EEE3093S - Extra Credit Assignment Submission

Samson Okuthe
OKTSAM001

October 16, 2025

Appendix A: Task 1 Web Server Code

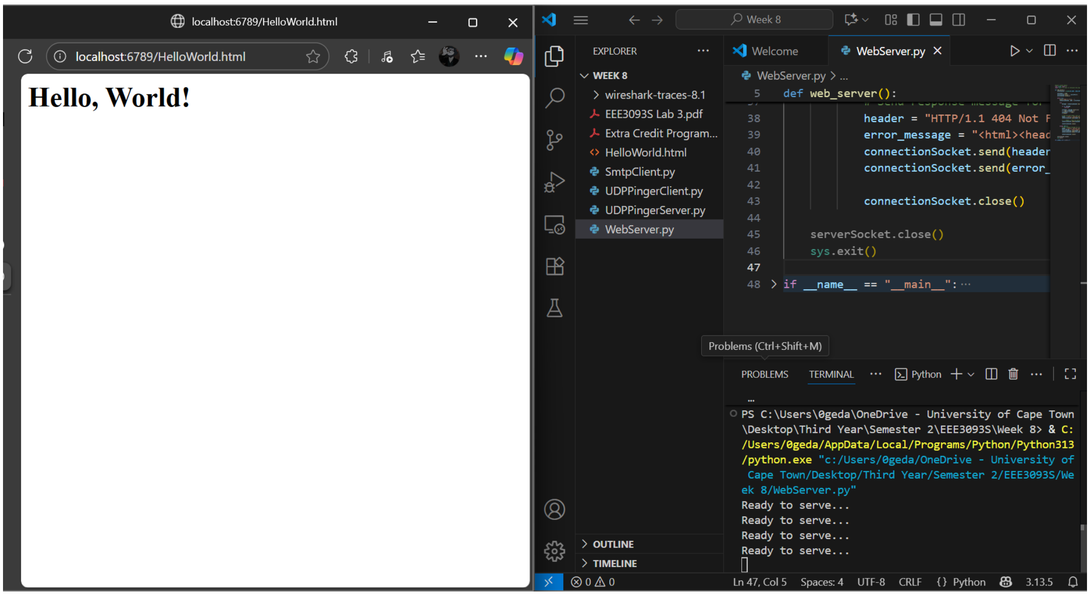
Here is the complete Python implementation for the simple TCP web server.

```python
#import socket module
from socket import *
import sys # In order to terminate the program

def web_server():
    serverSocket = socket(AF_INET, SOCK_STREAM)

    # Prepare a server socket
    serverPort = 6789
    serverSocket.bind((’’, serverPort))
    serverSocket.listen(1)

    while True:
        # Establish the connection
        print(’Ready to serve...’)
        connectionSocket, addr = serverSocket.accept()

        try:
            # Receive and parse the HTTP request
            message = connectionSocket.recv(1024).decode()

            # Handle empty request from some browsers
            if not message:
                continue

            filename = message.split()[1]

            # Open the requested file from the server’s file system
            f = open(filename[1:])
            outputdata = f.read()
            f.close()

            # Send the HTTP response header
            header = "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n"
            connectionSocket.send(header.encode())

            # Send the content of the requested file to the client
            connectionSocket.send(outputdata.encode())

            # Close the client connection socket
            connectionSocket.close()

        except IOError:
            # Send response message for file not found (404)
            header = "HTTP/1.1 404 Not Found\r\n\r\n"
            error_message = "<html><head></head><body><h1>404 Not Found</h1></body></html>\r\n"
            connectionSocket.send(header.encode())
            connectionSocket.send(error_message.encode())

            # Close the client connection socket
            connectionSocket.close()

    serverSocket.close()
    sys.exit()

# Run the web server
if __name__ == '__main__':
    web_server()
```

Listing 1: WebServer.py - A simple HTTP server

Appendix B: Demonstration Screenshots

Below are the screenshots verifying the functionality of the web server.

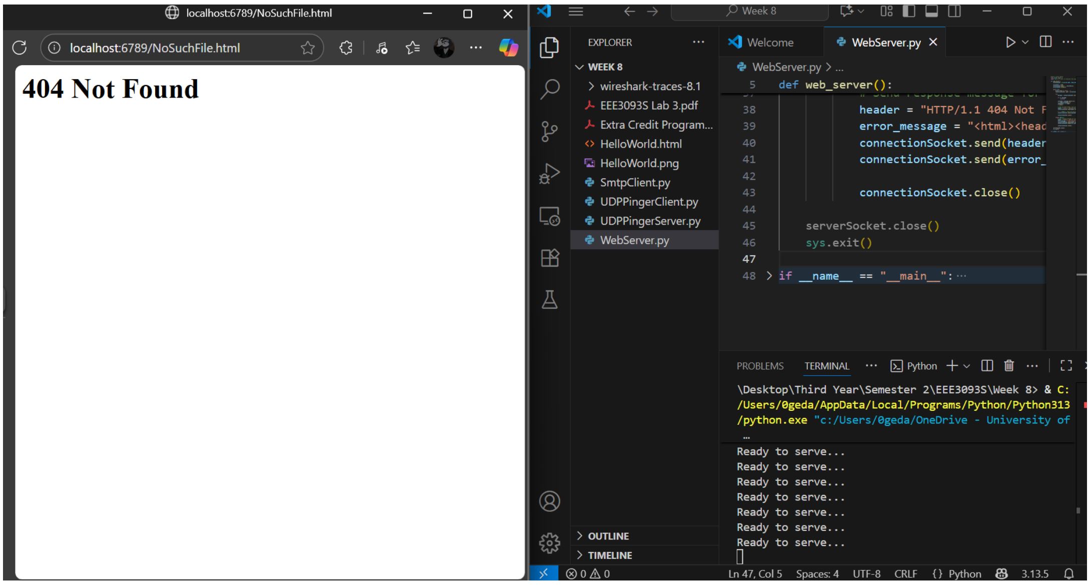

localhost:6789/HelloWorld.html

Hello, World!

```
def web_server():
            header = "HTTP/1.1 404 Not F
            error_message = "<html><head
            connectionSocket.send(header
            connectionSocket.send(error_

            connectionSocket.close()

    serverSocket.close()
    sys.exit()

if __name__ == "__main__": ···
```

```
PS C:\Users\0geda\OneDrive - University of Cape Town
\Desktop\Third Year\Semester 2\EEE3093S\Week 8> & C:
/Users/0geda/AppData/Local/Programs/Python/Python313
/python.exe "c:/Users/0geda/OneDrive - University of
 Cape Town/Desktop/Third Year/Semester 2/EEE3093S/We
ek 8/WebServer.py"
Ready to serve...
Ready to serve...
Ready to serve...
Ready to serve...
```

Figure 1: The browser successfully receives and displays the `HelloWorld.html` file from the server.


localhost:6789/NoSuchFile.html

404 Not Found

```
def web_server():
            header = "HTTP/1.1 404 Not F
            error_message = "<html><head
            connectionSocket.send(header
            connectionSocket.send(error_

            connectionSocket.close()

    serverSocket.close()
    sys.exit()

if __name__ == "__main__": ···
```

```
\Desktop\Third Year\Semester 2\EEE3093S\Week 8> & C:
/Users/0geda/AppData/Local/Programs/Python/Python313
/python.exe "c:/Users/0geda/OneDrive - University of
…
Ready to serve...
Ready to serve...
Ready to serve...
Ready to serve...
Ready to serve...
Ready to serve...
Ready to serve...
```

Figure 2: The server correctly sends a "404 Not Found" error message when the requested file does not exist.

Optional Exercise 1: Multithreaded Web Server

This section contains the implementation for the first optional exercise. The single-threaded web server from Task 1 was modified to handle multiple client requests simultaneously. This is achieved by creating a new thread for each incoming connection, allowing the main thread to continue listening for other clients.

Appendix C: Multithreaded Web Server Code

The code below defines a `handle_client` function that contains the logic for processing a single HTTP request. The main server loop accepts new connections and spawns a new thread to execute this function for each client.

```python
from socket import *
import sys
import threading

# This function handles a single client connection and runs in a separate thread.
def handle_client(connectionSocket, addr):
    print(f"Accepted connection from {addr}")
    try:
        message = connectionSocket.recv(1024).decode()
        if not message:
            connectionSocket.close()
            return

        filename = message.split()[1]
        f = open(filename[1:])
        outputdata = f.read()
        f.close()

        # Send HTTP OK header and the file content
        header = "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n"
        connectionSocket.send(header.encode())
        connectionSocket.send(outputdata.encode())

    except IOError:
        # Send 404 Not Found response
        header = "HTTP/1.1 404 Not Found\r\n\r\n"
        error_message = "<html><head></head><body><h1>404 Not Found</h1></body></html>"
        connectionSocket.send(header.encode())
        connectionSocket.send(error_message.encode())

    finally:
        # Close the connection with this specific client
        print(f"Closing connection with {addr}")
        connectionSocket.close()

def main():
    serverSocket = socket(AF_INET, SOCK_STREAM)
    serverPort = 6789
    serverSocket.bind(('', serverPort))
    serverSocket.listen(5) # Listen for up to 5 queued connections

    print(f"Server is ready and listening on port {serverPort}")

    while True:
        # The main thread waits for a new connection request
        connectionSocket, addr = serverSocket.accept()

        # A new thread is created to handle the client's request
        client_thread = threading.Thread(target=handle_client, args=(connectionSocket, addr))
        client_thread.start()

if __name__ == "__main__":
    main()
```

Listing 2: WebServer_Threaded.py - A multithreaded HTTP server

Demonstration Note

From a single client's perspective (like a web browser), the behavior of this server is identical to the single-threaded version. Its advanced capability would be demonstrated by having multiple, separate clients connect at the same time. The server's terminal would show it accepting and handling these connections concurrently, rather than one after the other.

Optional Exercise 2: HTTP Client

Appendix E: HTTP Client Code

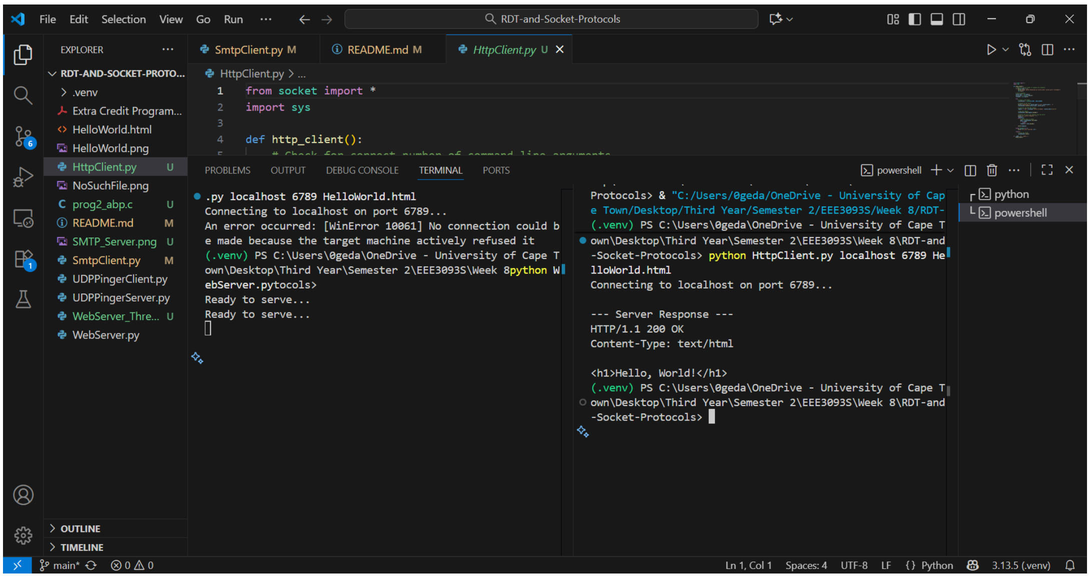
This is the implementation of a simple command-line HTTP client capable of sending a GET request to a server.

```python
from socket import *
import sys

def http_client():
    # Check for correct number of command-line arguments
    if len(sys.argv) != 4:
        print("Usage: python HttpClient.py <server_host> <server_port> <filename>")
        sys.exit()

    # Parse arguments
    server_host = sys.argv[1]
    server_port = int(sys.argv[2])
    filename = sys.argv[3]

    try:
        # Create a TCP socket
        clientSocket = socket(AF_INET, SOCK_STREAM)

        # Connect to the server
        print(f"Connecting to {server_host} on port {server_port}...")
        clientSocket.connect((server_host, server_port))

        # Construct the HTTP GET request
        request = f"GET /{filename} HTTP/1.1\r\nHost: {server_host}\r\n\r\n"

        # Send the request
        clientSocket.send(request.encode())

        # Receive and print the response from the server
        print("\n--- Server Response ---")
        response = ""
        while True:
            # Receive data in chunks
            data = clientSocket.recv(1024)
            if not data:
                break
            response += data.decode()

        print(response)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        # Close the socket
        clientSocket.close()

if __name__ == '__main__':
    http_client()
```

Listing 3: HttpClient.py

Appendix F: HTTP Client Demonstration

The screenshot below demonstrates the functionality of the HTTP client. The left terminal pane shows the web server running and ready to accept connections. The right terminal pane shows the client script being executed with command-line arguments, connecting to the server, and successfully printing the HTTP response and the content of the requested 'HelloWorld.html' file.



```
from socket import *
import sys

def http_client():
```

```
.py localhost 6789 HelloWorld.html
Connecting to localhost on port 6789...
An error occurred: [WinError 10061] No connection could b
e made because the target machine actively refused it
(.venv) PS C:\Users\0geda\OneDrive - University of Cape T
own\Desktop\Third Year\Semester 2\EEE3093S\Week 8python W
ebServer.pytocols>
Ready to serve...
Ready to serve...
```

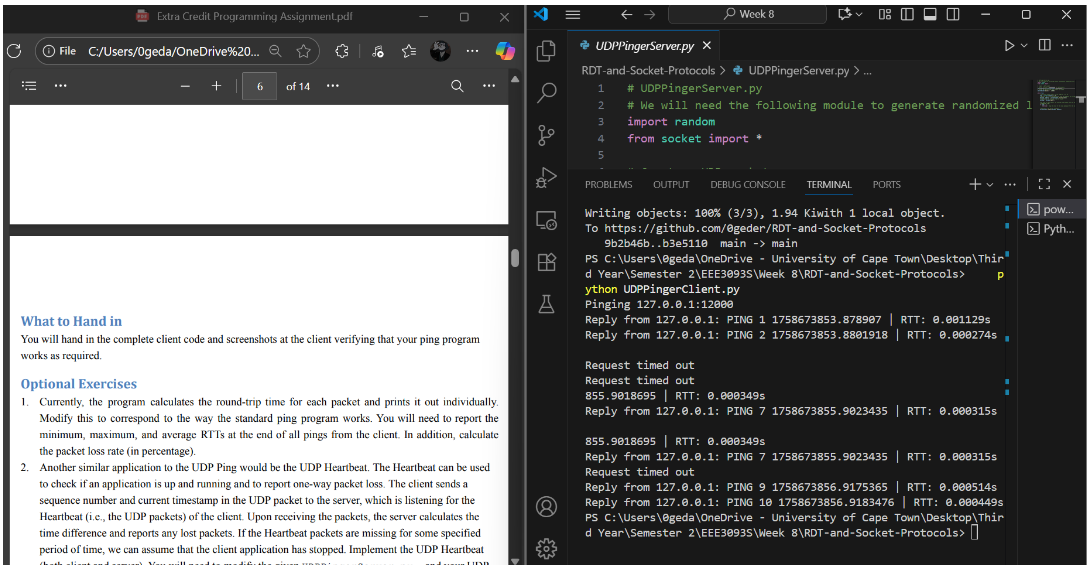
```
Protocols> & "C:/Users/0geda/OneDrive - University of Cap
e Town/Desktop/Third Year/Semester 2/EEE3093S/Week 8/RDT-
(.venv) PS C:\Users\0geda\OneDrive - University of Cape T
own\Desktop\Third Year\Semester 2\EEE3093S\Week 8\RDT-and
-Socket-Protocols> python HttpClient.py localhost 6789 He
lloWorld.html
Connecting to localhost on port 6789...

--- Server Response ---
HTTP/1.1 200 OK
Content-Type: text/html

<h1>Hello, World!</h1>
(.venv) PS C:\Users\0geda\OneDrive - University of Cape T
own\Desktop\Third Year\Semester 2\EEE3093S\Week 8\RDT-and
-Socket-Protocols>
```

Figure 3: Demonstration of `HttpClient.py` fetching a page from `WebServer.py`.

Task 2

```python
import time
from socket import *

def pinger_client():
    # Server details
    server_host = ’127.0.0.1’  # localhost
    server_port = 12000

    # Create a UDP socket
    clientSocket = socket(AF_INET, SOCK_DGRAM)

    # Set a timeout of 1 second for the socket
    clientSocket.settimeout(1)

    print(f"Pinging {server_host}:{server_port}")

    # Send 10 pings
    for sequence_number in range(1, 11):
        # Get the current time as a float
        start_time = time.time()

        # Format the message
        message = f’Ping {sequence_number} {start_time}’

        try:
            # Send the message to the server
            clientSocket.sendto(message.encode(), (server_host, server_port))

            # Wait to receive the reply from the server
            modifiedMessage, serverAddress = clientSocket.recvfrom(1024)

            # Get the time when reply was received
            end_time = time.time()

            # Calculate Round Trip Time (RTT)
            rtt = end_time - start_time

            # Print the response and RTT
            print(f’Reply from {serverAddress[0]}: {modifiedMessage.decode()} | RTT: {rtt:.6f}s’)

        except timeout:
            # If a ’timeout’ exception occurs, the packet was lost
            print(’Request timed out’)

    # Close the socket
    clientSocket.close()

if __name__ == ’__main__’:
    pinger_client()
```

Listing 4: UDPPingerClient.py


Extra Credit Programming Assignment.pdf
File C:/Users/0geda/OneDrive%20...
6
of 14
Week 8
UDPPingerServer.py
RDT-and-Socket-Protocols > UDPPingerServer.py > ...
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
pow...
Pyth...

What to Hand in

You will hand in the complete client code and screenshots at the client verifying that your ping program works as required.

Optional Exercises

1. Currently, the program calculates the round-trip time for each packet and prints it out individually. Modify this to correspond to the way the standard ping program works. You will need to report the minimum, maximum, and average RTTs at the end of all pings from the client. In addition, calculate the packet loss rate (in percentage).
2. Another similar application to the UDP Ping would be the UDP Heartbeat. The Heartbeat can be used to check if an application is up and running and to report one-way packet loss. The client sends a sequence number and current timestamp in the UDP packet to the server, which is listening for the Heartbeat (i.e., the UDP packets) of the client. Upon receiving the packets, the server calculates the time difference and reports any lost packets. If the Heartbeat packets are missing for some specified period of time, we can assume that the client application has stopped. Implement the UDP Heartbeat (both client and server). You will need to modify the given UDPPingerServer.py, and your UDP

```
# UDPPingerServer.py
# We will need the following module to generate randomized l
import random
from socket import *
```

```
Writing objects: 100% (3/3), 1.94 Kiwith 1 local object.
To https://github.com/0geder/RDT-and-Socket-Protocols
   9b2b46b..b3e5110  main -> main
PS C:\Users\0geda\OneDrive - University of Cape Town\Desktop\Thir
d Year\Semester 2\EEE3093S\Week 8\RDT-and-Socket-Protocols>     p
ython UDPPingerClient.py
Pinging 127.0.0.1:12000
Reply from 127.0.0.1: PING 1 1758673853.878907 | RTT: 0.001129s
Reply from 127.0.0.1: PING 2 1758673853.8801918 | RTT: 0.000274s

Request timed out
Request timed out
855.9018695 | RTT: 0.000349s
Reply from 127.0.0.1: PING 7 1758673855.9023435 | RTT: 0.000315s

855.9018695 | RTT: 0.000349s
Reply from 127.0.0.1: PING 7 1758673855.9023435 | RTT: 0.000315s
Request timed out
Reply from 127.0.0.1: PING 9 1758673856.9175365 | RTT: 0.000514s
Reply from 127.0.0.1: PING 10 1758673856.9183476 | RTT: 0.000449s
PS C:\Users\0geda\OneDrive - University of Cape Town\Desktop\Thir
d Year\Semester 2\EEE3093S\Week 8\RDT-and-Socket-Protocols> 
```

Figure 4: The terminal output of the `UDPPingerClient.py` script. This demonstrates the client successfully receiving replies, calculating the Round Trip Time (RTT), and handling simulated packet loss with 'Request timed out' messages.

Task 3: SMTP Mail Client

Appendix C: Task 3 SMTP Mail Client Code

The following is the complete Python code for the SMTP client, designed to connect to a mail server and send a text-based email. For testing purposes, it was configured to connect to a local debugging server.

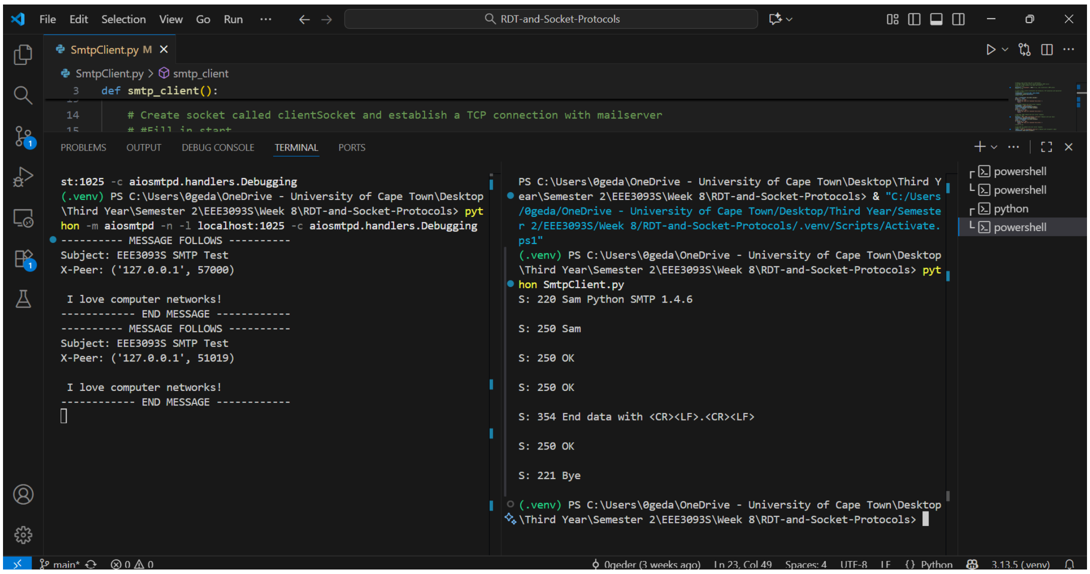
```
from socket import *

def smtp_client():
    msg = "\r\n I love computer networks!"
    endmsg = "\r\n.\r\n"

    # Choose a mail server and call it mailserver
    # You MUST replace this with a valid, accessible SMTP server.
    # Port 25 is the standard, but many ISPs block it.
    # #Fill in start
    mailserver = ("localhost", 1025) # e.g., your university’s SMTP server
    # #Fill in end

    # Create socket called clientSocket and establish a TCP connection with mailserver
    # #Fill in start
    clientSocket = socket(AF_INET, SOCK_STREAM)
    clientSocket.connect(mailserver)
    # #Fill in end

    recv = clientSocket.recv(1024).decode()
    print("S:", recv)
    if recv[:3] != ’220’:
        print(’220 reply not received from server.’)
        return

    # Send HELO command and print server response.
    heloCommand = ’HELO Alice\r\n’
    clientSocket.send(heloCommand.encode())
    recv1 = clientSocket.recv(1024).decode()
    print("S:", recv1)
    if recv1[:3] != ’250’:
        print(’250 reply not received from server.’)
        return

    # Send MAIL FROM command and print server response.
    # #Fill in start
    mailFrom = "MAIL FROM:<samson@test.com>\r\n" # Replace with your email
    clientSocket.send(mailFrom.encode())
    recv2 = clientSocket.recv(1024).decode()
    print("S:", recv2)
    if recv2[:3] != ’250’:
        print(’250 reply not received from server.’)
        return
    # #Fill in end

    # Send RCPT TO command and print server response.
    # #Fill in start
    rcptTo = "RCPT TO:<okuthe@test.com>\r\n" # Replace with recipient’s email
    clientSocket.send(rcptTo.encode())
    recv3 = clientSocket.recv(1024).decode()
    print("S:", recv3)
    if recv3[:3] != ’250’:
        print(’250 reply not received from server.’)
        return
    # #Fill in end

    # Send DATA command and print server response.
    # #Fill in start
    dataCommand = "DATA\r\n"
    clientSocket.send(dataCommand.encode())
    recv4 = clientSocket.recv(1024).decode()
    print("S:", recv4)
    if recv4[:3] != ’354’:
        print(’354 reply not received from server.’)
        return
    # #Fill in end
```

```python

    # Send message data.
    # #Fill in start
    # You can add email headers here for a proper email
    subject = "Subject: EEE3093S SMTP Test\r\n"
    clientSocket.send(subject.encode())
    clientSocket.send(msg.encode())
    # #Fill in end

    # Message ends with a single period.
    # #Fill in start
    clientSocket.send(endmsg.encode())
    recv5 = clientSocket.recv(1024).decode()
    print("S:", recv5)
    if recv5[:3] != '250':
        print('250 reply not received from server.')
        return
    # #Fill in end

    # Send QUIT command and get server response.
    # #Fill in start
    quitCommand = "QUIT\r\n"
    clientSocket.send(quitCommand.encode())
    recv6 = clientSocket.recv(1024).decode()
    print("S:", recv6)
    if recv6[:3] != '221':
        print('221 reply not received from server.')
    # #Fill in end

    clientSocket.close()

if __name__ == '__main__':
    smtp_client()
```

Listing 5: SmtpClient.py - A simple SMTP client

Appendix D: Task 3 Demonstration Screenshot

The screenshot below shows the output of the local Python debugging SMTP server. This output serves as proof that the client successfully connected and transmitted the email headers and body, which the server then printed to the console.



```python
def smtp_client():

    # Create socket called clientSocket and establish a TCP connection with mailserver
    # #Fill in start
```

```
st:1025 -c aiosmtpd.handlers.Debugging
(.venv) PS C:\Users\0geda\OneDrive - University of Cape Town\Desktop
\Third Year\Semester 2\EEE3093S\Week 8\RDT-and-Socket-Protocols> pyt
hon -m aiosmtpd -n -l localhost:1025 -c aiosmtpd.handlers.Debugging
---------- MESSAGE FOLLOWS ----------
Subject: EEE3093S SMTP Test
X-Peer: ('127.0.0.1', 57000)

 I love computer networks!
------------ END MESSAGE ------------
---------- MESSAGE FOLLOWS ----------
Subject: EEE3093S SMTP Test
X-Peer: ('127.0.0.1', 51019)

 I love computer networks!
------------ END MESSAGE ------------
```

```
PS C:\Users\0geda\OneDrive - University of Cape Town\Desktop\Third Y
ear\Semester 2\EEE3093S\Week 8\RDT-and-Socket-Protocols> & "C:/Users
/0geda/OneDrive - University of Cape Town/Desktop/Third Year/Semeste
r 2/EEE3093S/Week 8/RDT-and-Socket-Protocols/.venv/Scripts/Activate.
ps1"
(.venv) PS C:\Users\0geda\OneDrive - University of Cape Town\Desktop
\Third Year\Semester 2\EEE3093S\Week 8\RDT-and-Socket-Protocols> pyt
hon SmtpClient.py
S: 220 Sam Python SMTP 1.4.6

S: 250 Sam

S: 250 OK

S: 250 OK

S: 354 End data with <CR><LF>.<CR><LF>

S: 250 OK

S: 221 Bye

(.venv) PS C:\Users\0geda\OneDrive - University of Cape Town\Desktop
\Third Year\Semester 2\EEE3093S\Week 8\RDT-and-Socket-Protocols>
```

Figure 5: Output from the local SMTP debugging server, verifying the receipt of the email sent by `SmtpClient.py`.