

Transport Layer

Chapter 3: Transport Layer

Our goals:

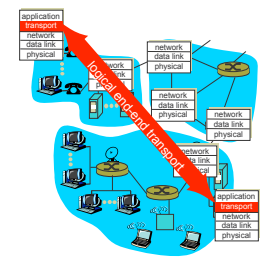
- understand principles behind transport layer services:
 - multiplexing/demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- learn about transport layer protocols in the Internet:
 - UDP: connectionless transport
 - TCP: connection-oriented transport
 - TCP congestion control

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



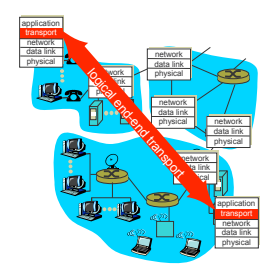
Transport vs. network layer

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

 - processes = people
 - app messages = letters in envelopes
 - hosts = houses
 - transport protocol = sorting and collecting mail within house
 - 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

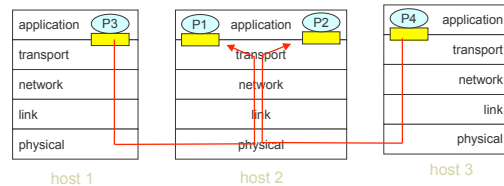
7 I

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)

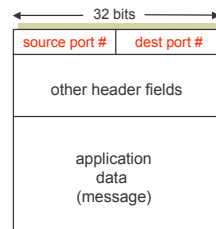
■ = socket ○ = process



8 I

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

9 I

Connectionless demultiplexing

- Create sockets with port numbers:

```
sin1.sin_port = 1234;
bind(socket1, &sin1, ...);
sin2.sin_port = 1235;
bind(socket2, &sin2, ...);
```

- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:

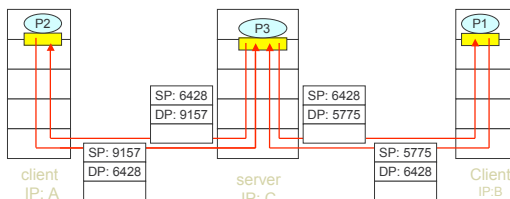
- checks destination port number in segment
- directs UDP segment to socket with that port number

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

10 I

Connectionless demux (cont)

sin.sin_port = 6428; bind(sock, &sin, ...);



SP provides "return address" (returned by recvfrom)

11 I

Connection-oriented demux

- TCP socket identified by 4-tuple:

- source IP address
- source port number
- dest IP address
- dest port number

- rcv host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:

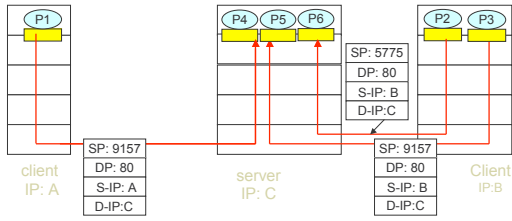
- each socket identified by its own 4-tuple

- Web servers have different sockets for each connecting client

- non-persistent HTTP will have different socket for each request

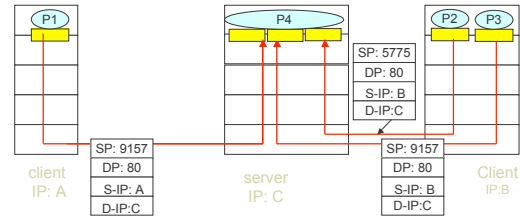
12 I

Connection-oriented demux (cont)



13

Connection-oriented demux: Threaded Web Server



14

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

15

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

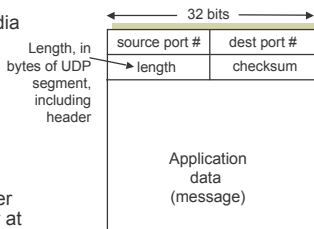
Why is there a UDP?

- 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

16

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!



17

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*

18

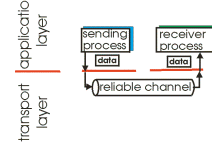
Chapter 3 outline

- 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

19 I

Principles of Reliable data transfer

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



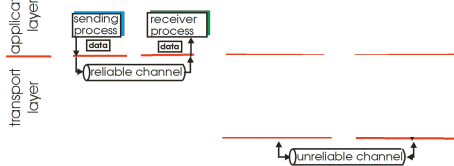
(a) provided service

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

20 I

Principles of Reliable data transfer

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



(a) provided service

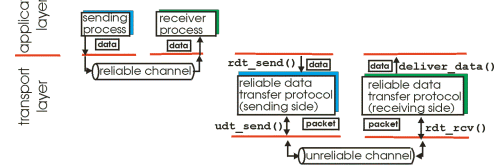
(b) service implementation

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

21 I

Principles of Reliable data transfer

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



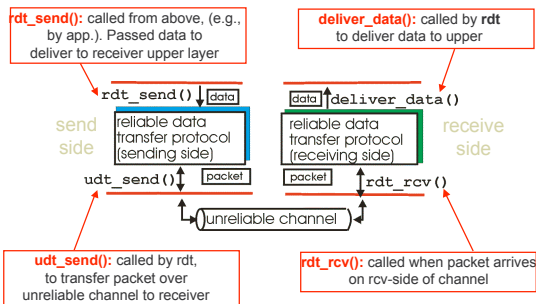
(a) provided service

(b) service implementation

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

22 I

Reliable data transfer: getting started

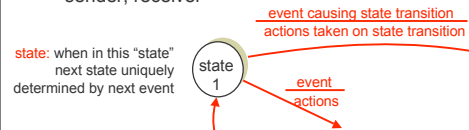


23 I

Reliable data transfer: getting started

We'll:

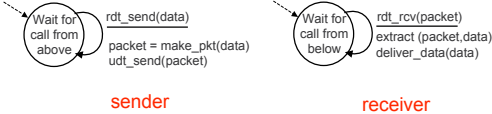
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
 - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver



24 I

Rdt1.0: reliable transfer over a reliable channel

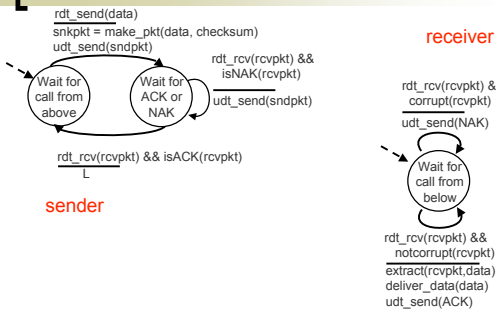
- underlying channel perfectly reliable
 - no bit errors
 - no loss of packets
- separate FSMs for sender, receiver:
 - sender sends data into underlying channel
 - receiver read data from underlying channel



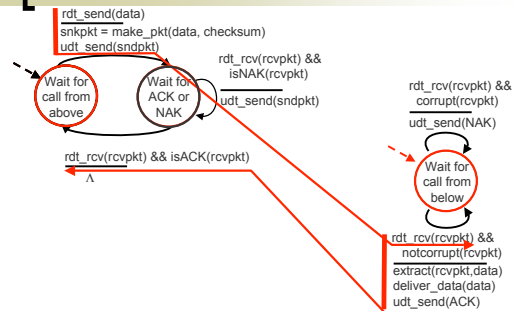
Rdt2.0: channel with bit errors

- 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

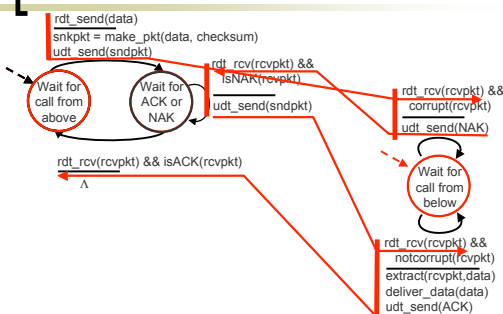
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

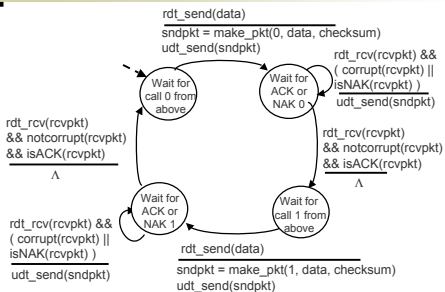
What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

Handling duplicates:

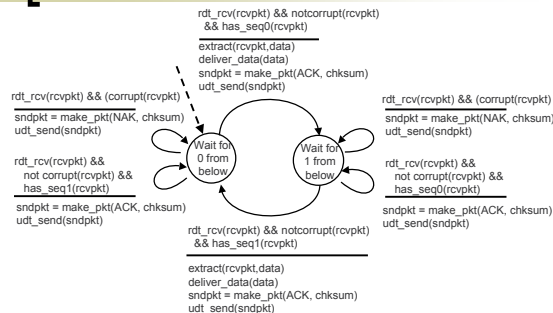
- sender retransmits current pkt if ACK/NAK garbled
 - sender adds **sequence number** to each pkt
 - receiver discards (doesn't deliver up) duplicate pkt
- stop and wait**
 Sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



31

rdt2.1: receiver, handles garbled ACK/NAKs



32

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

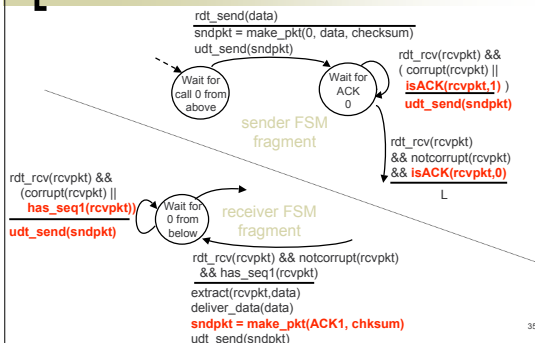
33

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*

34

rdt2.2: sender, receiver fragments



35

rdt3.0: channels with errors and loss

New assumption:

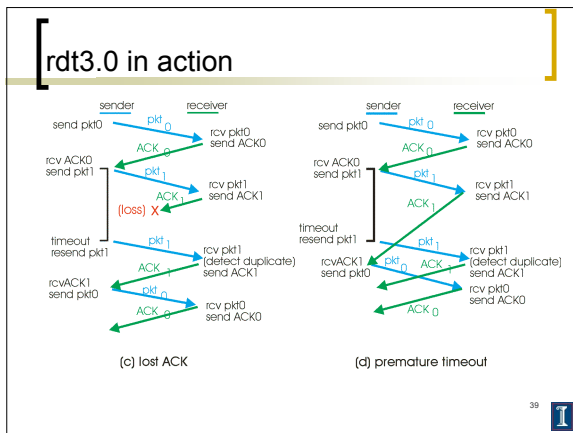
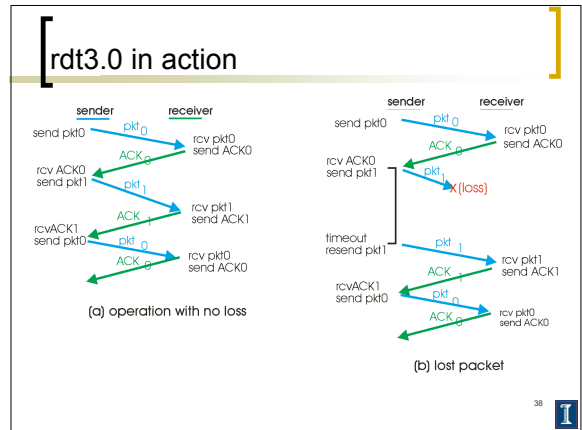
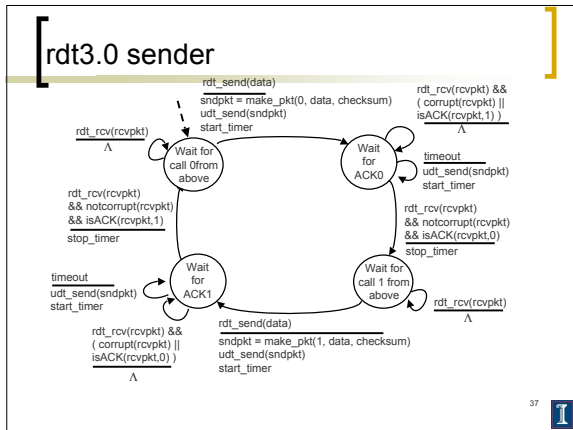
underlying channel can also lose packets (data or ACKs)

- 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

36



Performance of rdt3.0

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

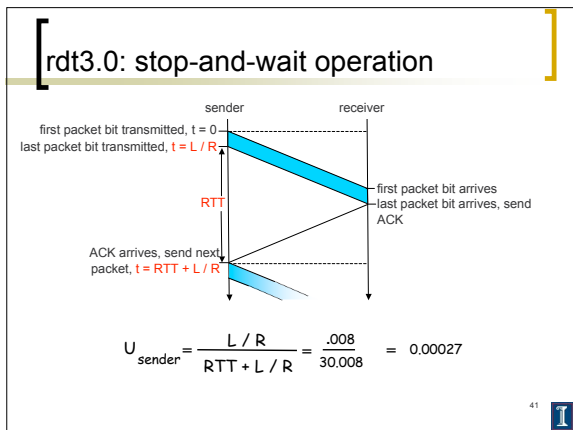
$$T_{\text{transmit}} = \frac{L (\text{packet length in bits})}{R (\text{transmission rate, bps})} = \frac{8\text{kb/pkt}}{10^{**9} \text{ b/sec}} = 8 \text{ microsec}$$

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

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

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

40



Pipelined protocols

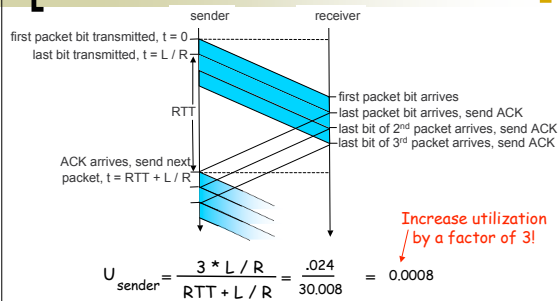
Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

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

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

42

Pipelining: increased utilization



43

Go-Back-N

Sender:

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



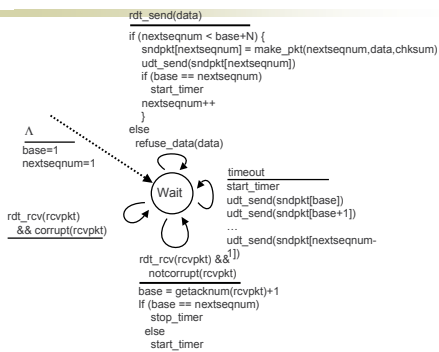
ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
may receive duplicate ACKs (see receiver)

timer for each in-flight pkt

timeout(n): retransmit pkt n and all higher seq # pkts in window

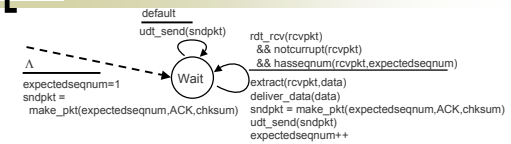
44

GBN: sender extended FSM



45

GBN: receiver extended FSM



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

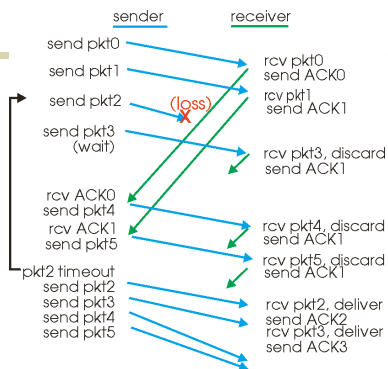
- may generate duplicate ACKs
- need only remember **expectedseqnum**

out-of-order pkt:

- discard (don't buffer) -> **no receiver buffering!**
- Re-ACK pkt with highest in-order seq #

46

GBN in action



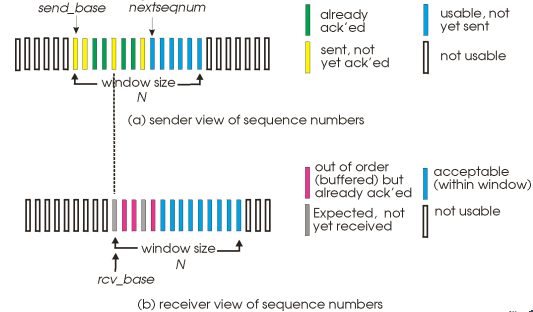
47

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

48

Selective repeat: sender, receiver windows



49

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]

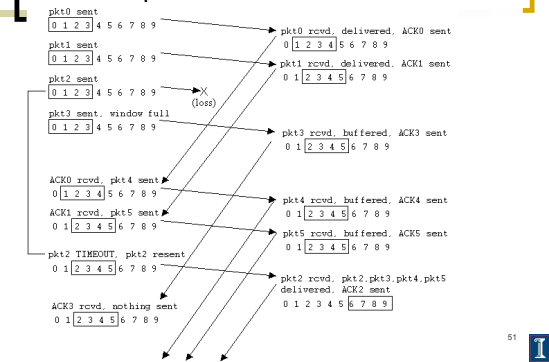
ACK(n)

otherwise:

ignore

50

Selective repeat in action



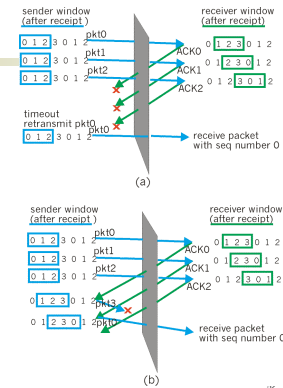
Selective repeat: dilemma

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?



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

53

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:**
 - one sender, one receiver
- reliable, in-order byte stream:**
 - no "message boundaries"
- pipelined:**
 - TCP congestion and flow control set window size
- send & receive buffers**
- full duplex data:**
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- connection-oriented:**
 - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:**
 - sender will not overwhelm receiver



