CMPE 150: Introduction to Computer Networks

Katia Obraczka
Computer Engineering
UCSC Baskin Engineering
Lecture 11
Announcements/Reminders

- **Midterm exam on 02.19.**
  - Closed books, closed notes.
  - Bring a picture id!
  - Covers material up and including Thursday’s (02.11) lecture.

- **No lab on 02/18 (President’s Day).**
  - Need to join another lab that week.
  - Friday lab as make-up.
Last Class

- Revisited reliable data transfer.
  - rdt protocols.
  - **Mechanisms**
    - Feedback.
    - Sequence numbers.
    - Checksum.
    - Retransmission timers.
  - **Retransmission timers.**
    - Trade-offs.
    - How to set retransmission timer.

- **ARQ protocols.**
Today

- ARQ protocols (cont’d).
- TCP.
Stop-and-wait operation (rdt 3.0)

\[ U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30.008} = 0.00027 \]

Utilization is very low!
How can we improve performance?
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
ARQ protocols

- Stop-and-wait
- Pipelined protocols
  - Go-back-N
  - Selective repeat
Pipelining: increased utilization

\[ U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008 \]

Increase utilization by a factor of 3!
Pipelining Protocols

**Go-back-N: overview**
- **sender**: up to N unACKed pkts in pipeline
- **receiver**: only sends cumulative ACKs
  - doesn’t ACK pkt if there’s a gap
- **sender**: has timer for oldest unACKed pkt
  - if timer expires: retransmit all unACKed packets

**Selective Repeat: overview**
- **sender**: up to N unACKed packets in pipeline
- **receiver**: ACKs individual pkts
- **sender**: maintains timer for each unACKed pkt
  - if timer expires: retransmit only unACKed packet
Go-Back-N

Sender:
- k-bit seq # in pkt header
- “window” of up to N, consecutive unACKed 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

CMPE 150 Winter13
GBN in action

sender
- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)
- rcv ACK0
- send pkt4
- rcv ACK1
- send pkt5
- pkt2 timeout
- send pkt2
- send pkt3
- send pkt4
- send pkt5

receiver
- rcv pkt0
- send ACK0
- rcv pkt1
- send ACK1
- rcv pkt3, discard
- send ACK1
- rcv pkt4, discard
- send ACK1
- rcv pkt5, discard
- send ACK1
- rcv pkt2, deliver
- send ACK2
- rcv pkt3, deliver
- send ACK3

CMPE 150 Winter13
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
Selective Repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers

CMPE 150 Winter13
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
TCP: Connection-Oriented Transport
TCP: Overview

- **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

RFCs: 793, 1122, 1323, 2018, 2581

CMPE 150 Winter13
**TCP segment structure**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>head len</td>
<td>not used</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>Options</td>
<td>Options (variable length)</td>
</tr>
<tr>
<td>application data</td>
<td>Application data (variable length)</td>
</tr>
</tbody>
</table>

**Fields: URG, ACK, PSH, RST, SYN, FIN**
- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum**: (as in UDP)

**Other notes:**
- Counting by bytes of data (not segments!)
- # bytes rcvr willing to accept
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 host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

**Step 2:** server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data
TCP Connection Establishment

Three way handshake:

**Step 1:** client host sends TCP SYN segment to server
- specifies initial seq #
- no data

**Step 2:** server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data
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.
TCP Connection Management (cont)

Typical TCP client lifecycle

- CLOSED
  - wait 30 seconds
  - receive FIN, send ACK
- TIME_WAIT
  - receive ACK, send nothing
- FIN_WAIT_2
  - receive FIN, send ACK
- FIN_WAIT_1
  - client application initiates close connection
- ESTABLISHED
  - send FIN
  - receive ACK, send nothing

Typical TCP server lifecycle

- CLOSED
  - server application creates a listen socket
- LISTEN
  - receive SYN, send SYN & ACK
- SYN_RCVD
  - receive ACK, send nothing
- CLOSE_WAIT
  - receive FIN, send ACK
- LAST_ACK
  - receive FIN, send ACK
- ESTABLISHED
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 implementer

Host A
User types ‘C’
Seq=42, ACK=79, data = ‘C’
Seq=79, ACK=43, data = ‘C’
Seq=43, ACK=80

Host B
host ACKs receipt of ‘C’, echoes back ‘C’

simple telnet scenario

CMPE 150 Winter13
**TCP Round Trip Time and Timeout**

**Q:** how to set TCP timeout value?
- longer than RTT  
  - but RTT varies
- 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
- SampleRTT will vary, want estimated RTT “smoother”  
  - average several recent measurements, not just current SampleRTT
TCP Round Trip Time and Timeout

Estimated\(\text{RTT} = (1- \alpha)\)\(\times\)Estimated\(\text{RTT} + \alpha\)\times\)Sample\(\text{RTT}\)

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: \(\alpha = 0.125\)
Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>SampleRTT</th>
<th>Estimated RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>150</td>
</tr>
<tr>
<td>15</td>
<td>200</td>
<td>250</td>
</tr>
<tr>
<td>22</td>
<td>250</td>
<td>300</td>
</tr>
<tr>
<td>29</td>
<td>300</td>
<td>350</td>
</tr>
<tr>
<td>36</td>
<td>350</td>
<td>400</td>
</tr>
<tr>
<td>43</td>
<td>400</td>
<td>450</td>
</tr>
<tr>
<td>50</td>
<td>450</td>
<td>500</td>
</tr>
<tr>
<td>57</td>
<td>500</td>
<td>550</td>
</tr>
<tr>
<td>64</td>
<td>550</td>
<td>600</td>
</tr>
<tr>
<td>71</td>
<td>600</td>
<td>650</td>
</tr>
<tr>
<td>78</td>
<td>650</td>
<td>700</td>
</tr>
<tr>
<td>85</td>
<td>700</td>
<td>750</td>
</tr>
<tr>
<td>92</td>
<td>750</td>
<td>800</td>
</tr>
<tr>
<td>99</td>
<td>800</td>
<td>850</td>
</tr>
<tr>
<td>106</td>
<td>850</td>
<td>900</td>
</tr>
</tbody>
</table>

CMPE 150 Winter13
**TCP Round Trip Time and Timeout**

### Setting the timeout

- **EstimatedRTT** plus “safety margin”
  - large variation in EstimatedRTT $\Rightarrow$ larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

$$\text{DevRTT} = (1-\beta) \ast \text{DevRTT} + \beta \ast |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

Then set timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \ast \text{DevRTT}$$
TCP reliable data transfer

- TCP creates rdt service on top of IP’s unreliable service
- pipelined segments
- cumulative ACKs
- TCP uses retransmission timer

- retransmissions are triggered by:
  - timeout events
  - duplicate ACKs

- initially consider simplified TCP sender:
  - ignore duplicate ACKs
  - ignore flow control, congestion control

CMPE 150 Winter13
TCP sender events:

data rcvd from app:
- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unACKed segment)
- expiration interval: TimeOutInterval

timeout:
- retransmit segment that caused timeout
- restart timer

ACK rcvd:
- if acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are outstanding segments
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)

    event: data received from application above
    create TCP segment with sequence number NextSeqNum
    if (timer currently not running)
        start timer
    pass segment to IP
    NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
    retransmit not-yet-acknowledged segment with
    smallest sequence number
    start timer

    event: ACK received, with ACK field value of y
    if (y > SendBase) {
        SendBase = y
        if (there are currently not-yet-acknowledged segments)
            start timer
    }

} /* end of loop forever */
TCP: retransmission scenarios

- **Host A**
  - Seq=100, 20 bytes data
  - Time:
    - Premature timeout
  - ACK=100
  - SendBase = 100

- **Host B**
  - Seq=92, 8 bytes data
  - Seq=92 timeout
  - ACK=100

- **Lost ACK scenario**
  - Seq=92, 8 bytes data
  - ACK=100

- **Premature timeout**
  - Seq=92 timeout
  - ACK=100
  - SendBase = 120
TCP retransmission scenarios (more)

Host A
Seq=92, 8 bytes data
ACK=100

Host B
SendBase = 120

SendBase = 120

Cumulative ACK scenario

Seq=100, 20 bytes data
ACK=100

Seq=92, 8 bytes data

ACK=120

X

loss
## TCP ACK generation  [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>Immediately send <em>duplicate ACK</em>, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
TCP Flow Control

- receive side of TCP connection has a receive buffer:

  - speed-matching service: matching send rate to receiving application's drain rate

- app process may be slow at reading from buffer

- flow control: sender won't overflow receiver's buffer by transmitting too much, too fast
TCP Flow control: how it works

(receiver: advertises unused buffer space by including rwnd value in segment header)

(sender: limits # of unACKed bytes to rwnd)

- guarantees receiver’s buffer doesn’t overflow

(unused buffer space:
  - \( rwnd \)
  - \( RcvBuffer - [LastByteRcvd - LastByteRead] \)

(suppose TCP receiver discards out-of-order segments)

unused buffer space:
= \( rwnd \)
= \( RcvBuffer - [LastByteRcvd - LastByteRead] \)