University Of Science and Technology of Hanoi
ICT Department

Distributed System Report

Lecturer: Le Nhu Chu Hiep

Practical Work 1: TCP File transfer

Student: Nguyễn Thị Vàng Anh - 23BI14032

1. Overview

This project implements a 1-1 file transfer system over TCP/IP in a command-line interface (CLI) using Python sockets. The architecture consists of:

- One server: listens on a fixed IP/port, manages files under server_files/, and handles requests.
- One client: connects to the server, provides a text menu, and allows the user to upload/download files or start a chat session.

The system extends a simple chat-style protocol with two additional operations: UPLOAD and DOWNLOAD.

1. Protocol Design

The application-layer protocol is a line-based, text protocol on top of TCP:

- All control messages (commands, file names, sizes, status) are sent as UTF-8 text terminated by \n.
- Binary file data is sent as raw bytes immediately after the corresponding header lines.

2.1 Commands

For each connection, the client sends commands; the server parses them and reacts:

- UPLOAD
- DOWNLOAD
- CHAT
- EXIT

2.1.1 UPLOAD Flow

1. Client sends:
    a. UPLOAD
    b. <filename>
    c. <filesize_in_bytes>
2. Client sends exactly filesize bytes of file content.
3. Server:
    a. Reads filename and size.
    b. Receives size bytes and writes to server_files/<filename>.
    c. Replies with OK\n on success, or ERR\n if incomplete.

2.1.2 DOWNLOAD Flow

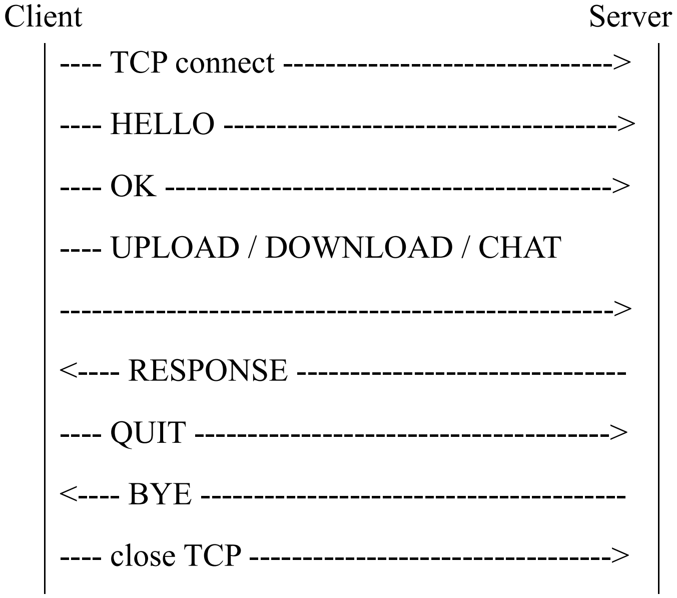
1. Client sends:
    a. DOWNLOAD
    b. <filename>
2. Server:
    a. If file does not exist → ERR
    b. Else:

            i. Gets file size and sends: <filesize_in_bytes>
            ii. Sends exactly that many bytes.
3. Client:
    a. Receives size.
    b. Reads size bytes and saves to client_files/<filename>.

2.1.3 CHAT Flow

1. Client sends: CHAT.
2. Both sides enter a loop exchanging text lines.
3. If either side sends bye, the chat ends.

Figure 1: Simple text-based protocol


Client
Server
---- TCP connect ------------------------------->
---- HELLO ------------------------------------->
---- OK -->
---- UPLOAD / DOWNLOAD / CHAT
-->
<---- RESPONSE -------------------------------
---- QUIT --------------------------------------->
<---- BYE ---------------------------------------
---- close TCP --------------------------------->

2. System Organization

The system is organized into modular functions on both sides.

3.1 Server Structure

- Global settings: HOST, PORT, BUF, SERVER_DIR.
- Utility:
    - recv_line(conn): read until
    - send_line(conn, text): send a text line
- Handlers:
    - handle_upload(conn)
    - handle_download(conn)
    - handle_chat(conn)

    - handle_client(conn, addr): main loop per connection, dispatching by command.
- Entry point:
    - main(): creates listening socket, accepts one client at a time, calls handle_client.

Figure 2: System Organization

```
+-------------------+                   +-------------------+
|     Client CLI    |                   |   Server Process  |
|-------------------|                   |-------------------|
| main()            |                   | main()            |
|  - connect()      |---- TCP socket -->|  - listen()       |
|  - menu loop      |                   |  - accept()       |
|                   |                   |                   |
| upload_file()     |                   | handle_upload()   |
| download_file()   |                   | handle_download() |
| chat()            |                   | handle_chat()     |
+-------------------+                   +-------------------+
             ^                                     ^
             |                                     |
      client_files/                          server_files/
```

3. File Transfer Implementation

4.1 Common Utilities

Server

```
def recv_line(conn):
data = b""
while not data.endswith(b"\n") and (chunk := conn.recv(1)):
data += chunk
return data[:-1].decode() if data else ""


def send_line(conn, text):
conn.sendall((text + "\n").encode("utf-8"))
```

Client

```
def recv_line(sock):
data = b""
while not data.endswith(b"\n"):
chunk = sock.recv(1)
if not chunk:
break
data += chunk
return data.rstrip(b"\n").decode("utf-8")
```

```
def send_line(sock, text):
sock.sendall((text + "\n").encode("utf-8"))
```

4.2 Upload Implementation

Server

```
def handle_upload(conn):
filename = recv_line(conn)
size = int(recv_line(conn) or 0)

os.makedirs(SERVER_DIR, exist_ok=True)
path = os.path.join(SERVER_DIR, filename)

remaining = size
with open(path, "wb") as f:
while remaining > 0:
chunk = conn.recv(min(BUF, remaining))
if not chunk:
break
f.write(chunk)
remaining -= len(chunk)

send_line(conn, "OK" if remaining == 0 else "ERR")
```

Client

```
def upload_file(sock):
path = input("File to upload: ").strip()
if not os.path.exists(path):
print("File does not exist.")
return

filename = os.path.basename(path)
size = os.path.getsize(path)

send_line(sock, "UPLOAD")
send_line(sock, filename)
send_line(sock, str(size))

with open(path, "rb") as f:
while True:
chunk = f.read(BUF)
if not chunk:
break
sock.sendall(chunk)

resp = recv_line(sock)
print("Upload:", "success" if resp == "OK" else "failed")
```

4.3 Download Implementation

Server

```
def handle_download(conn):
filename = recv_line(conn)
path = os.path.join(SERVER_DIR, filename)

if not os.path.exists(path):
send_line(conn, "ERR")
return

size = os.path.getsize(path)
send_line(conn, str(size))

with open(path, "rb") as f:
while True:
chunk = f.read(BUF)
if not chunk:
break
conn.sendall(chunk)
```

Client

```
def download_file(sock):
filename = input("File to download: ").strip()
send_line(sock, "DOWNLOAD")
send_line(sock, filename)

size_line = recv_line(sock)
if not size_line or size_line == "ERR":
print("File not found on server.")
return

size = int(size_line)
os.makedirs(CLIENT_DIR, exist_ok=True)
path = os.path.join(CLIENT_DIR, filename)

remaining = size
with open(path, "wb") as f:
while remaining > 0:
chunk = sock.recv(min(BUF, remaining))
if not chunk:
break
f.write(chunk)
remaining -= len(chunk)

if remaining == 0:
print("Downloaded to:", path)
else:
print("Download incomplete.")
```

4. Conclusion

The system demonstrates a simple but complete end-to-end file transfer over TCP/IP using Python sockets and a custom text-based protocol. It supports upload, download, and interactive chat mode, with clear separation of concerns between networking (socket handling), protocol (line-based commands), and application logic (file operations and directories).