Practical Work 2: MPI File Transfer

USTH – Distributed Systems
Do Minh Tien – 23BI14421

December 10, 2025

1 Introduction

In Practical Work 1, we implemented a basic TCP file-transfer service using sockets. In this Practical Work (Practical Work 3), we upgrade the system to use MPI (Message Passing Interface) instead of TCP.

MPI is a standardized communication library widely used in distributed and parallel computing. The goal of this practical work is to replace TCP send/receive operations with MPI message exchanges while keeping the same file-transfer logic: the client requests a filename, and the server returns the file size and file contents.

2 Choice of MPI Implementation

We chose **MPICH** for this practical work due to several advantages:

- Lightweight, easy to install, and available on USTH lab machines.
- Fully supports the MPI-3 standard.
- Widely used for educational and research purposes.
- Provides simple tools: `mpicc`, `mpirun`, `mpiexec`.

3 MPI Communication Design

MPI assigns each running process a numeric ID called a **rank**. For this assignment, we use two processes:

- **Rank 0: Client** — sends the filename and receives the file.
- **Rank 1: Server** — reads the file and sends it in chunks.

We define three message tags to separate communication phases:

- **TAG_FILENAME** — the client sends the filename.
- **TAG_FILESIZE** — the server sends the file size.
- **TAG_DATA** — the server sends binary file chunks.

Design Overview

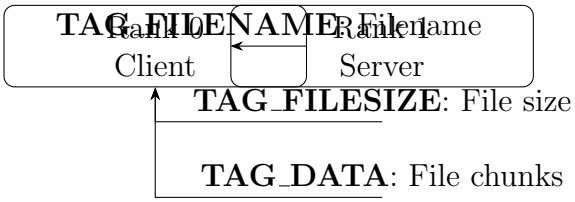

Figure 1: MPI File Transfer Communication Flow

4 System Organization

The file-transfer workflow is as follows:

1. The client (Rank 0) sends a filename.
2. The server (Rank 1) checks whether the file exists.
3. If the file does not exist, the server sends `-1`.
4. If the file exists, the server sends the file size in bytes.
5. The server sends the file in chunks of 4096 bytes.
6. The client writes received chunks to a local output file.

System Architecture

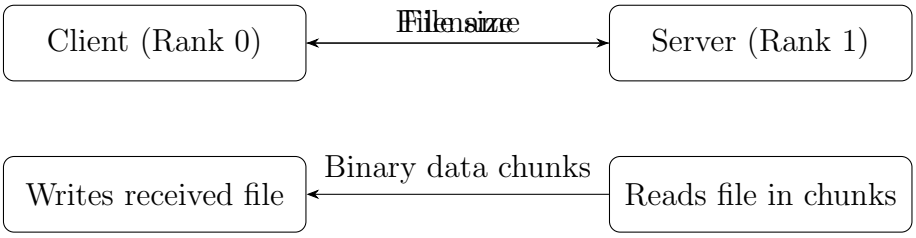

Figure 2: Overall MPI File Transfer Architecture

5 Implementation

Below is the core Python implementation using `mpi4py`. It demonstrates the required functionality with clear separation of client and server roles.

MPI Implementation Code

```
from mpi4py import MPI
import os

CHUNK = 4096
TAG_FILENAME = 0
TAG_FILESIZE = 1
TAG_DATA = 2

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# ============================
# Rank 1 = Server
# ============================
if rank == 1:
    print("[Server] Waiting for filename...")

    # Receive filename from client
    filename = comm.recv(source=0, tag=TAG_FILENAME)
    print(f"[Server] Client requested: {filename}")

    # File existence check
    if not os.path.exists(filename):
        print("[Server] File not found.")
```

```
        comm.send(-1, dest=0, tag=TAG_FILESIZE)

    else:
        size = os.path.getsize(filename)
        comm.send(size, dest=0, tag=TAG_FILESIZE)
        print(f"[Server] Sending {size} bytes...")

        with open(filename, "rb") as f:
            while True:
                data = f.read(CHUNK)
                if not data:
                    break
                comm.send(data, dest=0, tag=TAG_DATA)

        print("[Server] Transfer complete.")


# ============================
# Rank 0 = Client
# ============================
elif rank == 0:

    filename = input("Enter filename to download: ").strip()
    print(f"[Client] Requesting: {filename}")

    comm.send(filename, dest=1, tag=TAG_FILENAME)

    # Receive file size
    file_size = comm.recv(source=1, tag=TAG_FILESIZE)

    if file_size < 0:
        print("[Client] Server reports: File not found.")
    else:
        print(f"[Client] File size: {file_size}")

        output = "received_" + filename
        received = 0

        print("[Client] Receiving file...")

        with open(output, "wb") as f:
            while received < file_size:
                chunk = comm.recv(source=1, tag=TAG_DATA)
```

```
            f.write(chunk)
            received += len(chunk)

    print(f"[Client] File received successfully: {output}")
```

6 Who Did What

- **Rank 0 (Client)**: Sends the requested filename, receives the file size, reads incoming data chunks, reconstructs the file, and handles all error cases.
- **Rank 1 (Server)**: Receives the filename, validates file existence, sends the size or an error signal, and transmits the file in chunks.

7 Conclusion

This practical work successfully demonstrates how a conventional TCP-based file-transfer service can be migrated to the MPI environment. MPI's message-passing model simplifies communication and removes the need for socket management. The result is a clean, structured file-transfer workflow that works reliably across distributed processes.