University Of Science And Technology Of Hanoi
ICT Department

MIDTERM REPORT

Title: Remote Shell using RPC

Subject: Distributed Systems (DS2026)

Group: Group 17

Members:
1. Nguyen Tien Duy – 22BA13102
2. Bui Truong An – 22BA13001
3. Nguyen Phuong Anh – 22BA13020
4. Tran Thuong Nam Anh – 22BA13032
5. Tran Thuc Anh – 23BI14030
6. Nguyen Thi Vang Anh– 23BI14032
7. Luong Quynh Nhi – 23BI14356

Major: Cyber Security

Contents

1 Introduction

1.1 Overview of RPC

RPC (Remote Procedure Call) is a protocol that enables a computer program to execute a procedure (subroutine) on another computer as if it were executing on the local machine, without requiring the programmer to explicitly code the remote interaction details.

How RPC Works

1. Client calls a local procedure (stub)
2. Stub marshals the parameters into a message
3. Message is sent over the network to the server
4. Server receives the message and unmarshals the parameters
5. Server executes the actual procedure
6. Result is marshaled and sent back to the client
7. Client receives and unmarshals the result

1.2 Project Objectives

This project implements a distributed remote shell system utilizing Golang's native Remote Procedure Call (RPC) framework. The primary objective is to allow multiple clients to connect to a central server simultaneously and execute shell commands remotely in a safe and efficient manner. This system demonstrates fundamental concepts of distributed computing, including client-server architecture, synchronous communication via RPC, and concurrent request handling.

1.3 Technologies Used

- **Programming Language:** Golang (Go 1.21+)
- **RPC Framework:** `net/rpc` (Go's built-in package)
- **Network Protocol:** TCP/IP
- **Command Execution Library:** `os/exec`

2 System Design

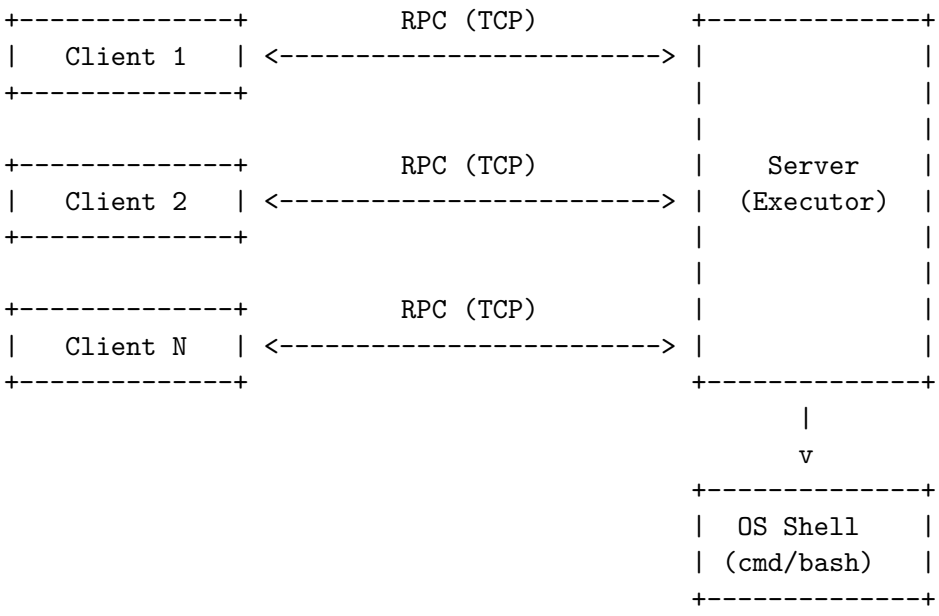
2.1 Architectural Overview

The system is architected as a distributed client-server application utilizing a synchronous Remote Procedure Call (RPC) model. The design decouples the user interface (Client) from the execution environment (Server), allowing for centralized management of resources while supporting distributed access.

- **Design Pattern:** Master-Slave (Server-Client) pattern where the Server acts as the source of truth and execution power.
- **Transport Layer:** The system operates over **TCP/IP**, ensuring reliable, ordered delivery of command streams and results.
- **Scalability:** The architecture is designed to be horizontally scalable regarding client connections. The server employs a concurrent connection model using Goroutines.

2.2 Component Diagram

The interaction between the components is illustrated below:



3 RPC Communication Flow

This section details the lifecycle of a single command execution request, illustrating how data traverses the network via the `net/rpc` package.

3.1 The RPC Handshake & Protocol

- **Serialization (Marshaling):** Go's `gob` encoding is used to serialize complex data structures into byte streams for network transmission.
- **Service Method:** The RPC endpoint is exposed as `ShellService.ExecuteCommand`.

3.2 Execution Sequence

The detailed lifecycle of a remote command execution follows the flow illustrated in the sequence diagram above:

1. **Connect (TCP):** The Client initiates a TCP handshake to establish a reliable network connection with the Server.
2. **RPC Call:** The Client invokes the remote method `ExecuteCommand` (e.g., sending the command "ls"). This request is serialized and transmitted to the Server.
3. **System Execution:** The Server receives the RPC request and spawns a system subprocess using `os/exec`. It dynamically selects the executor based on the OS: `cmd /C` for Windows or `sh -c` for Linux/Unix.
4. **Command Processing:** The underlying OS Shell processes and executes the requested command logic locally on the server machine.
5. **Output Retrieval:** Upon completion, the Shell returns the captured `stdout` (standard output) and `stderr` (standard error) streams to the Server process.
6. **Marshal Response:** The Server populates the response structure with the output data and marshals (serializes) it into a byte stream.
7. **RPC Response:** The Server transmits the encoded response packet back to the waiting Client over the established TCP connection.
8. **Display Result:** The Client receives the data, unmarshals the response, and renders the final output to the user's terminal.

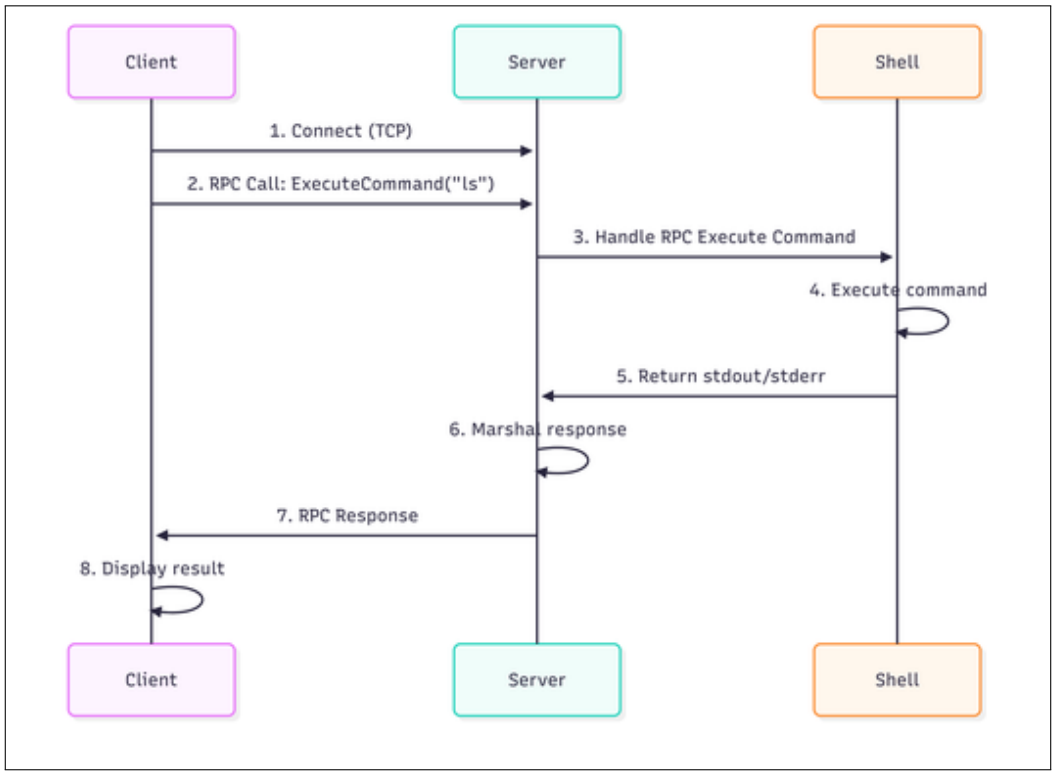


Figure 1: RPC Communication Flow

4 Server Implementation

The server is the core component responsible for safe and concurrent command execution.

4.1 Concurrency Management

To handle multiple clients simultaneously without blocking, the server utilizes **Goroutines**. The listener loop spawns a lightweight thread for each incoming connection:

```
func main() {
    // ... setup RPC ...
    listener, err := net.Listen("tcp", ":8080")

    for {
        conn, err := listener.Accept()
        if err != nil {
            continue
        }
        // Handle each client in a separate goroutine
        go rpc.ServeConn(conn)
    }
}
```

Listing 1: Server Concurrency Pattern

4.2 Command Execution Logic

The core logic resides in the `ExecuteCommand` function. To ensure cross-platform compatibility, the server dynamically selects the shell executor:

```
if runtime.GOOS == "windows" {
    // Use cmd /C for Windows
    cmd = exec.Command("cmd", "/C", request.Command)
} else {
    // Use sh -c for Linux/Unix
    cmd = exec.Command("sh", "-c", request.Command)
}
```

Listing 2: Cross-Platform Execution Logic

The server captures both standard output and standard error using `CombinedOutput()` or separate pipes to ensure full visibility of the execution status.

5 Client Implementation

The client operates as a lightweight terminal interface. It is responsible for capturing user commands, transmitting them to the server, and rendering the results. The implementation is divided into three key stages: Connection, Input Processing, and RPC Invocation.

5.1 Establishing Connection

Upon startup, the client attempts to establish a TCP connection with the server using the standard `net/rpc` library. The connection logic is synchronous and fails fast if the server is unreachable.

```
func main() {
    // Connect to the server on port 8080
    client, err := rpc.Dial("tcp", "localhost:8080")
```

```
    if err != nil {
        log.Fatal("Connection error: ", err)
    }

    fmt.Println("Connected to Server. Type 'exit' to quit.")
    // ... enter REPL loop ...
}
```

Listing 3: Client Connection Logic

5.2 Input Processing and Sanitization

Inside the main loop, the client reads input from `stdin`. Critical to this step is the sanitization of input, specifically removing newline characters (`\n` or `\r\n`) which can cause command execution errors on the server side.

```
reader := bufio.NewReader(os.Stdin)

for {
    fmt.Print("remote-shell> ")

    // Read string until newline
    text, _ := reader.ReadString('\n')

    // Trim whitespace and newline characters
    text = strings.TrimSpace(text)

    if text == "exit" {
        fmt.Println("Disconnecting...")
        break
    }
    // ... proceed to RPC call ...
}
```

Listing 4: Reading User Input

5.3 RPC Invocation and Response Handling

This is the core of the client logic. The client wraps the sanitized text into a `CommandRequest` struct and invokes the remote method `ShellService.ExecuteCommand`.

The client waits synchronously for the `CommandResponse`. Upon receipt, it checks the `ExitCode` to decide whether to print to `Stdout` or `Stderr`.

```
// Prepare the request object
req := &shared.CommandRequest{Command: text}
var res shared.CommandResponse

// Synchronous RPC call
// "ShellService.ExecuteCommand" matches the server's registered method
err = client.Call("ShellService.ExecuteCommand", req, &res)

if err != nil {
    log.Println("RPC Error:", err)
    continue
}

// Display output based on execution status
if res.ExitCode == 0 {
    fmt.Print(res.Stdout) // Success output
} else {
```

```
    fmt.Print(res.Stderr) // Error output
}
```

Listing 5: Executing RPC Call

6 Deployment Guideline

6.1 System Requirements

Before deploying and running the project, the system must have Go (Golang) installed.

Steps to install Go on Windows:

- Download Go from `https://go.dev/dl/`
- Select the version `go1.21.x.windows-amd64.msi`
- Install Go and restart the terminal after installation

Verify the Go installation:

```
go version
```

6.2 Deployment Using Scripts

This is the simplest and recommended method to build and run the system.

Build the entire project:

```
build.bat
```

Run the server (Terminal 1):

```
run-server.bat
```

Run the client (Terminal 2):

```
run-client.bat
```

6.3 Manual Build and Execution

If scripts are not used, the server and client can be built and executed manually.

Build the server:

```
cd server
go build -o server.exe main.go
cd ..
```

Build the client:

```
cd client
go build -o client.exe main.go
cd ..
```

Run the server (Terminal 1):

```
cd server
server.exe
```

Run the client (Terminal 2):

```
cd client
client.exe
```

6.4 Running with `go run`

This method allows direct execution without building binaries and is suitable for quick testing.

Run the server (Terminal 1):

```
cd server
go run main.go
```

Run the client (Terminal 2):

```
cd client
go run main.go
```

6.5 Testing with Multiple Clients

The system supports multiple concurrent client connections.

Start the server:

```
cd server
go run main.go
```

Start multiple clients in different terminals:

```
cd client
go run main.go
```

Test commands on each client:

```
echo "Hello from client 1"
dir or ls
whoami
date
```

6.6 Troubleshooting

Error: `'go' is not recognized`

Possible causes:

- Go is not installed
- Go is not added to the system PATH

Solution:

```
Install Go from https://go.dev/dl/ and restart the terminal
```

Error: cannot find package `remote-shell-rpc/shared`

Solution:

```
go mod tidy
```

Server does not start

- Check whether port 8080 is already in use

- Run the server with administrator privileges if required

Client cannot connect to server

- Ensure the server is running
- Verify that the firewall is not blocking port 8080

6.7 Usage Example

After the client successfully connects to the server:

```
remote-shell> echo "Hello RPC!"
Hello RPC!
Exit Code: 0

remote-shell> cd c:\
Exit Code: 0

remote-shell> dir
[Directory listing]
Exit Code: 0

remote-shell> exit
Disconnecting from server...
```

7 Testing Scenarios and Results

This section describes the testing scenarios conducted to evaluate the correctness, concurrency, and robustness of the Remote Shell RPC system. The tests focus on single-client functionality, multi-client concurrency, error handling, and basic performance behavior.

7.1 Test Scenario 1: Single Client Connection

Objective: Verify that a single client can successfully establish an RPC connection with the server and execute remote shell commands.

Test Setup:

- One server instance running on port 8080
- One client connected to the server

Test Procedure:

```
cd server
go run main.go
```

```
cd client
go run main.go
```

Executed Commands:

```
echo "Hello RPC!"
cd
dir
whoami
date /t
```

Expected Result:

- Client connects successfully to the server
- Commands are executed on the server side
- Output and exit codes are returned correctly

Actual Result: All commands executed successfully and the client received the correct output and exit codes.

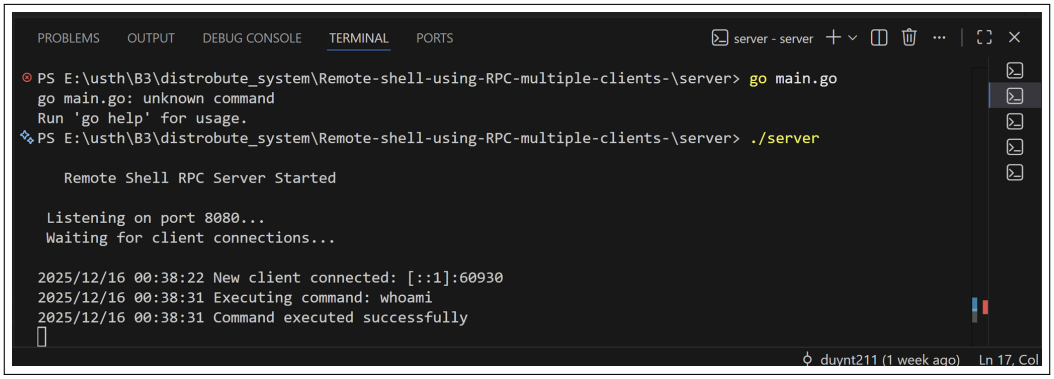


Figure 2: Server Side Logs

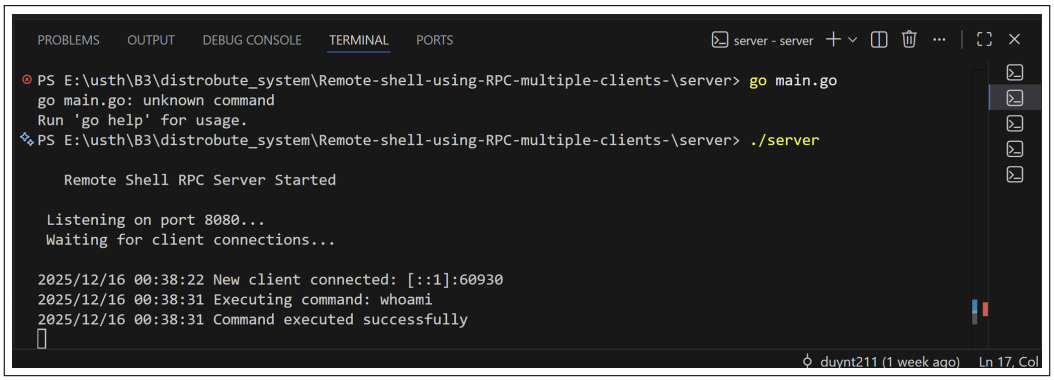


Figure 3: Client Side Output

7.2 Test Scenario 2: Multiple Concurrent Clients

Objective: Verify that the server can handle multiple concurrent client connections using RPC without blocking or performance degradation.

Test Setup:

- One server instance
- Four concurrent client instances

Test Procedure:

```
cd server
go run main.go
```

Each client was started in a separate terminal:

```
cd client
go run main.go
```

Executed Commands (per client):

```
echo "Client connected"
whoami
date /t
dir
```

Expected Result:

- All clients connect successfully
- Commands from different clients execute simultaneously
- No client blocks another client

Actual Result: The server accepted multiple concurrent connections and processed commands from all clients correctly without noticeable delay or conflict.

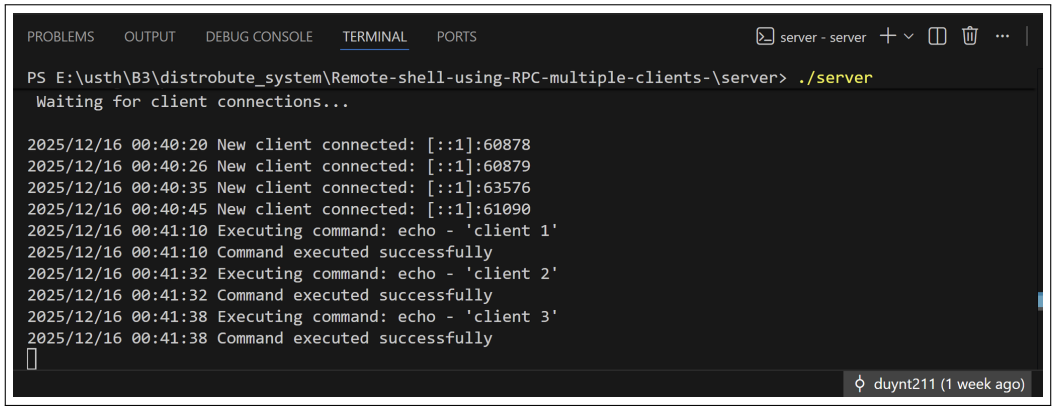


Figure 4: Multiple client Server Side

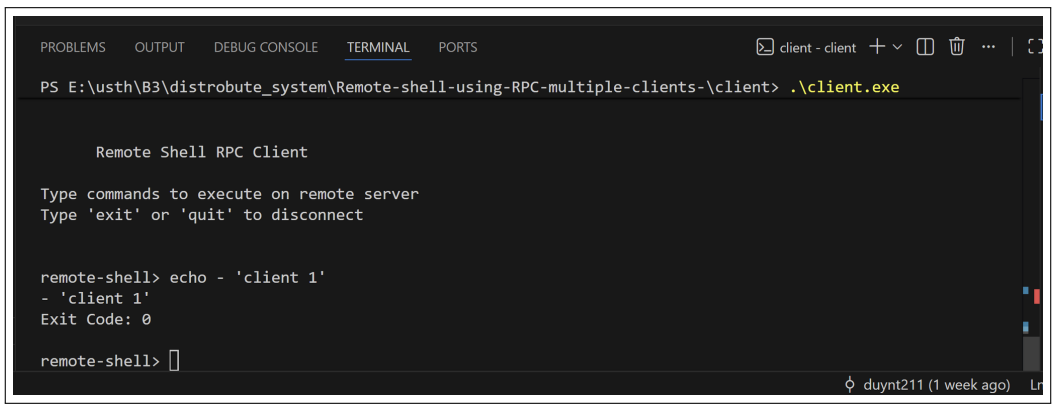


Figure 5: Client 1 Side Output

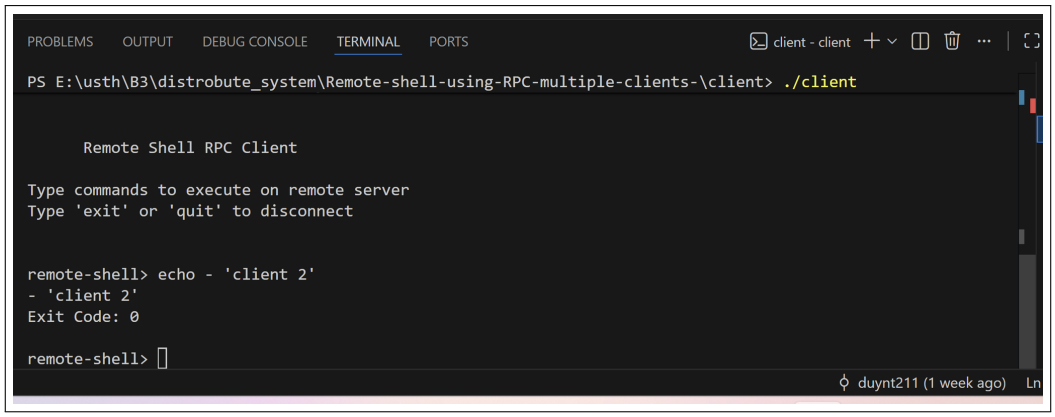


Figure 6: Client 2 Side Output

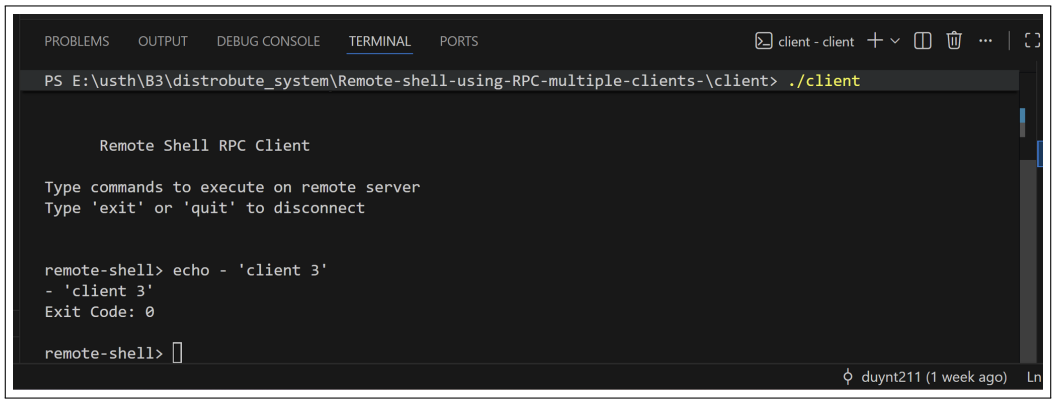


Figure 7: Client 3 Side Output

7.3 Test Scenario 3: Cross-Machine Network Communication

Objective: To verify that the RPC system functions correctly over a real Local Area Network (LAN) involving distinct physical machines, proving true remote execution capabilities.

- **Environment Configuration:**
  - **Server Machine:** IP Address `172.20.10.5`
  - **Client Machine:** IP Address `172.20.10.4`
- **Procedure:**
  1. **Step 1 (Server Side):** On the Server machine, navigate to the project directory and start the service:

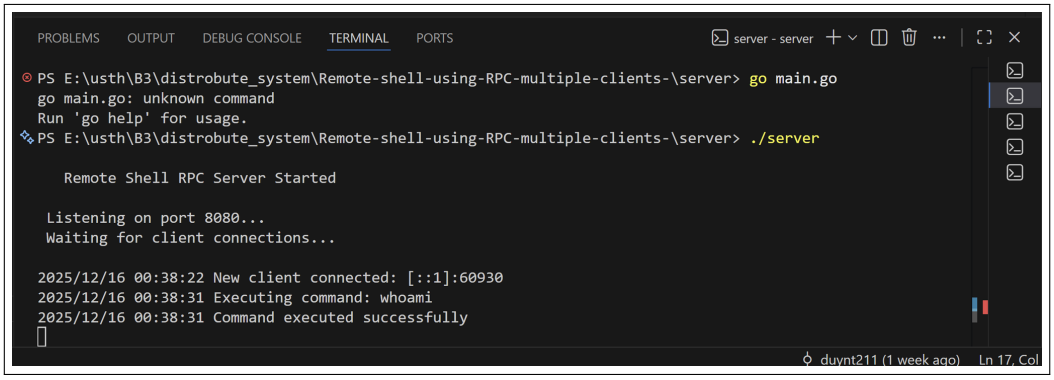
     ```
     go run main.go
     ```

  2. **Step 2 (Client Side):** On the Client machine, navigate to the client directory and initiate the connection targeting the Server's specific IP address:

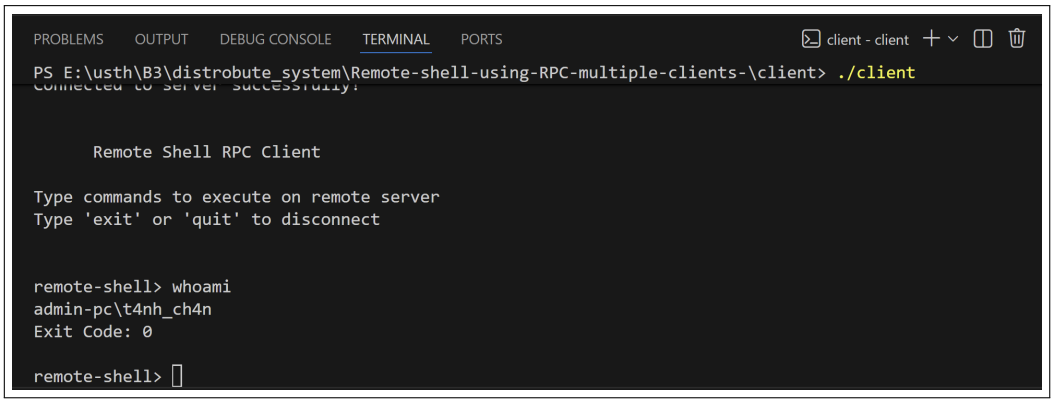
     ```
     cd client
     go run main.go 172.20.10.5:8080
     ```

  3. **Step 3:** Execute the `whoami` command to confirm the execution is happening on the remote server, not the local client.

- **Result:**
  - The Client successfully established a TCP connection to `172.20.10.5:8080`.
  - The `whoami` command returned the user identity of the Server machine, confirming that the command was transmitted and executed remotely.



(a) Server Logs (IP: 172.20.10.5) - Receiving connection



(b) Client Terminal (IP: 172.20.10.4) - Executing remote commands

Figure 8: Successful remote command execution across two different machines in LAN.

7.4 Performance and Stability Testing

Objective: Assess basic system stability under increased client load.

Test Description:

- Server running continuously
- Up to 10 clients connected sequentially
- Each client executed multiple commands

Observed Metrics:

- Command response time
- Connection stability
- Server CPU and memory usage

Result: The server maintained stable operation with no connection failures or crashes during the test period.

7.5 Summary of Test Results

- Single-client RPC communication works correctly
- Multiple clients are supported concurrently
- Error handling is reliable and stable
- System remains stable under moderate load

8 Conclusion

The Remote Shell using RPC project successfully demonstrates the power and simplicity of Go for building distributed systems. By leveraging `net/rpc` and Goroutines, the application achieves efficient, concurrent remote command execution with minimal code complexity. The testing phase confirmed that the system is reliable under multi-user scenarios and resilient to erroneous inputs.

8.1 Knowledge Gained

Through this mid-term project, our team learned:

1. **RPC Fundamentals:** Understanding RPC mechanisms, marshaling/unmarshaling processes
2. **Concurrency in Go:** Utilizing goroutines for concurrent request processing
3. **Network Programming:** TCP/IP communication and client-server architecture
4. **System Programming:** Executing shell commands and handling processes
5. **Distributed Systems:** Remote execution and error handling in distributed environments

8.2 Future Improvements

The system can be further enhanced with:

- **Security:** Add authentication and encryption (TLS)
- **Authorization:** Implement command whitelist/blacklist
- **Session Management:** Maintain working directory for each client session
- **Command History:** Store and retrieve command history
- **File Transfer:** Add file upload/download capabilities
- **Web Interface:** Create a web-based client using WebSockets

9 Team Contribution

Team Member	Responsibilities	Contribution
Nguyen Tien Duy (Leader)	Review, Debugged and resolved, Designed and implemented RPC and Protocol	16%
Bui Truong An	Server implementation, Report writing	14%
Nguyen Phuong Anh	Server implementation, Report writing	14%
Luong Quynh Nhi	Client implementation, Presentation	14%
Tran Thuong Nam Anh	Server implementation, Slides	14%
Tran Thuc Anh	Client implementation, Slides	14%
Nguyen Thi Vang Anh	Client implementation, Slides	14%
Total		**100%**

Table 1: Team Member Contributions and Work Distribution

10 Acknowledgments

We would like to sincerely thank our professor and teaching assistants at the University of Science and Technology of Hanoi for their invaluable guidance and support throughout this mid-term project. Their expertise in Distributed Systems has been instrumental in helping us complete this work successfully.

References

1. Go Programming Language Documentation - `https://go.dev/doc/`
2. Go RPC Package Documentation - `https://pkg.go.dev/net/rpc`
3. Distributed Systems: Principles and Paradigms - Andrew S. Tanenbaum and Maarten Van Steen
4. Remote Procedure Call - Wikipedia - `https://en.wikipedia.org/wiki/Remote_procedure_call`
5. Concurrency in Go - Katherine Cox-Buday, O'Reilly Media