MyFileTransferProtocol

Andrei Cristian-George

Faculty of Computer Science Iasi
https://www.info.uaic.ro/

Keywords: File Transfer · Networking · C/C++

1 Introduction

The MyFileTransferProtocol project is designed to facilitate file transfer between clients and a server over a TCP connection. The server manages user authentication, authorization, and executes various file operations based on client requests. The communication is structured using a custom protocol involving message headers and payloads. The implementation also incorporates a logging mechanism, SQLite for database management, password hashing for enhanced security.

2 Technologies Applied

The MyFileTransferProtocol project incorporates several technologies to achieve its goals. These technologies have been carefully selected based on their suitability for network communication, database management and security.

The technologies employed in the project include the C++ programming language, TCP communication protocol, SQLite database, custom Logger for logging, custom Hash Function for password hashing and POSIX Threads (pthread) for concurrent server handling.

To ensure efficient handling of client-server communication, the server implementation utilizes POSIX Threads for concurrent processing. The server operates with two main threads: one dedicated to handling administrative inputs, and the other serving as the client manager. The client manager thread dynamically spawns a new thread for each connected client, allowing simultaneous handling of multiple client inputs and making the MyFileTransferProtocol a concurrent server. This approach enhances system responsiveness and scalability, ensuring optimal performance in scenarios with multiple concurrent client connections.

These technologies collectively form the foundation of the MyFileTransferProtocol, addressing the complexities associated with secure file transfer, user authentication, and system robustness.

3 Structure of the Application

The MyFileTransferProtocol application is designed with a clear and modular structure, incorporating various concepts to ensure efficient and secure file transfer. This section explains the key concepts used in the modeling and presents a detailed diagram of the application's architecture.

3.1 Client-Server Architecture

The application follows a classic client-server architecture. Clients initiate communication with the server to perform file operations and request information. The server manages user authentication, authorization, and executes file operations on behalf of clients. This architecture allows multiple clients to interact with the server concurrently.

3.2 Concurrent Server

To enhance system responsiveness and scalability, the server adopts a concurrent design using POSIX Threads (pthread). The server's main thread handles administrative commands, while a separate thread is dynamically spawned for each connected client. This approach ensures that multiple clients can be served simultaneously without blocking the server's ability to handle new connections.

3.3 Message Header and Payload

Communication between the client and server is structured using a message header and payload. The message header contains information about the type of message, content size, and the requested path. This structured approach ensures that both the client and server can interpret incoming messages correctly.

3.4 Custom Hash Function for Passwords

The application employs a custom hash function for password hashing during the login process. This function adds a layer of security by transforming plaintext passwords into hashed values before transmission. The hash function involves an iterative process based on the ASCII values of the characters in the password.

3.5 SQLite Database

The application utilizes the SQLite database to manage user information. Usernames, hashed passwords, and authorization status are stored in the database. This provides a reliable and secure means of storing user credentials and allows for efficient user management.

3.6 Logging Mechanism

A custom Logger class is implemented to log significant events within the application. This includes user login attempts, file operations, and server-related activities. The logging mechanism aids in system monitoring, issue troubleshooting, and security auditing.

3.7 Command-Based Interaction

The interaction between clients and the server is command-based. Clients issue commands to perform file operations or request information, and the server responds accordingly. Both clients and the server have a set of available commands, each serving a specific purpose.

3.8 Application Architecture Diagram

The following diagram illustrates the architecture of the MyFileTransferProtocol application:

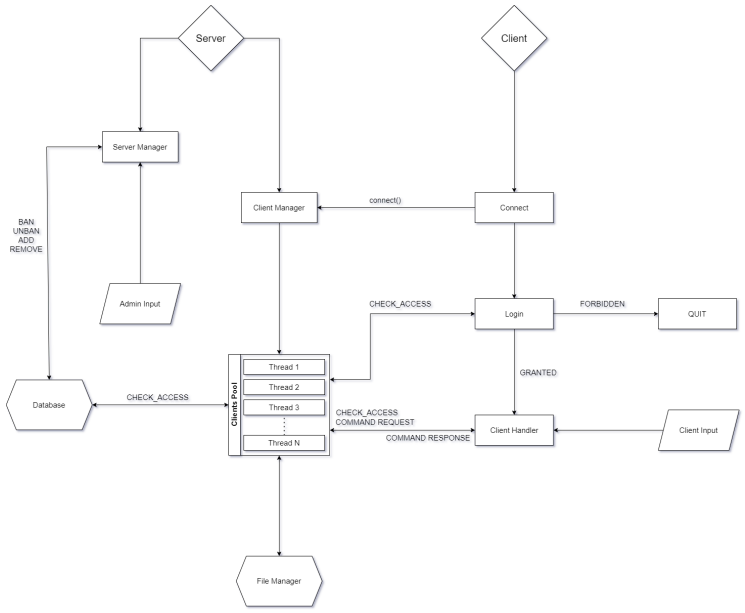

Fig. 1. MyFileTransferProtocol Application Architecture Diagram

4 Implementation Aspects

4.1 Message Header Structure

The communication protocol between the client and server relies on a well-defined message header structure. This structure, referred to as `msg_header`, encapsulates essential information for interpreting messages. Here is the definition of the `msg_header` structure:

```
struct msg_header {
    enum types type = types::NONE;
    size_t content_size = 0;
    char username[MAX_USERNAME_SIZE]{0};
    char path[MAX_LOCATION_SIZE]{0};
};
```

The `msg_header` structure includes fields such as `type` to denote the message type, `content_size` specifying the size of the payload, and `username` and `path` providing additional context for the message. This structured approach ensures consistent communication between the client and server.

4.2 Macros for Error Handling

To enhance code readability and streamline error handling, several macros have been introduced. A notable one include:

```
#define HANDLE(f) \
    if ((f) < 0) { \
        printf("FILE: %s\n", __FILE__); \
        perror(#f); \
        exit(1); \
    }
```

The `HANDLE` macro simplifies the handling of file-related operations, printing the filename and an error message in case of failure. The `NULLCHECK` macro performs a similar function for checking the validity of pointers.

4.3 Login Header Structure

During the login process, a specialized structure called `login_header` is employed to convey user credentials securely:

```
struct login_header {
    enum loginTypes type;
    char username[MAX_USERNAME_SIZE]{0};
    char password[MAX_PASSWORD_SIZE]{0};
};
```

This structure contains fields for the login type, username, and password. The `login_header` structure plays a crucial role in authenticating users securely.

4.4 FileManager Class

The `FileManager` class encapsulates file-related operations and manages the current state of the file system. Key functionalities and methods of the class include:

```
class FileManager {
    string currentPath;
    int remote;
    Logger* logger;
    pthread_t tid;
    pthread_mutex_t mutex;
public:
    // ... (other functions)

    string ls(const string& path);
    bool cd(const string& path);
    string statFile(const string& path);
    bool makeDirectory(const string& path);
    bool transfer(const string& local, const string& remote, const types type);
    bool acceptTransfer(const msg_header header);
};
```

The `FileManager` class facilitates operations such as listing directory contents (`ls`), changing the current directory (`cd`), obtaining file information (`statFile`), creating directories (`makeDirectory`), and handling file transfers (`transfer` and `acceptTransfer`). It plays a pivotal role in orchestrating file-related interactions between the client and server.

4.5 Usage Scenario

To illustrate the practical application of the implemented features, let's consider a typical usage scenario. Suppose a client wishes to transfer a file to the server. The client initiates the transfer by sending a message with the appropriate `msg_header` and payload. The server, utilizing the `FileManager` class, receives the request, validates it, and performs the file transfer operation. The `acceptTransfer` method is responsible for handling incoming transfer requests, ensuring a smooth and secure data exchange.

5 Conclusions

Currently, there are many aspects of the MyFileTransferProtocol application that demonstrate its functionality and effectiveness. However, as with any software project, there is room for improvement and future development. Some areas that could be enhanced include:

- **Enhanced Security Measures:** The current implementation relies on a custom hash function for password hashing, which provides a basic level of security. However, considering the evolving landscape of cybersecurity, the application could benefit from the implementation of more advanced security measures. One potential enhancement is the incorporation of RSA for packet encryption, adding an extra layer of protection to the communication between clients and the server. This would ensure that sensitive information is transmitted securely.
- **Graphical User Interface (GUI):** The current version of the MyFileTransferProtocol is primarily command-line based. Introducing a graphical user interface could significantly improve the user experience, making the application more accessible to users who may not be familiar with command-line interfaces. A well-designed GUI can provide a user-friendly environment, allowing users to interact with the application more intuitively.
- **Logging and Auditing Improvements:** While the current logging mechanism serves its purpose, further enhancements could be made to provide more detailed and insightful logs. Additionally, implementing an auditing system could help in monitoring and analyzing security-related events, providing administrators with a comprehensive overview of system activities.
- **Error Handling and Resilience:** Strengthening error-handling mechanisms and introducing fault tolerance features can contribute to a more robust and resilient system. This includes handling unexpected scenarios gracefully, providing meaningful error messages, and implementing recovery mechanisms to ensure the application can handle failures gracefully.
- **Optimization for Large-Scale Deployments:** To scale the application for larger deployments, optimizations in terms of performance and resource utilization may be necessary. This could involve refining algorithms, minimizing resource consumption, and conducting thorough performance testing under various conditions.
- **Cross-Platform Compatibility:** Currently, the application may be designed for a specific operating system. Achieving cross-platform compatibility would broaden its reach and allow users to run the MyFileTransferProtocol on different operating systems without modification.

These potential improvements represent avenues for future development, ensuring that the MyFileTransferProtocol continues to meet the evolving needs and expectations of its users.

References

1. OpenSSH. (n.d.). scp - secure copy protocol. Retrieved from `https://en.wikipedia.org/wiki/Secure_copy_protocol`
2. File Transfer Protocol. Retrieved from `https://ro.wikipedia.org/wiki/File_Transfer_Protocol`
3. Bernstein, D. J. (1994). djb2: Hash function. Retrieved from `http://www.cse.yorku.ca/~oz/hash.html`
4. SQLite. (n.d.). SQLite Library. Retrieved from `https://www.sqlite.org/`