ECE 358: Computer Networks

Project 2: Socket Programming using Python

Borna Ansari <20822816>

Fares Al-Tantawi <20838014>

Table of Contents

Introduction

Please note that all required lab files have been submitted alongside this document in LEARN. Both task 1 and task 2 have been uploaded, with two separate READMEs that contain instructions on how to run the files.

Task 1.

First, a socket is created and bound using the following lines;

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
    server_socket.bind((HOST, PORT))
    server_socket.listen(1)
```

- `socket.socket(socket.AF_INET, socket.SOCK_STREAM)`: This line creates a new socket using the socket library. AF_INET specifies the address family (IPv4 in this case), and SOCK_STREAM indicates that this is a TCP socket.
- `server_socket.bind((HOST, PORT))`: Binds the socket to a specific host (IP address) and port number. HOST is set to '127.0.0.1' (the loopback address for local testing), and PORT is a user-defined port number greater than 1023.
- `server_socket.listen(1)`: Puts the server into listening mode. The argument 1 specifies the maximum number of queued connections.

The following lines handle incoming connections;

```
    while True:
        client_socket, _ = server_socket.accept()
        handle_request(client_socket)
        client_socket.close()
```

- The server runs an infinite loop `(while True:)` to continuously listen for incoming connections.
- `server_socket.accept()`: Accepts an incoming connection. It returns a new socket object client_socket for the connection and the address of the client.
- `handle_request(client_socket)`: This function is called to process the incoming request. It takes the client socket as an argument.
- `client_socket.close()`: After handling the request, the server closes the client socket.

The following function is what handles requests;

```
def handle_request(client_socket):
    request = client_socket.recv(BUFFER_SIZE).decode()
```

- `client_socket.recv(BUFFER_SIZE).decode()`: Receives data from the client. `BUFFER_SIZE` defines the maximum amount of data to be received at once (here, it is set to 1024 bytes). The received data is decoded from bytes to a string.
- The function then parses the HTTP request, extracts the requested file name, and decides how to respond.

This following function is what generates the HTTP response headers;

```
def generate_headers(code, filepath):
```

- This function generates the HTTP response headers. It includes necessary headers like status code, Date, Server, Connection, etc.
- If the requested file exists (HTTP 200), it adds additional headers like Content-Length and Content-Type. If the file doesn't exist (HTTP 404), only the basic headers are added.

Finally, the following lines are used to send the response back from the server, note that this isn't particularly in order and has been taken as a snippet for simplicity;

```
            response_header = generate_headers(200, filepath)
            response = response_header.encode() + response_data
        client_socket.sendall(response)
```

- The response consists of the header and, for GET requests, the requested file's content.
- `response_header.encode()` converts the header string into bytes.
- The file content (`response_data`) is read and appended to the header if it's a GET request.
- `client_socket.sendall(response)` sends the complete response back to the client.

Figure 1 below shows the screenshot of the client browser, as requested by the lab manual.

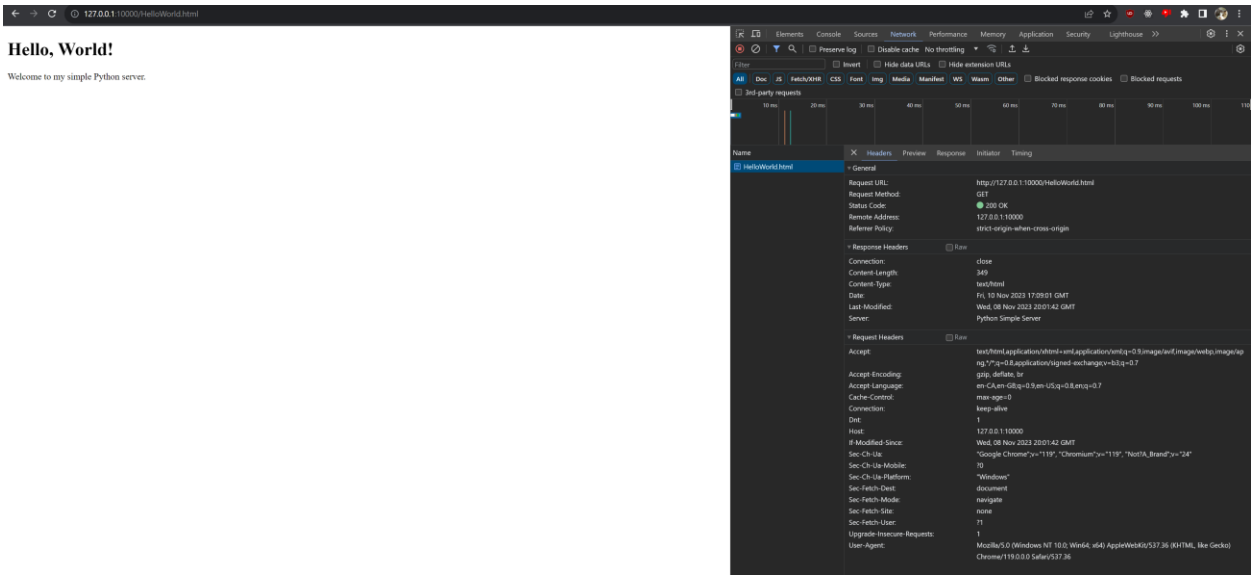


Figure 1: Client browser for task 1

Figure 2 below shows the Postman GET response.

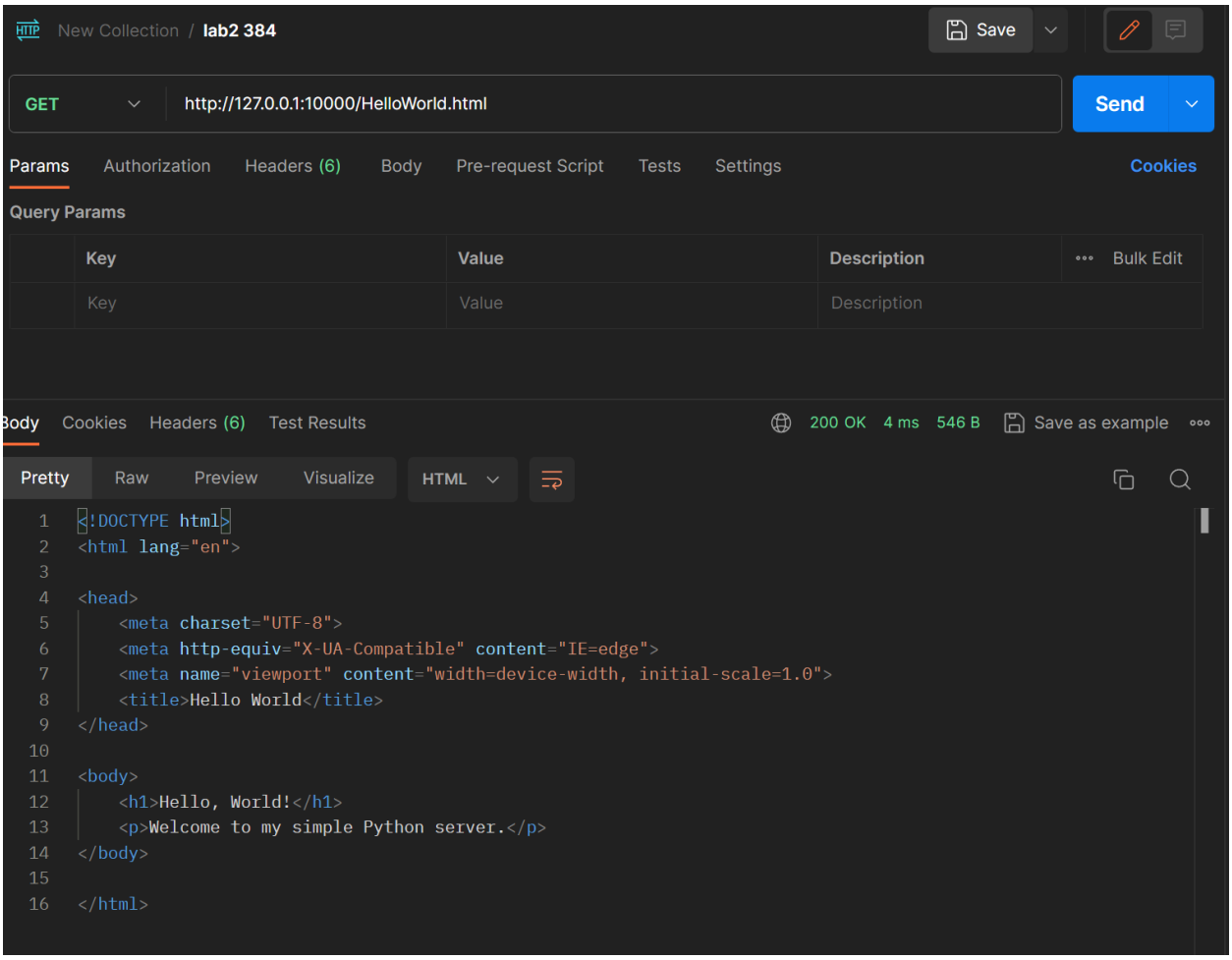


Figure 2: Postman GET response for task 1

Figure 3 below shows the Postman HEAD response.

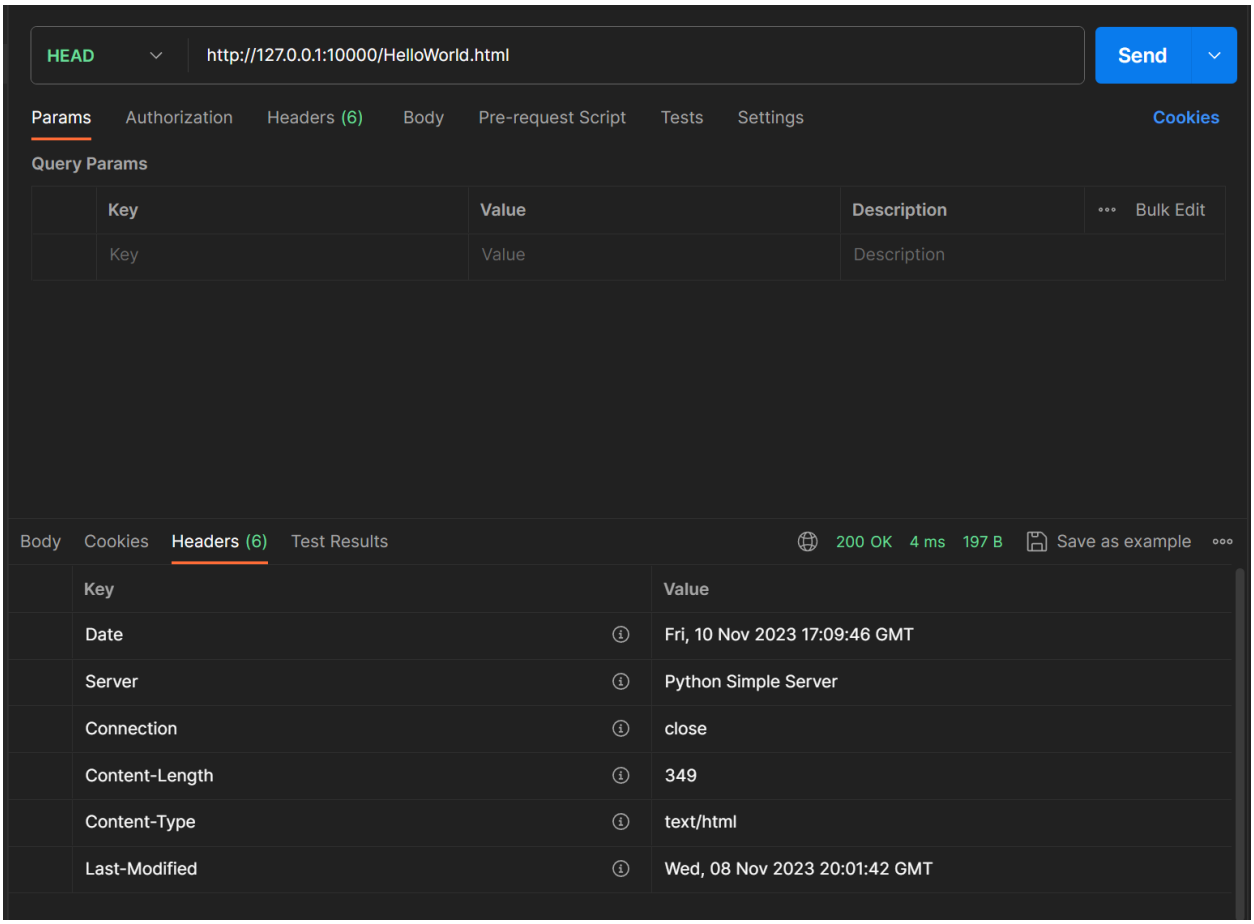


Figure 3: Postman HEAD response for task 1

Task 2.

The following code snippet is whats used to create and bind the socket;

```
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind(('127.0.0.1', 10053))
```

- `socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`: Creates a new socket using the socket library. AF_INET specifies IPv4, and SOCK_DGRAM indicates that this is a UDP socket, which is standard for DNS servers.
- `server_socket.bind(('127.0.0.1', 10053))`: Binds the socket to the loopback address (127.0.0.1) and a port number (here, 10053).

This following dictionary is what stores all of the DNS records we intend to use for the purposes of this lab;

```
DNS_RECORDS = {
    'google.com': ['192.165.1.1', '192.165.1.10'],
    'youtube.com': ['192.165.1.2'],
    'uwaterloo.ca': ['192.165.1.3'],
    'wikipedia.org': ['192.165.1.4'],
```

```
    'amazon.ca': ['192.165.1.5'],
}
```

The following line of code is what receives and processes our DNS queries;

```
        query, client_address = server_socket.recvfrom(512)
```

- The server listens for incoming DNS queries using `recvfrom(512)`, which waits for data from a client. 512 bytes is a standard size for DNS queries.
- `query` contains the received data, and `client_address` has the address information of the client.

We utilize a small helper function called `parse_domain_name` below to parse our domain name from the user query.

```
        domain_name = parse_domain_name(query)
```

- `parse_domain_name(query)`: A function that extracts the domain name from the DNS query. It handles the query format and decodes the domain name from the DNS query bytes.

To generate a DNS response, we use the following line where we call `create_dns_response`;

```
            response = create_dns_response(query, domain_name)
```

- If the domain name is found in DNS_RECORDS, create_dns_response is called to construct the DNS response message.
- This function creates a response that includes the transaction ID from the query, the DNS flags, the count of questions and answers, and the answer section containing the IP addresses associated with the domain name.

We then send the DNS response using the following line;

```
            server_socket.sendto(response, client_address)
```

- The constructed DNS response is sent back to the client using sendto, which targets the client_address received with the query.

After we send the response, we then display both the request and response in hex.

```
        print("Request (in Hex):", display_hex(query))
        print("Response (in Hex):", display_hex(response))
```

- `display_hex` is a helper function that converts the query and response bytes into a human-readable hex format.

The following is the server output for inputting google.com.

Request (in Hex): aa bb 01 00 00 01 00 00 00 00 00 00 06 67 6f 6f 67 6c 65 03 63 6f 6d 00 00 01 00 01

Response (in Hex): aa bb 81 80 00 01 00 02 00 00 00 00 c0 0c 00 01 00 01 00 00 01 04 00 04 c0 a5 01 01 c0 0c 00 01 00 01 00 00 01 04 00 04 c0 a5 01 0a

Color-coding it according to table 2 within the lab manual would yield the following;

Query: AA BB 01 00 00 01 00 00 00 00 00 00 06 67 6f 6f 67 6c 65 03 63 6f 6d 00 00 01 00 01

Response: AA BB 81 80 00 01 00 02 00 00 C0 0C 00 01 00 01

Figures 4 and 5 below show the client output and server output for google.com respectively.

```
PS C:\Users\tanta\Desktop\school\ece 358\task2> python3 .\client.py
Enter Domain Name: google.com
> google.com: type A, class IN, TTL 260, addr (4) 192.165.1.1
> google.com: type A, class IN, TTL 260, addr (4) 192.165.1.10
Enter Domain Name:
```

Figure 4: Client output for google.com

```
PS C:\Users\tanta\Desktop\school\ece 358\task2> python3 .\server.py
DNS Server is running on 127.0.0.1:10053
Request from: ('127.0.0.1', 51900)
Request (in Hex): aa bb 01 00 00 01 00 00 00 00 00 00 06 67 6f 6f 67 6c 65 03 63 6f 6d 00 00 01 00 01
Response (in Hex): aa bb 81 80 00 01 00 02 00 00 00 00 c0 0c 00 01 00 01 00 00 01 04 00 04 c0 a5 01 01 c0 0c 00 01 00 01 00 00 01 04 00 04 c0 a5 01 0a
```

Figure 5: Server output for google.com

Figures 6 and 7 below show the client output and server output for wikipedia.org respectively.

```
Enter Domain Name: wikipedia.org
> google.com: type A, class IN, TTL 260, addr (4) 192.165.1.4
Enter Domain Name:
```

Figure 6: Client output for wikipedia.org

```
DNS Server is running on 127.0.0.1:10053
Request from: ('127.0.0.1', 64074)
Request (in Hex): aa bb 01 00 00 01 00 00 00 00 00 00 09 77 69 6b 69 70 65 64 69 61 03 6f 72 67 00 00 01 00 01
Response (in Hex): aa bb 81 80 00 01 00 01 00 00 00 00 c0 0c 00 01 00 01 00 00 01 04 00 04 c0 a5 01 04
```

Figure 7: Server output for wikipedia.org

Task 3.

Question 1

A socket is an endpoint for sending and receiving data across a computer network. It is a fundamental building block for network communications between two machines running network software, allowing them to communicate.

Question 2

Sockets work by establishing a communication channel between processes on the same machine or across different machines over a network. They are required for network communication because they provide a standardized way for software to transfer data between different machines or processes. It achieves this by abstracting the complexities of the underlying network protocols.

Question 3

Stream sockets (TCP Sockets): Use the Transmission Control Protocol for data transmission. They are reliable, connection-oriented sockets that guarantee that data will be delivered in the same order it was sent.

Datagram sockets (UDP Sockets): Use the User Datagram Protocol. They are connectionless and do not guarantee order or delivery. They are typically used for broadcasting messages over a network.

Raw sockets: Allow direct transmission and reception of IP packets without any protocol-specific transport layer formatting. They are used for low-level networking, such as building custom protocols.

Question 4

Stream sockets: Used for applications that require a reliable connection, such as web browsing (HTTP/HTTPS) or file transfer (FTP).

Datagram sockets: Used in applications where speed is critical and loss of packets is tolerable, such as online games or video streaming.

Raw sockets: Used for network utilities like ping and traceroute, which are used for diagnostic or routing purposes.

Question 5

socket(): Creates a socket descriptor that defines the protocol, address family, and socket type.

bind(): Associates a socket with a specific port on the local machine, allowing the socket to receive data destined for this port.

connect(): Initiates a connection on a socket or connects a socket to a remote address and port.

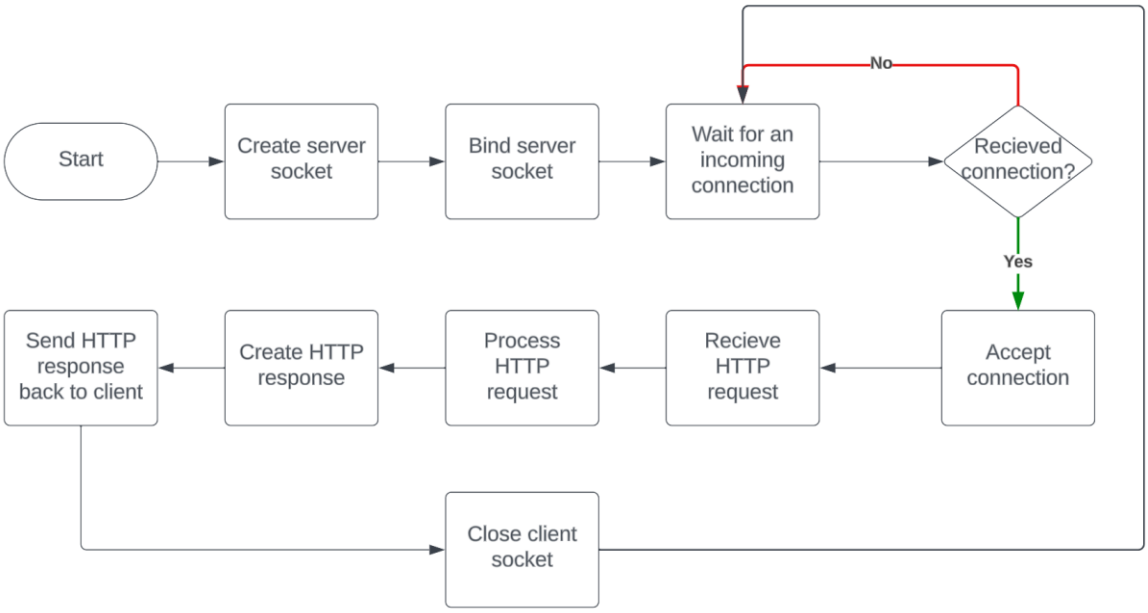
listen(): Marks the socket as a passive socket, that will be used to accept incoming connection requests using accept().

accept(): Accepts an incoming connection request on a listening socket and creates a new socket for the established connection.

close(): Closes the socket, releasing the resources allocated to the socket.

Question 6

Figure 8 below shows the flow diagram for our socket programming use for the lab.



Question 7

Sockets execute at the transport layer of the Internet protocol stack.

Question 8

Table 1 is filled in with the required answer for each protocol mentioned in question 8.

Table 1: Question 8 completed table

Protocol	Port #	Common Function
HTTP	80	Web browsing
FTP	21	File transfer
IMAP4	143	Email retrieval
SMTP	25	Email sending
Telnet	23	Remote terminal connection
POP3	110	Email retrieval