Lab #8: TCP Analysis

Previous reading:

• Unit 4 of the course: Transport Layer. (Kurose: chapter 3)

• Lab #1 of the course: Introduction to Wireshark program.

Lab #8 aim

Lab #8 aim is that the student will be able to understand the workings of TCP transport protocol, studied in the theoretical sessions, through network traffic analysis with the Wireshark analyser.

Introduction

In Lab #8, we are interested in delving into the study of the operation and implementation of TCP without worrying about the data contained in the segments transferred. Therefore, we will indicate to Wireshark that we only want to visualize the TCP segments. The reason is to focus only on the transport information, not on the interpretation that Wireshark can make of the application layer transported by the TCP segments. If you remember the first practices of the course, we analysed HTTP traffic (as we are going to do now) in a comfortable way since the analyser interpreted the data of the TCP segments and indicated on the screen if it was performing, for example, a GET, instead of having to look for that method in the data area of the TCP segment. Now we are not interested in seeing the GET, POST, etc. That is, we are not interested in interpreting the protocol that is used in the data area of a segment at the transport level, but only the format of the TCP segments and the interpretation of their fields. For this reason, in the option **Analyse → Enabled Protocols, we will remove the mark of the HTTP protocol**. In this way, although HTTP is transported in a TCP segment, the analyser will not interpret the HTTP protocol and will only show the TCP segment that transports it.

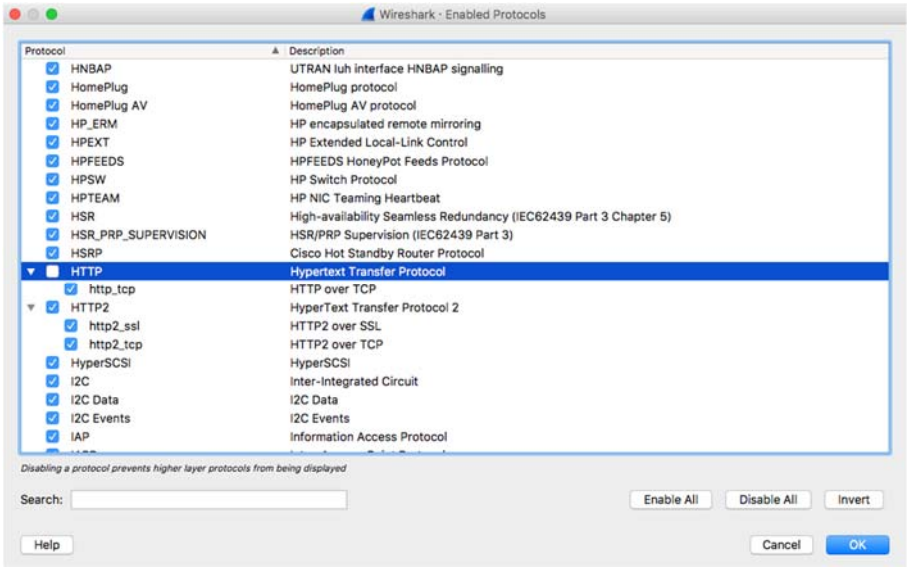

Figure 1. Analyse option -> Enabled Protocols. Uncheck HTTP

Exercise 1:

Make a capture of the traffic between your browser and the UPV web server (http://www.upv.es) using the Wireshark It may be useful to set a filter to port 80 (from the Capture → Options menu or tcp.port == 80 from the display filter) to facilitate the interpretation of the data.

a) Analyse the connection establishment. Identify the TCP 3-way handshake (SYN,SYN-ACK, ACK) segments. What MSS is chosen to carry out the transfer? What other options establish each end system of the connection?

b) Determines the initial sequence numbers of each end system of the connection. Notice that Wireshark replace the real sequence number by a relative number to make a more comfortable tracking of segments (click on the relative sequence number and you will see the real number that appears in the segment transferred in packets byte pane (the lower window of the main window -contained in hexadecimal). Similarly, by selecting a TCP segment, with the right mouse button (Protocol preferences) we can indicate that we do not want to use relative numbers (although it is more convenient to continue using the relative numbers in the rest of this Lab). (Figure 2)

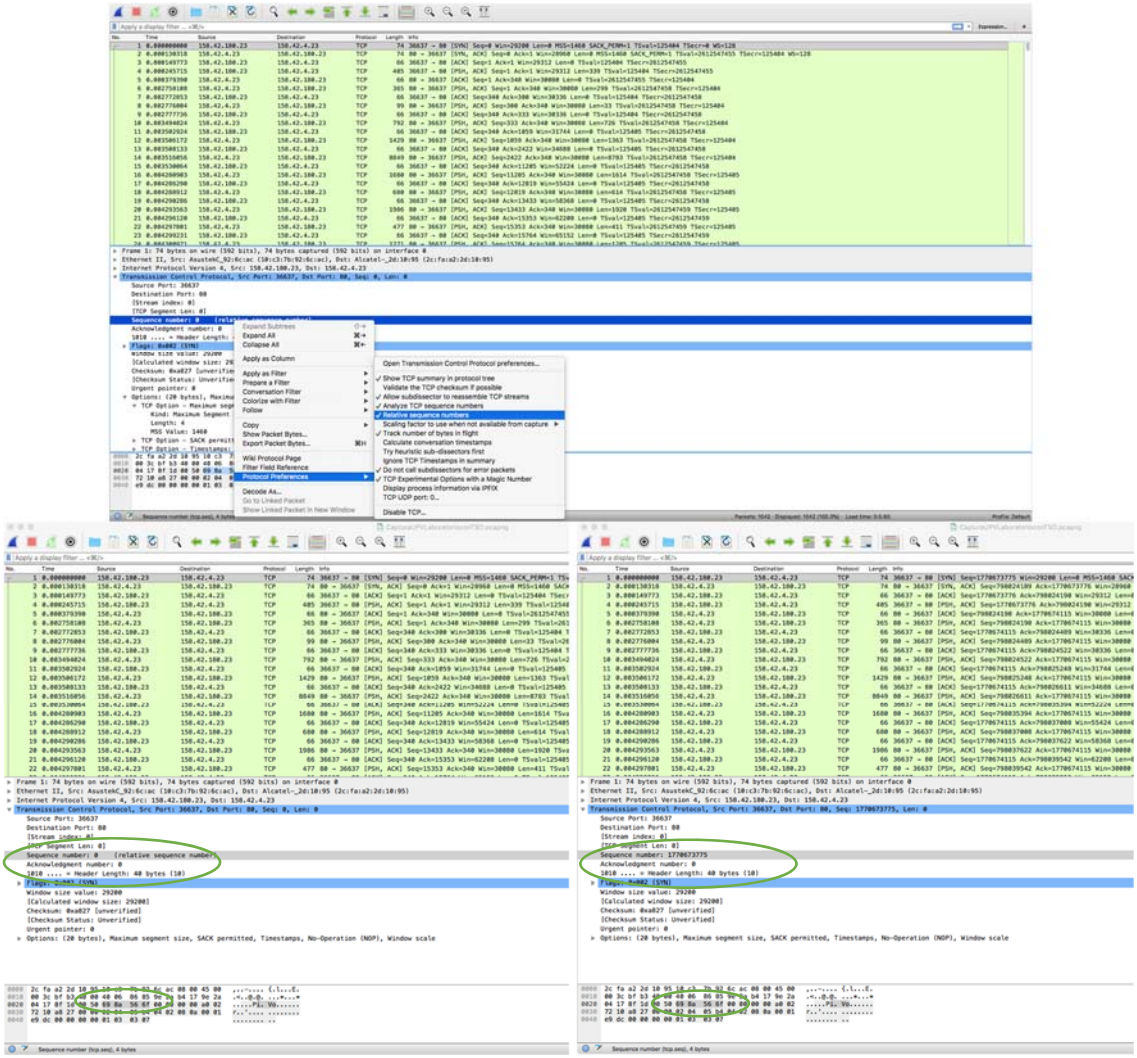

Figure 2. Sequence numbers: relative / absolute

c) Now we want to analyse the connection closure. First we have to select the first TCP flow (the HTML file download flow). You can do it selecting the GET / HTTP/1.1 segment in the packet list and then select the Follow TCP Stream menu item from **Analyze-> Follow TCP Stream** menu (or use the context menu in the packet list). Wireshark will set an appropriate display filter and pop up a window with all the data from the TCP stream laid out in order. Close this window that shows the TCP stream and scroll to the end of the main screen (packet list pane), where you will see the end of the connection. Who takes the initiative? Is it a closure in three or four phases?

d) Next, execute the option of the **Statistics → Flow Graph** menu. In the new window select the option Show to Display packets (Show: Displayed packets) (Figure 3). What can you see there?

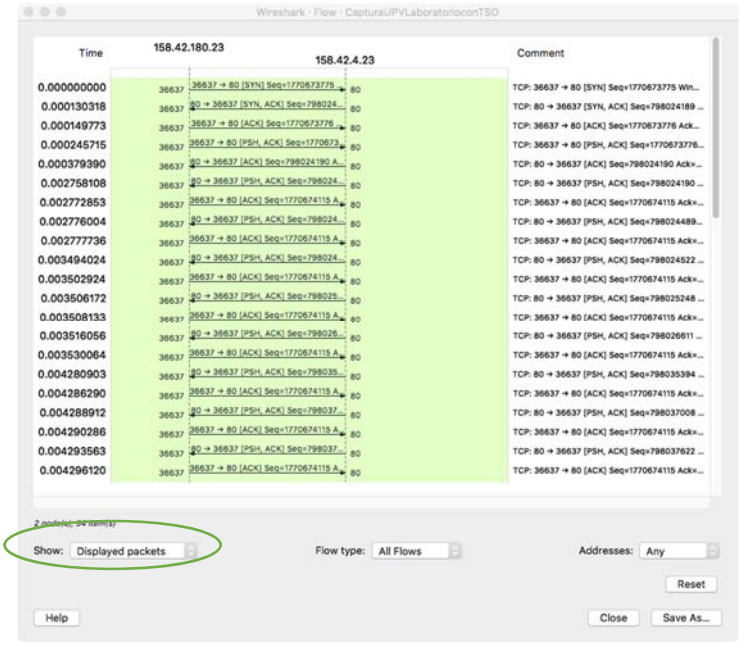

Figura 1.OpciÓn Statistics->Flow Graph

e) Next, analyses the evolution of the acknowledgments (ACK). For this, look at the segments that the web server sends, look its length and its sequence number and observe when the client (your host) sends the ACKs and until which byte it is acknowledged. This can be seen both in the *Statistics → Flow Graph* window that you have opened in the previous section, and in the main window with the selected flow. Notice that when you mark a segment in the *Statistics → Flow Graph* window, it is also selected in the main window. (Note: to know which segment of the server is acknowledged in a client acknowledgment segment, analyse the sequence number of the sender and add the segment length, then look for the ACK that carries that number).

Is there an ACK every two TCP segments? Why? Try to find an explanation (Think about the convenience of using delayed ACK during the first RTT when the slow start algorithm is running).

f) To expand the previous section, having selected the first flow as in the previous sections and selecting a segment sent by the UPV web server, execute the option *Statistics-> TCP Stream Graphs -> Time Sequence (tcptrace)*. You should obtain a result similar to the one shown in the following figure:

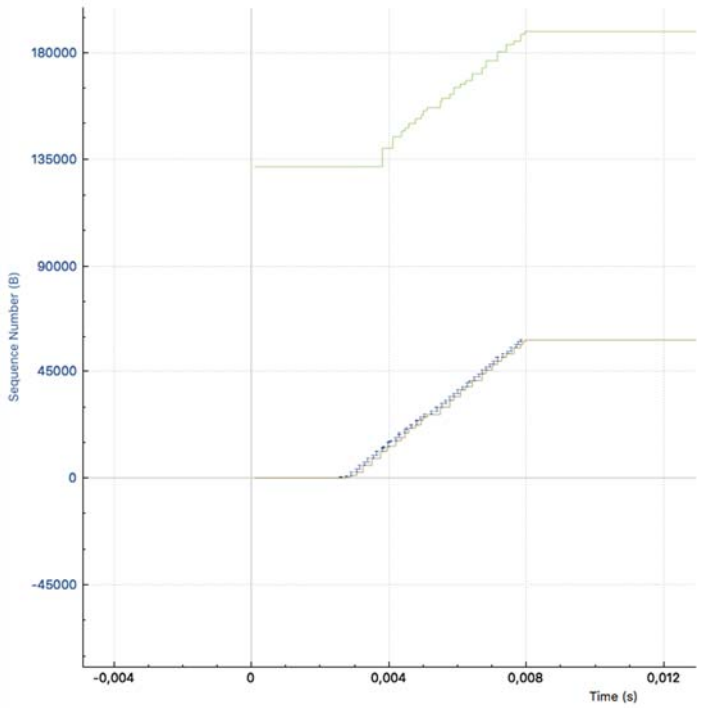

Figure 4. Statistics-> TCP Stream Graphs -> Time Sequence (tcptrace)

In Figure 4 (or the one you obtain as it depends on the transfer you just made between your host and the UPV web server and the TCP libraries implemented in the operating system) and its more detailed extension in Figure 3, the x-axis shows the time and the y-axis the sequence number of each of the segments sent.
As you can see in this figure, you can see three lines or graphs: the top line corresponds to the reception window (that is, up to what byte of the data flow your host could receive). The second one, which is represented by small vertical lines, indicates each of the TCP segments (initial byte and final byte of the segment) that is sent. The longer it is, it represents that this segment is sending more quantity than the others (figure 5 shows this concept better).
Finally, the bottom line represents the sending of the ACKs. The difference between the ACK line and the line of sending segment determines the amount of "in flight" data not recognized yet. Figure 5 shows a zoom similar to what you will have obtained and it is focused on the two lower lines: sending segments and ACK.

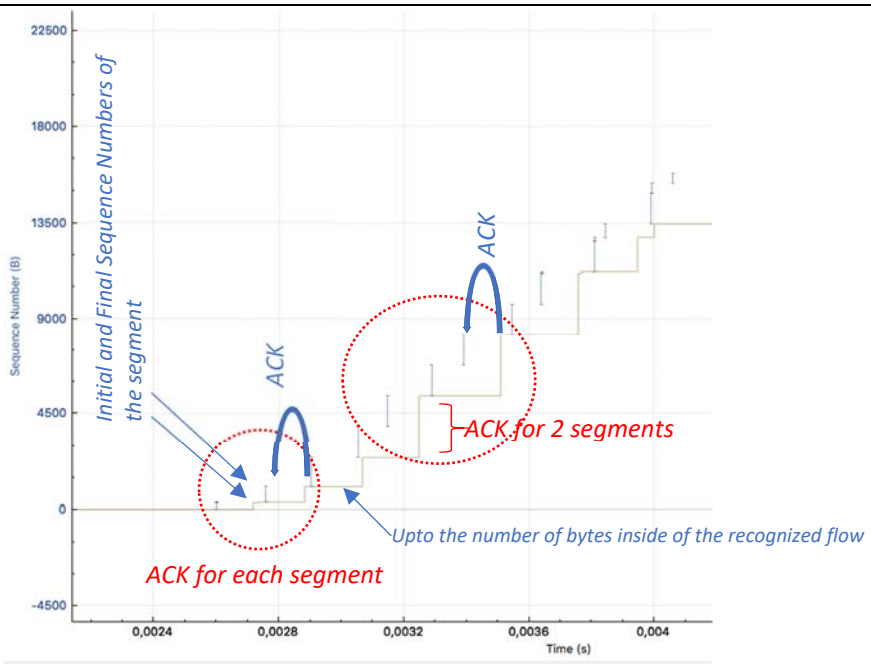

Figure 5. Statistics-> TCP Stream Graphs -> Time Sequence (tcptrace). Zoom of Figure 4

As shown in Figure 5 (your capture may have generated a different graph), most of the segments in this case are of similar size (MSS) and the sending of ACKs is in some cases for each segment and, in others, for each 2.

Now interpret what you are seeing in Wireshark regarding the capture you have made:

• Regarding the reception window, do you think that it will generate a problem? Is there enough space?
• Regarding the sending of ACK, do you reaffirm your conclusions of section e)? After how many segments are sent the ACKs? What differences do you see with the Figure 3? Which can be the causes?
• Based on the captured packets in the packet list pane of Wireshark (main window) or intuitively in the previous graph, observe the evolution of the TCP segments size: Do you see any length greater than the MSS? If yes, since it depends on the configuration of the TCP libraries of each machine, the reason is as follows:

Generic segmentation offload: (from https://en.wikipedia.org/wiki/Large_send_offload)

As the transmission speed of the networks increases, the number of segments that can be sent or received per unit of time increase. Thus, since the CPU of the computer will receive in the same time a greater number of segments, it will dedicate more time to the communication tasks.
To avoid this overload of the CPU and allow it to be dedicated to other things, the operating systems allow, if the network interface supports it, to activate the transmission and / or reception Offload. What does this mean? Basically, the TCP libraries of the operating system, instead of passing segments of maximum size (MSS) to the network card, it can pass much larger segments and the network card is responsible for fragmenting them when transmitting and reassembling them when receiving. To determine if a system, for example, Linux, has the offload mode enabled (to transmit, receive or both), execute the command: `ethtool -k -show-offload` *from a terminal. Also, the* `ethtool -k eth0` *command shows different parameters of the network card, including the offload mode. You can verify that Linux installation of the Lab's computers has enabled the reception offload.*

g) Select any of the frames sent from the web server to your browser and execute the option `Statistics-> TCP Stream Graph → Round Trip`. What information are we seeing? What is the approximate RTT of this connection?

h) Repeat section c) for a connection established through your browser to the URL http://www.redes.upv.es. Do you see the closure of the connection? Why? If not, repeat it but this time closing the browser before finishing the capture with the analyser. Justify what you are seeing.

Exercise 2:

Make the capture of the traffic generated when your browser requests a web page to the web server of the University of Valencia (http://www.uv.es:81) on port 81 using the Wireshark analyser. Be careful if you have an active filter and set it to port 81. Analyse the capture made. How is it indicated that there is not a service in that port? How is the connection closed? What is the answer of your browser? Does your browser try the connection again? If yes, how many times?

Repeat it, but now make the connection with the server of our university: http://www.upv.es:81. Does it happen the same as in the previous case? What do you think is the reasons for this behaviour?

Exercise 3:

In this exercise we are going to dig deeper into the implementation of the congestion control algorithm (slow start mainly). Thus, we will analyse the number of segments that can be transmitted and how they are recognized to increase the windows of congestion and transmission. Also, we will see some problem caused by retransmissions and the use of selective acknowledments. To do this, download the file *CapturaPráctica.pcapng* that you can find in *Poliformat->Practicas* and load it in Wireshark.

The file contains the capture of the traffic generated by a client that uploads some information to a server. So we can see how many MSS the client is sending, how the acknowledgments are arriving and, therefore, how the congestion window (and the transmission window) evolves.

If you are interested in knowing which congestion control algorithm your computer uses, you can execute the command from a terminal: *`sysctl -a | grep tcp> output`*. Then edit the output file and look for "congestion_control". There you will find the name of the algorithm used (currently the so-called cubic is one of the most used). This algorithm does not work exactly the same as the one explained in the theory classes (it is much more complex and uses a cubic function to determine the congestion window), but in the start it shows a similar behaviour to that already studied: it is doubled the amount of segments sent according to the receipt of ACKs.

At this point, and as you have seen in the first part of Lab #8, TCP implementations differ in the implementation of some recommended mechanisms. Further to the information given in the theory classes, we explain some concepts that include current implementations that can help you to understand correctly what you are going to see:

During the beginning of a connection, some systems have a mode called ***QuickACK****. In this case, the delayed sending of ACK does not start until several segments have been transmitted (for example 16 in some implementations). What do you get with this? The effect is to send an ACK for each segment and then the congestion window grows faster than if the receiver sends an ACK every two segments.*

However, there are implementations that incorporate the technique called TCP Congestion Control with ***Appropriate Byte Counting*** *(rfc3465). In the systems that implement it, the congestion window is not increased constantly with the reception of each ACK, but it increases as much as the information recognized by the ACK segments. Thus, if only one ACK arrives, but recognizes 2, 3 or 4 segments, the window is increased according to the information recognized. In this way, it is not necessary to send an ACK per segment at the beginning to increase the congestion window faster. These two techniques are not exclusive and we can find implementations that have both (for example, think that a machine can use QuickACK so that the other end -who does not know if it uses Appropriate Byte Counting or not- increase in any case the congestion window faster).*
Finally, it is also interesting to know that in current implementations the number of segments that are sent when a connection is initiated does not always have to be equal to 2 (the rfc5681 standard provides some recommendations). There are studies that determine, in addition, that it would be better to increase the number of segments that are initially transmitted and many systems allow sending up to 10 segments at the beginning of a connection.

Once we have briefly described the enormous casuistry of the implementations, improvements, proposals for improvements, etc., we are going to return to the capture to try to understand what we are seeing. To simplify and better understand the capture, the "offload" capacity of the computer has been deactivated, so that it will send segments of the maximum size determined at the beginning of the connection (MSS = 1460 in this case).

Load the file and get the Time-Sequence Graph as before (Statistics → TCP Stream Graph → Time-Sequence Graph). With the mouse or with the + and - keys you can zoom (zoom) any of the areas of interest that we are going to analyse. You will see a graph like the following:

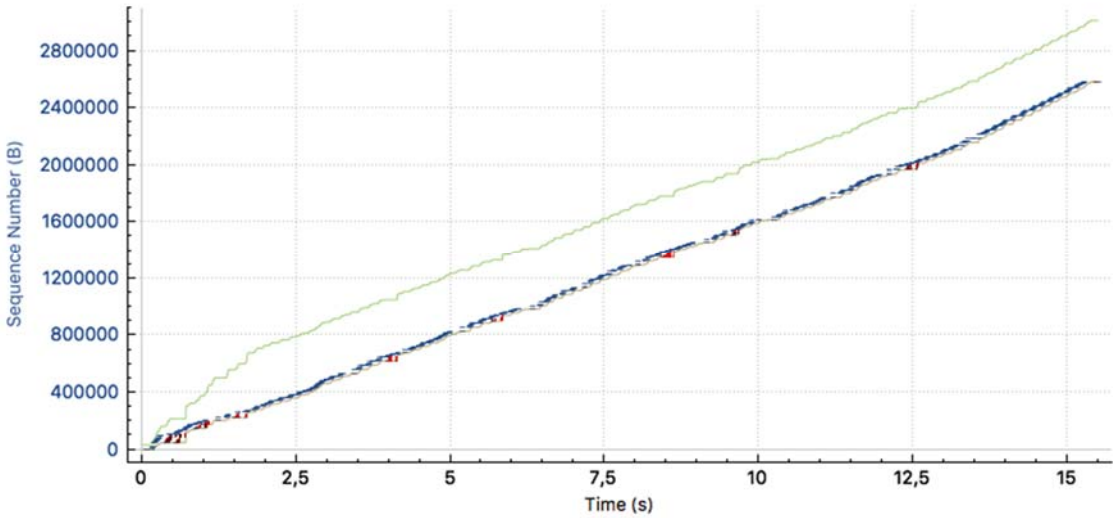

Figure 6. Statistics-> TCP Stream Graphs -> Time Sequence (tcptrace) of the captured trace.

When you zoom in on the initial area of Figure 6 you will get an image like the following (Figure 7):

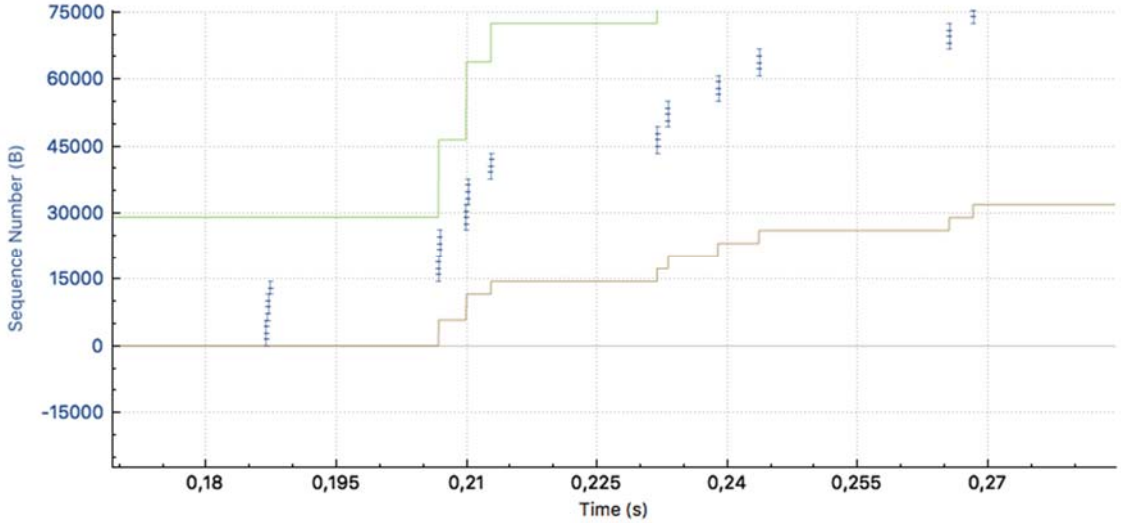

Figure 7. Statistics->TCP Stream Graphs -> Time Sequence (tcptrace). Zoom of Figure 6

Reason on the following aspects from the previous image:

1. How many segments does the client seem to send to the server at the beginning of the connection?

2. Every time an ACK is sent, How many segments are recognized?

3. How many new segments does the client send when it receives an ACK?

4. Do you think the client is using the Appropriate Byte Counting technique? Justify your answer

5. After sending 10 MSS, how many do you think it sends next?

6. From the moment 0.4 of the graph (which corresponds to the 1.23 of the main window of Wireshark) we observe small vertical lines above the sending.

a) What are the segments sent at the moment 0.4? Why are those segments sent now? What problem may have occurred?
b) Analyse the segment number 91 of the main window. Independently that Wireshark catalogues it as Window Update, analyses the rest of the information. What is the value of the ACK? Had that value come out before? What does it mean? Look at the options field for that segment. Do you remember what selective acknowledgement (SACK) is? What is the receiver indicating? Do you already know what those new vertical lines mean on top of those sending?