Lab2: Wireshark (TCP)

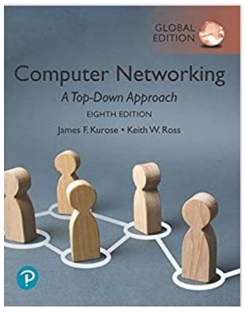
Computer Communication

Romaric Duvignau

February 10, 2025

Adopted and modified from **Wireshark Lab from J.F. Kurose and K.W. Ross** with contributions and support from Ali Salehson and Marina Papatriantafilou.

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



Lab2, 7 tasks, 23 questions.

Packet and traffic analysis (TCP part) **Purpose:** To listen to and interpret TCP traffic while using HTTP for retrieving and uploading large files as the application example.

Background/Preparation: Having read relevant sections in **Chapter 3** (especially **§3.5 and 3.7**) and having completed "Lab1".

Contents

3 Introduction to capturing TCP traffic

3.1 Retrieving long HTTP documents

In the examples of the first lab, the files that were retrieved have been simple and short HTML files. Let's next see what happens when you download a long HTML file. Do the following:

- Start up your web browser, and make sure that browser's cache is cleared.
- Start up the Wireshark capture and apply the filter: "`http && ip.addr==128.119.245.12`" (only HTTP traffic exchanged with the umass web server).
- Enter the following URL into your browser. `http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file3.html`.

  Your browser should display the rather lengthy **Bill of Rights**.
- Stop Wireshark packet capture.

In the packet-list panel, you should see the HTTP GET message, followed by a multiple-packet TCP response to the HTTP GET request. This multiple-packet response deserves a bit of explanation. Recall from **Section 2.2** of the course book (see Figure 2.9) 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 HTTP response, the entity body in the message is the entire requested HTML file. In this case, the HTML file is rather long (4861 bytes) and too large to fit in one TCP segment. The single HTTP response message is thus broken into several pieces by TCP, i.e. *segmented*, with each piece being contained within a separate TCP segment.

HTTP reassemble ON (default mode) In recent versions, Wireshark by default will indicate each TCP segment carried in a separate packet (not displayed when using a `http` filter as in this task), and the fact that the single HTTP response is segmented across multiple TCP segments is indicated by an extra line "**X Reassembled TCP segments (total bytes): #segmentID1(payload size), ...** " in the packet inspection panel between the lines for TCP and HTTP. This information is for example visible upon selecting the "HTTP/1.1 200 OK" response packet. Clicking on the "reassemble line" gives you which portion of the payload is covered by each TCP segment.

Task 1. Insert a screenshot of your packet capture and answer the following questions:

(a) How many HTTP GET request messages has your browser sent?

(b) Which packet in the trace contains the status code and phrase associated with the response to the HTTP GET request? Where can you find those information in the HTTP response?

(c) What is the length of the web page in bytes (i.e. length of the HTTP DATA or "Content Length")?

(d) How many TCP segments are needed to carry the single HTTP response and the text of the *Bill of Rights*? How large is the payload in each of the TCP segments? (Hint: look into the "X Reassembled TCP Segments" part). Subtract your previous answer from the total TCP payload to find the length of the HTTP headers.

Tips

The length of the HTTP headers in the server's response may vary depending on which browser you are using.

3.2 Exploring the TCP timeline

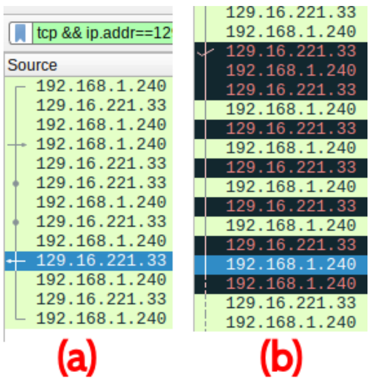
Important information about TCP (connection establishment, ACK, etc) is displayed in the leftmost part of the Wireshark packet list and this timeline will help you locate the different packets of a TCP stream. To explore the timeline, we will reproduce the previous task but this time working with TCP segments. Do the following:

- Start up your web browser, and make sure that the browser's cache is cleared.
- Start up the Wireshark capture and apply the filter: "`tcp && ip.addr==128.119.245.12`" (only TCP traffic exchanged with the umass web server).
- Open the following URL into your browser: `http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file3.html`.
- Stop Wireshark packet capture.

TCP timeline: Some precious information is provided left of the *Source* column (cf. Figure 1 for example of timelines):

- Upon selecting any TCP segment, all other TCP segments belonging to the same **TCP connection** are connected by a line (or **timeline**) with ⌜ marking the first and ⌞ marking the last; the line gets dashed when encountering packets that do not belong to the selected TCP connection.
- Upon selecting a TCP segment containing an **ACK**, the segment that was acked is marked in the timeline with ✓.
- Selecting an **HTTP request or response**, makes it appear as → for "request" (GET, POST, etc) and ← for responses; other HTTP messages on the same connection are also marked in the timeline: the corresponding request/response to the selected one appears as either → or ←, whereas other HTTP messages appear as •.

Figure 1 presents examples of timelines: (a) HTTP GET request → and response ← with HTTP DATA transmitted in between as •; (b) packet acked ✓ by the selected segment (blue-highlighted) with dashed timeline (another TCP connection at the bottom) and TCP events highlighted in black (eg TCP retransmission, out-of-order segments, receive window full, etc).



```
tcp && ip.addr==12
Source
192.168.1.240
129.16.221.33
192.168.1.240
192.168.1.240
129.16.221.33
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
```

```
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
129.16.221.33
192.168.1.240
192.168.1.240
129.16.221.33
192.168.1.240
```

Figure 1: TCP timelines.

HTTP messages and TCP segments in Wireshark The TCP segments containing the HTTP GET and the last piece of data are marked as "HTTP" in the *Protocol* column; all other TCP segments (ACKs and intermediary segments) are labelled with TCP. Upon inspecting one of the "HTTP-labeled" packets, all TCP segments that have been used to assemble them are accessible by double clicking on each of them in the **Reassemble** section (quick navigation shortcut); they also appear as • in the TCP timeline. When selecting an intermediary TCP-labeled segment, the last segment containing some data (the one labeled HTTP) appears as • in the timeline. Hence, it is quick to come back to the first segment (containing the request/response lines) by double clicking the first segment in the list. Note, there is an alternative presentation mode "HTTP reassemble OFF" in Wireshark in order to turn off HTTP reassembling, cf. the extra information given in Appendix A.

Task 2. Insert a screenshot of your packet capture and answer the following questions:

(a) Your browser has likely sent a GET request for the `favicon.ico`. Explore the timeline and identify if the same TCP connection was re-used for that purpose (persistent HTTP) or a parallel TCP connection was used (non-persistent HTTP). How can you tell?

(b) What is the source IP address and source port number `abcde` used by your computer for the TCP connection with the gaia web server (for the webpage's request)?

(c) What is the port number used by the server for the connection?

(d) Identify the TCP segments that are used to initiate the TCP connection between the client computer and the web server. How many segments are used? What is in the TCP header that identifies each segment as a handshaking segment?

(e) Identify two different ACK messages.

Tips

It's likely that you'll notice packets whose length exceeds the MTU (Maximum Transmission Unit) of your access link (1500 bytes for Ethernet, 2304 for 802.11, etc). This is due to your Operating System (OS) and Network Interface Card (NIC) applying so-called **large send offload** or **TCP segmentation offload**. In short, offloading means that your OS is able to send large frames (and thus less overhead for it) thanks to your NIC doing the job of fragmenting those large frames into smaller ones suited for your local link. Since you are capturing packets at the host-level, you will not be able to "see" the real individual packets transmitted over your link in this case (except of course, if you could capture them somehow on the other side of the link!).

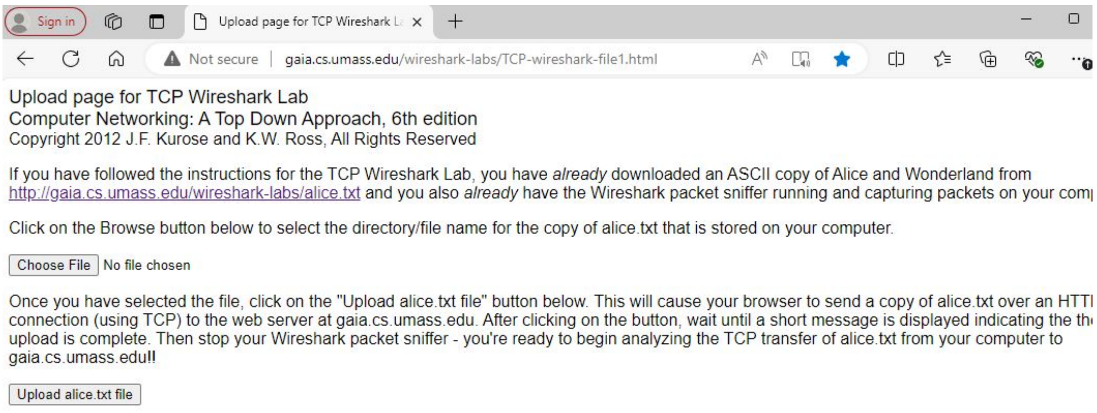
4 Exploring the TCP events

4.1 Capturing a bulk TCP transfer from your computer to a remote server

In order to perform an exploration of TCP, you will use Wireshark to obtain a packet trace of the TCP transfer of a large file from your computer to a remote server. You'll do so by accessing a Web page that will allow you to enter the name of a file stored on your computer (which contains 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**). You're going to use the POST method rather than the GET method to transfer a large amount of data *from* your computer to another computer. Of course, you'll be running Wireshark during this time to obtain the trace of the TCP segments sent and received from your computer. Do the following:

- Start up your web browser.
- Go to: `http://gaia.cs.umass.edu/wireshark-labs/alice.txt`.

  Retrieve an ASCII copy of *Alice in Wonderland* and store this file somewhere on your computer.
- Next go to:
  `http://gaia.cs.umass.edu/wireshark-labs/TCP-wireshark-file1.html` You should see a screen that looks like Figure 2.
- Use the Browse button in this form to enter the name of the file (full path name) on your computer containing Alice in Wonderland (or do so manually). **Don't press yet the "Upload alice.txt file" button or your browser will upload a blank file and you'll get only a single uploaded segment!**


Sign in
Upload page for TCP Wireshark L
Not secure
gaia.cs.umass.edu/wireshark-labs/TCP-wireshark-file1.html

Upload page for TCP Wireshark Lab
Computer Networking: A Top Down Approach, 6th edition
Copyright 2012 J.F. Kurose and K.W. Ross, All Rights Reserved

If you have followed the instructions for the TCP Wireshark Lab, you have *already* downloaded an ASCII copy of Alice and Wonderland from http://gaia.cs.umass.edu/wireshark-labs/alice.txt and you also *already* have the Wireshark packet sniffer running and capturing packets on your com

Click on the Browse button below to select the directory/file name for the copy of alice.txt that is stored on your computer.

Choose File No file chosen

Once you have selected the file, click on the "Upload alice.txt file" button below. This will cause your browser to send a copy of alice.txt over an HTT connection (using TCP) to the web server at gaia.cs.umass.edu. After clicking on the button, wait until a short message is displayed indicating the th upload is complete. Then stop your Wireshark packet sniffer - you're ready to begin analyzing the TCP transfer of alice.txt from your computer to gaia.cs.umass.edu!!

Upload alice.txt file

Figure 2: Upload page for TCP lab.

- Now start up Wireshark and begin packet capture and make sure the filter "`tcp && ip.addr==128.119.245.12`" is applied.
- Returning to your browser, press the "Upload alice.txt file" button to upload the file to the cse chalmers web server. Once the file has been uploaded, a short congratulations message will be displayed in your browser window.
- Wait a few seconds extra to let the time for TCP to close the unused connection.
- Stop Wireshark packet capture and save the trace you've just captured using the File → "Export Specified Packets..." with options *All Packets* and *Displayed* selected.

Tips

There are different options to save the list of captured packets (called "packet trace") for later use. Using **Save as...** will save all captured packets and may contain lots of packets irrelevant for the lab (you will need to reapply the same filter in this case). To save only packets displayed with the current filter, use instead **Export Specified Packets...** using the **Displayed** column when saving.

Side note: If by curiosity you were wondering what happened to the file you just uploaded, since the target of the html form is just a simple html page, it is just discarded upon reception.

4.2 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. What you should see is series of TCP segments between your computer and the web server. You should be able to identify:

- the initial three-way handshake achieved by SYN, SYN/ACK and ACK segments.
- the first TCP data-segment containing the header of an HTTP POST message and probably the first part of the uploaded file. To find it, look at the data part in the first TCP segment sent to the server after the handshake.
- TCP ACK-segments being returned from the webserver to your computer.
- the final four FIN/ACK and ACK segments closing the TCP connection at the end.

Tips

If several TCP connections are interlaced in your wireshark packet list, identify the source port `srcPort` of the one corresponding to the file upload (by eg inspecting the packet "HTTP/1.1 OK") and add to your Wireshark filter "`&& tcp.port == srcPort`".

Task 3. Insert a screenshot of the trace (once filtered with the above tips if needed) and answer the following questions:

(a) What is usually the packet length of the captured packets transmitted to the server? How large is usually the TCP payload within these packets?

(b) What is specified by the value of the Acknowledgement field in any received ACK-segment? How does `gaia.cs.umass.edu` determine this value? (use your own knowledge for this question!)

(c) How much data (number of bytes) does the receiver typically acknowledge in one ACK? (check with what you answered previously!)

(d) Identify the 4 packets closing the connection. How much time after the last HTTP DATA segment have they been sent?

5 Explore TCP behaviors

5.1 TCP basics and behavior

Now that you have familiarized yourself on your own captured data, we will use prerecorded data for the next task. The packet traces we will use are similar to the same text upload as you performed previously, just filtered and possibly shortened for better clarity.

- Download the following trace:
  `http://www.cse.chalmers.se/~duvignau/wireshark-labs/tcp_trace1.pcap`
- Open the packet trace with Wireshark.

Task 4. (a) Reproduce and fill up Table 1. The packets considered are the first six data-carrying segments in the TCP connection, that are exactly those following right after the handshake took place (with the first of them containing the HTTP POST and the rest containing HTTP Data). The different columns correspond to:

1) **Sequence Numbers** of the TCP segments (only relative numbers here).
2) **TCP Payload Length** (this is different from the packet length from the *Length* column!).
3) **Time sent:** the time that each of the six segments has been sent.
4) **Time ACKed:** the time when an acknowledgement (for each data-carrying segment) has been received.
5) **RTT:** Knowing the time when each TCP segment was sent, and the time when its acknowledgement was received, we can calculate the *sample RTT value.*
6) **EstimatedRTT:** Calculate the EstimatedRTT value using the measured RTT values in the answer of the previous question. Assume that the initial value of the EstimatedRTT is equal to the measured RTT for the first (POST) segment, and then as described in **Section 3.5.3** in the course book (using the EstimatedRTT equation on page 270) calculate the EstimatedRTT for the subsequent segments.

(b) Open the packet trace you saved before Task 3 and calculate the average RTT over the first 3 segments sent to the server with data. Those packets correspond to the HTTP POST and the beginning of the file upload similarly to the previous question.

Tips

The sequence and acknowledgement numbers in the previous task should be taken relatively to the initial sequence number (number 0). Wireshark will do this by default, displaying relative numbers.

5.2 TCP congestion control in action

For this part, we will be using pre-recorded packet traces and finally compare our findings with TCP traces captured on your own machine.

Warm-up 1: a slow start on TCP

We are using in this task the same trace that you opened for Task 4. Let's examine the amount of data (in bytes) sent in each round of the transfer session from the client to the server. Rather than (tediously!) calculating this from the trace in Wireshark, you will use one of Wireshark's TCP graphing utilities "Time-Sequence-Graph (Stevens)" to plot out the amount of data versus

No.	Seq. Number	Payload Length	Time sent	Time ACKed	RTT	Estimated RTT
4	1	565	0.026477	0.053937	0.02746	0.02746
5	566	1460	0.041737	0.077294	0.035557	0.028472125
7						
8						
10						
11						

Table 1: Fill up the table!

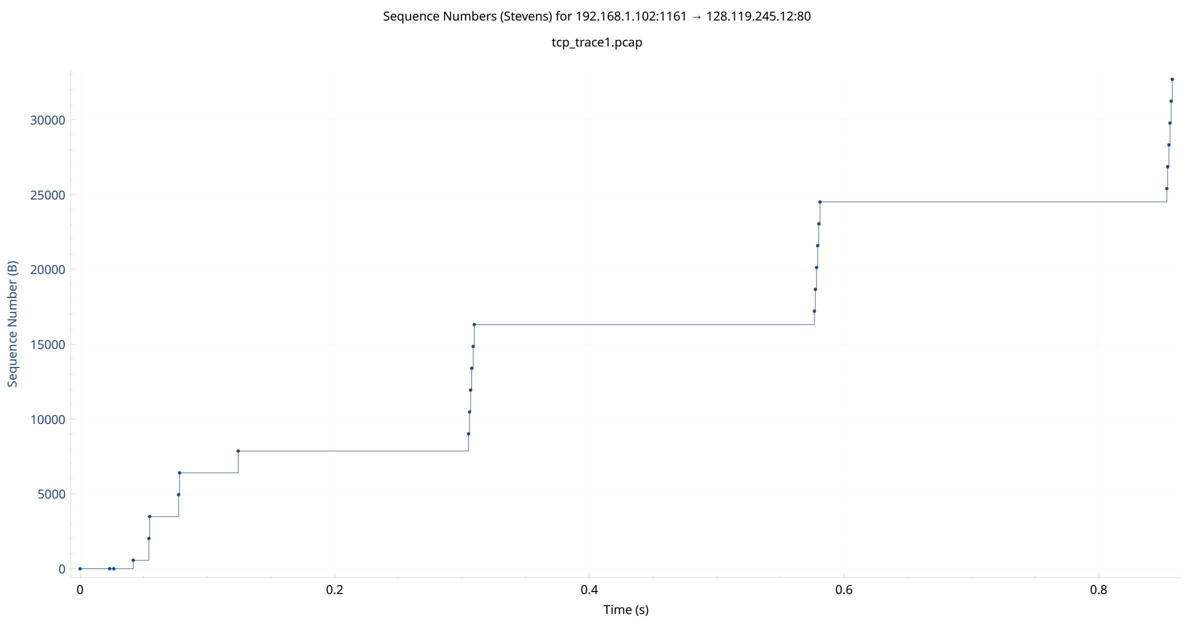


Figure 3: Time-Sequence-Graph (Stevens) generated from `tcp_trace1.pcap`.

time (cf. Figure 3). This Stevens-graph tool is plotting the sequence number of each segment versus the time at which it has been sent (or received if a packet coming from the server was selected upon creating the graph).

- In Wireshark, select the first packet transmitted (from the host to the server) then select in the menu: *Statistics → TCP Stream Graph → Time Sequence (Stevens)*.
- You should see a window identical to Figure 3 with a graph of dots where each • represents a TCP segment sent. Note that a set of • stacked above each other represents a series of segments that were sent back-to-back (pipelined) by the TCP sender.

Task 5. (a) By observing the time/sequence graph, can you see if any re-transmission (due to a packet loss or timeout) took place during the time covered by the trace? Explain briefly how (think how a re-transmission will appear in the graph).

(b) Where is the amount of available buffer space at the receiver (the server) advertised? Does the lack of receiver buffer space ever throttle the sender? Explain briefly.

Warm-up 2: acking the TCP events

- Download the second trace (yet another capture of the upload of alice1.txt): `http://www.cse.chalmers.se/~duvignau/wireshark-labs/tcp_trace2.pcap`
- Open the packet trace with Wireshark.
- Generate the Stevens graph as previously.

Task 6. (a) Can you identify a packet loss in the trace and how? Which sequence number was lost and at which timestamp does the sender notice the loss? (make use of the below hint!)

(b) Are there cases where the receiver (the web server) is ACKing cumulatively? At which timestamp does that happen and how many packets are acked at once? (Hint: go quickly through the acks and check if there are any suspiciously high jump in the amount of data ACKed, then use the timeline to navigate to the corresponding packet and get the confirmation that one or more acks were either skipped or lost.)

Hint

You can click on individual packets (● in the time/sequence graph) to jump to the corresponding segment in the TCP trace, then you can inspect its content.

Tips

You should notice that TCP offloading is in place in this second trace. So everytime a packet with a length greater than ≈1500 bytes is observed (either sent or received), smaller packets have been transmitted along the link. This can create some discrepancies when analyzing individual sequence numbers.

Observing TCP in real action!

- Start a packet capture with filter `tcp && ip.addr==129.16.222.33`
- Download one of the following files:
  1. `http://www.cse.chalmers.se/~duvignau/wireshark-labs/alice100.txt` (14.5MiB)
  2. `http://www.cse.chalmers.se/~duvignau/img/icecold2.ppm` (42.2MiB)
- Identify the "FIN, FIN-ACK, etc" packets sent to close the TCP connection after the last data segment has been sent, and ignore them by pressing **Ctrl+D** once they are selected; this will help you get a much better picture of the Stevens graph!
- Generate the Stevens graph of your packet trace (after selecting the first segment sent from client to server) and save the graph on your machine. If you only see a flat line, click "switch direction". Save as well a copy of your throughput graph: to check the instant throughput either pick the *Throughput* option in *TCP Stream Graphs* or change *type* once the graph is already generated

Task 7. Insert your Stevens graph and of your throughput graph in your report, and answer the following questions:

(a) What was the *overall throughput* in bit/s (the average number of bits transferred per time unit) for the *whole session* of transferring the file? Explain how you calculate these values.

(b) How did the throughput evolve through the transfer?

(c) Describe your Stevens graph, eg can you easily identify approximately the main TCP phases, "slow start" and "congestion avoidance" phases, on your graph?

(d) Comment on ways in which the measured data differs from the idealized behavior of TCP that we've studied in the text. Consider for instance in your answer the main TCP phases and TCP events **triple duplicated ACK** and **timeout**.

Hint

Remember that during the slow start phase, you should see an **exponential growth** of the amount of received data and that during the congestion avoidance phase, you should see a **linear increase** in the amount of received data.

Tips

Switch to *Mouse* → *zooms* to inspect in more details the graph for some time period; revert to original view with *Reset*.

Tips

Add `&& !tls` to the filter if you want only the TCP packets and not the TLS packets caused by the secure https connection to chalmers.

A Appendix: Turning off HTTP reassembling

It is possible to turning off HTTP reassembling which can ease to understand the flow of HTTP packets as displayed in Wireshark but makes it a bit harder to navigate through the TCP timeline.

HTTP Reassemble OFF: This mode is activated by unchecking 2 boxes (for body & header lines) in Wireshark's settings: **Edit** → **Preferences** → **Protocols** → **HTTP**.

The first TCP segment sent is then labelled as HTTP in *Protocol* column and TCP segments sent after that are marked as **"Continuation"** in the *Info* Column; received ACKs are labeled TCP. All TCP segments sent after that are marked as **"Continuation"** in the *Info* Column. Wireshark uses the "Continuation" phrase to indicate that the entire content of an HTTP message was broken across multiple TCP segments. It should be obvious here that there is no "Continuation" message in HTTP!

There is no easy way to quickly associate the "Continuation" segments to the corresponding HTTP request/response message except manually screening upwards packet per packet till finding the HTTP request/response and being carefull to stay on the same timeline (that is the same TCP connection).

HTTP Reassemble ON: This presentation mode is reactivated by re-checking the boxes previously unchecked.