Today's Lecture

I. TCP Error Detection and Correction
II. "Lost" ACKs and the Persist Timer
III. Estimating the Round Trip Time
IV. Calculating Retransmission Timeout Intervals
V. TCP Congestion Control
VI. The "Slow Start" Algorithm

TCP Error Detection and Correction

TCP Reliability

• Lost segments are detected and recovered
• Corrupted segments (invalid checksum) are detected and recovered
• Lost and corrupted segments are detected by failure to receive an acknowledgment
  – sender has to decide how long to wait before retransmitting (and how many times to retransmit)

TCP Reliability (cont’d)

• Duplicate segments are detected (using Sequence Number) and ignored by receiver (but acknowledgment will be sent to sender)
• Out-of-order segments are detected (using Sequence Number) and reordered (in reordering buffer)
  – receiver acknowledges only consecutively received segments
• Lost Acknowledgments may also lead (incorrectly) to retransmission of data
  – usually not a problem; why not?

Example of Lost ACKs
In-Class Exercise

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQ=5000, ACK=4444</td>
<td>A sends 1000 byte segment</td>
</tr>
<tr>
<td>SEQ=4444, ACK=6000</td>
<td>B sends 100 byte segment</td>
</tr>
<tr>
<td>SEQ=6000, ACK=4544</td>
<td>A sends 1000 byte segment</td>
</tr>
<tr>
<td>SEQ=7000, ACK=4544</td>
<td>A sends 1000 byte segment</td>
</tr>
<tr>
<td>SEQ=8000, ACK=4544</td>
<td>A sends 1000 byte segment</td>
</tr>
<tr>
<td>SEQ=4544, ACK=7000</td>
<td>B sends 100 byte segment</td>
</tr>
<tr>
<td>SEQ=4644, ACK=9000</td>
<td>B sends 0 byte segment</td>
</tr>
<tr>
<td>SEQ=4644, ACK=9000</td>
<td>B sends 0 byte segment</td>
</tr>
<tr>
<td>SEQ=10000, ACK=4644</td>
<td>A sends 1000 byte segment</td>
</tr>
</tbody>
</table>

**Problem Caused by A Lost ACK**

- It is possible for receiver to advertise a Window Size = 0
  - fast sender + slow receiver = full receiver buffer
- When receiver (eventually) empties the buffer, requests more data by sending an ACK with Window Size > 0
- If this ACK is lost by network → **deadlock**!
  - sender can’t send anything
  - receiver has no reason to send another ACK

**The Persist Timer**

- Solution: after receiving any ACK with Window Size = 0, start a **persist timer**
  - upon expiration of the persist timer, send a **probe packet**
    - probe packet = 1 more byte of data (technically, violates Window Size advertisement)
- Normal persist timer intervals
  - 1.5s, 3s, 6s, 12s, 24s, 48s, 60s, 60s, 60s, ...
- The ACK of this probe replaces the lost ACK
  - i.e., transmitting the probe “stimulates” new advertisements

**Persist Timer, Illustrated**

- Send 1K of data
- Receiver buffer is full
- Send a “probe” packet with 1 byte of data
- Application removes data from buffer, window opens up
- Receive new window size, clear to transmit more data

**ESTIMATING THE ROUND-TRIP TIME (RTT)**
Motivation

• How long should we wait before deciding a segment has been lost and should be retransmitted?
  – too short → ?
  – too long → ?
  – just right → just a little bit longer than the longest time it could take to transmit a segment and receive an ACK

Estimating RTT

• Notation
  – \( rtt_i \) = time between transmission of \( i \)th packet until receive ACK of \( i \)th packet
  – \( RTT_i \) = estimate of average round trip time after \( i \)th packet

• Exponentially weighted moving average (EWMA):
  \[ RTT_i = \alpha \cdot RTT_{i-1} + (1-\alpha) \cdot rtt_i \]
  – (assume \( RTT_0 = 0 \))
  – Jacobson’s algorithm: \( \alpha = .875 \)

Example of RTT Calculation

• Notes: calculations in full precision, results shown only to nearest integer, time before packet #20 not shown

<table>
<thead>
<tr>
<th>Pkt #</th>
<th>( rtt_i ) (ms)</th>
<th>( RTT_i ) (ms)</th>
<th>( \alpha = .875 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>30</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>80</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>40</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>130</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>40</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

Small Correction

• Problem: if a segment is retransmitted, is an ACK for the first, second, … or \( n \)th transmission of that segment?
  – can’t tell how to compute \( rtt_i \) in this case

• Solution (“Karn’s Algorithm”):
  don’t use (i.e., ignore) \( rtt_i \) from any retransmitted segments to update \( RTT_i \)

Calculating the Variation in the RTT

• Motivation: ???

• True standard deviation (expensive to compute):
  \[ STDDEV = \left( \left( rtt_1 - \text{avg}(rtt) \right)^2 + \left( rtt_2 - \text{avg}(rtt) \right)^2 + \ldots \right)^{1/2} \]

• The “mean deviation” (cheap to compute):
  \[ MDEV_i = (1-\rho) \cdot MDEV_{i-1} + \rho \cdot \left| rtt_i - RTT_{i-1} \right| \]
  – another EWMA!
  – recommended value for \( \rho = .25 \)

Example of MDEV Calculation

<table>
<thead>
<tr>
<th>Pkt #</th>
<th>( rtt_i ) (ms)</th>
<th>( RTT_i ) (ms)</th>
<th>( MDEV_i ) (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>40</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>21</td>
<td>30</td>
<td>39</td>
<td>10</td>
</tr>
<tr>
<td>22</td>
<td>80</td>
<td>44</td>
<td>18</td>
</tr>
<tr>
<td>23</td>
<td>40</td>
<td>43</td>
<td>14</td>
</tr>
<tr>
<td>24</td>
<td>130</td>
<td>54</td>
<td>32</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>54</td>
<td>25</td>
</tr>
</tbody>
</table>
Determining RTO

- RFC 793 originally recommended:
  \[ RTO_i = 2 \times RTT_{i-1} \]
  (i.e. allow two roundtrip times before timing out)

- Problem
  - when RTT has a high standard deviation, this method frequently times out too quickly (doesn’t wait long enough)

Example of RTO Calculation (Original)

<table>
<thead>
<tr>
<th>Pkt #</th>
<th>rtt (_i) (ms)</th>
<th>RTO (_i)</th>
<th>RTT (_i) (ms)</th>
<th>(\alpha = .875)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>30</td>
<td>80</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>80</td>
<td>78</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>40</td>
<td>88</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>130</td>
<td>86</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>50</td>
<td>108</td>
<td>54</td>
<td></td>
</tr>
</tbody>
</table>

Retransmission timeouts would occur!
(retransmissions not shown, Karn’s algorithm not used)

Improved Method of Computing RTO

- Solution
  - set \(RTO = RTT + \text{a multiple of the mean deviation}\)

- Jacobson’s algorithm:
  \[ RTO_i = RTT_{i-1} + 4 \times \text{MDEV}_{i-1} \]
  - calculations can be implemented in a very efficient way

Example of RTO Calculation (Improved)

<table>
<thead>
<tr>
<th>Pkt #</th>
<th>rtt (_i) (ms)</th>
<th>RTO (_i)</th>
<th>RTT (_i) (ms)</th>
<th>MDEV (_i) (ms)</th>
<th>(\rho = .25)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td>40</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>30</td>
<td>120</td>
<td>39</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>80</td>
<td>109</td>
<td>44</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>40</td>
<td>138</td>
<td>43</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>130</td>
<td>118</td>
<td>54</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>196</td>
<td>54</td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>

Retransmission still occurs!
(no retransmissions shown, Karn’s algorithm not used)
The TCP “Coarse-Grained” Timer

- For efficiency, TCP timers tick in “large” increments
  - e.g., rtt is measured to the nearest 500ms
  - result: RTTs and RTOs may differ from expected times due to this discretization
- See Stephens Vol I for more implementation details (fixed point arithmetic, etc.)

Retransmission Timer Backoff

- Retransmission may occur because RTO is too short
  - e.g., if the network congestion increases, RTO should be increased
  - problem: Karn’s algorithm says don’t modify RTT / MDEV for retransmitted segments!
- Solution: exponential backoff
  - upon timeout, retransmit and compute:
    \[ RTO_i = 2 \times RTO_{i-1} \]
    and no change to RTT

Retransmission Timer Backoff (cont’d)

- Example: RTO = 2s, after first timeout = 4s, after second timeout = 8s, etc.
- Questions
  - why is this necessary?
  - similar to anything else you know about?
  - how long should TCP keep trying until giving up?

Fairness

- When congestion occurs, everyone using the same bottleneck link should reduce their transmission rate in a way that is fair
  - the most ambiguous word in English language?
  - TCP congestion control is proportional-fair
- Control mechanism is end-to-end, operated by end systems
  - who enforces fairness and what’s the punishment for violating fairness?

TCP Congestion Control

- Congestion control: reduce transmission rate to match the current network bandwidth available
  - depends on the path to the destination
  - dictated by the “bottleneck” link
- Assumption: packet loss is due to congested routers, not transmission errors
  - may be incorrect for some technologies, such as wireless
  - what difference does it make?
TCP Feedback

- Implicit feedback: the source **infers** network conditions from the feedback (acknowledgments) it receives
- Info available to the source
  - ACKs received
  - measured *rtt's*
  - retransmission timeouts

TCP Sliding Windows

- Adapts rate by controlling the *congestion window*
  - smaller congestion window ⇒ send at lower rate
  - larger congestion window ⇒ send at higher rate
- Sliding windows are used for **two purposes**
  - flow control: *Window Size* advertisement
  - congestion control: *cwnd* (congestion window)
    - computed by source, not explicitly communicated
- Transmission Window = \(\min\{\text{Window Size}, \text{cwnd}\}\)
  - sender slows down to the rate of the slowest component (network or receiver)

TCP Congestion Control Overview

1. Start with very slow transmission rate
   \(cwnd = 1\) segment per *rtt*
2. Probe for available bandwidth by increasing *cwnd* until packet losses start occurring
   - Slow Start = increase rapidly
   - *congestion avoidance* = increase more slowly
   - Slow Start threshold \(ssthresh\) marks transition from rapid to slower increase phase
3. When a packet loss is detected (whoops, increased too far), start over (reset *cwnd* to 1)

TCP "SLOW START"

**Notes**
- *MSS* = maximum segment size
- all quantities in units of bytes

At connection establishment:
\[cwnd \leftarrow \text{MSS}\]
\[\text{ssthresh} \leftarrow 65535\]

Upon arrival of each new ACK:
\[cwnd \leftarrow cwnd + \text{MSS}\]

- Not a particularly slow increase!
  - *cwnd* doubles once every round-trip time (*RTT*)

**Graphs:**
- Graph A shows the slow start phase with *cwnd* increasing from 1 to 8192 segments.
- Graph B illustrates how the *ssthresh* is adjusted after packet losses.
Slow Start, Illustrated

- With Slow Start, in roughly $N$ RTTs, $cwnd = 2N \cdot MSS$
  - can quickly congest the network

Congestion Avoidance

- Slow down the rate of increase as you approach the most recent "point of congestion"
  - $ssthresh$ is used for this purpose
- If $cwnd > ssthresh$, upon arrival of each new ACK:
  $cwnd \leftarrow cwnd + MSS^2 / cwnd$
- Result: $cwnd$ approximately increases by one segment every $cwnd$ ACKs

Congestion Avoidance, Illustrated

Reacting to Congestion

- Congestion may affect delivery of data and/or acknowledgments
- When a retransmission timeout occurs,
  $cwnd \leftarrow MSS$ \quad /\quad \text{"start all over again"}/
  $ssthresh \leftarrow \max(2, \frac{1}{2} \cdot \text{Transmission Window})$
- Result: start over, but switch to congestion avoidance halfway to point where congestion previously occurred

Effect of Timeouts, Illustrated

Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>cwnd</th>
<th>ssthresh</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>(assume)</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td>t2</td>
<td>send S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>receive ACK of S1</td>
<td>2048 (=1024+1024)</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>send S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>send S3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6</td>
<td>receive ACK of S2</td>
<td>3072 (=2048+1024)</td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td>send S4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t8</td>
<td>send S5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t9</td>
<td>receive ACK of S3</td>
<td>3413 (=3072+341)</td>
<td></td>
</tr>
<tr>
<td>t10</td>
<td>receive ACK of S4</td>
<td>3720 (=3413+307)</td>
<td></td>
</tr>
<tr>
<td>t11</td>
<td>send of S6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t12</td>
<td>send of S7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t13</td>
<td>receive ACK of S5</td>
<td>4002 (=3720+282)</td>
<td></td>
</tr>
<tr>
<td>t14</td>
<td>receive ACK of S6</td>
<td>4264 (=4002+262)</td>
<td></td>
</tr>
<tr>
<td>t15</td>
<td>timeout of S7 occurs</td>
<td>1024</td>
<td>2132</td>
</tr>
</tbody>
</table>
Summary

1. TCP uses Sequence # and ACK # to detect errors

2. Calculating the retransmission timeout interval is surprisingly complicated
   - RTT average computed as EWMA of rtts
   - Mean deviation computed
   - RTO = RTT + 4*MDEV
   - exponential backoff when losses occur

Summary (cont’d)

3. TCP uses end-to-end congestion control with implicit feedback

4. The congestion window provides congestion control
   - Slow start uses rapid initial increase, then slows down to avoid congestion

Next Lecture

- TCP, lecture 4