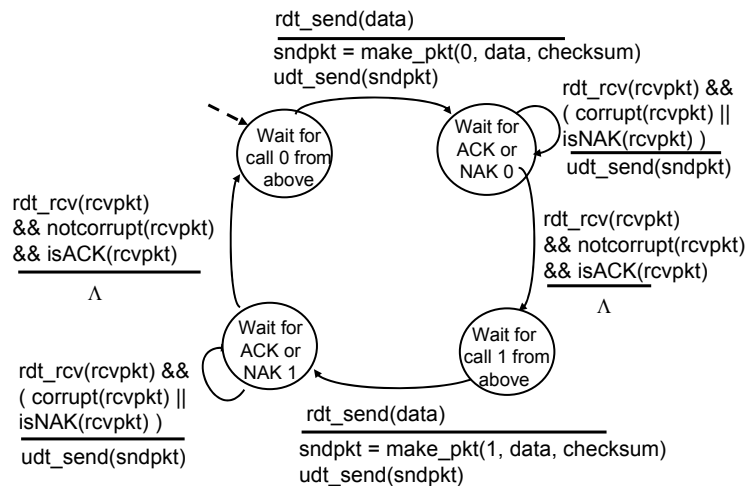
- Sender retransmits current pkt if ACK/NAK garbled
- Sender adds *sequence number* to each pkt
- Receiver discards (doesn't deliver up) duplicate pkt

stop and wait

Sender sends one packet, then waits for receiver response

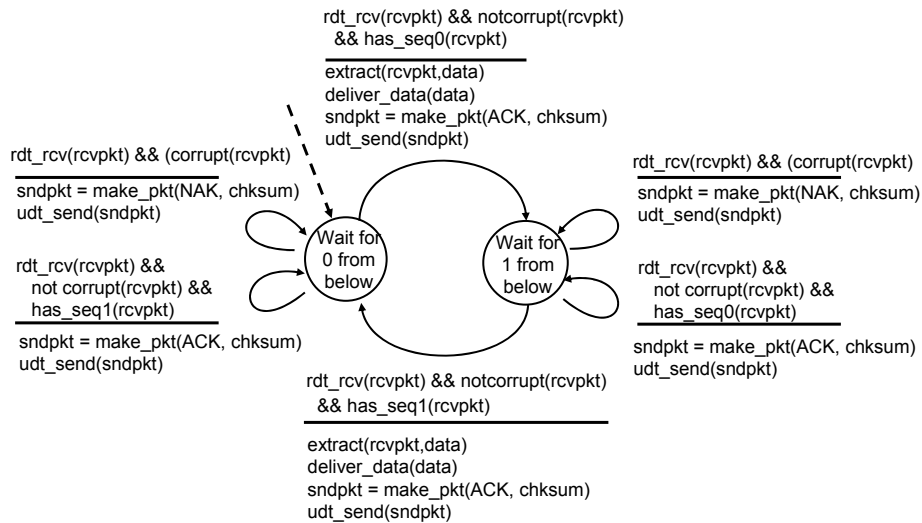
27

Rdt2.1: Sender with garbled ACK/NAKs



28

Rdt2.1: Receiver with garbled ACK/NAKs



29

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

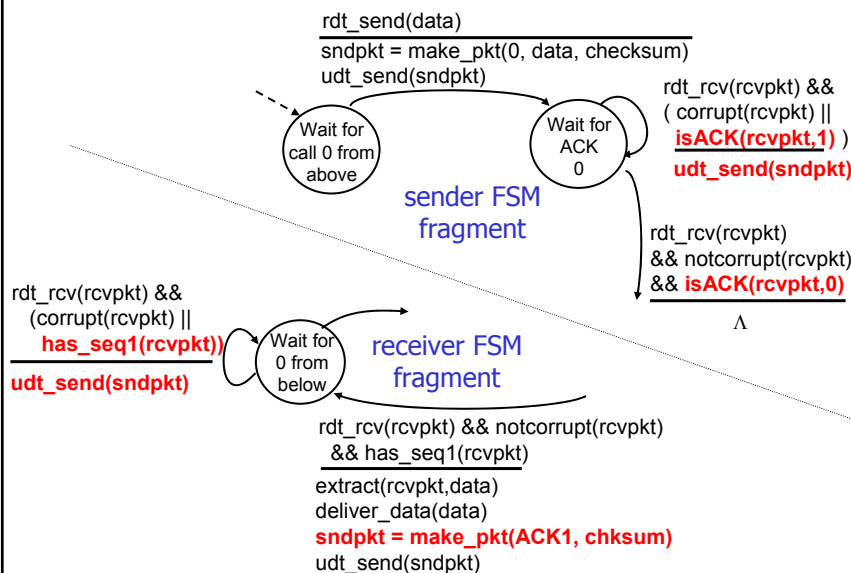
30

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*

31

Rdt2.2: Sender, receiver fragments



32

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

Q: How to deal with loss?

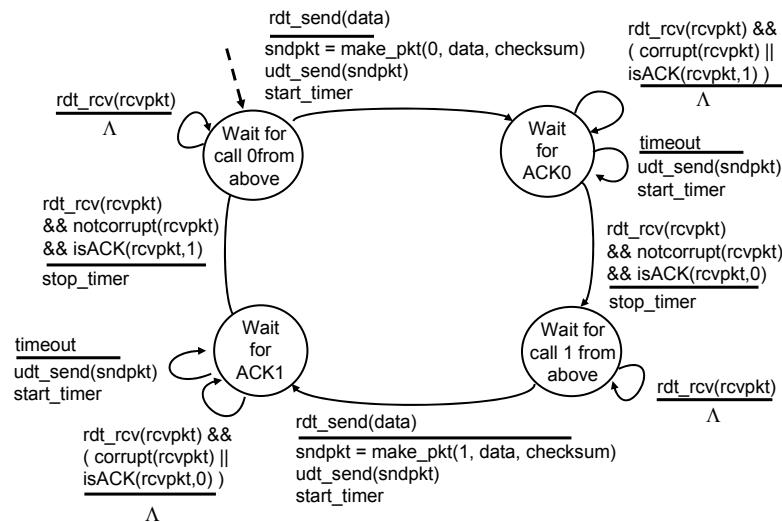
- Sender waits until certain data or ACK lost, then retransmits

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

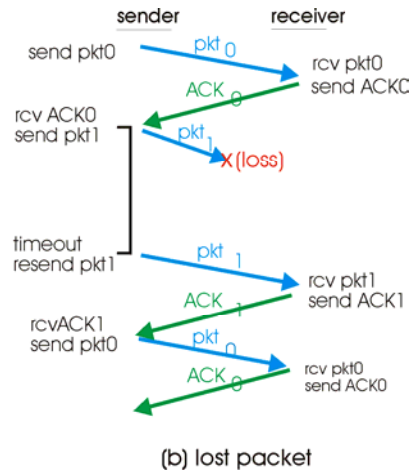
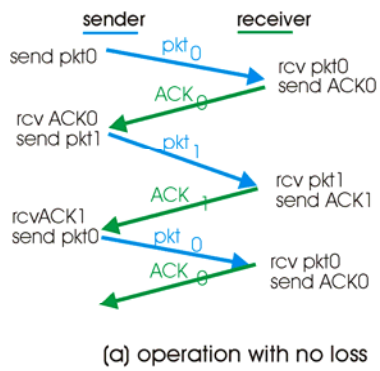
33

Rdt3.0 sender



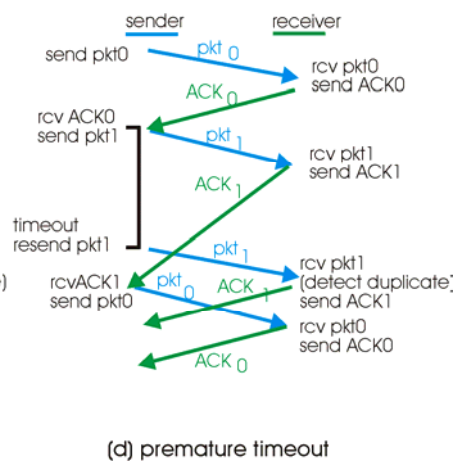
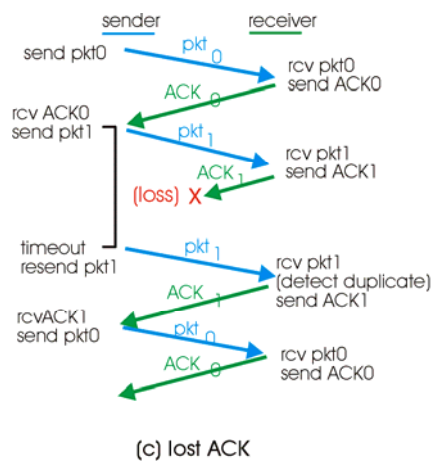
34

Rdt3.0 in action



35

Rdt3.0 in action (cont.)



36

Performance of rdt3.0

- Rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

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

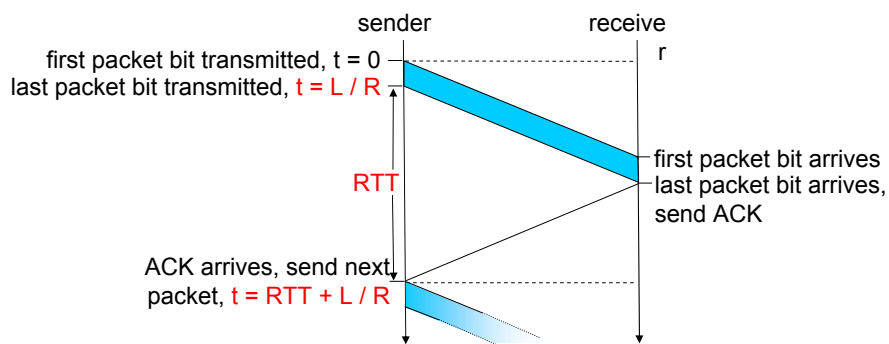
- U_{sender} : **utilization** – fraction of time sender busy sending

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008ms}{30.008ms} = 0.00027$$

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

37

Rdt3.0: Stop-and-wait operation



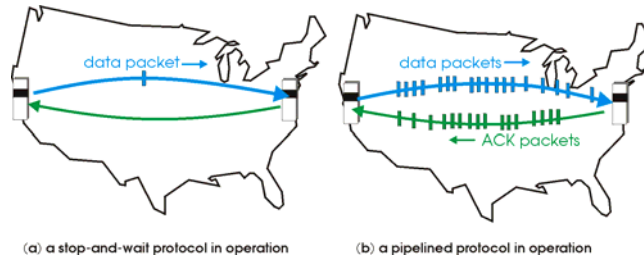
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008ms}{30.008ms} = 0.00027$$

38

Pipelined protocols

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

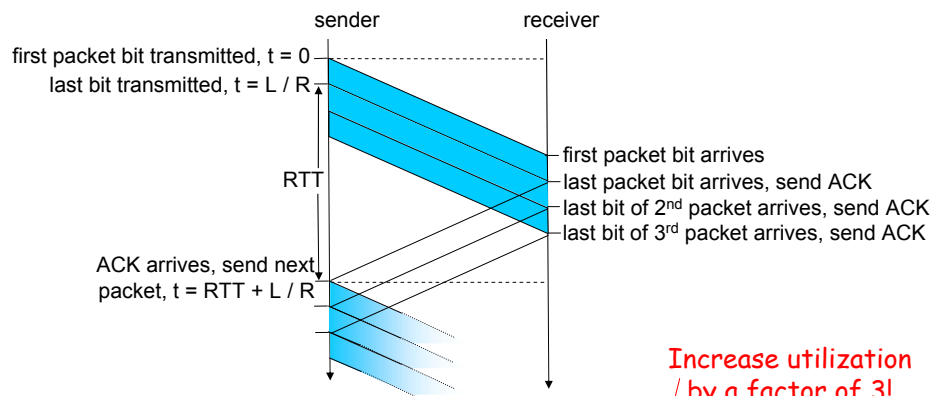
- Range of sequence numbers must be increased
- Buffering at sender and/or receiver



- Two generic forms of pipelined protocols: *go-Back-N*, *selective repeat*

39

Pipelining: Increased utilization



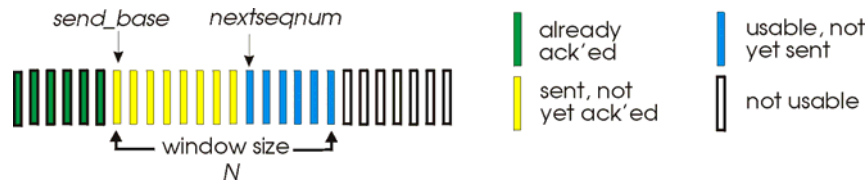
$$U_{\text{sender}} = \frac{3 \cdot L/R}{RTT + L/R} = \frac{0.024\text{ms}}{30.008\text{ms}} = 0.00079$$

40

Go-Back-N

Sender:

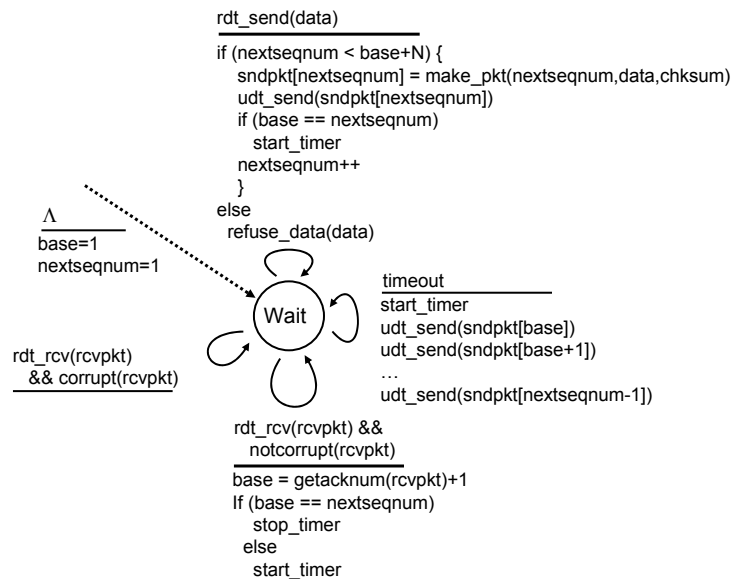
- "Window" of up to N, consecutive unack'ed pkts allowed



- Why limit to N? → later (flow control, congestion control)
- 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
- K-bit seq # in pkt header ($\Rightarrow N < 2^K$)

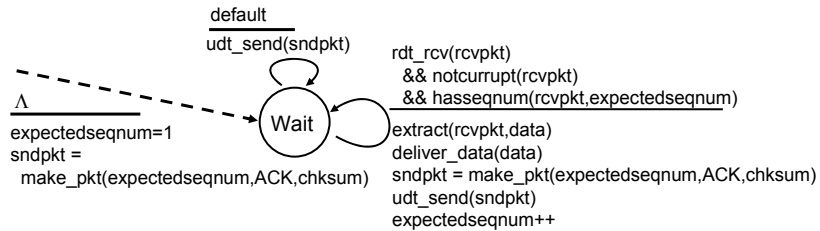
41

GBN: Sender extended FSM



42

GBN: Receiver extended FSM

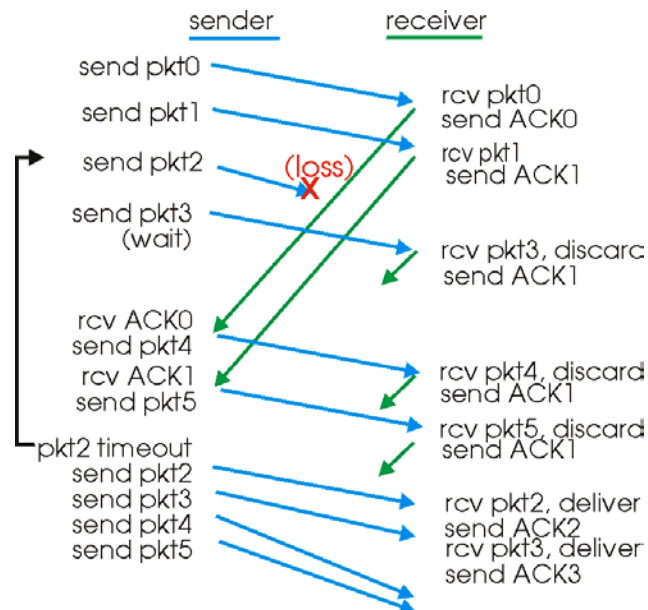


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #

43

GBN in action



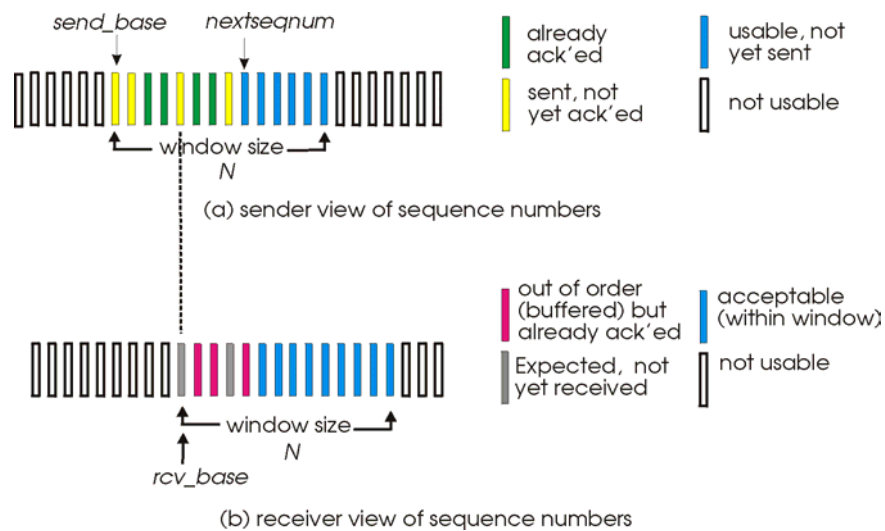
44

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
- ⇒ No need to retransmit correctly received, but out-of-order packets
- Sender window
 - N consecutive seq #'s
 - Again limits seq #'s of sent, unACKed pkts

45

Selective repeat: Sender, receiver windows



46

Selective repeat

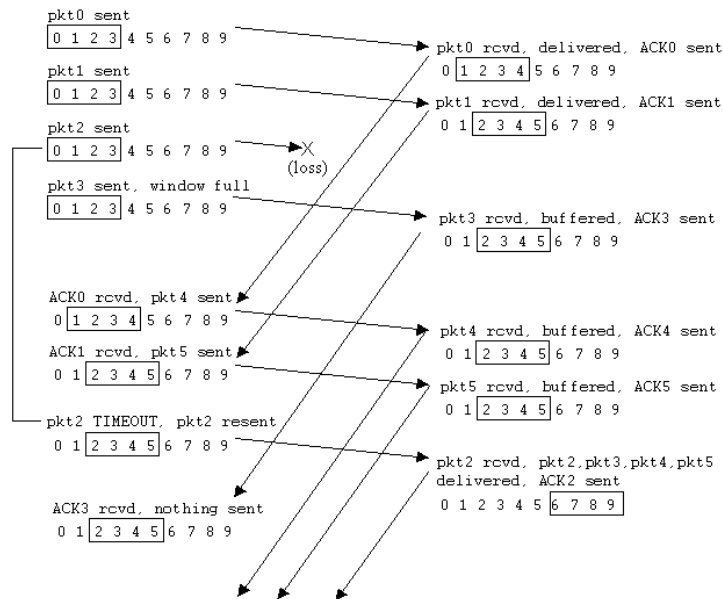
Sender

- 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
 - Cancel timer for n
 - 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?

