Practical Work 1 – TCP File Transfer
Students: Vu Xuan Thai -23BI14397

1. Introduction

The objective of this practical work is to implement a 1-to-1 file transfer system over TCP/IP using a command-line interface. The system consists of two programs:

- A TCP server that waits for incoming connections and receives a file.
- A TCP client that connects to the server and sends a file.

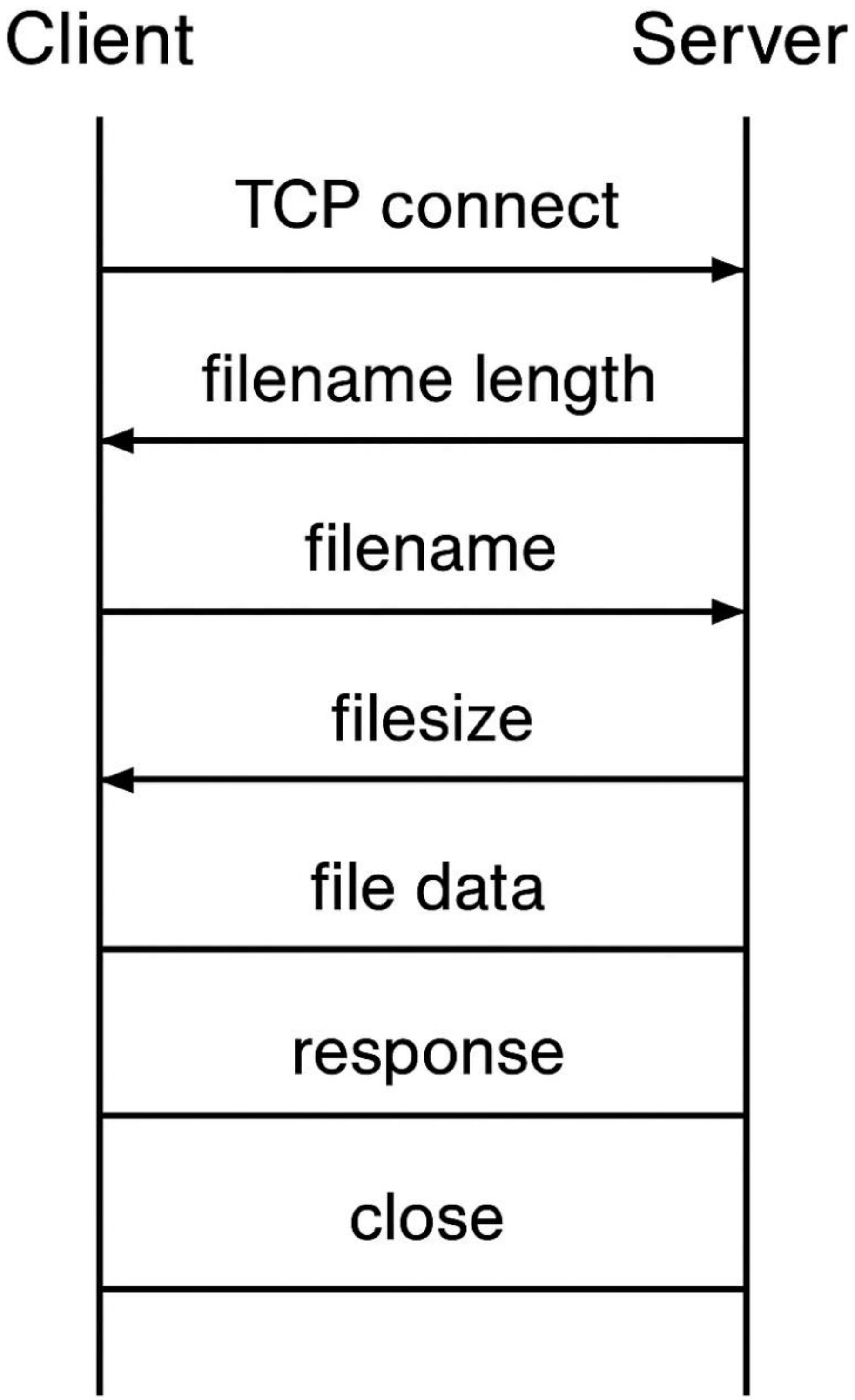
The project demonstrates how to use sockets, understand TCP connection setup, and implement a custom communication protocol for transferring files correctly and reliably.

2. Protocol Design

To ensure reliable communication between the client and server, we implemented a custom binary protocol based on the following structure:

1. Header
   - 4 bytes: File name length (uint32, network byte order)
   - N bytes: File name (UTF-8)
   - 8 bytes: File size (uint64, network byte order)
2. Body
   - File data bytes: Exactly equal to file size
3. Server response
   - "OK" – file received successfully
   - "INCOMPLETE" – bytes missing
   - "ERROR" – internal write error

This small protocol ensures that both sides know exactly how many bytes to read, preventing partial reads or misalignment issues.


Client
Server
TCP connect
filename length
filename
filesize
file data
response
close

3. System Architecture

The system contains two main programs:

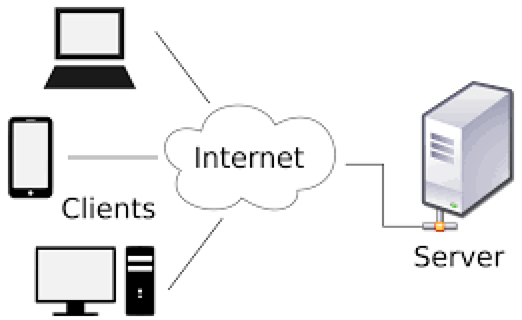
3.1 Server (server.c)

- Opens a TCP socket
- Binds to the configured IP address and port
- Waits for a client using listen() and accept()
- Receives the header (file name + file size)
- Reads exactly filesize bytes into an output file
- Sends a status message back to the client

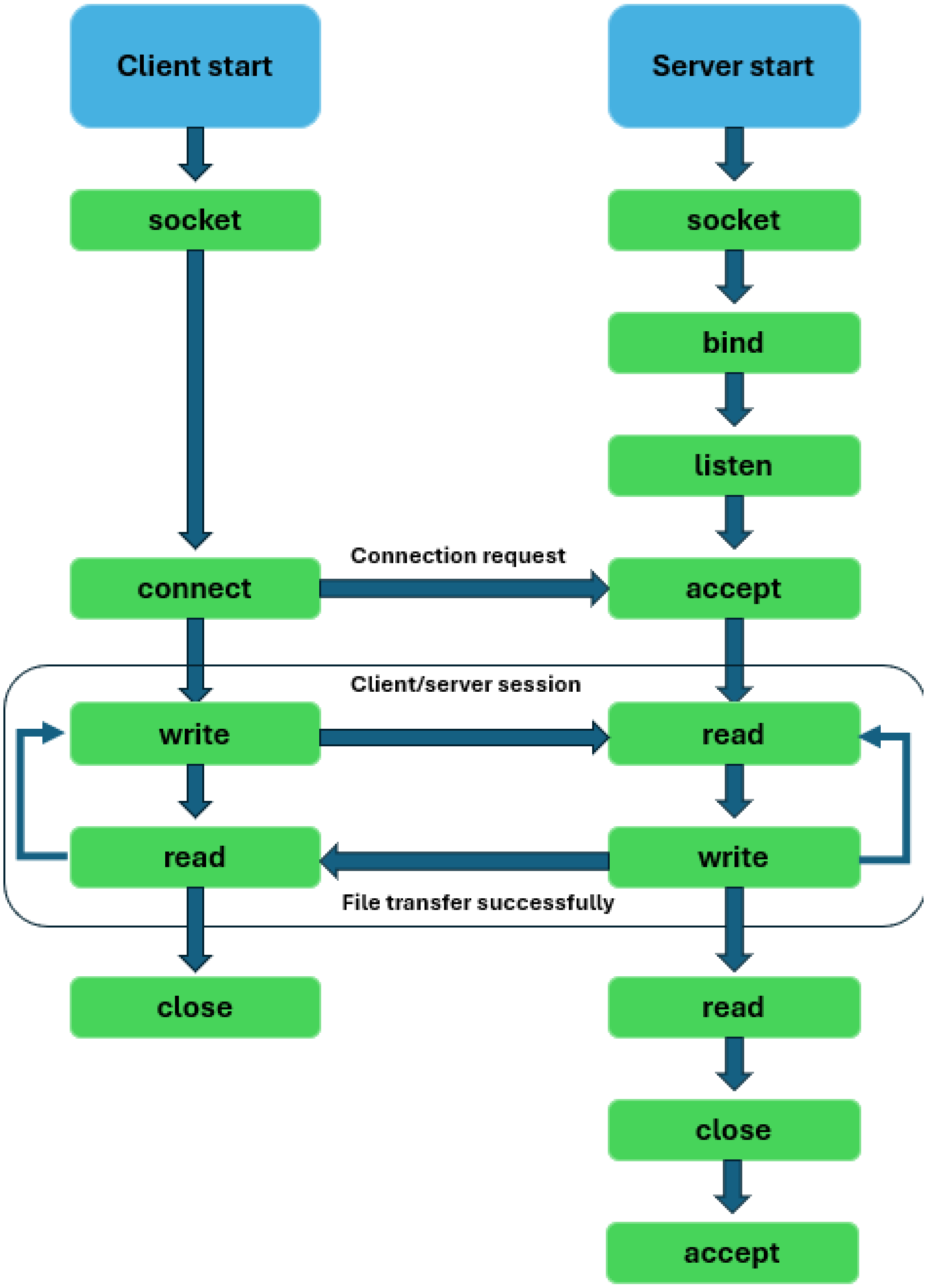
3.2 Client (client.c)

- Creates a socket and connects to the server
- Reads the file from disk and determines file size
- Sends the header (file name length, file name, file size)
- Streams the file bytes to the server
- Waits for a response message

4. Sequence Diagram



5. Implementation Summary



5.1 Key Features

- TCP stream-based data transfer
- Custom header for metadata
- Compatible across Linux/Unix systems
- Uses recv() loops to ensure all bytes are received
- Simple error handling with status messages

5.2 Important Code Concepts

- socket(), bind(), listen(), accept()
- connect() for the client
- Network byte order conversion using:
  - htonl(), ntohl()
  - Custom htonll(), ntohll() for 64-bit integers
- File reading and writing using read()/write()

6. How to Compile and Run

Compile

```
gcc server.c -o server
```

```
gcc client.c -o client
```

Run the server

```
./server --host 0.0.0.0 --port 9000 --outdir received_files
```

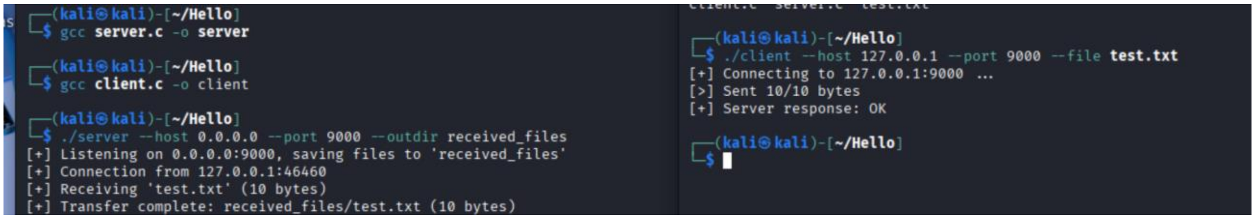
Run the client

```
./client --host <SERVER_IP> --port 9000 --file path/to/file
```

7. Testing

Test 1: Small text file

- Sent a 1 KB .txt file
- Server response: OK
- Output file matches exactly



```
┌──(kali㉿kali)-[~/Hello]
└─$ gcc server.c -o server

┌──(kali㉿kali)-[~/Hello]
└─$ gcc client.c -o client

┌──(kali㉿kali)-[~/Hello]
└─$ ./server --host 0.0.0.0 --port 9000 --outdir received_files
[+] Listening on 0.0.0.0:9000, saving files to 'received_files'
[+] Connection from 127.0.0.1:46460
[+] Receiving 'test.txt' (10 bytes)
[+] Transfer complete: received_files/test.txt (10 bytes)
```

```
client.c  server.c  test.txt

┌──(kali㉿kali)-[~/Hello]
└─$ ./client --host 127.0.0.1 --port 9000 --file test.txt
[+] Connecting to 127.0.0.1:9000 ...
[>] Sent 10/10 bytes
[+] Server response: OK

┌──(kali㉿kali)-[~/Hello]
└─$ █
```

8. Conclusion

In this practical work, we successfully implemented a TCP-based file transfer system using C sockets. The project provided hands-on experience with:

- TCP communication
- Custom binary protocols

- File handling
- Byte-order conversions
- Network programming fundamentals

The client and server communicate reliably, transfer files correctly, and handle various error conditions. This project forms a strong foundation for more advanced networking applications.