(V.1.2, 30 November 2017)

Practical Exercises in Computer Networks and Distributed Systems

TCP Exerciser (1/2, WIP)

This practical aims to experiment with the SWS avoidance algorithm and to serve as the basis for other TCP algorithms such as Nagle's, Slow Start, etc.

Exercises

Perform the next steps to download, compile and experiment with the provided C/S programs:

1. Download the Netbeans project from http://paloalto.unileon.es/ds/Lab/TCP/TCP-Exerciser.zip

2. Either uncompress the source tree or directly import the project from the Netbeans IDE by choosing "Project | Import from zip file". If at some point in the development of the practical you want to modify the provided Java Swing GUI, then you should import the project, otherwise, editing the relevant source code and compiling from the command line will suffice, that is, if you don't need to modify the GUI.

    a. Assuming you currently are at directory /Users/estudiante and that you want to build the code from the command line, cd to the directory resulting from uncompressing the zip file (TCPZeroAWSTester/src), and then compile the source tree:

    ```
    $ cd /Users/estudiante/TCPZeroAWSTester/src

    $ javac ./gui/NewJFrame.java
    ```

    and compile the rest of the project:

    ```
    $ javac ./sockets/*.java
    ```

3. First, run the server from the command line:

    ```
    $ cd /Users/estudiante/TCPZeroAWSTester/src/

    $ java gui.NewJFrame
    ```

    Shortly afterwards, the following Window will appear on your screen:

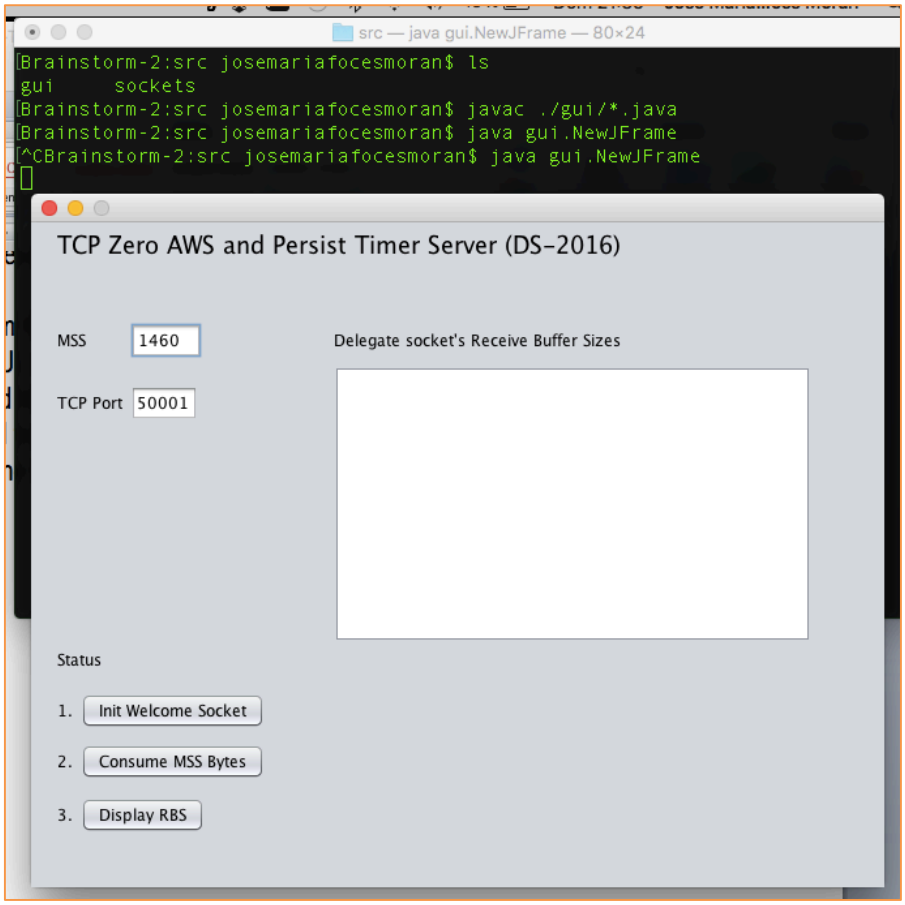

Figure 1. GUI of the TCP flow control tester program

- The "Init Welcome Socket" button creates a Welcome Socket that will be contacted by the client program to create the TCP connection that we want to test

- Once the client connects with this server, it will send 10 large blocks of bytes, giving you the opportunity to analyze the protocol with Wireshark. Every time you press "Consume MSS Bytes", the server will consume MSS Bytes from its Receive Buffer, thereby giving you a chance to observe the evolution of the Advertised Window Size field included in the response ACKs

- Button "Display RBS" prints the current size of the ReceiveBuffer, if the underlying API/Operating System allow

- When running Wireshark for this practical, you might find the following display filters handy:

  ; Display frames containing TCP segments to/from TCP port 50001
  tcp.port==50001

  ; Display frames containing TCP segments from TCP port 50001 (From server, receiver)
  tcp.srcport==50001

; Display frames containing TCP segments to TCP port 50001 (From client, transmitter)
tcp.dstport==50001

4. Start the server program and press no button; then, start the client:

```
$ java sockets.ZeroWindowClient <server ip addr> 50001 1500000
```

   a. Observe the 3-way handshake if any and explain what you see on the resulting Wireshark trace

   b. Kill the server if it is running now, then restart it and press the "Init Welcome Socket" button. Now, run the client program and explain the resulting trace of the 3-way handshake which you can watch in Wireshark.

   c. Observe the TCP options set in the 3-way handshake and explain their purpose

   d. Is the Window Scale option set? What's its value, that is, the multiplier?

   e. Are there any timestamps supported by the involved TCPs?

   f. What's the MSS? Is the MSS the same in both directions?

   g. Are there any other relevant TCP options? SACK, for example?

5. Identify the following types of frames which must appear in the experiment traces, in each case, explain their semantics:

   a. TCP zero window probe

   b. TCP zero window ack

   c. Window Full

   d. Window update

   e. Keep alive

6. Observe the segments exchanged by client and server when the two, independently of the other, closes the connection, first client and afterwards the server.

7. Kill the server program and start it again, then, press the "Init Welcome Socket" button so the server program creates a Welcome Socket on TCP port 50001 (See the upper textbox in the Server X-Window). Welcome Socket is created, and we are ready for executing the client.

- Has the correct Welcome Socket been created in your system? Check it.

8. Run the client, preferably from a host other than the one in which the server is running

9. $ pwd
   /Users/estudiante/TCPZeroAWSTester/src/

   $ java sockets.ZeroWindowClient 192.168.2.105 50001 155000

   The client sends 155000 Bytes in ten blocks to the server, more or less guaranteeing filling up its receive buffer in a few seconds. You can check this by monitoring the traffic with Wireshark

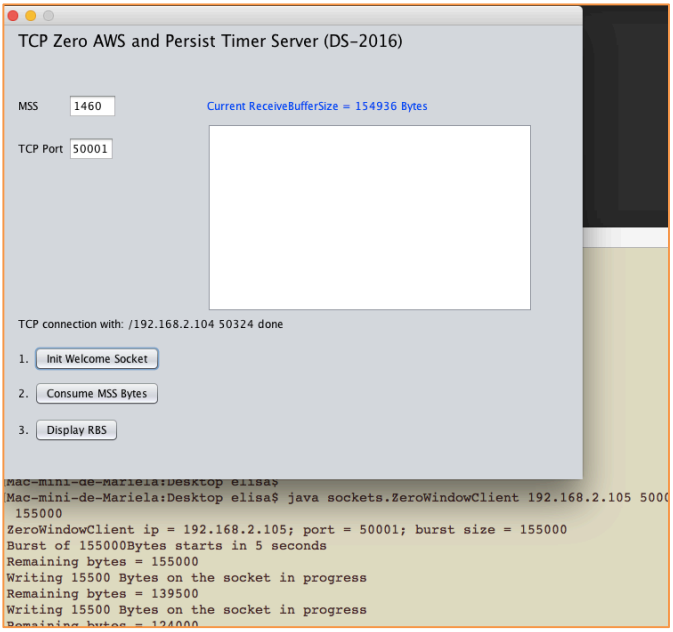

Figure 2. TCP connection completed with client at IP 192.168.2.104

10. While the client is transferring that many bytes to the server, it consumes no bytes from its Receive Buffer, which is the primary cause the window fills up.

    a. Observe whether the receiver's (server) window has been filled up.

    b. Press the "Consume MSS Bytes" button to have the server consume one full MSS bytes length.

    c. Observe the first Window Update form the server advertising a non-zero window, you may have to press the consume button more times: as you press the button, observe the server's reaction, if any.

    d. In accordance with the SWS avoidance algorithm proper of TCP, you should observe no AWS value

smaller than an MSS (1460 bytes if you are using Wi-Fi or Ethernet).

- If you see some value of receiver's AWS smaller than MSS, that is, probably an indication of an effect known as Window Shrinkage which is due to the receiver's window being sized according to the Window Size multiplier established in the 3-way handshake via TCP options.
- When Window Shrinkage occurs, the receiver opts for avoiding it announcing an AWS smaller than an MSS which might cause Silly Window Syndrome.

e. Did you see any receiver's AWS values smaller than one MSS?

11. As the client sends the blocks to the server, observe the AWS from the receiver and explain whether or not it monotonically decreases, eventually to zero (When the window becomes full). More specifically, we'd like you to tell whether some *unexpected* growth in AWS took place.

    Depending on the network stack implementation (Linux, OS-X, etc), you might observe that the API implementation enlarges the Receive Buffer Size because the operating system's state can afford it and it sure might be good for the socket's receive buffer since the application is not consuming. Operating systems of today implement a technology known as TCP auto-tuning which allows TCP to dynamically resize a socket's buffer according to the operating system memory subsystem's state and the buffer demand. These TCPs usually resize the buffer as the connection evolves.

    Under Linux, you must observe the effect of TCP auto-tuning in the traces resulting from this experiment. Auto-tuning causes an *unexpected* growth of the AWS in situations where the receiver is not consuming and the sender is continually transmitting.

    a. Did you see an unexpected, transient growth in the receiver's AWS?

    b. If you actually saw that unexpected growth in receiver's AWS, it's because your API implementation decided to enlarge the receiver buffer. The Java sockets API allows us to set a socket's receive buffer size (Check the sockets.AppProtocol.java class in this practical's source code).

    c. When the transfers finish and you have pressed the "Consume MSS" button several times, you can, if you wish, printout the values of the receiver's receive buffer size which were recorded as the transfers took place.

    d. Was the receive buffer size constant as the transfers took place?

12. Select one of the ACKs transmitted by the Server, for example, the 10th ACK and respond to the following questions:

    a. If the MSS established in the 3-way handshake is 1460 bytes, what do you think

that accounts for the difference between this MSS and the size of the frame encapsulating the segment which is 1514 bytes?

b. Identify the segments spanned by the ACK sequence number of the ACK segment that you chose.

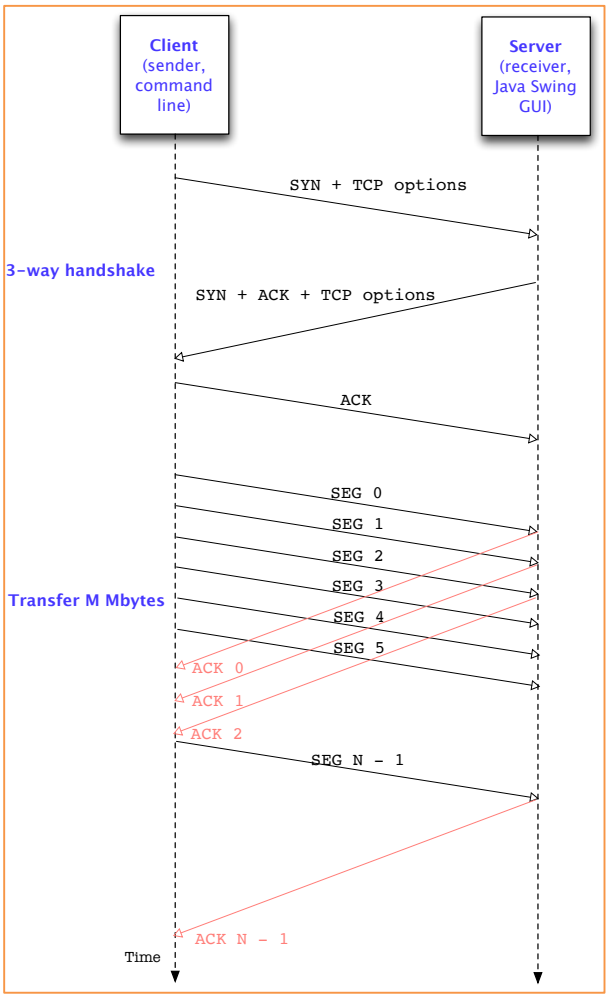

Figure 3. TCP connection and data transfers of several segments