Practical Work 1: TCP File Transfer Report

Student Name: Nguyen Viet Hung
Student ID: 23BI14188

November 27, 2025

1 Goal

The goal of this practical work is to implement a 1-1 file transfer system over **TCP/IP** using **sockets** in a Command Line Interface (CLI), based on the provided framework.

2 Protocol Design

2.1 How You Design Your Protocol

To enable the Server to correctly anticipate and receive the file, the communication protocol is designed in **two phases**:

1. **Phase 1: Send File Information Header** The Client sends a **single** data packet containing the file metadata structured as a string:

   Filename|Filesize_in_bytes

2. **Phase 2: Send File Data** The Client sequentially reads the file in chunks of `BUFFER_SIZE` bytes and sends them over the socket until the entire file is transferred.

2.2 Data Header Structure

The Client creates a header string (e.g., `test_file.txt|12345`), where `12345` is the exact file size in bytes. This allows the Server to determine exactly how many bytes to receive before closing the file.

3 System Organization

3.1 How You Organize Your System

The system is organized into a classic **Client-Server** model using TCP/IP sockets.

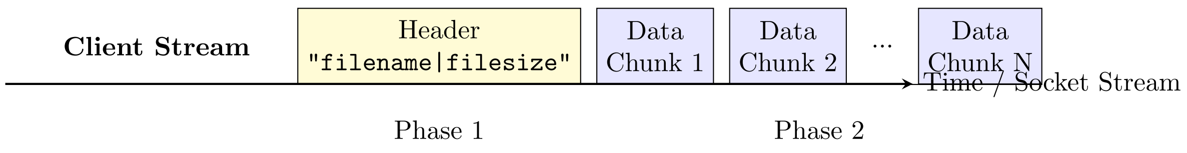

Figure 1: File transfer protocol structure (Header followed by Data)

1. **Server (`server.py`):** A passive component that listens for incoming connections on a predefined address and port (`127.0.0.1:65432`).
2. **Client (`client.py`):** An active component that initiates the connection to the Server and drives the file transfer process.

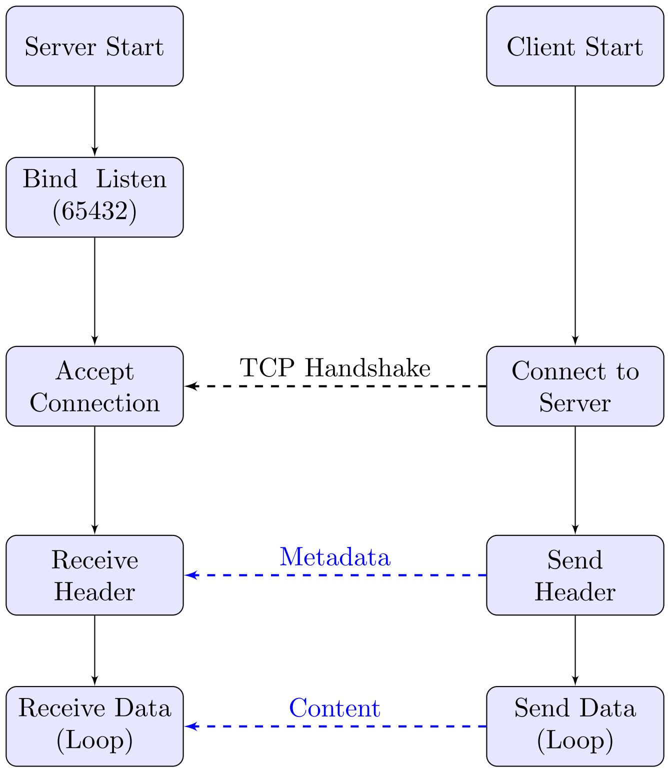

Figure 2: File transfer interaction flow

4 File Transfer Implementation

4.1 How You Implement the File Transfer: Client (`client.py`)

The Client handles connection, header preparation, and data sending.

```python
# Get file info
filesize = os.path.getsize(FILE_TO_SEND)
filename = os.path.basename(FILE_TO_SEND)

# File info
header = f"{filename}|{filesize}"
s.send(header.encode())

print(f"[*] File: {filename}, Size: {filesize} bytes")

# Send file data
bytes_sent = 0
with open(FILE_TO_SEND, "rb") as f:
    while True:
        bytes_read = f.read(BUFFER_SIZE)
        if not bytes_read:
            break

        s.sendall(bytes_read)
        bytes_sent += len(bytes_read)
        print(f"\rSent {bytes_sent}/{filesize} bytes", end="")
```

Listing 1: Code Snippet from Client: Sending Header and File Data

4.2 How You Implement the File Transfer: Server (`server.py`)

The Server accepts the connection, receives the header, parses the file information, and writes the incoming data to a local file.

```python
# File info
received = client_socket.recv(BUFFER_SIZE).decode()
filename, filesize_str = received.split('|')
filesize = int(filesize_str)
filename = os.path.basename(filename)

print(f"[*] File: {filename}, Size: {filesize} bytes")

# Receive file data
bytes_received = 0
with open("received_" + filename, "wb") as f:
    while bytes_received < filesize:
        remaining_bytes = filesize - bytes_received

        bytes_to_read = min(BUFFER_SIZE, remaining_bytes)
        bytes_read = client_socket.recv(bytes_to_read)

        if not bytes_read:
            break

        f.write(bytes_read)
        bytes_received += len(bytes_read)
        print(f"\rReceived {bytes_received}/{filesize} bytes", end=
    "")
```

Listing 2: Code Snippet from Server: Receiving Header and File Data

5 Who Does What

The responsibilities are clearly divided between the Client and the Server:

Component	**Key Responsibilities**
Server (`server.py`)	• Creates the socket, binds, and listens on port `65432`. • Accepts the connection from the Client. • **Receives and parses the Header** (`filename|filesize`). • **Receives file data** in chunks until `filesize` is reached. • Writes the received data to a new file (`received_...`). • Closes the connections.
Client (`client.py`)	• Creates the socket and connects to the Server. • Retrieves source file information (`test_file.txt`). • **Creates and sends the Header** (`filename|filesize`). • **Reads and sends file data** in chunks (`BUFFER_SIZE`). • Closes the connection.

Table 1: Task Allocation