PRACTICAL 1 - TCP File Transfer System (Python)

1. Introduction

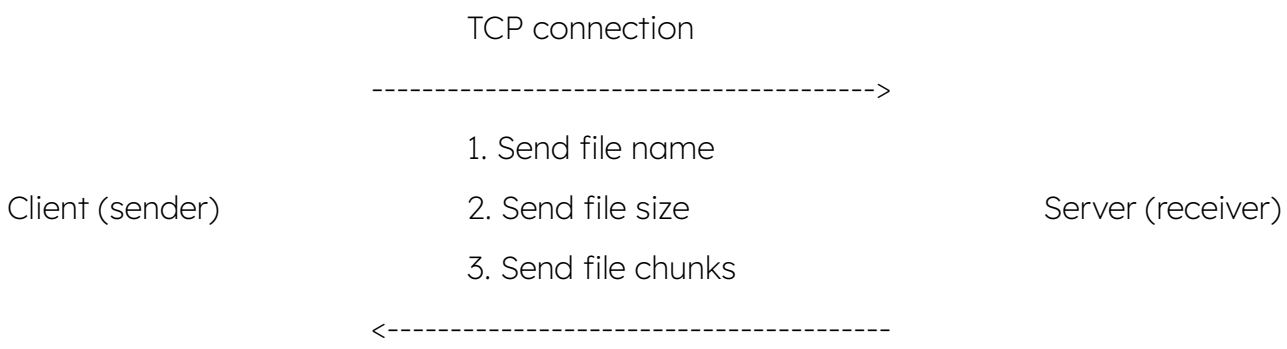
The purpose of this practical work is to implement a simple file transfer system based on a Client–Server model using the Transmission Control Protocol (TCP).
The objective is to allow a client to send a file to a server through a reliable communication channel, following a custom application-level protocol.
This assignment demonstrates socket programming fundamentals, byte-level communication, message framing, and basic network architecture concepts.

2. System Architecture

The system includes two components:

- **FileClient (Sender):** Reads a file from disk, connects to the server, sends metadata and file contents.
- **FileServer (Receiver):** Listens on a TCP port, accepts a connection, receives the file, and saves it locally.

The architecture is simple, linear, and synchronous—only one client is handled at a time.



The server runs first and waits for incoming connections.
The client initiates the connection, sends data, and terminates after completing the transfer.

3. Communication Protocol

Protocol Steps

Step 1: **Filename length (4 bytes)**

The client sends a 32-bit unsigned integer (big-endian) describing how many bytes the filename contains.

Step 2: **Filename (UTF-8)**

The client sends the actual filename.

Step 3: **Filesize (8 bytes)**

A 64-bit unsigned integer indicating the total size of the file in bytes.

Step 4: **File content (stream of chunks)**

The file is sent in chunks (typically 4096 bytes each) until the entire file is transmitted.

Protocol Data Flow

Client		**Server**
connect()	---------->	accept()
send(filename_len)	---------->	recv(4 bytes)
send(filename)	---------->	recv(N bytes)
send(filesize)	---------->	recv(8 bytes)
send(file data)	---------->	write to disk
close()	---------->	close()

4. Implementation Summary

The system is implemented using Python sockets with two components: a client and a server. The server listens on a fixed TCP port, accepts one connection, and receives the filename, file size, and file data according to a simple custom protocol. It writes the incoming bytes to a new file on disk.

The client connects to the server, reads a file from disk, and sends metadata (filename length, filename, file size) followed by the file content in 4096-byte chunks. Using fixed-size chunks allows the program to transfer files of any size efficiently without loading them entirely into memory.

Both programs use only Python's standard library.

5. Testing & Results

Test setup

On client:

- create file text.txt with message "hello from python":

```
echo "hello from python" > test.txt
```

- send file test.txt:

```
python3 client.py 127.0.0.1 9000 test.txt
```

On server:

- run:

```
python3 server.py 9000 --out received_files
```

Results

On client:

```
[+] Connecting to 127.0.0.1:9000
[+] Connected.
    Sent 18/18 bytes
[+] File sent successfully.
```

On server:

```
[+] Listening on 0.0.0.0:9000
[+] Connection from ('127.0.0.1', 46978)
[+] Receiving file: test.txt -> saving as
received_files/received_test.txt
[+] Expected size: 18 bytes
    Received 18/18 bytes
[+] File transfer complete.
[+] Connection closed.
```

6. Conclusion

This practical work demonstrates a complete implementation of a TCP-based file transfer system using Python.

The system successfully handles structured communication, including metadata and streaming file content.

Key learning outcomes include:

- Using low-level TCP sockets
- Designing a custom message protocol
- Handling binary data and byte framing
- Implementing chunk-based streaming
- Understanding fundamental client–server communication patterns

I. System Architecture (Design Overview)

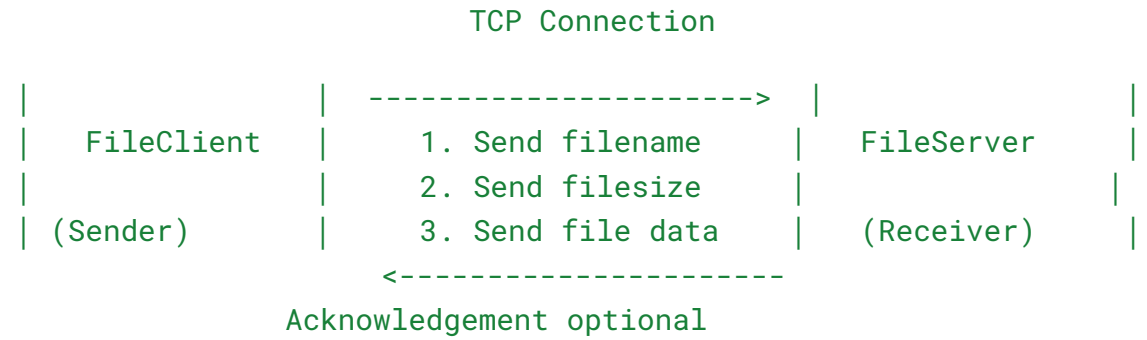
In this practical work, we implemented a simple **client–server architecture** for transferring files over TCP. The system consists of two independent components:

1. **FileServer** – Listens on a TCP port, accepts an incoming connection, receives metadata and file content, and stores the file locally.

2. **FileClient** – Connects to the server, reads a file from disk, sends its metadata, and transmits the file in byte chunks.

Both components communicate using a **point-to-point TCP connection**, ensuring reliable, ordered delivery of bytes. The server blocks on `accept()` until a client connects, and the client initiates the session using a standard TCP handshake.

Architecture diagram:



The architecture is intentionally simple and sequential (one server → one client) to match the objectives of the lab.

II. Communication Protocol

Since TCP is a **stream-based** protocol and does not preserve message boundaries, we designed a minimal custom application-level protocol to describe how data is structured during transmission.

The communication protocol consists of three phases:

1. Metadata Transmission

Before sending the file content, the client transmits:

- **Filename** as a UTF-8 string (using `writeUTF()` in Java)
- **File size** as a 64-bit long integer (using `writeLong()`)

This allows the server to:

- Create a file with the correct name
- Allocate buffers
- Know exactly how many bytes to expect

2. File Data Transmission

After metadata, the client reads the file from disk and sends it in fixed-size byte blocks (e.g., 4096 bytes):

```
while (bytes remain):
    read chunk from file
    send chunk to server
```

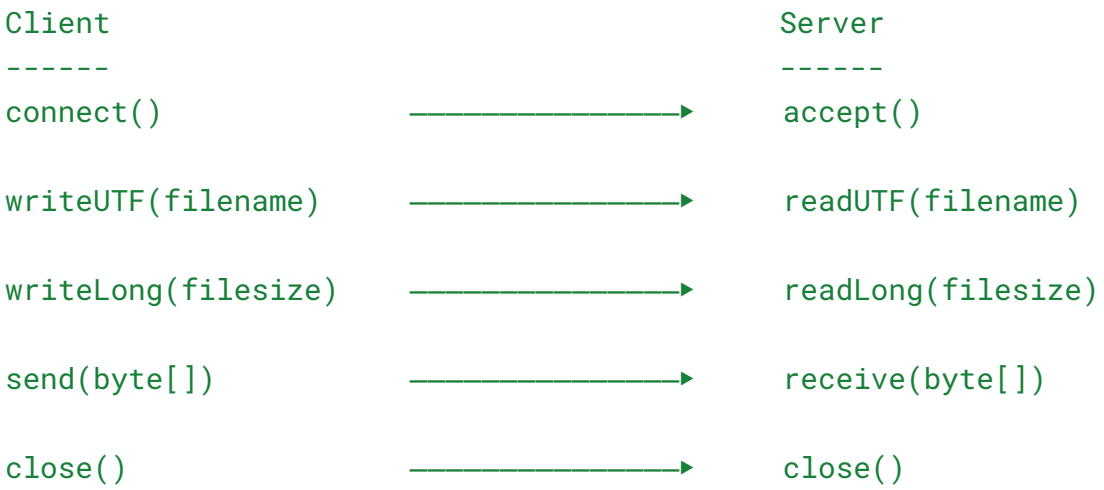
The server reads from the input stream until it receives exactly the number of bytes specified in the `filesize` field.

3. Completion and Clean-up

Once all bytes are received:

- The server writes the data to `received_<filename>`
- Both sides close their streams and the socket
- The server returns to listening state (if extended)

III. Data Flow Diagram



IV. Design Rationale

- **TCP** is chosen because it guarantees ordered, lossless transmission, which is essential for binary file transfer.
- **Explicit metadata** (filename + size) enables predictable and structured parsing on the server side.
- **Chunk-based transmission** avoids memory overflow and supports large files.
- **Simplicity**: The design is intentionally linear (one client, one server) to satisfy project constraints and keep implementation readable.