

Systems Software Report CA2

TU856

BSc in Computer Science

**Aaron Fu Siang Ann**

**D19126659**

School of Computer Science

TU Dublin – City Campus

**8/12/2021**

**Table of Contents**

*Functionality Checklist* 3

*Feature 1 - Client Program* 4

*Feature 2 – Server Program* 4

*Feature 3 - Multithreaded connections* 4

*Feature 4 - File Transfer* 5

*Feature 5 - Transfer Authentication using Real and Effective ID’s* 5

*Feature 6 - Synchronisation (Mutex Locks)* 5

*Conclusion* 5

# *Functionality Checklist*

|  |  |  |
| --- | --- | --- |
| ***Feature*** | ***Description*** | ***Implemented*** |
| F1 | Client | Yes/No |
| F2 | Server | Yes/No |
| F3 | Multithreaded connections | Yes/No |
| F4 | File Transfer | Yes/No |
| F5 | Transfer Authentication using Real and Effective ID’s | Yes/No |
| F6 | Synchronisation (Mutex Locks) | Yes/No |

# *Feature 1 - Client Program*

The client program takes in 2 arguments in the command line, the filename and path to transfer the file to. Client communicates with server back and forth passing messages to ensure each message was sent successfully. First, the client gets the current user id and creates a socket. It attempts to connect to the server with the socket created and start sending messages. The first message **initTransfer** will be sent and waits for a reply from the server. If this process went smoothly, the client would receive a **user** message indicating the server is requesting the user id which the client retrieved earlier on. The client will then now send the user id to the server. It then resets the message received from the server to again prepare for another incoming message from the server. The **path** will be sent next, and the process repeats itself again. Finally, the server will start sending the requested file from the client folder. The client attempts to open the file if it exists, then reads the data from the file and sends it. After this, the client closes the socket and disconnects itself from the server.

# *Feature 2 – Server Program*

The server program first creates a socket when launched. It attempts to bind an address with the socket descriptor created. On successful bind, server will start listening for incoming connection. Any incoming connection from the client, the **accept** call will accept the connection. When a connection is established, the socket created is ready for use to read data from the process that requested the connection. A thread is created here within the process to allow multithreaded connections. In other words, multiple clients can attempt a file transfer at the same time. The thread is given a start routine called **connection\_handler** and given the client socket as argument. In the **connection\_handler**, the server receives the first message from the client **initTransfer**. If the message received successfully, it would send the message **user** back to the client by using **write** with the socket descriptor which is the client socket passed in as argument when creating thread. The server resets the message and waits for the next message from the client. The process goes back and forth similarly to the client program above, requesting **path** then the **file** to transfer. With the **user id** and **path** received from the client, the server swaps from root to the given **user id.** This is so that it attempts to write the file requested for transfer to the **path** specified with the current **user** permissions and not root permissions. Since the directories manufacturing and distribution should not be access by others except the department itself, this step is necessary. If the user has the right permissions, the file will be successfully transferred to the desired directory. The server then swaps back from the **user** to **root**. This process of swapping is done by **seteuid** which sets the effective user of the calling process. Lastly, **fflush** is called to clear the output buffer. Mutex is also implemented because server is a shared resource and it allows synchronisation.

# *Feature 3 - Multithreaded connections*

Concurrent connection is achieved with multithreading. When the server receives an incoming connection from the client and successfully creates a socket, it creates a thread for that specific client with the socket. In other words, this is a **one thread per client** server. However, issues such as synchronisation will arise with just multithreading. As a result, mutex is implemented to prevent this.

# *Feature 4 - File Transfer*

File transfer will begin when the server sends a **begin** message to the client. This indicates that previous messages were successful and ready to begin the transfer process. Client will search for the file specified in **client\_folder**. With a while loop, the bytes are read with **fread** until it reaches 0. Bytes are sent in blocks of size 512. The server then reads the blocks of size 512 bytes sent from the client. As stated above in the server program, the path is already constructed before the file transfer begins. The received blocks will be written to the file in the path opened by the server.

# *Feature 5 - Transfer Authentication using Real and Effective ID’s*

The folders are set to only allow specific group to perform a file transfer. For example, the manufacturing folder will only permit manufacturing group to transfer a file to the folder. To achieve this, the folder permission has to first be set to the allowed group. Described above, the server swaps to the user calling the file transfer to attempt this function. This process of swapping is by setting the effective user of the calling process. If user has the right permissions, the file transfer will be successful. It will then swap back to root after the attempt at file transfer. Unprivileged processes may only set the effective user ID to the real user ID, which is the root.

# *Feature 6 - Synchronisation (Mutex Locks)*

How synchronisation was achieved for the concurrent access to shared resources.

A mutex is initialized at the beginning of the server program. Before attempting the file transfer, the mutex will be locked and will execute the rest of the code which is the transfer before unlocking the mutex. The point of this is to synchronise threads. When there are multiple threads attempting to access the server (single resource), a race condition occurs. This is where mutex is useful, for instance, imagine a thread locking the mutex and executing the code. Then another thread attempts to access and execute the code. This second thread will be denied access because of the mutex being locked by the first thread. It sits and waits for the mutex to unlock before taking its turn to continue.

# *Conclusion*

In conclusion, the implementation is a simple client server program with the server as a one thread per client server. The transfer authentication was tricky due to not having a good understanding of the topic but nevertheless it was integrated after research. Overall the process of the implementation gave me a great understanding of threads, synchronisation, mutex and how client communicates with server transferring data back and forth.