CS 356: Introduction to Computer Networks

Lecture 19: Transmission Control Protocol (TCP)  
Chap. 5.2

Xiaowei Yang  
xwy@cs.duke.edu
Overview

• TCP
  – Connection management
  – Flow control
  – When to transmit a segment
  – Adaptive retransmission
  – TCP options
    • Modern extensions
Transmission Control Protocol

- Connection-oriented protocol
- Provides a reliable unicast end-to-end byte stream over an unreliable internetwork
TCP performance is critical to business

Impact of site performance on overall site conversion rate.

Baseline – 1 in 2 site visits had response time > 4 seconds
* Sharp decline in conversion rate as average site load time increases from 1 to 4 seconds
* Overall average site load time is lower for the converted population (3.22 Seconds) than the non-converted population (6.03 Seconds)

Connection establishment/tear down

- Active/passive open

- Active/passive close, simultaneous close
Two events trigger transitions:
- Packet arrival
- Application operations

Half-close end may still receive data

Time_WAIT
- Must wait because ACK may be lost
- The other end may retransmit FIN
TCP States in “Normal” Connection Lifetime

SYN_SENT (active open)
SYN (SeqNo = x)
SYN (SeqNo = y, AckNo = x + 1)
(AckNo = y + 1)
SYN_RCVD
ESTABLISHED
FIN_WAIT_1 (active close)
FIN (SeqNo = m)
(AckNo = m + 1)
CLOSE_WAIT (passive close)
FIN_WAIT_2
FIN (SeqNo = n)
(AckNo = n + 1)
LAST_ACK
TIME_WAIT
CLOSED
2MSL Wait State

2MSL = 2 * Maximum Segment Lifetime

2MSL Wait State = TIME_WAIT

- When TCP does an active close, and sends the final ACK, the connection **must** stay in the TIME_WAIT state for **twice the maximum segment lifetime**.
- The socket pair (srcIP, srcPort, dstIP, dstPort) cannot be reused

**Why?**

- To prevent mixing packets from two different incarnations of the same connection
Resetting Connections

• Resetting connections is done by setting the RST flag

• When is the RST flag set?
  – Connection request arrives and no server process is waiting on the destination port
  – Abort a connection causes the receiver to throw away buffered data
  – Receiver does not acknowledge the RST segment
  – Abused in practice to block applications
Flow control
Sliding window revisited

- **Invariants**
  - \( \text{LastByteAacked} \leq \text{LastByteSent} \)
  - \( \text{LastByteSent} \leq \text{LastByteWritten} \)
  - \( \text{LastByteRead} < \text{NextByteExpected} \)
  - \( \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \)

- **Limited sending buffer and Receiving buffer**
Buffer Sizes vs Window Sizes

- Maximum SWS $\leq$ MaxSndBuf
- Maximum RWS $\leq$ MaxRcvBuf $-$ ((NextByteExpected-1) – LastByteRead)
TCP Flow Control

Q: how does a receiver prevent a sender from overrunning its buffer?
A: use AdvertisedWindow
Invariants for flow control

- Receiver side:
  - LastByteRcvd – LastByteRead ≤ MaxRcvBuf
  - AdvertisedWindow = MaxRcvBuf – ((NextByteExpected - 1) – LastByteRead)
Invariants for flow control

- Sender side:
  \[ \text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked}) \]

- \( \text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSndBuf} \)
  - Sender process would be blocked if send buffer is full
Window probes

• What if a receiver advertises a window size of zero?
  – Problem: Receiver can’t send more ACKs as sender stops sending more data

• Design choices
  – Receivers send duplicate ACKs when window opens
  – Sender sends periodic 1 byte probes

• Why?
  – Keeping the receive side simple → Smart sender/dumb receiver
When to send a segment?

• App writes bytes to a TCP socket
• TCP decides when to send a segment

• Design choices when window opens:
  – Send whenever data available
  – Send when collected Maximum Segment Size data ✔

  • Why?
  • More efficient
Push flag

- What if App is interactive, e.g. ssh?
  - App sets the PUSH flag
  - Flush the sent buffer
Silly Window Syndrome

• Now considers flow control
  – Window opens, but does not have MSS bytes

• Design choice 1: send all it has
  • E.g., sender sends 1 byte, receiver acks 1, acks opens the window by 1 byte, sender sends another 1 byte, and so on
Sending smaller segments

1. Send 360-Byte Segment
   - SND.UNA = 1
   - SND.WND = 360
   - SND.NXT = 1

2. Receive Segment; Send Ack, Reduce Window To 120
   - RCV.WND = 360
   - RCV.NXT = 1
   - Acknowledgment
     - Ack Num = 361
     - Window = 120

3. Reduce Send Window to 120; Send 120-Byte Segment
   - SND.UNA = 1
   - SND.WND = 360
   - SND.NXT = 361

   - Segment
     - Length = 120
     - Seq Num = 361

4. Receive Segment; Send Ack, Reduce Window To 80
   - RCV.WND = 360
   - RCV.NXT = 361
   - Acknowledgment
     - Ack Num = 481
     - Window = 80

3. Reduce Send Window to 80; Send 80-Byte Segment
   - SND.UNA = 361
   - SND.WND = 360
   - SND.NXT = 481

   - Segment
     - Length = 80
     - Seq Num = 361

4. Receive Segment; Send Ack, Reduce Window To 67
   - RCV.WND = 360
   - RCV.NXT = 361
   - Acknowledgment
     - Ack Num = 561
     - Window = 67
Silly Window Syndrome
How to avoid Silly Window Syndrome

• Receiver side
  – Do not advertise small window sizes
  – Min (MSS, MaxRecvBuf/2)

• Sender side
  – Wait until it has a large segment to send
  – Q: How long should a sender wait?
Sender-Side Silly Window Syndrome avoidance

• Nagle’s Algorithm
  – Self-clocking

• Interactive applications may turn off Nagle’s algorithm using the TCP_NODELAY socket option

When app has data to send
  if data and window >= MSS
    send a full segment
  else
    if there is unACKed data
      buffer new data until ACK
    else
      send all the new data now
TCP window management summary

• Receiver uses AdvertisedWindow for flow control

• Sender sends probes when AdvertisedWindow reaches zero

• Silly Window Syndrome avoidance
  – Receiver: do not advertise small windows
  – Sender: Nagle’s algorithm
Overview

• TCP
  – Connection management
  – Flow control
  – When to transmit a segment
  – Adaptive retransmission
  – TCP options
    • Modern extensions
TCP Retransmission

• A TCP sender retransmits a segment when it assumes that the segment has been lost

• How does a TCP sender detect a segment loss?
  – Timeout
  – Duplicate ACKs (later)
How to set the timer

• Challenge: RTT unknown and variable

• Too small
  – Results in unnecessary retransmissions

• Too large
  – Long waiting time
Adaptive retransmission

- Estimate a RTO value based on round-trip time (RTT) measurements

- Implementation: one timer per connection

- Q: Retransmitted segments?
Karn’s Algorithm

• Ambiguity

• Solution: Karn’s Algorithm:
  – Don’t update RTT on any segments that have been retransmitted
Setting the RTO value

• Uses an exponential moving average (a low-pass filter) to estimate RTT ($srtt$) and variance of RTT ($rttvar$)
  – The influence of past samples decrease exponentially

• The RTT measurements are smoothed by the following estimators $srtt$ and $rttvar$:
  
  $$srtt_{n+1} = \alpha \cdot RTT + (1- \alpha) \cdot srtt_{n}$$
  $$rttvar_{n+1} = \beta (| RTT - srtt_{n} | ) + (1- \beta) \cdot rttvar_{n}$$
  $$RTO_{n+1} = srtt_{n+1} + 4 \cdot rttvar_{n+1}$$

  – The gains are set to $\alpha = 1/4$ and $\beta = 1/8$
  – Negative power of 2 makes it efficient for implementation
Setting the RTO value (cont’d)

• Initial value for RTO:
  – Sender should set the initial value of RTO to
    \[ RTO_0 = 3 \text{ seconds} \]

• RTO calculation after first RTT measurements arrived
  \[ srtt_1 = RTT \]
  \[ rttvar_1 = RTT / 2 \]
  \[ RTO_1 = srtt_1 + 4 rttvar_{n+1} \]

• When a timeout occurs, the RTO value is doubled
  \[ RTO_{n+1} = \max (2 RTO_n, 64) \text{ seconds} \]

This is called an exponential backoff
Overview

- TCP
  - Connection management
  - Flow control
  - When to transmit a segment
  - Adaptive retransmission
  - TCP options
    - Modern extensions
TCP header fields

- **Options**: (type, length, value)
- TCP hdrlen field tells how long options are

<table>
<thead>
<tr>
<th>Option</th>
<th>Kind</th>
<th>Length</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>End of Options</td>
<td>0</td>
<td>1 byte</td>
<td>1 byte</td>
</tr>
<tr>
<td>NOP (no operation)</td>
<td>1</td>
<td>1 byte</td>
<td>1 byte</td>
</tr>
<tr>
<td>Maximum Segment Size</td>
<td>2</td>
<td>len=4</td>
<td>maximum segment size</td>
</tr>
<tr>
<td>Window Scale Factor</td>
<td>3</td>
<td>len=3</td>
<td>shift count</td>
</tr>
<tr>
<td>Timestamp</td>
<td>8</td>
<td>len=10</td>
<td>timestamp value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>timestamp echo reply</td>
</tr>
</tbody>
</table>
TCP header fields

- **Options:**
  - **NOP** is used to pad TCP header to multiples of 4 bytes
  - **Maximum Segment Size**
  - **Window Scale Options**
    - Increases the TCP window from 16 to 32 bits, i.e., the window size is interpreted differently
    - This option can only be used in the SYN segment (first segment) during connection establishment time
  - **Timestamp Option**
    - Can be used for roundtrip measurements
Modern TCP extensions

• Timestamp

• Window scaling factor

• Protection Against Wrapped Sequence Numbers (PAWS)

• Selective Acknowledgement (SACK)

• References
Improving RTT estimate

• TCP timestamp option
  – Old design
    • One sample per RTT
    • Using host timer

• More samples to estimate
  – Timestamp option
    • Current TS, echo TS
Increase TCP window size

- 16-bit window size
- Maximum send window \(\leq 65535\)B
- Suppose a RTT is 100ms
- Max TCP throughput = \(65\)KB/100ms = 5Mbps
- Not good enough for modern high speed links!
Protecting against Wraparound

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Time until Wraparound</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>6.4 hours</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>57 minutes</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>13 minutes</td>
</tr>
<tr>
<td>Fast Ethernet (100 Mbps)</td>
<td>6 minutes</td>
</tr>
<tr>
<td>OC-3 (155 Mbps)</td>
<td>4 minutes</td>
</tr>
<tr>
<td>OC-12 (622 Mbps)</td>
<td>55 seconds</td>
</tr>
<tr>
<td>OC-48 (2.5 Gbps)</td>
<td>14 seconds</td>
</tr>
</tbody>
</table>

Time until 32-bit sequence number space wraps around.
Solution: Window scaling option

- All windows are treated as 32-bit
- Negotiating shift.cnt in SYN packets
  - Ignore if SYN flag not set
- Sending TCP
  - Real available buffer >> self.shift.cnt → AdvertisedWindow
- Receiving TCP: stores other.shift.cnt
  - AdvertisedWindow << other.shift.cnt → Maximum Sending Window

<table>
<thead>
<tr>
<th>Kind = 3</th>
<th>Length = 3</th>
<th>Shift.cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Protect Against Wrapped Sequence Number

• 32-bit sequence number space
• Why sequence numbers may wrap around?
  – High speed link
  – On an OC-45 (2.5Gbps), it takes 14 seconds < 2MSL

• Solution: compare timestamps
  – Receiver keeps recent timestamp
  – Discard old timestamps
Selective Acknowledgement

• More when we discuss congestion control

• If there are holes, ack the contiguous received blocks to improve performance
Summary

• Nitty-gritty details about TCP
  – Connection management
  – Flow control
  – When to transmit a segment
  – Adaptive retransmission
  – TCP options
  – Modern extensions
  – Next: Congestion Control
    • How does TCP keep the pipe full?
Midterm statistics
Overall Scores

- **MAXIMUM** – 96.5
- **MEAN** – 79.79
- **MEDIAN** – 82.0
- **STD DEV** – 10.69
Quiz Score

- Use mean to evaluate
- Take Quiz 1  82.021739
- Not Take Quiz 1  72.157895
- Take Quiz 2  82.025000
- Not Take Quiz 2  73.500000
- Either 1 or 2  81.800000
- Neither 1 nor 2  66.958333
Survey results
1. Has the course met your expectations of an undergraduate networking class so far? If not, please explain the areas where the course has not met your expectations.
2. Are the lectures given a) at an appropriate speed; b) too fast; c) too slow? Please circle one answer.
3. Do you have any suggestion on how the course materials might be improved?
4. Do you have any suggestion on how the instructor might improve her teaching quality?
• 5. Do you have any suggestion on how the TA might improve his teaching quality?