(v 3.1.1, 18 Oct 2019)

Lab Practices on Computer Networks and Distributed Systems

TCP Sliding Window algorithms (C and Java, WIP)

The TCP algorithms that we have studied in the lectures comprise Nagle algorithm, the Sliding Window algorithm, the flow control algorithm and lastly, the congestion control algorithm. In this practice we develop a few experiments that illustrate the Sliding Window Algorithm in a practical way. We set out to create an appropriate test context for each algorithm by sending messages on the socket interface from the server, written in Java and then some of the clients, written in C and others written in Java. The test context will help us watch the significant TCP messages interchanged by the server and the client, their structure and their meaning. Enabling ourselves to watch those messages entails our using a network analyzer, in our case the impervious, always toiling Wireshark. Eventually, we are seeking to foster our practical grasp about TCP.

Setting up a simple TCP exerciser application

The TCP exerciser is a client/server pair. Both programs, the client and the server are written in Java; the server deploys a GUI (Graphical User Interface) based on Java Swing that was designed by using the Netbeans IDE. Using the exerciser doesn't require your installing or using Netbeans, only if you wish to modify the server program's GUI will you need to run Netbeans and then have the provided zip file imported into it; otherwise, you simply edit, recompile the sources and run the resulting executable java class file.

The C/S pair allows the client to send large bursts of information continually with the objective of filling up the server's receive buffer, thereby allowing you to observe TCP behavior when the buffer is nearly full and when it is finally full. Interacting with the server's GUI, the user can consume blocks of received information in sizes of 1 MSS each time the button is pressed. Having the server process consume from the TCP connection's receive buffer will free some space in the receive buffer and we will be able to observe the messages sent and received by TCP.

If you do wish to modify the server's GUI, take the following steps to download, compile and run the provided C/S programs.

1. Download the *zipped* Netbeans project from:

   http://paloalto.unileon.es/ds/Lab/TCP/TCPZeroAWS-v2.zip

   This Java project contains the source code to the client and server programs which will serve us for sending and receiving and analyzing some interesting TCP facets. The server program deploys a Java Swing GUI (See Figure 1, below). The client program is not GUI-based.

2. Access Netbeans "File | Import Project | From zip" and proceed to import the zip file that we downloaded above. When it finishes loading, you'll be able to edit the GUI layout by using the GUI editor.

3. When you finish your modifications, recompile and run the class by pressing the Netbeans arrow button or maybe by accessing the relevant menus.

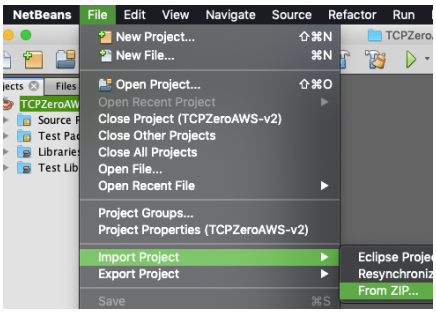

Figure 1. Importing a Netbeans project

If you **do not wish** to modify the server's GUI, take the following steps to download, compile and run the provided C/S programs.

1. Download the *zipped* Netbeans project from:

   http://paloalto.unileon.es/ds/Lab/TCP/TCPZeroAWS-v2.zip

2. Decompress the ZIP archive and two folders will result: sockets and gui. The client and the server program are stored into the sockets folder; the GUI is stored into the gui folder.

3. Assuming your current directory is /Users/estudiante and that you want to build the code from the command line, cd to the directory resulting from uncompressing the zip file (TCP-Exerciser-v2/src), and then compile the GUI (Graphical User Interface) server source tree:

```
$ cd /Users/estudiante/TCP-Exerciser-v2/src

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

   Finally, compile the rest of the project (The server and the client and other utilities under directory …/sockets/):

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

4. You start the server (GUI) first by typing the following command on a terminal shell:

```
$ cd /Users/estudiante/TCP-Exerciser-v2/src/

$ java gui.NewJFrame
```

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

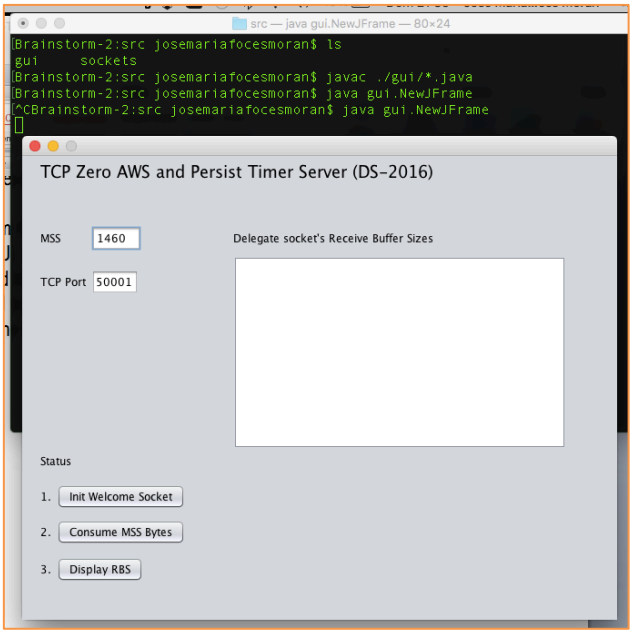

Figure 2. GUI of the TCP exerciser program

- The "Init Welcome Socket" button creates a Welcome Socket on port 50001, the one 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 the number of blocks specified by the user on the command line. The block size is also specified by the user on the client's command line. These transmissions from the client allow you to analyze with Wireshark the TCP messages that transport the information between Client and Server.

- Every time you press the button titled "Consume MSS Bytes", the server will consume MSS Bytes from its Receive Buffer, thereby giving you, in real time, a chance to observe the evolution of the Advertised Window Size field included in the ACKs[1] sent by the server. As the client sends more and more blocks of information, the server stores them in the Server's Receive Buffer. The information received by the server remains in the Receive Buffer until the application consumes it. As the application consumes information from the Receive Buffer, the available room in the buffer grows and the server informs the client about its receive buffer size in the ACK segments that it sends back. In each segment sent by the server to the client, the field used for reporting the available space is known as Advertised Window Size which is interpreted along with the Window Scale option factor that was announced in the 3-way handshake.

- Button "Display RBS" prints the current size of the server's receive buffer. This size is set by the API when the delegate socket is created and, in the current versions of Linux, it can be enlarged when the receiver (In this case, the server) is handed a huge amount of information or when information is piling up in the receive buffer without the server application consuming any

[1] ACK means acknowledgement, and its plural, we assume that is ACKs. Each ACK is a TCP segment that has flag ACK set and which ACK SN field is significant.

information from it. There exists one method in class Socket that permits changing the size of this buffer, however, it is nothing but a mere hint to the operating system to set some size, *i.e.,* if that is acceptable for the system.

- 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

Watching the sliding window protocol with the Exerciser

If you already compiled the sources, you can start the server program, now:

1. Start the server program and press no button yet;

   ```
   $ java gui.NewJFrame
   ```

2. Press the "Init Welcome Socket" button and check that the Welcome Socket has been created by executing the following command on a terminal:

   ```
   $ netstat --tcp -l -n | more
   ```

   A Welcome Socket must have been created by this time which local socket port is 50001

3. Does the server's creation of the Welcome Socket generate any IP traffic? Check this with Wireshark.

4. Keep your Wireshark running and have it start a new packet capture, maybe on the Client, and another on the Server. Now, we will start the client program by typing the following command. The client will quickly attempt to connect with the server. Since the Welcome Socket hasn't been created yet, the connection will not be completed (The 3 steps involved in the 3-way handshake)

   ```
   $ java sockets.ZeroWindowClient 192.168.1.254 50001 5 50000
   ```

   a. Observe the 3-way handshake if any and explain what you see on the resulting Wireshark trace. Observe what the client does and explain it, thoroughly.

   b. Kill the server if it is running now, then restart it and, this time do press the "Init Welcome Socket" button. Now, start the client program and explain the resulting Wireshark trace containing the segments that comprise the 3-way handshake. *Please, press no other button until we indicate you*

to do so.

c. If the connection is successful, observe the TCP options activated in the 3-way handshake segments and recall their purpose. Consult RFC 793 if necessary to find out the meaning of each of the activated options that you have observed.

d. Is the Window Scale option set? What's its value, that is, the multiplier? Could you justify the existence of the Window Scale option? Hint: Search your lecture notes for the concept of Delay x Bandwidth product, which we usually refer to as the 2BD product.

e. Are there any timestamps supported by the involved TCPs? What happens if the client does not support timestamps and the server does? What about the converse situation?

f. What's the MSS in each direction? Is the MSS the same in both directions? Should it be? Accuratley explain the meaning and the significance of the MSS option announced by the client and by the server.

g. Are there any other relevant TCP options? SACK, for example? What's SACK? (We did not explain this option thoroughly in the lectures, we only mentioned it).

5. Skim the semantics of each of the following segment types as labelled by Wireshark. Those messages will appear in question no. 6 below and the ensuing ones. Skim the list before you do question 6.

   The names assigned by Wireshark to the different types of segments that can appear on a TCP connection, heavily depends on the Wireshark version, thus, a concrete segment may be given different names by the different versions of Wireshark). Try to explain the meaning of each type of TCP segment attributed by Wireshark:

   a. **TCP ZeroWindowProbe:** When the receiver's window has been announced as AWS=0, the sender, repeatedly sends a segment which purpose consists of priming the receiver to send an update about its window size, just in case it might have grown. It contains one byte of data which sequence number is one beyond the window 0. The receiver will *not* acknowledge this byte.

   b. **TCP ZeroWindowProbeAck**: When a TCP receiver must respond to the preceding TCP segment type (TCP ZeroWindowProbe) it does so by sending a ZeroWindowProbeAck. It contains an AWS=0 alongside with an ACK corresponding to the next byte expected, as ususal.

   c. **TCP Window Full:** When the sender (In this case the client) fills up the server's AWS, *i.e.,* it sends a a number of bytes equal to the effective remaining window size, Wireshark informs us with the segment label "TCP Window Full". Find this type of segment in the trace resulting from the current experiment.

   d. **TCP zero window:** When the receiver for the first time announces a AWS=0, Wireshark informs us with this label. Note that the actual remaining space in the receiver buffer might not be zero, nevertheless, the receiver announces AWS=0 in an attempt to avoid the Silly Window Syndrome in case the transmitter has not activated Nagle's algorithm.

e. **TCP Window Update:** This segment is sent by a receiver which window has grown beyond a certain limit after being zero or close to zero. It may contain no data.

f. **Keep Alive:** When connections are idle the Keep Alive timer sends keep alive segments.

6. If your client, by this time sent enough bytes to cause the receive buffer to fill up, then, you must have seen the Advertised Window Size zero, AWS=0 at the receiver and must have seen the continual exchange of TCP ZeroWindowProbe/ZeroWindowProbeAck segments.

   a. Measure the time between each successive TCP ZeroWindowProbe sent by the sender when AWS=0 on the receiver.

   b. Do these time intervals between these successive TCP ZeroWindowProbe segments follow an exponential pattern? Do they have a maximum?

7. Now, we wish to free some space from the Server's receive buffer, to that end, press repeatedly the Consume MSS button, which will cause your server (The receiver) to make room for 1 MSS in its receive buffer which eventually, if you keep pressing the button, will cause the server TCP to update its AWS reflecting larger values. Observe the growth of the receiver AWS and how the exchange of TCP ZeroWindowProbe/ZeroWindowProbeAck segments has ceased altogether. If you consume all the data sent by the client, it will sleep for 3 minutes to give you a chance to observe, in real time, how the client closes the connection.

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

   b. Consult the TCP state diagram and locate the states and the transitions that comprise the TCP close operation that you saw on Wireshark.

8. 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). When the welcome socket is created, like we did above, we are ready to start the client program.

   a. Has the correct Welcome Socket been created in your system? Check it by finding a socket in the LISTEN state in the listing produced by the following command:

   ```
   $ netstat –p tcp –n | more
   ```

9. Run the client, preferably from a host other than the one in which the server is running and change directories to the project's base directory and verify it:

   $ pwd
   /Users/estudiante/TCPZeroAWSTester/src/

Assume the server is running on IP address 192.168.1.254, type this command:

```
$ java sockets.ZeroWindowClient  192.168.1.254  50001 10  30000
```

The client sends 10 blocks of 30000 Bytes each to the server, more or less guaranteeing that its receive buffer will fill up in a few seconds. Again, watch this by monitoring the traffic with Wireshark and continue to the next question.

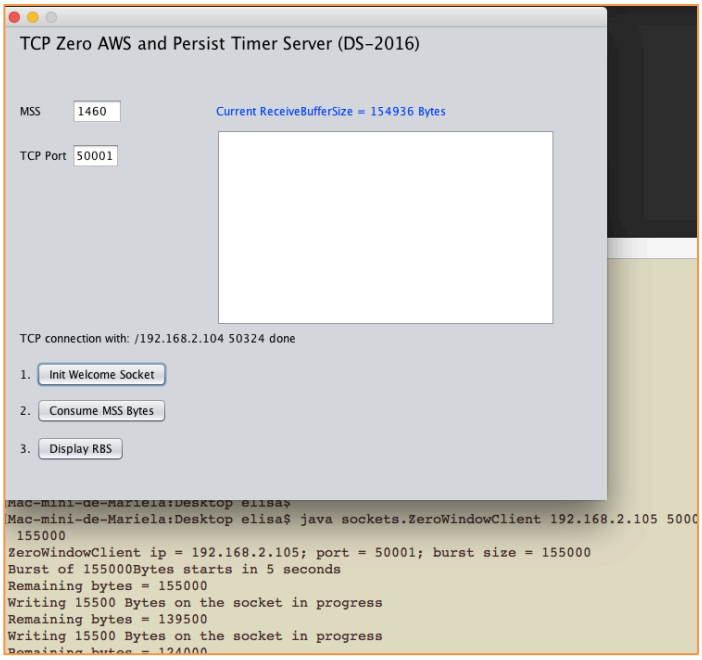

Figure 3. TCP connection established with client at IP 192.168.2.104[2]

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 that its 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 from 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). The behavior described below is highly dependent on the specific version of the protocol stack and the operating system type:

        - 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

[2] Recall, the concrete IP addresses used in the Lab will belong to the CIDR block 192.168.1/24

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 or maybe 1516 bytes?

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

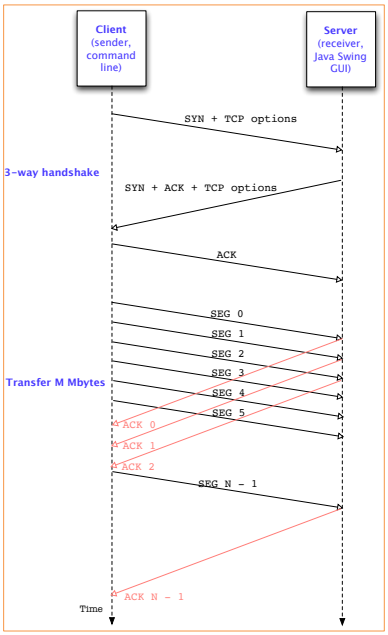

Figure 4. TCP connection and data transfers of several segments

13. Stop the current Wireshark trace in which we have captured the segments resulting after the Client and the Server have interacted a few times. Now, save the trace to a file having an extension of pcapng. Later, you'll be able to reload that trace into Wireshark and study it as much as you need as in the following questions which aim to illustrate the most fundamental features of TCP. In each of the answers that you provide to the following questions, include a small screen shot of the relevant Wireshark area (A frame, a frame listing, some specific fields of a frame, etc):

    a. Put an example of a segment sent by the client and containing data. Indicate its frame number and clearly write down the SNs covered by the segment and using the [start, finish] notation that we have used in the lectures. Calculate by hand the smallest ACK that will acknowledge the preceding SN block.

    b. Locate that ACK segment and include a screen shot of it, right here below. Recall that Wireshark allows you to obtain this info directly by unfolding the TCP subtree of this frame. Was your calculation correct?

    c. The ACK segment that you found is a quick ack, or is it a delack or maybe a stretch ack? Explain your answer to this question.

    d. What was the Rtt in the preceding case? Now, compare that Rtt with the Rtt determined in th SYN+ACK response from the server and tell us whether those two values are equal or not. Provide an explanation to your answer.

    e. Observe the dynamic behavior of TCP, have you observed the SS (Slow Start) state in which a newly formed connection is started?

    f. Did you observe some lost segment, and consequently, either a 3-DUP or a TO?

Lab Practices on Computer Networks and Distributed Systems

Stream Sockets and Client/Server (C language, WIP)

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

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 frame encapsulating the segment which length is 1514 bytes?

16. In this exercise we want Wireshark to generate an RTT and a Throughput graphs corresponding to the trace of an http download of a big file. The URL of the file to be downloaded is: http://paloalto.unileon.es/big.zip. Before starting the download trace, apply a convenient Wireshark display filter `((ip.addr eq 192.168.99.130 and ip.addr eq 192.168.99.99) and (tcp.port eq 49199 and tcp.port eq 80))`, then, start the trace and the download. When the operation finishes, request a RTT graph and a Throughput Graph from Wireshark statistics menu. Include them here alongside a brief interpretation of both and the number of Mbps attained.

17. In this exercise we wish to devise an experiment to discover whether ssh and scp activate Nagle's algorithm on the sockets. I suggest you start a Wireshark capture and a new session of ssh to student@paloalto.unileon.es, then, execute a few Unix commands (e.g.: ls –l, who, date). Finish the session and stop the capture, then, click with right mouse button on the first first frame and activate Conversation Filter | TCP, this will limit the frames displayed to those belonging to the TCP connection established. Use Wireshark statistics (Summary, Flow Graph and others) to establish the number of bytes exchanged and the number of segments exchanged.

Now, repeat the same process, but this time we want to analyze the segments exchanged in a scp session, for example, you can download the file student@paloalto.unileon.es:test.txt. How would you tell whether ssh and scp use the Nagle's alg. or not?