

# 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.

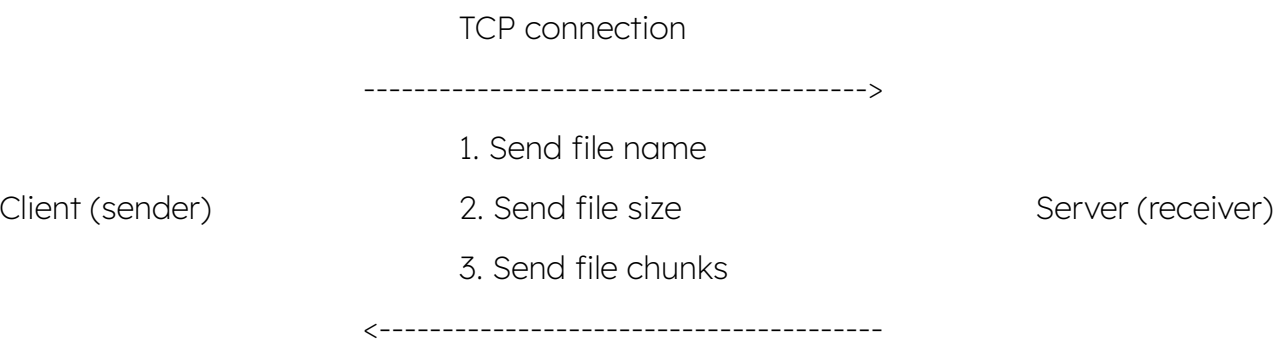
This assignment demonstrates socket programming fundamentals, byte-level communication, message framing, and basic network architecture concepts.

## 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 test.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