TCP - Part II

- Data Transfer in TCP
- Acknowledgements
- Flow Control

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 heuristics to deal with each type of applications efficiently.
Problems with interactive data transfer

- Interactive data transfer (rlogin, telnet) involves small amounts of data which are sent in infrequent intervals.

- **Problem #1:**
  - A lot of acknowledgements are sent for small amounts of data.
  - **Heuristic:** Delayed Acknowledgements

- **Problem #2:**
  - A lot of small segments are transmitted, especially, if the roundtrip delay is large
  - **Heuristic:** Nagle’s algorithm

TCPdump of an rlogin session

This is the output of typing 3 (three) characters:

```
44.062449 aida.poly.edu.1023 > catt.poly.edu.login: P 0:1(1) ack 1
44.063317 catt.poly.edu.login > aida.poly.edu.1023: P 1:2(1) ack 1 win 8760
44.182705 aida.poly.edu.1023 > catt.poly.edu.login:. ack 2 win 17520
48.946471 aida.poly.edu.1023 > catt.poly.edu.login: P 1:2(1) ack 2 win 17520
48.947326 catt.poly.edu.login > aida.poly.edu.1023: P 2:3(1) ack 2 win 8760
48.982786 aida.poly.edu.1023 > catt.poly.edu.login:. ack 3 win 17520
55:00.116581 aida.poly.edu.1023 > catt.poly.edu.login: P 2:3(1) ack 3 win 17520
55:00.117497 catt.poly.edu.login > aida.poly.edu.1023: P 3:4(1) ack 3 win 8760
55:00.183694 aida.poly.edu.1023 > catt.poly.edu.login:. ack 4 win 17520
```
Rlogin

• “Rlogin” is a remote terminal applications
• Originally build 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?

Rlogin

• We would expect that tcpdump shows this pattern:

• However, tcpdump shows this pattern:

• So, TCP has delayed the transmission of an ACK
Delayed Acknowledgement

- Delay transmission of ACKs for up to 200ms
- But:
  - Send an ACK at least for every other segment arrival
  - Send an ACK immediately if a duplicate is received

- Reasoning for Delayed ACKs: The hope is to have data ready after the 200ms delay. 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.

tcpcdump of a wide-area rlogin session

This is the output of typing 9 characters:

54:16.481929 tenet.CS.Berkeley.EDU.login > aida.poly.edu.1023: P 2:3(1) ack 2 win 16384
54:16.482154 aida.poly.edu.1023 > tenet.CS.Berkeley.EDU.login: P 2:3(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 there is always data at aida when the ACK arrives.

Wide-area Rlogin: Observation 2

- There is a few transmissions than there are characters.
- Aida never has multiple segments outstanding.
- This is due to a heuristic which is called Nagle’s Algorithm.
- Nagle’s algorithm prevents that a large number of small segments are transmitted.
Nagle’s Algorithm

- Send a new segment only if
  - data fills a maximal sized segment, or
  - data fills half the receiver buffer, or
  - all previously transmitted segments are acknowledged

- Thus, Nagle’s rule reduces the amount of small segments. Only one 1-byte segment can be in transmission.
- **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)
- The algorithm can be disabled.

Flow Control and Congestion Control

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

- 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

Example

- The following overhead slides (see handout) are taken from the textbook.
- The slides show a transfer of 4 * 1024 byte by a client (sock -i -s 7777 ; sock -l -n8 bsd 7777)
Example: Transmission 1

1 - 3: TCP connection is established

4 - 6: 3 segments are transmitted
7: ACK for segments 1+2 (because of delayed ACK)
8: Ack for segment 3 (with win reduced to 3072)
9 - 10: Transmission and ACK for segment 4
11 - 13: Transmission of segments 5+6+7
14: ACK for segments 5+6.
15: Transmission of segment 8
16: ACKs for segments 7+8

17 - 20: Close connection

Example: Transmission 2 (slow receiver)

1 - 3: TCP connection is established

4 - 7: 4 segments are transmitted
8: ACKs for all segments 1+2+3+4 (with win reduced to 0 !)
9: another ACK for all segments (but with win reset to 4096 !)
10 - 13: Transmission of segments 5+6+7+8 and active close.
11 - 13: Transmission of segments 5+6+7
8: ACKs for all segments 1+2+3+4 (with win reduced to 0 !)
9: another ACK for all segments (but with win reset to 4096 !)
14: ACK for segments 5+6.
15: Transmission of segment 8
16: ACKs for segments 7+8

17 - 20: Close connection
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

- Sent and acknowledged
- Sent but not acknowledged
- Can be sent
- Usable window
- Can't sent

- 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 is received

```
  1 2 3 4 5 6 7 8 9 10 11
```
Sliding Window: “Window Opens”

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

  ![Diagram showing window enlargement]

- 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):

  ![Diagram showing window reduction]

- Shrinking a window should not be used.
Window Management in TCP

- The receiver is returning two parameters to the sender

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

- The interpretation is:
  - I am ready to receive new data with 
    \( \text{SeqNo}= \text{AckNo}, \text{AckNo}+1, \ldots, \text{AckNo}+\text{Win}+1 \)

- Receiver can acknowledge data without opening the window
- Receiver can change the window size without acknowledging data

Sliding Window: Example

- Sender sends 2K of data
- AckNo=2048 Win=2048
- Sender sends 2K of data
- AckNo=4096 Win=0
- AckNo=4096 Win=1024
- AckNo=4096 Win=0
- AckNo=4096 Win=0

© Jörg Liebeherr, 1998-2000
TCP Congestion Control

• TCP has a mechanism for congestion control. The mechanism is implemented at the sender

• The window size at the sender is set as follows:
  \[ \text{Send Window} = \min (\text{flow control window}, \text{congestion window}) \]

where
• flow control window is advertised by the receiver
• congestion window is adjusted based on feedback from the network

TCP Congestion Control

• The sender has two additional parameters:
  – Congestion Window (cwnd)
    Initial value is 1 MSS (=maximum segment size) counted as bytes
  – Slow-start threshold Value (ssthresh)
    Initial value is the advertised window size)

• Congestion control works in two modes:
  – slow start \( (cwnd < ssthresh) \)
  – congestion avoidance \( (cwnd \geq ssthresh) \)
Slow Start

- Initial value:
  - $\text{cwnd} = \text{MSS bytes (\sim 1 segment)}$
- Each time an ACK is received, the congestion window is increased by MSS bytes.
  - $\text{cwnd} = \text{cwnd} + \text{MSS bytes}$
  - If an ACK acknowledges two segments, cwnd is still increased by only MSS bytes (= 1 segment).
  - Even if ACK acknowledges a segment that is smaller than MSS bytes long, cwnd is increased by MSS bytes.

- Does Slow Start increment slowly? Not really. In fact, the increase of cwnd can be exponential.

Slow Start Example

- 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 $\text{cwnd} > \text{ssthresh}$
**Congestion Avoidance**

- Congestion avoidance phase is started if cwnd has reached the slow-start threshold value.

- If \( \text{cwnd} \geq \text{ssthresh} \) then each time an ACK is received, increment cwnd as follows:
  - \( \text{cwnd} = \text{cwnd} + \text{MSS} \times \frac{\text{MSS}}{\text{cwnd}} + \frac{\text{segsize}}{8} \)

- So \( \text{cwnd} \) is increased by one segment (=MSS bytes) only if all segments have been acknowledged.

**Slow Start / Congestion Avoidance**

- Here we give a more accurate version than in our earlier discussion of Slow Start:

  ```
  \textbf{If} \quad \text{cwnd} \leq \text{ssthresh} \quad \textbf{then} \quad \\
  \text{Each time an Ack is received:} \quad \\
  \quad \text{cwnd} = \text{cwnd} + \text{MSS} \\
  \quad \textbf{else} \quad \text{/* cwnd > ssthresh */} \quad \\
  \quad \text{Each time an Ack is received:} \quad \\
  \quad \quad \text{cwnd} = \text{cwnd} + \text{MSS} \times \frac{\text{MSS}}{\text{cwnd}} + \frac{\text{segsize}}{8} \\
  \textbf{endif}
  ```
Example of Slow Start/Congestion Avoidance

Assume that \( ssthresh = 8 \)

Responses to Congestion

- Most often, a packet loss in a network is due to an overflow at a congested router (rather than due to a transmission error)

- So, TCP assumes there is congestion if it detects a packet loss

- A TCP sender can detect lost packets via:
  - Timeout of a retransmission timer
  - Receipt of a duplicate ACK

- When TCP assumes that a packet loss is caused by congestion and reduces the size of the sending window
Responses to Congestion

- There are different algorithms developed for TCP to respond to congestion
  - Tahoe TCP - the basic algorithm
  - Reno TCP - Tahoe + fast recovery
  - New Reno TCP - improvement to Reno

- and many more:
  - Vegas TCP
  - SACK TCP
  - Fack TCP

TCP Tahoe

- Congestion is assumed if sender has timeout or receipt of duplicate ACK

- Each time when congestion occurs,
  - cwnd is reset to one:
    \[ \text{cwnd} = 1 \]
  - ssthresh is set to half the current size of the congestion window:
    \[ \text{ssthresh} = \text{cwnd} / 2 \]
  - and slow-start is entered
Slow Start / Congestion Avoidance

- A typical plot of cwnd for a TCP connection (MSS = 1500 bytes) with TCP Tahoe:

TCP Reno

- Problem with Tahoe: If a segment is lost, there is a long wait until timeout
- Reno adds a fast retransmit and fast recovery mechanism

- Upon receiving 3 duplicate Acks
  - retransmit the presumed lost segment (“fast retransmit”)
  - Set $cwnd = cwnd / 2$
  - Set $ssthresh = cwnd$

  - So, do not enter slow-start, but enter congestion avoidance (“fast recovery”)
TCP New Reno

- Similar to Reno TCP agent, but modifies the action taken when receiving new ACKS.
- In order to exit fast recovery, the sender must receive an ACK for the highest sequence number sent. Thus, new "partial ACKs" (those which represent new ACKs but do not represent an ACK for all outstanding data) do not deflate the window.