Remote Shell using RPC

Group 6
Do Cong Tuan Hung
To Thanh Hai
Trinh Duc Anh
Nguyen Dung
Nguyen Dinh Hai

Contents

Chapter 1

Introduction

Remote Procedure Call (RPC) enables a program to execute a procedure on another machine, abstracting the complexities of the underlying communication. Distributed systems often face challenges related to network communication, security, and scalability. By implementing a secure, multi-client remote shell using RPC principles, this project demonstrates a practical solution for remote command execution.

Remote shells are critical for managing distributed environments, allowing administrators to interact with remote systems as if they were local. With secure communication protocols in place, this project addresses potential vulnerabilities such as unauthorized access and data interception.

Chapter 2

Background

2.1 What is RPC?

RPC is a protocol that allows the execution of code on a remote server as if it were local, abstracting away the details of network communication. It simplifies distributed computing by providing seamless communication between systems.

RPC emerged as a key enabler for distributed systems, providing developers with a higher level abstraction over socket-based communication. The mechanism works by bundling and unraveling data, allowing complex objects to be passed between systems effortlessly.

2.2 Remote Shell

A remote shell provides users with the ability to execute commands on a remote system, similar to local command-line interfaces. These tools are often used for system administration, automation tasks, and debugging distributed systems.

Remote shells have been a cornerstone of network communication since the advent of early protocols like Telnet, which enabled users to execute commands on remote systems as if they were physically present. Over time, these technologies evolved, giving rise to more robust and reliable methods for interacting with remote machines. This project builds upon these foundations, embracing contemporary approaches by integrating the Remote Procedure Call (RPC) model to enable seamless and efficient communication between clients and servers. The design reflects the progression from simplistic connections to sophisticated, structured interactions, ensuring flexibility and scalability in executing remote commands.

Chapter 3

System Design

3.1 Architecture

The system consists of two primary components:

- **Server**: Hosts the remote shell service, accepting client connections, executing commands, and returning results.
- **Client**: Connects to the server, sends commands, and receives the output.

The server employs a multi-threaded architecture, expertly designed to handle multiple client connections simultaneously. Each client session functions autonomously, ensuring stability while maintaining flexibility. To bolster security, our implementation introduces a tailored authentication mechanism that verifies client credentials, providing a foundational layer of protection. This approach ensures that communication between the server and its clients remains structured and reliable, offering a blend of efficiency and enhanced data safety.

3.2 Features

- **Multi-client Support**: The server can handle multiple client connections simultaneously, enabling scalability in distributed environments.
- **Secure Communication**: Custom-made authentication system that provides seamless but potent protection from external influences.
- **Session Management**: Each client is assigned a unique session ID, which is used for identification and tracking purposes.

- **Command Execution**: Clients can execute shell commands remotely, with the server processing and returning the results in real-time.

Chapter 4

Functionalities

4.1 Secure Communication

Initially, the security of this project was based on the use of Secure Socket Layers (SSL) and certificate-based encryption. Each client and server would have their own keystore to store certificate keys and a truststore to authenticate trusted keys. While this approach provided robust security, it significantly impacted the user experience. The requirement for both the client and server to exchange and store each other's certificate keys in their truststores before establishing a connection created unnecessary complexity. As a result, we decided to adopt a simpler, more streamlined authentication mechanism, relying on basic password matching.

The custom-built authentication mechanism serves as a protective barrier between the client, server, and external threats, guarding against unauthorized access. Designed with user experience in mind, it features a simple, intuitive structure that ensures seamless integration. While providing essential protection, its streamlined design maintains ease of use, balancing security with a smooth user experience.

4.2 Session Management

Session management plays a critical role in ensuring that client connections are handled in an organized and traceable manner. In this project, each client connection is assigned a unique session ID, which is generated using a timestamp. This approach ensures that each session is distinct, preventing conflicts and enabling accurate tracking of client activities. By using timestamps to create session IDs, we guarantee uniqueness and provide a reliable method for monitoring client interactions over time.

The server logs session initiation, activity, and termination events, providing a robust system for auditing and debugging. These logs are valuable for administrators, as they allow them to track when a client connects, what commands they execute, and when they disconnect. This data can be crucial for detecting issues, analyzing system performance, and identifying potential security breaches. Additionally, in case of any errors or unusual behavior, session logs can serve as a vital resource for troubleshooting and system diagnostics.

Session management not only helps organize client interactions but also adds a layer of security. By assigning unique session IDs, the server can ensure that only authorized clients are able to send commands within their own session, thus preventing potential unauthorized access. This organization enhances both the reliability and security of the system, making it easier for administrators to maintain control and monitor activities in a distributed environment.

Furthermore, session management provides flexibility for future expansions. For instance, additional metadata such as client IP addresses, session start times, and execution histories could be stored alongside session IDs to offer more detailed insights and improve overall system monitoring.

4.3 Command Execution

The command execution functionality allows the server to receive and process shell commands from the client using the Java Runtime API, which spawns new processes on the server's operating system. The server captures both the standard output and error streams in real-time, forwarding them back to the client as they are generated. This ensures immediate feedback, simulating a local terminal experience and enabling clients to execute a wide range of system administration tasks, such as querying system resources or managing processes remotely.

This feature is integral for efficient remote management, as it enables clients to interact with the server's underlying OS securely and seamlessly. The system supports diverse shell commands, providing flexibility for various use cases, from debugging to automation. Additionally, error handling is built into the process, where the server captures any command execution failures and transmits detailed error messages back to the client, ensuring transparency and facilitating troubleshooting.

4.4 Multi-client Handling

The server uses Java's threading mechanism to manage multiple client connections concurrently, with each client being handled by a dedicated thread. This allows requests to be processed independently, ensuring that the actions of one client do not affect others. By isolating client interactions, the server can scale effectively to accommodate many simultaneous connections without performance issues.

This multi-threaded design helps prevent bottlenecks, ensuring that the server remains responsive even under heavy load. Each thread operates independently, processing client commands and delivering responses promptly, which results in a smooth user experience. This approach ensures that the system can handle numerous client interactions efficiently, without sacrificing performance or responsiveness.

Chapter 5

Implementation

5.1 Server-side

The server-side implementation of the remote shell using RPC is designed to handle multiple client connections concurrently. The server listens for incoming connections on port 18463 and uses a thread pool to manage up to 10 client connections simultaneously. Upon receiving a connection from a client, the server authenticates the user by requesting a password. If the password matches the predefined value, the client is granted access to send shell commands for execution. The server then spawns a new process for each command received and streams the command output and errors back to the client. The server also tracks client connections and logs client activity, providing a simple mechanism to handle multiple users with basic command execution capabilities.

5.2 Client-side

The client-side implementation establishes a connection to the server using a socket and sends the password for authentication. Upon receiving a prompt from the server, the client provides the password, and if authentication is successful, the client is granted permission to send shell commands. The client reads the server's responses to the commands, displaying both the output and any errors. The user can continue sending commands until the user types 'exit', which gracefully terminates the connection. The client-side program ensures secure communication by handling potential exceptions and error messages, offering a user-friendly interface for interacting with the server remotely.

Chapter 6

Use Cases

6.1 System Administration

System administrators can securely execute remote commands for a wide range of monitoring and maintenance tasks. This includes checking disk usage, monitoring active processes, managing system configurations, and deploying scripts across multiple machines. The ability to interact with remote systems as if they were local allows for more efficient system management and quick response to issues, even in large, distributed environments.

6.2 Educational Purposes

This project serves as an excellent case study for understanding the design and implementation of distributed systems with secure communication. It provides students with hands-on experience in creating and securing remote interactions, offering practical insights into key concepts such as client-server architecture, multi-threading, and secure communication protocols. Students can explore the challenges of building scalable, reliable, and secure distributed systems in a real-world context.

6.3 Enterprise Applications

In enterprise environments, this system can be extended to provide centralized management of server clusters. IT teams can execute commands or scripts across multiple machines simultaneously, streamlining administrative tasks such as software updates, configuration changes, and troubleshooting. This centralized approach enhances operational efficiency, reduces the need

for manual intervention, and ensures consistent management across large-scale infrastructure.

Chapter 7

Credits

This project was developed by **Group 6**.
The work distribution is listed below:

- Do Cong Tuan Hung : Governance of the project, Strategic supervisor, Report orchestrator
- To Thanh Hai : Security architect
- Trinh Duc Anh : Client-server Infrastructure Developer
- Nguyen Dung : User experience craftsman
- Nguyen Dinh Hai : Client ecosystem management

End