Practical 1 Report - File Transfer over TCP/IP in CLI

Phung Dam Tien Si - 23BI14384

20 November, 2025

1 Protocol Design

The core challenge of file transfer over a stream socket (TCP) is that the receiver must know when to stop reading. Since TCP delivers a continous stream of bytes without message boundaries, a **pre-defined application layer protocol** is required to send metadata (filename, size) before the actual file data.

The protocol is designed in three sequential phases:

Phases 1: Filename meatadata

The server needs to know the name and length of the file it is receiving to save it locally.

Table 1: Phase 1: Filename Metadata Structure

Field	Size (Bytes)	Data Type	Purpose
Filename Length	4	Unsigned Integer	Tells the receiver how many bytes to read in Phase 2.
Filename (Variable)	*Varies*	UTF-8 Bytes	The actual name of the file (e.g., `document.pdf`).

Phase 2: File Size Header

After receiving the filename, the server needs to know the total number of bytes it should expect for the file content.

Table 2: Phase 2: File Size Header Structure

Field	Size (Bytes)	Data Type	Purpose
File Size	8	Unsigned Long Long	The total size of the file content to be transferred.

Phase 3: File Data

The client sends the raw binary content of the file. The server reads data in chunks until the number of received bytes equals the `File Size` received in Phase 2.

Table 3: Phase 3: File Data Structure

Field	Size (Bytes)	Data Type	Purpose
File Content	*Varies*	Raw Binary Bytes	The file's contents, sent in chunks (4096 bytes).

2 System Organization

The system is organized into a classic blocking TCP client-server architecture, where the server is designed to handle new connections continuously in a synchronous loop.

Server Organization (`server.py`)

1. `start_server()` **(Initialization and Listener)**: Creates, **Binds**, and **Listens** on the socket. It then enters an infinite `while True` loop, calling **Accept** to wait for new client connections.

2. `handle_client(conn, addr)` **(Session Logic)**: Processes the incoming connection (`conn`). It executes the three-phase protocol, writes the data to a local file, handles potential errors (e.g., incomplete transfer), and finally **Closes** the client connection.

3. `receive_all(sock, n)` **(Reliability Helper)**: A critical utility function that loops until the exact number of expected bytes (`n`) has been read from the stream, ensuring header integrity.

Client Organization (`client.py`)

1. `send_file(filepath)` **(Execution Logic)**: Validates the file and calculates its metadata. It then creates the socket and **Connects** to the server.

2. **Transfer Execution**: Sends the three protocol phases sequentially using `sendall()` to ensure reliable delivery of the metadata and the file content chunks.

3. **Termination**: **Closes** the client socket upon completion or error.

3 File Transfer Implementation

The reliability of the system hinges on the use of `receive_all` for header reception and a loop that tracks the total expected bytes for the file content.

The most critical part of the implementation is the Server's receiving loop, which handles reading the bulk data and tracking progress against the expected file size (`file_size`).

Code Snippet: Server Receiving Loop

This code is executed within the `handle_client` function on the server side:

```
bytes_received = 0

with open(new_filename, 'wb') as f:
    while bytes_received < file_size:
        # Calculate how many bytes are left to receive
        remaining_bytes = file_size - bytes_received

        # Determine the maximum size for the current chunk
        # (min of CHUNK_SIZE or remaining bytes)
        bytes_to_read = min(CHUNK_SIZE, remaining_bytes)

        # Receive the chunk
        chunk = conn.recv(bytes_to_read)

        if not chunk:
            # Connection closed before receiving the full file
            print(f"Connection␣closed␣prematurely.␣Received␣{bytes_received}␣of␣{file_size}␣
                bytes.")
            break

        f.write(chunk)
        bytes_received += len(chunk)

        # Simple progress indicator for CLI
        sys.stdout.write(f"\rProgress:␣{bytes_received␣/␣file_size␣*␣100:.2f}%␣|␣{
            bytes_received␣/␣(1024*1024):.2f}␣MB")
        sys.stdout.flush()

print("\nFile␣transfer␣complete.")

if bytes_received < file_size:
    os.remove(new_filename)
    print("Incomplete␣file␣deleted.")
```

Listing 1: Server-side File Receiving Loop (Python)