Homework 2: Socket Programming

Yousef Alaa Awad

February 16, 2026

Part 1: Run TCP Programs

Below is execution screenshot, followed by the given TCP program codes.

Execution Screenshot

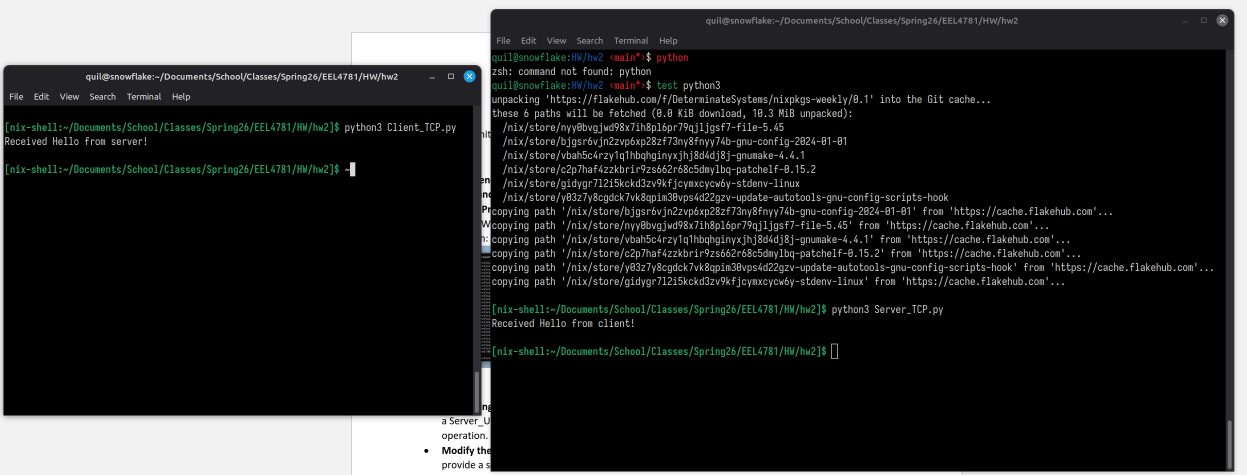

Figure 1: Server_TCP and Client_TCP in operation.

Server_TCP.py

```python
import socket

# Define IPs and Ports
serverPort = 12000
serverHost = "127.0.0.1"

# Define sockets for server and client
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind socket to the corresponding IP and Port
serverSocket.bind((serverHost, serverPort))

# Waiting connection
serverSocket.listen(1)

# Accept a connection with the client
clientSocket = serverSocket.accept()[0]

# Receive data from the client
data = clientSocket.recv(1024).decode()
print("Received", data)

# Send a response to the client
message = "Hello from server!"
if clientSocket.send(message.encode()) != len(message):
```

```
    print(f"Message sent had error and could not send all {len(message)} characters...")

# Close the connection
clientSocket.close()

# Close the service
serverSocket.close()
```

Client_TCP.py

```
import socket

# Define IPs and Ports
clientPort = 12001
clientHost = "127.0.0.2"
serverPort = 12000
serverHost = "127.0.0.1"

# Define sockets for server and client
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind socket to the corresponding IP and Port
clientSocket.bind((clientHost, clientPort))

# Connect to the server
clientSocket.connect((serverHost, serverPort))

# Send data to the server
message = "Hello from client!"
if clientSocket.send(message.encode()) != len(message):
    print(f"Message sent had error and could not send all {len(message)} characters...")

# Receive data from the server
data = clientSocket.recv(1024)
print("Received", data.decode())

# Close the connection
clientSocket.close()
```

Part 2: Developing UDP Programs

Using the TCP codes as references, the following UDP Client and Server programs were created.

Server_UDP.py

```python
import socket

# Define IPs and Ports
serverPort: int = 12000
serverHost: str = '127.0.0.1'

# Define socket for server
# UDP uses SOCK_DGRAM instead of SOCK_STREAM
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind socket to the corresponding IP and Port
serverSocket.bind((serverHost, serverPort))

print("The server is ready to receive")

# Receive data from the client
# recvfrom() blocks until data is received
# It returns (message, clientAddress) so we know where to send the reply
message, clientAddress = serverSocket.recvfrom(2048)
print('Received', message.decode())

# Send a response to the client
# We use sendto() and use the clientAddress we got from recvfrom()
response = 'Hello from server!'
serverSocket.sendto(response.encode(), clientAddress)

# Close the socket
serverSocket.close()
```

Client_UDP.py

```python
import socket

# Define IPs and Ports
clientPort: int = 12001
clientHost: str = '127.0.0.1'
serverPort: int = 12000
serverHost: str = '127.0.0.1'

# Define socket for client
# UDP uses SOCK_DGRAM instead of SOCK_STREAM
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind socket to the corresponding IP and Port
clientSocket.bind((clientHost, clientPort))
```

```
15
16 # Send data to the server
17 # UDP is connectionless, so we use sendto() with the destination address
18 # We do not use .connect()
19 message = 'Hello from client!'
20 clientSocket.sendto(message.encode(), (serverHost, serverPort))
21
22 # Receive data from the server
23 # recvfrom returns a tuple: (data, address)
24 data, serverAddress = clientSocket.recvfrom(2048)
25 print('Received', data.decode())
26
27 # Close the socket
28 clientSocket.close()
```

Execution Screenshot

Figure 2: Server_UDP and Client_UDP in operation.

Part 3: Modify the TCP Server

The `Server_TCP.py` was modified to create an always-on server.

Modified Code

Example:

```python
import socket

# Define IPs and Ports
serverPort: int = 12000
serverHost: str = "127.0.0.1"

# Define sockets for server and client
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Setting the socket to allow the reuse of addresses even after being used before.
serverSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind socket to the corresponding IP and Port
serverSocket.bind((serverHost, serverPort))

# Waiting connection
serverSocket.listen(1)

try:
    while True:
        # Accept a connection with the client
        clientSocket, addr = serverSocket.accept()
        print(f"Connection was established with {addr}")

        # Receive data from the client
        data = clientSocket.recv(1024).decode()
        print(f"Received: {data}")

        # Send a response to the client
        message = "Hello from server!"
        if clientSocket.send(message.encode()) != len(message):
            print("Message sent was not correct amount of bytes/characters ...")

        clientSocket.close()
        print("Client disconnected. Waiting for next connection...")

except KeyboardInterrupt:
    print("Server is shutting down...")

finally:
    # Close the service
    serverSocket.close()
```

Execution Screenshot

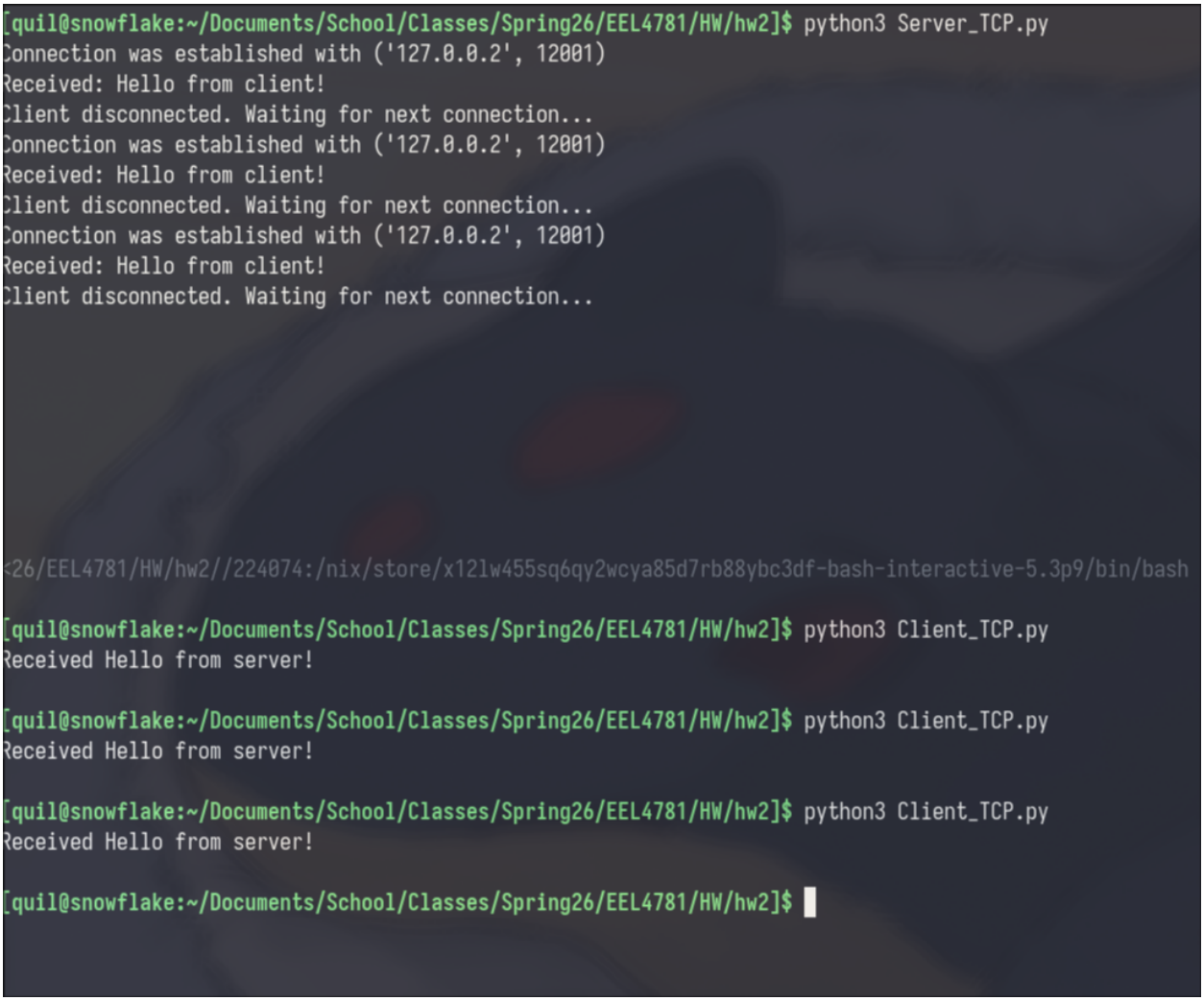

Figure 3: Modified Always-On TCP Server in operation.

Part 4: Questions

1. **(5 pts) The Server_TCP code does not explicitly declare the client's IP address and port in its initial setup. How does it send messages to the Client_TCP?**

   The server sends messages back to the client using a dedicated connection socket object returned by the `socket.accept()` method. This `clientSocket` instance internally stores the state of the specific connection, including the client's IP and port. When the server calls `clientSocket.send()`, the underlying operating system uses this stored connection to route the message to the correct destination, therefore eliminating the need for the Python code to have to explicitly store the address in a variable (if not wanted).

2. **(5 pts) Is it possible to avoid using serverSocket.bind or clientSocket.bind in these programs? Explain your answer.**

   - **For the Client: Yes.**
     In `Client_TCP.py` and `Client_UDP.py`, `bind()` is currently used to fix the source port to 12001. If this line is removed, the operating system will automatically assign an available port and use its local IP when the client calls `connect()` (TCP) or `sendto()` (UDP). The communication will still work correctly, however, due to the server determining the client's return address dynamically upon receiving the initial packet.
   - **For the Server: No.**
     In the server programs, `bind()` is necessary. The server must listen on a specific, known port (12000 in the python files above) so that clients know exactly where to send their connection requests or packets. If `bind()` were removed, the operating system would then assign a random port number, and the clients (which are hardcoded to target port 12000) would be unable to connect to the server.

3. **(15 pts) Summarize the differences between TCP and UDP programs.**

   There were 3 major differences that I found after writing the programs. First was the with the socket types. TCP programs utilize `SOCK_STREAM`, whereas the UDP programs employ `SOCK_DGRAM`. Secondly, the connection setup is also different. TCP is connection-oriented, meaning the server has to use `listen()` and `accept()[0]` while the client simply calls `connect()`. UDP, on the other hand, is connectionless and therefore involves no initial handshake and avoids these connection-specific methods allowing for you to just spit out messages whenever. Thirdly, when sending data, TCP relies on `send()`, where the destination is implicit in the connected socket, and UDP uses `sendto(msg, addr)`, requiring the destination address to be explicitly included in every packet. Along those line, when receiving data, TCP uses `recv()` (returning only the data string), whereas UDP uses `recvfrom()`, and returns a tuple containing both the message and the client address.