Department of Electrical, Electronic and Computer Engineering

Cape Peninsula University of Technology

Lab 3

Socket Programming TCP – Five-Layer Model

Name, Surname	Mnqobi Jeza
Student Number:	230878369
Date:	20 February 2025

TCP Socket Programming Practical

Objective: Students will learn how to implement a simple TCP-based client-server communication system in Python. Using the TCP protocol, they will explore how data is sent and received over a network.

Background: TCP (Transmission Control Protocol) is a connection-oriented transport-layer protocol that ensures reliable data transmission, sequencing, and error correction. Unlike UDP, TCP establishes a connection before transmitting data and ensures that data is received in order.

Reference: Section 2.7.2 in *Computer Networking: A Top-Down Approach, Seventh Edition* by Kurose and Ross.

Requirements:

1. Python 3 installed
2. Basic understanding of socket programming
3. Two machines or two terminals on the same machine

Task 1: Implementing the TCP Server

The server should:

- Create a TCP socket
- Bind it to a specific port

- Listen for incoming connections
- Accept a client connection and receive a message
- Send a response back to the client

tcp_server.py > ...

```python
from socket import *

serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)  # Use SOCK_STREAM for TCP

# Bind the server to the specified port
serverSocket.bind(('127.0.0.1', serverPort))

# Start listening for incoming connections (allow up to 1 pending connection)
serverSocket.listen(1)
print("The server is ready to receive...")

while True:
    # Accept an incoming client connection
    connectionSocket, clientAddress = serverSocket.accept()
    print(f"Connection established with {clientAddress}")

    # Receive message from client
    message = connectionSocket.recv(1024).decode()
    print(f"Received from client: {message}")

    # Process message (convert to uppercase)
    modifiedMessage = message.upper()

    # Send response back to client
    connectionSocket.send(modifiedMessage.encode())

    # Close the connection with the client
    connectionSocket.close()
```

Task 2: Implementing the TCP Client

The client should:

- Create a TCP socket
- Connect to the server
- Send a message to the server
- Receive a response from the server
- Print the response

tcp_client.py > ...

```python
from socket import *

serverName = '127.0.0.1'  # Use localhost or actual server IP
serverPort = 12000

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

# Establish connection to the server
clientSocket.connect((serverName, serverPort))

# Get user input
message = input("Input lowercase sentence: ")

# Send message to server
clientSocket.send(message.encode())

# Receive modified message from server
modifiedMessage = clientSocket.recv(1024).decode()

# Print the response from the server
print("From Server:", modifiedMessage)

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

Task 3: Running the Experiment

1. Run the server first:
   - Open a terminal and execute `python tcp_server.py o`

   The server will start listening for connections
2. Run the client:
   - Open another terminal and execute `python tcp_client.py`
   - Observe the messages sent and received
3. Modify the client to send multiple messages and analyse the response.
4. Test sending data from different machines on the same network by changing `SERVER_IP`.

Questions for Students:

1. What happens if the client tries to connect when the server is not running?
   If the client tries to connect when the **TCP server is not running**, it will fail with a **connection error** because there is no process listening on the specified IP and port.

2. How does TCP differ from UDP regarding reliability and connection management?
   UDP is a connectionless, unreliable protocol that transmits data without guaranteeing delivery, order, or error correction, making it faster but less dependable. TCP is a dependable, connection-oriented protocol that uses error checking, acknowledgments, and retransmissions to ensure that data is delivered accurately and in order. It is best suited for applications that require accuracy, such as file transfers and web browsing, while UDP works well for real-time applications, such as gaming and video streaming.

3. Implement error handling in both client and server.

```
except OSError as e:
    print(f"Server error: {e}")

except KeyboardInterrupt:
    print("\nServer shutting down gracefully...")
```

For your interest:

1. Modify the server to handle multiple clients simultaneously.

END