

Systems Software Report CA2

TU858

BSc in Computer Science

**Armando Cardozo**

**C19739031**

School of Computer Science

TU Dublin – City Campus

**20 August 2023**

**Table of Contents**

Table of Contents

[*Functionality Checklist* 3](#_Toc129253098)

[*Feature 1 - Client Program* 4](#_Toc129253099)

[*Feature 2 – Server Program* 4](#_Toc129253100)

[*Feature 3 – Multithreaded connections* 4](#_Toc129253101)

[*Feature 4 – File Transfer* 5](#_Toc129253102)

[*Feature 5 - Transfer Authentication* 5](#_Toc129253103)

[*Feature 6 – Synchronisation (Mutex Locks)* 5](#_Toc129253104)

[*Conclusion* 5](#_Toc129253105)

# *Functionality Checklist*

|  |  |  |
| --- | --- | --- |
| ***Feature*** | ***Description*** | ***Implemented*** |
| F1 | Client Program | Yes |
| F2 | Server Program | Yes |
| F3 | Multithreaded Connections | Yes |
| F4 | File Transfer | Yes |
| F5 | Transfer Functionality | Yes |
| F6 | Transfer Authentication | Yes |
| F7 | Synchronisation | Yes |

Have you included a video demo as part of the assignment: No

Link to Video: will update. Tomorrow.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

\_\_\_\_Armando Cardozo \_\_\_\_\_

Armando Cardozo

20/08/2023

# *Feature 1 - Server Program*

In this program, I've implemented a server that facilitates secure file transfers and authorization of clients. The server listens on a specified port, and upon receiving a connection request from a client, it authorizes the client based on provided credentials and subsequently handles the file transfer.

Upon accepting a client connection, the server creates a new thread to handle the client's requests concurrently, ensuring efficient multitasking. This is because the system needs to handled several users updating files at the same time if necessary.

Each users can access only specific directories and files, for this clients are required to provide authorization data, the UIDand GID are provided by the program itself identificating each user running it , they have to enter the file path they intend to access. The server parses this data, verifies the UID and GID against the directory's permissions, and authorizes the client accordingly. If the UID or GID does not match the directory's permissions, authorization fails, and the server sends an appropriate error message to the client.

Once authorized, the server proceeds with the file transfer. It opens the specified file for writing and receives the data sent by the client. The received data is then written to the file. This ensures that the server can securely receive and save files from clients.

Throughout the process, the server maintains communication with the client, sending messages to update on the progress of the transfer or to report any issues encountered. It also employs mutex locks to synchronize access to critical sections, ensuring thread safety in handling multiple clients.

The server's core logic is encapsulated within the handle\_client function, which manages client authorization and file transfer. It communicates with the client using socket programming.

To enhance maintainability and code readability, the program uses meaningful log messages with different logging levels, such as LOG\_INFO and LOG\_ERR. These log messages provide insights into the server's activities and errors, making it easier to diagnose and troubleshoot issues.

# *Feature 2 – Client Program*

In this Client program, I've created a client that facilitates secure file transfers and authorization with a server. The client initiates a connection to the server, sends authorization data, and transfers files.

I start by obtaining the user's UID and GID, which are essential for authorization. These values are then converted to strings for later use. The program also defines variables for socket communication, file handling, and message storage.

The user is prompted to enter the file path and filename they wish to send to the server. This input is stored in the filepath variable.

Next, I create a socket using the socket() function, specifying the address family, socket type, and protocol. The program checks if the socket creation was successful and prints a corresponding message.

I set the server's address (IP and port) to connect to. The IP address "127.0.0.1" is used for local testing, but it can be changed to the server's IP. The program then initiates a connection request to the server using connect(). If successful, a connection message is printed.

To ensure proper authorization, the program constructs an Authorization string containing the UID, GID, and the entered filepath. This string is sent to the server using the send() function, and any errors in sending are handled.

The client receives an authorization message from the server and checks whether it's authorized to send the file. If authorized ("yes"), the client proceeds with the file transfer; otherwise, it exits with a failure message.

The program opens the specified file for reading and sends the filepath as metadata to the server. It then opens the file for binary reading (rb) and begins sending the file data in chunks. Errors during file reading and data sending are handled and reported.

Once the file is sent, a success message is printed along with any status message received from the server. The client closes the file and socket connections before exiting.

In summary, this C client program enables users to securely send files to a server after authorization. It handles socket communication, file handling, and authorization checks, making it a useful component for secure file transfer applications.

*Feature 3 – Multithreaded connections*

In my server.c program, I've implemented a multithreaded approach to handle concurrent client connections. This multithreaded design significantly enhances the server's functionality by allowing it to process multiple client requests simultaneously.

The main function serves as the entry point for managing client connections. When a client connects to the server, a new thread is created using pthread\_create to execute the handle\_client function. This function handles authorization, file transfer, and communication with the connected client. The use of threads ensures that each client connection is managed independently, preventing one client from blocking the server's ability to handle other clients.

The pthread\_mutex\_t lock is used to protect the critical section of the code, particularly the file operations involving reading and writing data. When a thread enters the critical section, it locks the mutex using pthread\_mutex\_lock(&lock), ensuring exclusive access to the shared resource. This prevents multiple threads from simultaneously accessing and modifying the same file, which could lead to data corruption.

Once a thread completes its operations within the critical section, it unlocks the mutex using pthread\_mutex\_unlock(&lock). This action allows other threads to enter the critical section and perform their tasks, ensuring safe concurrent execution.

To ensure proper thread synchronization and resource cleanup, I utilize pthread\_join within the main function. This function waits for each thread to complete its execution. By doing so, it guarantees that all threads finish their tasks before the server exits. This is crucial for maintaining the server's stability and preventing any resource leaks.

Additionally, the use of pthread\_exit within the handle\_client function allows threads to exit gracefully once their tasks are completed. This ensures that system resources allocated to each thread are released efficiently, preventing memory leaks and maintaining the overall server's robustness.

In summary, the multithreaded design of the server enables it to efficiently manage multiple client connections concurrently. The combination of pthread\_create, pthread\_join, and pthread\_exit ensures proper thread management, synchronization, and resource cleanup, resulting in a highly responsive and stable server capable of handling concurrent file transfers and client requests effectively.

Top of Form

Bottom of Form

*Feature 4 – File Transfer*

In the file transfer implementation of my client.c and server.c programs, I've established a mechanism for sending and receiving files. This mechanism ensures the reliable exchange of data between clients and the server.

Client-Side File Transfer: In the client code, I initiate the file transfer by opening the specified file for reading ("r"). If the file cannot be opened, an error message is displayed, and the program exits. Once the file is successfully opened, I read it in chunks (buffered reads) of 1024 bytes or less and send these chunks to the server using the send() function. This iterative process continues until the entire file has been read and sent. If any issues occur during data transmission, such as a failed send() operation, an error message is displayed, and the program exits. Finally, I close the file and the socket to release resources.

Server-Side File Transfer: On the server side, I first create or open a file for writing received data using the specified file path. This file is opened in binary append mode ("ab"), ensuring that the received data is appended to the file if it already exists or created as a new file if it doesn't. If the file cannot be created or opened, an error message is sent back to the client, indicating that the transfer has failed.

Once the file is successfully opened, I proceed to receive data from the client. The server uses the recv() function to receive data in chunks, with a maximum buffer size of 1024 bytes. These received chunks of data are then written to the file using the fwrite() function. This process continues until all data sent by the client has been received and written to the file.

After the file transfer is completed, I close the file to ensure that all data is flushed and saved. Additionally, thesocket used for communication with the client is closed to release associated resources.

Overall, this file transfer implementation in both the client and server programs ensures that files are transmitted reliably and efficiently. It accounts for potential errors during transmission and provides appropriate error handling, allowing for a smooth and secure exchange of files between clients and the server.

# *Feature 5 - Transfer Authentication*

**On Client Side (client.c):** I began by identifying the user and group associated with the client process using getuid() and getgid(). These identifiers played a crucial role in our authentication process. Then, I configured a socket to establish a connection with the server, ensuring that we were connecting to the right destination.

To authenticate with the server, I implemented an authorization message containing the user's UID, GID, and the requested file's path and name. This message was sent to the server for verification. Upon receiving an authorization response from the server, the client would decide whether it was authorized to proceed with file transfer. If the server's response was "yes," signifying authorization, we initiated the file transfer. However, if the response was negative, indicating unauthorized access, the client terminated the process promptly.

**On Server Side (server.c):** On the server, I parsed the authorization message received from the client, which contained the UID, GID, and the requested file's information. I then converted these received values from string format to integers and proceeded to validate them. To validate authorization, I compared the received UID and GID with those associated with the directory containing the requested file. If there was a mismatch, it was considered an authorization failure.

Based on the results of this authentication, the server responded accordingly. If the UID and GID matched the directory, indicating successful authorization, the server sent an "authorization granted" response ("yes") back to the client. If there was an authorization failure, an appropriate error message was dispatched.

This authentication process ensured that only clients with the proper credentials and permissions could send files to the server. It added an extra layer of security to our file transfer system, preventing unauthorized access to sensitive files.

# *Feature 6 – Synchronisation (Mutex Locks)*

The pthread\_mutex\_t lock is used to protect the critical section of the code, particularly the file operations involving reading and writing data. When a thread enters the critical section, it locks the mutex using pthread\_mutex\_lock(&lock), ensuring exclusive access to the shared resource. This prevents multiple threads from simultaneously accessing and modifying the same file, which could lead to data corruption.

Once a thread completes its operations within the critical section, it unlocks the mutex using pthread\_mutex\_unlock(&lock). This action allows other threads to enter the critical section and perform their tasks, ensuring safe concurrent execution.

# *Conclusion*

Summary of the implementation and achievement

In conclusion, I have implement most of the functionality require but the project, though a better result could have been achieve with more time and testing.

I established a secure authentication mechanism where clients were required to provide their user and group identifiers (UID and GID) along with the file path they intended to access.

I implemented multithreading in the server, enabling it to handle multiple client connections concurrently.

I ensured data integrity during file transfers by dividing them into smaller chunks, which were then transmitted and reassembled at the destination. This approach minimized data loss and optimized the file transfer process, even for large files.

I made the use of mutex locks to safeguard the critical sections of code, preventing data corruption that might occur when multiple threads access shared resources concurrently.

Overall, this implementation achieved the core objectives of secure and efficient file transfer in a client-server architecture. By focusing on user authentication, multithreading, and data integrity. This project has enhanced my understanding of sockets and multithread in C and has also provided a solid foundation for building secure and scalable communication system, although better results could be achieve with further research and better time management. It is definitely something to have in consideration for future projects like this.