Electrical Engineering E6761
Computer Communication Networks

Lecture 3
Transport Layer Services:
reliability, connection setup, flow control

Professor Dan Rubenstein
Tues 4:10-6:40, Mudd 1127
Course URL:
http://www.cs.columbia.edu/~danr/EE6761

Today

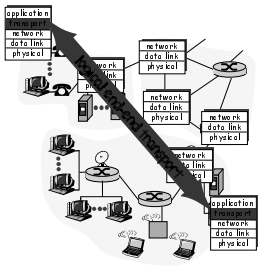
- PA#2 – due 10/3
- HW#0 – solutions on-line (see week2 on materials pg)
- HW#1 – changes, questions?
- Java for PA#2: Yes – all Java-related questions to Vasillis, please
- Transport Layer
  - e2e argument
  - multiplexing / demultiplexing
  - reliability
  - connection setup / teardown
  - flow control
  - Example protocol: TCP
  - congestion control… next time

Policy Refresh…

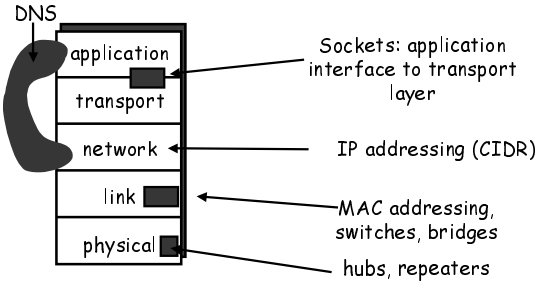
- Collaboration is O.K. on
  - Homework
  - Programming Assignments
  - Project
- How much help should you get?
  - so that next time, you could do similar types of problems on your own
- How much help should you give?
  - enough to get the person moving again
- Who should you help?
  - anybody who asks you

Transport services and protocols

- provide *logical communication* between apps' processes running on different hosts
- transport protocols
  - run in end systems
  - transfer info between **processes**
- *runs on top of the network layer, which :*
  - transfers info between network components
  - can delay, reorder or drop packets



What we've seen so far (layered perspective)...



Today: part1 of transport layer details

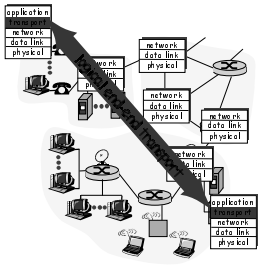
Transport Layer "view" of the network

- A "pipe" connects every pair of hosts
- packets sent into the pipe might
  - come out the other end quickly
  - come out the other end eventually
  - disappear

Transport-layer protocols

Internet transport services:

- reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup/teardown
- unreliable ("best-effort"), unordered unicast or multicast delivery: UDP
- services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast



e2e argument [see Saltzer, Reed, Clark article]

- Philosophy behind Intenet design: Move complex operations to the edges of the network
  - Why? Not all apps may require complex ops, e.g.,
    - reliability (audio, video)
    - security
  - Also, some functionality difficult to implement in network core
    - duplicates suppression
    - FIFO ordering
  - ops often repeated at edge anyways as safety check
- Implications of e2e argument to the Internet: most complex ops should be performed toward the top of the protocol stack

e2e argument pros and cons

- Pros:
  - reduces network complexity - eases deployment, network recovery
  - reduces redundant checks since app often provides checks anyways
  - bugs harder to fix
- Cons:
  - network less efficient (e.g., hop-to-hop reliability would reduce b/w reqmts and delivery delays)
  - more responsibility lies with the application
    - longer development cycle, frequent bugs

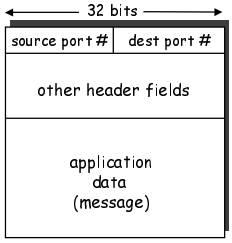
9

Multiplexing/demultiplexing

Multiplexing: gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

multiplexing/demultiplexing in the Internet:

- based on sender, receiver port numbers, IP addresses
  - source, dest port #s in each segment
  - recall: well-known port numbers for specific applications



TCP/UDP segment format

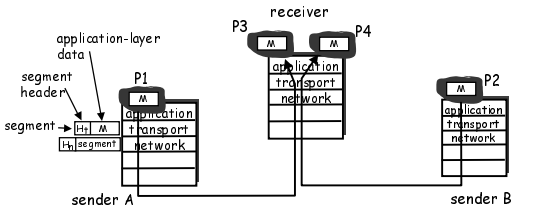
11

Multiplexing/demultiplexing

How hosts handle more than one session simultaneously

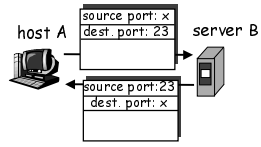
segment - unit of data exchanged between transport layer entities

Demultiplexing: delivering received segments to correct app layer processes

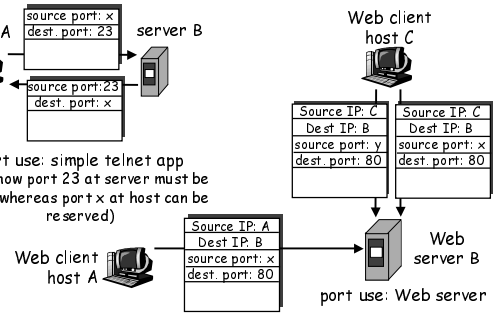


10

Multiplexing/demultiplexing: examples



port use: simple telnet app
(Note how port 23 at server must be shared whereas port x at host can be reserved)



port use: Web server

Q: how does the server know which packets go with which process?

12

UDP: User Datagram Protocol [RFC 768]

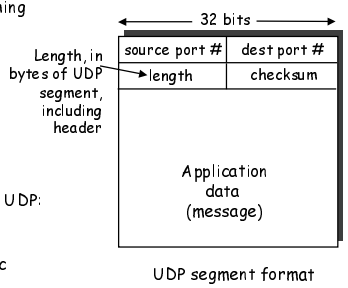
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

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

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 recovery!



UDP segment format

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

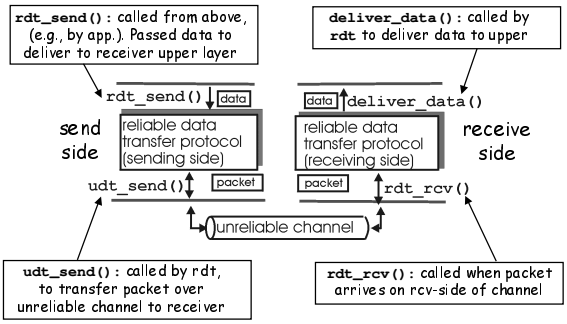
Sender:

- 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 ….

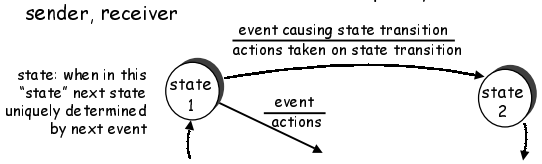
Reliable data transfer: getting started



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 in both directions!
- use finite state machines (FSM) to specify sender, receiver

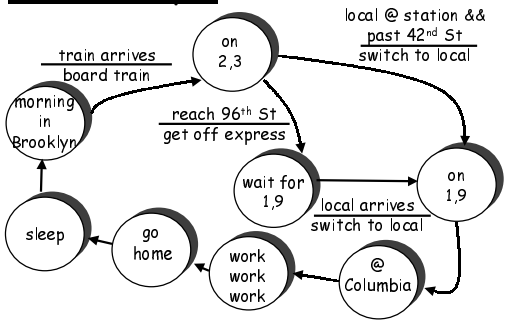


Rdt1.0: reliable transfer over a reliable channel

- 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

(a) rdt1.0: sending side (b) rdt1.0: receiving side

FSM example

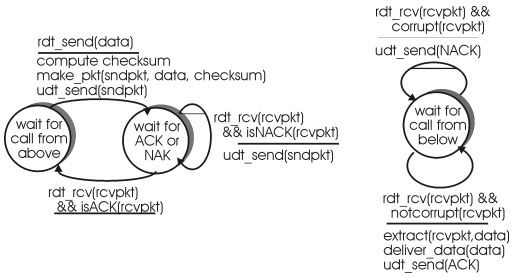


A day in the life of Prof. Rubenstein

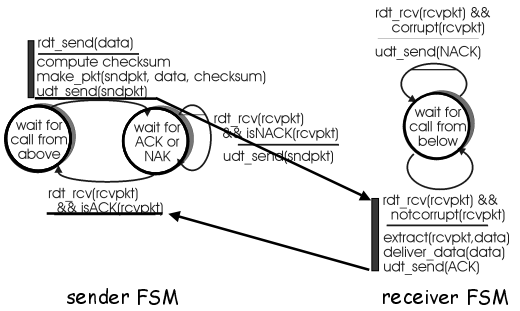
Rdt2.0: channel with bit errors

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

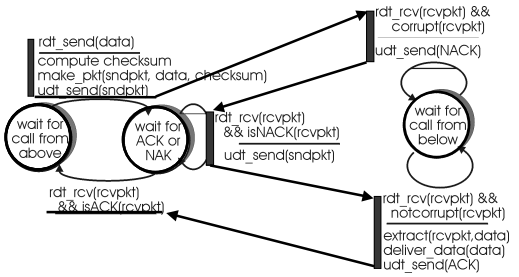
rdt2.0: FSM specification



rdt2.0: in action (no errors)



rdt2.0: in action (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

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!

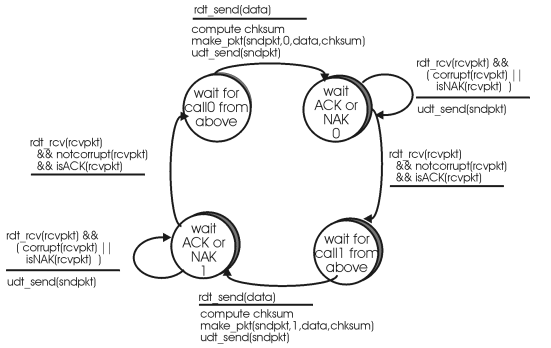
Handling duplicates:

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

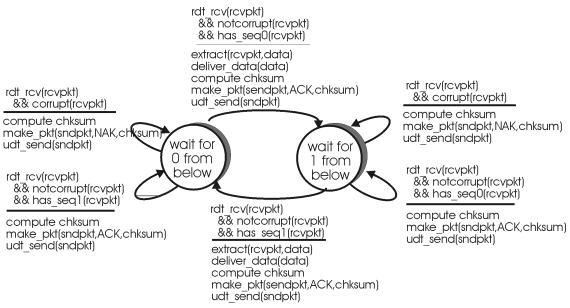
stop and wait

Sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



Note: sender & rcvr must agree on initial seqno

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. #

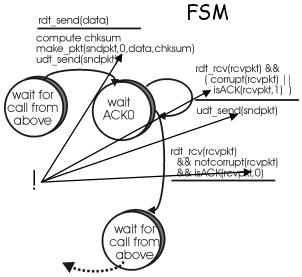
Receiver:

- 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

rdt2.2: 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*

sender FSM



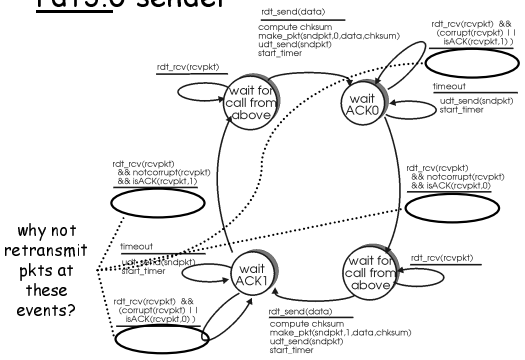
rdt3.0: channels with errors *and* loss

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

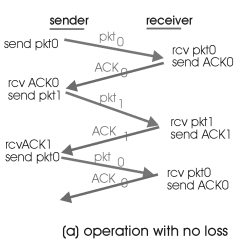
Q: how to deal with loss?
- one possibility: sender waits until certain data or ACK definitely lost, then retransmits
- drawbacks?

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

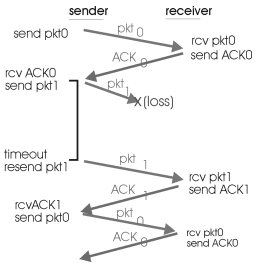
rdt3.0 sender



rdt3.0 in action

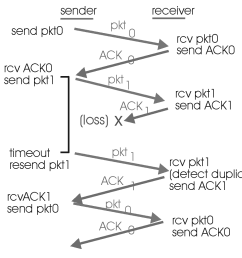


(a) operation with no loss

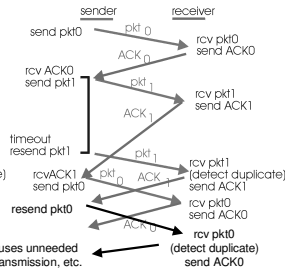


(b) lost packet

rdt3.0 in action



(c) lost ACK

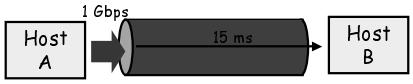


(d) premature timeout

Transmitting on those "other" events (2 slides ago) would have caused cascade of redundant retransmissions

Performance of rdt3.0

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



$$T_{transmit} = \frac{8kb/pkt}{10**9\ b/sec} = 8\ microsec$$

$$Utilization = U = \frac{fraction\ of\ time}{sender\ busy\ sending} = \frac{8\ microsec}{30.016\ msec} = 0.00015$$

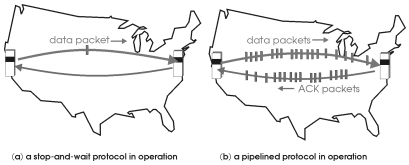
  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

33

Pipelined protocols

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

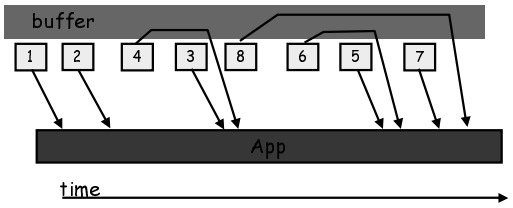
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver



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

34

In-order buffering



- Transport layer maintains a per-session buffer
  - pkts possibly placed in buffer out of order (e.g., due to network loss)
  - pkts are sent up to app (and then removed from buffer) in order

35

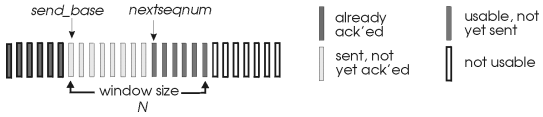
Go-Back-N

Sender:

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

Rcvr:

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



More Sender:

- timer for each in-flight pkt
- *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

36

GBN: receiver extended FSM

sender

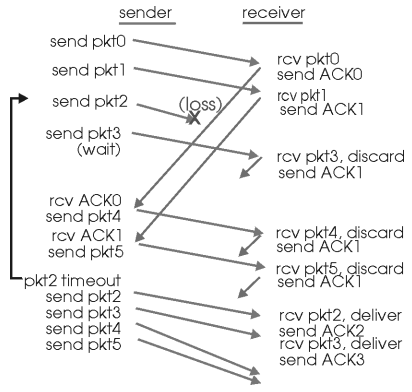
- have N pkts "in transit"
- roll window past largest ACK
- on timeout of lowest seqno packet in window, retransmit current window (and reset timers)

receiver simple:

- 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!
  - ACK pkt with highest in-order seq #

GBN in action

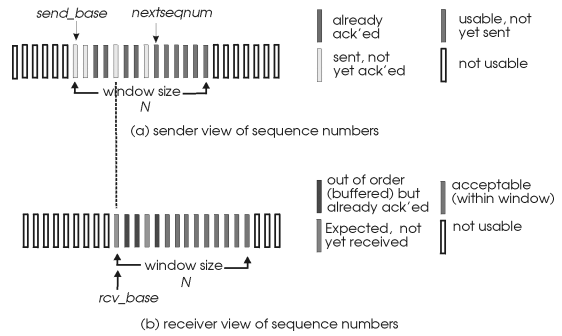
Here, N=3



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 maintains 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 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

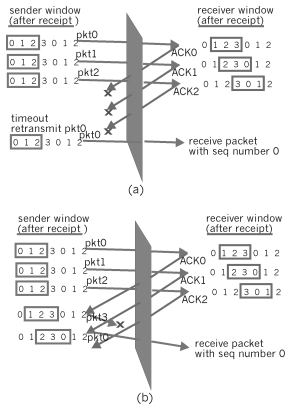
Selective repeat: dilemma

Example:

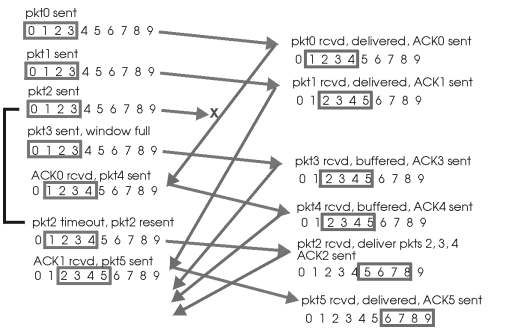
- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

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



Selective repeat in action



Go-back-N vs. Selective Repeat

- Q: How do bandwidth requirements compare?
- Let's do a simple analytical comparison
- Model:
  - any packet transmission lost with probability, p
  - ACKs never lost
  - selective repeat:
    - sender knows exactly what rcvr needs
  - Go-back-N
    - each round, sender transmits block of N pkts
    - rcvr informs sender of 1st lost pkt
    - sender sends N pkts starting at 1st point of loss
    - rcvr dumps any pkts in window after a loss

Selective Repeat Analysis

- Each pkt can be "examined" in isolation
- T_{SR} = # of transmissions of a pkt
- $P(T_{SR} > i) = p^i$
- $E[T_{SR}] = P(T_{SR}=1) + 2\,P(T_{SR}=2) + 3P(T_{SR}=3) + \ldots$
  $= P(T_{SR} > 0) + P(T_{SR} > 1) + P(T_{SR} > 2) + P(T_{SR} > 3) + \ldots$
  $= 1 / (1-p)$
- e.g., p = .2, then $E[T_{SR}] = 5$

Go-Back-N analysis

- S_N = # pkts arriving prior to loss in window of N
- $P(S_N > i) = (1-p)^{i+1}$, $0 \le i < N$, $= 0$ for $i \ge N$
- $E[S_N] = P(S_N > 0) + P(S_N > 1) + \ldots + P(S_N > N-1)$
  $= (1 - p - (1-p)^{N+1}) / p$
- Let $S_{N,j}$ = # of pkts accepted in the j^{th} transmission
- $E[T_{GBN}]$ = avg. # of transmissions of pkt

$$= \frac{\sum_{j=1}^{m} N}{\sum_{j=1}^{m} (S_{N,j})} \qquad = \frac{(\sum_{j=1}^{m} N) / m}{(\sum_{j=1}^{m} S_{N,j}) \ / m}$$

Go-back-N analysis (cont'd)

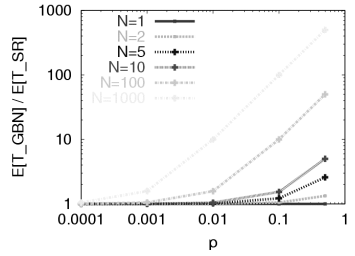
- as $m \to \infty$ $= N / E[S_N]$

$$E[T_{GBN}] = \frac{Np}{1 - p - (1-p)^{N+1}}$$

- How does $E[T_{SR}]$ compare with $E[T_{GBN}]$

Go-Back-N vs. Selective Repeat

- Using our analysis, for various N, p: how much more efficient is Selective Repeat vs. Go-Back-N?



TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*

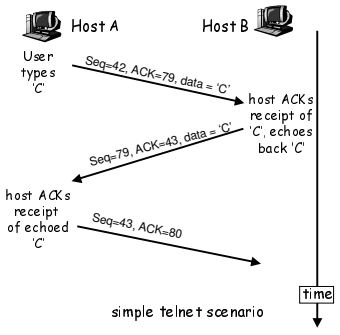
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP seq. #'s and ACKs

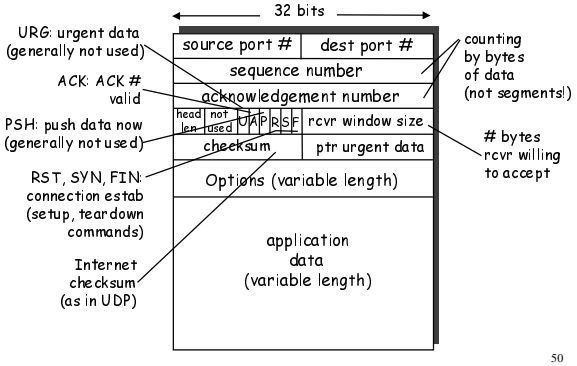
Seq. #'s:
- byte stream "number" of first byte in segment's data

ACKs:
- seq # of next byte expected from other side
- cumulative ACK

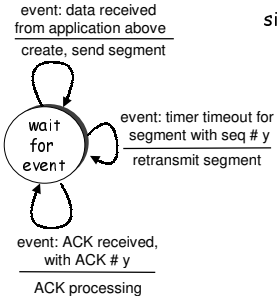
Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor



TCP segment structure



TCP: reliable data transfer



simplified sender, assuming
- one way data transfer
- no flow, congestion control

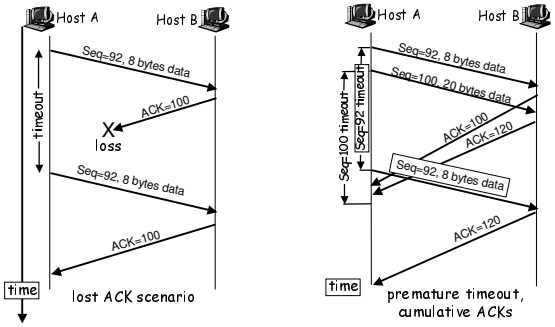
TCP: reliable data transfer

Simplified TCP sender

Similar to GBN, but some slight differences…

```
00  sendbase = initial_sequence number
01  nextseqnum = initial_sequence number
02
03   loop (forever) {
04    switch(event)
05    event: data received from application above
06         create TCP segment with sequence number nextseqnum
07         start timer for segment nextseqnum
08         pass segment to IP
09         nextseqnum = nextseqnum + length(data)
10    event: timer timeout for segment with sequence number y
11         retransmit segment with sequence number y
12         compute new timeout interval for segment y
13         restart timer for sequence number y
14    event: ACK received, with ACK field value of y
15         if (y > sendbase) { /* cumulative ACK of all data up to y */
16            cancel all timers for segments with sequence numbers < y
17             sendbase = y
18             }
19         else { /* a duplicate ACK for already ACKed segment */
20            increment number of duplicate ACKs received for y
21            if (number of duplicate ACKS received for y == 3) {
22               /* TCP fast retransmit */
23              resend segment with sequence number y
24              restart timer for segment y
25            }
26    } /* end of loop forever */
```


TCP: retransmission scenarios



TCP ACK generation [RFC 1122, RFC 2581]

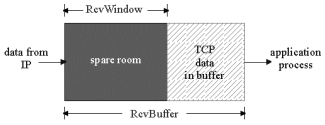
Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP Flow Control

flow control: sender won't overrun receiver's buffers by transmitting too much, too fast

`RcvBuffer` = size or TCP Receive Buffer

`RcvWindow` = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- `RcvWindow` field in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received `RcvWindow`

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - note: RTT will vary
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
- **SampleRTT** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

EstimatedRTT = (1−x)*EstimatedRTT + x*SampleRTT

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1

Setting the timeout

- **EstimtedRTT** plus "safety margin"
- large variation in **EstimatedRTT** -> larger safety margin

Timeout = EstimatedRTT + 4*Deviation

Deviation = (1−x)*Deviation + x*|SampleRTT−EstimatedRTT|

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **RcvWindow**)
- *client:* connection initiator

```
Socket clientSocket = new
Socket("hostname","port
number");
```

- *server:* contacted by client

```
Socket connectionSocket =
welcomeSocket.accept();
```

Three way handshake:

Step 1: client end system sends TCP SYN control segment to server

- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment

- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

Initial seqnos in both directions chosen randomly. Why?

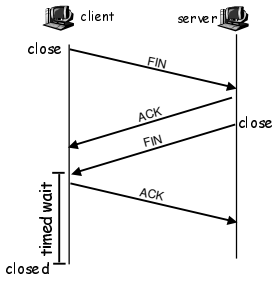
TCP Connection Management (cont.)

Closing a connection:

client closes socket:
clientSocket.close();

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



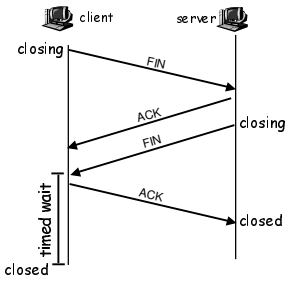
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

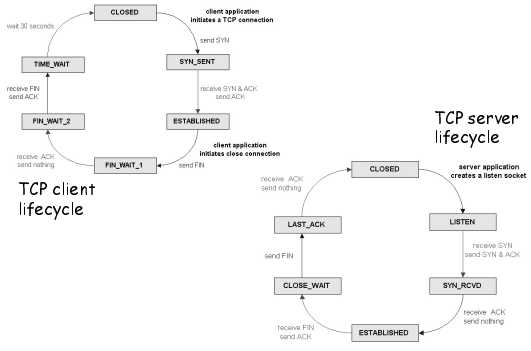
Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



61

TCP Connection Management (cont)



62