Practical Work 3 – MPI File Transfer

1. Introduction

The goal of this practical work is to upgrade an existing TCP-based file transfer system into a new version that uses **MPI (Message Passing Interface)**.
In the previous assignment, file transfer was implemented using sockets, with a traditional client–server model.
In this assignment, the system must be adapted to run on MPI, where multiple processes communicate using message-passing rather than network sockets.

The implementation was done using **OpenMPI** and **mpi4py** in a Linux environment.

2. Design of the MPI File Transfer Service

Architecture Overview

Unlike TCP (which uses IP + port + sockets), MPI has no central "server".
Every program instance is a rank, and `mpirun` launches all processes.
In our design:

- **Rank 0** acts as the **receiver**
- **Rank 1** acts as the **sender**

This design fits the original TCP architecture while respecting the MPI execution model.

```
mpirun -np 2 python3 mpi_transfer.py file.txt
```

Message Flow

The sender (rank 1) transmits the following information to the receiver (rank 0):

1. Filename length
2. Filename (UTF-8 encoded)
3. File size (in bytes)
4. File content split into fixed-size chunks
5. An empty chunk to signal end-of-file

The receiver reconstructs the file by writing the received chunks to disk.

3. Testing & Results

Test setup

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

```
echo "Test MPI Transfer" > test.txt
```

- run:

```
mpirun -np 2 python3 mpi_transfer.py test.txt
```

Results

```
[Rank 1] Sending file 'test.txt' (18 bytes) to rank 0.
[Rank 1] Sent 18/18 bytes
[Rank 1] File sent successfully.
[Rank 0] Receiving file 'test.txt' (18 bytes)
[Rank 0] Saving as: ./received_test.txt
[Rank 0] Received 18/18 bytes
[Rank 0] File transfer complete.
```

4. Conclusion

The migration from TCP sockets to MPI demonstrates key differences between distributed communication models:

- TCP uses persistent connections and explicit networking.
- MPI uses message passing between ranks without network programming.

The new MPI-based file transfer system successfully performs:

- Filename and metadata exchange
- Binary file transmission using fixed-size chunks
- End-of-file signaling
- Reassembly and storage on the receiver side

The assignment provided hands-on experience with multiprocessing, MPI primitives, and adapting real systems to new communication paradigms.

I. System Architecture (Design Overview)

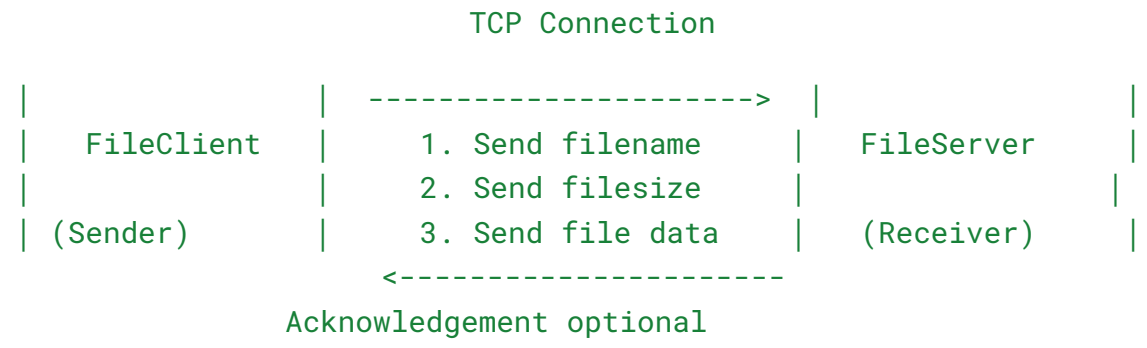
In this practical work, we implemented a simple **client–server architecture** for transferring files over TCP. The system consists of two independent components:

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.

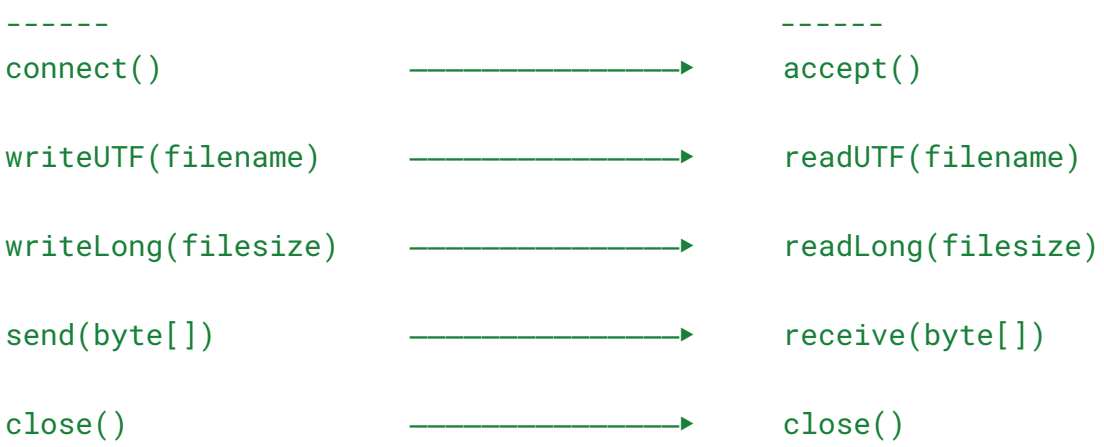
3. Completion and Clean-up

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

`Client` `Server`



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.