Practical Work 3: MPI-Based RPC File Transfer

Dng c Anh / 23BI14012

December 24, 2025

Abstract

This report details the implementation of a file transfer system using the Message Passing Interface (MPI). By utilizing MPI's rank-based architecture and custom data structures, we simulate Remote Procedure Call (RPC) semantics to transfer files between two processes. The system features a structured metadata handshake, synchronous acknowledgments, and chunk-based data streaming within an MPI communicator.

1 RPC Service Design

The system implements a single RPC-style method, `UploadFile`, which facilitates the transfer of binary or text data from a client process to a server process.

UploadFile(filename, filesize, data)

To handle the communication of complex metadata, we defined a custom MPI structure that mirrors our C structure. This ensures that the metadata is correctly serialized and deserialized between processes.

```c
typedef struct {
    char method[32];
    char filename[256];
    long long filesize;
} Metadata;
```

Listing 1: MPI Metadata Structure

Protocol Steps

1. **Handshake:** Client (Rank 1) sends the `Metadata` struct to the Server (Rank 0) using a custom MPI datatype.
2. **Verification:** Server receives the metadata and validates the requested "UploadFile" method.
3. **ACK:** Server sends an integer acknowledgment (code 200) to the Client.

4. **Streaming:** Client reads the file in 4096-byte chunks and sends them via `MPI_Send`.
5. **Reconstruction:** Server receives chunks, using `MPI_Get_count` to handle the exact number of bytes in the final chunk, and writes them to disk.
6. **Final Status:** Server sends a completion status (201 for success) back to the client.

2 System Organization

The system is organized into a single `MPI_COMM_WORLD` communicator where roles are assigned based on process rank.

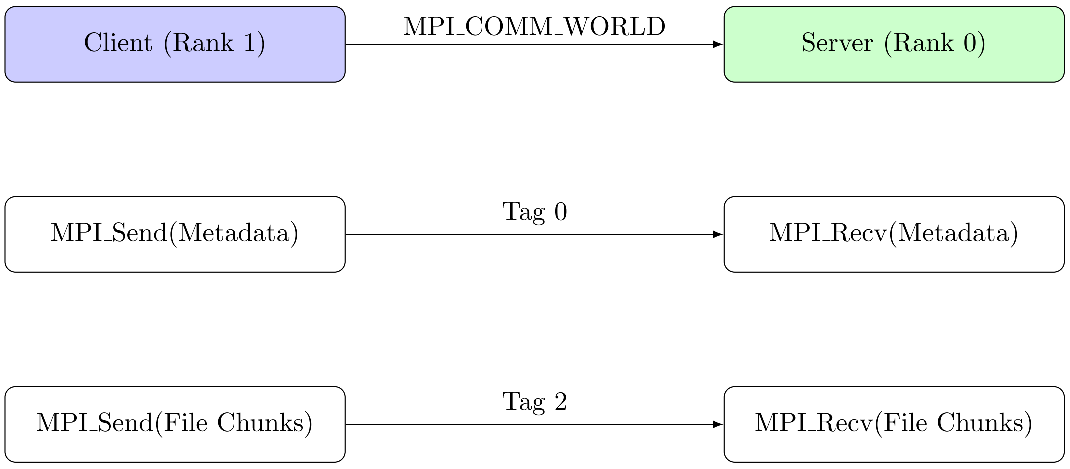

Figure 1: MPI-Based RPC File Transfer Architecture

3 Implementation

3.1 Server-Side (Rank 0)

The server utilizes `MPI_Recv` with `MPI_Status` to dynamically determine the size of the final incoming data chunk.

```
while (total_received < meta.filesize) {
    int count;
    MPI_Recv(buffer, CHUNK_SIZE, MPI_CHAR, CLIENT_RANK, 2,
             MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_CHAR, &count); // Handle
        last chunk size

    fwrite(buffer, 1, count, f);
    total_received += count;
}
```

Listing 2: Server Chunk Reception Loop

3.2 Client-Side (Rank 1)

The client is responsible for obtaining the file size via the `stat` system call and initiating the transfer.

```
FILE *f = fopen(filepath, "rb");
char buffer[CHUNK_SIZE];
size_t n;

while ((n = fread(buffer, 1, CHUNK_SIZE, f)) > 0) {
    MPI_Send(buffer, n, MPI_CHAR, SERVER_RANK, 2,
        MPI_COMM_WORLD);
}
```

Listing 3: Client Chunk Transmission Loop

4 Conclusion

By leveraging MPI's point-to-point communication and structured datatypes, we successfully simulated an RPC file transfer system. This implementation demonstrates that while MPI is traditionally used for high-performance computing, its robust message-passing capabilities are well-suited for distributed system tasks such as reliable file transfer and remote service simulation.