Lab 3: HTTP and QUIC

Reti di Calcolatori II – Università di Trieste – Martino Trevisan

In this lab, you will practice with experiment with the **TCP**, **HTTP** (various versions) and **QUIC** protocols. You will practice with the **Google Chrome debugger**, measure the **performance** of different protocols, and learn basics of ***scarping*** (also called crawling). The tools you will use are:

- **Google Chrome Debugger:** by means of which you can inspect all HTTP transactions issued by the browser. Use the "Network" Tab and select the "Preserve Log" flag
- **BrowserTime**: a dockerized test suite to run experiments with automated web browsing.
- **`curl`:** a command line tool to issue HTTP request. You will use the classical `curl` command-line version and also a version compiled with a QUIC implementation, so that you can use HTTP/3.

Google Chrome Debugger

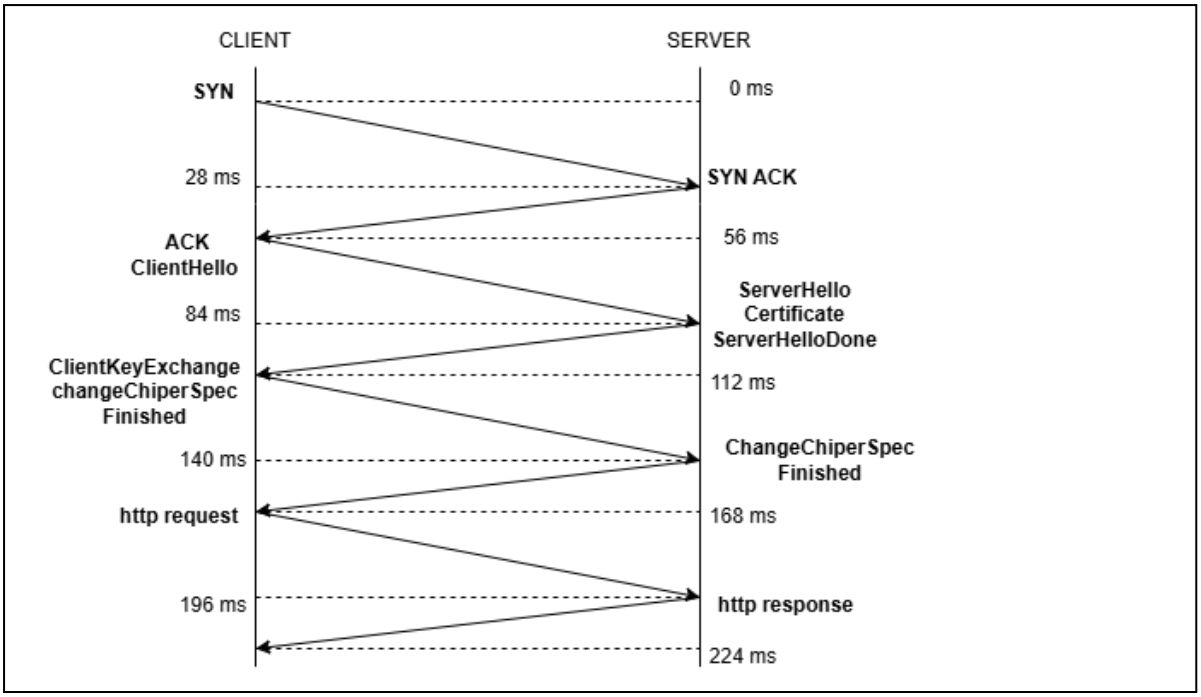
Practice with Google Chrome and with its **Debugger**. You must learn how to use the "*Network*" tab, which provides details on all HTTP requests. Check what happens when:

1. You access a webpage (e.g., www.units.it)
2. You scroll a web page
3. You log in into a website (hint: check what the "*Preserve Log*" flag does)

Question: When you log in into a website, which is the HTTP transaction carrying the **credentials**? Can you see the credentials through the **debugger**? Can a network observer (e.g., using **Wireshark**)?

> When you log into a website, the HTTP transaction carrying the credentials is typically the first request. In this case, the credentials j_username and j_password were included in the payload of the URL https://idp-cineca.units.it/idp/profile/SAML2/Redirect/SSO?execution=e3s1. This means that the username and password were directly part of the request payload, and the query string also included an execution parameter (execution=e3s1) tied to the SAML2 authentication process. Using Chrome's debugger in the "Network" tab, you could see these credentials clearly when inspecting the payload of the first request. This is because Chrome displays all outgoing requests before they are encrypted. However, a network observer, such as one using Wireshark, cannot see the credentials in plaintext, as they are encrypted with TLS. Only the encrypted data would be visible during the capture.

Question: Check the "*Timing*" details of various HTTP transactions. Create the trellis graph (such as this one) for a standard HTTP/1.1 transaction over TLS, showing the most important packets. Map each of the timings reported by Chrome in time intervals in the trellis graph.



Scraping

Scraping means automatically downloading a webpage to extract any useful information. Your goal is to make a Bash script printing the **current temperature** in a nice sea place: **Muggia**

You must gather the data by inspecting the webpage: https://www.3bmeteo.com/centraline-meteo

Use the Chrome debugger, the `curl` tool and other bash tools (e.g., `grep` or `tail`), to make a script which prints ***uniquely*** the temperature in Celsius degree.

Hint: use the Network Tab of the debugger to find the HTTP request carrying the data on the weather stations, and use the "***Copy as Curl***" option to obtain a Bash command that replicates the specific HTTP request.

Report the **script** in the following box.

Using the Chrome debugger's Network tab, I identified that the weather data is contained in a JSON file returned by request to rete_json.json?key=2973286....

The first step involved copying the relevant request using the "Copy as cURL" feature, which provides the full command necessary to replicate the HTTP request, including all headers and query parameters. This ensures the server responds as expected. I used this cURL command to fetch the data and saved the response to a temporary file on my system using the command less /tmp/meteo.

Once the JSON file was downloaded, I needed to extract the temperature value for Muggia. By using the command cat /tmp/meteo | tail -c +11 | jq . | grep -i muggia -A 10 | grep temperatura | cut -d '"' -f 4, I extracted the temperature, which was output as 9.9 degrees Celsius in this case.

Performance Measurement

Now, you will use **BrowserTime**, a tool to measure web page performance. It is a JavaScript tool that instruments Google Chrome (and other browsers) to automatically visit a webpage. It also collects various **statistics** on the issued HTTP transactions and performance.

You will use the **dockerized** version, so that you don't have to install all the dependencies. The image name is: `sitespeedio/browsertime`. To test a webpage, visiting it `N` times, you only need to run the following command line:

```
sudo docker run --rm sitespeedio/browsertime -n <N> <webpage>
```

On the standard output, you will get various statistics on performance, while the full log of the visit is stored **in the container** at the path: `/browsertime`

To access it, you must choose local path and **mount** it in `/browsertime` in the container. Use the option:

```
-v <localpath>:/browsertime
```

Assignment: select a **website supporting HTTP/3** and measure how its performance varies with **different HTTP versions**. You must visit the website instrumenting Chrome to support i) only HTTP/1.1, ii) HTTP/1.1 and 2, iii) All HTTP versions. By default Chrome supports all HTTP versions (case iii). You can disable HTTP/2 with the Chrome's command line option `--disable-http2` and HTTP/3 with `--disable-quic`. To pass an option to Chrome within BrowserTime, you should add the **`--chrome.args`** option in the command line. For example:

```
docker run --rm sitespeedio/browsertime <page> --chrome.args="--disable-quic"
```

You goal is to visit the website **20 times with each protocol version** and compare the value of:

1. **Page Load Time**: you can use the BrowserTime standard output or the `browsertime.json` log file
2. **Speed Index**: same hint as above

Assignment: Compute and compare the average values of the above metrics for the three cases.

To measure the performance of the webpage, I started by selecting **www.google.com** as the test site and verified that it supported HTTP/3 by inspecting the Network tab in the browser's developer tools. Once confirmed, I proceeded to measure its performance under three scenarios: HTTP/1.1 only, HTTP/1.1 and HTTP/2, and all HTTP versions (including HTTP/3). For each case, the webpage was visited 20 times using BrowserTime in a Dockerized environment, and the performance logs were saved locally for analysis.After running these tests, .har files containing detailed performance data were generated for each case and saved in the browsertime_logs directory. To compute the average Page Load Time (onLoad) and Speed Index, I wrote a Bash script to extract these metrics using the jq tool.

…..

extract_metrics() {

local har_file=$1

echo "Extracting metrics from $har_file

jq -r '.log.pages[] | {onLoad: .pageTimings.onLoad, speedIndex: ._visualMetrics.SpeedIndex}' "$har_file" echo

}

…

HTTP Version	Average onLoad (ms)	Average Speed Index
HTTP/1.1 Only	2280.08	1058.50
HTTP/1.1 and HTTP/2	2576.04	1307.85
All HTTP Versions	3452.94	1685.75

These averages clearly show that HTTP/1.1 was the fastest, with the lowest Page Load Time and Speed Index.

Optional – Artificial Delay

Repeat the same experiments as above imposing an **artificial delay** on the network.

To impose an artificial delay on Linux, you will use the **TC Netem** tool as follows:

1. Identify the physical WiFi or Ethernet interface name, e.g., **`eth0`**
2. Impose extra latency with the command:

   `tc qdisc add dev` **`eth0`** `root netem delay` **`100ms`**

3. Run the experiments using BrowserTime
4. Delete the artificial delay with: `tc qdisc del dev` **`eth0`** `root`

Report your results making plots in the form of a **CDF** (Cumulative Distribution Function).

Hint: to create the plots from data, you can use your favourite library, tool or programming language. We recommend `Python` and `MatPlotLib`. Otherwise, `gnuplot` or Excel are fine.