Group Project Report: Remote Shell using MPI

Group ID: 4

Student Name	Student ID
Nguyen Tuan Dung	22BI13107
Tran Duc Duong	22BI13117
Vu Hung Anh	22BI13045
Truong Dai An	22BI13008
Tran Trong Nghia	22BI13332
Nguyen Thanh Nam	22BI13325

Contents

1 Introduction **3**

2 Project Objectives **3**

3 System Design and Implementation **3**
3.1 Server Side (Server Process) 3
3.2 Client Side (Client Process) 4
3.3 Communication Protocol (Using MPI) 4

4 Technologies Used **4**

5 System Flow **5**

6 Example Usage **5**
6.1 Setup Instructions 5
6.2 Server Execution 5
6.3 Client Interaction 6

7 Diagrams and Documentation **6**
7.1 Use Case Diagram 6
7.2 Class Diagram 7
7.3 Sequence Diagram 8
7.4 Activity Diagram 9
7.5 Flowchart 10

8 Challenges and Solutions **11**
8.1 Challenges 11
8.2 Solutions 11

9 Conclusion **11**

1 Introduction

This project implements a remote shell system using the Message Passing Interface (MPI) standard. The system allows a client to execute commands on a server over a network, resembling tools like SSH. Unlike traditional remote shells, this implementation uses MPI for efficient communication between multiple clients and a single server. MPI provides parallel programming capabilities in distributed systems, making it suitable for handling multiple clients concurrently.

2 Project Objectives

The primary objectives of this project are:

- Develop a remote shell that supports multiple clients using MPI.
- Enable server-to-client command execution with response handling.
- Provide robust error handling and secure communication.
- Ensure the system is modular and easy to understand, with comprehensive documentation and diagrams to support further development.

3 System Design and Implementation

The system consists of two main components: the server and the clients, each with specific responsibilities.

3.1 Server Side (Server Process)

The server, running as the root process of MPI (rank 0), is responsible for receiving commands from the clients and executing them on the server's system. Key responsibilities of the server include:

- Receiving commands from clients using MPI communication.
- Executing commands on isolated terminals using the `pty` module to simulate terminal interactions.
- Sending the command results back to the respective clients.
- Handling client disconnections and performing cleanup tasks.

The server dynamically creates a unique directory for each session in the `/tmp` directory, ensuring isolated execution environments for each command. The server uses `uuid` to avoid name clashes between different server instances.

3.2 Client Side (Client Process)

Each client, identified by its rank (1 to N-1), is responsible for:

- Accepting shell commands from the user.
- Sending the commands to the server via MPI.
- Receiving the command output from the server.
- Displaying the output to the user.

Each client runs a Python script inside an `xterm` terminal window that communicates with the server. The client waits for command responses, handles exceptions, and displays the output.

3.3 Communication Protocol (Using MPI)

MPI is employed for communication between the server and the clients. The server listens for messages from clients using `comm.Iprobe`, and the clients send commands using `comm.send()` while receiving responses with `comm.recv()`.

- **Server to Client Communication:** The server executes a command and sends the output to the respective client.
- **Client to Server Communication:** Clients send commands to the server for execution.

The communication flow follows a simple pattern: 1. A client sends a command to the server. 2. The server executes the command, captures the output, and sends it back to the client. 3. The client displays the output and waits for further input.

4 Technologies Used

The following technologies were utilized in the implementation:

- **MPI (Message Passing Interface)**: For client-server communication, enabling parallel execution.
- **PTY (Pseudo-terminal)**: For simulating terminal interactions during command execution.
- **Xterm**: To run Python scripts in isolated terminal windows for each client.
- **UUID**: To create unique directories for each server session to avoid name collisions.
- **Python**: The language used to implement both the client and server logic.

5 System Flow

The system follows this flow:

1. **Initialization**: The server creates a temporary directory for command execution, and clients launch terminal windows running Python scripts.
2. **Command Execution**: Clients input commands that are sent to the server for execution.
3. **Output Return**: The server sends the command output back to the respective client.
4. **Termination**: Upon receiving the `exit` command, the server disconnects the client and cleans up all temporary files.

6 Example Usage

6.1 Setup Instructions

- Open the `setup.sh` file.
- Decide whether to use a new environment or an existing one:
  - If you choose to use a new environment, enter the name of the environment you wish to create.
  - If you choose to use an existing environment, enter the Path/To/Environment.
- The `setup.sh` file will:
  - Install Python 3 and pip.
  - Create a new environment, then the user names it or activates the existing environment.
  - Install MPICH to enable MPI communication.
  - Install the `mpi4py` package for Python-based MPI communication.
  - Install Xterm for terminal window support.
  - Save the project code file and name it `remote_shell_MPI.py`.
  - Run the system with one server and three clients.

6.2 Server Execution

To run the system, execute the server script with the following command:

```
mpiexec -n 4 remote_shell_MPI.py
```

This runs the server with 4 processes (one server and three clients).

6.3 Client Interaction

Each client opens a terminal and waits for shell commands. For example:

```
[Client 1]$ ls
[Server] Executing: ls
[Executed on server in /tmp/mpi_server_abcdef12]
```

7 Diagrams and Documentation

7.1 Use Case Diagram

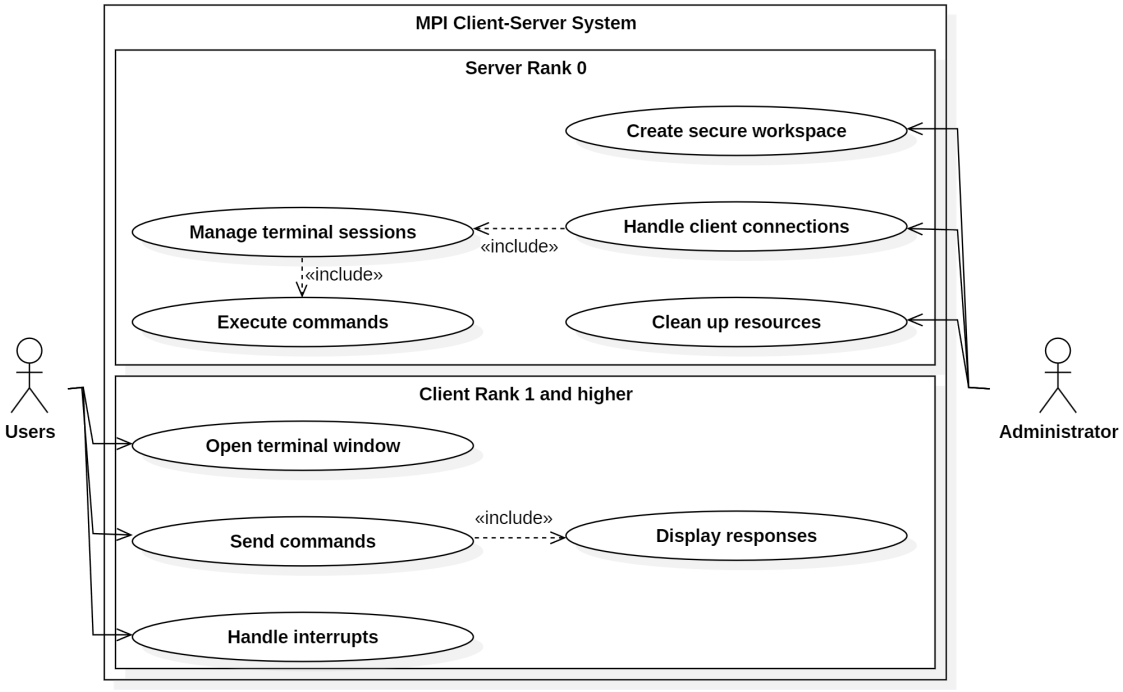

Figure 1: Use Case

7.2 Class Diagram

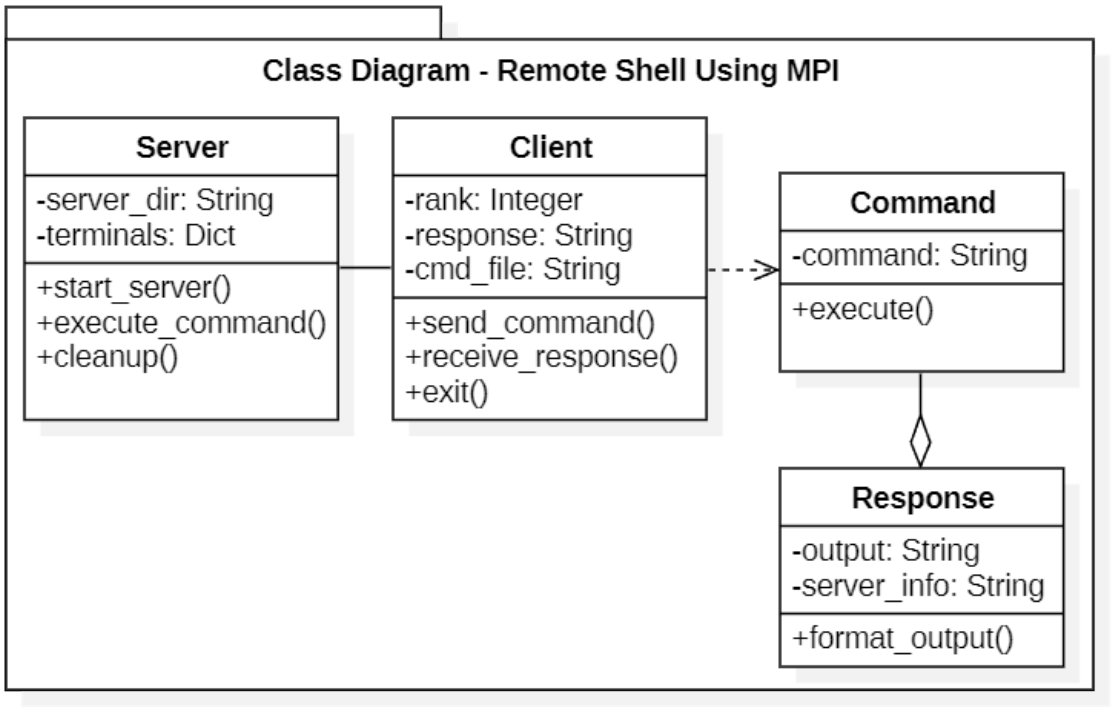

Figure 2: Class Diagram

7.3 Sequence Diagram

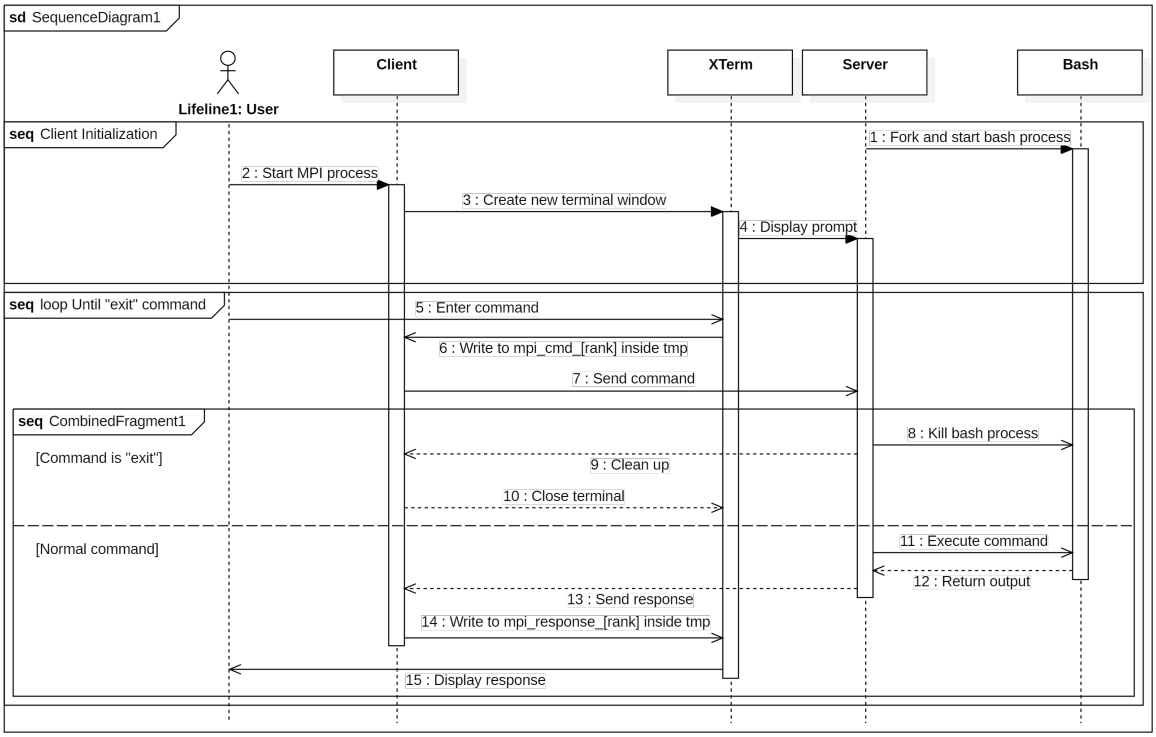

Figure 3: Sequence Diagram

7.4 Activity Diagram

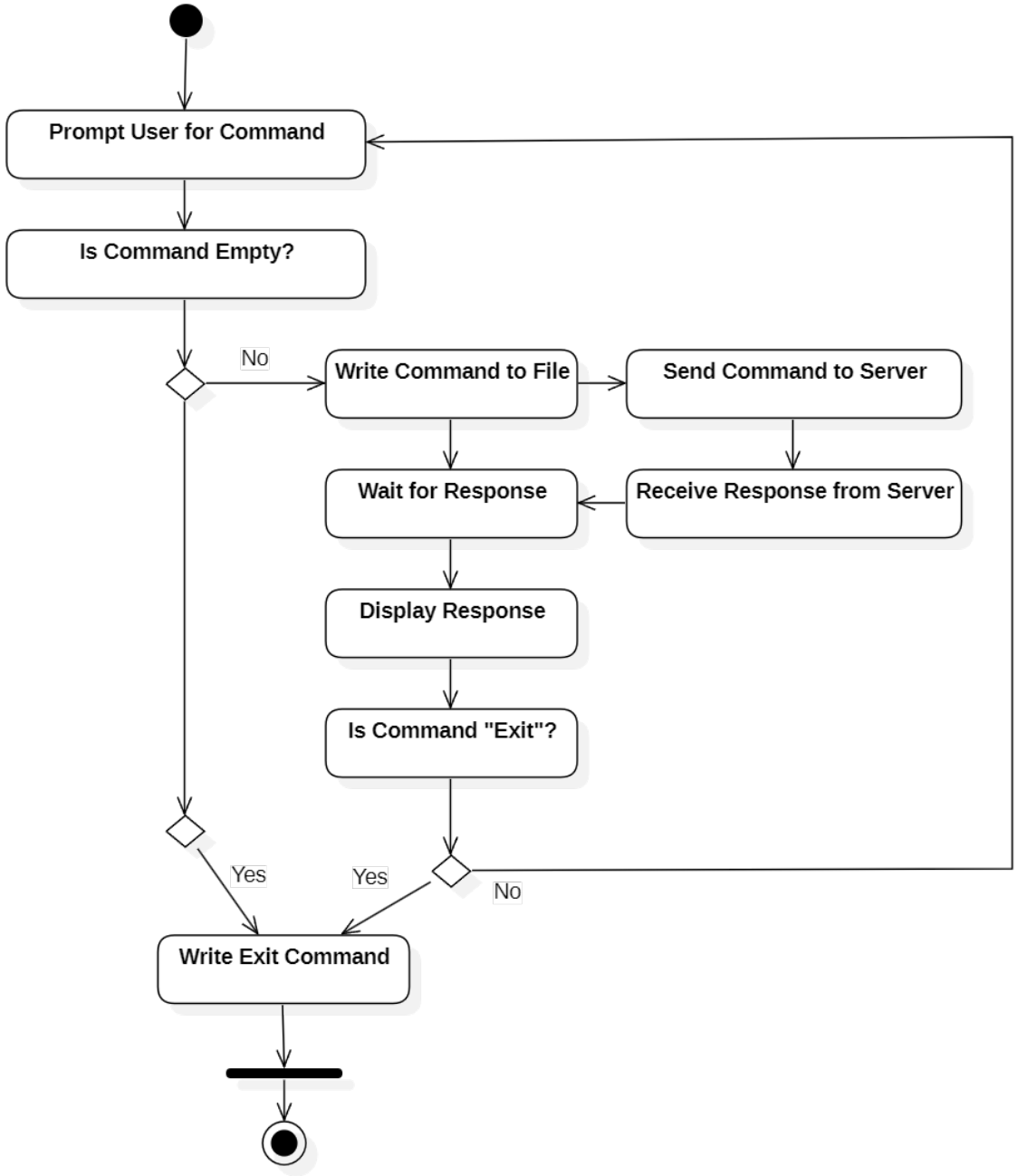

Figure 4: Activity Diagram

7.5 Flowchart

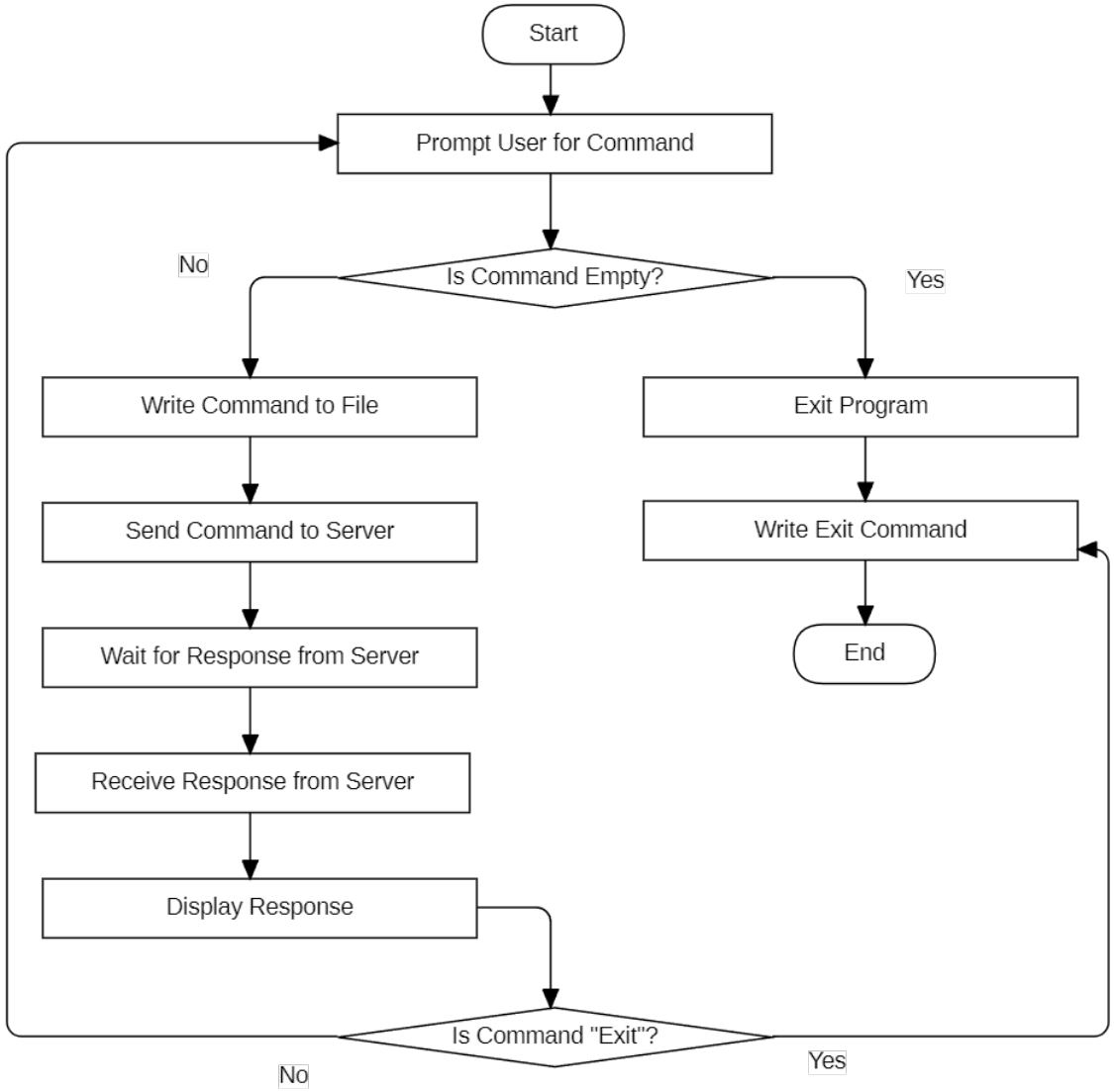

Figure 5: Flowchart

8 Challenges and Solutions

8.1 Challenges

- Handling multiple clients concurrently.
- Managing client disconnections and ensuring proper cleanup.
- Ensuring reliable and efficient communication between server and clients.

8.2 Solutions

- Used MPI tags to differentiate messages from different clients.
- Implemented robust error handling and client disconnection management.
- Used `MPI.Barrier()` for synchronization.

9 Conclusion

This project successfully implements a remote shell using MPI, allowing multiple clients to execute commands on a server concurrently. The system is robust, scalable, and provides a strong foundation for further development. Comprehensive documentation and diagrams ensure that the project can be easily understood and recreated.