Interactive and bulk data

• TCP applications can be put into the following categories
  – bulk data transfer - ftp, mail, http
  – interactive data transfer - telnet, rlogin

• TCP has algorithms to deal which each type of applications efficiently.
Rlogin

- “Rlogin” is a remote terminal application
- Originally built only for Unix systems.
- Rlogin sends one segment per character (keystroke)
- Receiver echoes the character back.
- So, we really expect to have four segments per keystroke

Q: What is the total number of bytes transmitted for a single keystroke?
A: TCP hdr = 20; IP hdr = 20; ethernet hdr/trailer: 18 bytes
59+58+59+58=234 bytes

Segment exchange in rlogin

- We would expect that tcpdump shows this pattern:
- However, tcpdump shows this pattern:
- So, TCP has delayed the transmission of an ACK
tcpdump of an rlogin session

This is the output of typing the first character:
44.062449 aida.poly.edu.1023 > catt.poly.edu.login: P 50:51(1) ack 1
44.063317 catt.poly.edu.login > aida.poly.edu.1023: P 1:2(1) ack 51 win 8760
44.182705 aida.poly.edu.1023 > catt.poly.edu.login: . ack 2 win 17520

This is the output of typing the second character:
48.946471 aida.poly.edu.1023 > catt.poly.edu.login: P 51:52(1) ack 2 win 17520
48.947326 catt.poly.edu.login > aida.poly.edu.1023: P 2:3(1) ack 52 win 8760
48.982786 aida.poly.edu.1023 > catt.poly.edu.login: . ack 3 win 17520

Delayed Acknowledgement

- TCP delays transmission of ACKs for up to 200ms
- The hope is to have data ready in that time frame. Then, the ACK can be piggybacked with the data segment.
- Delayed ACKs explain why the ACK and the “echo of character” are sent in the same segment.
- What are the delays of the ACKs in the tcpdump example?
Delayed ACK timer

- This timer ticks every 200ms.
- First timeout occurs based on when the timer was initialized, which is when the system was rebooted.
- The figure below explains why the delay for the ACKdelay is up to 200 ms (and not equal to 200 ms).

![Diagram showing delayed ACK timer]

TCP receives segment somewhere here

200 ms per tick

Delayed ACK timer expires (ACK has to be sent at this point whether or not TCP buffer has received data to enable piggybacking)

tcpdump of a wide-area rlogin session

This is the output of typing 7 characters:

54:16.481929 tenet.CS.Berkeley.EDU.login > aida.poly.edu.1023: P2:3(1) ack 2 win 16384
54:16.482154 aida.poly.edu.1023 > tenet.CS.Berkeley.EDU.login: P2:3(1) ack 3 win 16383
Wide-area Rlogin: Observation 1

- Transmission of segments follows a different pattern.
- The delayed acknowledgment does not kick in.
- Reason is that since the delays for sending segments across the wide area network are much larger, there is always data at aida when it needs to send an ACK for the echo’ed character.

Wide-area Rlogin: Observation 2

- Aida never has multiple segments outstanding (meaning unacknowledged segments).
- This is due to Nagle’s Algorithm:
  
  Each TCP connection can have only one small segment (less than MSS) outstanding that has not been acknowledged.

- Implementation: Send one byte and buffer all subsequent bytes until acknowledgement is received. Then send all buffered bytes in a single segment. (Only enforced if byte is arriving from application one byte at a time)
- Nagle’s rule reduces the amount of small segments. The algorithm can be disabled.
Disabling Nagle’s algorithm

- Typically function keys generate multiple bytes of data
- If TCP gets 1 byte of data at a time, it may send the first byte in a segment, and wait for an ACK before sending other bytes (if Nagle’s algorithm is enabled).
- The receiver does not generate an echo till the remaining bytes are received.
- This triggers delayed ACK, which means the receiver waits till the 200ms delayed ACK timer goes off before sending an ACK.
- The interactive user may see noticeable delays.
- Delay is low if Nagle’s algorithm is disabled, but throughput is also low.

Summary of interactive data transfer in TCP

- Delayed ACK timer
- Nagle’s algorithm
- Chapter 19 of EL537 textbook
Bulk data transfer (flow control)

- If TCP connections exchange a lot of data (bulk data transfer), then one deals with a different set of problem:

  - **Flow Control:** How to prevent that the sender overruns the receiver with information?

Flow Control in TCP

- TCP implements sliding window flow control
- Sending acknowledgements is separated from setting the window size at sender. Acknowledgements do not automatically increase the window size
- Acknowledgements are cumulative

Window Management in TCP

- The receiver is returning two parameters to the sender

<table>
<thead>
<tr>
<th>AckNo</th>
<th>window size (win)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- The interpretation is:
  - I am ready to receive new data with \( \text{SeqNo=} \text{AckNo, AckNo+1, \ldots, AckNo+Win-1} \)
  - Receiver can acknowledge data without opening the window
  - Receiver can change the window size without acknowledging data
Sliding Window Flow Control

- Sliding Window Protocol is performed at the byte level:

```
1  2  3  4  5  6  7  8  9  10  11

Advertised window
```

- Here: Sender can transmit sequence numbers 6,7,8.

---

Sliding Window: “Window Closes”

- Transmission of a single byte (with SeqNo = 6) and acknowledgement is received (AckNo = 5, Win=4):

```
1  2  3  4  5  6  7  8  9  10  11

Transmit Byte 6
```

```
1  2  3  4  5  6  7  8  9  10  11

AckNo = 5, Win = 4
```

```
1  2  3  4  5  6  7  8  9  10  11

is received
```
**Sliding Window: “Window Opens”**

- Acknowledgement is received that enlarges the window to the right (AckNo = 5, Win=6):

```
1 2 3 4 5 6 7 8 9 10 11
```

- A receiver opens a window when TCP buffer empties (meaning that data is delivered to the application).

**Sliding Window: “Window Shrinks”**

- Acknowledgement is received that reduces the window from the right (AckNo = 5, Win=3):

```
1 2 3 4 5 6 7 8 9 10 11
```

- Shrinking a window should not be used - Host requirements RFC strongly discourages this
Example

- The following overhead slides (see Fig. 20.1 on pg. 276 and Fig. 20.3 on page 279 of textbook) are taken from the textbook.
- The slides show a transfer of $8 \times 1024$ byte by a client
  
  ```
  (sock -i -s 7777 ; sock -i -n8 bsdi 7777)
  ```
Example: Transmission 1

Open your textbook to page 276

1 - 3: TCP connection is established
4 - 6: 3 segments are transmitted
7: ACK for segments 4+5 (because of delayed ACK)
8: Ack for segment 6 (with win reduced to to 3072)
9 - 10: Transmission and ACK for segment 9
11 - 13: Transmission of data segments
14: ACK for segments 11 and 12.
15: Transmission of data segment
16: ACKs for segments 13 and 15
17 - 20: Close connection

• ACK every other segment strategy (see page 277)

Example: Transmission 2 (slow receiver)

Open your textbook to page 279

1 - 3: TCP connection is established
4 - 7: 4 segments are transmitted
8: ACKs for all segments 4+5+6+7 (with win reduced to 0 !)
9: Window update segment with win being reset to 4096
10 - 13: Transmission of data segments and active close on segment 13
14: ACK for segments 10,11,12,13
15: Window update
16: Passive close
17: Ack for close
Congestion Control

- TCP implements congestion control at the sender
  - This control is intended to reduce congestion in the network.
- The sender has two parameters for congestion control:
  - Congestion Window \((cwnd)\); Initial value is MSS bytes
  - Threshold Value \((ssthresh)\); Initial value is 65536 bytes
- The window size at the sender is set as follows:
  Allowed Window = MIN (advertised window, congestion window)

MSS: Maximum Segment Size (set with option field in TCP header)

Slow Start

- Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
  - Set \(cwnd =\)MSS bytes (cwnd is stored in bytes)
  - Each time an ACK is received, the congestion window is increased by 1 segment (= MSS bytes).
  - If an ACK acknowledges two segments, cwnd is still increased by only 1 segment (Caveat: see slide 26)
  - Even if ACK acknowledges a segment that is smaller than MSS bytes long, cwnd is increased by MSS bytes.
  - If cwnd is 3 but there is still one outstanding ACK, the sender can only send two segments (See page 287).
- Does Slow Start increment slowly? Not really. In fact, the increase of cwnd is exponential
Example of Slow Start

- The congestion window size grows very rapidly
  - For every ACK, we increase cwnd by 1 irrespective of the number of segments ACK’ed (*)
- TCP slows down the increase of cwnd when \( cwnd > ssthresh \)

Unresolved question (*)

- Popular belief is that in slow start cwnd increases by the number of segments ack’ed. In other words, if an ack acknowledges two segments (cumulative ACK), then cwnd is increased by two segments (not 1 - as stated in previous slides). This leads to true exponential growth. Tanenbaum’s book states this.
- However the EL537 book says cwnd is increased by 1 segment for each ACK received without saying anything about the number of segments ack’ed by the ACK. We looked at TCP Reno implementation. It behaves in this manner as well.
- This is not in the TCP spec. as slow start was defined in a Van Jacobson paper later on (Jacobson 88 - see EL537 references. The ftp site mentioned does not have this paper).
- For purposes of this course, we will assume that cwnd increases by 1 segment for each ACK received, irrespective of the number of segments acknowledged.
Congestion Avoidance

- Slow down “Slow Start”
- If $cwnd > ssthresh$ then each time an ACK is received, increment $cwnd$ as follows:
  \[ cwnd = cwnd + \text{segsize} \times \text{segsize} / cwnd + \text{segsize} / 8 \]
- (We will learn later how to set $ssthresh$)

Slow Start/Congestion Avoidance Example

- Assume that $ssthresh = 8$
Example of slow start and congestion avoidance

- Assume MSS = 512 bytes; advertised window = 5120 bytes

\[
\begin{align*}
\text{PSH 1:513 (512) ack 10} \\
\text{cwnd=512; ssthresh=2560} \\
\text{ack 513} \\
\text{PSH 513:1025 (512) ack 10} \\
\text{cwnd=1536; ssthresh=2560} \\
\text{ack 1025} \\
\text{PSH 1025:1537 (512) ack 10} \\
\text{cwnd=2048} \\
\text{ack 1537} \\
\text{PSH 1537:2049 (512) ack 10} \\
\text{cwnd=2560; ssthresh=2560} \\
\text{ack 2049} \\
\text{PSH 2049:2561 (512) ack 10} \\
\text{cwnd=3072; ssthresh=2560} \\
\text{ack 2561} \\
\text{PSH 2561:3073 (512) ack 10} \\
\text{cwnd=3222; ssthresh=2560} \\
\text{ack 3073}
\end{align*}
\]

Enter congestion avoidance

Computation of cwnd on previous slide

- Upto and including ack 2561, this TCP connection is in slow start, and cwnd is increased by 1 MSS bytes each time an ACK is received.

- Note that when cwnd = ssthresh, slow start is still applied. Hence when ack 2561 is received, cwnd = 2560 + 512 = 3072.

- When the last ack shown on the previous slide is received, the TCP connection is in congestion avoidance since cwnd is > ssthresh. Therefore, \( cwnd = cwnd + \frac{MSS \times MSS}{cwnd + MSS} / 8 = 3072 + \frac{512 \times 512}{3072 + 3072 + 512} / 8 = 3222 \)
Is TCP selective repeat or Go-back-N?

- It is primarily Go-back-N because it uses cumulative ACKs
  - Need selective ACKs if selective repeat retransmissions are used
- But it appears to be selective repeat in certain cases: see example on page 308. After segment 65 is sent, the next segment sent is a new segment with sequence number 8961. Segments with sequence numbers 6913 through 8705 are not repeated. This is because this example shows fast retransmit, in which case only one segment is sent. Even without fast retransmit, if segment 65 was sent in response to a timeout, cwnd becomes 1 (in case of timeout); so only one segment can be sent; if ack received acknowledges intermediate segments, it appears like selective repeat.