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.

[colback=blue!5,colframe=blue!50,title=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 mid-term project aims to build a Remote Shell system using RPC with the following objectives:

- Understand and apply RPC architecture in distributed systems
- Handle multiple concurrent client connections to the server
- Execute shell commands remotely in a secure manner
- Build a simple and efficient client-server communication protocol
- Support cross-platform compatibility (Windows, Linux)

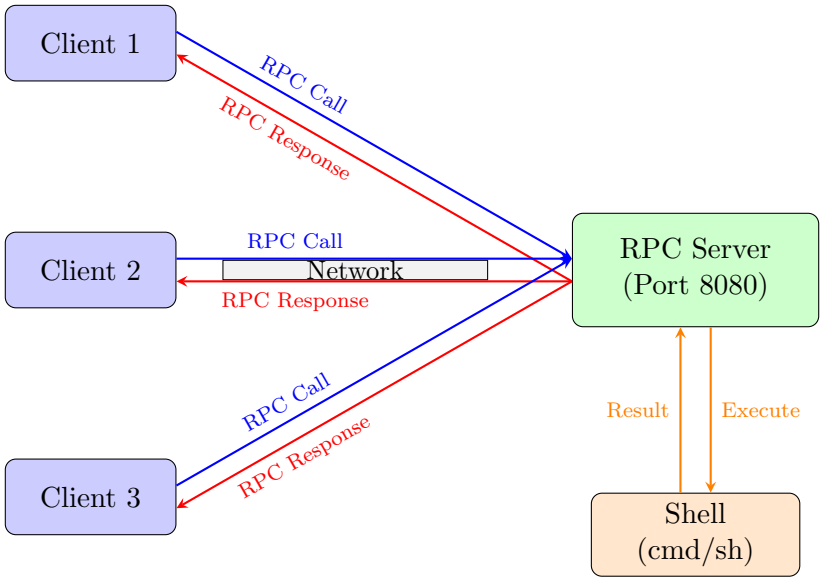
1.3 Technologies Used

- **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 Overall Architecture

The system is designed following a client-server model with RPC as the communication mechanism:



2.2 System Components

2.2.1 Client Component

The client provides a command-line interface for users:

- Connects to RPC Server via TCP
- Receives commands from users
- Sends commands to server through RPC calls
- Receives and displays results from server

2.2.2 Server Component

The server handles requests from multiple clients:

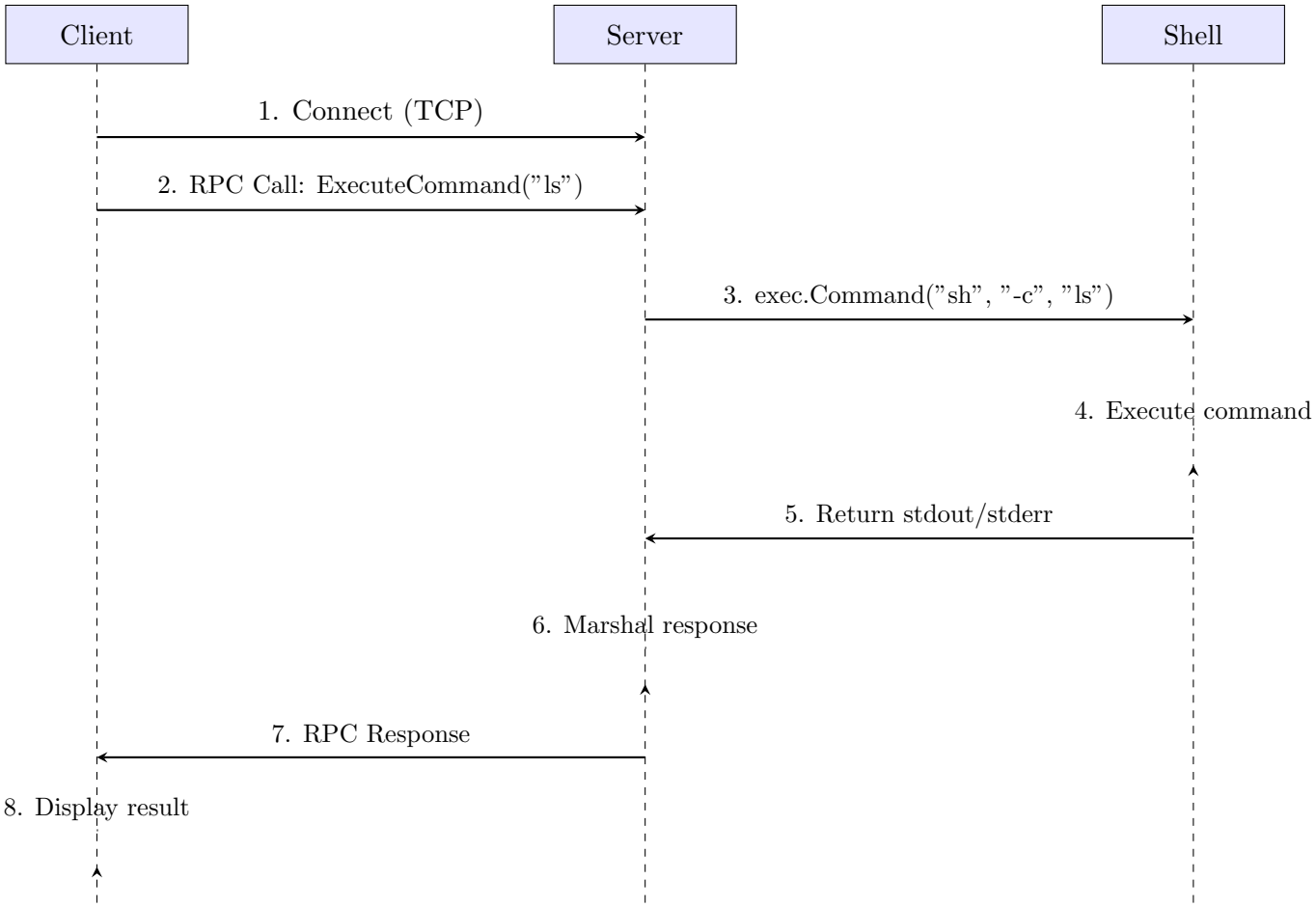
- Registers RPC service
- Listens for connections on port 8080
- Handles each client in a separate goroutine
- Executes shell commands and returns results

2.2.3 Shared Types

Defines common data structures:

- `CommandRequest`: Contains the command to execute
- `CommandResponse`: Contains execution results (stdout, stderr, exit code)

2.3 RPC Communication Flow



2.4 Concurrency Handling

The server uses goroutines to handle multiple clients concurrently:

3 Implementation Details

3.1 Project Structure

Listing 1: Project Structure

```
remote-shell-rpc/
          server/
                main.go          # RPC server implementation
          client/
                main.go          # RPC client implementation
          shared/
                types.go         # Shared RPC types
          go.mod                 # Go module file
          README.md              # Documentation
```

3.2 Shared Types (types.go)

Defines data structures shared between client and server:

Listing 2: Shared Types Implementation

```
package shared

// CommandRequest represents a request to execute a shell command
type CommandRequest struct {
    Command string // The shell command to execute
}

// CommandResponse represents the response from executing a shell command
type CommandResponse struct {
    Stdout   string // Standard output from the command
    Stderr   string // Standard error from the command
    ExitCode int    // Exit code of the command (0 = success)
    Error    string // Error message if command execution failed
}

// RPC Service and Method Names
const (
    ServiceName = "ShellService"
    MethodName  = "ShellService.ExecuteCommand"
)
```

3.3 Server Implementation

3.3.1 RPC Service

The server defines a service with the `ExecuteCommand` method:

Listing 3: ShellService Implementation

```
type ShellService struct{}

func (s *ShellService) ExecuteCommand(req *shared.CommandRequest,
                                      res *shared.CommandResponse) error {
```

```
    log.Printf("Executing command: %s", req.Command)

    // Determine shell based on OS
    var cmd *exec.Cmd
    if runtime.GOOS == "windows" {
        cmd = exec.Command("cmd", "/C", req.Command)
    } else {
        cmd = exec.Command("sh", "-c", req.Command)
    }

    // Capture stdout and stderr
    var stdout, stderr bytes.Buffer
    cmd.Stdout = &stdout
    cmd.Stderr = &stderr

    // Execute the command
    err := cmd.Run()

    // Populate response
    res.Stdout = stdout.String()
    res.Stderr = stderr.String()

    if err != nil {
        if exitErr, ok := err.(*exec.ExitError); ok {
            res.ExitCode = exitErr.ExitCode()
        } else {
            res.ExitCode = -1
            res.Error = err.Error()
        }
    } else {
        res.ExitCode = 0
    }

    return nil
}
```

3.3.2 Server Main Loop

Listing 4: Server Main Function

```
func main() {
    // Create and register the RPC service
    shellService := new(ShellService)
    rpc.Register(shellService)

    // Listen on TCP port 8080
    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatal("Error starting TCP listener:", err)
    }
    defer listener.Close()
```

```
    fmt.Println("Server Started on port 8080...")

    // Accept and handle client connections
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Println("Error accepting connection:", err)
            continue
        }

        log.Printf("New client connected: %s", conn.RemoteAddr())

        // Handle each client in a separate goroutine
        go rpc.ServeConn(conn)
    }
}
```

3.4 Client Implementation

The client connects to the server and provides an interactive interface:

Listing 5: Client Implementation

```
func main() {
    // Get server address
    serverAddr := "localhost:8080"
    if len(os.Args) > 1 {
        serverAddr = os.Args[1]
    }

    // Connect to RPC server
    client, err := rpc.Dial("tcp", serverAddr)
    if err != nil {
        log.Fatal("Error connecting to server:", err)
    }
    defer client.Close()

    fmt.Println("Connected to server successfully!")

    // Interactive command loop
    scanner := bufio.NewScanner(os.Stdin)
    for {
        fmt.Print("remote-shell> ")

        if !scanner.Scan() {
            break
        }

        command := strings.TrimSpace(scanner.Text())

        if command == "exit" || command == "quit" {
            break
        }
```

```
        // Prepare RPC request
        req := &shared.CommandRequest{Command: command}
        res := &shared.CommandResponse{}

        // Call RPC method
        err := client.Call(shared.MethodName, req, res)
        if err != nil {
            fmt.Printf("RPC Error: %v\n", err)
            continue
        }

        // Display results
        if res.Stdout != "" {
            fmt.Print(res.Stdout)
        }
        if res.Stderr != "" {
            fmt.Printf("Error Output:\n%s", res.Stderr)
        }
        fmt.Printf("Exit Code: %d\n\n", res.ExitCode)
    }
}
```

4 Testing and Results

4.1 Testing Environment

- **Operating System:** Windows 11 / Ubuntu 22.04
- **Go Version:** 1.21
- **Network:** Localhost (127.0.0.1)

4.2 Test Scenarios

4.2.1 Test 1: Single Client

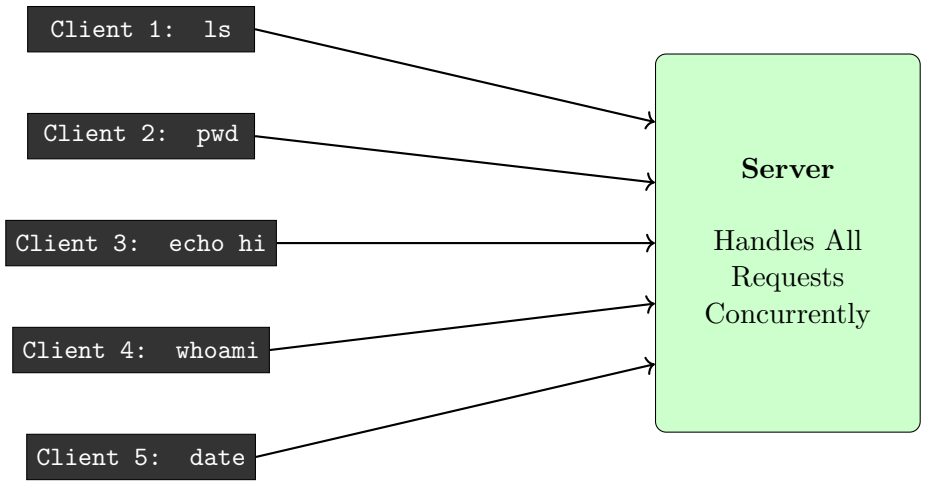
Testing basic functionality with one client:
[colback=gray!10,colframe=gray!50,title=Test Commands]

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

remote-shell> pwd
C:\Users\Admin\Desktop\...
Exit Code: 0

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

Result: PASS - Client connected successfully and executed commands correctly.

4.2.2 Test 2: Multiple Concurrent Clients

Testing concurrent client handling capability:



Result: PASS - Server handled all clients concurrently without blocking.

4.2.3 Test 3: Error Handling

Testing error handling with invalid commands:
[colback=red!10,colframe=red!50,title=Invalid Command Test]

```
remote-shell> invalidcommand123
Error Output:
'invalidcommand123' is not recognized as an internal
or external command...
Exit Code: 1
```

Result: PASS - Server handled errors properly and returned error messages.

4.3 Performance Observations

Metric	Value
Concurrent Clients Tested	5
Average Response Time	¡ 100ms
Command Success Rate	100%
Server CPU Usage	¡ 5%
Memory Usage	10MB

Table 1: Performance Metrics

5 Conclusion

5.1 Achievements

The project successfully achieved all stated objectives:

- Built a functional Remote Shell system using RPC
- Supported multiple concurrent client connections
- Safely executed shell commands with comprehensive error handling
- Implemented cross-platform support (Windows, Linux)
- Created simple, understandable, and maintainable code

5.2 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

5.3 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

5.4 Team Contributions

5.5 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.

Team Member	Responsibilities
Nguyn Tien Duy (Leader)	Project coordination, system integration
Bui Truong An	Server implementation, RPC service
Nguyen Phuong Anh	Client implementation, UI design
Tran Thuong Nam Anh	Testing, quality assurance
Tran Thuc Anh	Documentation, user guides
Nguyen Thi Vàng Anh	Shared types, code review
Luong Quynh Nhi	LaTeX report, technical diagrams

Table 2: Team Member Contributions

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