Receiver Window

• The receiver window manages flow control.
  • Receiver adjusts size to reflect buffer space.
  • Sends window size updates via “Window” field in TCP segment header.

• Sender won't send more data than receiver can handle.
  • ... even in the case where receiving application is busy elsewhere.
Sources of Slowness

- **Receiver**
  - Slow computer
  - Distracted program
    - Dealing with other tasks...
    - Processing received data is complicated...
  - *Receiver buffer fills and receiver window shrinks.*

- **Network**
  - Slow links
  - High traffic
  - *How is this handled?*
Congestion Window

• TCP maintains a second window.
  • *Estimate of the network's capacity to transmit data.*
  • **Sender** must compute the size of this window
    - Based on implicit feedback from the receiver
      • Successful ACKs
      • Timeouts
    - *Assumption: Lost segments are due to network congestion* (is this really true?)

• Actual window used for transmission is the smallest of (receiver, congestion).
Slow Start

- Congestion window size ($cwnd$) starts small and grows to “probe” the network capacity.
  - In what follows “one segment” means the MSS used by the sender (typically 1460 bytes on ethernet).
- Initialize with $cwnd = 1$ segment.
- Increment $cwnd$ by 1 segment for each segment acknowledged.
  - This increases $cwnd$ exponentially!
Exponentially?

• Consider...
  • Set $cwnd = 1$ segment. Send it.
  • Wait for ACK. Set $cwnd = 2$ segments. Send them.
  • After both ACKs...
    – Set $cwnd = 2 + 1 + 1 = 4$ segments. Send them.
  • After all four ACKS...
    – Set $cwnd = 4 + 1 + 1 + 1 + 1 = 8$ segments. Send them.
• In real life it is more complicated.
  • ACKs don't really arrive all together (in general).
  • TCP follows the same basic rule, however.
Slow Start Threshold

- A second value, `ssthresh`, defines when slow start ends and “congestion avoidance” begins.
  - After `cwnd` reaches `ssthresh`...
    - Increment `cwnd` by \(1/cwnd\) (as measured in segments) for each ACK.
      - Example: If `cwnd = 4` segments, then add \(\frac{1}{4}\) segment to `cwnd` in response to the next ACK.
      - Thus 4 ACKs needed to increase `cwnd` by 1 segment.
    - Thus `cwnd` increases by 1 for each round trip time regardless of segment count.
  - **Causes a linear increase of `cwnd`.**
Summary

TCP Congestion Window

$ssthresh$
Timeout!

• When a timeout occurs...
  • `ssthresh` is set to \( \frac{1}{2} \) the current `cwnd` value.
  • `cwnd` is set to 2.
  • Slow start begins again.

• TCP assumes timeout means data loss.
  • Backs off by reducing the congestion window size.
  • Begins probing the network again in case source of congestion is gone.
Remember...

- TCP uses the smallest of (receiver, congestion) windows.
  - Once $cwnd$ exceeds the receiver window, flow is limited by receiver window size.
  - This is the normal case on a clear network.
- On a WAN, however, $cwnd$ is often limiting.
- Many details left out of this discussion.
  - See references slide at end of this slide group.
How Long to Timeout?

- Too long...
  - If TCP waits too long to retransmit a lost segment time is wasted.
  - Slows down transmission.
- Too short...
  - If TCP doesn't wait long enough, it may retransmit unnecessarily.
  - Clogs the network.
  - Wastes bandwidth.
Round Trip Time?

• How long is a normal round trip?
  • LAN...
    – Transit time is sub-millisecond.
    – Usually steady.
  • WAN...
    – Transit time is multiple millisecond.
    – Often tens, hundreds, even thousands of milliseconds.
    – Often highly variable.
• Computation time is usually short.
  – TCP acknowledges, application not involved.
RTT Estimation (Old)

- RFC-793 contains an algorithm for estimating round trip time (RTT).
  - Associate a timer with each outgoing segment.
  - When an ACK comes in, note the measured RTT for that segment (M).
- Compute: $R_{\text{new}} = \alpha R_{\text{old}} + (1 - \alpha) M$
  - Where $\alpha$ is a scale factor (typically 0.9). $R$ is an estimate of the RTT.
- Compute timeout: $T = R_{\text{new}} \beta$
  - Where $\beta$ is another scale factor (typically 2).
Problems

- The previous algorithm is not that great.
  - Can't keep up with changes.
  - Doesn't deal with highly variable RTT values.
  - Tends to cause many unnecessary retransmissions.
- What is needed is a way to account for the degree of variability in the RTT.
Jacobson's Algorithm

- Compute both RTT and “deviation” estimates.
  - Compute \( E_r = M - R_{old} \)
    - Note that the error value is signed.
  - Compute \( R_{new} = R_{old} + g \cdot E_r \)
    - Here \( g \) is typically \( \frac{1}{8} \).
  - Compute \( D_{new} = D_{old} + h \left( |E_r| - D_{old} \right) \)
    - Here \( h \) is typically \( \frac{1}{4} \). \( D \) is an estimate of the deviation in observed RTT values.
  - Compute \( T_{new} = R_{new} + 4 \cdot D_{new} \)
    - Time is RTT with extra to account for variability of RTT.
- Note that computations above are easy.
### TCP Performance

<table>
<thead>
<tr>
<th></th>
<th>DATA Bytes</th>
<th>ACK Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Ethernet Header</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>IP Header</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>TCP Header</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Data</td>
<td>1460</td>
<td>0</td>
</tr>
<tr>
<td>Pad</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>CRC</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Interpacket Gap</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1538</td>
<td>84</td>
</tr>
</tbody>
</table>

Needed to meet ethernet minimum of 64 bytes per frame

9.6 microseconds on 10 Mbps ethernet.
Performance Computation

- Assume one ACK for every two data segments
  - In real life there are many possibilities.
- Assume 10 Mbps ethernet.

\[
\text{Real data} \quad \frac{2(1460) }{2(1538) + 84} \quad \text{Raw data rate (1,250,000 bytes/s)} \quad 10,000,000 \quad \frac{*}{8} = 1,155,063 \text{ bytes/s}
\]

Data+overhead \quad ACK overhead \quad Actual data rate
References

- RFC-896: Congestion Control in IP/TCP Internetworks. (Describes Nagle's Algorithm for interactive connections).
- RFC-2581: TCP Congestion Control.
- [http://www.winlab.rutgers.edu/~hongbol/tcpWeb/tcpTutorialNotes.html](http://www.winlab.rutgers.edu/~hongbol/tcpWeb/tcpTutorialNotes.html)