Development of a Cross-Platform Remote Shell with Authentication and Resource Monitoring Using Go and TCP Sockets

Freddy L. Vega Cisneros and Juan N. Cabrera Miranda, Students, Faculty of Basic Sciences and Engineering, Universidad de los Llanos, Villavicencio, Colombia

Abstract—This paper presents the development of a cross-platform remote shell system implemented in the Go programming language, which allows secure communication between a Windows client and a Linux server. The main objective is to enable remote command execution and real-time monitoring of system resources, using TCP sockets and concurrent processing through goroutines. The server authenticates users based on a configuration file and a plaintext user database with passwords encrypted using SHA-256. Once authenticated, the client can send UNIX commands that are executed remotely and receive periodic reports on CPU, memory, disk and active process usage. These tasks are executed in parallel without blocking interaction with the commands. The system was evaluated under different values of the reporting interval parameter (n seconds), measuring the impact of this on the average CPU and memory consumption of the server. The results show that values of n greater than 5 seconds provide a good balance between update frequency and efficient use of resources.

Index Terms—Authentication, Cross-platform software, Go programming language, Remote command execution, Resource monitoring, TCP sockets

I. Introduction

THIS article presents the development of a program for a cross-platform remote shell developed in the Go programming language. The system allows connection between a client on Windows and a server on Linux using TCP sockets, in order to execute commands remotely and receive periodic information about the resource consumption of the monitored system.

The server manages user authentication using credentials stored in a flat file, with passwords encrypted using the SHA-256 hash algorithm. Once authenticated, the client can send commands that will be executed on the remote system, and will also receive automatic reports that include memory, disk, processor, and running process usage.

All server configuration is defined through a .conf file, which sets parameters such as the IP allowed for connection, the listening port, the number of failed authentication attempts, and the list of allowed users. Communication between client and server is performed concurrently using goroutines, which allows commands to be executed and reports to be generated independently without blocking the connection.

The client must be run with three parameters: the server's IP address, the connection port, and the time interval in seconds for receiving reports. Once connected, it requests the user's credentials and allows remote interaction. The system remains active until the “bye” exit command is received. This article describes the system architecture, the main components of its implementation, the results obtained during operational testing, and the conclusions drawn from the development of the application.

II. Requirements and Tools

A. Functional requirements

The system consists of two modules that communicate remotely: a server running on Linux (Debian) and a client running on Windows. Both interact via TCP sockets and are developed in the Go language. The main features of each are described below:

Server (Linux):

- Reads a configuration file (config.conf) to obtain: the client's allowed IP address, the port through which it will serve the connection, the maximum number of failed authentication attempts, and the list of allowed users.
- Opens a TCP socket and waits for an incoming connection. Once the connection is established, it verifies whether the client's IP address is allowed

or not. If it is not allowed, it closes the connection immediately.

- It verifies the client's credentials using a flat file (users.bd) that contains user:password pairs, with the password encrypted using the SHA-256 algorithm.
- It sends periodic reports on system resource consumption (memory, CPU, processes, and disk) according to the interval specified remotely. It executes commands received from the client and returns the results through the socket.

Client (Windows):

- Runs with the following parameters in the console: <IP> <Port> <reportPeriod>. For example: ./client.exe 10.1.10.3 2025 5
- Asks the user for access credentials.
- Allows remote commands to be sent.
- Displays the reports sent by the server every n seconds on the screen.
- Ends the connection when the bye command is entered.

B. *Tools used*

- Programming language: Go (version 1.24.3)
- Server operating system: Linux (Debian).
- Client operating system: Windows.
- Concurrency management: Go goroutines.
- Communication between modules: TCP sockets.
- Password encryption: SHA-256 algorithm (implemented with Go's crypto/sha256 package).

C. *Important files*

- config.conf: contains the server configuration, including allowed IP, port, number of failed attempts, and enabled users.
- users.bd: plain text database with users and hashed passwords.
- client.exe: client executable.
- server: server executable (compiled for Linux).

III. System Architecture

The system is based on a client-server architecture over TCP, where the server operates in a Linux environment and the client in a Windows environment. Both components communicate through sockets, allowing remote command execution and periodic sending of resource usage reports. The implementation is organized into multiple source files in Go, with well-defined responsibilities for each module.

A. *Client-server architecture diagram*

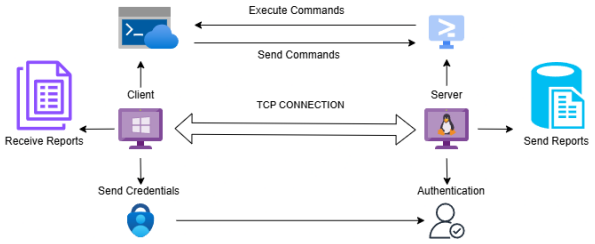

Fig. 1. General architecture of the client-server system

Figure 1 shows the general architecture of the system. In the server environment (Linux), the entry point is main.go, which configures the socket and delegates the handling of each client to the handleClient() function in the connection.go file. This function handles user validation, executes commands using commands.go, and sends periodic reports via reporter.go. All of these processes are executed in parallel using goroutines.

On the client side (Windows), the main.go file is responsible for initiating the connection to the server via connection.go, requesting credentials from the user (login.go), and managing both command execution (commands.go) and reporting (report.go).

B. *Authentication flow diagram*

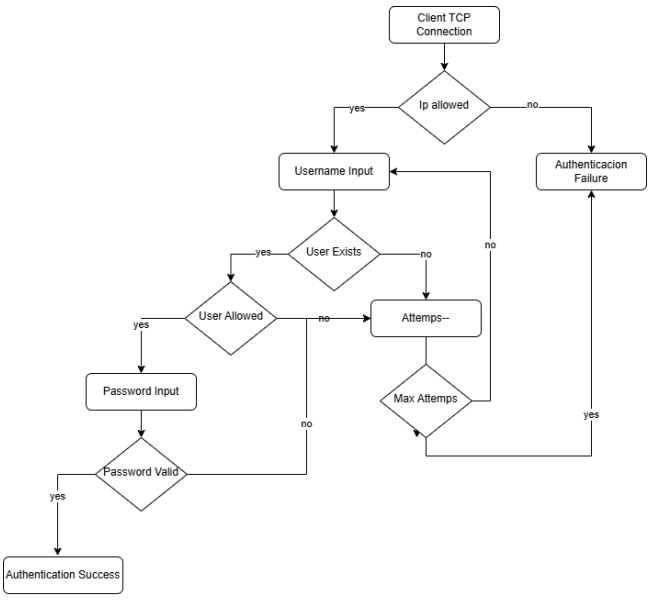

Fig. 2. User authentication flow.

Figure 2 details the system authentication flow. Once the TCP connection is established, the server validates that the client's IP matches the allowed IP defined in the config.conf file. It then requests the username and verifies that it is registered in the users.db database.

If the user exists, it checks whether they are authorized in the list of allowed users in the configuration file. If so, it

requests the password, which is verified by applying the SHA-256 hash algorithm. In case of error, the failed attempt is counted until the limit defined in INTENTOS_MAX is reached.

Once authenticated, the client can execute commands that are received and processed by ExecuteCommand() on the server. The output of each command is sent back to the client through the same TCP channel.

IV. Design and Implementation

A. Programming Language and Choice of Go

The programming language used to develop this remote shell is Go (Golang). Its choice is based on several key advantages:

- Native concurrency through goroutines, which allows multiple connections and processes to be handled efficiently in parallel.
- Powerful standard libraries for networking, files, cryptography, and operating system command execution.
- Cross-compilation, which facilitates the generation of binaries for multiple platforms, in this case, Windows (client) and Linux (server) systems.
- Simplicity and robustness in error handling and data structures, ideal for developing secure and reliable network services.

The system architecture benefits from Go's concurrency model, allowing for the effective separation of authentication tasks, remote command execution, and periodic resource monitoring using multiple goroutines.

B. Project Structure

The system is organized in a modular fashion in the REMOTESHELL/ folder, which contains two main subdirectories: client/ and server.

```
REMOTESHELL/
├── client/
│   ├── main.go          # Entry point and parameter handling
│   ├── connection.go    # Connecting to the server and getting the local IP address
│   ├── commands.go      # Running commands with a custom prompt
│   ├── login.go         # Handling user authentication
│   └── report.go        # Reporting
├── server/
│   ├── main.go          # Entry point
│   ├── auth.go          # Handling user authentication
│   ├── commands.go      # Running commands and custom monitoring
│   ├── connection.go    # Managing TCP connections
│   ├── config.go        # Reading configuration
│   ├── reporter.go      # Generating resource reports
│   ├── config.conf      # Configuration file
│   └── users.db         # User database
└── go.mod               # Project dependencies
```

Fig. 3. Structure of the cross-platform remote shell project.

Figure 3 illustrates the hierarchical organization of the modules, including authentication, communication via TCP sockets, command execution, and resource monitoring. This structure ensures a clear separation of responsibilities between the client and the server, which improves the maintainability and scalability of the system.

C. Configuration Files

The Linux server obtains its configuration parameters from a flat file called config.conf. This file contains:

```
IP_CLIENT=192.168.1.36
PORT=1625
MAX_ATTEMPTS=3
USERS=felipe,juanico,maria
```

The users.db file contains valid users and their passwords in SHA-256 hash format, such as:

```
felipe:caf90169eef...
maria:caf90169eef…
juanico:e16fef4138...
```

This guarantees a basic layer of secure authentication without the need for a relational database.

D. Authentication with SHA-256

Authentication is performed locally by comparing SHA-256 hashes. Each time a client connects, it is validated that the IP matches the allowed one, that the user is authorized according to the .conf file, and that the plain text password, when converted to SHA-256, matches the one registered in users.db.
This process is implemented in the auth.go file, using the crypto/sha256 and encoding/hex libraries of the Go language.

E. Remote Command Execution

The client can send Unix-style commands to the server. These commands are executed on the Linux server using the instruction:

```
exec.Command("/bin/bash", "-c", command)
```

If the command is cd, the current execution directory is changed. Valid commands are executed and their output is sent back to the client, which prints it on the screen. The "bye" command is used as a disconnection signal. All this logic is implemented in commands.go on both the client and the server.

F. Periodic Resource Report (CPU, RAM, Disk, and Processes)

The server automatically generates a system status report every n seconds, where n is sent by the client when the connection is established. This value defines the frequency of the generateSystemReport function.

The data is obtained by executing standard system commands such as vmstat, free, df, and ps, which are processed to generate a message with the percentages of CPU usage, RAM, disk space, and number of active processes.

Thanks to the use of goroutines, the server can maintain multiple concurrent tasks: one to handle interactive client commands, and another to send periodic reports without blocking the first.

The client, in turn, displays the reports received in parallel to the command prompt, improving the interactive experience.

V. Use Cases and Testing

A. Typical Usage Scenario

The typical operation of the REMOTESHELL client-server system begins with the execution of the client on a Windows system, using the syntax: ./client <server_IP> <Port> <ReportPeriod>.For example: ./client 10.1.3.3 2306 5

This execution establishes a TCP connection with a server running on a Linux machine. When the connection is established, the server requests authentication. The user enters their credentials, which are validated using SHA-256 hashing against a users.db file.

Once successfully authenticated, a bidirectional channel is established:

- The client can send UNIX commands (e.g., ls, pwd, mkdir, cd, rm -r) that the server executes, returning the results to the client.
- The server periodically sends reports on system resource consumption, including CPU, memory, disk, and active processes, at a frequency defined by the input parameter.

The session is terminated by entering the special command bye, which closes the connection at both ends.

B. Test Cases Performed

1. Connection and Authentication: It was verified that only connections from authorized IP addresses are allowed and that the authentication mechanism is functional. The scenarios evaluated were:
    a. Unauthorized IP → response: IP_ERROR
    b. Non-existent user → USER_NOT_FOUND
    c. User denied by configuration → USER_NOT_ALLOWED
    d. Incorrect password → PASSWORD_ERROR
    e. Excessive failed attempts → MAX_ATTEMPTS
2. Remote Command Execution: The correct processing of allowed commands and the special shutdown command was validated:
    a. Commands tested: ls, pwd, mkdir, cd, rm -r
    b. Special bye command: closed the session correctly
3. Report Generation: It was verified that the server sends reports with the following information:
    a. CPU usage (percentage).
    b. Memory usage (percentage) and status: total and free.
    c. Disk usage (percentage) and status: total and free.
    d. Total active processes.

Verified that the client receives and displays reports without interfering with command execution.

C. Differences between Client and Server Modules

The system is divided into two modules with different responsibilities:

- The server runs exclusively on Unix-based systems due to its dependence on Bash commands.
- The client operates on Windows systems, limited to connection, authentication, command sending, and result display.
- Local command execution is not allowed on the client.
- Differences in paths, permissions, and command behavior may vary depending on the Linux distribution used on the server.

D. Test Conclusion

The results obtained demonstrate that the REMOTESHELL system meets the requirements for functionality, security, and performance. The client-server architecture efficiently distributes responsibilities and allows remote command execution in a controlled and secure manner. The system proved to be resistant to invalid inputs and authentication errors, ensuring stability during the tests performed.

VI. Evaluation and Results

A. Metrics Evaluated

In this project, the following metrics were primarily evaluated to validate the system's performance:

- Stability: The server was verified to maintain a stable and operational connection during extended sessions, without unexpected shutdowns or failures in authentication or remote command execution.
- Concurrency: Since the server is designed to handle a single connection at a time, no evaluation was performed for multiple simultaneous clients. The system guarantees the correct handling and execution of commands for a single connected client.

B. Advantages and Limitations Found

Advantages:

- Simplicity and robustness: Serving a single connection at a time allows for a simpler design, with less complexity in resource management and synchronization.
- Basic security: Authentication using SHA-256 in flat files ensures that passwords are not transmitted in plain text.
- Multiplatform: The system works on Linux (server) and Windows (client), allowing flexibility in the monitoring environment.
- Configurable reporting: The interval parameter for sending reports is dynamic and configurable from the client.

Limitations:

- Single client support: The server does not support concurrency, which limits its use to a single client

at a time and reduces its scalability in multi-user environments.

- Basic authentication: The absence of additional mechanisms such as salting in hashes or advanced authentication protocols may expose vulnerabilities to sophisticated attacks.
- Local network dependency: Performance tested on a local network may vary on public networks with higher latency.
- Simple interface: The remote shell only supports basic UNIX commands without advanced validation or restrictions.

C. *Examples of Generated Reports*

Example of resource consumption report sent every 5 seconds:

Resource Report
CPU: 12.5%
Memory: 42.5% (3000MB / 8000MB)
Disk: 46.9% (12 GB / 50 GB)
Active Processes: 112
Hour: 2025-05-01 19:05:36

This report is sent periodically according to the configured interval.

D. *Performance Evaluation with Different Values of n*

The impact of the reporting interval (n seconds) on server resource consumption was evaluated with values n = 1, 5, 10, and 30 seconds. Average CPU and memory usage during the active session was measured, as shown in Table 1.

TABLE I
AVERAGE CPU AND MEMORY USAGE BY REPORTING INTERVAL

Interval (s)	Average CPU usage (%)	Average memory usage (%)
1	10.6	80.2
5	1.0	77.5
10	0.9	75.4
30	0.8	73.8

It has been observed that the frequency of report submissions directly impacts CPU and memory usage. Intervals of 5 seconds or longer are recommended to achieve an adequate balance between updates and resource consumption.

VII. CONCLUSIONS

The development of this cross-platform remote shell represented a comprehensive exercise in applying key concepts from the Operating Systems course, including concurrency, interprocess communication, sockets, and authentication mechanisms. The choice of the Go language was crucial for efficiently managing concurrency through goroutines, allowing multiple clients to interact simultaneously without interference, while maintaining a modular and scalable architecture.

From a technical perspective, this project consolidated knowledge of the client-server model and reinforced practical skills in the use of TCP sockets, configuration file management, concurrent command reading, and real-time system metrics reporting. It also allowed for experimentation with secure authentication mechanisms based on SHA-256 hashing, essential for ensuring a minimum layer of security in network communications.

The implemented system satisfactorily meets the established functional requirements: remote client-server connection between Windows and Linux, user authentication via flat file, remote command execution, and periodic system monitoring. Communication between client and server is non-blocking thanks to concurrency, and reports are received at the intervals established by the user.

During development, multiple challenges arose, including interoperability between platforms (Windows and Linux), error management in concurrent environments, and the correct implementation of password hashing. Difficulties were also encountered in synchronized socket management in parallel channels (commands vs. reports), which required careful structuring of the execution flow.

Future improvements include the addition of end-to-end encryption to protect the entire communication channel using TLS, a graphical interface to facilitate user interaction with the system, and a logging system to record executed commands for both auditing and subsequent analysis. These extensions would strengthen the system's security, accessibility, and traceability.

REFERENCES

[1] The Go Authors, “The Go Programming Language Documentation,” [Online]. Available: https://golang.org/doc/

[2] M. Butcher, *Concurrency in Go: Tools and Techniques for Developers*, O’Reilly Media, 2017.

[3] T. Nguyen, “Golang TCP Server and Client Example [Tutorial],” *GoLinuxCloud*, Nov. 2, 2022. [Online]. Available: https://www.golinuxcloud.com/golang-tcp-server-client/

[4] Linode Docs, “Create a TCP and UDP Client and Server using Go,” *Linode Documentation*. [Online]. Available: https://www.linode.com/docs/guides/developing-udp-and-tcp-clients-and-servers-in-go/

[5] "Capítulo 1. Tutoriales de GNU/Linux," *Debian Reference*. [Online]. Available: https://www.debian.org/doc/manuals/debian-reference/ch01.es.html