University of Dhaka

CSE:3111 Computer Networking Lab

Lab Report

Title: Implementing File transfer using Socket Programming and HTTP GET/POST requests

Diponker Roy
Roll: 28
and
Abdullah Ashik
Roll: 32

Submitted to:

Dr. Md. Abdur Razzaque
Dr. Muhammad Ibrahim
Dr. Md. Redwan Ahmed Rizvee
Dr. Md. Mamun Or Rashid

Submitted on: 2024-01-26

1 Introduction

In the dynamic landscape of computer networking, efficient **file transfer** mechanisms are essential for seamless **data exchange** between systems. This laboratory experiment focuses on the implementation of **file transfer** using two distinct approaches: **Socket Programming** and **HTTP GET/POST requests**. **Socket Programming**, a low-level network communication interface, enables the creation of robust communication channels between processes on different devices. On the other hand, **HTTP**, particularly through **GET** and **POST** requests, provides a higher-level abstraction for **data transmission**, integral to the functioning of the World Wide Web. This experiment aims to explore and compare the intricacies of these two methods, shedding light on their respective strengths, weaknesses, and practical implications in real-world scenarios. Through hands-on implementation and analysis, participants will gain valuable insights into the fundamental principles underlying these approaches, facilitating informed decision-making in selecting the most suitable method for specific **file transfer** requirements.

1.1 Objectives

The primary **objectives** of this laboratory experiment are twofold. First, to comprehensively understand the foundational principles of **Socket Programming**, exploring the creation and utilization of **sockets** for efficient **data transfer** between networked systems. The implementation of a **file transfer system** using **Socket Programming** will be a key focus, delving into the intricacies of establishing reliable **communication channels** and facilitating the exchange of **files** between a **server** and a **client**. Second, the experiment aims to explore the fundamentals of **HTTP**, with a specific emphasis on the **GET** and **POST** request methods. Participants will then implement a **file transfer mechanism** utilizing these **HTTP** methods, scrutinizing the higher-level abstraction they provide compared to **Socket Programming**. Furthermore, the experiment involves a comparative analysis of the strengths, weaknesses, and practical implications of **file transfer mechanisms** implemented through **Socket Programming** and **HTTP GET/POST requests**. This comparative evaluation will enable participants to discern the

suitability of each approach for different scenarios, ultimately providing practical insights into the underlying principles of **network communication**, **file transfer protocols**, and **HTTP methods**, facilitating informed decision-making for specific application requirements.

2 Theoritical Background

2.1 Socket Programming

Socket Programming serves as a fundamental concept in network communication, providing a mechanism for processes on different devices to establish communication channels. In this context, a **socket** represents an endpoint for sending or receiving data across a computer network. Through the use of sockets, applications can communicate seamlessly, either locally or over a network, utilizing various protocols such as Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). The fundamental steps in Socket Programming involve socket creation, binding to an address and port, listening for incoming connections (for server applications), and establishing connections (for client applications). By understanding these principles, participants gain insight into the low-level details of data exchange between networked systems.

2.2 HTTP and File Transfer

Hypertext Transfer Protocol (HTTP) is the foundation of data communication on the World Wide Web. It operates as a request-response protocol, where clients make requests, and servers respond with the requested information. HTTP encompasses various methods, with **GET** and **POST** being prominent for retrieving and sending data, respectively. When it comes to file transfer, HTTP provides a platform-independent and language-independent means for exchanging files. The GET method retrieves files, while the POST method sends files to the server. This higher-level abstraction simplifies the implementation of web-based file transfer systems. Participants will explore these HTTP methods and their application in facilitating file transfers within the context of web communication.

2.3 comparative Analysis

The experiment involves a comparative analysis of Socket Programming and HTTP-based file transfer mechanisms. Socket Programming, being a lower-level approach, offers fine-grained control over data exchange but requires explicit handling of details like connection management. On the other hand, HTTP provides a more abstract and standardized approach suitable for web-based applications, offering simplicity and interoperability. The analysis aims to highlight the strengths and weaknesses of each approach, considering factors such as efficiency, ease of implementation, and suitability for different use cases. This theoretical foundation will guide participants in evaluating the trade-offs between the two methods and making informed decisions based on specific application requirements.

By delving into these theoretical aspects, participants will acquire a solid understanding of the principles behind Socket Programming and HTTP, laying the groundwork for the practical implementation and comparison of file transfer mechanisms in the subsequent sections of the laboratory experiment.

3 Methodology

The implementation of file transfer mechanisms through Socket Programming and HTTP GET/POST requests involves a step-by-step approach to ensure a comprehensive understanding of the underlying principles and practical application. The following methodology outlines the key steps of the laboratory experiment:

1. **Setup and Environment Preparation:**
   - Set up a development environment with the necessary tools and libraries for programming in a language that supports Socket Programming (e.g., Python) and HTTP requests (e.g., Python with **requests** library).
   - Ensure network connectivity between the server and client systems.
2. **Socket Programming Implementation:**

- Create two separate scripts for the server and client using the chosen programming language.
- Implement socket creation, binding, listening (for the server), and connection establishment (for the client) in the respective scripts.
- Develop file transfer logic by specifying protocols for data exchange, error handling, and connection termination.

3. **HTTP GET/POST Implementation:**
   - Choose a web framework or library, such as **FastAPI for Python**, to simplify HTTP server implementation.
   - Create an HTTP server script using **FastAPI** to handle GET and POST requests for file transfer.
   - Design and implement client-side scripts to send GET requests for file retrieval and POST requests for file uploads.
4. **File Preparation:**
   - Prepare sample files for testing the **file transfer mechanisms**.
   - Ensure that these files vary in size and content to comprehensively assess the performance of the implemented solutions.
5. **Execution and Testing:**
   - Execute the **Socket Programming** scripts on the server and client systems.
   - Initiate **file transfers** and monitor the communication process.
   - Execute the **HTTP server** script (implemented with **FastAPI**) and test **GET** and **POST requests** for **file transfer**.
   - Observe and record the performance metrics, such as transfer speed, latency, and resource utilization.
6. **Comparative Analysis:**
   - Analyze the obtained results from both **Socket Programming** and **HTTP** implementations.

- Evaluate the strengths and weaknesses of each approach in terms of simplicity, efficiency, and flexibility.
- Consider factors such as ease of implementation, scalability, and suitability for different network conditions.

7. Task 1: File Transfer via Socket Programming

1. Implement a simple file server that listens for incoming connections on a specified port. The server should be able to handle multiple clients simultaneously.
2. When a client connects to the server, the server should prompt the client for the name of the file they want to download.
3. The server should then locate the file on disk and send it to the client over the socket.
4. Implement a simple file client that can connect to the server and request a file. The client should save the file to disk once it has been received.

Server.py Code

```
import socket
import os
import time
import threading


HEADER = 64
PORT = 5004
FORMAT = 'utf-8'
DISCONNECTMESSAGE = "!DISCONNECT"
SERVER = socket.gethostbyname(socket.gethostname())
ADDR = (SERVER, PORT)


server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(ADDR)
```

```
def handleClient(conn, addr):
    print(f"[NEW CONNECTION] {addr} connected.")

    connected = True

    while connected:
        fileName = input("Enter the name: ")
        conn.send(fileName.encode(FORMAT))
        fileSize = os.path.getsize(fileName)

        conn.send(str(fileSize).encode(FORMAT))
        choice = conn.recv(1024).decode(FORMAT)
        if choice == "n":
            print("File transfer cancelled")
            conn.send(DISCONNECTMESSAGE.encode(FORMAT))
            conn.close()
            print(f"[DISCONNECTED] {addr} has been disconnected")
            break
        time.sleep(1)
        print(f"[SENDING] Sending {fileName} to {addr}")
        with open(fileName, "rb") as f:
            start = time.time()
            bytesToSend = f.read(1024)
            conn.send(bytesToSend)
            while bytesToSend != "":
                bytesToSend = f.read(1024)
                conn.send(bytesToSend)
            end = time.time()

        print(f"time taken to send the file: {end - start} seconds")

        print(f"[SENT] {fileName} has been sent")
        connected = False
        conn.send(DISCONNECTMESSAGE.encode(FORMAT))
        conn.close()
        print(f"[DISCONNECTED] {addr} has been disconnected")
```

```
            break

        conn.close()


def start():
    server.listen()
    print(f"[LISTENING] Server is listening on {SERVER}")
    while True:
        conn, addr = server.accept()
        thread = threading.Thread(target=handleClient, args=(conn, addr))
        thread.start()
        print(f"[ACTIVE CONNECTIONS] {threading.activeCount() - 1}")


print("[STARTING] server is starting...")
start()
```

Client.py Code

```
import os
import socket
import time

my_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostbyname(socket.gethostname())
format = "utf-8"

try:
    my_socket.connect((host, 5004))
    print("Connected to the server...")
except Exception as e:
    print(f"Connection failed: {e}")
    exit(0)

filename = my_socket.recv(1024).decode(format)
```

```
filesize = my_socket.recv(1024).decode(format)

str = f"File {filename} of size {filesize} bytes is being received. Do you wa
choice = input(str)
my_socket.send(choice.encode(format))
if choice == "n":
    exit(0)


with open("./received_" + filename, 'wb') as file:

    c = 0
    starttime = time.time()

    while c < int(filesize):
        data = my_socket.recv(1024)
        c += len(data)
        file.write(data)
        print(f"Received {c} bytes")

    endtime = time.time()
    print(f"Time taken to receive the file: {endtime - starttime} seconds")

my_socket.close()
exit(0)
```

Experimental Result

The provided Python code establishes a server for file transfer using sockets. Here's a breakdown of the key components:

- **Socket Setup:** The server sets up a socket using the 'socket' module and binds it to a specific address and port.
- **Client Handling:** The server handles incoming client connections using threading to manage multiple clients simultaneously. The server is designed to handle multiple clients

concurrently, as indicated by the [ACTIVE CONNECTIONS] output, which increments as new clients connect.

- **File Transfer:** Upon connection, the server prompts each client for a file name, sends the file's name and size to the respective client, and transfers the file in chunks.
- **Disconnection:** The server provides an option to cancel file transfer, and both the client and server disconnect after the transfer is complete or cancelled. Each client connection is tracked, and the [DISCONNECTED] message indicates when a client disconnects.

The code demonstrates a basic file transfer mechanism between a server and multiple clients concurrently. Further analysis could focus on performance metrics, error handling, and potential improvements for scalability.

```
user@user-Inspiron-3501:~/Desktop/networking/Networking-Lab/Lab3/Task1$ python3 server.py
[STARTING] server is starting...
[LISTENING] Server is listening on 127.0.1.1
[NEW CONNECTION] ('127.0.0.1', 56686) connected.
Enter the name: /home/user/Desktop/networking/Networking-Lab/Lab3/Task1/server.py:67: DeprecationWarning:
activeCount() is deprecated, use active_count() instead
  print(f"[ACTIVE CONNECTIONS] {threading.activeCount() - 1}")
[ACTIVE CONNECTIONS] 1
assignment.pdf
[SENDING] Sending assignment.pdf to ('127.0.0.1', 56686)
[NEW CONNECTION] ('127.0.0.1', 37404) connected.
Enter the name: [ACTIVE CONNECTIONS] 2
[NEW CONNECTION] ('127.0.0.1', 43080) connected.
[ACTIVE CONNECTIONS] 3
assignment.pdf
Enter the name: assignment.pdf
[SENDING] Sending assignment.pdf to ('127.0.0.1', 37404)
File transfer cancelled
[DISCONNECTED] ('127.0.0.1', 43080) has been disconnected
```

Figure 1: Server Output

The provided Python client code connects to the server and initiates the file transfer process. Here's a breakdown of the key components:

- **Socket Setup:** The client sets up a socket using the 'socket' module and attempts to connect to the server at a specified address and port.

- **File Retrieval:** Upon successful connection, the client receives the file name and size from the server and prompts the user whether to continue with the file transfer.
- **User Choice:** The user can choose whether to proceed with the file transfer by entering 'y' or cancel the transfer by entering 'n'.
- **File Transfer:** If the user chooses to continue, the client receives the file in chunks and writes them to the local file system. The client displays progress information during the transfer.
- **Disconnection:** After the file transfer is complete or cancelled, the client closes the connection to the server and exits.

The client code provides an interactive approach, allowing the user to decide whether to download the file from the server. Further analysis could focus on additional features, error handling, and user experience improvements.

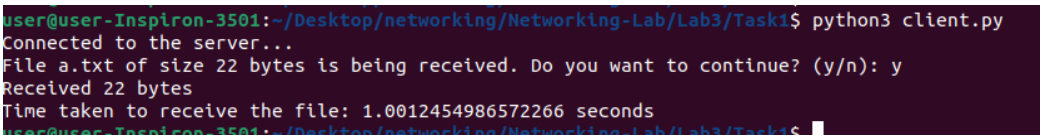

Figure 2: client accept the download request Output

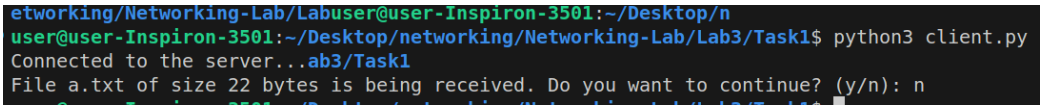

Figure 3: Client reject the download request Output

8. Task 2: File Transfer via HTTP

1. Implement a simple HTTP file server that listens for incoming connections on a specified port. The server should be able to handle multiple clients simultaneously.

2. When a client sends a GET request to the server with the path of the file they want to download, the server should locate the file on disk and send it to the client with the appropriate HTTP response headers. The client should save the file to disk once it has been received.

3. When a client sends a POST request to the server with a file, the server saves it.

Server.py Code

```
#run it with uvicorn server:app --host <ip address> --port 8000 --reload


from fastapi import FastAPI, File, UploadFile
from fastapi.responses import HTMLResponse, FileResponse


app = FastAPI()

uploadFile = []


@app.get("/", response_class=HTMLResponse)
def read_root():
    return ("ok")


@app.get("/download/{fileName}")
def download_file(fileName: str):
    return FileResponse(fileName, media_type="application/octet-stream", file


@app.post("/upload")
def upload_file(file: UploadFile = File(...)):
    with open(file.filename, "wb") as buffer:
        buffer.write(file.file.read())
    uploadFile.append(file.filename)
    return {"filename": file.filename}


@app.get("/list")
def list_all_files():
    return {"files": uploadFile}
```

Client.py Code

```
#change the localhost to the server ip address for accessing lab pc or same r
import requests

localhost ="localhost"

def download_file(url):
    response = requests.get(url)

    if response.status_code == 200:
        with open('downloaded_file.pdf', 'wb') as f:
            f.write(response.content)
        print("File downloaded successfully.")
    else:
        print(f"Failed to download file. Status code: {response.status_code}"


def list_files(url):
    response = requests.get(url)

    if response.status_code == 200:
        print(response.json())
    else:
        print(f"Failed to list files. Status code: {response.status_code}")


def upload_file(url, file_path):
    with open(file_path, 'rb') as f:
        files = {'file': f}
        response = requests.post(url, files=files)

    if response.status_code == 200:
        print("File uploaded successfully.")
```

```
    else:
        print(f"Failed to upload file. Status code: {response.status_code}")


if __name__ == "__main__":
    while True:
        print("1. List files")
        print("2. Upload file")
        print("3. Download file")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            list_files(f"http://{localhost}:8000/list")
        elif choice == 2:
            file_path = input("Enter the file path: ")
            upload_file(f"http://{localhost}:8000/upload", file_path)
        elif choice == 3:
            file_name = input("Enter the file name: ")
            download_file(f"http://{localhost}:8000/download/{file_name}")
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

Experimental Result

The provided FastAPI server code establishes an HTTP server capable of handling file uploads and downloads. Here's a breakdown of the key components:

- **Root Endpoint:** The root endpoint ("/") returns a simple response indicating the server's status.
- **Download Endpoint:** The "/download/fileName" endpoint allows clients to download files by specifying the desired file name.

- **Upload Endpoint:** The "/upload" endpoint handles file uploads. Clients can upload files, and the server saves them to the local file system.
- **List Endpoint:** The "/list" endpoint provides a list of all uploaded files on the server.

The server is designed to be accessible via any web browser, including local browsers. Users can upload files, download them, and view the list of available files.

```
INFO:     Application shutdown complete.
INFO:     Finished server process [276700]
INFO:     Started server process [276752]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
WARNING:  WatchFiles detected changes in 'client.py'. Reloading...
INFO:     Shutting down
INFO:     Waiting for application shutdown.
INFO:     Application shutdown complete.
INFO:     Finished server process [276752]
INFO:     Started server process [278788]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
WARNING:  WatchFiles detected changes in 'client.py'. Reloading...
INFO:     Shutting down
INFO:     Waiting for application shutdown.
INFO:     Application shutdown complete.
INFO:     Finished server process [278788]
INFO:     Started server process [278850]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Figure 4: FastAPI Server Output

The provided Python client code interacts with the FastAPI server, enabling users to perform file download, upload, and listing operations. Here's a breakdown of the key components:

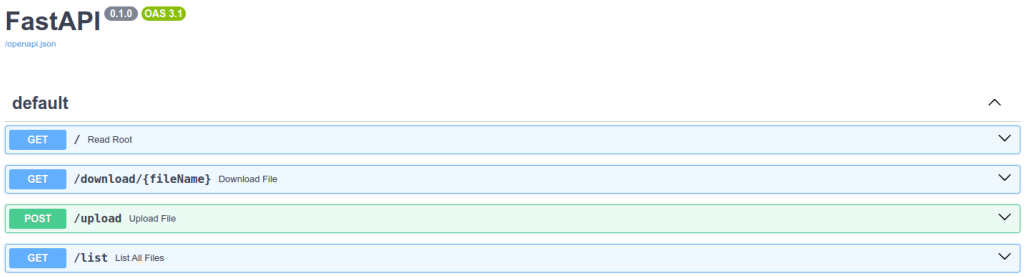

Figure 5: swagger ui endpoint visualization

- **Download File:** The client initiates file download from the server by providing the desired file name. The downloaded file is saved to the local file system.
- **List Files:** The client retrieves and displays the list of available files on the server.
- **Upload File:** The client uploads a file to the server by specifying the file path. The file is transmitted to the server and stored in the local file system.

The client code provides a user-friendly, interactive interface for file operations with the FastAPI server. Users can choose to download, list, or upload files based on their preferences. Further enhancements could include robust error handling, improved user experience, and additional features.

4 Conclusion

Figure 6: Client Output

Figure 7: Root Output

Figure 8: Download endpoint

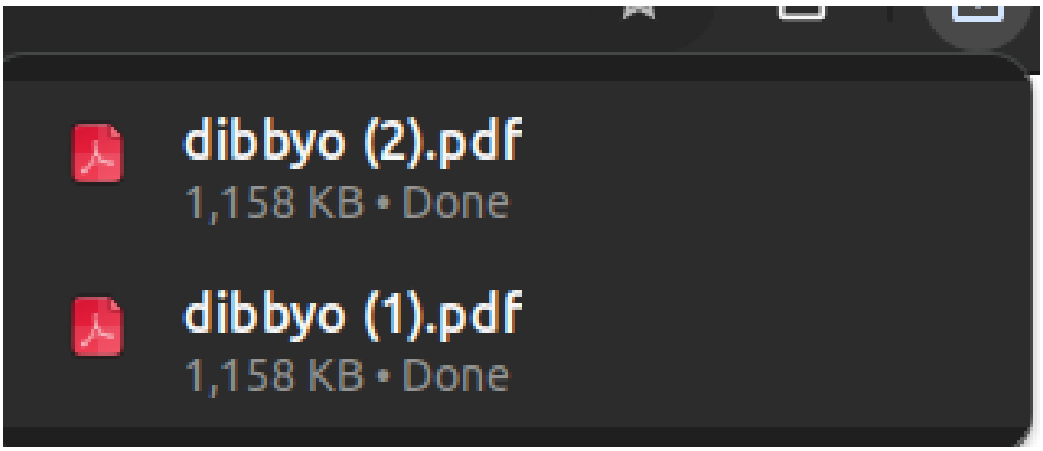

Figure 9: Downloaded from the endpoint

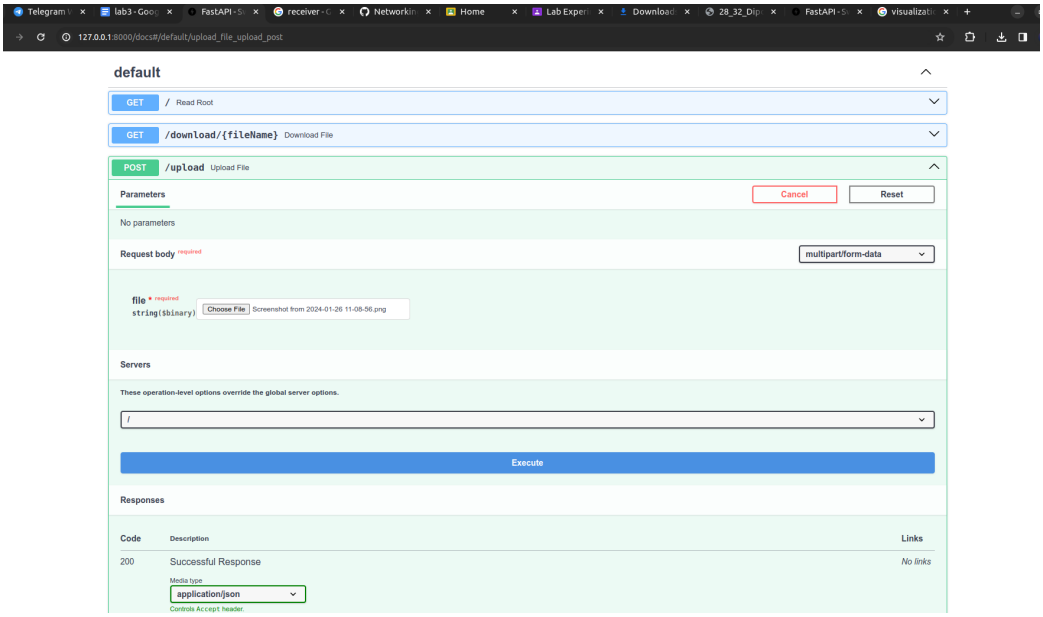

Figure 10: Uploading file to the server

Figure 11: Uploaded file to the server

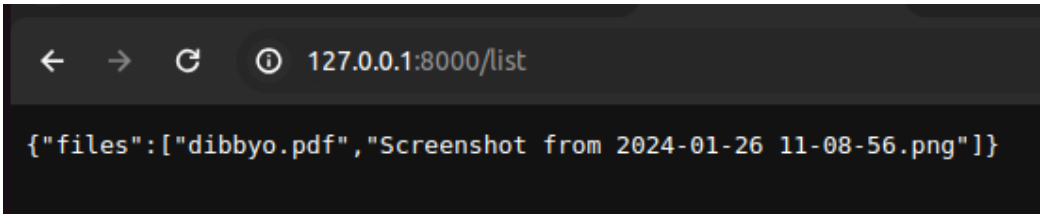

Figure 12: List of files in the server