University of St. Gallen

Institute of Computer Science

Fundamentals of Computer Science (3,125), HS2019
I. Mizutani, D. Vachtsevanou, A. Ciortea, S. Mayer
iori.mizutani@unisg.ch

Assignment 5: Networking and Web Mashups

Deadline: Nov 5, 2019; 12:00 CET

Introduction In this assignment, you will investigate the networking stack and develop a simple distributed application on top of the network layer. You will also get to know the HTTP protocol in depth, and create distributed applications on top of the World Wide Web. After focusing explicitly on the *Network* and *Transport* layers in the first and second part, you will create a Web application dealing with a public data API on the Web in the third part of the assignment:

- Part 1: Self-Study (Task 1)
- Part 2: Network Programming (Tasks 2-3)
- Part 3: APIs and Web Applications (Task 4-5)

Rationale Fundamental knowledge about networks and the Web is a very useful asset. One of the major benefits of the Web (and specifically the HTTP(S) protocol) is that almost all widespread programming languages have (good) libraries for it, which makes the Web a great linking pin between heterogeneous systems.

Prerequisites A computer that is connected to the Internet and has the Wireshark[1] tool installed. You also need to install the *requests* library in your Python environment. As part of this assignment, you **ONLY** need to edit the following files:

- `task2/udp_echo_server.py`: the template file for Task 2.
- `task3/simple_web_browser.py`: the Template file for Task 3.
- `task4/reqres_client.py`: the template file for Task 4.
- `task5/bus_finder.py`: the template file for Task 5.

1 The Networking Stack and HTTP (Self-study; Nothing to submit)

In this task, you will get some practical experience on top of the theoretical knowledge about networks that we discussed during the lectures. Make sure that you are able to answer the following questions – they will prepare you for Tasks 2-5 of this assignment.

(a) What is the purpose of the Transmission Control Protocol (TCP)? What is the purpose of the sequence numbers that are embedded in TCP packets?

(b) As demonstrated in the exercise session, start recording your "main network interface" that is connected to the Internet (e.g., `en0`, `Wi-Fi`, etc.) with Wireshark and open `http://example.com` in your browser, then stop the Wireshark recording. In Wireshark, identify the HTTP packet your machine sent to the remote server `example.com` (IP address: `93.184.216.34`) by applying a filter `ip.addr == 93.184.216.34`. Click that packet to see its details, which are displayed conveniently split by networking layer, and find answers to the following questions:

[1] `https://www.wireshark.org/`

- **Network Layer (IP)**
  - Which version of the IP protocol was used?
  - What is the source IP address (i.e., your machine's IP address) and what is the destination IP address?
- **Transport Layer (TCP)**
  - What local port did your Web browser use to communicate with the server that hosts `example.com`?
  - At what port does the remote server host this website?
- **Application Layer (HTTP)**
  - In the packet, what is the payload that your machine sent to the remote server?
  - Which version of the HTTP protocol was used to send this payload?
  - What is the name of the *HTTP Method* that your browser used to retrieve data from `example.com`?

(c) From the raw HTTP response of the previous task, find the meaning and the values of the following parameters:

- The *HTTP Status Code*
- The *Content-Type* header
- The *Content-Length* header

(d) The response to your request in the previous exercise most probably includes an HTTP status code of `200 OK`. Look up the semantics of the following HTTP status codes:

- *304* Not Modified
- *401* Unauthorized
- *404* Not Found
- *418* I'm a Teapot

(e) Finally, clear the filter `ip.addr == 93.184.216.34` from Wireshark, and request the *secure* Website `https://unisg.ch` in your browser. You should now observe lots of packets that use the `TLS` protocol to communicate with the server hosting that website. Try to read the payload of any of these packets! Why can you not find HTTP packets anymore?

2 UDP Echo Server (4 Points)

In the previous task, the application-layer payload (i.e., the HTTP request and response from the server) were transmitted using a TCP connection. We now take a deep dive into the Transport Layer of the networking stack and implement a server that communicates with clients using the *User Datagram Protocol* (UDP).

`task2/udp_echo_server.py` contains a `UDPEchoServer` class whose object starts listening for UDP connections on port 22222 (see the instance method `start()`). In general, by "Echo Server" we refer to a server that responds to clients with the same (or similar) data it received. In this task, you will complete the `UDPEchoServer` class and observe the packets created by your own program!

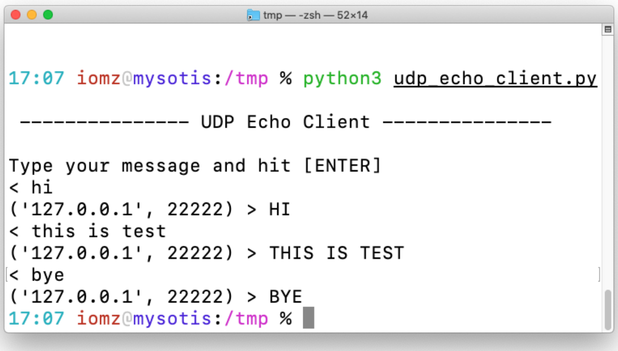
`task2/udp_echo_client.py` is provided for you to test your `UDPEchoServer`. It prompts users to type text in the console, send the message to your `UDPEchoServer`, then receive the response to display.

(a) Study the following sections of the (**Nothing to submit**):

- L.10 of `task2/udp_echo_server.py`: the IP address `127.0.0.1` has a special meaning. What is it? Why does `task2/udp_echo_client.py` have the same address in L.11?

- L.30 of `task2/udp_echo_server.py`: what is the `self._sock` object?

(b) Read the docstring of the instance method `start()` of `task2/udp_echo_server.py` and implement the method. After completing the method, open the system shell and run `task2/udp_echo_client.py`, then run `udp_echo_server.py` in another shell or on Thonny. Type a few messages in the shell for `udp_echo_client.py` and confirm the behavior of our Echo Server (see Figure 1).

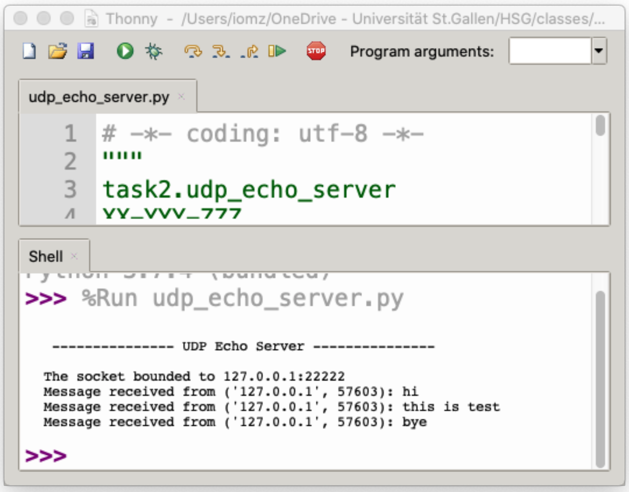


```
17:07 iomz@mysotis:/tmp % python3 udp_echo_client.py

 --------------- UDP Echo Client ---------------

Type your message and hit [ENTER]
< hi
('127.0.0.1', 22222) > HI
< this is test
('127.0.0.1', 22222) > THIS IS TEST
< bye
('127.0.0.1', 22222) > BYE
17:07 iomz@mysotis:/tmp %
```

(a) udp_echo_client.py



Thonny - /Users/iomz/OneDrive - Universität St.Gallen/HSG/classes/...
Program arguments:
udp_echo_server.py
Shell

```
# -*- coding: utf-8 -*-
"""

task2.udp_echo_server
```

```
>>> %Run udp_echo_server.py

  --------------- UDP Echo Server ---------------

 The socket bounded to 127.0.0.1:22222
 Message received from ('127.0.0.1', 57603): hi
 Message received from ('127.0.0.1', 57603): this is test
 Message received from ('127.0.0.1', 57603): bye

>>>
```

(b) udp_echo_server.py

Figure 1: Testing your Echo UDP Server.

(c) Monitor the communication between the server and the client with Wireshark by selecting the loopback interface. Find the packets that are passed between our client and server by applying the filter `udp.port == 22222`. Pick one UDP packet you observed and explain why its Frame Length[2] is of that specific size in your `README.txt` (max 2 sentences).

(3) Simple Web Browser (6 Points)

The goal of the next task is to create a simple Web browser that connects to a remote server using TCP, sends an HTTP GET request, and receives the response. You need to complete `task3/simple_web_browser.py` for this task. Example output is also provided as `task3/simple_web_browser_output.txt`. Figure 2 shows each task in a sequence diagram.

(a) To send an HTTP Request, you first need to create a bytestring of the request to be sent. Read the docstring of the function `create_http_request(host, path, method='GET')` and implement the function. After completed, uncomment the comment block for Task 3(a) in the `main()` function and run the script to test the function.

(b) After sending the request, you should receive a response from the remote server. We first start receiving the response data byte-by-byte until the end of the HTTP response header. Read the docstring of the function `receive_http_response_header(sock)` and implement the function. After completed, uncomment the comment block for Task 3(b) in the `main()` function and run the script to test your implementation.

[2] The total size of the Frame transferred on the wire (=Data Link layer); i.e., it includes the Frame, Packet, and Segment headers in addition to the data payload

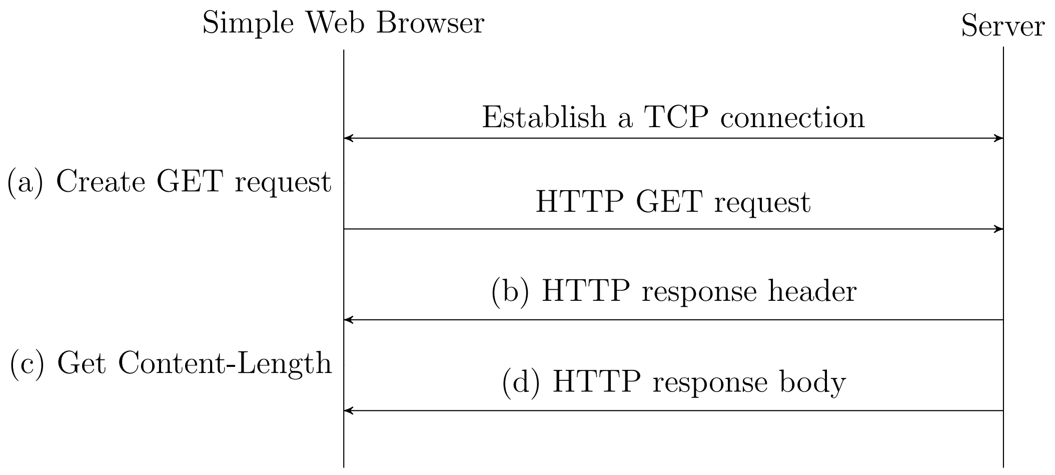


Figure 2: The sequence diagram for your simple Web browser and the remote server.

(c) From the HTTP response header, we want to extract the `Content-Length` of the HTTP packet to retrieve the rest of the data payload at once. Read the docstring of the function `get_content_length(header)` and implement it. When ready, uncomment the comment block for Task 3(c) in the `main()` function and run the script to test your implementation.

(d) Finally, receive the rest of the HTTP packet payload (i.e., the response `body` of size `Content-Length`). Read the docstring of the function `receive_body(sock, content_length)` and implement the function. When done, uncomment the comment block for Task 3(d) in the `main()` function and run the script to test your implementation.

4 Interaction with a Public Web API (4 Points)

HTTP requests are typically not implemented *by hand* as in the previous task, but rather using one of many different libraries that take care of the intricacies of the HTTP protocol and connection management.

Your task is to read data from a public Web API, namely *Reqres*[3], and to push data to this API using the *requests*[4] library. *Reqres* hosts a simulated "user management" application that exposes a REST-like HTTP API. Using this HTTP API, you can query user information, lists of users, update user information, create and delete users – all by using HTTP requests. Refer to the *requests* library documentation and the *Reqres* API specification (see links in footnote) to find out how to send HTTP requests to the Reqres API and what kind of data is returned in the HTTP responses.

After you complete `task4/reqres_client.py`, the output of your solutions should be similar to the sample output in `task4/reqres_client_output_sample.txt`.

(a) **HTTP Requests using the *requests* Library** The payloads of HTTP responses received from *Reqres* are in the JSON[5] format and contain information about a few (fake) users. JSON is a lightweight data-interchange format that is often used in Web applications and HTTP APIs because it can directly be translated to objects in various programming languages – e.g., to *dictionaries* in Python when using the *requests* library. The function `query_user_list (url)` in `task4/reqres_client.py` uses the *requests* library for sending an `HTTP GET` request that retrieves the `/users` resource from the Reqres API. The response to your successful `HTTP GET` request should have a `200 OK` status code. Read the docstring of the function

[3] https://reqres.in/
[4] https://requests.kennethreitz.org/en/master/
[5] https://json.org

`query_user_list(url)` and implement the function. After completing this task, uncomment the comment block for Task 4(a) in the `main()` function and run the script to test your code.

(b) **Switching Gears: POSTing Data** Up until now, you have only used HTTP's `GET` method to obtain data from the API. For this task, first look up the exact semantics of a few other methods that are specified in the HTTP protocol: `POST`, `PUT`, `DELETE`, `HEAD`, and `OPTIONS` (nothing to submit). The function `create_new_user(url, user)` in `task4/reqres_client.py` sends an `HTTP POST` request to Reqres API's `/users` resource to create a new user. The payload of your `HTTP POST` request should be in JSON format and should contain the following data:

```
{
    "first_name" : "John",
    "last_name" : "Smith",
    "email" : "jsmith@example.com"
}
```

The response to your successful `HTTP POST` request should have a `201 Created` status code. Read the docstring of the function `create_new_user(url, user)` and implement the function. After completing this task, uncomment the comment block for Task 4(b) in the `main()` function and run the script to test your function.

5 A Simple Web Application for VBSG (6 Points)

In this last task, you will work with the Swiss Public Transport API[6] to create a simple application that retrieves the bus schedule of Verkehrsbetriebe St.Gallen (VBSG) in real time: this Web API provides real-time information about the Swiss public transport system, which includes VBSG. Read the API's documentation[7] to understand how to send HTTP requests to this API and what data objects are returned in the HTTP responses.

Once you complete `task5/bus_finder.py`, the script should list the next 5 buses departing from St. Gallen Bahnhof to Uni/Dufourstrasse at the time when you run the script. The output should be similar to the sample output in `task5/bus_finder_output_sample.txt`.

(a) Read the docstring of the function `get_location_name(location_id)` and implement the function. After completing this task, uncomment the comment block for Task 5(a) in the `main()` function and run the script to test your code.

(b) Read the docstring of the function `find_next_bus(origin, destination, time, limit=5)` and implement the function. After completing this task, uncomment the comment block for Task 5(b) in the `main()` function and run the script to test your implementation.

Hand-in Instructions Create a file `README.txt` that includes 1.) answers to the questions, 2.) any reference that you used to complete this assignment, 3.) pitfalls you encountered and 4.) short explanations of your solution if necessary. Fill in your name and student ID in the comment header of files you edited.

Compress the whole folder with the Python files and the `README.txt` to a `zip` file named "`assignment5.zip`". Upload the zip file to your exercise group's Course page on Canvas. See Figure.3 for the list of files that should be included in the zip file.

[6] https://transport.opendata.ch

[7] https://transport.opendata.ch/docs.html

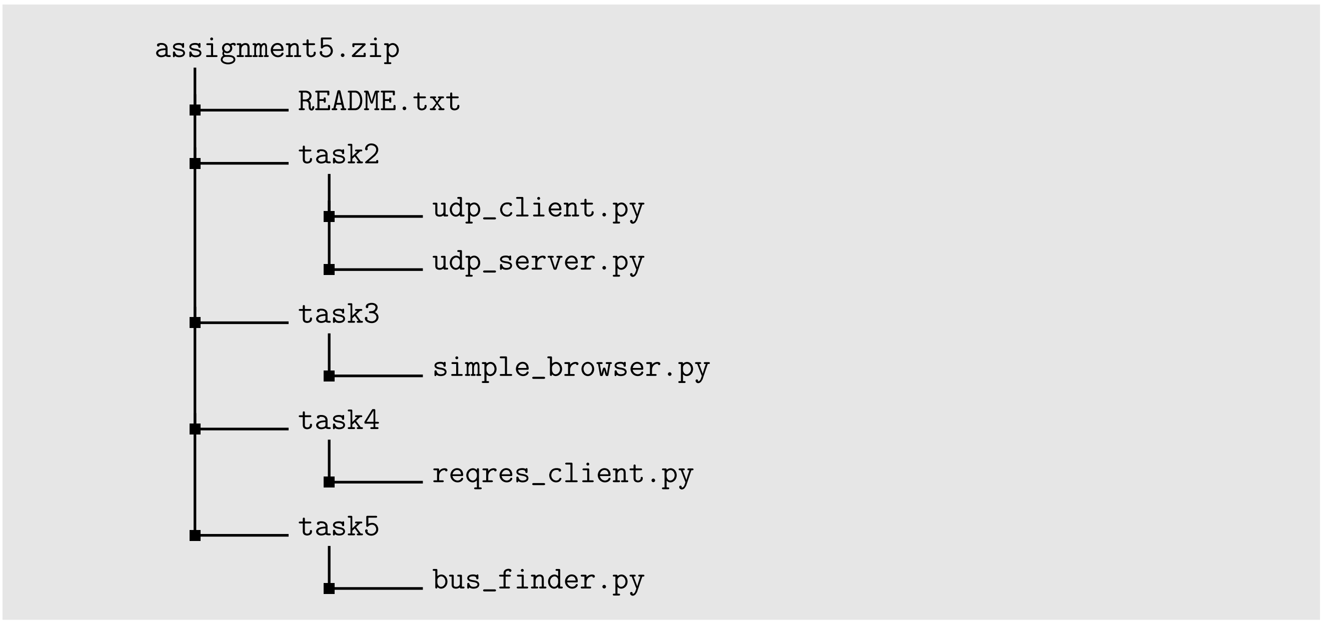


Figure 3: The required files for submission.