Labo computernetwerken 1 – Wireshark

Dit labo laat jullie enerzijds kennis maken met de software Wireshark, een packet sniffer programma dat toelaat om elk pakket (of meer correct, elk Ethernet *frame*) dat de computer verlaat te kopiëren, en te gaan analyseren. Wireshark probeert steeds zo goed als mogelijk alle protocollen die het kan herkennen in de inhoud van de gekopieerde pakketten, te benoemen. Anderzijds gebruiken we Wireshark om bepaalde protocollen, die reeds in de theorie besproken werden, van naderbij te bekijken.

Beide labo-opdrachten zijn opgesteld in de Engelse taal, de lingua franca van de informatica. Wie een stuk niet begrijpt, kan steeds verduidelijking vragen tijdens het labo.

Introductie Wireshark

Wie het softwareprogramma wil verkennen, kan het labo "Wireshark Lab: Getting Started" doorlopen. De opdrachten die hierin vermeld staan, moeten niet ingediend worden.

http://gaia.cs.umass.edu/wireshark-labs/Wireshark_Intro_v7.0.pdf

Opdracht 1: Wireshark HTTP-lab (Kurose Ross, modified) - applicatielaag

Het originele labo "HTTP", horend bij het handboek van Kurose Ross, vertrekt vanuit de opdracht om "live captures" te maken van verbindingen met een webserver. Doordat een computer sowieso al heel veel netwerkverkeer produceert, en hierdoor heel veel pakketten sluipen tussen degene van het te bestuderen HTTP-verkeer, werd het labo aangepast: de opdracht hierna vertrekt van "capture files". Deze "capture files" bevatten reeds het te bestuderen HTTP-verkeer, zonder al te veel andere pakketten. Met Wireshark kunnen deze bestanden geopend worden, en de vragen van het labo geven je een richting over hoe je aan de slag kan met de informatie die je kan terugvinden in de pakketten.

Niet elke luik van dit HTTP-lab is o.i. even interessant, we raden aan volgende delen te bestuderen:

- Par. 1.1: questions 1 to 7
- Par. 1.4: questions 16, 17
- Par. 1.5: questions 18, 19

Paragraaf 1.2 & 1.3 kan je doorlopen als je wil ; de antwoorden moet je niet uitwerken.
Tot slot: wie toch eens een "*live capture*" overweegt, kan in de appendix van dit HTTP labo deel terecht.

Opdracht 2: Wireshark TCP-lab (Kurose Ross, modified) – transportlaag

Ook deze opdracht vertrekt van een labo horend bij het handboek; opnieuw is het nemen van een "live capture" verwezen naar de appendix.

Het labo verkent enerzijds het afspreken van de sequence numbers in de three-way-handshake; anderzijds wordt stilgestaan bij de effectieve hoeveelheid bytes er per pakket verzonden worden (payload vs. headers).

Eigen verslag

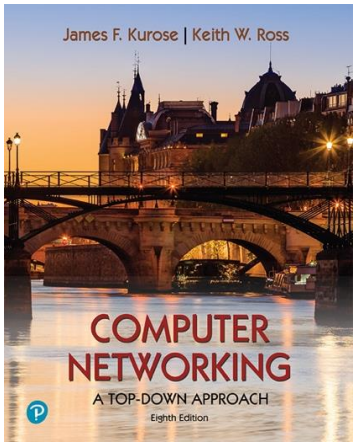
Nota nemen terwijl je werkt wordt verwacht tijdens al onze labo's. Deze Wireshark labs bestaan uit meerdere vragen; enkel bij de gefluoresceerde vragen raden we je aan om een kort antwoord te formuleren in je eigen verslag. Markdown is een ideaal formaat om in te werken.

Het verslag moet je niet indienen. Voor de start van het volgende labo worden enkele inzichtsvragen gesteld, je kan je eigen verslag (en enkel je eigen verslag) gebruiken om deze te beantwoorden.

Wireshark Lab: HTTP v8.1IDlab

Supplement to *Computer Networking: A Top-Down Approach, 8th ed.*, J.F. Kurose and K.W. Ross

"Tell me and I forget. Show me and I remember. Involve me and I understand." Chinese proverb



In this lab, we'll explore several aspects of the HTTP protocol: the basic GET/response interaction, HTTP message formats, retrieving large HTML files, retrieving HTML files with embedded objects, and HTTP authentication and security. Before beginning these labs, you might want to review Section 2.2 of the text.[1]

Download the zip file **`exercise.cap.zip`** from Ufora and extract the file **`01.http-wireshark-trace-1.pcap`**.The traces in this zip file were collected by Wireshark running on one of the author's computers, while performing the steps indicated in this Wireshark lab (see Appendix A).

1.1. The Basic HTTP GET/response interaction

Once you have downloaded the trace, you can load it into Wireshark and view the trace using the *File* pull down menu, choosing *Open*, and then selecting the **`01.http-wireshark-trace-1.pcap`** trace file.

Let's begin our exploration of HTTP by opening the provided packet trace file. Your Wireshark window should look similar to the window shown in Figure 1.

[1] References to figures and sections are for the 8th edition of our text, *Computer Networks, A Top-down Approach, 8h ed., J.F. Kurose and K.W. Ross, Addison-Wesley/Pearson, 2020.* Our authors' website for this book is http://gaia.cs.umass.edu/kurose_ross You'll find lots of interesting open material there.

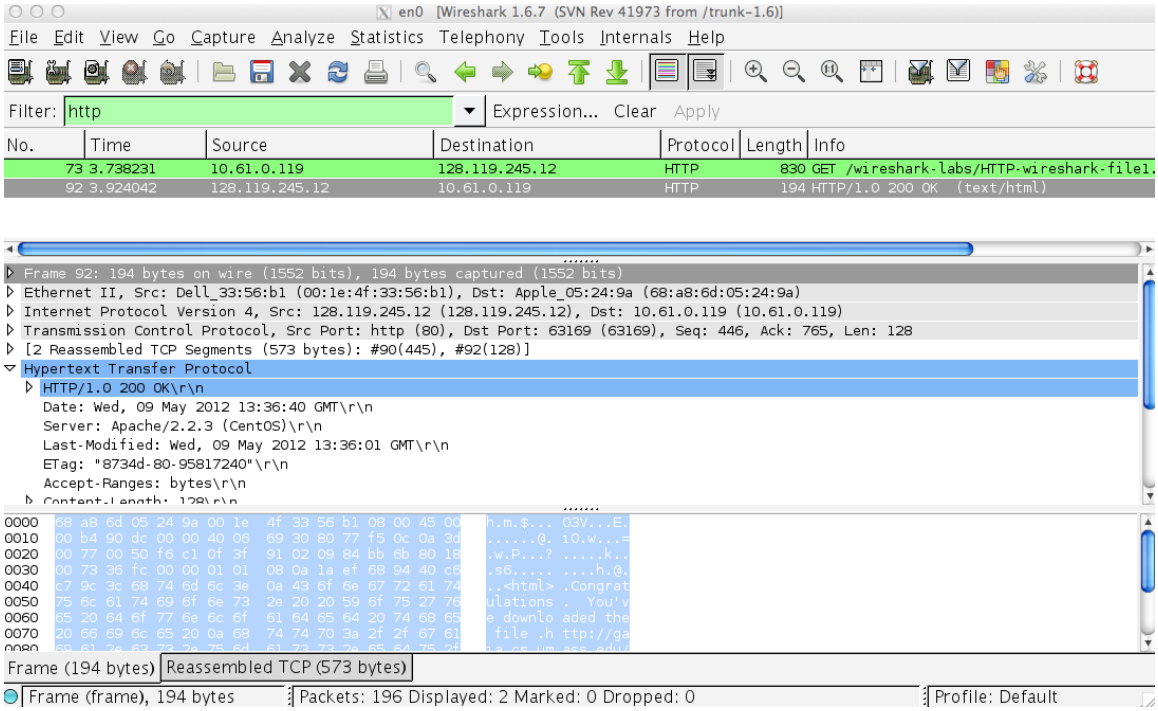


Figure 1: Wireshark Display after http://gaia.cs.umass.edu/wireshark-labs/ HTTP-wireshark-file1.html has been retrieved by your browser

The example in Figure 1 shows in the packet-listing window that two HTTP messages were captured: the GET message (from your browser to the gaia.cs.umass.edu web server) and the response message from the server to your browser. The packet-contents window shows details of the selected message (in this case the HTTP OK message, which is highlighted in the packet-listing window). Recall that since the HTTP message was carried inside a TCP segment, which was carried inside an IP datagram, which was carried within an Ethernet frame, Wireshark displays the Frame, Ethernet, IP, and TCP packet information as well. We want to minimize the amount of non-HTTP data displayed (we're interested in HTTP here, and will be investigating these other protocols is later labs), so make sure the boxes at the far left of the Frame, Ethernet, IP and TCP information have a plus sign or a right-pointing triangle (which means there is hidden, undisplayed information), and the HTTP line has a minus sign or a down-pointing triangle (which means that all information about the HTTP message is displayed).

> (*Note:* You should ignore any HTTP GET and response for favicon.ico. If you see a reference to this file, it is your browser automatically asking the server if it (the server) has a small icon file that should be displayed next to the displayed URL in your browser. We'll ignore references to this pesky file in this lab.).

By looking at the information in the HTTP GET and response messages, answer the following questions. When answering the following questions, you should display the GET and response messages (see the introductory Wireshark lab for an explanation of how to do this) and indicate where in the message you've found the information that answers the following questions.

1. Is your browser running HTTP version 1.0, 1.1, or 2? What version of HTTP is the server running?
2. What languages (if any) does your browser indicate that it can accept to the server?
3. What is the IP address of your computer? What is the IP address of the gaia.cs.umass.edu server?
4. What is the status code returned from the server to your browser?
5. When was the HTML file that you are retrieving last modified at the server?
6. How many bytes of content are being returned to your browser?
7. By inspecting the raw data in the packet content window, do you see any headers within the data that are not displayed in the packet-listing window? If so, name one.

In your answer to question 5 above, you might have been surprised to find that the document you just retrieved was last modified within a minute before you downloaded the document. That's because (for this particular file), the gaia.cs.umass.edu server is setting the file's last-modified time to be the current time, and is doing so once per minute. Thus, if you wait a minute between accesses, the file will appear to have been recently modified, and hence your browser will download a "new" copy of the document.

1.2. The HTTP CONDITIONAL GET/response interaction

*The questions in this section are **optional**, meaning that you can skip to section 1.3. If you choose not to skip to section 1.3, then we recommend to use the provided trace file:* **`01.http-wireshark-trace-2.pcap`**.

Recall from Section 2.2.5 of the text, that most web browsers perform object caching and thus often perform a conditional GET when retrieving an HTTP object.

Answer the following questions:

8. Inspect the contents of the first HTTP GET request from your browser to the server. Do you see an "IF-MODIFIED-SINCE" line in the HTTP GET?
9. Inspect the contents of the server response. Did the server explicitly return the contents of the file? How can you tell?
10. Now inspect the contents of the second HTTP GET request from your browser to the server. Do you see an "IF-MODIFIED-SINCE:" line in the HTTP GET? If so, what information follows the "IF-MODIFIED-SINCE:" header?
11. What is the HTTP status code and phrase returned from the server in response to this second HTTP GET? Did the server explicitly return the contents of the file? Explain.

1.3. Retrieving Long Documents

*The questions in this section are **optional**, meaning that you can skip to section 1.4. If you choose not to skip to section 1.4, then we recommend to use the provided trace file:* **`01.http-wireshark-trace-3.pcap`**.

In our examples thus far, the documents retrieved have been simple and short HTML files. Let's next see what happens when we download a long HTML file.

In the packet-listing window, you should see your HTTP GET message, followed by a multiple-packet TCP response to your HTTP GET request. This multiple-packet response deserves a bit of explanation. Recall from Section 2.2 (see Figure 2.9 in the text) that the HTTP response message consists of a status line, followed by header lines, followed by a blank line, followed by the entity body. In the case of our HTTP GET, the entity body in the response is the *entire* requested HTML file. In our case here, the HTML file is rather long, and at 4500 bytes is too large to fit in one TCP packet. The single HTTP response message is thus broken into several pieces by TCP, with each piece being contained within a separate TCP segment (see Figure 1.24 in the text). In recent versions of Wireshark, Wireshark indicates each TCP segment as a separate packet, and the fact that the single HTTP response was fragmented across multiple TCP packets is indicated by the "TCP segment of a reassembled PDU" in the Info column of the Wireshark display. Earlier versions of Wireshark used the "Continuation" phrase to indicated that the entire content of an HTTP message was broken across multiple TCP segments.. We stress here that there is no "Continuation" message in HTTP!

Answer the following questions:

12. How many HTTP GET request messages did your browser send? Which packet number in the trace contains the GET message for the Bill or Rights?
13. Which packet number in the trace contains the status code and phrase associated with the response to the HTTP GET request?
14. What is the status code and phrase in the response?
15. How many data-containing TCP segments were needed to carry the single HTTP response and the text of the Bill of Rights?

1.4. HTML Documents with Embedded Objects

We recommend to use the provided trace: **`01.http-wireshark-trace-4.pcap`**.

Now that we've seen how Wireshark displays the captured packet traffic for large HTML files, we can look at what happens when your browser downloads a file with embedded objects, i.e., a file that includes other objects (in the example below, image files) that are (possibly) stored on another server(s).

Answer the following questions:

16. How many HTTP GET request messages did your browser send? To which Internet addresses (URL) were these GET requests sent?
17. Can you tell whether your browser downloaded the two images serially, or whether they were downloaded from the two web sites in parallel? Explain.

1.5 HTTP Authentication

Finally, let's try visiting a web site that is password-protected and examine the sequence of HTTP message exchanged for such a site. The URL http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html is password protected. The username is "wireshark-students" (without the quotes), and the password is "network" (again, without the quotes). So let's access this "secure" password-protected site.
The result of this can be found in the trace: **`01.http-wireshark-trace-5.pcap`**.

Now let's examine the Wireshark output. You might want to first read up on HTTP authentication by reviewing the easy-to-read material on "HTTP Access Authentication Framework" at http://frontier.userland.com/stories/storyReader$2159

Answer the following questions:

18. What is the server's response (status code and phrase) in response to the initial HTTP GET message from your browser?
19. When your browser's sends the HTTP GET message for the second time, what new field is included in the HTTP GET message?

The username (wireshark-students) and password (network) that you entered are encoded in the string of characters (d2lyZXNoYXJrLXN0dWRlbnRzOm5ldHdvcms=) following the "`Authorization: Basic`" header in the client's HTTP GET message. While it may appear that your username and password are encrypted, they are simply encoded in a format known as Base64 format. The username and password are *not* encrypted! To see this, go to http://www.motobit.com/util/base64-decoder-encoder.asp and enter the base64-encoded string d2lyZXNoYXJrLXN0dWRlbnRz and decode. *Voila!* You have translated from Base64 encoding to ASCII encoding, and thus should see your username! To view the password, enter the remainder of the string Om5ldHdvcms= and press decode. Since anyone can download a tool like Wireshark and sniff packets (not just their own) passing by their network adaptor, and anyone can translate from Base64 to ASCII (you just did it!), it should be clear to you that simple passwords on WWW sites are not secure unless additional measures are taken.

Fear not! As we will see in Chapter 8, there are ways to make WWW access more secure. However, we'll clearly need something that goes beyond the basic HTTP authentication framework!

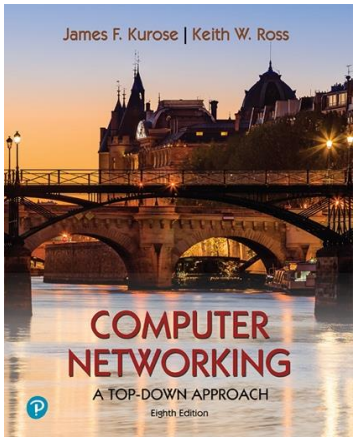
1.A. Appendix: making a live capture

Our IDlab lab works with *pre-captured* files: we have made these exercises for you, captured the relevant IP packets (or better, Ethernet frames) and put them in a zipfile for your convenience. If you want to make a *live capture*, this explains how to. Before performing the steps below, make sure your browser's cache is empty. (To do this under Firefox, select *Tools->Clear Recent History* and check the Cache box, or for Internet Explorer, select *Tools->Internet Options->Delete File;* these actions will remove cached files from your browser's cache.). Do the following:

1. Start up your web browser.
2. Start up the Wireshark packet sniffer, as described in the Introductory lab (but don't yet begin packet capture). Enter "http" (just the letters, not the quotation marks) in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window. (We're only interested in the HTTP protocol here, and don't want to see the clutter of all captured packets).
3. Wait a bit more than one minute (we'll see why shortly), and then begin Wireshark packet capture.
4. Enter the following to your browser
   (par.1) http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file1.html
   (par.2) http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html
   (par.3) http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file3.html
   (par.4) http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file4.html
   (par.5) http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html
   Your browser should display the very simple, one-line HTML file.
5. Stop Wireshark packet capture.

Wireshark Lab: TCP v8.1IDlab

Supplement to *Computer Networking: A Top-Down Approach, 8th ed.*, J.F. Kurose and K.W. Ross



"Tell me and I forget. Show me and I remember. Involve me and I understand." Chinese proverb

---

In this lab, we'll investigate the behavior of the celebrated TCP protocol in detail. We'll do so by analyzing a trace of the TCP segments sent and received in transferring a 150KB file (containing the text of Lewis Carrol's *Alice's Adventures in Wonderland*) from a computer to a remote server. We'll study TCP's use of sequence and acknowledgement numbers for providing reliable data transfer; we'll see TCP's receiver-advertised flow control mechanism. We'll also briefly consider TCP connection setup and we'll investigate the performance (throughput and round-trip time) of the TCP connection between your computer and the server.

Before beginning this lab, you'll probably want to review sections 3.5 in the text[1].

2.0 The capture file

Once you have downloaded the trace, you can load it into Wireshark and view the trace using the *File* pull down menu, choosing *Open*, and then selecting the **`02.tcp-wireshark-trace1-1.pcap`** trace file.

This trace has been created by accessing a Web page that allows to enter the name of a file stored on your computer (which, in this case, contained the ASCII text of *Alice in Wonderland*), and then transfer the file to a Web server using the HTTP POST method (see section 2.2.3 in the text). We're using the POST method rather than the GET method as we'd like to transfer a large amount of data *from* your computer to another computer.

---

[1] References to figures and sections are for the 8th edition of our text, *Computer Networks, A Top-down Approach, 8th ed., J.F. Kurose and K.W. Ross, Addison-Wesley/Pearson, 2020.* Our website for this book is http://gaia.cs.umass.edu/kurose_ross You'll find lots of interesting open material there.

2.1 A first look at the captured trace

Before analyzing the behavior of the TCP connection in detail, let's take a high-level view of the trace.

Let's start by looking at the HTTP POST message that uploaded the alice.txt file to gaia.cs.umass.edu from your computer. Find that file in your Wireshark trace, and expand the HTTP message so we can take a look at the HTTP POST message more carefully. Your Wireshark screen should look something like Figure 3.

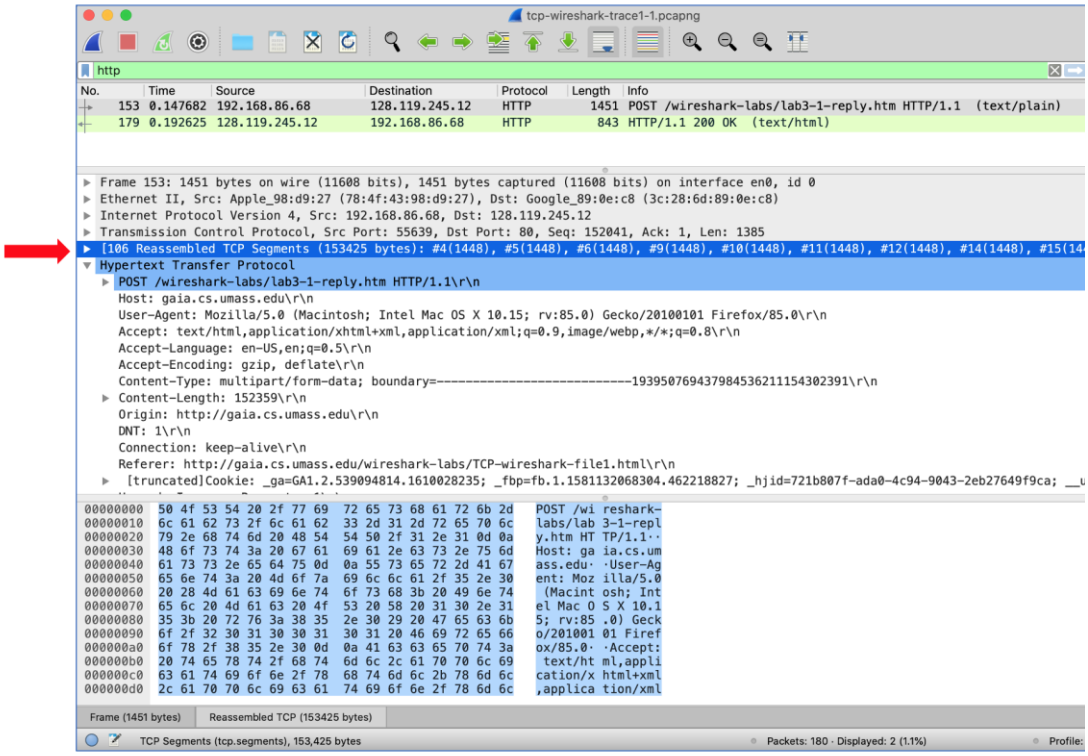


Figure 3: expanding the HTTP POST message that uploaded alice.txt from your computer to gaia.cs.umass.edu

There are a few things to note here:

- The body of your application-layer HTTP POST message contains the contents of the file alice.txt, which is a large file of more than 152K bytes. OK – it's not *that* large, but it's going to be too large for this one HTTP POST message to be contained in just one TCP segment!
- In fact, as shown in the Wireshark window in Figure 3 we see that the HTTP POST message was spread across 106 TCP segments. This is shown where the red arrow is placed in Figure 3 [Aside: Wireshark doesn't have a red arrow like that; we added it to the figure to be helpful ☺]. If you look even more carefully there, you can see that Wireshark is being really helpful to you as well, telling you that the first TCP segment containing the beginning of the POST message is packet #4 in the particular trace for the example in Figure 3, which is the trace *tcp-wireshark-trace1-1* noted in footnote 2. The second TCP segment containing the POST message in packet #5 in the trace, and so on.

Let's now "get our hands dirty" by looking at some TCP segments.
First, filter the packets displayed in the Wireshark window by entering "tcp" (lowercase, no quotes, and don't forget to press return after entering!) into the display filter specification window towards the top of the Wireshark window. Your Wireshark display should look something like Figure 4. In Figure 4, we've noted the TCP segment that has its SYN bit set – this is the first TCP message in the three-way handshake that sets up the TCP connection to gaia.cs.umass.edu that will eventually carry the HTTP POST message and the alice.txt file. We've also noted the SYNACK segment (the second step in TCP three-way handshake), as well as the TCP segment (packet #4, as discussed above) that carries the POST message and the beginning of the alice.txt file.

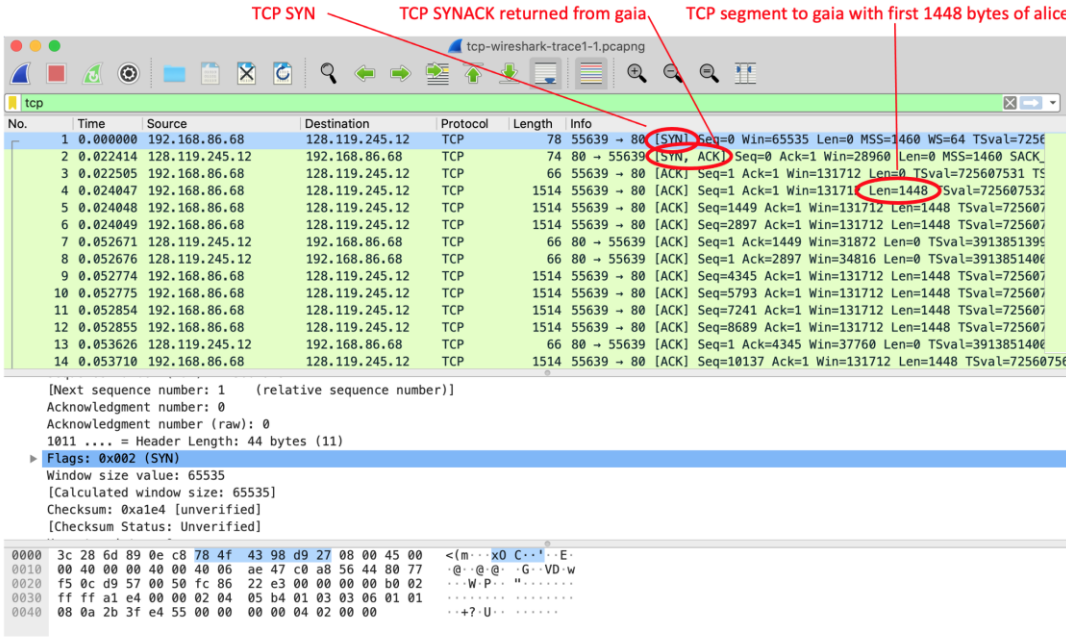


Figure 4: TCP segments involved in sending the HTTP POST message (including the file alice.txt) to gaia.cs.umass.edu

Answer the following questions:

1. What is the IP address and TCP port number used by the client computer (source) that is transferring the alice.txt file to gaia.cs.umass.edu? To answer this question, it's probably easiest to select an HTTP message and explore the details of the TCP packet used to carry this HTTP message, using the "details of the selected packet header window" (refer to Figure 2 in the "Getting Started with Wireshark" Lab if you're uncertain about the Wireshark windows).
2. What is the IP address of gaia.cs.umass.edu? On what port number is it sending and receiving TCP segments for this connection?

Since this lab is about TCP rather than HTTP, now change Wireshark's "listing of captured packets" window so that it shows information about the TCP segments containing the HTTP messages, rather than about the HTTP messages, as in Figure 4 above. This is what we're looking for–a series of TCP segments sent between your computer and gaia.cs.umass.edu!

2.2 TCP Basics

Answer the following questions for the TCP segments:

3. What is the *sequence number* of the TCP SYN segment that is used to initiate the TCP connection between the client computer and gaia.cs.umass.edu? (Note: this is the "raw" sequence number carried in the TCP segment itself; it is *NOT* the packet # in the "No." column in the Wireshark window. Remember there is no such thing as a "packet number" in TCP or UDP; as you know, there *are* sequence numbers in TCP and that's what we're after here. Also note that this is not the relative sequence number with respect to the starting sequence number of this TCP session.). What is it in this TCP segment that identifies the segment as a SYN segment?
4. What is the *sequence number* of the SYNACK segment sent by gaia.cs.umass.edu to the client computer in reply to the SYN? What is it in the segment that identifies the segment as a SYNACK segment? What is the value of the Acknowledgement field in the SYNACK segment? How did gaia.cs.umass.edu determine that value?
5. What is the sequence number of the TCP segment containing the header of the HTTP POST command? Note that in order to find the POST message header, you'll need to dig into the packet content field at the bottom of the Wireshark window, *looking for a segment with the ASCII text "POST" within its DATA field*[2,3]. How many bytes of data are contained in the payload (data) field of this TCP segment? Did all of the data in the transferred file alice.txt fit into this single segment?
6. What is the length (header plus payload) of each of the first four data-carrying TCP segments?[4]
7. How much data does the receiver typically acknowledge in an ACK among the first ten data-carrying segments sent from the client to gaia.cs.umass.edu? Can you identify cases where the receiver is ACKing every other received segment (see Table 3.2 in the text) among these first ten data-carrying segments?

[2] *Hint:* this TCP segment is sent by the client soon (but not always immediately) after the SYNACK segment is received from the server.

[3] Note that if you filter to only show "http" messages, you'll see that the TCP segment that Wireshark associates with the HTTP POST message is the *last* TCP segment in the connection (which contains the text at the *end* of alice.txt: "THE END") and *not* the first data-carrying segment in the connection. Students (and teachers!) often find this unexpected and/or confusing.

[4] The TCP segments in the tcp-wireshark-trace1-1 trace file are all less than 1480 bytes. This is because the computer on which the trace was gathered has an interface card that limits the length of the maximum IP datagram to 1500 bytes, and there is a *minimum* of 40 bytes of TCP/IP header data. This 1500-byte value is a fairly typical maximum length for an Internet IP datagram.